Proceedings of

CICLOPS 2008

8th International Colloquium on Implementation of Constraint and LOgic Programming Systems

Udine, 12th-13th December, 2008

Organizers:

Manuel Carro Liñares

UPM

Bart Demoen

KUL

Preface

As previous editions of CICLOPS, the 2008 version in Udine brings together researchers interested in the sequential and parallel implementation of logic and constraint programming languages and systems. CICLOPS promotes the free exchange of ideas and early dissemination of potentially premature and promising ideas. CICLOPS 2008 continues a tradition of successful workshops on Implementations of Logic Programming Systems, previously held with considerable success in Budapest (1993) and Ithaca (1994), the Compulog Net workshops on Parallelism and Implementation, and CICLOPS-es in Paphos (2001), Copenhagen (2002), (2003), St.-Malo (2004), Sitges (2005), Seattle (2006), and Porto (2007). With thirteen papers, the program is full, varied and interesting, and it shows both the need for this workshop and the potential for the future: you should feel very sorry if you cannot attend the workshop. The program committee did a great job in reviewing the papers and they have already given valuable feedback to the authors. Together with the interaction during the workshop, which is by tradition informal, lively and open, this CICLOPS again truly serves as a credible stepping stone to formal publication. We thank all authors, reviewers, and attendees for their efforts and interest.

Manuel Carro
Bart Demoen
Madrid and Leuven, November 2008

Program Committee

Slim Abdennadher (German University in Cairo)
Roberto Bagnara (University of Parma)
Amadeo Casas (Microsoft Co.)
Henning Christiansen (Roskilde University)
Gregory Duck (NICTA)
Hai-Feng Guo (University of Nebraska at Omaha)
Remy Haemmerle (Technical University of Madrid)
José Morales (Complutense University of Madrid)
Ulrich Neumerkel (Technische Universität Wien)
Phuong-Lan Nguyen (IMA, Université Catholique de l'Ouest)
Ricardo Rocha (University of Porto)
Tom Schrijvers (K.U. Leuven)
Paul Tarau (University of North Texas)
Jan Wielemaker (Universiteit van Amsterdam)
Manuel Carro (Technical University of Madrid — Co-chair)
Bart Demoen (K.U. Leuven — Co-chair)

Table of Contents

Implementing Thread Cancellation in Multithreaded Prolog Systems

Atef Suleiman and John Miller

School of Electrical Engineering and Computer Science
Washington State University (Tri-Cities)
West 201B, Richland, WA 99354-2125, USA
{asuleiman,jhmiller}@tricity.wsu.edu

Abstract. The Prolog primitive `thread_cancel/1`, which simply cancels a thread as recommended in ISO/IEC DTR 13211-5:2007, is conspicuously absent in well-maintained, widely used multithreaded Prolog systems. The ability to cancel a thread is useful for application development and is critical to Prolog embeddability. The difficulty of cancelling a thread is due to the instant mapping of Prolog multithreading primitives to the native-machine thread methods. This paper reports on an attempt to implement thread cancellation using self-blocking threads. A thread blocks at the same safe execution point where the state of the underlying virtual machine is defined. A blocked thread awaits a notification to resume or terminate. A resumed thread may be redirected to self-block by a blocking primitive. Experimental results based on a prototype implementation show that using self-blocking threads not only simplifies the implementation of thread cancellation but also improves the performance of message-passing primitives.

Key words: Prolog, concurrency, threads

1 Introduction

Explicit expressions of concurrency advance Prolog's standing as a practical programming language capable of exploiting modern multiprocessor computers. Prolog programs consist largely of static code, knowledge expressed as facts and rules, accessible to any number of execution threads running concurrently, in parallel or otherwise. Additionally, due to their declarative and high-level nature, Prolog programs retain and expose opportunities for parallel execution unparalleled in conventional programming languages. To facilitate expressions of concurrency, a thread model is proposed in ISO/IEC DTR 13211-5:2007 [1], variants of which are implemented in well-maintained, widely used Prolog systems, such as Ciao [2], SWI-Prolog [3], XSB [4], Yap [5] and others. The model includes a set of low-level primitives for thread creation, synchronization and communication. In addition to sharing the static database on a read-only basis, Prolog threads may modify and share the dynamic database in a mutually exclusive manner. Recent research in definition and implementation of high-level

parallelism primitives shows that a relevant speedup is obtainable by exploiting parallelism expressly at the source-language level [6, 7]. As this research activity illustrates, there are situations in which a need to cancel a thread arises after the thread has already started.

The need for cancelling a thread is illustrated by the high-level primitive `threaded/1` defined in [6]. Given a conjunction of well-formed goals, this primitive simulates an and-parallel operator executing the goals concurrently using a dedicated thread for each goal. The primitive succeeds if all goals succeed; otherwise, if a goal fails or raises an exception, it fails. Hence, once a thread at some point returns a failure result for a goal it has executed, the remaining threads should be cancelled since they serve no purpose at that point. A similar need for cancelling a thread arises when a threaded goal executes successfully as part of a deterministic disjunction executing concurrently. These and other practical examples, such as an asynchronously generated cancel condition initiated by a user request to exit a running program, show that thread cancellability is a desirable method of Prolog threads.

The option of cancelling a Prolog thread is provided by the primitive `thread_cancel/1`, described in [1] as follows:

> *`thread_cancel/1` cancels a thread. Any mutexes held by the thread shall be automatically released. The main Prolog thread cannot be cancelled. Other than this, any thread can cancel any other thread. It is expected that all the resources consumed by the thread be released upon thread cancellation.*

Prolog systems, however, implement `thread_cancel/1` in a variable way. XSB shares the responsibility for cancelling a thread with the programmer, whereas SWI-Prolog defers the implementation of `thread_cancel/1` altogether to the programmer, with the insight that the primitive is best implemented depending on the thread model of the problem at hand. In Ciao, the outcome of cancelling a thread is partly defined and depends wholly on the state of the target thread. The implementation of `thread_cancel/1` in these and other otherwise-compliant Prolog systems suggests that the above description for `thread_cancel/1` may be easier said than done.

The difficulty of cancelling a thread is due to blocking functions. Standard library functions, such as `read`, `accept`, `wait`, are subject to blocking as they are dependent on external events, e.g., the availability of input, establishment of a network connection, occurrence of a specified event. A thread attempting to cancel a blocked thread must be able to interact with the function inside which the target thread is blocked. The interaction may be initiated by either the cancelling thread, by means of signalling, or the cancelled thread, by means of polling. The latter method is adapted by POSIX threads [8], on which the majority of Prolog systems is based.

Also referred to as *Pthreads*, POSIX threads is a set of C functions for managing threads, mutual exclusion and condition variables.[1] A Prolog thread is directly mapped to a POSIX thread, running a Prolog engine within a multi-engined Prolog virtual machine. Cancelling a Prolog thread in the context of POSIX threads is a well-defined process insofar as the semantics of the latter is concerned.

POSIX specifies a subset of blocking functions as *cancellation points*. A blocking function designated as cancellation-point is expected to call an internal or external Pthreads function, e.g., `pthread_testcancel`, at sufficient intervals and be prepared for the possibility that the function may not return due to the thread being cancelled. Consequently, any function calling a cancellation-point function must be equally prepared to give up control without further notice. A function prepares for the possible loss of control by registering thread-specific cleanup functions to be executed in the event of thread cancellation. The process of cancelling a Prolog thread may, thus, proceed as follows. Given a proper accounting of consumed resources using `pthread_cleanup_push` and `pthread_cleanup_pop` within every lexical scope containing a cancellation point, a thread cancels another thread asynchronously by calling `pthread_cancel`, which flags the target thread as cancelled and returns immediately. If the target thread is active, the Prolog engine traps the thread at a safe execution point and destroys it by exiting the thread startup function. Otherwise, if the target thread is blocked or is to block, Pthreads takes over control at the next cancellation point and begins the actual cancellation process by calling the thread cleanup functions in a last-in-first-out order. Apart from excluding certain blocking functions, most notably `pthread_mutex_lock`, from the standard list of cancellation points, the process of cancelling a POSIX thread seems transparent enough to support an orderly cancellation of the adjoining Prolog thread.

However, as evident by the lack of support for `thread_cancel/1` in well-maintained Prolog systems, the direct mapping approach to thread cancellation faces implementation issues related, in part, to Prolog signals and garbage collection. As recommended in [1], a Prolog thread should be able to signal another thread to execute a goal as a soft interrupt at safe points, including, for example, the point at which a Prolog thread is suspended waiting for a message from a message queue. At that point, neither POSIX signals nor Pthreads cancellation points provide a mechanism for processing Prolog signals. While a Prolog thread can process a POSIX signal, thus receive a Prolog signal, it can not execute the signal, while the thread is blocked by a cancellation-point function. Similarly, memory and atom garbage collection algorithms require a high level of cooperation among Prolog threads incompatible with low-level mapping of Prolog threads to Pthreads. For example, when an active Prolog thread triggers atom garbage-collection, all other threads must suspend and produce their list of atoms. Here, again, a Prolog thread blocked by a cancellation-point function can not be guaranteed to heed a garbage-collection interrupt in any specifiable

[1] Condition variables are synchronization objects that allow threads to wait for certain events (conditions) to occur.

manner. In order to effect a working level of cooperation among Prolog threads, a high-level mapping of Prolog threads to Pthreads is required.

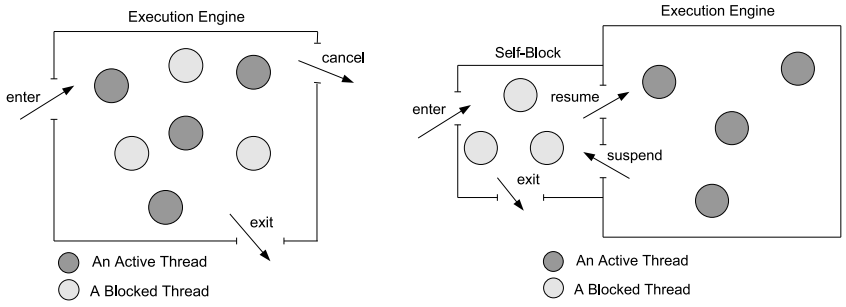
This paper reports on an attempt to implement thread cancellation using self-blocking threads. A self-blocking thread blocks at the same safe execution point where the state of the underlying Prolog engine is defined. A blocked thread awaits a notification to resume or terminate. A resumed thread may be reinstructed to self-block by a blocking primitive. Experimental results based on a proof-of-concept implementation show that using self-blocking threads is a viable approach for creating Prolog threads with the provision of facilitating their cancellation at any point during execution.

Section 2 presents the approach of self-blocking threads in the context of enabling synchronous cancellation of active and blocked threads. Section 3 includes implementation notes related to select blocking primitives. Section 4 presents the results of a performance comparison between self-blocking and directly-mapped threads. Section 5 briefly describes existing implementations of `thread_cancel/1`. Section 6 concludes with a summary of the cost-benefits of self-blocking threads.

2 An Execution Engine and Self-Block

Cancelling an active thread is a straightforward task. The thread is simply tagged as cancelled and the actual cancellation takes place upon the thread reaching a safe execution point. Cancelling a blocked thread, on the other hand, is a complex task requiring the consent and cooperation of the blocking function. Figure 1(a) shows a conceptual depiction of active and blocked threads inside the execution engine of a Prolog virtual machine. The difficulty of cancelling a thread lies with those threads that are blocked as a result of calling blocking functions. Figure 1(b) depicts the same threads in a new formation: active threads continue to be active; blocked threads are blocked on their own accord, using a self-blocking mechanism. The blocking functions are replaced by their cooperative counterparts, which are asynchronous, persistent and capable of instructing threads to block (suspend) or unblock (resume) as it may be warranted during execution. The task of cancelling a blocked thread is specifiable independent of the cancel method of the underlying native thread.

A blocking function directs a calling thread to self-block by returning a code indicating a *pending result*, based on which the thread self-blocks (suspends) waiting to be resumed or cancelled. A blocked thread is resumed by notifying the thread to continue execution from the point at which it was suspended, and is cancelled by notifying the thread to exit using the same control path used by an active thread exiting normally. A blocked thread may also be notified to perform atom-garbage collection or execute a goal as an interrupt. Multiple notifications are serialized using mutual exclusion. A notifying thread acquires exclusive control of the target thread prior to notification, with the caveat that control is granted only if the thread is suspended. A thread is suspended using



(a) Direct mapping (b) Self-blocking

Fig. 1. Graphical depiction of active and blocked threads.

the interrupt-vector mechanism commonly used in single-threaded systems to break into the top-level loop.

2.1 Implementing the Self-Block

The self-block is implemented using a standard synchronization composite of a *mutex*, *condition variable* and *counter*. Each thread is associated with a composite instance, which is initialized upon thread creation *in sync* with the creating thread. A blocking thread atomically unlocks the mutex and waits for the condition variable to be signalled by another thread. A signalling thread locks the mutex momentarily and signals the condition variable of the target thread. A blocked thread whose condition variable has been signalled re-locks its mutex, increments the counter and resumes execution. In addition to its standard role of preventing a race condition, the mutex is used to query the status of a thread. A thread queries the status of another thread by attempting to lock its mutex. If the attempt is successful, the thread is idle; otherwise, it is running. The counter is intended to be used in a test-yield loop to compel a signalled thread to assume ownership of its mutex.

The start-up algorithm for self-blocking threads is outlined in Figure 2. The algorithm takes a Prolog engine as input and proceeds as follows. First, it initializes a synchronization composite and swaps a reference to it with that of the temporary composite initialized by the creating thread for synchronizing with the current, newly created, thread (Lines 1-3). Second, it momentarily locks the mutex and signals the condition variable of the creating thread so that the latter may proceed (Line 4). Third, the algorithm iteratively suspends and resumes calling the execution engine for as often as the latter indicates a pending result (Lines 6-10). Lastly, the synchronization composite is destroyed and the native thread of control exits detaching from the adjoining Prolog engine.

```
    Input : A Prolog engine self
 1: initialize (composite = {mutex,condition,counter})
 2: lock (mutex)
 3: swap (self.composite, composite)
 4: lock (mutex), signal (condition), unlock (mutex)
 5: composite ← self.composite
 6: do
 7:     wait (condition, mutex)
 8:     counter ← counter + 1
 9:     execution_engine (self)
10: while self.result is a pending result
11: terminate (composite)
```

Fig. 2. Start-up algorithm of self-blocking threads.

2.2 Implementing `thread_cancel/1`

Cancelling a thread involves first suspending the thread, then destroying it. Since suspending and destroying a thread are well-defined tasks, they are implemented by the predicates `thread_suspend/1` and `thread_destroy/1`. With a negligible risk of raising an unintended exception, `thread_cancel/1` is defined as follows:

```
thread_cancel(Thread) :-
    thread_suspend(Thread),
    thread_destroy(Thread).
```

The algorithms for implementing *thread_suspend* and *thread_destroy* are listed in Figure 3 and 4, respectively. Both algorithms begin by decoding the target thread *thread* from the current actual arguments of the calling thread $self$. It is assumed that access to shared resources, such as the list of existing threads *list_of_threads*, is serialized using a locking mechanism.

The algorithm for *thread_suspend* starts by locking the list of existing threads and performing a series of tests, including whether the target thread is non-existent (Lines 3-6), referenced by other threads (Lines 7-10) or itself the calling thread (Lines 11-14), in which cases it throws an appropriate error-term or returns a pending result; otherwise, it increments the reference counter of the target thread and unlocks the list of existing threads (Lines 15-16). Next, the algorithm suspends the target thread by first setting its interrupt vector, then locking its mutex momentarily (Lines 17-22). Since it is possible that the target thread suspends for a reason other than having been interrupted, the thread interrupt vector is reset based on the return result. Lastly, the algorithm decrements the reference counter of the target thread and continues execution with the following instruction (Lines 23-26). Chances are that the next instruction to be executed corresponds to *thread_destroy*. In a like manner, *thread_destroy* algorithm destroys a target thread, provided the thread exists, is idle, different from the calling thread and unreferenced by any other threads.

```
1:  thread ← decode (self,1)
2:  lock (thread_resource)
3:  if thread ∉ list_of_threads then
4:      unlock (thread_resource)
5:      throw existence_error
6:  end
7:  if thread.reference > 0 then
8:      unlock (thread_resource)
9:      throw permission_error
10: end
11: if thread = self then
12:     unlock (thread_resource)
13:     thread.signal ← thread.signal ∨
        suspend_signal
14:     return signal_result
15: end
16: thread.reference ← thread.reference + 1
17: unlock (thread_resource)
18: thread.signal ← thread.signal ∨
    suspend_signal
19: lock (thread.mutex)
20: if thread.result ≠ signal_result then
21:     thread.signal ← thread.signal ∧
        ¬suspend_signal
22: end
23: unlock (thread.mutex)
24: lock (thread_resource)
25: thread.reference ← thread.reference − 1
26: unlock (thread_resource)
27: goto next_instruction
```

Fig. 3. *thread_suspend* algorithm

```
1:  thread ← decode (self,1)
2:  lock (thread_resource)
3:  if thread ∉ list_of_threads then
4:      unlock (thread_resource)
5:      throw existence_error
6:  end
7:  if thread = self ∨ thread.reference ≠ 0
    ∨ ¬ trylock(thread.mutex) then
8:      unlock (thread_resource)
9:      throw permission_error
10: end
11: destroy (thread)
12: unlock (thread_resource)
13: goto next_instruction
```

Fig. 4. *thread_destroy* algorithm

3 Implementing Thread-Blocking Predicates

Blocking predicates, be they built-in or user-defined, i.e., foreign, block by instructing the calling thread to self-block. For Prolog systems that provide a foreign-language interface, blocking foreign code communicates its blocking instructions by calling an appropriate interface function. The following are implementation notes related to select blocking predicates.

`get_code(+Stream, ?Code)` gets the character code of a single character from the (non-standard) input stream *Stream* and unifies it with the term *Code*. The predicate behaves like the standard built-in `get_code/2`, except that if the stream position of *Stream* is *end-of-stream* and *eof_action(suspend)* is a property of *Stream*, then the calling thread suspends, with the expectation that the foreign module that created *Stream* (e.g., an embedding application or shared library) will call an appropriate interface function to resume the calling thread when new characters become available.

`thread_get_message(+Queue, ?Term)` searches the message queue *Queue* for a term unifiable with the term *Term*. If a term is found, the term is unified with *Term* and deleted from *Queue*. Otherwise, if a term is not found, the calling thread is added to a waiting list associated with *Queue* and instructed to

block (suspend). The search, deletion and addition are performed in a mutually exclusive manner.

`thread_send_message(+Queue, @Term)` searches the waiting list of the message queue *Queue* for a thread whose receiving term is unifiable with the term *Term*. If a thread is found, then the thread is deleted from the waiting list, the receiving term is unified with *Term*, and the thread is instructed to unblock (resume). Otherwise, if a receiving thread is not found, *Term* is added to *Queue*. The search, deletion and addition are performed in a mutually exclusive manner.

`mutex_lock(+Mutex)` acquires the Prolog mutex *Mutex* blocking if necessary. If *Mutex* is already acquired by a thread other than the calling thread, then the calling thread is added to a waiting list associated with *Mutex* and instructed to suspend. If Mutex is previously acquired by the calling thread, then the recursion counter of *Mutex* is incremented. Otherwise, if *Mutex* is free, the calling thread acquires *Mutex*. The conditionals and corresponding actions are performed in a mutually exclusive manner.

`mutex_unlock(+Mutex)` releases the Prolog mutex *Mutex*. If *Mutex* is acquired by the calling thread and the recursion counter of *Mutex* is greater than zero, then the recursion counter is decremented. If *Mutex* is acquired by the calling thread and the recursion counter of *Mutex* is zero, then *Mutex* is first released, then acquired by the first thread, if any, on the waiting list of *Mutex* and the thread is instructed to resume. The conditionals and corresponding actions are performed in a mutually exclusive manner.

`sleep(+Interval)` suspends execution of the calling thread for the interval *Interval*. If *Interval* is an integer greater than zero, then the calling thread *Self* is suspended immediately and resumed after *Interval* is elapsed as follows. If an alarm is already set for a thread *Thread* and is expected to set off after interval $Interval_{thread}$ is elapsed, and $Interval > Interval_{thread}$, then the pair (*Self*, *Interval* - $Interval_{thread}$) is inserted into list *List*, containing ordered pairs of alarms to be set and threads to be resumed. Otherwise, if $Interval < Interval_{thread}$, then the alarm is cancelled, a new alarm is created to set off after *Interval* is elapsed, and the pair (*Thread*, $Interval_{thread}$ - *Interval*) is inserted into *List*. The insertion and cancellation are performed in a mutually exclusive manner. The alarm is a special thread that sleeps synchronously for and on behalf of the intervals and threads in *List*.

4 Performance Evaluation

A prototype Prolog implementation was developed to assess the performance of self-blocking threads on two popular operating systems: Linux and Windows. The prototype is a simple compiler and emulator comparable in performance to SWI-Prolog [3]. A select number of multithreading primitives were implemented using the self-blocking method, as described in Section 3, and the direct mapping method, as implemented in SWI-Prolog. The method in effect is determined at build time using conditional compilation. Three performance parameters were

measured: thread-creation time, message-passing time and synchronization time. The latter parameters were also measured using SWI-Prolog. All measurements were obtained by averaging ten runs per input per program. The computing environment is comprised of a single computer, equipped with Intel Core 2 Quad processor (2.5GHz), 3GB RAM (800MHz), dual-bootable with Linux Debian version 4.0 and Windows Vista (32-bit).

It should be noted that although both Linux and Windows use one-to-one mapping between user threads and kernel threads, Linux threads appear to be considerably more lightweight than Windows threads, possibly due in part to the Windowing system of Windows being an integral part of Windows kernel. The objective of this evaluation is to compare the performance of self-blocking threads to that of directly mapped threads. A thread performance comparison between Linux and Windows is outside the scope of this paper, let alone the interests of its authors.

4.1 Thread Creation

As described in Section 2.1, the procedure for creating a self-blocking thread requires that the calling thread blocks until the newly created thread initializes its self-blocking mechanism. The thread-creation time parameter is intended to quantify the overhead incurred by self-blocking threads during thread creation.

The execution time of thread creation of self-blocking and directly mapped threads was measured directly using two simple programs written in C. The first program measures the execution time of thread creation of directly mapped threads. It trivially creates a variable number of threads by calling the function `pthread_create`, tracking the wall time elapsed using the function `clock`. The second program measures the execution time of thread creation of self-blocking threads. It has the structure of the first program except that the call to `pthread_create` is embedded in a new function responsible for synchronizing the calling thread with the thread to be created. The new function initializes a temporary synchronization composite comprised of a mutex and condition variable, and calls `pthread_create`, passing a reference to the composite. It then calls `pthread_cond_wait` and blocks waiting for the composite to be signalled by the newly created thread. Meanwhile, the new thread first initializes its self-blocking mechanism, then signals the composite of the calling thread so that the latter may proceed.

As shown in Table 1, self-blocking threads are more expensive to create than directly mapped threads. The average execution time of thread creation of a self-blocking thread is about twice that of a directly mapped thread on both Linux and Windows. On Linux, the execution time of thread creation increases as the number of threads increases, approaching a measurable value when the number of threads equals or exceeds $1,000$. On Windows, the execution time of

thread creation is stable, around 200 μs per self-blocking thread and 100 μs per directly mapped thread, regardless of the number of threads.[2]

Table 1. Comparison of average execution time of thread creation (μs *per thread*).

# of threads	Linux		Windows	
	Direct mapping	Self-blocking	Direct mapping	Self-blocking
100	0	0	107	205
200	0	0	106	207
500	0	0	106	206
1000	2	6	107	205
2000	7	12	106	204
4000	10	15	106	204

4.2 Message Passing

The message-passing time parameter was first measured for the case of a single sender/receiver, where neither implementation method has an apparent advantage over the other. Here, passing a message involves sending the message and waking up the receiving thread. The time measurements were obtained using the program described in [9]. The program involves passing a message between N threads M times. The threads are linked in a ring structure. The message is an integer specifying the number of times the message is to be passed. Upon receiving the integer-message, a thread decrements the integer and passes it to the next thread. The message passing between threads continues until the integer becomes less than zero, at which point a thread simply exits. The program is listed in Figure 5. The message-passing time measurements were estimated for select numbers of threads performing message passing $1,000,000$ times. The results are presented in Table 2.[3]

Overall, the performance of self-blocking threads and directly mapped threads are comparable on both Linux and Windows. On a closer examination, however, the self-blocking approach is consistently, albeit slightly, faster than the direct mapping approach as implemented in both the prototype and SWI-Prolog. The

[2] On Windows, according to spawn-time measurement results obtained from Prototype and SWI-Prolog, the execution time of POSIX thread creation is the dominant component of the execution time of Prolog thread creation.

[3] For assurance and sheer curiosity, the time measurements of Java threads were also obtained and presented. On Linux, Java threads perform simple message passing twice as fast as Prolog threads using either approach. The Java speedup is likely due to Prolog's need to validate, in a mutually exclusive manner, the existence of a thread prior to accessing its message queue. The question as to why Java threads were unable to maintain a similar speedup factor on Windows is outstanding.

```
start(N, M) :-
    N1 is N - 1,
    thread_self(Thread),
    setup(N1, Thread, NextThread),
    thread_send_message(NextThread, M),
    catch(process(NextThread), _, true).

process(Thread) :-
    repeat,
        thread_get_message(M),
        M1 is M - 1,
        thread_send_message(Thread, M1),
        M1 < 0,
    !.

setup(0, Thread, Thread) :- !.
setup(N, Thread, NextThread) :-
    Goal = process(Thread),
    thread_create(Goal, NewThread, [detached(true)]),
    N1 is N - 1,
    setup(N1, NewThread, NextThread).
```

Fig. 5. Program for measuring simple message-passing time.

number of threads that can be created in SWI-Prolog is limited to less than 100 threads, thus the time measurements corresponding to numbers of threads equal or exceeding 100 are unobtainable. The simple message-passing time is relatively stable, around 4 μs on Linux, 12 μs on Windows, per message, for a range of 10 to 400 threads. However, this parameter is likely to increase as the number of threads increases due in part to cache exhaustion due, in turn, to the uncommon memory requirements of Prolog threads.

Table 2. Comparison of average execution time of threads performing simple message-passing (μs *per message*).

(a) Average execution time on Linux

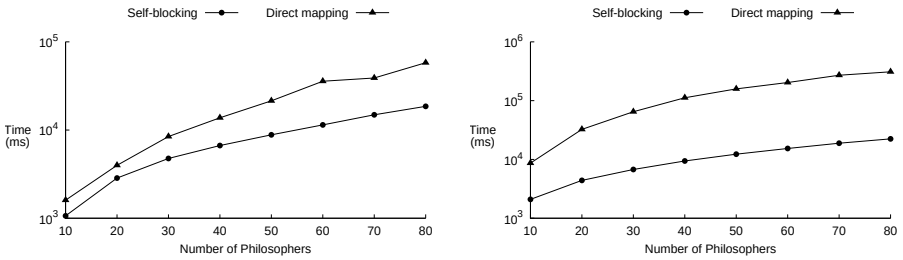
# of threads	self-blocking	direct mapping	SWI-Prolog 5.6.61	Java 1.6.0_06
10	5.86	5.90	5.99	3.03
20	4.36	5.26	4.73	2.94
40	4.26	4.78	4.58	2.91
80	4.02	4.53	4.94	3.21
100	4.15	4.38	–	3.24
200	4.12	4.38	–	3.35

(b) Average execution time on Windows

# of threads	self-blocking	direct mapping	SWI-Prolog 5.6.61	Java 1.6.0_06
10	11.75	13.21	14.54	11.75
20	11.95	12.20	13.71	11.75
40	11.95	12.73	13.29	11.22
80	11.95	12.48	13.38	11.26
100	12.04	12.83	–	11.39
200	12.78	13.51	–	11.39

The message-passing time parameter was, second, measured for the case of multiple senders/receivers, where self-blocking threads have a decisive advantage over directly-mapped threads. Here, message passing may involve a series of time-consuming operations, including adding (copying) a sender's message to a message queue, searching a list of waiting receivers for one whose skeletal message matches a newly added message, searching a message queue for a message matching a receiver's skeletal message, waking up potential receivers or just a matching receiver, and adding a new receiver to a list of waiting receivers.

The classic concurrency problem of the dining philosophers was used to illustrate the speed advantage of self-blocking threads in programs that require extensive message passing. The solution found in [10] was adapted to obtain wall time measurements for a variable number of philosophers. The measurements are depicted graphically in Figure 6.



(a) Linux (b) Windows

Fig. 6. The Dining Philosophers benchmark (10,000 eat-think cycle per philosopher.)

As expected, self-blocking threads outperform directly mapped threads, by a factor of 2 on Linux and by an order of magnitude on Windows. The source of the speedup is transparent. In the self-blocking approach, a new sender signals at most one potential receiver, whereas in the direct-mapping approach, the sender must signal all waiting receivers, even though only one of which might succeed in getting the sender's message while the other receivers will attempt in vain to unify their skeletal messages with the old messages of previous senders. In addition to performing needless unification, the majority of receivers effects needless task-switches performed by the operating system at the behest of unassuming senders.

4.3 Synchronization

The synchronization time parameter was measured using a simple program, which creates a variable number of threads, each of which updates a shared resource 10,000 times. Mutual exclusion is achieved using a global mutex and the synchronization primitives `mutex_lock/1` and `mutex_unlock/1`. The average

execution time per mutual exclusion was estimated by subtracting the wall time required to execute an equal number of updates sequentially. The results are presented in Table 3.

Table 3. Comparison of average execution time of threads updating a shared resource (*μs per mutual exclusion*).

(a) Average execution time on Linux

# of threads	self-blocking	direct mapping (compliant)	direct mapping (incompliant)	SWI-Prolog 5.6.61
10	7.53	7.97	0.56	0.86
20	7.90	8.01	0.75	0.95
40	7.47	8.52	0.91	1.02
80	7.53	8.70	0.96	1.08
100	6.88	8.78	0.99	–
200	7.89	8.93	1.01	–

(b) Average execution time on Windows

# of threads	self-blocking	direct mapping (compliant)	direct mapping (incompliant)	SWI-Prolog 5.6.61
10	11.06	16.10	1.53	11.31
20	10.90	17.04	1.49	11.91
40	10.46	17.37	1.51	12.52
80	10.89	17.39	1.49	12.49
100	10.57	17.52	1.49	–
200	10.75	18.18	1.48	–

The performance of self-blocking and directly mapped threads in programs that require extensive synchronization varies depending on the implementation of Prolog mutex. For implementations potentially compliant with [1], self-blocking threads compare favorably to directly mapped threads on Linux. On Windows, the former (self-blocking) threads outperform the latter threads by a factor as high as 1.7. Moreover, on Windows, the prototype's compliant implementation using self-blocking threads outperforms SWI-Prolog incompliant implementation using directly mapped threads. The criteria for compliance, for the purpose of this comparison, is that a Prolog mutex is indestructible while it is in use, e.g., one or more threads are blocked attempting to acquire the mutex. As shown in Table 3, lifting this requirement of indestructibility can result in a synchronization speed characteristic of low-level programming languages, however, to the negation of the premise of using self-blocking threads, which is to provide a safe and user-friendly Prolog multithreaded environment.

5 Related work

While Prolog systems agree on how to create threads, they differ widely on how to cancel them.

SWI-Prolog [3] and Yap [5] defer the implementation of `thread_cancel/1` to the programmer with the insight that thread cancellation is best implemented based on the thread model of the application at hand. In the boss/worker thread model, for example, `thread_cancel/1` may be implemented by communicating to the thread to be cancelled a specially coded message instructing the thread to exit or abort. In a computation-intensive application, for another example, cancelling a thread may be achieved by signalling the thread to execute a goal quoting a control primitive, such as `thread_exit(cancelled)`.

In XSB [4], thread cancelation is a joint responsibility of the system and the application. The latter initiates the process of canceling a thread by calling `thread_cancel/1`, giving the thread to be cancelled as an argument. For its part, XSB internally flags the given thread as canceled and waits for the thread execution to reach a call or execute port, at which point XSB throws a cancelation error ending its role in the thread cancelation process. The target thread, henceforth, is expected to catch the error, release any allocated resources and exit voluntarily.

Ciao [2] provides a primitive named `eng_kill/1`, which attempts to cancel the thread associated with a given goal identifier. The attempt may succeed, fail, block or render the system in an unstable state, depending on whether, irrespectively, the thread to be cancelled is trappable at a standard port, the goal identifier is valid, the thread is blocked by a system call, or other noted, however unspecified, situations.

Other Prolog systems, such as BinProlog and Qu-Prolog, provide other variations on the theme of thread cancellation. However, the primitives tasked with cancelling a thread are summarily documented. Attempts to learn of the internals of these primitives, through haphazard queries written with ill intents, showed that thread cancellation in these systems is problematic.

6 Conclusion

This paper presented an experimental implementation approach for creating Prolog threads with the provision of facilitating their destruction at any point during execution. The approach is based on self-blocking threads, a common implementation technique for managing thread interactions in multithreaded applications. The ability to cancel a thread safely and synchronously improves Prolog's standing as a useful programming language, capable of expressing variable solutions to complex concurrent problems for prototyping or production purposes. Additionally, it preserves the integrity of Prolog's traditional top-level loop program and improves Prolog's embeddability into multi-paradigm, multi-language applications.

Thread cancellability with self-blocking threads increases the complexity of system and extension development, as might be expected of features of high-level programming languages. Standard library functions, such as `seek`, `sleep`, `select`, may not be used directly to implement built-in and library predicates. Instead, these functions are reemployed within newly designed, more complex functions which are reentrant, persistent, asynchronous and able to communicate intermediate results. This added complexity may be viewed as a fair price, paid at the right layer in the right currency, C, in exchange for preserving Prolog's dictum of combining simplicity and power at the user level.

Although native in their own right, self-blocking threads exhibit the programmability of green threads,[4] as they are at most one standard port away from relinquishing processor control and one wake-up call from regaining it. As such, they are fit to yield the main benefits of both native and green multithreaded environments, namely parallelism and portability. Used in this capacity, the self-blocking approach constitutes a cost-efficient compromise between using native preemptive threads [11] and nonnative cooperative threads [12].

The performance of self-blocking threads compares favorably to that of directly mapped threads, despite that the time cost of creating a self-blocking thread is twice that of a directly mapped thread, due to the initial cost of the former's self-blocking mechanism. Self-blocking threads support a wide range of algorithms for implementing message passing, a primary means of thread communication and synchronization [1]. For programs that require extensive message passing, experimental results showed that execution times vary by up to an order of magnitude, depending on the operating system and the algorithm used for matching the messages of senders and receivers. Given that directly mapped threads can hardly do without a message queue and message passing, the run time advantage of self-blocking threads should offset the initial cost of their self-blocking mechanisms.

The utility of self-blocking threads extends beyond simplifying thread cancellation to enabling the implementation of high-level features, such as the separation of thread creation and execution, the implementation of suspend and resume primitives, backtracking, multiple executions and execution modes. The ability to separate thread creation from execution, proposed in passing in [13], facilitates the implementation of a high level API, which subsumes the one recommended in [1], which in turn facilitates the implementation of yet higher-level parallel operators analogous to those introduced in [6] and [7]. Experiments are being conducted to evaluate the merits of new multithreading primitives in terms of simplicity and expressiveness, as well as performance.

Acknowledgments. The authors thank Jan Wielemaker and Richard O'Keefe for their insightful, differing views. This research was supported by the Office of Science (BER), U.S. Department of Energy, Grant No. DE-FG02-05ER64105.

[4] Green threads are threads that are scheduled by a virtual machine instead of natively by the underlying operating system.

References

1. Wielemaker, J., Moura, P., Nunes, P., Robinson, P., Marques, R., Swift, T.: ISO/IEC DTR 13211-5:2007 Prolog Multi-threading Predicates. (2007)
2. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., Lopez-Garcıa, P., Puebla, G.: The Ciao Prolog System. Reference Manual (v1. 8). The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of Madrid (UPM), May (2002)
3. Wielemaker, J.: SWI Prolog 5.6 Reference Manual. Department of Social Science Informatics, University of Amsterdam, Amsterdam, Marz (2006)
4. Sagonas, K., Swift, T., Warren, D., Freire, J., Rao, P.: XSB Prolog. The XSB System Version 3.1 Volume 1: Programmers Manual (2007)
5. Santos-Costa, V., Damas, L., Reis, R., Azevedo, R.: The Yap Prolog Users Manual. Universidade do Porto and COPPE Sistemas (2006)
6. Moura, P., Crocker, P., Nunes, P.: High-Level Multi-threading Programming in Logtalk. In Hudak, P., Warren, D.S., eds.: PADL. Volume 4902 of Lecture Notes in Computer Science., Springer (2008) 265–281
7. Casas, A., Carro, M., Hermenegildo, M.: Towards a high-level implementation of flexible parallelism primitives for symbolic languages. Proceedings of the 2007 international workshop on Parallel symbolic computation (2007) 93–94
8. The IEEE and The Open Group: 1003.1 Standard for Information Technology-Portable Operating System Interface (Posix) System Interfaces, Issue 6. IEEE Std 1003.1-2001. System Interfaces, Issue 6 (2001)
9. Halen, J., Karlsson, R., Nilsson, M.: Performance measurements of threads in Java and processes in Erlang. Webpage, Last visit January (2006)
10. de Bosschere, K., Tarau, P.: Blackboard-based extensions in Prolog. Software—Practice & Experience **26** (1996) 49–69
11. Wielemaker, J.: Native preemptive threads in SWI-prolog. In Palamidessi, C., ed.: ICLP. Volume 2916 of Lecture Notes in Computer Science., Springer (2003) 331–345
12. Eskilson, J., Carlsson, M., Palamidessi, C., Glaser, H., Meinke, K.: SICStus MT—A Multithreaded Execution Environment for SICStus Prolog. Programming Languages: Implementations, Logics, and Programs (**1490**) 36–53
13. Carro, M., Hermenegildo, M.: Concurrency in Prolog Using Threads and a Shared Database. Logic Programming: Proceedings of the 1999 International Conference on Logic Programming (1999)

Interactors: Logic Engine Interoperation with Pure Prolog Semantics

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
E-mail: tarau@cs.unt.edu

Abstract. We introduce a new programming language construct, *Interactors*, supporting the agent-oriented view that programming is a dialog between simple, self-contained, autonomous building blocks.
We define *Interactors* as an abstraction of answer generation and refinement in *Logic Engines* resulting in expressive language extension and metaprogramming patterns.
As a first step toward a declarative semantics, we sketch a pure Prolog specification showing that Interactors can be expressed at source level, in a relatively simple and natural way.
Interactors extend language constructs like Ruby, Python and `C#`'s multiple coroutining block returns through *yield* statements and they can emulate the action of fold operations and monadic constructs in functional languages.
Using the Interactor API, we describe at source level, language extensions like dynamic databases and algorithms involving combinatorial generation and infinite answer streams.
Keywords*: Prolog language extensions, logic engines, semantics of metaprogramming constructs, generalized iterators, agent oriented programming language constructs*

1 Introduction

Interruptible Iterators are a new Java extension described in [1]. The underlying construct is the `yield` statement providing multiple returns and resumption of iterative blocks, i.e. for instance, a `yield` statement in the body of a `for` loop will return a result for each value of the loop's index.

The `yield` statement has been integrated in newer Object Oriented languages like `Ruby` [2, 3] `C#` [4] and `Python` [5] but it goes back to the *Coroutine Iterators* introduced in older languages like CLU [6] and ICON [7].

A natural generalization of Iterators, is the more radical idea of allowing clients to communicate to/from inside blocks of arbitrary recursive computations. The challenge is to achieve this without the fairly complex interrupt based communication protocol between the iterator and its client described in [1]. This suggests some form of structured two-way communication between a client and the usually autonomous service the client requires from a given language construct, often encapsulating an independent component.

Agent programming constructs have influenced design patterns at "macro level", ranging from interactive Web services to mixed initiative computer human interaction. *Performatives* in Agent communication languages [8] have made these constructs reflect explicitly the intentionality, as well as the negotiation process involved in agent interactions. At a more theoretical level, it has been argued that *interactivity*, seen as fundamental computational paradigm, can actually expand computational expressiveness and provide new models of computation [9].

In a logic programming context, the Jinni agent programming language [10] and the BinProlog system [11] have been centered around logic engine constructs providing an API that supported reentrant instances of the language processor. This has naturally led to a view of logic engines as instances of a generalized family of iterators called *Fluents* [12], that have allowed the separation of the first-order language interpreters from the multi-threading mechanism, while providing a very concise source-level reconstruction of Prolog's built-ins.

Building upon the *Fluents* API described in [12], this paper will focus on bringing interaction-centered, agent oriented constructs from software design frameworks and design patterns to programming language level.

The resulting language constructs, that we shall call *Interactors*, will express control, metaprogramming and interoperation with stateful objects and external services. They complement pure Horn Clause Prolog with a significant boost in expressiveness, to the point where they allow emulating at source level virtually all Prolog builtins, including dynamic database operations.

2 First Class Logic Engines

Our *Interactor API* is a natural extension of the *Logic Engine API* introduced in [12]. An *Engine* is simply a language processor reflected through an API that allows its computations to be controlled interactively from another *Engine* very much the same way a programmer controls Prolog's interactive toplevel loop: launch a new goal, ask for a new answer, interpret it, react to it.

A *Logic Engine* is an *Engine* running a Horn Clause Interpreter with LD-resolution [13] on a given clause database, together with a set of built-in operations. The command

```
new_engine(AnswerPattern,Goal,Interactor)
```

creates a new Horn Clause solver, uniquely identified by `Interactor`, which shares code with the currently running program and is initialized with `Goal` as a starting point. `AnswerPattern` is a term, usually a list of variables occurring in `Goal`, of which answers returned by the engine will be instances. Note however that `new_engine/3` acts like a typical constructor, no computations are performed at this point, except for allocating data areas. In our actual implementation, with all data areas dynamic, engines are lightweight and engine creation is extremely fast.

The `get/2` operation is used to retrieve successive answers generated by an Interactor, on demand. It is also responsible for actually triggering computations in the engine.

```
get(Interactor,AnswerInstance)
```

It tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`. If an answer is found, it is returned as `the(AnswerInstance)`, otherwise the atom `no` is returned. As in the case of the `Maybe` Monad in Haskell, returning distinct functors in the case of success and failure, allows further case analysis in a pure Horn Clause style, without needing Prolog's CUT or if-then-else operation.

Note that bindings are not propagated to the original `Goal` or `AnswerPattern` when `get/2` retrieves an answer, i.e. `AnswerInstance` is obtained by first standardizing apart (renaming) the variables in `Goal` and `AnswerPattern`, and then backtracking over its alternative answers in a separate Prolog interpreter. Therefore, backtracking in the caller interpreter does not interfere with the new Interactor's iteration over answers. Backtracking over the Interactor's creation point, as such, makes it unreachable and therefore subject to garbage collection.

An Interactor is stopped with the `stop/1` operation that might or might not reclaim resources held by the engine. In our actual implementation we are using a fully automated memory management mechanism where unreachable engines are automatically garbage collected.
So far, these operations provide a minimal *Coroutine Iterator API*, powerful enough to switch tasks cooperatively between an engine and its client and emulate key Prolog built-ins like `if-then-else` and `findall` [12], as well as higher order operations like *fold* and *best_of*.

3 From Fluents to Interactors

We will now describe the extension of the *Fluents* API of [12] that provides a minimal bidirectional communication API between interactors and their clients.

The following operations provide a "mixed-initiative" interaction mechanism, allowing more general data exchanges between an engine and its client.

3.1 A yield/return operation

First, like the `yield return` construct of `C#` and the `yield operation` of Ruby and Python, our `return/1` operation

```
return(Term)
```

will save the state of the engine and transfer *control* and a *result* `Term` to its client. The client will receive a copy of `Term` simply by using its `get/2` operation. Similarly to Ruby's `yield`, our `return` operation suspends and returns data from arbitrary computations (possibly involving recursion) rather than from specific language constructs like a `while` or `for` loop.

Note that an Interactor returns control to its client either by calling `return/1` or when a computed answer becomes available. By using a sequence of `return/get` operations, an engine can provide a stream of *intermediate/final results* to its client, without having to backtrack. This mechanism is powerful enough to implement a complete exception handling mechanism (see [12]) simply by defining

```
throw(E):-return(exception(E)).
```

When combined with a `catch(Goal,Exception,OnException)`, on the client side, the client can decide, upon reading the exception with `get/2`, if it wants to handle it or to throw it to the next level.

3.2 Interactors and Coroutining

The operations described so far allow an engine to return answers from any point in its computation sequence. The next step is to enable an engine's client to *inject* new goals (executable data) to an arbitrary inner context of an engine. Two new primitives are needed:

```
to_engine(Engine,Data)
```

used to send a client's data to an Engine, and

```
from_engine(Data)
```

used by the engine to receive a client's Data.

A typical use case for the *Interactor API* looks as follows:

1. the *client* creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*, parameterized as follows:
   (a) the *client* passes some data and a new goal to the *engine* and issues a `get` operation that passes control to it
   (b) the *engine* starts a computation from its initial goal or the point where it has been suspended and runs (a copy of) the new goal received from its *client*
   (c) the *engine* returns (a copy of) the answer, then suspends and returns control to its *client*
3. the *client* interprets the answer and proceeds with its next computation step
4. the process is fully reentrant and the *client* may repeat it from an arbitrary point in its computation

Using a metacall mechanism like `call/1` (which can also be emulated in terms of engine operations [12]) or directly through a source level transformation [14], one can implement a close equivalent of Ruby's `yield` statement as follows:

```
ask_engine(Engine,(Answer:-Goal), Result):-
  to_engine(Engine,(Answer:-Goal)),
  get(Engine,Result).

engine_yield(Answer):-
  from_engine((Answer:-Goal)),
  call(Goal),
  return(Answer).
```

where `ask_engine` sends a goal (possibly built at runtime) to an engine, which in turn, executes it and returns a result with an `engine_yield` operation.

As the following example shows, this allows the client to use from outside the (infinite) recursive loop of an engine as a form of *updatable persistent state*.

```
sum_loop(S1):-engine_yield(S1⇒S2),sum_loop(S2).

inc_test(R1,R2):-
   new_engine(_,sum_loop(0),E),
   ask_engine(E,(S1⇒S2:-S2 is S1+2),R1),
   ask_engine(E,(S1⇒S2:-S2 is S1+5),R2).

?- inc_test(R1,R2).
R1=the(0 ⇒ 2),
R2=the(2 ⇒ 7)
```

Note also that after parameters (the increments 2 and 5) are passed to the engine, results dependent on its state (the sums so far 2 and 7) are received back. Moreover, note that an arbitrary goal is injected in the local context of the engine where it is executed, with access to the engine's *state variables* `S1` and `S2`. As engines have separate garbage collectors (or in simple cases as a result of tail recursion), their infinite loops run in constant space, provided that no unbounded size objects are created.

4 A (mostly) Pure Prolog Specification

At a first look, Interactors deviate from the usual Horn Clause semantics of pure Prolog programs. A legitimate question arises: are they not just another procedural extension, say, like assert/retract, setarg, global variables etc.?

We will show here that the semantic gap between pure Prolog and its extension with Interactors is much narrower than one would expect. The techniques that we will describe can be seen as an executable specification of Interactors within the well understood semantics of logic programs (SLDNF resolution).

Toward this end, we will sketch an emulation, in pure Prolog, of the key constructs involved in defining Interactors.

There are four distinct concepts to be emulated:

1. we need to eliminate backtracking to be able to access multiple answers at a time
2. we need to emulate `copy_term` as different search branches and multiple uses of a given clause require fresh instances of terms, with variables standardized apart
3. we need to emulate suspending and resuming an engine
4. engines should be able to receive and return Prolog terms

We will focus here on the first two, that are arguably less obvious, by providing actual implementations. After that, we will briefly discuss the feasibility of the last two.

4.1 Metainterpreting Backtracking

First, let's define a clause representation, that can be obtained easily with a source-to-source translator. Clauses in the database are represented with difference-list terms, structurally isomorphic to the binarization transformation described in [14]. The code of a classic Prolog naive reverse + permutation generator program becomes:

```
:-op(1150,xfx,<=).

clauses([
        [app([],A,A)|B]<=B,
        [app([C|D],E,[C|F])|G]<=[app(D,E,F)|G],

        [nrev([],[])|H]<=H,
        [nrev([I|J],K)|L]<=[nrev(J,M),app(M,[I],K)|L],

        [perm([],[])|N]<=N,
        [perm([O|P],Q)|R]<=[perm(P,S),ins(O,S,Q)|R],

        [ins(T,U,[T|U])|V]<=V,
        [ins(W,[X|Y],[X|Z])|XO]<=[ins(W,Y,Z)|XO]
]).
```

Note that we can assume that variables are local to each clause and therefore they have been standardized apart accordingly[1].

First, let's define the basic inference step (equivalent to an LD-resolution step, [13]) as a simple "arrow composition" operation:

```
compose(F1,F2,A<=C):-copy_term(F1,A<=B),copy_term(F2,B<=C).
```

We can now add a new "arrow" to a list of existing arrows, provided that the composition succeeds:

```
match_one(F1,F2,Fs,[NewF|Fs]):-compose(F1,F2,F3),!,NewF=F3.
match_one(_,_,Fs,Fs).
```

and lift this to have an arrow (seen as representing the current goal), select from a *list* of clauses the ones that match:

```
match_all([],_,Fs,Fs).
match_all([Clause|Cs],Arrow,Fs1,Fs3):-
  match_one(Arrow,Clause,Fs1,Fs2),
  match_all(Cs,Arrow,Fs2,Fs3).
```

We can add a stopping condition to mark the success of an LD-derivation as matching an arrow of the form `Answer<=[]`

```
derive_one(Answer<=[],_,Fs,Fs,As,[Answer|As]).
derive_one(Answer<=[G|Gs],Cs,Fs,NewFs,As,As):-
  match_all(Cs,Answer<=[G|Gs],Fs,NewFs).
```

[1] Allowing shared variables would bring a different, but nevertheless interesting semantics, with "inter-clausal variables" seen as write-once global variables.

With these building blocks in place, an LD-derivation of *all answer instances* of a query can be defined as:

```
all_instances(AnswerPattern,Goal,Clauses,Answers):-
  Gs=[AnswerPattern⇐[Goal]],
  derive_all(Gs,Clauses,[],Answers).
```

where `derive_all` lifts the derivation process to progressively solve all existing and newly generated goals:

```
derive_all([],_,As,As).
derive_all([Arrow|Fs],Cs,OldAs,NewAs):-
  derive_one(Arrow,Cs,Fs,NewFs,OldAs,As),
  derive_all(NewFs,Cs,As,NewAs).
```

Finally, we can integrate the clause database:

```
all_answers(X,G,R):-clauses(Cs),all_instances(X,G,Cs,R).
```

and try out a few goals:

```
?- all_answers(Xs+Ys,app(Xs,Ys,[1,2,3]),Rs).
Rs = [[]+[1, 2, 3], [1]+[2, 3], [1, 2]+[3], [1, 2, 3]+[]]

?- all_answers(P,perm([1,2,3],P),Ps).
Ps = [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

Note, that for non-ground queries, answers computed this way keep variable equalities as expected:

```
?- List=[A,B,B,A],all_answers(R,nrev(List,R),Rs).
List = [A, B, B, A],
Rs = [[_A, _B, _B, _A]]
```

Note that, except for relying on `copy_term` and a cut that can be replaced with a negation as failure, the metainterpreter is entirely written in pure Prolog.

4.2 Emulating `copy_term`

We can emulate the effect of `copy_term` in the previously described metainterpreter by observing that a logical variable can be "split" into two new ones and consequently a Prolog term can be recursively deconstructed and rebuilt as two fresh terms, identical to it up to uniform variable renamings.

```
fork_term('$v'(T1,T2), R1,R2):-R1=T1,R2=T2.
fork_term(T, T1,T2):-
  nonvar(T),functor(T,F,N),(F/N) \== ('$v'/2),
  functor(T1,F,N),functor(T2,F,N),
  fork_args(N,T,T1,T2).

fork_args(0,_,_,_).
fork_args(I,T,T1,T2):-I>0,
  I1 is I-1,arg(I,T,X),
```

```
  fork_term(X,A,B),
  arg(I,T1,A),arg(I,T2,B),
  fork_args(I1,T,T1,T2).
```

One can see that this produces indeed two fresh copies of the original term:

```
?- fork_term(f(A,B,g(B,A)),T1,T2).
A = '$v'(_A1, _A2),
B = '$v'(_B1, _B2),
T1 = f(_A1, _B1, g(_B1, _A1)),
T2 = f(_A2, _B2, g(_B2, _A2)).
```

Note that `functor` and `arg` can be seen as generic abbreviations for predicates describing the building/decomposition operations for each function symbol occurring in the program and `$v/2` can be assumed to be any function symbol not occurring in the program. Along the lines of [15] one can see that this functionality can be also expressed through a simple program transformation provided that `nonvar/1` can be expressed using negation as failure as

```
  nonvar(X):- not(X=0),not(X=1).
```

We will obtain a slightly different definition of composition, that would require replacing both the clause and the resolvent with one of the copies while using the other pair of copies for the arrow compositions.

```
compose(F1,F2, A<=C, NewF1,NewF2):-
   fork_term(F1,A<=B,NewF1),
   fork_term(F2,B<=C,NewF2).
```

One can now see that after propagating the extra arguments through the clauses of the metainterpreter described in subsection 4.1, together with the source level transformations we just mentioned, a metainterpreter that does not require `copy_term` can be derived.

4.3 Implementing suspend/resume and term/exchanges

The metainterpreter described in subsection 4.1 can be easily modified to return the current goal list when observing a `return(X)` instruction and then be resumed at will, by adding a clause similar to the one handling the case `Answer<=[]`. At this point, data exchange operations and `to_engine` and `from_engine` can be implemented through an extra argument added to the metainterpreter.

5 Interactors and Higher Order Constructs

As a first glimpse at the expressiveness of the Interactor API, we will implement, in the tradition of higher order functional programming, a *fold* operation [16] connecting results produced by independent branches of a backtracking Prolog engine:

```
efoldl(Engine,F,R1,R2):-
  get(Engine,X),
  efoldl_cont(X,Engine,F,R1,R2).

efoldl_cont(no,_Engine,_F,R,R).
efoldl_cont(the(X),Engine,F,R1,R2):-
  call(F,R1,X,R),
  efoldl(Engine,F,R,R2).
```

Classic functional programming idioms like *reverse as fold* are then implemented simply as:

```
reverse(Xs,Ys):-
  new_engine(X,member(X,Xs),E),
  efoldl(E,reverse_cons,[],Ys).

reverse_cons(Y,X,[X|Y]).
```

Note also the automatic *deforestation* effect [17] of this programming style - no intermediate list structures need to be built, if one wants to aggregate the values retrieved from an arbitrary generator engine with an operation like sum or product.

6 Emulating Dynamic Databases with Interactors

The gain in expressiveness coming directly from the view of logic engines as answer generators is significant. We refer to [12] for source level implementations of virtually all essential Prolog built-ins (exceptions included). The notable exception is Prolog's dynamic database, requiring the bidirectional communication provided by interactors.

The key idea for implementing dynamic database operations with Interactors is to use a logic engine's state in an infinite recursive loop, similar to the coinductive programming style advocated in [18], to emulate state changes in its client engine.

First, a simple difference-list based infinite server loop is built:

```
queue_server:-queue_server(Xs,Xs).

queue_server(Hs1,Ts1):-
  from_engine(Q),
    server_task(Q,Hs1,Ts1,Hs2,Ts2,A),
  return(A),
  queue_server(Hs2,Ts2).
```

Next we provide the queue operations, needed to maintain the state of the database.

```
server_task(add_element(X),Xs,[X|Ys],Xs,Ys,yes).
server_task(push_element(X),Xs,Ys,[X|Xs],Ys,yes).
server_task(queue,Xs,Ys,Xs,Ys,Xs-Ys).
```

```
server_task(delete_element(X),Xs,Ys,NewXs,Ys,YesNo):-
  server_task_delete(X,Xs,NewXs,YesNo).
```

Then we implement the auxiliary predicates supporting various queue operations:

```
server_task_remove(Xs,NewXs,YesNo):-
  nonvar(Xs),Xs=[X|NewXs],!,
  YesNo=yes(X).
server_task_remove(Xs,Xs,no).

server_task_delete(X,Xs,NewXs,YesNo):-
  select_nonvar(X,Xs,NewXs),!,
  YesNo=yes(X).
server_task_delete(_,Xs,Xs,no).

select_nonvar(X,XXs,Xs):-nonvar(XXs),XXs=[X|Xs].
select_nonvar(X,YXs,[Y|Ys]):-nonvar(YXs),YXs=[Y|Xs],
  select_nonvar(X,Xs,Ys).
```

Finally, we put it all together, as a dynamic database API:

We can create a new engine server providing Prolog database operations:

```
new_edb(Engine):-new_engine(done,queue_server,Engine).
```

We can add new clauses to the database

```
edb_assertz(Engine,Clause):-
  ask_engine(Engine,add_element(Clause),the(yes)).

edb_asserta(Engine,Clause):-
  ask_engine(Engine,push_element(Clause),the(yes)).
```

and we can return fresh instances of asserted clauses

```
edb_clause(Engine,Head,Body):-
  ask_engine(Engine,queue,the(Xs-[])),
  member((Head:-Body),Xs).
```

or remove them from the the database

```
edb_retract1(Engine,Head):-Clause=(Head:-_Body),
  ask_engine(Engine,
    delete_element(Clause),the(yes(Clause))).
```

Finally, the database can be discarded by discarding the engine that hosts it:

```
edb_delete(Engine):-stop(Engine).
```

The following example shows how the database generates the equivalent of `clause/2`, ready to be passed to a Prolog metainterpreter.

```
test_clause(Head,Body):-
  new_edb(Db),
    edb_assertz(Db,(a(2):-true)),
```

```
    edb_asserta(Db,(a(1):-true)),
    edb_assertz(Db,(b(X):-a(X))),
  edb_clause(Db,Head,Body).
```

As a side note, combining this emulation with the metainterpreter described in section 4, provides an executable specification of Prolog's dynamic database operations in pure Prolog, worth investigating in depth, as future work.

Externally implemented dynamic databases can also be made visible as Interactors and reflection of the interpreter's own handling of the Prolog database becomes possible. As an additional benefit, multiple databases can be provided. This simplifies adding module, object or agent layers at source level. By combining database and communication Interactors, software abstractions like mobile code and autonomous agents can be built as shown in [19]. Encapsulating external stateful objects like file systems or external database or Web service interfaces as Interactors can provide a uniform interfacing mechanism and reduce programmer learning curves in practical applications of Prolog.

Moreover, Prolog operations traditionally captive to predefined list based implementations (like DCGs) can be made generic and mapped to work directly on Interactors encapsulating file, URL and socket Readers.

7 Refining control: a backtracking if-then-else

Modern Prolog implementations (SWI, SICStus, BinProlog, Jinni) also provide a variant of `if-then-else` that either backtracks over multiple answers of its `then` branch or switches to the `else` branch if no answers in the `then` branch are found. With the same API, we can implement it at source level as follows:

```
if_any(Cond,Then,Else):-
  new_engine(Cond,Cond,Engine),
  get(Engine,Answer),
  select_then_or_else(Answer,Engine,Cond,Then,Else).

select_then_or_else(no,_,_,_,Else):-Else.
select_then_or_else(the(BoundCond),Engine,Cond,Then,_):-
  backtrack_over_then(BoundCond,Engine,Cond,Then).

backtrack_over_then(Cond,_,Cond,Then):-Then.
backtrack_over_then(_,Engine,Cond,Then):-
  get(Engine,the(NewBoundCond)),
  backtrack_over_then(NewBoundCond,Engine,Cond,Then).
```

8 Simplifying Algorithms: Interactors and Combinatorial Generation

Various combinatorial generation algorithms have elegant backtracking implementations. However, it is notoriously difficult (or inelegant, through the use of impure side effects) to compare answers generated by different OR-branches of Prolog's search tree.

8.1 Comparing Alternative Answers

Such optimization problems can easily be expressed as follows:

- running the generator in a separate logic engine
- collecting and comparing the answers in a client controlling the engine

The second step can actually be automated, provided that the comparison criterion is given as a predicate

```
compare_answers(First,Second,Best)
```

to be applied to the engine with an `efold` operation

```
best_of(Answer,Comparator,Generator):-
  new_engine(Answer,Generator,E),
  efoldl(E,
    compare_answers(Comparator),no,
  Best),
  Answer=Best.

compare_answers(Comparator,A1,A2,Best):-
  if((A1\==no,call(Comparator,A1,A2)),
    Best=A1,
    Best=A2
  ).

?-best_of(X,>,member(X,[2,1,4,3])).
X=4
```

8.2 Counting Answers without Accumulating

Problems as simple as counting the number of solutions of a combinatorial generation problem can become tricky in Prolog (unless one uses impure side effects) as one might run out of space by having to generate all solutions as a list, just to be able to count them. The following example shows how this can be achieved using an `efold` operation on an integer partition generator:

```
integer_partition_of(N,Ps):-
  positive_ints(N,Is),
  split_to_sum(N,Is,Ps).

split_to_sum(0,_,[]).
split_to_sum(N,[K|Ks],R):-N>0,sum_choice(N,K,Ks,R).

sum_choice(N,K,Ks,[K|R]):-
  NK is N-K,split_to_sum(NK,[K|Ks],R).
sum_choice(N,_,Ks,R):-split_to_sum(N,Ks,R).

positive_ints(1,[1]).
positive_ints(N,[N|Ns]):-N>1,N1 is N-1,
```

```
  positive_ints(N1,Ns).

% counts partitions by running
% the generator on an engine that returns
% 1 for each answer that is found
count_partitions(N,R):-
  new_engine(1,
    integer_partition_of(N,_),Engine),
  efoldl(Engine,+,0,R).
```

8.3 Encapsulating Infinite Computations Streams

An infinite stream of natural numbers is implemented as:

```
loop(N):-return(N),N1 is N+1,loop(N1).
```

The following example shows a simple space efficient generator for the infinite stream of prime numbers:

```
prime(P):-prime_engine(E),element_of(E,P).

prime_engine(E):-new_engine(_,new_prime(1),E).

new_prime(N):-N1 is N+1,
  if(test_prime(N1),true,return(N1)),
  new_prime(N1).

test_prime(N):-
  M is integer(sqrt(N)),between(2,M,D),N mod D =:=0
```

Note that the program has been wrapped, using the `element_of` predicate defined in [12], to provide one answer at a time through backtracking. Alternatively, a forward recursing client can use the `get(Engine)` operation to extract primes one at a time from the stream.

9 Applications of Interactors and Practical Language Extensions

Interactors and Multi-Threading As a key difference with typical multi-threaded Prolog implementations like Ciao-Prolog and SWI-Prolog [20, 21], our Interactor API is designed up front with a clear separation between *engines* and *threads* as we prefer to see them as orthogonal language constructs.

While one can build a self-contained lightweight multi-threading API solely by switching control among a number of cooperating engines, with the advent of multi-core CPUs as the norm rather than the exception, the need for *native* multi-threading constructs is justified on both performance and expressiveness grounds. Assuming a dynamic implementation of a logic engine's stacks, Interactors provide lightweight independent computation states that can be easily mapped to the underlying native threading API.

A minimal native Interactor based multi-threading API, has been implemented in the Jinni Prolog system [10] on top of a simple thread launching built-in

```
run_bg(Engine,ThreadHandle)
```

This runs a new Thread starting from the engine's `run()` predicate and returns a handle to the Thread object. To ensure that access to the Engine's state is safe and synchronized, we hide the engine handle and provide a simple producer/consumer data exchanger object, called a `Hub`. Some key components of the multi-threading API, partly designed to match Java's own threading API are:

- `bg(Goal)`: launches a new Prolog thread on its own engine starting with `Goal`.
- `hub_ms(Timeout,Hub)`: constructs a new `Hub` - a synchronization device on which `N` consumer threads can wait with `collect(Hub,Data)` (similar to a synchronized `from_engine` operation) for data produced by `M` producers providing data with `put(Hub,Data)` (similar to a synchronized `from_engine` operation.

Interactor Pools `Thread Pools` have been in use either at kernel level or user level in various operating system and language implementations to avoid costly allocation and deallocation of resources required by Threads. Likewise, for Interactor implementations that cannot avoid high creation/initialization costs, it makes sense to build *Interactor Pools*. An Interactor Pool is maintained by a dedicated Logic Engine that keeps track of the state of various Interactors and provides recently freed handles, when available, to `new_engine` requests.

Associative Interactors The message passing style interaction shown in the previous sections between engines and their clients, can be easily generalized to associative communication through a unification based blackboard interface [22]. Exploring this concept in depth promises more flexible interaction patterns, as out of order `ask_engine` and `engine_yield` operations would become possible, matched by association patterns.

10 Interactors Beyond Logic Programming Languages

We will now compare Interactors with similar constructs in other programming paradigms.

10.1 Interactors in Object Oriented Languages

Extending Interactors to mainstream Object Oriented languages is definitely of practical importance, given the gain in expressiveness. An elegant open source Prolog engine `Yield Prolog` has been recently implemented in terms of Python's

yield and `C#`'s *yield return* primitives [23]. Extending Yield Prolog to support our Interactor API only requires adding the communication operations `from_engine` and `to_engine`. In older languages like `Java`, `C++` or `Objective C` one needs to implement a more complex API, including a `yield return` emulation.

10.2 Interactors and similar constructs in Functional Languages

Interactors based on logic engines encapsulate future computations that can be unrolled on demand. This is similar to lazy evaluation mechanisms in languages like Haskell [24]. Interactors share with Monads [25, 26] the ability to sequentialize functional computations and encapsulate state information. With higher order functions, monadic computations can pass functions to inner blocks. On the other hand, our `ask_engine` / `engine_yield` mechanism, like Ruby's `yield`, is arguably more flexible, as it provides arbitrary switching of control (coroutining) between an Interactor and its client. The ability to define Prolog's `findall` construct as well as `fold` operations in terms of Interactors, is similar to definition of comprehensions [26] in terms of Monads.

11 Conclusion

We have shown that Logic Engines encapsulated as Interactors can be used to build on top of pure Prolog a practical Prolog system, including dynamic database operations, entirely at source level. We have also provided a sketch of an executable semantics for Logic Engine operations in pure Prolog. This shows that, in principle, their exact specification can be expressed declaratively.

In a broader sense, Interactors can be seen as a starting point for rethinking fundamental programming language constructs like Iterators and Coroutining in terms of language constructs inspired by *performatives* in agent oriented programming.

Beyond applications to logic-based language design, we hope that our language constructs will be reusable in the design and implementation of new functional and object oriented languages.

References

1. Liu, J., Kimball, A., Myers, A.C.: Interruptible iterators. In Morrisett, J.G., Jones, S.L.P., eds.: POPL, ACM (2006) 283–294
2. Matsumoto, Y.: The Ruby Programming Language. (June 2000)
3. Sasada, K.: YARV: yet another RubyVM: innovating the ruby interpreter. In Johnson, R., Gabriel, R.P., eds.: OOPSLA Companion, ACM (2005) 158–159
4. Microsoft Corp.: Visual `C#`. Project URL http://msdn.microsoft.com/vcsharp.
5. van Rossum, G.: A Tour of the Python Language. In: TOOLS (23), IEEE Computer Society (1997) 370
6. Liskov, B., Atkinson, R.R., Bloom, T., Moss, J.E.B., Schaffert, C., Scheifler, R., Snyder, A.: CLU Reference Manual. Volume 114 of Lecture Notes in Computer Science. Springer (1981)

7. Griswold, R.E., Hanson, D.R., Korb, J.T.: Generators in Icon. ACM Trans. Program. Lang. Syst. **3**(2) (1981) 144–161
8. Mayfield, J., Labrou, Y., Finin, T.W.: Evaluation of KQML as an Agent Communication Language. In Wooldridge, M., Müller, J.P., Tambe, M., eds.: ATAL. Volume 1037 of Lecture Notes in Computer Science., Springer (1995) 347–360
9. Wegner, P., Eberbach, E.: New Models of Computation. Comput. J. **47**(1) (2004) 4–9
10. Tarau, P.: Orthogonal Language Constructs for Agent Oriented Logic Programming. In Carro, M., Morales, J.F., eds.: Proceedings of CICLOPS 2004, Fourth Colloquium on Implementation of Constraint and Logic Programming Systems, Saint-Malo, France (September 2004)
11. Tarau, P.: BinProlog 11.x Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp. (2006)
12. Tarau, P.: Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In Lloyd, J., ed.: Computational Logic–CL 2000: First International Conference, London, UK (July 2000) LNCS 1861, Springer-Verlag.
13. Tarau, P., Boyer, M.: Nonstandard Answers of Elementary Logic Programs. In Jacquet, J., ed.: Constructing Logic Programs. J.Wiley (1993) 279–300
14. Tarau, P., Boyer, M.: Elementary Logic Programs. In Deransart, P., Maluszyński, J., eds.: Proceedings of Programming Language Implementation and Logic Programming. Number 456 in Lecture Notes in Computer Science, Springer (August 1990) 159–173
15. Warren, D.H.D.: Higher-order extensions to Prolog – are they needed? In Michie, D., Hayes, J., Pao, Y.H., eds.: Machine Intelligence 10. Ellis Horwood (1981)
16. Bird, R.S., de Moor, O.: Solving optimisation problems with catamorphism. In Bird, R.S., Morgan, C., Woodcock, J., eds.: MPC. Volume 669 of Lecture Notes in Computer Science., Springer (1992) 45–66
17. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2) (1990) 231–248
18. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-logic programming: Extending logic programming with coinduction. In Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A., eds.: ICALP. Volume 4596 of Lecture Notes in Computer Science., Springer (2007) 472–483
19. Tarau, P., Dahl, V.: High-Level Networking with Mobile Code and First Order AND-Continuations. Theory and Practice of Logic Programming **1**(3) (May 2001) 359–380 Cambridge University Press.
20. Carro, M., Hermenegildo, M.V.: Concurrency in prolog using threads and a shared database. In: ICLP. (1999) 320–334
21. Wielemaker, J.: Native preemptive threads in swi-prolog. In Palamidessi, C., ed.: ICLP. Volume 2916 of Lecture Notes in Computer Science., Springer (2003) 331–345
22. De Bosschere, K., Tarau, P.: Blackboard-based Extensions in Prolog. Software — Practice and Experience **26**(1) (January 1996) 49–69
23. Jeff Thompson: Yield Prolog. Project URL http://yieldprolog.sourceforge.net.
24. Peyton Jones, S.L., ed.: Haskell 98 Language and Libraries: The Revised Report. (September 2002) http://haskell.org/definition/haskell98-report.pdf.
25. Moggi, E.: Notions of computation and monads. Information and Computation **93** (1991) 55–92
26. Wadler, P.: Comprehending monads. In: ACM Conf. Lisp and Functional Programming, Nice, France, ACM Press (1990) 61–78

Secure Implementation of Meta-predicates *

Paulo Moura

Dep. of Computer Science, University of Beira Interior, Portugal
Center for Research in Advanced Computing Systems, INESC–Porto, Portugal
pmoura@di.ubi.pt

Abstract. This paper identifies potential security loopholes in the implementation of support for meta-predicates. Closing these loopholes depends on three conditions: a clear distinction between closures and goals, support for an extended meta-predicate directive that allows the specification of closures, and the availability of the `call/2-N` family of built-in meta-predicates. These conditions provide the basis for a set of simple safety rules that allows meta-predicates to be securely supported. These safety rules are currently implemented by Logtalk, an object-oriented logic programming language, and may also be applied in the context of Prolog predicate-based module systems. Experimental results illustrate how these rules can prevent several security problems, including accidental or malicious changes to the original meta-predicate arguments and bypassing of predicate scope rules and predicate scope directives.

Keywords: logic-programming, meta-predicates, security

1 Introduction

Prolog and Logtalk [1, 2] meta-predicates are predicates with one or more arguments that are called as goals on the body of a predicate clause. A typical example is the `findall/3` predicate whose second argument is used for generating solutions that are collected into a list. Meta-arguments may also be closures. In the context of this paper, a closure is defined as a callable term used to construct a goal by appending one or more arguments. The archetypal example is a list mapping predicate that succeeds when a closure can be successfully applied to each element in the list. Meta-predicates are particularly useful in the presence of an encapsulation mechanism such as a module system or an object-oriented extension. Defining an exported or public meta-predicate within a module or an object allows client modules and objects to reuse predicates customized by calls to local predicates.

Meta-predicates require special care in the context of Prolog module systems and object-oriented extensions as meta-calls must be executed in the meta-predicate *calling context* and not in the meta-predicate *definition context*.

* This work is partially supported by the FCT research project MOGGY (PTDC/EIA/70830/2006).

A recent paper [3] showed that the implementation of meta-predicates found in most Prolog *predicate-based* module systems allows a module to call non-exported predicates of another module, thus breaking encapsulation. This problem is usually absent from *atom-based* module systems such as XSB [4] where atoms, including predicate functors, are internally tagged with the definition module. The lack of enforcement of module encapsulation can, however, be thought as a consequence of the original design goals of module systems. Traditional Prolog module systems never aimed to fulfill any security role, being designed instead as a simple solution for partitioning code in different namespaces. Moreover, in most Prolog module systems, any module predicate can be called by using explicit module qualification (Ciao [5, 6] and ECLiPSe [7] are notable exceptions, only allowing calls to exported module predicates). Prolog extensions such as Logtalk, however, are designed to enforce encapsulation and predicate scope rules. In this case, meta-predicates must be properly supported without the danger of providing the means of accidental or malicious bypassing of predicate scope directives. The same paper also exposed flaws in the Logtalk support of meta-predicates which allowed bypassing of predicate scope directives. These flaws resulted from clever use of closures and from unsafe handling of goal execution context in the presence of meta-calls. During our research to correct these problems, we uncovered other meta-predicate implementation flaws that are not necessarily related to bypassing of predicate scope directives. In fact, potential loopholes exist that may allow accidental or carefully crafted meta-predicate definitions to change the original meta-predicate call. These changes may allow calling a different predicate in the calling context or calling the intended predicate with corrupted arguments. Calling a predicate different from the one specified in the original meta-predicate call is always a flaw, even when the called predicate is public or exported. Corrupting the original meta-predicate arguments can be done conditionally, resulting in hard to find problems as only specific usage patterns will lead to compromised results.

Correcting these flaws can be accomplished by finding and implementing a set of safety rules that ensures secure compilation and use of meta-predicates. Although our research takes place in the context of the Logtalk programming language, these safety rules are equally relevant in the context of predicate-based Prolog module systems (the proposed safety rules are not tied to the semantic differences between objects and modules). These safety rules are useful even in the context of module systems that allow the `:/2` control construct to bypass predicate scope rules, promoting better coding standards for meta-predicate definitions.

This paper is organized as follows. Section 2 describes an extended meta-predicate declaration directive, which supports the specification of both goals and closures as meta-arguments. Section 3 discusses how meta-calls can be constructed from closures. Section 4 enumerates potential loopholes in the implementation of meta-predicate support. Section 5 presents and discusses the safety rules applied by Logtalk to compile and execute meta-predicates. Section 6 identifies limitations imposed by our safety rules on meta-predicate definitions. Sec-

tion 7 presents experimental results in testing common Prolog module systems for the loopholes discussed in this paper. Section 8 presents our conclusions on safe compilation and use of meta-predicates, together with some remarks on the importance of increasing the awareness of security issues among the Logic Programming community.

2 Extended Meta-predicate Directive

User meta-predicates are declared using *meta-predicate directives*. These directives use a meta-predicate template to specify which arguments are *meta-arguments*, i.e. which arguments will be used as goals or closures in the body of the meta-predicate clauses. In plain Prolog, meta-predicate directives are optional and primarily useful for cross-reference tools. When module or object systems are present, meta-predicates directives are required for proper compilation of meta-predicates. An example of a Logtalk meta-predicate directive where the meta-arguments are goals is:

```
:- meta_predicate(findall(*, ::, *)).
```

In meta-predicate templates, the atom `::` represents a meta-argument that will be called as a goal. Normal arguments are represented by the atom `*`. This is similar to the declaration of meta-predicates found in most Prolog compilers and in the ISO Prolog standard for modules [8] (the atom `::` is used instead of the atom `:` for consistency with the Logtalk message sending operators). A positive integer, `N`, specifies a closure that will be used to construct a call by appending `N` arguments. For example:

```
| ?- map(double, [1, 2, 3], L).
L = [2, 4, 6]
yes
```

The corresponding `meta_predicate/1` directive would be:

```
:- meta_predicate(map(2, *, *)).
```

The first argument in the `map/3` template specifies that the meta-argument is a closure that will be used to construct a meta-call by appending two arguments. In the example above, this requires the existence of a `double/2` predicate in the calling context of the meta-predicate.

The use of non-negative integers to specify closures was first introduced in Quintus Prolog [9] for providing information to predicate cross-reference tools. A description of this usage can also be found on a recent Prolog standardization proposal [10]. Other Prolog compilers, such as SICStus Prolog [11] and YAP [12], also accept this notation for compatibility with existing code. As discussed later in this paper, the support for specifying closures in meta-predicate directives is essential to ensure safe compilation and use of meta-predicates. The Ciao Prolog system defines an alternative but equivalent syntax for specifying closures, using a compound term `pred(I)` where `I` is the number of extra arguments.

3 From Closures to Meta-calls

Given a closure and its additional arguments, the corresponding meta-call is constructed by appending the extra arguments to the existing ones. Although it is always possible to use the standard predicate `=../2` and a list append predicate to construct the meta-call, the preferable and simpler solution is to use the `call/N` family of built-in meta-predicates found in Logtalk and in most Prolog compilers. The first argument of these predicates must be a closure, with the remaining arguments being interpreted as the closure extra arguments. For example, the query `call(integer, 3)` is equivalent to the query `integer(3)`. These predicates provide improved performance when compared with the explicit construction of meta-calls (which requires building temporary lists).

As discussed later in the paper, the use of the `call/N` family of built-in meta-predicates is mandatory when working with closures as they avoid the introduction of new variables to explicitly represent the constructed meta-calls.

4 Potential Meta-predicate Loopholes

When reasoning about meta-predicate semantics, it is helpful to define a set of terms which helps us visualize how and where meta-calls take place:

Definition context This is the object or module containing the meta-predicate definition.

Calling context This is the object or module from which a meta-predicate is called. This can be the object or module where the meta-predicate is defined in the case of a local call or another object or module assuming that the meta-predicate is within scope.

Execution context This comprises both the calling context and the definition context. It includes all the information needed for the language runtime to execute a meta-predicate call.

Our research is focused on three potential loopholes when implementing meta-predicate support. The first loophole can be exploited to corrupt the original meta-arguments when a meta-predicate is executed:

Making malicious changes to meta-arguments Using unification with the meta-arguments may allow a meta-predicate to test for specific goals and closures and modify them before making the corresponding meta-calls. This potential loophole can be exploited by testing only for some very specific usage patterns, thus making its detection harder.

The two following loopholes can be exploited to bypass predicate scope directives or to break predicate scope rules. In the case of Logtalk, predicate scope rules are supported using predicate scope directives (object predicates are private by default). In the case of Prolog module systems, it should not be possible to call non-exported predicates from client modules.

Hijacking of the predicate execution context Hijacking a predicate execution context may allow a meta-predicate to gain access to predicates within the calling context other than the ones specified in the meta-predicate call.

Using closures for constructing unintended meta-calls A potential loophole exists when appending additional arguments to a closure in order to construct a meta-call. This loophole can be exploited by constructing a call to a predicate with the same functor of the closure but with an arity different to that intended by the caller of the meta-predicate.

5 Compiling Meta-predicates for Safety

This section describes four safety rules, illustrated with examples,[1] intended to close the loopholes discussed above in the context of predicate-based encapsulation module and object systems. The ideal rules would allow catching all problems at compile time. Unfortunately, as we will illustrate in this section, this is not always possible. Some deceiving meta-predicates definitions constitute perfectly valid code; the potential for trouble resulting only from the use of such definitions. For these cases, the compiler can still print a warning. At runtime, our safety rules ensure that any inappropriate use of a meta-predicate definition is caught by generating an appropriate exception.

The first two rules check for the context for meta-predicate calls. The last two rules check for the consistency of meta-predicate directives and the consistency between meta-predicate directives and meta-calls. The rules presentation is conceptual: actual implementations may choose to combine the first and second rules and combine the third and fourth rules. The first three rules are expected to be implemented at the compiler level. The fourth rule may be implemented instead in a programming code style or policy checker.

(a) The meta-arguments on a meta-predicate clause head must be variables.

This simple rule helps to prevent a meta-predicate from modifying the original arguments of a meta-call. By testing and acting upon the actual meta-arguments, a meta-predicate could try to make a meta-call different from the original one to be executed in the calling context. Consider the following example `(a)`:

```
:- object(library).

    :- public(map/3).
    :- meta_predicate(map(*, 2, *)).
    map(In, scale(_), Out) :-
        !, map_(In, scale(3), Out).
    map(In, Closure, Out) :-
        map_(In, Closure, Out).
```

[1] These examples use Logtalk objects. Converting them to Prolog modules requires replacing object directives with module directives, removing the explicit predicate scope directives, and rewriting the meta-predicate directives.

```
    :- meta_predicate(map_(*, 2, *)).
    map_([], _, []).
    map_([X| Xs], Closure, [Y| Ys]) :-
        call(Closure, X, Y),
        map_(Xs, Closure, Ys).

:- end_object.
```

The `map/3` meta-predicate in this `library` object behaves as expected except when the closure argument unifies with the term `scale(_)`. In this case, the original predicate argument is simply ignored and replaced by a fixed value. Assume now that we define the following client object:

```
:- object(client).

    :- public(double/2).
    double(Ints, Doubles) :-
        library::map(Ints, scale(2), Doubles).

    scale(Scale, X, Xscaled) :-
        Xscaled is X*Scale.

:- end_object.
```

In the absence of this safety rule, the compromised behavior of the `map/3` meta-predicate could be illustrated by the following goal:

```
| ?- client::double([1,2,3], Doubles).
Doubles = [3,6,9]
yes
```

By implementing this safety rule, Logtalk generates a compile time error[2] for the first clause of the `map/3` predicate in the `library` object:

```
type_error(variable, scale(_))
```

This rule is, however, easy to circumvent by simply moving the unification from the meta-predicate clause head into the clause body. The meta-predicate `map/3` in the example above can be easily rewritten as:

```
map(In, Closure, Out) :-
    (   Closure = scale(_) ->
        map_(In, scale(3), Out)
    ;   map_(In, Closure, Out)
    ).
```

Despite this weakness, there are three reasons to include this rule. First, it provides a necessary condition for the second safety rule, described next. Second, rule violations result in compile time errors, which are always preferable to runtime errors. Third, it is trivial to implement: the compiler can apply it before any other rule by simply checking the meta-arguments in the clause heads.

[2] Arguably, this error is more of a representation error than a type error; nevertheless, we decided to follow the practice established by the current ISO Prolog standard.

(b) Meta-calls whose arguments are not variables appearing in meta-argument positions in the clause head must be compiled as calls to local predicates.

This rule applies to the compilation of both meta-predicates and normal predicates. It prevents hijacking of the execution context, which could otherwise be used to call predicates in the calling context not passed as meta-arguments. This problem can occur with e.g. a naive implementation of execution context passing from a clause head to the goals in the clause body.

This rule is trivial to implement when compiling clauses of normal predicates: any meta-call in a clause body must be compiled as a local meta-call. This rule is also easy to implement when compiling clauses of meta-predicates since the corresponding meta-predicate directive is mandatory.

As a consequence of this rule, when a meta-predicate calls a second meta-predicate, the meta-arguments executed in the calling context will be strictly the ones coming from the call to the first meta-predicate. That is, the programmer cannot use a second meta-predicate to construct a meta-call different from the one intended by the original caller of the meta-predicate. Consider the following example (`b1`):

```
:- object(library).

    :- public(meta/2).
    :- meta_predicate(meta(::, ::)).
    meta(Goal1, Goal2) :-
        call(Goal1), call(Goal2).

    :- public(meta/1).
    :- meta_predicate(meta(::)).
    meta(Goal1) :-
        meta(Goal1, local).

    local :-
        write('local predicate in object library'), nl.

:- end_object.
```

The rule requires that client calls to the `meta/1` predicate must result in the interpretation of `local/0` as a call to a local predicate, thus executed in the context of the object `library`. We use the following client object to illustrate the correct behavior:

```
:- object(client).

    :- public(test/0).
    test :-
        library::meta(goal).

    goal :-
        write('goal meta-argument in object client'), nl.
```

```
    local :-
        write('local predicate in object client'), nl.

:- end_object.
```

This safety rule will ensure the following result:

```
| ?- client::test.
goal meta-argument in object client
local predicate in object library
yes
```

Meta-calls can also appear in the body of normal predicates. This rule ensures that an object cannot hijack the execution context of the original, non meta-predicate call and use it through a local meta-predicate to construct arbitrary calls to predicates in the calling context. Therefore, we cannot convert a normal argument into a meta-argument by calling a local meta-predicate. Consider the following simplified version of an example found in [3] (`b2`):

```
:- object(library).

    :- meta_predicate(meta(::)).
    meta(Goal) :-
        call(Goal).

    :- public(normal/1).
    normal(Arg) :-
        meta(Arg).

:- end_object.
```

In this case, the argument in the meta-predicate call, `Arg`, must be interpreted as a local meta-call. Consider now the following client object:

```
:- object(client).

    :- public(test/0).
    test :-
        library::normal(term).

    term :-
        write('Some local, private predicate.').

:- end_object.
```

This safety rule will ensure the following result:

```
| ?- catch(client::test, E, write(E)).
E = error(existence_error(procedure,term), context(object,library,_))
yes
```

Therefore, the predicate `term/0` in the object `client` (which is the calling context for the `normal/1` predicate) will not be called.

Although the two examples above make use of additional user-defined meta-predicates whose meta-arguments are goals, the rule also applies when working with closures and when calling built-in meta-predicates. For example, consider the following tentative exploit (`b3`) using the `call/1` built-in meta-predicate and a meta-predicate definition that does not comply with the corresponding directive (as two arguments are appended to the closure instead of one):

```
:- object(library).

    :- public(m/2).
    :- meta_predicate(m(1, *)).
    m(Closure, Arg) :-
        Closure =.. List,
        list::append(List, [Arg, _], NewList),
        Call =.. NewList,
        call(Call).

:- end_object.
```

With this safety rule in place, the meta-call `call(Call)` above is compiled as a local meta-call since the variable `Call` does not occur in the head of the meta-predicate clause in a meta-argument position. The following definition of a simple client object illustrates the consequences of the meta-predicate definition above:

```
:- object(client).

:- public(test/1).
    test(X) :-
        library::m(a, X).

    a(1). a(2).

    a(3, three). a(4, four).

:- end_object.
```

After compiling and loading these two objects, an example test query would be:

```
?- catch(client::test(X), E, true).
E = error(existence_error(procedure, a/2), context(object, library, _))
yes
```

As the exception term shows, the meta-call is compiled and executed as a local call in the context of the `library` object. Without this safety rule in place, a faulty implementation would wrongly call the predicate `a/2` defined in the object `client`:

```
?- catch(client::test(X), E, true).
X = 3 ;
X = 4
yes
```

The above example shows that meta-predicates with meta-arguments that are closures cannot be defined using `call/1` calls as explicitly constructing the meta-call from the closure results in a new variable not occurring in the clause head. It follows that the use of the `call/2-N` built-in predicates is mandatory for defining meta-predicates that work with closures. This is subsumed by the third rule:

(c) Meta-predicate closures must be used within a `call/2-N` built-in predicate call that complies with the corresponding meta-predicate directive.

The number of additional arguments appended to a closure in a `call/2-N` call must comply with the meta-predicate declaration; simply ensuring that a closure is a variable occurring in a meta-argument position is not a sufficient condition. This rule ensures that a meta-predicate cannot construct a predicate call with the same functor but with a different arity of the original meta-argument. For example, a meta-predicate definition `(c)` such as:

```
:- meta_predicate(map(1, *)).
map(Closure, [Element| Rest]) :-
    ..., call(Closure, Element, Result), ...
```

would result in the following compile time error:

```
arity_mismatch(closure, call(map, Element, Result), map(1, *))
```

The `call/3` meta-call in this example does not comply with the meta-predicate specification, which requires a single additional argument. In fact, the actual meta-call would not be the one that the programmer intended when calling the meta-predicate. Moreover, the call could correspond either to a predicate in the calling context that is not within scope of the meta-predicate definition context or to a non-existing predicate (which would result in a runtime existence error).

(d) The meta-predicate arity should be equal to the sum of the extra arguments specified by each closure plus the number of normal, non meta-arguments.

Assume that we correct the meta-predicate directive used to illustrate the previous rule in order to be consistent with the `call/2-N` call by writing `(d)`:

```
:- meta_predicate(map(2, *)).
```

Trying to compile the updated code would result in the following error:

```
arity_mismatch(closure, map(Closure, [Element| Rest]), map(2, *))
```

This error results from the meta-predicate directive specifying a closure requiring two extra arguments while only one normal argument is declared. This is potentially misleading for a client that may expect the library meta-predicate to call a unary predicate based on the meta-predicate arity.

6 Known Limitations

6.1 Closures with a Variable Number of Arguments

The proposed safety rules and the extended meta-predicate directive do not support the specification of meta-predicates that allow a *variable* number of arguments to be appended to a closure. This restriction makes some common meta-predicates such as `apply/2` useless as a public or exported predicate. The usual definition of this predicate is:

```
apply(Closure, Args) :-
    Closure =.. List,
    append(List, Args, NewList),
    Call =.. NewList,
    call(Goal).
```

As the variable `Goal` is not a meta-argument in the clause head, the meta-call `call(Goal)` is compiled as a call to a local predicate (as per the second safety rule) and not as a call to a predicate in the calling context of the meta-predicate. This restriction is not considered, however, a serious limitation as the number of extra closure arguments is usually known *a priori*, therefore allowing the use of the `call/2-N` built-in meta-predicates.

6.2 Meta-predicates Implemented in Foreign Code

Prolog compilers often include libraries with predicates implemented using a foreign language interface. It is also possible to implement meta-predicates this way. A common example is the implementation of callbacks to Prolog code in the context of GUI extensions (see e.g. the SWI-Prolog XPCE package [13]). In this case, the verification of the safety rules described in the previous section would require manual verification of the source code in the foreign language. It should be noted, however, that the use of foreign language resources rises its own set of security issues that goes well beyond meta-predicates issues.

7 Prolog Module Systems

In this section, we test five Prolog compilers for the potential meta-predicates loopholes described earlier: Ciao 1.10#8, ECLiPSe 5.10#141, SICStus Prolog 4.0.2, SWI-Prolog 5.6.59, and YAP 5.1.3. Although there are other Prolog compilers supporting predicate-based module systems, we believe this is a representative set of module implementation solutions.

Our experiments are complicated by two problems. First, the details of the module versions of the examples in Section 4 differ for each compiler due to the lack of a de-facto standard for Prolog module systems.[3] In particular, the five

[3] The full source code used in the examples for both Logtalk and the tested Prolog compilers is available at `http://logtalk.org/papers/simp/mptests.tar.gz`.

tested systems provide three different materializations of a meta-predicate declaration directive. Second, the documentation of the Prolog module systems often forces us to resort to experimentation in order to find out the exact operational semantics of modules, meta-predicate directives, and meta-calls.

The experimental results are presented in Table 1. In this table, a value of `N/A` means that the `meta_predicate/1` directive or its equivalent does not support the specification of meta-predicate templates. The results for the example `(d)` indicate if a Prolog compiler checks for the consistency between meta-predicate directives and the number of extra arguments required by the declared closures. This consistency check should result, at least, in a compilation warning but it is not performed by any of the tested Prolog compilers.

Table 1. Experimental results for the safety rule examples.

Examples	Ciao	ECLiPSe	SICStus	SWI (mp)	SWI (mt)	YAP
(a1)	ok	wrong	ok	wrong	wrong	ok
(a2)	ok	ok	wrong	ok	ok	wrong
(b1)	ok	wrong	ok	wrong	wrong	ok
(b1)	ok	wrong	ok	wrong	wrong	ok
(b2)	ok	ok	ok	wrong	ok	ok
(b3)	ok	wrong	wrong	wrong	wrong	wrong
(c)	ok	N/A	wrong	wrong	N/A	wrong
(d)	wrong	wrong	wrong	wrong	wrong	wrong

The conversion of the Logtalk example `(a)` into Prolog module code rises an interesting issue with the module systems of SICStus Prolog and YAP. These systems expand meta-arguments in goals appearing in the body of meta-predicate clauses but not in the head of meta-predicate clauses. As a consequence, the first clause of the `map/3` is never used, making the test result for these Prolog compilers misleading. One workaround is to rewrite this clause using explicit module qualification, which allows all the clauses to be used. Although this rewrite defeats the purpose of the meta-predicate directive, it is also a possible exploit vector. Therefore, we chose to split the example `(a)` in two tests. Test `(a1)` uses the same exact clauses as in example `(a)`. Test `(a2)` uses explicit module qualification for the `scale/1` arguments in the first clause of the meta-predicate `map/3`.

The results for test `(a2)` are interesting and a bit surprising. While the results for SICStus Prolog and YAP are expected, the changes in test `(a2)` allow both ECLiPSe and SWI-Prolog to return correct results, reversing the bad score in test `(a1)` (it is worth noting that the module systems of ECLiPSe and SWI-Prolog are distinct). The Ciao compiler is not fooled by these tricks.

Another interesting result concerns the `(b2)` and `(b3)` examples of our second security rule, `(b)`. All compilers behaved correctly in example `(b2)`. However, with the exception of Ciao, all compilers provided a wrong answer for example `(b3)`, allowing access to a private predicate, `a/2`, in the client module, instead of restricting the access to the predicate `a/1` used as argument in the meta-predicate call. In this case, these Prolog compilers acted properly when meta-arguments are goals but not when the meta-arguments are closures.

Some brief, Prolog compiler-specific comments about the results follow:

Ciao. This is the only tested Prolog compiler that disallows writing meta-predicate directives inconsistent with the meta-predicate definitions. It is also the Prolog compiler that scored the best test results (as expected, giving the emphasis by Ciao developers in static code analysis). The test results for the third example of our second security rule (`b3`) are particularly interesting. The Ciao compiler correctly catches our attempts to specify a closure with a single extra argument while, at the same time, defining the meta-predicate to call the closure with two extra arguments.[4] Correcting the meta-predicate directive to specify a closure with two extra arguments, however, results in the definition of a meta-predicate that only allows a single extra argument to be passed. The Ciao compiler fails to warn the user of this potential problem when compiling the example (`d`).

ECLiPSE. This compiler does not provide a `meta_predicate/1` directive, relying instead on a proprietary `tool/2` directive whose arguments are predicate indicators. Thus, this directive does not allow the programmer to define meta-predicate templates. The test examples are modified to use the `tool/2` directive and the built-in predicate `@/2` as suggested in the ECLiPSe documentation.

SICStus Prolog. This compiler allows the specification of closures in the directive `meta_predicate/1` but only for compatibility with existing code. Correcting the directive in the test example (`b3`) to make it consistent with the meta-predicate definition does not lead to a correct answer.

SWI-Prolog. We present two sets of results for SWI-Prolog. The first set, *mp*, uses an emulation of the `meta_predicate/1` directive provided in the compatibility libraries distributed with SWI-Prolog. The second set, *mt*, uses the SWI-Prolog native directive `module_transparent/1` whose argument is a predicate indicator. Therefore, it does not allow the programmer to define meta-predicate templates. We are discussing with the main SWI-Prolog developer the possible implementation of our safety rules as a component of a general style or policy checker, integrated with the current cross-referencer tool. This would allow existing code to be checked for possible violations without the danger of breaking it.

YAP. Similarly to SICStus Prolog, YAP accepts the specification of closures in the `meta_predicate/1` directive but only for compatibility with existing code. Correcting the directive in the example (`b3`) to match the meta-predicate definition does not result in a correct answer. The safety rules described in this paper are expected to be implemented in a forthcoming version of YAP. Their use is expected to be optional, enabled by a Prolog compiler flag.

[4] There is a typo in the Ciao documentation of the meta-predicate specification for closures. The notation `pred(N)` indicates the number of extra arguments, with the closure being used within a `call/N+1` predicate, not within a `call/N` predicate as described in the documentation.

8 Discussion and Conclusions

The safety rules described in this paper fix all known flaws on the Logtalk support for meta-predicates.[5] These rules may also be adapted and applied in the context of predicate-based Prolog module systems in order to correct the flaws uncovered by our experiments. However, given the syntactic and semantic differences among the implementations of Prolog modules systems, the existence of other loopholes is to be expected. Nevertheless, the lack of a formal guarantee that the proposed rules close all loopholes in current implementations should not excuse not fixing the known loopholes.

The safety rules are easy to implement and computationally inexpensive, as exemplified in the current Logtalk compiler implementation. These rules enjoy the nice property of all the required computations being performed at compile time. In the worst case, some of the rules imply that the use of a flawed meta-predicate definition results in a runtime exception due to the meta-calls being compiled as calls to local predicates and not as calls in the meta-predicate calling context. This is an unfortunate consequence of the fact that some safety violations only occur when using meta-predicate definitions that, per se, constitute perfectly valid code. It follows that the worst case cannot be improved by finding stronger compiler checking rules. At best, the compiler could issue a warning when compiling a public meta-predicate whose meta-calls are compiled as a local calls for safety reasons.

The extended `meta_predicate/1` directive described in this paper provides essential information for preventing misuse of closures. We show that specifying closures using positive integers is not just an optional feature, useful for cross-reference and documenting tools or for compatibility reasons, but a necessary feature for safe compilation and use of meta-predicates.

Calls constructed from closures must be made by using the `call/2-N` built-in predicates. This allows the consistency between the meta-predicate directives and definitions to be checked at compile time, preventing loopholes when appending arguments to a closure in order to construct a meta-call. The `call/2-N` family of built-in predicates is already provided by most Prolog compilers and is included in the current draft of the ISO Prolog Core revision standardization proposal.[6]

There is currently no formal proof that the described safety rules are sufficient to prevent highjacking of predicate execution context and the misuse of closures in the context of Logtalk. In the case of Prolog module systems each module system needs a proof, as there is no de-facto standard. These proofs would need to be based on formal descriptions of the module systems, to be provided by their authors; these descriptions are beyond the scope of this paper.

[5] All the safety rules are implemented by the Logtalk compiler since version 2.30.6.

[6] In the case of Logtalk, although its current version uses a Prolog system as a back-end compiler, its implementation of the `call/2-N` built-in predicate does not depend on the availability of the `call/2-N` Prolog built-in predicates.

The problems described in this paper are representative of what can go wrong when using meta-predicates in field applications where security is a basic requirement. It is worth noting that the flaws described in this paper are not always evident from a quick inspection of compromised source code (which, by itself, assumes its availability). Despite existing research on improving module systems (see e.g. [3, 6]), security concerns are often overlooked by Prolog implementors and programmers. Secure implementation of meta-predicates is just one of the topics where compilers and language runtimes must perform securely. In a scenario of increasing industrial use of Prolog-based solutions, either in embedded form or as stand-alone applications, preemptive thinking about security issues is necessary. In this regard, the Prolog community is still far from the security mindset found in other programing communities.

Acknowledgements. We are grateful to Rémy Haemmerlé and François Fages for bringing to our attention flaws on the implementation of meta-predicates in earlier versions of Logtalk. We thank also Jan Wielemaker and Sara Madeira for their comments and help in revising this paper.

References

1. Moura, P.: Logtalk 2.33.0 User Manual. (September 2008)
2. Moura, P.: Logtalk – Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal (September 2003)
3. Haemmerlé, R., Fages, F.: Modules for Prolog Revisited. In Etalle, S., Truszczyński, M., eds.: International Conference on Logic Programming 2006. Number 4079 in LNCS, Springer-Verlag (August 2006) 41–55
4. Group, T.X.R.: The XSB Programmer's Manual: version 3.1. (2007)
5. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López, P., Puebla, G.: The Ciao Prolog System. Technical Report CLIP 3/97.1, The CLIP Group, School of Computer Science, Technical University of Madrid (December 2002)
6. Gras, D.C., Hermenegildo, M.V.: A New Module System for Prolog. In: CL'00: Proceedings of the First International Conference on Computational Logic, London, UK, Springer-Verlag (2000) 131–148
7. Cheadle, A.M., Harvey, W., Sadler, A.J., Schimpf, J., Shen, K., Wallace, M.G.: ECLiPSe: A tutorial introduction. Technical Report IC-Parc-03-1, IC-Parc, Imperial College, London (2003)
8. ISO/IEC: International Standard ISO/IEC 13211-2 Information Technology — Programming Languages — Prolog — Part II: Modules. ISO/IEC (2000)
9. for Computer Science, S.I.: Quintus Prolog 3.5 User's Manual. (2003)
10. O'Keefe, R.: An Elementary Prolog Library. `http://www.cs.otago.ac.nz/staffpriv/ok/pllib.htm`
11. for Computer Science, S.I.: SICStus Prolog 4.0.2 User Manual. (2007)
12. Costa, V.S.: The YAP User's Manual: version 5.1.3. (2008)
13. Wielemaker, J., Anjewierden, A.: An Architecture for Making Object-Oriented Systems Available from Prolog. In: Proceedings of the 12th International Workshop on Logic Programming Environments. (2002) 97–110

Tabling Logic Programs in a Common Global Trie

Jorge Costa and Ricardo Rocha

DCC-FC & CRACS
University of Porto, Portugal
c0607002@alunos.dcc.fc.up.pt ricroc@dcc.fc.up.pt

Abstract. The performance of tabled evaluation largely depends on the implementation of the table space. Arguably, the most successful data structure for tabling is tries. However, while tries are efficient for variant based tabled evaluation, they are limited in their ability to recognize and represent repeated answers for different calls. In this paper, we propose a new design for the table space where terms in a tabled subgoal call or/and answer are stored in a common global trie instead of being spread over several different tries. Our preliminary experiments using the YapTab tabling system show very promising reductions on memory usage.

Keywords: Tabling, Table Space, Implementation.

1 Introduction

Tabling [1–3] is an implementation technique where intermediate answers for subgoals are stored and then reused whenever a repeated call appears. The performance of tabled evaluation largely depends on the implementation of the table space – being called very often, fast lookup and insertion capabilities are mandatory. Applications can make millions of different calls, hence compactness is also required. Arguably, the most successful data structure for tabling is *tries* [4]. Tries meet the previously enumerated criteria of efficiency and compactness.

Used in applications that pose many queries, possibly with a large number of answers, tabling can build arbitrarily many and/or very large tables, quickly filling up memory. A possible solution for this problem is to dynamically abolish some of the tables. This can be done using explicit tabling primitives or using a memory management strategy that automatically recovers space among the least recently used tables when memory runs out [5]. An alternative approach is to store tables externally in a relational database management system and then reload them back only when necessary [6].

A complementary approach to the previous problem is to study how less redundant, more compact and more efficient data structures can be used to better represent the table space. While tries are efficient for variant based tabled evaluation, they are limited in their ability to recognize and represent repeated

answers for different calls. In [7], Rao *et al.* proposed a table organization using *Dynamic Threaded Sequential Automata* (DTSA) which recognizes reusable subcomputations for subsumption based tabling. In [8], Johnson *et al.* proposed an alternative to DTSA, called *Time-Stamped Trie* (TST), which not only maintains the time efficiency of the DTSA but has better space efficiency.

In this paper, we propose a different approach. We propose a new design for the table space where all terms in a tabled subgoal call or/and answer are stored in a *common global trie* instead of being spread over several different trie data structures. Our approach resembles the *hash-consing* technique [9], as it tries to share data that is structurally equal. An obvious goal is to save memory usage by reducing redundancy in term representation to a minimum. We will focus our discussion on a concrete implementation, the YapTab system [10, 11], but our proposals can be easy generalized and applied to other tabling systems.

The remainder of the paper is organized as follows. First, we briefly introduce some background concepts about tries and the table space. Next, we describe YapTab's new design for the table space organization using the common global trie and then, we describe how we have extended YapTab to provide engine support for our approach. At last, we present some preliminary experimental results and we end by outlining some conclusions.

2 Table Space

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for tabled subgoals in an appropriate data space, called the *table space*. Whenever a repeated tabled call is found, the subgoal's answers are recalled from the table space instead of being re-evaluated against the program clauses. The table space may be accessed in a number of ways: **(i)** to find out if a subgoal is in the table and, if not, insert it; **(ii)** to verify whether a newly found answer is already in the table and, if not, insert it; and **(iii)** to load answers to variant subgoals. With these requirements, YapTab implements its table space using *tries* [12] which is regarded a very efficient way to implement tables [4].

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term. Each root-to-leaf path represents a term described by the tokens labelling the nodes traversed. Two terms with common prefixes will branch off from each other at the first distinguishing token. For example, the tokenized form of the term $p(X, q(Y, X), Z)$ is the stream of 6 tokens: $p/3, VAR_0, q/2, VAR_1, VAR_0, VAR_2$. Variables are represented using the formalism proposed by Bachmair *et al.* [13], where each variable in a term is represented as a distinct constant. Formally, this corresponds to a function, $numbervar()$, from the set of variables in a term t to the sequence of constants $VAR_0, ..., VAR_N$, such that $numbervar(X) < numbervar(Y)$ if X is encountered before Y in the left-to-right traversal of t.

Internally, the trie nodes are 4-field data structures. The first field stores the node's token, the second field stores a pointer to the node's first child, the third field stores a pointer to the node's parent and the fourth field stores a pointer

to the node's next sibling. Each node's outgoing transitions may be determined by following the child pointer to the first child node and, from there, continuing through the list of sibling pointers. To increase performance, YapTab enforces the *substitution factoring* [4] mechanism and implements tables using two levels of tries - one for subgoal calls, the other for computed answers. More specifically, the table space of YapTab is organized in the following way:

- each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the predicate's *subgoal trie*.
- each different subgoal call is represented as a unique path in the subgoal trie, starting at the predicate's table entry and ending in a *subgoal frame* data structure, with the argument terms being stored within the path's nodes.
- the subgoal frame data structure acts as an entry point to the *answer trie*.
- each different subgoal answer is represented as a unique path in the answer trie. Oppositely to subgoal tries, answer trie paths hold just the substitution terms for the free variables which exist in the argument terms of the corresponding subgoal call.
- the leaf's child pointer of answers is used to point to the next available answer, a feature that enables answer recovery in insertion order. The subgoal frame has internal pointers that point respectively to the first and last answer on the trie. Whenever a variant subgoal starts consuming answers, it sets a pointer to the first leaf node. To consume the remaining answers, it must follow the leaf's linked list, setting the pointer as it consumes answers along the way. Answers are loaded by traversing the answer trie nodes bottom-up.

An example for a tabled predicate `t/2` is shown in Figure 1. Initially, the subgoal trie is empty. Then, the subgoal `t(a(1),X)` is called and three trie nodes are inserted: one for the functor `a/1`, a second for the constant `1` and one last for variable `X`. The subgoal frame is inserted as a leaf, waiting for the answers. Next, the subgoal `t(a(2),X)` is also called. It shares one common node with `t(a(1),X)` but, having `a/1` a different argument, two new trie nodes and a new subgoal frame are inserted. At the end, the answers for each subgoal are stored in the corresponding answer trie as their values are computed. Note that, for this particular example, the completed answer trie for both subgoal calls is exactly the same.

3 Common Global Trie

We next describe YapTab's new design for the table space organization. In this new design, all terms in a tabled subgoal call or/and answer are now stored in a common global trie (GT) instead of being spread over several different trie data structures. The GT data structure still is a tree structure where each different path through the trie nodes corresponds to a term. However, here a term can end at any internal trie node and not necessarily at a leaf trie node.

The previous subgoal trie and answer trie data structures are now represented by a unique level of trie nodes that point to the corresponding terms in the GT

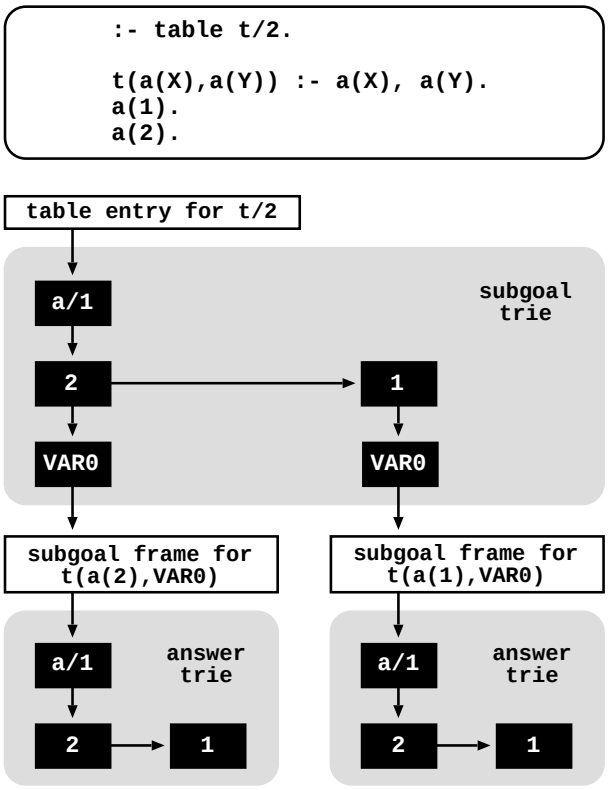


Fig. 1. YapTab's original table design

(see Figure 2 for details). For the subgoal tries, each node now represents a different subgoal call where the node's token is the pointer to the unique path in the GT that represents the argument terms for the subgoal call. The organization used in the subgoal tries to maintain the list of sibling nodes and to access the corresponding subgoal frames remains unaltered. For the answer tries, each node now represents a different subgoal answer where the node's token is the pointer to the unique path in the GT that represents the substitution terms for the free variables which exist in the argument terms. The organization used in the answer tries to maintain the list of sibling nodes and to enable answer recovery in insertion order remains unaltered. With this organization, answers are now loaded by following the pointer in the node's token and then by traversing the corresponding GT's nodes bottom-up.

On completion of a subgoal, a strategy exists that avoids answer recovery using bottom-up unification and performs instead what is called a *completed table optimization* [4]. This optimization implements answer recovery by top-down traversing the completed answer trie and by executing specific WAM-like code from the answer trie nodes. With our new design, the nodes in the GT can

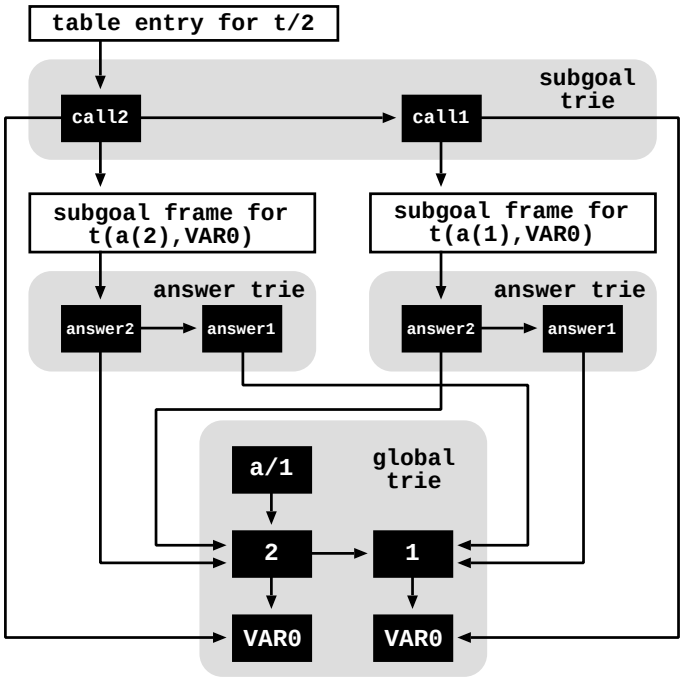


Fig. 2. YapTab's new table design

belong to several different subgoal/answer tries, and thus this optimization is no longer possible.

Figure 2 uses again the example from Figure 1 to illustrate how the GT's design works. Initially, the subgoal trie and the GT are empty. Then, the first subgoal `t(a(1),X)` is called and three nodes are inserted on the GT: one to represent the functor `a/1`, another for the constant `1` and a last one for variable `X`. Next, a node representing the path inserted on the GT is stored in the subgoal trie (node labeled `call1`). The token field for the `call1` node is made to point to the leaf node of the GT's inserted path and the child field is made to point to a new subgoal frame. For the second subgoal call, `t(a(2),X)`, we start again by inserting the call in the GT and then we store a node in the subgoal trie (node labeled `call2`) to represent the path inserted on the GT.

As we saw in the previous example, for each subgoal call we have two answers: the terms `a(1)` and `a(2)`. However, as these terms are already represented on the GT, we need to store only two nodes, in each answer trie, to represent them (nodes labeled `answer1` and `answer2`). The token field for these answer trie nodes are made to point to the corresponding term representation on the GT. With this example we can see that terms in the GT can end at any internal trie node (and not necessarily at a leaf trie node) and that a common path on the GT can simultaneously represent different subgoal and answer terms.

4 Implementation Details

We then describe in more detail the data structures and algorithms for YapTab's new table design based on the GT. We start with Figure 3 showing in more detail the table organization previously presented in Figure 2.

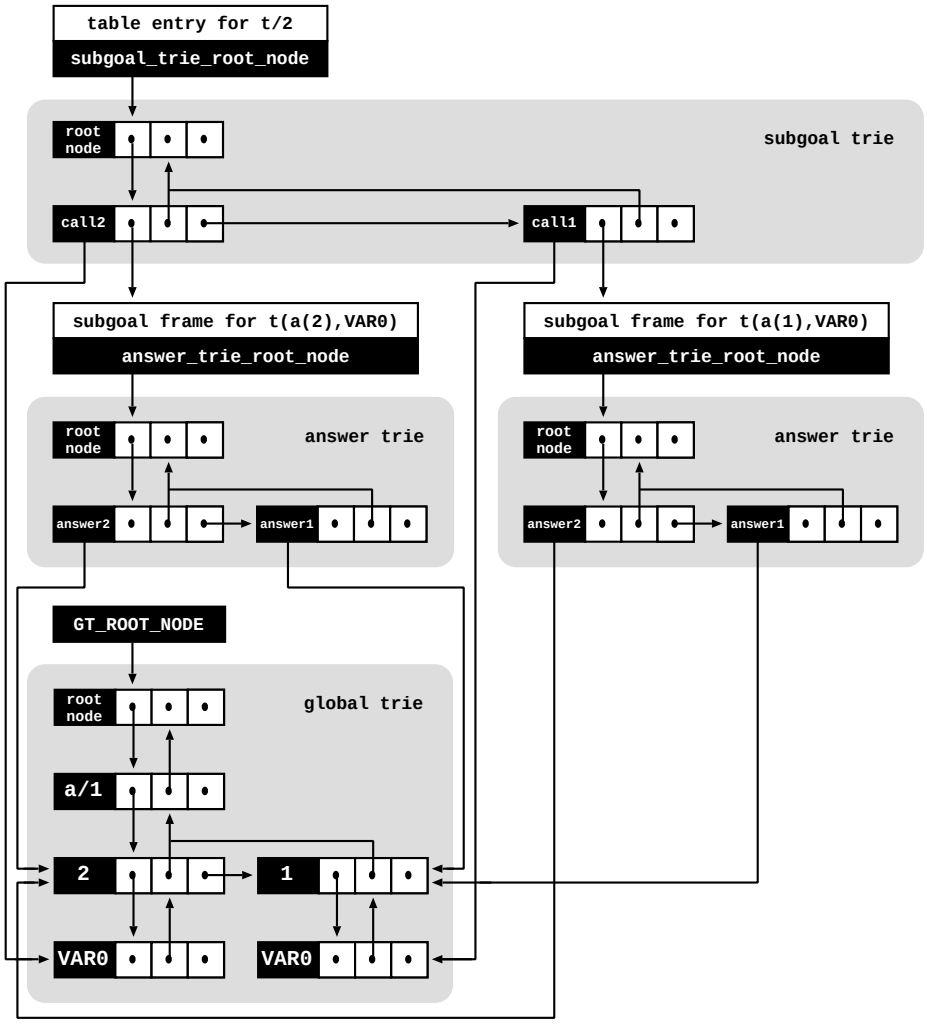


Fig. 3. Implementation details for YapTab's new table design

Internally, tries are represented by a top *root node*, acting as the entry point for the corresponding subgoal, answer or global trie data structure. For the subgoal tries, the root node is stored in the corresponding table entry's

`subgoal_trie_root_node` data field. For the answer tries, the root node is stored in the corresponding subgoal frame's `answer_trie_root_node` data field. For the global trie, the root node is stored in the `GT_ROOT_NODE` global variable.

Regarding the trie nodes, remember that they are internally implemented as 4-field data structures. The first field (`token`) stores the token for the node and the second (`child`), third (`parent`) and fourth (`sibling`) fields store pointers, respectively, to the first child node, to the parent node, and to the sibling node.

Traversing a trie to check/insert for new calls or for new answers is implemented by repeatedly invoking a `trie_node_check_insert()` procedure for each token that represents the call/answer being checked. Given a trie node `parent` and a token `t`, the `trie_node_check_insert()` procedure returns the child node of `parent` that represents the given token `t`. Figure 4 shows the pseudo-code for this procedure.

```
trie_node_check_insert(TRIE_NODE parent, TOKEN t) {
  child = parent->child
  if (child == NULL) {              // the list of sibling nodes is empty
    child = new_trie_node(t, NULL, parent, NULL)
    parent->child = child
  } if (is_not_a_hash_table(child)) {    // sibling nodes without hashing
    sibling_nodes = 0             // to count the number of sibling nodes
    do {           // check if token t is already in the list of siblings
      if (child->token == t)
        return child
      sibling_nodes++
      child = child->sibling
    } while (child)
    child = new_trie_node(t, NULL, parent, parent->child)
    if (sibling_nodes > MAX_SIBLING_NODES_PER_LEVEL) {  // alloc new hash
      hash = new_hash_table(child)
      parent->child = hash
    } else
      parent->child = child
  } else {                                  // sibling nodes with hashing
    hash = child
    bucket = hash_function(hash, t)    // get the hash bucket for token t
    child = bucket
    sibling_nodes = 0
    while (child) {     // check if token t is already in the hash bucket
      if (child->token == t)
        return child
      sibling_nodes++
      child = child->sibling
    }
    child = new_trie_node(t, NULL, parent, bucket)
    if (sibling_nodes > MAX_SIBLING_NODES_PER_BUCKET)      // expand hash
      expand_hash_table(hash)
  }
  return child
}
```

Fig. 4. Pseudo-code for the `trie_node_check_insert()` procedure

Initially, the procedure checks if the list of sibling nodes is empty. If this is the case, a new trie node representing the given token `t` is initialized and inserted as the first child of the given `parent` node. To initialize new trie nodes, we use a `new_trie_node()` procedure with four arguments, each one corresponding to the initial values to be stored respectively in the `token`, `child`, `parent` and `sibling` fields of the new trie node.

Otherwise, if the list of sibling nodes is not empty, the procedure checks if they are being indexed through a hash table. Searching through a list of sibling nodes is initially done sequentially. This could be too expensive if we have hundreds of siblings. A threshold value (`MAX_SIBLING_NODES_PER_LEVEL`) controls whether to dynamically index the nodes through a hash table, hence providing direct node access and optimizing search. Further hash collisions are reduced by dynamically expanding the hash tables when a second threshold value (`MAX_SIBLING_NODES_PER_BUCKET`) is reached for a particular hash bucket.

If not using hashing, the procedure then traverses sequentially the list of sibling nodes and checks for one representing the given token `t`. If such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the list. If reaching the threshold value `MAX_SIBLING_NODES_PER_LEVEL`, a new hash table is initialized and inserted as the first child of the given `parent` node.

If using hashing, the procedure first calculates the hash bucket for the given token `t` and then, it traverses sequentially the list of sibling nodes in the bucket checking for one representing `t`. Again, if such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the bucket list. If reaching the threshold value `MAX_SIBLING_NODES_PER_BUCKET`, the current hash table is expanded.

To manipulate tries we use two interface procedures. For traversing a trie to check/insert for new calls or for new answers we use the

```
trie_check_insert(TRIE_NODE root, TERM term)
```

procedure, where `root` is the root node of the trie to be used and `term` is the call/answer term to be inserted. The `trie_check_insert()` procedure invokes repeatedly the previous `trie_node_check_insert()` procedure for each token that represents the given `term` and returns the reference to the leaf node representing its path. Note that inserting a term requires in the worst case allocating as many nodes as necessary to represent its complete path. On the other hand, inserting repeated terms requires traversing the trie structure until reaching the corresponding leaf node, without allocating any new node.

To load a term from a trie back to the Prolog engine we use the

```
trie_load(TRIE_NODE leaf)
```

procedure, where `leaf` is the reference to the leaf node of the term to be returned. When loading a term, the trie nodes are traversed in bottom-up order.

When inserting terms in the table space we need to distinguish two situations: **(i)** inserting tabled calls in a subgoal trie structure; and **(ii)** inserting

answers in a particular answer trie structure. The former situation is handled by the `subgoal_check_insert()` procedure as shown in Figure 5 and the latter situation is handled by the `answer_check_insert()` procedure as shown in Figure 6.

```
subgoal_check_insert(TABLE_ENTRY te, SUBGOAL_CALL call) {
  st_root_node = te->subgoal_trie_root_node
  if (GT_ROOT_NODE) {                                  // new table design
    leaf_gt_node = trie_check_insert(GT_ROOT_NODE, call)
    leaf_st_node = trie_node_check_insert(st_root_node, leaf_gt_node)
  } else {                                            // original table design
    leaf_st_node = trie_check_insert(st_root_node, call)
  }
  return leaf_st_node
}
```

Fig. 5. Pseudo-code for the `subgoal_check_insert()` procedure

In the original table design, the `subgoal_check_insert()` procedure simply uses the `trie_check_insert()` procedure to check/insert the given `call` in the subgoal trie corresponding to the given table entry `te`. In the new design based on the GT, the `subgoal_check_insert()` procedure now first checks/inserts the given `call` in the GT. Then, it uses the reference to the GT's leaf node representing `call` (`leaf_gt_node` in Figure 5) as the token to be checked/inserted in the subgoal trie corresponding to the given table entry `te`. Note that this is done by calling the `trie_node_check_insert()` procedure, thus if the list of sibling nodes in the subgoal trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, then a new hash table is initialized as described before.

```
answer_check_insert(SUBGOAL_FRAME sf, ANSWER answer) {
  at_root_node = sf->answer_trie_root_node
  if (GT_ROOT_NODE) {                                  // new table design
    leaf_gt_node = trie_check_insert(GT_ROOT_NODE, answer)
    leaf_at_node = trie_node_check_insert(at_root_node, leaf_gt_node)
  } else {                                            // original table design
    leaf_at_node = trie_check_insert(at_root_node, answer)
  }
  return leaf_at_node
}
```

Fig. 6. Pseudo-code for the `answer_check_insert()` procedure

The `answer_check_insert()` procedure works similarly. In the original table design, it checks/inserts the given `answer` in the answer trie corresponding to the given subgoal frame `sf`. In the new design based on the GT, it first checks/inserts the given `answer` in the GT and, then, it uses the reference to the GT's leaf node representing `answer` (`leaf_at_node` in Figure 6) as the token to be checked/inserted in the answer trie corresponding to the given sub-

goal frame `sf`. Again, if the list of sibling nodes in the answer trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, a new hash table is initialized.

Finally, the `answer_load()` procedure is used to consume answers. Figure 7 shows the pseudo-code for it. In the original table design, it simply uses the `trie_load()` procedure to load from the answer trie the answer given by the trie node `leaf_at_node`. In the new design based on the GT, the `answer_load()` procedure first accesses the GT's leaf node represented in the `token` field of the given trie node `leaf_at_node` (`leaf_gt_node` in Figure 7). Then, it uses the `trie_load()` procedure to load from the GT back to the Prolog engine the answer represented by the obtained GT's leaf node.

```
answer_load(ANSWER_TRIE_NODE leaf_at_node) {
  if (GT_ROOT_NODE) {                                  // new table design
    leaf_gt_node = leaf_at_node->token
    answer = trie_load(leaf_gt_node)
  } else {                                          // original table design
    answer = trie_load(leaf_at_node)
  }
  return answer
}
```

Fig. 7. Pseudo-code for the `answer_load()` procedure

5 Preliminary Experimental Results

We next present some preliminary experimental results comparing YapTab with and without support for the common global trie data structure. The environment for our experiments was an AMD Athlon XP 2800+ with 1 GByte of main memory and running the Linux kernel 2.6.24-19.

To evaluate the impact of our proposal, we have defined a tabled predicate `t/5` that simply stores in the table space terms defined by `term/1` facts, and then we used a top query goal `test/0` to recursively call `t/5` with all combinations of one and two free variables in the arguments. An example of such code for functor terms of arity 1 (500 terms in total) is shown next.

```
:- table t/5.
t(A,B,C,D,E) :- term(A), term(B), term(C), term(D), term(E).

test :- t(A,f(1),f(1),f(1),f(1)), fail.        term(f(1)).
...                                            term(f(2)).
test :- t(f(1),f(1),f(1),f(1),A), fail.        ...
test :- t(A,B,f(1),f(1),f(1)), fail.           term(f(499)).
...                                            term(f(500)).
test :- t(f(1),f(1),f(1),A,B), fail.
test.
```

We experimented the `test/0` predicate with 7 different kinds of 500 `term/1` facts: integers, atoms and functor terms of arity 1 to 5. Table 1 shows the memory

usage, in KBytes, and the running times, in milliseconds, to store to the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers for YapTab with (column ***YapTab+GT***) and without (column ***YapTab***) support for the common global trie data structure.

Terms	*YapTab (a)*			*YapTab+GT (b)*			*Ratio (b)/(a)*		
	Mem	*Store*	*Load*	*Mem*	*Store*	*Load*	*Mem*	*Store*	*Load*
500 int	49074	490	155	52803	738	164	1.08	1.51	1.06
500 atom	49074	508	158	52803	770	167	1.08	1.52	1.06
500 f/1	49172	693	242	52811	1029	243	1.07	1.48	1.00
500 f/2	98147	842	314	56725	1298	310	**0.58**	1.54	**0.99**
500 f/3	147122	1098	377	60640	1562	378	**0.41**	1.42	1.00
500 f/4	196097	1258	512	64554	1794	435	**0.33**	1.43	**0.85**
500 f/5	245072	1418	691	68469	2051	619	**0.28**	1.45	**0.90**

Table 1. Memory usage (in KBytes) and store/load times (in milliseconds) for YapTab with and without support for the common global trie data structure

The results show that GT support can reduce memory usage proportionally to the depth and redundancy of the terms stored in the GT. In particular, for functor terms of arity 2 to 5, the results show an increasing and very significant reduction on memory usage. The results for integer and atoms terms are also very interesting as they show that the cost of representing only atomic terms in the GT (between 7% and 8% in these experiments) can be manageable when we increase redundancy. Note that integers and atoms terms are represented by a single node in the original YapTab design, and by an extra node (therefore requiring two nodes) if using the GT approach.

On the other hand, these results seem to indicate that memory reduction comes at a price in execution time. With GT support, we need to navigate in two tries when checking/inserting a term. Moreover, in some situations, the cost of inserting a new term in an empty/small trie can be less than the cost of navigating in the GT, even when the term is already stored in the GT. However, our results seem to suggest that this cost decreases also proportionally to the depth and redundancy of the terms stored in the GT.

The results obtained for loading terms do not suggest significant differences. However and surprisingly, the GT approach showed to outperform the original YapTab design in some experiments.

6 Conclusions and Further Work

We have presented a new design for the table space organization that uses a common global trie to store terms in tabled subgoal calls and answers. Our goal is to reduce redundancy in term representation, thus saving memory by sharing data that is structurally equal. Our preliminary experiments showed very significant reductions on memory usage.

Further work will include exploring the impact of applying our proposal to real-world applications that pose many subgoal queries, possibly with a large number of redundant answers, such as ILP applications, seeking real-world experimental results allowing us to improve and expand our current implementation. In particular, we intend to study how alternative designs for the table space organization can further reduce redundancy in term representation.

Acknowledgements

This work has been partially supported by the research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/ EIA/66924/2006) and by Fundação para a Ciência e Tecnologia.

References

1. Michie, D.: Memo Functions and Machine Learning. Nature **218** (1968) 19–22
2. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
3. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
4. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (1999) 31–54
5. Rocha, R.: On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In: International Symposium on Practical Aspects of Declarative Languages. Number 4354 in LNCS, Springer-Verlag (2007) 155–169
6. Costa, P., Rocha, R., Ferreira, M.: Tabling Logic Programs in a Database. In: Workshop on (Constraint) Logic Programming. (2007) 125–135
7. Rao, P., Ramakrishnan, C.R., Ramakrishnan, I.V.: A Thread in Time Saves Tabling Time. In: Joint International Conference and Symposium on Logic Programming, The MIT Press (1996) 112–126
8. Johnson, E., Ramakrishnan, C.R., Ramakrishnan, I.V., Rao, P.: A Space Efficient Engine for Subsumption-Based Tabled Evaluation of Logic Programs. In: Fuji International Symposium on Functional and Logic Programming. Number 1722 in LNCS, Springer-Verlag (1999) 284–300
9. Goto, E.: Monocopy and Associative Algorithms in Extended Lisp. Technical Report TR 74-03, University of Tokyo (1974)
10. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
11. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming **5**(1 & 2) (2005) 161–205
12. Fredkin, E.: Trie Memory. Communications of the ACM **3** (1962) 490–499
13. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: International Joint Conference on Theory and Practice of Software Development. Number 668 in LNCS, Springer-Verlag (1993) 61–74

Efficient Evaluation of Deterministic Tabled Calls

Miguel Areias and Ricardo Rocha

DCC-FC & CRACS
University of Porto, Portugal
c0507028@alunos.dcc.fc.up.pt ricroc@dcc.fc.up.pt

Abstract. The execution model in which most tabling engines are based allocates a choice point whenever a new tabled subgoal is called. This happens even when the call is deterministic. However, some of the information from the choice point is never used when evaluating deterministic tabled calls with batched scheduling. Thus, if tabling is applied to a long deterministic computation, the system may end up consuming a huge amount of memory inadvertently. In this paper, we propose a solution that reduces this memory overhead to a minimum. Our results show that, for deterministic tabled calls with batched scheduling, it is possible not only to reduce the memory usage overhead, but also the running time of the evaluation.

Keywords: Tabling, Deterministic Calls, Implementation.

1 Introduction

Tabling [1, 2] is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. Implementations of tabling are now widely available in systems like XSB Prolog [3], Yap Prolog [4], B-Prolog [5], ALS-Prolog [6], Mercury [7] and more recently Ciao Prolog [8]. Actual implementations differ in the execution rule, in the data structures used to implement tabling, and in the changes to the underlying Prolog engine. Arguably, the SLG-WAM [9] is the most popular execution rule, but even here several issues require careful research, such as engine integration, execution data structures, termination detection, and scheduling support.

The increasing interest in tabling technology led to further developments and proposals that improve some practical deficiencies of current tabling execution models in key aspects of tabled evaluation like re-computation [10, 11], scheduling [12] and memory recovery [13]. The discussion we address in this work also results from practical deficiencies that we have found in the execution data structures used to evaluate deterministic tabled calls if applying batched scheduling [14].

The execution model in which most tabling engines are based allocates a choice point whenever a new tabled subgoal is called. This happens even when the call is deterministic, i.e., defined by a single matching clause. This is necessary

since the information from the choice point is crucial to correctly implement some tabling operations. However, some of this information is never used when evaluating deterministic tabled calls with batched scheduling. Thus, if tabling is applied to a long deterministic computation, the system may end up consuming a huge amount of memory inadvertently. In this paper, we propose a solution that reduces this memory overhead to a minimum. We will focus our discussion on a concrete implementation, the YapTab system [4], an efficient suspension-based tabling engine that extends the state-of-the-art Yap Prolog system [15] to support tabled evaluation for definite programs, but our proposal can be generalized and applied to other tabling engines.

The remainder of the paper is organized as follows. First, we briefly introduce the main background concepts about tabled evaluation. Next, we discuss in more detail how YapTab compiles and dynamically indexes deterministic tabled calls. We then describe how we have extended YapTab to provide engine support to efficiently deal with deterministic tabled calls. At last, we present some preliminary experimental results and we end by outlining some conclusions.

2 Basic Tabling Concepts

Tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears[1]. Whenever a tabled subgoal is first called, a new entry is allocated in an appropriated data space, the *table space*. Table entries are used to collect the answers found for their corresponding subgoals. Moreover, they are also used to verify whether calls to subgoals are repeated. Repeated calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all repeated calls. Within this model, the nodes in the search space are classified as either: *generator nodes*, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to repeated calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals.

The YapTab design follows the seminal SLG-WAM design [9]: it extends WAM's execution model [16] with a new data area, the *table space*; a new set of registers, the *freeze registers*; an extension of the standard trail, the *forward trail*; and four new operations for definite programs:

Tabled Subgoal Call: this operation is a call to a tabled subgoal. It checks if the subgoal is in the table space. If so, it allocates a consumer node and starts consuming the available answers. If not, it adds a new entry to the table space, and allocates a new generator node. When the call is deterministic, the tabled subgoal call operation is implemented by the `table_try_single` WAM-like instruction.

[1] We say that a subgoal repeats a previous subgoal if they are the same up to variable renaming.

New Answer: this operation verifies whether a newly found answer is already in the table, and if not, inserts the answer. Otherwise, the operation fails.

Answer Resolution: this operation verifies whether extra answers are available for a particular consumer node and, if so, consumes the next one. If no answers are available, it suspends the current computation and schedules a possible resolution to continue the execution.

Completion: this operation determines whether a tabled subgoal is completely evaluated. A subgoal is said to be complete when no more answers can be generated, that is, when its set of stored answers represent all the conclusions that can be inferred from the set of facts and rules in the program. If the subgoal has been completely evaluated, the operation closes the subgoal's table entry and reclaims stack space. Otherwise, control moves to a consumer with unconsumed answers.

During tabled evaluation, at several points, we can choose between continuing forward execution, backtracking to interior nodes, returning answers to consumer nodes, or performing completion. The decision on which operation to perform is determined by the *scheduling strategy*. Different strategies may have a significant impact on performance, and may lead to a different ordering of solutions to the query goal. Arguably, the two most successful tabling scheduling strategies are batched scheduling and local scheduling [14]. YabTab supports both batched scheduling, local scheduling and the dynamic intermixing of batched and local scheduling at the subgoal level [12]. Local scheduling does not have any relevance for this work, so we will not consider it.

Batched scheduling schedules the program clauses in a depth-first manner as does the WAM. It favors forward execution first, backtracking next, and consuming answers or completion last. It thus tries to delay the need to move around the search tree by batching the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues. For some situations, this results in creating dependencies to older subgoals, therefore enlarging the current SCC (*Strongly Connected Component*) and delaying the completion point to an older generator node. By default in YapTab, tabled predicates are evaluated using batched scheduling [12].

3 Deterministic Tabled Calls in YapTab

In this section we discuss how tabled predicates are compiled in YapTab and, in particular, we show how YapTab uses the Yap compiler to generate compiled and indexed code for deterministic tabled calls.

3.1 Compilation of Tabled Predicates

Tabled predicates defined by several clauses are compiled using the `table_try_me`, `table_retry_me` and `table_trust_me` WAM-like instructions in a similar manner to the generic `try_me`/`retry_me`/`trust_me` WAM sequence. The `table_try_me`

instruction extends the WAM's `try_me` instruction to support the tabled subgoal call operation. The `table_retry_me` and `table_trust_me` differ from the generic WAM instructions in that they restore a generator choice point rather than a standard WAM choice point. Tabled predicates defined by a single clause are compiled using the `table_try_single` WAM-like instruction. This instruction optimizes the `table_try_me` instruction for the case when the tabled predicate is defined by a single clause. Figure 1 shows the YapTab's compiled code for a tabled predicate `t/1` defined by a single clause and for a tabled predicate `t/3` defined by several clauses.

```
% predicate definitions
:- table t/1.
t(X) :- ...

:- table t/3.
t(a1,b1,c1) :- ...
t(a2,b2,c2) :- ...
t(a2,b1,c3) :- ...
t(a2,b3,c1) :- ...
t(a3,b1,c2) :- ...


% compiled code generated by YapTab for predicate t/1
t1_1:  table_try_single t1_1a
t1_1a: 'WAM code for clause t(X) :- ...'


% compiled code generated by YapTab for predicate t/3
t3_1:  table_try_me t3_2
t3_1a: 'WAM code for clause t(a1,b1,c1) :- ...'
t3_2:  table_retry_me t3_3
t3_2a: 'WAM code for clause t(a2,b2,c2) :- ...'
t3_3:  table_retry_me t3_4
t3_3a: 'WAM code for clause t(a2,b1,c3) :- ...'
t3_4:  table_retry_me t3_5
t3_4a: 'WAM code for clause t(a2,b3,c1) :- ...'
t3_5:  table_trust_me
t3_5a: 'WAM code for clause t(a3,b1,c2) :- ...'
```

Fig. 1. Compilation of tabled predicates in YapTab

As `t/1` is a deterministic tabled predicate, the `table_try_single` instruction will be executed for every call to this predicate. On the other hand, `t/3` is a non-deterministic tabled predicate, but some calls to this predicate can be deterministic, i.e., defined by a single matching clause. Consider, for example, the previous definition of `t/3` and the calls `t(a3,X,Y)` and `t(X,Y,c3)`. These two calls are deterministic as they only match with a single `t/3` clause, respectively, the 5th and 3rd clause. We next show how YapTab uses the demand-driven indexing mechanism of Yap to dynamically generate `table_try_single` instructions for this kind of deterministic calls.

3.2 Demand-Driven Indexing

Yap implements *demand-driven indexing* (or *just-in-time indexing*) [17] since version 5. The idea behind it is to generate flexible multi-argument indexing of Prolog clauses during program execution based on actual demand. This feature is implemented for static code, dynamic code and the internal database. All indexing code is generated on demand for all and only for the indices required. This is done by building an indexing tree using similar building blocks to the WAM but it generates indices based on the instantiation on the current goal, and expands indices given different instantiations for the same goal.

This powerful optimization provides that YapTab can execute calls to non-deterministic tabled predicates like deterministic tabled predicates. This happens when Yap's indexing scheme finds that for a particular call to a non-deterministic tabled predicate, there is only a single clause that matches the call. Figure 2 shows an example illustrating the indexed code generated for a non-deterministic call and two deterministic calls to the previous `t/3` tabled predicate.

```
% indexed code generated by YapTab for call t(a2,X,Y)
table_try   t3_2a
table_retry t3_3a
table_trust t3_4a

% indexed code generated by YapTab for call t(a3,X,Y)
table_try_single t3_5a

% indexed code generated by YapTab for call t(X,Y,c3)
table_try_single t3_3a
```

Fig. 2. Demand-driven indexing of tabled predicates in YapTab

The call `t(a2,X,Y)` is non-deterministic as it matches the 2nd, 3rd and 4th clauses of `t/3`, so a `table_try`/`table_retry`/`table_trust` sequence is generated. The other two calls, `t(a3,X,Y)` and `t(X,Y,c3)`, are both deterministic as they only match a single `t/3` clause, so a `table_try_single` instruction can be generated. Note however, that there are situations where a call can be deterministic, but Yap's indexing scheme cannot detect it as deterministic in order to generate the appropriate `table_try_single` instruction. In such cases, we cannot benefit directly from our approach, but we can take advantage of the similarities between the `table_try_single` instruction and the *last matching clause* of a non-deterministic tabled call to apply our approach later.

3.3 Last Matching Clause

When evaluating a tabled predicate, the last matching clause of a call to the predicate is implemented either by the `table_trust_me` instruction or by the `table_trust` instruction. The former situation occurs when we have a generic

call to the predicate (all the arguments of the call are unbound variables) and the latter situation occurs when we have a more specific call (some of the arguments are at least partially instantiated) optimized by indexing code.

In a WAM-based implementation [16], the last matching clause of a call is implemented by first restoring all the necessary information from the current choice point (usually pointed to by the WAM's `B` register) and then, by discarding the current choice point by updating `B` to its predecessor. In a tabled implementation, the `table_trust_me` and `table_trust` instructions also restore all the necessary information from the current choice point `B`, but instead of updating `B` to its predecessor, they update the next clause field of `B` to the `completion` instruction. By doing that, they force completion detection when the computation backtracks again to `B`, i.e., whether the clauses for the subgoal call at hand are all exploited.

Hence, the computation state that we have when executing a `table_trust_me` or `table_trust` instruction is similar to that one of a `table_try_single` instruction, that is, in both cases the current clause can be seen as deterministic as it is the last (or single) matching clause for the subgoal call at hand. Thus, we can view the `table_trust_me` and `table_trust` instructions as a special case of the `table_try_single` instruction. This means that the approach used for the `table_try_single` instruction to efficiently deal with deterministic tabled calls can be applied to the `table_trust_me` and `table_trust` instructions. We discuss the implementation details for these instructions in the next section.

4 Implementation Details

In this section, we describe in detail how we have extended YapTab to provide engine support to efficiently deal with deterministic tabled calls.

4.1 Generator Nodes

In YapTab, a generator node is implemented as a WAM choice point extended with some extra fields. The format of a generic generator choice point of YapTab is depicted in Figure 3. Fields that are not found in standard WAM choice points are coloured gray. A generator choice point is divided in three sections. The top section contains the usual WAM fields needed to restore the computation on backtracking plus two extra fields [12]: `cp_dep_fr` is a pointer to the corresponding dependency frame, used by local scheduling for fixpoint check, and `cp_sg_fr` is a pointer to the associated subgoal frame where answers should be stored. The middle section contains the argument registers of the subgoal and the bottom section contains the *substitution factor*, i.e., the set of free variables which exist in the terms in the argument registers. The substitution factor is an optimization that allows the new answer operation to store in the table space only the substitutions for the free variables in the subgoal call [18].

If we now turn our attention to how generator choice points are handled during evaluation, we find that some of this information is never used when evaluating deterministic tabled calls with batched scheduling. This happens mainly

cp_b	Failure continuation CP
cp_ap	Next unexploit alternative
cp_tr	Top of trail
cp_cp	Success continuation PC
cp_h	Top of global stack
cp_env	Current Environment
cp_dep_fr	Dependency frame
cp_sg_fr	Subgoal frame

An	Argument Register n
⋮	⋮
A1	Argument Register 1

m	Number of Substitution Vars
Vm	Substitution Variable m
⋮	⋮
V1	Substitution Variable 1

Fig. 3. Format of a generic generator choice point in YapTab

because, with batched scheduling, the computation is never resumed in a deterministic generator choice point. This allow us to remove the argument registers and the standard `cp_cp`, `cp_h` and `cp_env` fields. The `cp_dep_fr` field can also be removed because it is only necessary with local scheduling [12], which is never the case. Figure 4 shows the new format of YapTab's deterministic generator choice point with the strictly necessary fields.

The `cp_b` field is needed for failure continuation; the `cp_ap` and `cp_tr` are required when backtracking to the choice point; the `cp_sg_fr` is required by the new answer and completion operations; and the substitution factor fields are required by the new answer operation. In order to avoid extra overheads when manipulating the different kinds of choice points that can coexist in an evaluation, we have rearranged all kinds of choice points in such a way that the top three fields are now the same as the ones for a deterministic generator choice point: the `cp_b`, `cp_ap` and `cp_tr` fields.

The memory reduction obtained with the new representation for deterministic generator choice points increases when the number of argument registers (the arity of the predicate being called) and the number of substitution variables are, respectively, bigger and smaller. Considering that A is the number of arguments registers and that S is the number of substitution variables, the percentage of memory saved with the new representation can be expressed as follows:

cp_b	Failure continuation CP
cp_ap	Next unexploit alternative
cp_tr	Top of trail
cp_sg_fr	Subgoal frame

m	Number of Substitution Vars
Vm	Substitution Variable m
⋮	⋮
V1	Substitution Variable 1

Fig. 4. Format of a deterministic generator choice point in YapTab

$$1 - \frac{4 + 1 + S}{8 + A + 1 + S}$$

4.2 Tabling Operations

In order to deal with the new representation for deterministic generator choice points, this required small changes to the tabled subgoal call, new answer and completion operations. Figures 5, 6, 7 and 8 show in more detail the changes (blocks of code marked with comment '`// new`') made to the `table_try_single`, `table_trust_me`[2], `new_answer` and `completion` instructions. Figure 9 shows the pseudo-code for the auxiliary procedure `is_deterministic_generator_cp()`. We assume that memory addresses grow downwards and that the choice point stack grows upwards.

```
table_try_single(TABLED_CALL tc) {
  sg_fr = subgoal_check_insert(tc)   // sg_fr is the subgoal frame for tc
  if (new_tabled_subgoal_call(sg_fr)) {
    if (evaluation_mode(tc) == batched_scheduling)                // new
      store_deterministic_generator_node(sg_fr)
    else                                                 // local scheduling
      store_generic_generator_node(sg_fr)
    ...
    goto next_instruction()
  }
  ...
}
```

Fig. 5. Pseudo-code for the `table_try_single` instruction

[2] The changes made to the `table_trust` instruction are identical to the ones made to the `table_trust_me` instruction.

The table_try_single instruction now tests whenever the subgoal being called is to be evaluated using batched or local scheduling. If batched, it allocates a deterministic generator choice point. If local, it proceeds as before and allocates a generic generator choice point.

```
table_trust_me(TABLED_CALL tc) {
  // the B register points to the current choice point
  restore_generic_generator_node(B, COMPLETION)
  if (evaluation_mode(tc) == batched_scheduling &&
      not_in_a_frozen_segment(B) {                                  // new
    subs_factor = B + sizeof(generic_generator_cp) + arity(tc)
    gen_cp = subs_factor - sizeof(deterministic_generator_cp)
    gen_cp->cp_sg_fr = B->cp_sg_fr
    gen_cp->cp_tr = B->cp_tr
    gen_cp->cp_ap = B->cp_ap
    gen_cp->cp_b  = B->cp_b
    B = gen_cp
  }
  ...
}
```

Fig. 6. Pseudo-code for the table_trust_me instruction

The table_trust_me instruction now tests if the current tabled call is being evaluated using batched scheduling and if the current choice point is not in a frozen segment[3]. If these two conditions hold, we can recover some memory space by transforming the current generator choice point into a deterministic generator choice point. To do that, we need to copy the cp_sg_fr, cp_tr, cp_ap and cp_b fields in the current choice point to their new position, just above the substitution factor variables.

```
new_answer(TABLED_CALL tc, ANSWER ans) {
  if (is_deterministic_generator_cp(B)) {                           // new
    gen_cp = deterministic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
    subs_factor = gen_cp + sizeof(deterministic_generator_cp)
  } else {                              // generic generator choice point
    gen_cp = generic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
    subs_factor = gen_cp + sizeof(generic_generator_cp) + arity(tc)
  }
  ...
}
```

Fig. 7. Pseudo-code for the new_answer instruction

[3] The YapTab system uses frozen segments to protect the stacks of suspended computations [4]. Thus, if the current choice point is trapped in a frozen segment it is worthless to try to recover memory from it using our approach.

```
completion() {
  ...                                                  // fixpoint check loop
  // subgoal completely evaluated
  if (is_deterministic_generator_cp(B)) {                          // new
    gen_cp = deterministic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
  } else {
    gen_cp = generic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
  }
  complete_subgoal(sg_fr)
  ...
}
```

Fig. 8. Pseudo-code for the `completion` instruction

For the new answer and completion operations, since both generator types have different sizes, we need a way to correctly identify which is the type of the generator in order to correctly access the required fields on each structure. To do that, we use the `is_deterministic_generator_cp()` auxiliary procedure to test if a generator choice point is deterministic or not. Figure 9 shows the pseudo-code for it.

The `is_deterministic_generator_cp()` procedure assumes that, by default, we have a generic generator choice point and we check if the `cp_h` field (which is aligned with the field representing the number of substitution variables in a deterministic generator choice point) is less than the maximum number of allowed substitution variables (`MAX_SUBSTITUTION_VARS`). If this is case, then we know that we have a deterministic generator choice point.

```
is_deterministic_generator_cp(CHOICE_POINT cp) {
  gen_cp = generic_generator_cp(cp)
  if (gen_cp->cp_h <= MAX_SUBSTITUTION_VARS)
    return TRUE
  else
    return FALSE
}
```

Fig. 9. Pseudo-code for the `is_deterministic_generator_cp()` procedure

5 Preliminary Experimental Results

We next present some preliminary experimental results comparing YapTab with and without support for deterministic tabled calls. The environment for our experiments was a AMD Athlon(tm) 64 Processor 3200+ processor with 2 GByte of main memory and running the Linux kernel 2.6.24-19 with YapTab 5.1.3.

To evaluate the impact of our proposal, first we have defined three deterministic tabled predicates, respectively with arities 5, 11 and 17, that simply call themselves recursively:

```
:- table t/5, t/11, t/17.

t(N,A2,A3,A4,A5) :-
  N > 0, N1 is N - 1,
  t(N1,A2,A3,A4,A5).

t(N,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11) :-
  N > 0, N1 is N - 1,
  t(N1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11).

t(N,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17) :-
  N > 0, N1 is N - 1,
  t(N1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17).
```

The first argument `N` controls the number of times the predicate is executed. It thus defines the number of generator choice points to be allocated (we used a value of 100,000 in our experiments). In order to have specific combinations of argument registers and substitution variables, we have ran each predicate with three different sets of free variables in the arguments:

```
:- t(100000,A2,A3,A4,A5).
:- t(100000,A2,A3,0,0).
:- t(100000,0,0,0,0).

:- t(100000,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11).
:- t(100000,A2,A3,A4,A5,A6,0,0,0,0,0).
:- t(100000,0,0,0,0,0,0,0,0,0,0).

:- t(100000,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17).
:- t(100000,A2,A3,A4,A5,A6,A7,A8,A9,0,0,0,0,0,0,0,0).
:- t(100000,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0).
```

These experiments are a kind of best-case scenario as they only allocate generator choice points and they do not store permanent variables for *environment frames* [16]. Table 1 shows the memory usage, in KBytes, for the local stack[4] and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls. A third column ***Ratio (1–b/a)*** shows the memory and running time ratio between both approaches. For the memory ratio, we show in parentheses the percentage of memory saved if using the formula presented at the end of section 4.1.

The results in Table 1 indicate that YapTab with support for deterministic tabled calls can decrease, on average, memory usage by 48% and running time by 23%. These results also confirm that memory reduction increases when the number of argument registers is bigger and the number of substitution variables is smaller. This is coherent with the formula presented in section 4.1. The small difference between our experiments and the values obtained when using the formula came from the fact that, in the formula, we are considering a local stack without environment frames.

[4] In YapTab, the local stack contains both choice points and environment frames. Other systems, like XSB Prolog, have separate choice point and environment stacks.

Args	***Subs***	***YapTab (a)***		***YapTab+Det (b)***		***Ratio (1–b/a)***	
		Memory	***Time***	***Memory***	***Time***	***Memory***	***Time***
5	4	9,376	82	5,860	70	0.37 (0.50)	0.15
5	2	8,594	78	5,079	66	0.41 (0.57)	0.15
5	0	7,813	80	4,297	65	0.45 (0.64)	0.19
11	10	14,063	137	8,204	96	0.42 (0.50)	0.30
11	5	12,110	136	6,251	89	0.48 (0.60)	0.35
11	0	10,157	124	4,297	108	0.58 (0.75)	0.13
17	16	18,751	173	10,547	129	0.44 (0.50)	0.25
17	8	15,626	164	7,422	109	0.53 (0.62)	0.34
17	0	12,501	153	4,297	114	0.66 (0.81)	0.25
Average						0.48 (0.61)	0.23

Table 1. Memory usage (in KBytes) and running times (in milliseconds) for YapTab without and with the new support for deterministic tabled calls

Next, we tested our approach with the *sequence comparisons* problem [19]. In this problem, we have two sequences A and B, and we want to determine the minimal number of operations needed to turn A into B. We used the original tabled program from [19] and a transformed tabled program that forces all calls to use the `table_try_single` instruction. We experimented these two versions with sequences of length 500, 1000, 1500 and 2000. Table 2 shows the memory usage, in KBytes, for the local stack and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls. A third column ***Ratio (1–b/a)*** shows the memory and running time ratio between both approaches.

Version	***Length***	***YapTab (a)***		***YapTab+Det (b)***		***Ratio (1–b/a)***	
		Memory	***Time***	***Memory***	***Time***	***Memory***	***Time***
Original	500	51,774	1,548	44,938	1,264	0.13	0.18
	1000	207,063	13,548	179,719	11,212	0.13	0.17
	1500	465,868	60,475	404,344	50,631	0.13	0.16
	2000	828,188	189,647	718,813	157,213	0.13	0.17
Transformed	500	45,915	1,172	39,051	848	0.15	0.28
	1000	183,625	10,024	156,227	8,460	0.15	0.16
	1500	413,133	45,874	351,528	36,106	0.15	0.21
	2000	734,438	140,068	624,953	113,011	0.15	0.19
Average						0.14	0.19

Table 2. Memory usage (in KBytes) and running times (in milliseconds) for YapTab without and with the new support for deterministic tabled calls

In general, for memory usage, the results in Table 2 are slightly different from the previous results obtained in Table 1. For both version of the *sequence comparisons* program, YapTab with support for deterministic tabled calls can

decrease, on average, memory usage by 14%. This reduction on memory saving, compared with the results on Table 1, happens mainly because of the existence of permanent variables in the body of the clauses in the *sequence comparisons* program. On the other hand, for the running times, the results in Table 2 confirm the previous results obtained in Table 1.

The results in Table 2 also show very similar memory and running time ratios for both versions of the *sequence comparisons* program. This suggests that we can take advantage of our approach by using the last matching clause optimization and not only when a program contains deterministic tabled predicates.

Finally, we tested our approach with a *path* program that computes the transitive closure of a NxN grid using a right recursive algorithm:

```
:- table path/2.

path(X,Z) :- edge(X,Z).
path(X,Z) :- edge(X,Y), path(Y,Z).
```

Regarding the `edge/2` facts, we used four grid configuration with 30x30, 40x40, 50x50 and 60x60 nodes. Table 3 shows the memory usage, in KBytes, for the local stack and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls. Again, a third column ***Ratio (1–b/a)*** shows the memory and running time ratio between both approaches.

Grid	***YapTab (a)***		***YapTab+Det (b)***		***Ratio (1–b/a)***	
	Memory	***Time***	***Memory***	***Time***	***Memory***	***Time***
30x30	119	1,304	98	1,464	0.18	-0.12
40x40	211	4,400	175	4,024	0.17	0.09
50x50	330	11,208	273	10,996	0.17	0.02
60x60	476	28,509	393	28,213	0.17	0.01
Average					0.17	0.00

Table 3. Memory usage (in KBytes) and running times (in milliseconds) for YapTab without and with the new support for deterministic tabled calls

The *path* program confirms tendency to memory reduction, this case in 17%, on average. Running time gets sightly worse, thought comparison between both approaches remains in positive territory in three cases. Note however, that our approach was mainly designed to achieve a reduction on memory usage by paying a small cost on running time due to the extra code needed to deal with the new data structures and algorithms. Despite of this fact, on average, our approach showed a very good performance in all experiments.

6 Conclusions and Further Work

We have presented a proposal for the efficient evaluation of deterministic tabled calls with batched scheduling. A well-known aspect of tabling is the overhead in terms of memory usage compared with standard Prolog. This raised us the question of whether it was possible to minimize this overhead when evaluating deterministic tabled computations. Our preliminary results are quite promising, they suggest that, for deterministic tabled calls with batched scheduling, it is possible not only to reduce the memory usage overhead, but also the running time of the evaluation for certain class of applications.

Further work will include exploring the impact of applying our proposal to more complex problems, seeking real-world experimental results allowing us to improve and expand our current implementation.

Acknowledgements

This work has been partially supported by the research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/ EIA/66924/2006) and by Fundação para a Ciência e Tecnologia.

References

1. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
2. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
3. Rao, P., Sagonas, K., Swift, T., Warren, D.S., Freire, J.: XSB: A System for Efficiently Computing Well-Founded Semantics. In: International Conference on Logic Programming and Non-Monotonic Reasoning. Number 1265 in LNCS, Springer-Verlag (1997) 431–441
4. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming **5**(1 & 2) (2005) 161–205
5. Zhou, N.F., Sato, T., Shen, Y.D.: Linear Tabling Strategies and Optimizations. Theory and Practice of Logic Programming **8**(1) (2008) 81–109
6. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 181–196
7. Somogyi, Z., Sagonas, K.: Tabling in Mercury: Design and Implementation. In: International Symposium on Practical Aspects of Declarative Languages. Number 3819 in LNCS, Springer-Verlag (2006) 150–167
8. Chico, P., Carro, M., Hermenegildo, M.V., Silva, C., Rocha, R.: An Improved Continuation Call-Based Implementation of Tabling. In: International Symposium on Practical Aspects of Declarative Languages. Number 4902 in LNCS, Springer-Verlag (2008) 197–213

9. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20**(3) (1998) 586–634
10. Sagonas, K., Stuckey, P.: Just Enough Tabling. In: ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM Press (2004) 78–89
11. Saha, D., Ramakrishnan, C.R.: Incremental Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 235–249
12. Rocha, R., Silva, F., Santos Costa, V.: Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 250–264
13. Rocha, R.: On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In: International Symposium on Practical Aspects of Declarative Languages. Number 4354 in LNCS, Springer-Verlag (2007) 155–169
14. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: International Symposium on Programming Language Implementation and Logic Programming. Number 1140 in LNCS, Springer-Verlag (1996) 243–258
15. Santos Costa, V., Damas, L., Reis, R., Azevedo, R.: YAP User's Manual. Available from `http://www.dcc.fc.up.pt/~vsc/Yap`.
16. Aït-Kaci, H.: Warren's Abstract Machine – A Tutorial Reconstruction. The MIT Press (1991)
17. Santos Costa, V., Sagonas, K., Lopes, R.: Demand-Driven Indexing of Prolog Clauses. In: International Conference on Logic Programming. Number 4670 in LNCS, Springer-Verlag (2007) 395–409
18. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (1999) 31–54
19. Warren, D.S.: Programming in Tabled Prolog. Technical report, Department of Computer Science, State University of New York (1999) Available from `http://www.cs.sunysb.edu/~warren/xsbbook/book.html`

A Program Transformation for Continuation Call-Based Tabled Execution

Pablo Chico de Guzman[1] Manuel Carro[1] Manuel V. Hermenegildo[1,2]
pchico@clip.dia.fi.upm.es {mcarro,herme}@fi.upm.es

[1] School of Computer Science, Univ. Politecnica de Madrid, Spain
[2] IMDEA Software, Spain

Abstract. The advantages of tabled evaluation regarding program termination and reduction of complexity are well known —as are the significant implementation, portability, and maintenance efforts that some proposals (especially those based on suspension) require. This implementation effort is reduced by program transformation-based continuation call techniques, at some efficiency cost. However, the traditional formulation of this proposal by Ramesh and Cheng limits the interleaving of tabled and non-tabled predicates and thus cannot be used as-is for arbitrary programs. In this paper we present a complete translation for the continuation call technique which, using the runtime support needed for the traditional proposal, solves these problems and makes it possible to execute arbitrary tabled programs. We present performance results which show that CCall offers a useful tradeoff that can be competitive with state-of-the-art implementations.
Keywords: Tabled logic programming, Continuation-call tabling, Implementation, Performance, Program transformation.

1 Introduction

Tabling [18, 19, 4] is a strategy for executing logic programs which uses *memoization* of already processed calls and their answers to improve several of the limitations of SLD resolution. It brings termination for bounded term-size programs and improves efficiency in programs which perform repeated computations and has been successfully applied to deductive databases [14], program analysis [20, 5], reasoning in the semantic Web [23], model checking [13], etc.

However, tabling also has certain drawbacks, including that predicates to be tabled have to be selected carefully[3] in order not to incur in undesired slowdowns and, specially relevant to our discussion, that its efficient implementation is generally complex. In *suspension-based tabling* the computation state of suspended tabled subgoals has to be preserved to avoid backtracking over them. This is done either by *freezing* the stacks, as in XSB [17], by copying to another

[3] XSB includes an `auto_table` declaration which triggers a conservative analysis to detect which predicates are to be tabled in order to ensure termination. However, more predicates than needed can be selected.

area, as in CAT [8], or by using an intermediate solution as in CHAT [9]. *Linear tabling* maintains instead a single execution tree without requiring suspension and resumption of sub-computations. The computation of the (local) fixpoint is performed by making subgoals "loop" in their alternatives until no more solutions are found. This may make some computations to be repeated. Examples of this method are the linear tabling of B-Prolog [22, 21] and the DRA scheme [10]. Suspension-based mechanisms achieve very good performance but, in general, require deeper changes to the underlying implementation. Linear mechanisms, on the other hand, can usually be implemented on top of existing sequential engines without major modifications.

The Continuation Call (`CCall`) approach to tabling [15, 16] tries to combine the best of both worlds: it is a reasonably efficient suspension-based mechanism which requires relatively simple additions to the Prolog implementation / compiler,[4] thus making maintenance and porting much easier. In [6] we proposed a number of optimizations to the `CCall` approach and showed that with such optimizations performance could be competitive with traditional implementations. However, this was only partially satisfactory since the `CCall` tabling approach is restricted to programs with a certain interleaving of tabled and non-tabled predicate calls (see Figure 3 and Section 3.1), and thus cannot execute general tabled programs.

In this paper we present an extension of the `CCall` translation which, using the same runtime support of the traditional proposal, overcomes the problems pointed out above. In Section 5 we present a complexity comparison of the proposed approach with CHAT. Finally, we present performance results from our implementation. These results show that our approach offers a useful tradeoff which can be competitive with state of the art implementations, while keeping implementation efforts relatively low.

2 The Continuation Call Technique

We sketch now how tabled evaluation [4, 17] works from a user point of view and we briefly describe the Continuation Call technique, on which we base our work.

2.1 Tabling Basics

We will use as example the program in Figure 1, whose purpose is to determine the reachability of nodes in a graph. If the graph contains cycles, there will be queries which will make the program loop forever under the standard SLD resolution strategy, regardless of the order of the clauses. Tabling changes the operational semantics for predicates marked with the `:- table` declaration, which forces the compiler and runtime system to distinguish the first occurrence of a tabled goal (the *generator*) and subsequent calls which are identical up to variable renaming (the *consumers*). The generator applies resolution using the

[4] As an example, no modification to the underlying engine is needed.

program clauses to derive answers for the goal. Consumers *suspend* the current execution path (using implementation-dependent means) and start execution on a different branch corresponding to another clause of the predicate within which the execution was suspended. When such an alternative branch finally succeeds, the answer generated for the initial query (the generator) is inserted in a table associated with that generator. This makes it possible to reactivate consumers and to continue execution at the point where they were stopped. Thus, consumers do not use SLD resolution, but obtain instead the answers from the table where they were previously inserted by the generator. Predicates not marked as tabled are executed according to SLD resolution, hopefully with minimal overhead due to the availability of tabling. This can be graphically seen as the ability to suspend execution in a part of the tree which cannot progress (because it enters a loop) and continue it somewhere else, where a solution for the looping goal can be produced.

2.2 `CCall` by Example

`CCall` implements tabling by a combination of program transformation and side effects in the form of insertions into and retrievals from a table which relates calls, answers, and the continuation code to be executed after consumers read answers from the table. We will now sketch how the mechanism works using the `path/2` example (Figure 1). The original code is transformed into the program in Figure 2 which is the one actually executed.

Roughly speaking, the transformation for tabling is as follows: an auxiliary predicate (`slg_path/2`) for `path/2` is introduced so that calls to `path/2` made from regular (SLD) Prolog execution do not need to be aware of the fact that `path/2` is being tabled. The primitive `slg/1` will make sure that its argument is executed to completion and will return, on backtracking, all the solutions found for the tabled predicate. To this end, `slg/1` checks if the call has already been executed. If so, all its answers are returned by backtracking. Otherwise, control is passed to a new predicate (`slg_path/2` in this case).[5] `slg_path/2` receives in its first argument the original call to `path/2` and in the second argument the identifier of its generator, which is used to relate operations on the table with this initial call. Each clause of `slg_path/2` is derived from a clause of the original `path/2` predicate by:

- Adding an `answer/2` primitive at the end of each clause of the original tabled predicate. `answer/2` is responsible for inserting answers in the table after checking for redundancy.
- Instrumenting calls to tabled predicates using the `slgcall/1` primitive. If this tabled call is a consumer, `path_cont/3`, along with its arguments, is recorded as (one of) the continuation(s) of its generator. If the tabled call is a generator, it is associated with a new call identifier and execution follows using the `slg_path/2` program clauses to derive new answers (as done

[5] The unique name has been created for simplicity by prepending `slg_` to the predicate name –any safe means of constructing a unique predicate symbol can be used.

```
:- table path/2.

path(X, Z):-
    edge(X, Y),
    path(Y, Z).
path(X, Z):-
    edge(X, Z).
```

Fig. 1. A sample program.

```
path(X, Y):- slg(path(X, Y)).
slg_path(path(X, Y), Id):-
    edge(X, Y),
     slgcall (path_cont(Id, [X], path(Y, Z))).
slg_path(path(X, Y), Id):-
    edge(X, Y),
    answer(Id, path(X, Y)).

path_cont(Id, [X], path(Y, Z)):-
    answer(Id, path(X, Z)).
```

Fig. 2. The program in Figure 1 after being transformed for tabled execution.

by `slg/1`). Besides, `path_cont/3` will be recorded as a continuation of the generator identified by `Id` if the tabled call cannot be completed (there were dependencies on previous generators). The `path_cont/3` continuation will be called consuming found answers or erased upon completion of its generator.
– Encoding the remaining of the clause body of `path/2` after the recursive call by using `path_cont/3`. It is constructed similarly to `slg_path/2`, i.e., applying the same transformation as for the initial clauses and calling `slgcall/1`.

The second argument of `path_cont/3` is a list of bindings needed to recover the environment of the continuation call. Note that, in the program in Figure 1, an answer to a query such as `?- path(X, Y)` may need to bind variable `X`. This variable does not appear in the recursive call to `path/2`, and hence it does not appear in the `path/2` term passed on to `slgcall/1` either. In order for the body of `path_cont/3` to insert in the table the answer corresponding to the initial query, variable `X` (and, in general, any other necessary variable) has to be passed down to `answer/2`. This is done with the list `[X]`, which is inserted in the table as well and completes the environment needed for the continuation `path_cont/3` to resume the previously suspended call.

A safe approximation of the variables which should appear in this list is the set of variables which appear in the clause before the tabled goal and which are used in the continuation, including the `answer/2` primitive. Variables appearing in the tabled call itself do not need to be included, as they will be passed along anyway. This list of bindings corresponds to the frame of the parent call if the `answer/2` primitive is added to the end of the body being translated. More details about `CCall` approach and their primitives can be found at [15].

Key Contribution of CCall: a new predicate name is created for all points where suspension can happen. Suspension is performed by saving this predicate name, a list of bindings, and a generator identifier. Resumption is performed by constructing a Prolog goal with the information saved on suspension plus the answer which raised the resumption. It is clear that this is significantly simpler

```
:- table t/1.

t(A):-
    p(B),
    A is B + 1.
t(0).

p(B):- t(B), B < 1.
```

Fig. 3. A program for which the original `CCall` transformation fails.

```
t(A):- slg(t(A)).
slg_t(t(A), Id):-
    p(B), A is B + 1,
    answer(Id, t(A)).

slg_t(t(0), Id):-
    answer(Id, t(0)).

p(B):- t(B), B < 1.
```

Fig. 4. The program in Figure 3 after being transformed for tabled execution.

to implement than other approaches as XSB or CHAT, where changes in the abstract machine have to be introduced. Consequently, porting and maintainability are simpler too, since `CCall` is independent of the compiler and how to create a Prolog term on the heap is the only one low level operation to implement.

3 Mixing Tabled and Non-Tabled Predicates

A continuation is the way `CCall` tabling preserves both the environment and the code of a consumer to be resumed. The list of bindings contains the same variables as the frame of the predicate where the `slgcall/1` primitive is executed, taking into account the `answer/2` primitive added at the end of the clause. However, the `CCall` approach to tabling, as originally proposed, has a problem when Prolog predicates appear between generators and consumers: the environments created by the non-tabled predicates are not taken into account, and they may be needed to correctly suspend and resume tabled predicates, as the example in the following section shows.

3.1 An Ill-Behaved Transformation

Figure 3 shows an example of a tabled program, where tabled and non-tabled execution (`t/1` and `p/1`) are mixed. The translation of the program is shown in Figure 4, taking into account the rules in Section 2.2.

The execution of the program with the query `t(A)` is shown in Figure 5. The execution is correct until `slg/1` is called again by `p/1`. At that point execution should suspend (and later resume), but `slg/1` does not have any associated continuation, and it does not have any pointer to the code to be executed on resumption (partially in `p/1` and partially in `slg_t/2`): `B < 1, A is B + 1, answer(Id,t(A))` is lost on backtracking and it is not reachable when resuming. Consequently, the second answer to the query, `t(1)`, is lost.

The call to `t(B)` made by `p(B)` could have been translated as if it were in the body of a tabled clause, but in that case the piece of code `A is B + 1` in

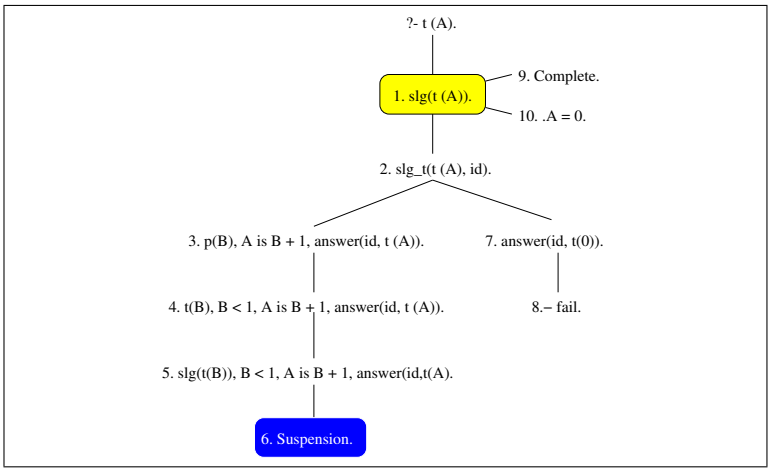


Fig. 5. Tabling execution of example of Figure 1.

the first clause of `t/1` would be lost anyway. This is an example of why all the frames between a consumer and its nearest generator have to be saved when suspending, and it is not enough to save just the last one, as in the original `CCall` proposal [15], which does work, however, when all the calls to the tabled predicates appear in the body of the clause of a tabled predicate. In that case, it is enough to save the last frame with the associated continuation code. Note that all the suspension-based tabling approaches preserve the frames / environments from the consumer until the corresponding generator.

To solve this problem, we have extended the translation to take into account a new kind of predicates, named *bridges*. A bridge predicate is a non-tabled Prolog predicate whose clauses generate frames which have to be saved in the continuation of a consumer. In the example of Figure 3, `p/1` is a bridge predicate.

3.2 Marking Predicates as Bridges

Bridge predicates are all the non-tabled predicates which can appear in the execution tree of a query between a generator and each of its consumers, i.e., the predicates whose environments are in the local stack between the environment of the generator and the environment of each of its consumers. Note that tabled predicates do not need to be included as bridge predicates as their environment will be already saved by the translation. Additionally, only recursive calls which can lead to infinite loops under SLD resolution have to actually be taken into account, because these are the only ones which can suspend and later be resumed. Programs for which tabling merely speeds up already terminating computations are not subject to the problem outlined above, and therefore do not benefit from the improved translation shown herein.

Thus, in order to determine a minimal set of bridge predicates, B_{min}, we need to determine before the minimum set of tabled predicates, T_{min}, which ensures

Make a graph G with an edge $(p1/n1,\ p2/n2) \Leftrightarrow p2/n2$ **is** called from $p1/n1$
$Bridges = \emptyset$
FOR each predicate T in TABLED PREDICATES
  $Forward =$ All predicates reached from T in G
  $Backward =$ All predicates from which T **is** reached in G
  $Bridges = Bridges \cup (Forward \cap Backward)$
$Bridges = Bridges -$ TABLED PREDICATES

Fig. 6. Safe approximation to look for bridge predicates.

termination. When T_{min} is found, B_{min} is the set of non-tabled predicates which are "in the middle" of two calls to predicates belonging to T_{min}. Since looking for T_{min} is undecidable (because it implies detecting infinite failures), looking for B_{min} is also undecidable and a *safe approximation*, which may mark as bridge some predicates which do not need to be, is needed.

As we will see in Section 4.2, the only disadvantage of such an over-approximation is that some code will be duplicated (to accept a new argument for the case where a bridge predicate is called from a tabled execution), and that bridge predicates, having an extra argument, can be called when this is not needed. The algorithm we have implemented (Figure 6) only looks for tabled predicates which can recursively call themselves. For the examples used for performance evaluation in Section 6, using the safe approximation algorithm produces an average slowdown of only 3% with respect to a perfect characterization of bridge predicates.

4 A General Translation for Tabled Programs

In this section we present program transformation rules which take into account bridge predicates. This transformation assumes that the safe approximation algorithm for bridge predicates has already been run, and all the bridge predicates have been marked by adding a `:- bridge P/N` declaration in the program.

As seen in Section 2.2, a continuation is the way to save an environment, because the predicate name is the same as the PC counter of the environment and the list of bindings is the same as the variables that a environment saves. Consequently, the goal of the new translation is to associate a continuation with each of the bridge predicates to save their associated environment. These continuations receive a new argument (the continuation to be executed) which is used to push a pointer (i.e., the name of a predicate) to the code to continue with, in a way similar to environments in local stacks.

4.1 Translation Rules

The rules for the original translation have three different goals: to maintain the interface with the rest of the code, to manage tabled calls which appear in the body of the clauses of a tabled predicate, and to insert answers at the end of

```
trans(C, C) :- \+ table(C), \+ bridge(C).
trans(( :- table P/N ), ( P(X1..Xn) :- slg(P(X1..Xn)) )).
trans(( Head :- Body ), LC) :-
    table(Head),
    Head_tr =.. ['slg_' ∘ Head, Head, Id],
    End = answer(Id, Head),
    transBody(Head_tr, Body, Id, [], End, LC).
trans(( Head :- Body ), ( Head :- Body ) ∘ LC) :-
    bridge(Head),
    Head_tr =.. [Head ∘ '_bridge', Head, Id, Cont],
    End = call(Cont),
    transBody(Head_tr, Body, Id, Cont, End, LC).

transBody([], [], _, _, [], []).
transBody(Head, Body, Id, ContPrev, End, ( Head :- Body_tr ) ∘ RestBody_tr) :-
    following(Body, Pref, Pred, Suff),
    getLBinds(Pref, Suff, LBinds),
    updateBody(Pred, End, Id, Pref, LBinds, ContPrev, Cont, Body_tr),
    transBody(Cont, Suff, Id, ContPrev, End, RestBody_tr).

following(Body, Pref, Pred, Suff) :-
    member(Body, Pred),
    (table(Pred); bridge(Pred)), !,
    Body = Pref ∘ Pred ∘ Suff.

updateBody([], End, _Id, Pref, _LBinds, _ContPrev, [], Pref ∘ End).
updateBody(Pred, _End, Id, Pref, LBinds, ContPrev, Cont, Pref ∘ slgcall(Cont)) :-
    table(Pred),
    getNameCont(NameCont),
    Cont = NameCont(Id, LBinds, Pred, ContPrev).
updateBody(Pred, _End, Id, Pref, LBinds, ContPrev, Cont, Pref ∘ Bridge_call) :-
    bridge(Pred),
    getNameCont(NameCont),
    Cont = NameCont(Id, LBinds, Pred, ContPrev),
    Bridge_call =.. [Pred ∘ '_bridge', Pred, Id, Cont] .
```

Fig. 7. The Prolog code of the translation rules.

the evaluation of each clause. The same points have to be addressed for bridge clauses, taking into account that a tabled or bridge call has to be translated if it appears in the body of a tabled predicate or a bridge predicate.

The rules for the new translation, which uses the same primitives as the original `CCall` proposal, are shown in Figure 7, where for conciseness we have used a sugared Prolog-like language. For example, a functional syntax is implicitly assumed where needed, and infix '∘' is a general `append` function which joins either (linear) structures or, when applied to atoms, concatenates them. It may appear in an output head position with the expected semantics.

The `trans/2` predicate receives a clause to be translated and returns the list of clauses resulting from the translation. Its first clause ensures that predicates which are non-tabled and non-bridge are not transformed.[6] The second one is to generate the interface of table predicates with the rest of the code: if there is a tabled declaration, the interface is generated. The third clause translates clauses of tabled predicates, and the fourth one translates clauses of bridge predicates, where the original one is maintained in case it is called outside a tabled call (this is in order to preserve the interface with non-tabled code). They generate the new head of the clause, `Head_tr`, and the code which has to be appended at the end of the body, `End`, before calling `transBody/6` with these arguments. `End` can be the `answers/2` primitive for tabled clauses or `call(Cont)`, which invokes the following pushed continuation, stored in the fourth argument.

`transBody/6` generates, in its last argument, the translation of the body of a clause by taking care, in each iteration, of the code until the next tabled or bridge call, or until the end the clause, and appending the translation of the rest of the clause to this partial translation. In other words, it calls `updateBody/8` to translate tabled or bridge calls and continues translating the rest of the body.

The `following/4` splits a clause body in three parts: a prefix, until the first time a tabled or bridge call appears, the tabled or bridge call itself, and a suffix from this call until the end of the clause. `getLBinds/3` obtains the list of variables which have to be saved to recover the environment of the consumer, based on the ideas of Section 2.2.

The `updateBody/8` predicate completes the body prefix until the next tabled or bridge call. Its first six arguments are inputs, the seventh one is the head of the continuation for the suffix of the body, and the last argument is the new translation for the prefix. The first clause takes care of the base case, when there are no calls to bridge or tabled predicates left, the second clause generates code for a call to a tabled predicate, and the last one does the same with a bridge predicate. That `getNameCont/1` generates a unique name for the continuation.

We will now use the example in Figure 3, adding a `:- bridge p/1` declaration, to exemplify how a translation would take place.

4.2 The Previous Example with the Correct Transformation

The translation of the first clause of `t/1` is done by the third clause of `trans/2`, which makes the head of the translated clause to be `slg_t(t(A), Id)` and states that the final call of that clause has to be `answer(Id, t(A))` —i.e., when the clause successfully finishes, it adds the answer to the table.

`transBody/6` takes care then of the rest of the body, which identifies which environment variables (`A`, in this case) have to be saved and matches `Pref`, `Pred`, and `Suff` with the goals before the call to the bridge predicate (none —

[6] The predicates `table/1` and `bridge/1` are dynamically generated by the compiler from the corresponding declaration. They check if their argument is a clause of a tabled or bridge predicate, or if their argument is a functor corresponding to a tabled or bridge predicate, respectively.

```
t(A) :- slg(t(A)).
slg_t(t(A), Id) :-
    p_bridge(p(B), Id, slg_t0(Id, [A], p(B), [])).

slg_t(t(0), Id) :- answer(Id, t(0)).

slg_t0(Id, [A], p(B), []) :-
    A is B + 1,
    answer(Id, t(A)).

p(B) :- t(B), B < 1.
p_bridge(p(B), Id, Cont) :-
    slgcall(p_bridge0(Id, [], t(B), Cont)).

p_bridge0(Id, [], t(B), Cont) :-
    B < 1,
    call(Cont).
```

Fig. 8. The program in Figure 3 after being transformed for tabled execution.

and empty conjunction), the call to the bridge predicate (`p(B)`), and the goals after this call (`A is B + 1`). The third clause of `updateBody/8` generates the body of `Head_tr`, to give the first clause of `slg_t/2`. A continuation is generated for the rest of the body; the code of the continuation is a predicate whose head is `slg_t0/3` and its body is generated by the first clause of `updateBody/8`.

The translation of the second clause of `t/1` is simpler, as it only has to add `answer(Id, t(0))` at the end of the body of the new predicate.

The clause for `p/1` is kept to maintain its interface when it is not called from inside a another tabled execution. The translation for the clause of `p/1` is made by the fourth clause of `trans/2` where `Head_tr` is unified with `p_bridge(p(B), Id, Cont)`. `End` is unified with `call(Cont)` — a call to the continuation code to be resumed by the following pushed continuation. `transBody/6` finds an empty list of environment variables and unifies `Pref`, `Pred` and `Suff` with `[]`, `t(B)` and `B < 1`, respectively. The second clause of `updateBody/8` generates the body for the new predicate `p_bridge/3`. A continuation is generated to execute the rest of the body, whose head is `p_bridge0/3` and whose body is generated by the first clause of `updateBody/8`. As we can see, bridge predicates are pushing continuations which are sequentially called when consumers are resumed.

4.3 Execution of the Transformed Program

The execution tree of the transformed program is shown in Figure 9. It is similar to that in Figure 5, but a continuation `slg_t0(id, [A], p(B), [])` is passed to the transformed clause of `p/1`. This continuation contains the code to be executed after the execution of `p(B)` and the list `[A]` needed to recover its environment. Consequently, there are two continuations associated with the suspension: one continuation to execute the rest of the code of `p(B)` and another one to execute the rest of the code of `t(A)`.

After the first answer is found, this double continuation is resumed. It is executed as a normal Prolog and the second answer, `t(1)`, is found.

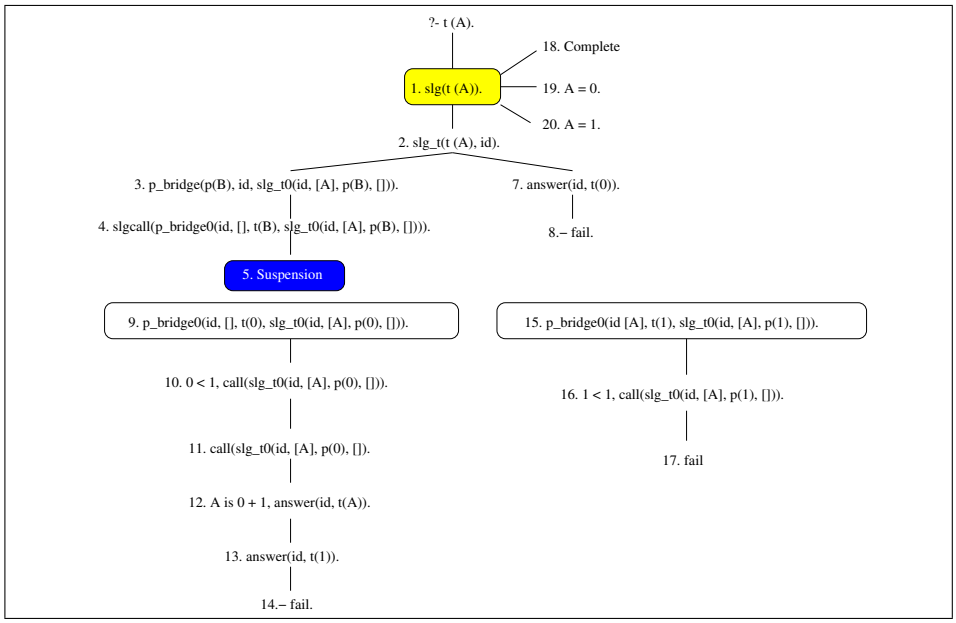


Fig. 9. New `CCall` tabling execution.

5 Θ(CHAT) is not comparable with Θ(CCall)

In this section we present a comparative analysis of the complexity of `CCall` and CHAT, which is an efficient implementation of tabling with a comparatively simple machinery. Since it is known that Θ(CHAT) is Θ(`SLG-WAM`) [7], the comparative analysis applies to the `SLG-WAM` as well.

The complexity analysis focuses on the operations of suspension and resumption. The environment of a consumer has to be protected when suspending to reinstall it when resuming. `CCall` achieves that by copying the continuation associated with the consumer in a special memory area to be protected on backtracking. In the original implementation [15] this continuation is copied from the heap to a separate table (when suspending) and back (when resuming). As proposed in [6], continuations can be saved in a special memory area with the same data format as the heap. This makes it possible to use WAM instructions and additional machinery on them and, when resuming, they can be used as normal Prolog data and code, without being recopied each time a consumer is resumed.

On the other hand, CHAT freezes the heap and the frame stack when resuming. The heap and frame stack are frozen by traversing the choice point stack. For all the choice points between the consumer choice point and its generator, the pointer to the end of the heap and frame stack are changed to the values of the consumer choice point values. By doing that, heap and frame stack are protected on backtracking. However, the consumer choice point has to be

copied to a special memory area as well as the segment trail (with its associated values) between the consumer and the generator, to reinstall the values of the bound variables at the time of suspension which backtracking will unbind. In consequence, when resuming the trail values have to be reinstalled as well as the consumer choice point.

Each consumer is suspended only once, and it can be resumed several times. The rest of the operations, i.e., checking if a tabled call is a generator or a consumer, are not analyzed, because they are common to both systems. In addition, we will ignore the cost of working at the Prolog level, since this is an orthogonal issue: `CCall` primitives could be compiled to WAM instructions and working at Prolog level does not increase the system complexity.

Θ(`CCall`): when suspending, `CCall` has to copy all the environments until the last generator and the structures in the heap which hang from them. If we name `E` the size of all the environments and `H` the size of the structures in the heap, the time consumption when suspending is: Θ(`E + H`).

When resuming, `CCall` just has to perform pattern matching of the continuation against its clause. The time taken by the pattern matching depends on the size of the list of bindings, which is known to be Θ(`E`). Since each consumer can be resumed `N` times, the time consumption of resuming consumers is Θ(`N`$\times$`E`).

Θ(CHAT): when suspending, CHAT has to traverse the frame and choicepoint stacks, but with the improvements presented in [7], the time this takes can be neglected because a choice point is only traversed once for all the consumers. The trail and the last choice point have to be copied. If we call `T` the size of the trail and `C` the size of the choice point, which is bound by a constant for a given program, the time consumption when suspending is: Θ(`T`).

When resuming, CHAT has to reinstall the values of the frame and the choice point. Since each consumer can be resumed `N` times, the time consumption of resuming is Θ(`N`$\times$`T`).

Analyzing the worst cases of both systems: we can conclude `E + H` $\geq$ `T`, because each variable can only be once in the trail, and then `CCall` is worse than CHAT when suspending. On the other hand, in case that `E < T`, `CCall` is better than CHAT when resuming. Consequently, for a plausible general case, the more resumptions there are, the better `CCall` behaves in comparison with CHAT, and conversely. In any case, the worst and best cases for each implementation are different, which makes them difficult to compare. For example, if there is a very large structure pointed to from the environments, and none of its elements are pointed to from the trail, `CCall` is slower than CHAT, since it has to copy all the structure in a different memory area when suspending and CHAT does nothing both when suspending and when resuming.

On the other hand, if all the elements of the structure are pointed to from the trail, `CCall` has to copy all the structure on suspension in a different memory area

to protect it on backtracking, but it is ready to be resumed without any other operation (just a unification with the pointer to the structure). CHAT has to copy all the structure on suspension too, because all the structure is in the trail. In addition, each time the consumer is resumed, all the elements of the structure have to be reinstalled using the trail, and CHAT has to perform more operations than `CCall`, and then, the more resumptions there are, the worse CHAT would be in comparison with `CCall`. Anyway, as the trail is usually much smaller than the heap, in general cases, CHAT will have an advantage over `CCall`.

6 Performance Evaluation

We have implemented the proposed technique as an extension of the Ciao system [1]. Tabled evaluation is provided to the user as a loadable *package* that implements the new directives and user-level predicates, performs the program transformations, and links in the low-level support for tabling. We have implemented `CCall` tabling with the efficiency improvements presented in [6] and the new translation for general programs explained in this paper.

Table 1 aims at determining how the proposed implementation of tabling compares with state-of-the-art systems —namely, the latest available versions of XSB, YapTab, and B-Prolog, at the time of writing, using the typical benchmarks which appear in other performance evaluations of tabling approaches.[7] In this table we provide, for several benchmarks, the raw time (in milliseconds) taken to execute them using tabling. Measurements have been made with Ciao-1.13, using the standard, unoptimized bytecode-based compilation, and with the `CCall` extensions loaded, as well as in XSB 3.0.1, YapTab 5.1.1, and B-Prolog 7.0. Note that we did not compare with CHAT, which was available as a configuration option in the XSB system and which was removed in recent XSB versions. CHAT can be expected to be at least as fast (if not slightly faster) than XSB.

All the executions were performed using local scheduling and disabling garbage collection; in the end this did not impact execution times very much. We used `gcc 4.1.1` to compile all the systems, and we executed them on a machine with Fedora Core Linux, kernel 2.6.9, and an Intel Xeon DESCHUTES processor.

The first benchmark is `path`, the same as Figure 1, which has been executed with a chain-shaped graph. Since this is a tabling-intensive program with no consumers in its execution, the difference with other systems is mainly due to having large parts of the execution done at Prolog level. The following five benchmarks, until `atr2`, are also tabling intensive. As their associated environments are very small, `CCall` is far from its worst case (see Section 5), and the difference with other systems is similar to that in `path` and for a similar reason. The worst case in this set is `tcn` because there are two calls to `slgcall/1` per generator, and the overhead of working at the Prolog level is duplicated.

B-Prolog, which uses a linear tabling approach, suffers if costly predicates have to be recomputed: this is what happens in benchmarks from `pg` until `peep`,

[7] This is in contrast to [6] where, due to the limitations of the `CCall` approach the benchmarks presented did not need the use of bridge predicates.

Program	CCall	XSB	YapTab	BProlog	# Bridges
path	517.92	231.4	151.12	206.26	0
tcl	96.93	59.91	39.16	51.60	0
tcr	315.44	106.91	90.13	96.21	0
tcn	485.77	123.21	85.87	117.70	0
sgm	3151.8	1733.1	1110.1	1474.0	0
atr2	689.86	602.03	262.44	320.07	0
pg	15.240	13.435	8.5482	36.448	6
kalah	23.152	19.187	13.156	28.333	20
gabriel	23.500	19.633	12.384	40.753	12
disj	18.095	15.762	9.2131	29.095	15
cs_o	34.176	27.644	18.169	85.719	14
cs_r	66.699	55.087	34.873	170.25	15
peep	68.757	58.161	37.124	150.14	10

Table 1. Comparing Ciao+CCall with XSB, YapTab, and B-Prolog.

where tabled and non-tabled execution is mixed. This is a well-known disadvantage of linear tabling techniques which does not affect suspension-based approaches. It has to be noted, however, that latest versions of B-Prolog implement an optimized variant of its original linear tabling mechanism [21] which tries to avoid reevaluation of looping subgoals.

In order to compare our implementation with XSB and YapTab, we must take into account that the speeds of XSB, and YapTab[8] are different, at least in those cases where the program execution is large enough to be really significant (between 1.8 and 2 times slower in the case of XSB and 1.5 times faster in the case of YapTab).

In non-trivial benchmarks, from `pg` until `peep`, which at least in principle should reflect more accurately what one might expect in larger applications using tabling, execution times are in the end very competitive when comparing with XSB or YapTab. This is probably due to the fact that the raw speed of the basic engine in Ciao is higher than in XSB and closer to YapTab, rather than to factors related to tabling execution, but it also implies that the overhead of the approach to tabling used is reasonable after the proposed optimizations in [6]. In this context it should be noted that in these experiments we have used the baseline, bytecode-based compilation and abstract machine. Turning on global analysis and using optimizing compilers and abstract machines [11, 3, 12] can further improve the speed of the SLD part of the computation.

[8] Note that we are comparing the tabled-enabled version of Yap, which is somewhat slower than the regular Yap.

7 Conclusions

We have presented an extension of the continuation call technique which does not have the limitations of the original continuation call approach regarding the interleaving of tabled and non-tabled predicates. This approach has the advantage of being easier to implement and maintain than other techniques which require non-trivial modifications to low-level machinery. Although there is an overhead imposed by executing at Prolog level, we expect the speed of the source (Prolog) language to gradually improve by using global analysis, optimizing compilers, and better abstract machines. Accordingly, we expect the performance of `CCall` to improve in the future and thus gradually gain ground in the comparisons.

Although a non optimal tabled execution is obviously a disadvantage, it is worth noting that, since our implementation introduces only minimal changes in the WAM and none in the associated Prolog compiler, the speed at which regular Prolog is executed remains unchanged. In addition to this, the modular design of our approach gives better chances of making it easier to port to other systems. In our case, executables which do not need tabling have very little tabling-related code, as the data structures (for tries, etc.) are handled by dynamic libraries loaded on demand, and only stubs are needed in the regular engine. The program transformation is taken care of by a plugin for the Ciao compiler [2] (a "package," in Ciao's terms) which is loaded and active only at compile time, and which does not remain in the final executable.

References

1. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.
2. D. Cabeza and M. Hermenegildo. The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
3. M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo. High-Level Languages for Small Devices: A Case Study. In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.
4. Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
5. S. Dawson, C.R. Ramakrishnan, and D.S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In *Proceedings of PLDI'96*, pages 117–126, New York, USA, 1996. ACM Press.
6. P. Chico de Guzmán, M. Carro, M. Hermenegildo, Claudio Silva, and Ricardo Rocha. An Improved Continuation Call-Based Implementation of Tabling. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 198–213. Springer-Verlag, January 2008.

7. Bart Demoen and K. Sagonas. CHAT is θ(SLG-WAM). In D. Mc. Allester H. Ganzinger and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning*, volume 1705 of *Lectures Notes in Computer Science*, pages 337–357. Springer, September 1999.
8. Bart Demoen and Konstantinos Sagonas. CAT: The Copying Approach to Tabling. In *Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 21–35. Springer-Verlag, 1998.
9. Bart Demoen and Konstantinos F. Sagonas. Chat: The copy-hybrid approach to tabling. In *Practical Applications of Declarative Languages*, pages 106–121, 1999.
10. Hai-Feng Guo and Gopal Gupta. A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In *International Conference on Logic Programming*, pages 181–196, 2001.
11. J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.
12. J. Morales, M. Carro, and M. Hermenegildo. Comparing Tag Scheme Variations Using an Abstract Machine Generator. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 32–43. ACM Press, July 2008.
13. Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 143–154. Springer Verlag, 1997.
14. Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
15. R. Ramesh and Weidong Chen. Implementation of tabled evaluation with delaying in prolog. *IEEE Trans. Knowl. Data Eng.*, 9(4):559–574, 1997.
16. R. Rocha, C. Silva, and R. Lopes. On Applying Program Transformation to Implement Suspension-Based Tabling in Prolog. In V. Dahl and I. Niemelä, editors, *23rd International Conference on Logic Programming*, number 4670 in LNCS, pages 444–445, Porto, Portugal, September 2007. Springer-Verlag.
17. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, May 1998.
18. H. Tamaki and M. Sato. OLD resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, 1986. Lecture Notes in Computer Science, Springer-Verlag.
19. D.S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.
20. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.
21. Neng-Fa Zhou, T. Sato, and Yi-Dong Shen. Linear Tabling Strategies and Optimizations. *Theory and Practice of Logic Programming*, 8(1):81–109, 2008.
22. Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming*, 2001(10), October 2001.
23. Youyong Zou, Tim Finin, and Harry Chen. F-OWL: An Inference Engine for Semantic Web. In *Formal Approaches to Agent-Based Systems*, volume 3228 of *Lecture Notes in Computer Science*, pages 238–248. Springer Verlag, January 2005.

Extending Tabled Logic Programming with Multi-Threading: A Systems Perspective

Rui Marques[1], Terrance Swift[2], and José Cunha[1]

[1] CITI, Dep. Informática – FCT, Universidade Nova de Lisboa
[2] CENTRIA — Universidade Nova de Lisboa

Abstract. Tabled Logic Programming (TLP) has proven a useful paradigm for application areas such as natural language grammars, program analysis, model checking, ontology management, collaborative agents, and the semantic web. The benefits of TLP arise from the fact that tabling factors out redundant subcomputations when evaluating a goal, leading to powerful termination and complexity properties. While the design and implementation of sequential TLP systems has been heavily studied, multi-threaded TLP systems are much newer. Tabling can be integrated with multi-threading in a variety of ways. Different threads may use *private* tables to support their own computations, while *shared* tables can be used as a basis of communication among threads to amortize repeated queries and to exploit a measure of parallelism from a computation. This paper discusses multi-threaded TLP in the context of XSB, a leading open-source Prolog whose tabling engine has recently been extended for multi-threading, including tabled negation, tabled constraints, and subsumptive tabling.

Tabled Logic Programming (TLP) has proven to be an important area of Logic Programming (LP) over the last decade, with research and commercial use in such areas as natural language grammars, program analysis, model checking, ontology management, collaborative agents, and the semantic web. Following the initial implementation of tabling in XSB, various forms of tabling have been added to other open-source Prologs including B-Prolog, YAP, Mercury, ALS and Ciao. There are a number of reasons for the adoption of tabling. TLP is more declarative than LP: it ensures termination and polynomial complexity for logic programs with negation that have the *bounded term size* property – i.e. those for which the size of terms constructed during an evaluation is bounded. Tabling can also evaluate negation according to the Well-Founded Semantics, which among other advantages allows an integration of Prolog-style systems with ASP systems that solve combinatorial problems. Finally, tabling can be closely integrated with Prolog systems so that constraints, cuts, and exceptions are supported, and implemented through extensions of Prolog's virtual machine.

Multi-threaded Prolog has been developed as a research activity for many years (cf. e.g [1]) and a draft ISO standard is available [2]. Many Prologs, including Ciao, SWI, YAP, Qu-Prolog and XSB, support multi-threaded programming, allowing programmers to benefit from parallelism by manually decompos-

ing queries. They provide a sophisticated environment for a number of applications, but most of them do not support multi-threaded TLP (MT-TLP).

This paper presents the approach to MT-TLP taken by XSB Prolog, which supports a wide variety of TLP features, including tabled negation, tabled constraints, call subsumption, answer subsumption, incremental recomputation of tables, tabled dynamic code and garbage collection of abolished tables. When multi-threading is added, tables may be *private* to a thread, or *shared* among threads, leading to several design goals:

- Any tabling function should be available to any active thread using tables that are private to a thread.
- Any tabling function should be available to any active thread using tables that are shared among threads.
- Private tables should be highly scalable up to the number of cores available.
- For problems that support large amounts of parallelism, shared tables should be able to provide speedup proportional to the number of cores available.

Although these goals are ambitious, many are already supported in Version 3.2 of XSB [3]. We first review aspects of TLP in Section 1. We then describe MT-TLP in XSB Version 3.2, including a high-level description of algorithms for multi-threaded computations that share tables. In order to illustrate how MT-TLP functions can be used in practice, Section 3 presents extended examples of its application to various types of Petri-Net formalisms. Finally, Section 4 discusses performance of these and other examples.

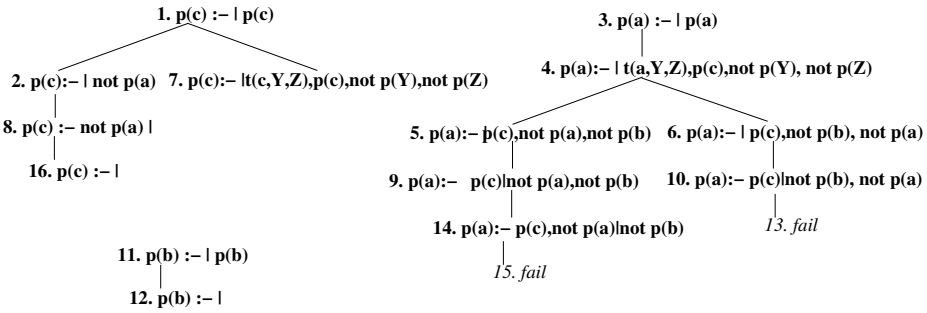
1 Tabling

We review aspects of tabling relevant to our presentation. Due to space limitations, the presentation is informal: references for formalisms of tabling and for proofs of its complexity and termination properties can be found in e.g. [3].

Example 1. Figure 1 shows a program P_1 along with the tabled evaluation of the goal `?- p(c)` represented as a forest of trees. In Figure 1, the number beside each node indicates the order of its creation. Accordingly, the evaluation begins with a tree for `p(c)` with root node labeled with the clause `p(c):- |p(c)`. The head of each node label represents the bindings, if any, to the subgoal made during the course of evaluation. The body of a node label has two parts: the literals on the right of the `|` symbol represent unresolved goals, which in this paper are selected in a fixed left-to-right order; the literals on the left represent *delayed* literals. PROGRAM CLAUSE RESOLUTION is used to produce the children of the root of a tree: this operation produces node 1, with selected literal `not p(a)`. The atom `p(a)` is new to the evaluation, so a NEW SUBGOAL operation creates a tree rooted at node 3 for `p(a)`. PROGRAM CLAUSE RESOLUTION creates node 4, and because the selected literal of node 4 `t(a,Y,Z)` is not tabled, PROGRAM

[3] Version 3.2 has not yet been released, but all features mentioned in this paper are available in the CVS version of XSB available at `xsb.sourceforge.net`.

CLAUSE RESOLUTION creates node 5 also. Since the selected literal of node 5, `p(c)` has a tree in the forest, but has no answers, the evaluation backtracks to node 4 and produces node 6. Once again, there are no answers to return so the evaluation backtracks to the tree for `p(c)` and produces node 7. At this stage none of the selected literals for nodes 1-7 have answers, nor unresolved program clauses. However, SLG provides a DELAYING operation which allows a selected negative literal to be moved to the left of the `|` symbol. This operation creates node 8, which is considered to be an answer, since it has no unresolved goals; at the same time, since node 8 has a delayed literal it is a *conditional* answer. This conditional answer can be used to resolve against the selected literals of nodes 5 and 6, producing nodes 9 and 10. Note that using a conditional answer for resolution causes the resolved goal to be delayed. The atom `p(b)` of the selected literal of node 10 is new to the evaluation, so a new tree is created for `p(b)` which produces an unconditional answer causing the derivation path from node 10 to be failed as indicated by the failure node 13. At this point, node 9 has `not p(a)` as its selected goal. Another DELAYING operation is performed to produce node 14, which again has `not p(b)` as its selected literal and produces a failure node. At this point, all operations have been performed on the selected literals of all nodes. The trees are *completely evaluated* and can be marked as *complete*. Once they are completed and it is determined that `p(a)` has no (conditional or unconditional) answers, a SIMPLIFICATION operation removes `not p(a)` from the delay list of node 8 to produce the unconditional answer in node 16.




```
:– table p/1.

p(c):– not p(a).        p(b).        p(X):– t(X,Y,Z),p(c),not p(Y),not p(Z).

t(a,b,a).               t(a,a,b).
```

Fig. 1. The program P_1 and tabled evaluation of goal `?- p(c)` to P_1

Example 1 illustrates a number of operational aspects of tabling. First, a tabled evaluation needs to be able to *suspend* and later *resume* a computation path, as when the path to node 5 is suspended and later resumed to produce node 9. Next, since non-completed subgoals require execution stack space while completed subgoals require only table space to store their answers, a practical tabled evaluation must be able to *incrementally complete* tabled subgoals to ensure space efficiency. For instance, the tree for `p(b)` can be completed immediately after node 12 is produced.

However, there are aspects of tabled evaluation that Example 1 does not explicitly demonstrate. Example 1 implicitly uses *call variance* – a NEW SUBGOAL operation is performed on a tabled subgoal S if no variant of S has previously been encountered. In some evaluations, it can be useful to restrict NEW SUBGOAL operations to occur only if no *subsuming* call for S had been encountered. Call subsumption can be efficient for applications that can exploit it – for instance computing a bottom-up fixed point for program analysis or for RDF inference. However call subsumption introduces overheads when it is not used (about 20% in XSB). Furthermore, there may be situations in which it is important to maintain the call patterns of an evaluation, as in tabling a meta-interpreter: such patterns are preserved by call variance, but not by call subsumption. In addition, Example 1 does not make use of a tabling feature called *answer subsumption*. Rather than returning every answer for a (perhaps completed) table, it may be best to return answers that are optimal according to some partial order. Similarly, answer subsumption may return an answer that is a function of other answers. For instance, an answer may be resolved against a consuming subgoal only if it is the join of other answers [4], or if the answer is the summation of independently derived probabilities [5]. As shown by the Petri Net example in Section 3 that uses ω-sequences, answer subsumption also can be useful for ensuring termination by abstracting answers.

Example 1 also does not use dynamic code, which interacts with tabling in two ways. First, dynamic predicates can be tabled in XSB, a handy feature for applications that generate tabled code. Second, when a tabled evaluation relies on a dynamic predicate D_p information in the table may become out of date as clauses of D_p are asserted or retracted. By using suitable declarations, *incremental recomputation* can automatically maintain the consistency of tables with dynamic code. While in Version 3.2 of XSB incremental recomputation is restricted to definite programs, it has proven useful in applications [6].

The ability to maintain constrained variables in the subgoals and answers of tables is useful for the analysis of temporal systems (see for instance [7]). A final critical, but often overlooked feature of tabling systems is the ability to abolish tables and reclaim their space. Version 3.2 of XSB allows reclamation of table space for abolished completed tables at the predicate and subgoal level. It may not be safe to immediately reclaim the space of an abolished table, as choice points may point into the table's code. Thus, in a manner similar to reclaiming retracted dynamic clauses, a pointer to the predicate or subgoal is put on a list of elements to later garbage collect when it is safe to do so.

Scheduling Strategies Two popular strategies for performing tabling operations are Local evaluation (the underlying strategy of Example 1) and Batched evaluation. Local evaluation is based on a *Subgoal Dependency Graph (SDG)* constructed from a forest of trees, $\mathcal{F}$ (cf. Figure 1). This graph has as its vertices each non-completed tabled subgoal in the forest, and has a link (S_1, S_2) if a node in the tree for subgoal S_1 has subgoal S_2 in its selected literal or in a delayed literal. Since $SDG(\mathcal{F})$ is a directed graph, a Strongly Connected Component (SCC) can be defined; As terminology a maximal SCC is an SCC that is contained in no other SCC, while an independent SCC $\mathcal{S}$ is an SCC such that there is no edge from a vertex in $\mathcal{S}$ to a vertex not contained in $\mathcal{S}$. In Local evaluation, tabling operations are performed only in trees whose subgoals are in an independent maximal SCC. Because of this restriction, Local evaluations have a behavior similar to a depth-first search. As a result, a given state of a Local evaluation generally has few uncompleted subgoals, and so is space efficient. In addition, Local evaluation prevents the return of an answer to a node in a tree that is not in an independent SCC. Along with other scheduling constraints, including ensuring that all SIMPLIFICATION operations are performed as early as possible, Local evaluation can guarantee that if a conditional answer with head A is returned to a node N outside of an independent SCC, then no unconditional answer with head A will ever be available to be returned to N. In Example 1 this would mean that if `p(a)` were part of a larger evaluation, its conditional answer (node 8) would never be returned outside of its SCC: only the unconditional answer (node 16) would be thus returned. Local evaluation is also advantageous for answer subsumption since it returns only the best answers (according to a given ordering) outside of an SCC.

However Local evaluation, is not useful for applications that require a single answer, or for applications where a table produces an answer that can be concurrently consumed by some other thread. For these purposes, Batched Evaluation is superior. Batched evaluation treats bindings made by answer resolution in the same way substitutions are treated in Prolog: the binding is propagated to all ancestor environments, thus "returning" an answer to its calling environment immediately. Answers are also returned to consuming nodes upon backtracking. Upon backtracking to the oldest subgoal in an SCC $\mathcal{S}$, $\mathcal{S}$ is either completed or a backtracking chain is created to return unresolved answers to consuming nodes for subgoals in $\mathcal{S}$. In this manner, answers are scheduled for return a batch at a time. For left recursion, Batched evaluation is about 10-20% faster than Local evaluation. The decision of whether to use Batched or Local evaluation is thus application dependent. XSB must be configured with one or the other strategy, but YAP allows dynamic mixing of the strategies [8].

2 Multi-Threaded Tabling

A multi-threaded tabling engine [9] was first made available in Version 3.0 of XSB, and has been substantially refined and extended since then. The simplest execution model is based on private tables, where each thread keeps its own copy of tabled information. This model has several advantages:

- Private tables use sequential tabling algorithms. The main implementation problems are to make the tabling engine reentrant with a low overhead, to allow each thread to reclaim its own table space and to ensure that allocation of table space does not affect scalability. Private tables in XSB support all tabling features that were present at the time of implementation, including tabled negation, tabled constraints, and call and answer subsumption.
- Private tables generally require no synchronization among threads above the level of memory allocation.
- Private tables are suitable to ensure query completeness or to support a particular semantics. Tables are automatically reclaimed when the thread that computed them exits. This reclamation includes not only subgoal and answer tries, but the delay lists and supporting structures used to compute the Well-Founded Semantics.

Shared tables tables are also important:

- If different threads require the same tables, memory usage for shared tables will be significantly lower than for private tables.
- Shared tables amortize execution time for (sub-)queries that are repeated by more than one thread.
- Shared tables allow the decomposition of a program, so that a set of threads computes a set of tables, partially supporting Table-Parallelism [10].

Execution Models for Shared Tables In [9] two models for shared tables, *Concurrent Local Evaluation* and *Concurrent Batched Evaluation* were proposed and implemented. In these models, the SLG forest is dynamically partitioned among threads, each thread evaluating a set of subgoals. In Concurrent Local Evaluation, which relies on Local Scheduling, when a thread T encounters a tabled subgoal S that has not been encountered by any thread, T evaluates S. Other threads are only allowed to use the table for S after T has completed S. Concurrency control for tables mainly arises when more than one thread evaluates different tabled subgoals in the same SCC at the same time. In this case, a deadlock will occur, which the engine detects and resolves, so that a single thread assumes computation of all tabled subgoals in the SCC. In Figure 1 such as situation would occur if a thread T_1 called `p(a)` and another called `p(c)` before it was called by T_1. Tabled subgoals that are computed by a new thread must have their answers recomputed. It is shown in [9] that recomputation does not add to the abstract complexity of the Well-Founded Semantics. Just as Local evaluation is the default scheduling strategy for sequential XSB and for thread-private tables, Concurrent Local Evaluation is the default scheduling strategy for thread-shared tables.

Because it is a type of Local Evaluation, Concurrent Local Evaluation does not allow a consuming node to use answers produced by a subgoal outside of its SCC until the table for the answers is completed – a restriction that prevents producer-consumer models of parallelism. This limitation is overcome by *Concurrent Batched Evaluation* which allows several threads to compute (inter-)dependent tabled subgoals in parallel. As with Concurrent Local Evaluation,

each subgoal can be computed by only one thread. However, a given thread may consume answers as they are produced by another thread. Within XSB, the implementation of Concurrent Batched Evaluation extends the implementation of sequential Batched Evaluation. In sequential Batched Evaluation, when the engine backtracks to the oldest subgoal in an SCC, it schedules the return of unconsumed answers for each consuming node in the SCC by creating a chain of choice points, and then backtracks into the newly created chain. This is extended to a multi-threaded context as follows. If different threads compute different SCCs, they can work independently, and can consume answers from other threads as they become available. However, let $\mathcal{S}$ be an SCC computed by multiple threads. All threads concurrently consume answers and perform other operations while they have work to do. Suppose a thread T_1 computing subgoals in $\mathcal{S}$ backtracks to the oldest subgoal that it "owns" in $\mathcal{S}$. If any other thread computing $\mathcal{S}$ is active, T_1 will suspend and will be re-awakened when a thread performs batch scheduling for $\mathcal{S}$; otherwise if T_1 is the last unsuspended thread computing subgoals in $\mathcal{S}$, T_1 itself will perform a fixed point check and batched scheduling and awaken the other threads computing $\mathcal{S}$ — either to return further answers or to complete their tables. As implemented in XSB, Concurrent Batched Evaluation thus allows parallel computation of subgoals, but has a sequential fixpoint check that synchronizes multiple threads when they compute the same SCC.

Implementation Status The status of MT-TLP in XSB Version 3.2 is shown in Table 1. Private tables support all features except for incremental recomputation (cf. Section 1, which was introduced after the multi-threaded engine was introduced into XSB. Concurrent Local Evaluation supports most features, but does not yet support call subsumption. In addition, it only partially supports space reclamation since shared tables can be abolished, but their space will not be reclaimed until there is only a single active thread in the engine. Both private tables and shared tables under Concurrent Local Evaluation have been heavily tested. XSB can also be configured to use Concurrent Batched Evaluation, however this model has been less thoroughly tested than Concurrent Local Evaluation and should be considered experimental. Nonetheless, Concurrent Batched Completion supports a number of tabling features, but is currently restricted to left-to-right dynamically stratified programs.

Related Work The approach to MT-TLP in XSB can be contrasted to that of OptYap [11]. OptYap extends an Or-parallel Prolog system with tabling, while XSB extends a Tabled Prolog system to allow multi-threading. The different starting points lead to different strengths in the current implementation of each system. In OptYap, different workers can collaborate to solve the same goal – leading to impressive speedups even in programs using left recursion. As will be shown in Section 4, shared tables in XSB can be used to speed up evaluations, but only for problems that are easily decomposable. Thus for definite programs, to which OptYap is currently restricted, OptYap can exploit much more parallelism than can XSB. On the other hand, XSB's implementation supports more tabling features within multi-threading, and integrates multi-threaded tabling more thoroughly with other system features, such as dynamic code and space

Feature	Private Tables	Shared Tables (Local)	Shared Tables (Batched-β)
Tabled constraints	Supported	Supported	Supported
Answer subsumption	Supported	Supported	Supported
Tabled Dynamic Code	Supported	Supported	Supported
Tabled negation	Supported	Supported	Partially Supported
Space reclamation	Supported	Partially Supported	Partially Supported
Call subsumption	Supported	Not supported	Not Supported
Incremental recomputation	Not supported	Not supported	Not Supported

Table 1. Multi-threaded functionality in XSB v. 3.2

reclamation. As a result, XSB can multi-thread computations that OptYap cannot (currently) evaluate, including several examples from Section 3.

3 Analysis of Petri Nets and Workflow Nets

The analysis of process logics in the style of Petri Nets illustrates a use of various tabled evaluations can exploit multi-threading. Reachability is a central problem for Petri Net analysis, to which problems such as liveness, deadlock-freedom, and the existence of homes states can be reduced. While we have taken care that the programs shown are correct and motivated by use cases, we stress that the methods described in this section are intended primarily to illustrate MT-TLP and to support the performance studies of Section 4, but do not represent fully developed analysis systems for Petri or Workflow Nets [4].

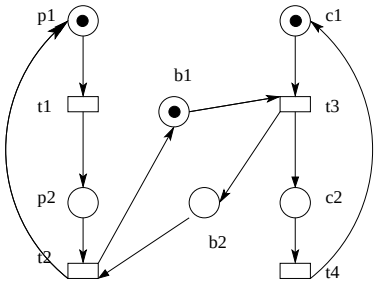


Fig. 2. A Simple Producer-Consumer Net

Using Tabling for Elementary Petri Nets Elementary Petri Nets (EPNs) or 1-safe Petri Nets (cf. [12]) are particularly simple to analyze. Consider the EPN shown in Figure 2, which depicts a simple producer consumer system. An EPN allows a place to contain at most 1 token; thus a finite EPN will have

[4] All programs can be obtained via `http://xsb.cvs.sourceforge.net/xsb/mttests/benches`.

only a finite number of configurations so that determining reachability of an EPN configuration is decidable. Our encoding represents the configuration of an EPN by a list of its marked places: thus the configuration in Figure 2 is represented as the list `[b1,c1,p1]`. Next, a transition T is represented by a list of places with input arcs to T ($\bullet T$) and output arcs from T ($T\bullet$). Predicate `trans/3` in Figure 3 shows each transition of Figure 2 represented as a Prolog fact, and that the transitions use XSB's trie indexing to obtain full indexing on list elements. Figure 3 shows a program for determining reachability in an EPN; so that solutions to the goal `reachable([b1,c1,p1],X)` are configurations reachable from the EPN in Figure 2. For efficiency the reachability program assumes that the lists in all transitions and configurations are sorted. For a transition T to have concession in a configuration C of an EPN, every place in $\bullet T$ must be marked, and no place in $T\bullet$ can be marked. These conditions are checked by the predicate `hasTransition/2` in Figure 3 which recurses through the places in the current configuration (`Conf`) to find sets of transitions that might have concession. This recursion (in `get_trans_for_conf_1/3`) allows indexed calls to transitions to be made based on each place in the input configuration. Each set of possible transitions is then filtered to include only those transitions that actually have concession in `Conf`, using operations on ordered sets (via `check_concession/2`). `hasTransition/2` succeeds when the first of these transitions is applied; further transitions are applied upon backtracking.

Based on `hasTransition/2`, a tabled reachability predicate can be written as a simple left-recursion. Tabling `reachable/2` is useful in two ways: it prevents looping when a given configuration is reachable from itself; and it also filters out redundant paths to a reachable configuration. By using the left recursive form of `reachable/2`, a typical call such as `reachable([b1,c1,p1],X)` with first argument bound and second free, would require a single tabled subgoal, and would have as answers all configurations reachable from `[b1,c1,p1]`. XSB's use of tries to represent tabled subgoals and their answers, allows efficient checking of answers and efficient use of memory, since the trie data structure factors out common list prefixes. If `reachable/2` is made thread-shared, then various threads can access the table to determine useful transitions, isolated places, and other information. Reachability analysis can exploit multi-threading if there is more than one initial configuration of interest or if a Petri Net is coarsely decomposable.

Using Petri Nets to Model Workflows The analysis and verification workflows is a promising direction for MT-TLP. Petri net-based formalisms, called Workflow Nets, are suitable to represent control and data flows, such as loops, I/O preconditions, if/then clauses and other synchronization dependencies between workflow units. To model reachability in a Workflow Net, the EPN is first extended to allow multiple tokens in a given place, and to change the representation of marked place from a constant such as `p1` to a Prolog term that is marked with a given instance and perhaps other information, e.g. `p1(instance(7))`. Transitions are then extended with functionality to dynamically evaluate guard conditions, to create sub-instances, to check for the absence of tokens in given places (which allows merging of dynamically created paths

```
% Prolog representation of the Producer-Consumer Net
:- index(trans/2,trie).
trans([p1],[p2],t1).          trans([b2,p2],[p1,b1],t2).
trans([b1,c1],[b2,c2],t3).    trans([c2],[c1],t4).

% Program to determine reachability of an elementary net
:- table reachable/2.
reachable(InConf,NewConf):-
   reachable(InConf,Conf),
   hasTransition(Conf,NewConf).
reachable(InConf,NewConf):-
   hasTransition(InConf,NewConf).

hasTransition(Conf,NewConf):-
   get_trans_for_conf(Conf,AllTrans),
   member(Trans,AllTrans),
   apply_trans_to_conf(Trans,Conf,NewConf).

get_trans_for_conf(Conf,Flattrans):-
   get_trans_for_conf_1(Conf,Conf,Trans),
   flatten(Trans,Flattrans).

get_trans_for_conf_1([],_Conf,[]).
get_trans_for_conf_1([H|T],Conf,[Trans1|RT]):-
   findall(trans([H|In],Out,Tran),trans([H|In],Out,Tran),Trans),
   check_concession(Trans,Conf,Trans1),
   get_trans_for_conf_1(T,Conf,RT).

check_concession([],_,[]).
check_concession([trans(In,Out,Name)|T],Input,[trans(In,Out,Name)|T1]):-
   ord_subset(In,Input),
   ord_disjoint(Out,Input),!,
   check_concession(T,Input,T1).
check_concession([_Trans|T],Input,T1):-
   check_concession(T,Input,T1).

apply_trans_to_conf(trans(In,Out_Name),Conf,NewConf):-
   ord_subtract(Conf,In,Diff),
   flatten([Out|Diff],Temp),
   sort(Temp,NewConf).
```

Fig. 3. TLP Program for analyzing Elementary Petri Nets

through the net), and to delete tokens from places if a transition is taken (which allows cancellation). Transitions for Workflow Net have the abstract form

```
trans(InConf,OutConf,dyn(Conditions,Effects))
```

where the last argument contains dynamic conditions that must be satisfied before the transition can be taken, and dynamic effects to be applied upon taking the transition (e.g. cancellation). The Workflow Net evaluator based on this syntax is approximately twice the size of that of Figure 3, and can emulate nearly all common workflow control patterns [13]. In fact, the emulator has been used with MT-TLP to analyze health workflows based on clinical care guidelines.

Using Answer Subsumption for ω Sequences Workflow nets are an extension of Place/Transition Petri Nets, which do not distinguish between tokens, but do allow a place to hold more than one token. Reachability is decidable in Place/Transition Nets, and can be determined using a method called ω-sequences (see e.g. [14]). The main idea in determining ω sequences is to define a partial order $\geq_\omega$ as follows. If configurations C_1 and C_2 are both reachable, C_1 and C_2 have tokens in the same set Pl of places, and there exists a non-empty $PL_{sub} \subseteq PL$, such that for each $pl \in Pl_{sub}$ C_1 has strictly more tokens than C_2, then $C_1 >_\omega C_2$. When evaluating reachability, if C_2 is reached first, and then C_1 was subsequently reached, C_1 is abstracted by marking each place in PL_{sub} with the special token ω which is taken to be greater than any integer. If C_1 was reached first and then C_2, C_2 is treated as having already been seen.

From the viewpoint of TLP, ω-abstractions form an example of answer subsumption. To compute reachability with ω abstractions, when each solution S to `reachable/2` is obtained, the solution S is compared to answers in the table. If some answer in the table is greater than or equal to S in $\geq_\omega$ then S is not added to the table; however if S is greater than some set $\mathcal{S}_A$ of answers, the answers $\mathcal{S}_A$ are removed from the table and the ω abstraction of S with respect to $\mathcal{S}_A$ is added. The main top-level change to Figure 3 needed for implementation is the use of the XSB library predicate `filterPOA/5` as the top-level call and in the first clause of `reachable/2`.

```
reachable(InConf,NewConf):-
   filterPOA(reachable(InConf),Conf,gte_omega,omega_abstr,call_abstr),
   hasTransition(Conf,NewConf).
```

`filterPOA/5` takes the call and argument to which answer subsumption is to be applied as its first two arguments, while the third argument, `gte_omega` is the name of the partial order itself. The forth argument is the name of the predicate to use to perform ω-abstraction of answers. Finally, the fifth argument, is the name of the predicate to compare a candidate solution Sol to answers in the table. The predicate, `call_abstr/2` abstracts Sol to form a call to the table so that only a small set of answers will be compared with Sol to determine if Sol should be added to the table and possibly ω-abstracted [5]. In other words, upon derivation of Sol, a term $Call_{Sol}$ is created using `call_abs/2` and all answers in

[5] `filterPOA/5` is itself tabled and uses thread-private tables; for shared tables `shared_filterPOA/5` is used.

the current table that unify with $Call_{Sol}$ are collected. Each of these is compared to Sol using `gte_omega/2`. If one of the answers $>_\omega$ than Sol, the predicate fails; otherwise the set $\mathcal{A}$ of answers that Sol is $>_\omega$ than is collected, and if non-empty, the abstraction of Sol with respect to $\mathcal{A}$, Sol_{abs} is taken; the answers in $\mathcal{A}$ deleted from the table, and Sol_{abs} added.

Extending nets with Constraint-based Reasoning A variety of formalisms extend Place/Transition Nets to add conditions that must be evaluated for a transition to fire and effects that must occur upon its firing. In the Workflow nets described above conditions and effects were Prolog predicates, but there is no reason why a condition could not be the entailment of a formula in a given constraint domain, and the effect the propagation of new constraints to variables associated with given places in the net. Using such an approach, constraint-based reasoning can be incorporated into workflow or other process specifications. The top-level change required to implement constraint nets occurs when actually applying a transition to a configuration, in `apply_trans_to_conf/3`:

```
apply_trans_to_conf(trans(In,Entailment,Out),Conf,NewConf):-
   unify_for_entailment(In,Conf,MidConf),
   entailed(Entailment),
   call_new_constraints(Out,OutPlaces),
   flatsort([OutPlaces|MidConf],NewConf).
```

First, variables in the transition are unified with those of the configuration to produce a new constraint store. If the formula `Entailment` is entailed by the constraint store, new constraints from the transition are placed on the output variables via calling the constraints in the list `Out`. Note that this extension is not specific to a given constraint domain, but its use for reachability does depend on tabled constraints.

Using Tabled Negation for Preferences on Nets Preferences can be combined with Workflow nets so that if more than one transition is possible for a given configuration C of a workflow instance, only preferred transitions from C are taken. This has two practical uses. First, the preferences may check run-time information from a database or other store to determine what transitions to avoid: in fact, since the preference relation is simply a (tabled) Prolog predicate the preference relation may perform sophisticated run-time look-aheads. Second, since preferences can be dynamic, they may be used to fine-tune a general workflow to local policies – for instance adjusting a clinical workflow system to policies of a given hospital, medical department, or ward. Adapting the methodology of [15], the top-level change to the code of Figure 3 is to the `hasTransition/2` predicate

```
hasTransition(Conf,NewConf):-
   get_trans_for_conf(Conf,AllTrans),
   member(Trans,AllTrans),
   sk_not(unpreferred(Trans,AllTrans,Conf)),
   apply_trans_to_conf(Trans,Conf,NewConf).
```

`sk_not/1` is an XSB predicate that soundly evaluates non-ground tabled negation by skolemizing variables, ensuring here that only preferred transitions are taken. Since the basis for preferences is the well-founded semantics, if a transition is

preferred to itself at a given configuration, `hasTransition/2` will produce an answer that is neither true nor false.

Summary: Tabling for Petri Nets Tabling provides a concise means for coding reachablility (and other analysis problems) for a variety of Petri-net formalisms. At the same time, tabling may not be the best approach for all such problems. Reachability in EPNs is in *PSPACE* [16], while in the worst case, tabling requires 2^N states for an EPN with N places. At the same time, algorithms for reachability that stay in *PSPACE* (e.g. by using loop-checking) will in the worst case require time proportional to the number of traces (paths) rather than to the number of states, as required by tabling.

4 Performance Results

Table 2 shows the performance results for benchmarks on a machine with a 4 core AMD 64 processor running Debian Linux. All times were taken as the best of three runs and are presented in seconds. The programs `Elementary`, `Workflow`, `Omega`, `Constraint`, and `Preferences` were all discussed in the previous section. `Dynamic Elementary` is the same as `Elementary` except that it uses tabled dynamic code for `reachable/2`. The nets tested vary with each type of benchmark. For (Dynamic) `Elementary`, the underlying nets are designed to capture the effects of repeatedly locking and unlocking mutexes, while in `Workflow` the net is designed to use a number of standard workflow control patterns from [13]. The net for `Omega` was synthesized to have a relatively small number of places in which ω-abstractions were necessary, although the *check* for whether an ω-abstraction was needed was necessary in all places. For `Constraints`, the network was designed so that places compete for a shared resource represented by a term with variables constrained using CLP(R). Once a place obtains a resource, various transitions fire to constrain the variables of a resource until they entail the guard of a transition that moves the term to another place along a path, and eventually back to the initial configuration. The net for `Preferences` extends a workflow net to prefer those transitions from a given configuration that cannot lead to proscribed configurations: the preferences thus model look-ahead within a workflow state. Due to differingzd limitations on the sizes of shared and of multiple copies of private tables, the sizes of the nets differ between private and shared versions of each benchmark, resulting in different performance numbers.

The benchmark `Call Subsumption` does not use a Petri Net formalism, but rather evaluates the goal `?- ranc(A,B)` to the tabled predicate

```
ranc(X,Y):- edge(X,Y).          ranc(X,Y):- edge(X,Z),ranc(Z,Y).
```

where `ranc/2` uses call subsumption and `edge/2` is a chain of 2048 vertices.

Table 2 presents the results of the benchmarks; however two other features of the benchmarks must be be explained before evaluating the results. First, the sizes of the underlying nets vary greatly from test to test, as do the number of reachable states – thus the absolute times should not be used to compare the benchmark of one kind of net to another. Second, shared table benchmarks test the time for N threads to each traverse 4/N identical nets. Thus, the shared

Programs Using Private Tables (Local Evaluation)					
N. threads	1	2	Overhead	4	Overhead
`Private Elementary`	5.94	6.23	4.8%	6.25	5.2%
`Private Dynamic Elementary`	6.03	6.03	0%	6.03	0%
`Private Workflow`	19.21	19.68	2.4%	19.95	3.8%
`Private Omega`	7.18	8.33	16.0%	10.3	46.0%
`Private Omega Specialized`	6.37	6.37	0%	6.37	0.0%
`Private Constraint`	2.75	2.84	3.2%	2.85	3.6%
`Private Preferences`	3.74	3.77	0.8%	3.82	2.1%
`Call Subsumption`	.86	1.04	20.0%	1	43%

Programs Using Shared Tables (Local Evaluation)					
N. threads	1	2	Speedup	4	Speedup
`Shared Elementary`	25.12	13.00	1.93	6.55	3.83
`Shared Dynamic Elemtary`	24.8	13.02	1.90	6.59	3.76
`Shared Workflow`	41.25	20.78	1.98	10.58	3.89
`Shared Omega`	19.58	10.38	1.88	5.57	3.51
`Shared Constraint`	11.13	5.56	2.00	2.83	3.93
`Shared Preferences`	3.73	1.86	1.99	0.95	3.92

Table 2. Scalability Results for Private and Shared Tables (Local Evaluation)

benchmarks test a "best case" situation for exploiting parallelism by shared tables. In Table 2 the scalability for both private and shared tables is usually linear to 4 cores, and the times for `Dynamic Elementary` are nearly the same as for `Elementary`. The first exception is the `Omega` benchmark using private tables. The slowdown in `Omega` was determined to arise from the use of `call/[2,3]` in the library predicate `filterPOA/5` (See Section 3). This use caused contention for the mutex protecting XSB's predicate table. When `filterPOA/5` was specialized to avoid `call/[2,3]` in `Omega Specialized`, the contention disappears, and the benchmark becomes scalable. The second exception to scalability is `Call Subsumption`. Executing `Call Subsumption` requires a large amount of space to be allocated for 2049 tabled calls and over $2k * 1k$ answers. While other benchmarks, such as `Elementary` also have a large number of answers, `Call Subsumption` spends nearly all of its time doing tabling operations — and memory management. Although XSB manages memory for private tables within a thread and so reduces contention for process-level memory managers, it cannot eliminate this contention. As a result, the high proportion of time spent on memory management in `Call Subsumption` reduces its scalability on `ranc/2`.

5 Discussion

We have described the approach to MT-TLP in XSB and shown how it can be used to evaluate sophisticated process and workflow formalisms in a simple and direct manner. The goals stated in Section 1 are ambitious: still, they are

largely met. Except for incremental recomputation, all the features in Table 1 are supported by private tables, while and nearly all except incremental recomputation and call subsumption are at least partially supported by shared tables. When supported, the tabled features can be almost always be made to scale linearly to the number of cores available for our benchmarking. Several existing XSB applications will benefit from the MT-TLP model as described in this paper. These include the ontology management system CDF [17], the object-logic system Flora-2 [18] and the model-checking system XMC [19]. The first two of these applications rely on tabled negation, while applications of XMC to real-time systems and security protocols rely on tabled constraints. For these and other applications, the MT-TLP model can increase availability and speed.

Acknowledgements The authors thank David S. Warren for help in implementing private tables and tabled dynamic code. Performance results we obtained on hardware supported with CITI FCTMCTES Plurianual Funding.

References

1. Wielemaker, J.: Native preemptive threads in SWI-Prolog. In: Practical Aspects of Declarative Languages. (2003) 331–345 LNCS 2916.
2. Moura, P.: Prolog multi-threading support. ISO/IEC. (5 2007) DTR 13211.
3. XSB: The XSB Programmer's Manual: Vols. 1 and 2. (2007) Available via `http://xsb.sourceforge.net`.
4. Swift, T.: Tabling for non-monotonic programming. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 201–240
5. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: ICLP. (2004)
6. Ramakrishnan, C., Ramakrishnan, I., Warren, D.S.: XcelLog: A deductive spreadsheet system. Knowledge Engineering Review **22**(3) (2007) 269–279
7. Sarna-Starosta, B.: Constraint-based Analysis of Security Protocols. PhD thesis, SUNY Stony Brook (2005)
8. Rocha, R., Silva, F., Costa, V.S.: Dynamic mixed-strategy evaluation of tabled logic programs. In: ICLP. (2005) 250264
9. Marques, R.: Concurrent Tabling: Algorithms and Implementation. PhD thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa (2007) Available from `http://asc.di.fct.unl.pt/~rfm/research.html`.
10. Freire, J., Hu, R., Swift, T., Warren, D.S.: Parallelizing tabled evaluation. In: 7th International PLILP Symposium, Springer-Verlag (1995) 115–132
11. Rocha, R., Silva, F., Costa, V.S.: On applying or-parallelism and tabling to logic programs. TPLP **5**(1 & 2) (2005) 161–205
12. Rozenberg, G., Engelfriet, J.: Elemenary net systems. In: Lectures on Petri Nets I: Basic Models. Springer LNCS 1491 (1998) 12–121
13. van der Aalst, W., ter Hofstede, A.: YAWL: Yet another workflow language (revised version). Technical report, Queensland University of Technology (2003)
14. Desel, J., Reisig, W.: Place/transition Petri nets. In: Lectures on Petri Nets I: Basic Models. Springer LNCS 1491 (1998) 122–174
15. Cui, B., Swift, T.: Preference logic grammars: Fixed-point semantics and application to data standardization. Artificial Intelligence **138** (2002) 117–147

16. Esparza, J., Nielsen, M.: Decidability issues for Petri nets. J. Inform Process. Cybernet **30**(3) (1994) 143–160
17. Swift, T., Warren, D.S.: The meaning of cold dead fish. Available via `http://www.cs.sunysb.edu/~tswift` (2003)
18. Yang, G., Kifer, M.: Flora: Implementing an efficient dood system using a tabling logic engine. In: DOOD. (2000)
19. Ramakrishnan, C., Ramakrishnan, I., Smolka, S., Dong, Y., Du, X., Roychoudhury, A., Venkatakrishnan, V.: XMC: A logic-programming-based verification toolset. In: CAV. (2000) 576–590

Declarative Combinatorics in Prolog: Shapeshifting Data Objects with Isomorphisms and Hylomorphisms

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
E-mail: tarau@cs.unt.edu

Abstract. This paper is an exploration in a logic programming framework of isomorphisms between elementary data types (natural numbers, sets, finite functions, graphs, hypergraphs) and their extension to hereditarily finite universes through *hylomorphisms* derived from *ranking/unranking* and *pairing/unpairing* operations.
An embedded higher order combinator language provides any-to-any encodings automatically.
A few examples of "free algorithms" obtained by transferring operations between data types are shown. Other applications range from stream iterators on combinatorial objects to succinct data representations and generation of random instances.
The self-contained source code of the paper, as generated from a literate Prolog program, is available at `http://logic.csci.unt.edu/tarau/research/2008/pISO.zip`
Keywords: Prolog data representations, computational mathematics, ranking/unranking, Ackermann encoding, hereditarily finite sets and functions, pairing/unpairing

1 Introduction

Data structures in imperative languages have traditionally been designed with *mutability* in mind and therefore with space saving strategies based on in-place updates. On the contrary, the dominance of *immutable* data structures in declarative languages suggests *sharing* "equivalent" immutable components as an effective space saving alternative.

Moreover, in the presence of higher order constructs, function sharing among heterogeneous data objects, is also appealing, as a way to borrow or lend "free algorithms".

The closest analogy to this, drawn from everyday thinking, is ... *analogy*. Analogical/metaphoric thinking routinely shifts entities and operations from a field to another hoping to uncover similarities in representation or use.

However, this rises the question: what guaranties do we have that doing this between data types is useful and safe?

Also sharing heterogeneous data objects faces two problems:

- some form of equivalence needs to be proven between two objects A and B before A can replace B in a data structure, a possibly tedious and error prone task
- the fast growing diversity of data types makes harder and harder to recognize sharing opportunities.

The techniques introduced in this paper provide a generic solution to these problems, through isomorphic mappings between heterogeneous data types, such that unified internal representations make equivalence checking and sharing possible. The added benefit of these "shapeshifting" data types is that the functors transporting their data content will also transport their operations, resulting in shortcuts that provide, for free, implementations of interesting algorithms. The simplest instance is the case of isomorphisms – reversible mappings that also transport operations. In their simplest form such isomorphisms show up as *encodings* – to some simpler and easier to manipulate representation – for instance natural numbers.

Such encodings can be traced back to Gödel numberings [1, 2] associated to formulae, but a wide diversity of common computer operations, ranging from wireless data transmissions to cryptographic codes qualify.

Encodings between data types provide a variety of services ranging from free iterators and random objects to data compression and succinct representations. Tasks like serialization and persistence are facilitated by simplification of reading or writing operations without the need of special purpose parsers. Sensitivity to internal data representation format or size limitations can be circumvented without extra programming effort.

2 An Embedded Data Transformation Language

It is important to organize such encodings as a flexible embedded language to accommodate any-to-any conversions without the need to write one-to-one converters. Toward this end we will organize our encodings as a group of isomorphisms within a (mildly) category theory-inspired design.

We will start by designing an embedded transformation language as a set of operations on this group of isomorphisms. We will then extend it with a set of higher order combinators mediating the composition of encodings and the transfer of operations between data types.

2.1 The Group of Isomorphisms

We implement an isomorphism between two objects X and Y as a Prolog data type (a term with functor `iso/2`) `iso(F,G)`, encapsulating a bijection F and its inverse G.

$$X \underset{g = f^{-1}}{\overset{f = g^{-1}}{\rightleftarrows}} Y$$

As a well-known mechanism to embed higher order functions in Prolog [3], we will use `iso/2` as a *closure* (higher order predicate) to be applied to an input argument and an output argument. We assume the presence of Prolog's `call/N` predicate that applies a closure to `N-1` extra arguments and `maplist/N` that applies a closure to `N-1` extra list arguments. We can organize the *group* of isomorphisms as follows.

First we define the group structure as a set of isomorphism transformers:

```
compose(iso(F,G),iso(F1,G1),iso(fcompose(F1,F),fcompose(G,G1))).
itself(iso(id,id)).
invert(iso(F,G),iso(G,F)).
```

Then, we provide evaluators for isomorphisms, that apply their left or right functions to actual arguments. Note that like `iso/2`, compose/3 is a closure to be applied to 2 extra arguments with `call/2` or `maplist/2`.

```
fcompose(G,F,X,Y):-call(F,X,Z),call(G,Z,Y).
id(X,X).
from(iso(F,_),X,Y):-call(F,X,Y).
to(iso(_,G),X,Y):-call(G,X,Y).
```

The *from* function extracts the first component (a *section* in category theory parlance) and the *to* function extracts the second component (a *retraction*) defining the isomorphism. We can now formulate *laws* about isomorphisms that can be used to test correctness of implementations.

Proposition 1 *The data type* `iso/2` *specifies a group structure, i.e. the* compose *operation is associative,* itself *acts as an identity element and* invert *computes the inverse of an isomorphism.*

It is convenient to give a name to each isomorphism as a unary predicate

```
<name>(iso(From,To)).
```

We can transport operations from an object to another with *borrow* and *lend* combinators defined as follows:

```
borrow(IsoName,H,X,Y):-call(IsoName,iso(F,G)),
  fcompose(F,fcompose(H,G),X,Y).

lend(IsoName,H,X,Y):-call(IsoName,Iso),
  invert(Iso,iso(F,G)),
  fcompose(F,fcompose(H,G),X,Y).
```

The combinators `fit` and `retrofit` just transport an object `x` through an isomorphism and apply to it an operation `op` available on the other side:

```
fit(Op,IsoName,X,Y):-
  call(IsoName,Iso),fit_iso(Op,Iso,X,Y).

fit_iso(Op,Iso,X,Y):-
  from(Iso,X,Z),call(Op,Z,Y).
```

```
retrofit(Op,IsoName,X,Y):-call(IsoName,Iso),
  retrofit_iso(Op,Iso,X,Y).

retrofit_iso(Op,Iso,X,Y):-
  to(Iso,X,Z),call(Op,Z,Y).
```

We can see the combinators `from, to, compose, itself, invert, borrow, lend, fit etc.` as part of an *embedded data transformation language*. Various examples for their use will be given as soon as we populate our universe with interesting isomorphisms.

2.2 Choosing a Root

To avoid defining $n(n-1)/2$ isomorphisms between n objects, we choose a *Root* object to/from which we will actually implement isomorphisms. We will extend our embedded combinator language using the group structure of the isomorphisms to connect any two objects through isomorphisms to/from the *Root*.

Choosing a *Root* object is somewhat arbitrary, but it makes sense to pick a representation that is relatively easy convertible to various others, efficiently implementable and, last but not least, scalable to accommodate large objects up to the runtime system's actual memory limits.

We will choose as our *Root* object *Finite Sequences of Natural Numbers*. They can be seen as as finite functions from an initial segment of Nat, say $[0..n]$, to Nat. We will represent them as lists i.e. their Prolog type is $[Nat]$. Alternatively, an array representation can be chosen. Note that in the case of a Prolog not supporting arbitrary precision integers or rationals, such lists could be used, in principle, to emulate them at source level, through the use of isomorphisms mapping them to natural numbers, signed integers and then rational numbers, following the techniques described in [4, 5].

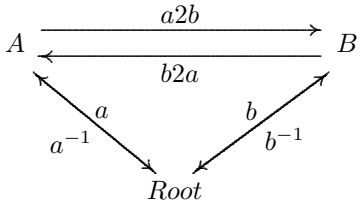
We can now define an *Encoder* as an isomorphism connecting an object to *Root* together with the combinators *with* and *as* providing an *embedded transformation language* for routing isomorphisms through two *Encoders*.

```
with(Iso1,Iso2,Iso):-invert(Iso2,Inv2),
  compose(Iso1,Inv2,Iso).

as(That,This,X,Y):-
  call(That,ThatF),call(This,ThisF),
  with(ThatF,ThisF,Iso),
  to(Iso,X,Y).
```

The combinator `with` turns two Encoders into an arbitrary isomorphism, i.e. acts as a connection hub between their domains. The combinator `as` adds a more convenient syntax such that converters between "a" and "b" can be designed as:

```
'a2b'(X,Y) :- as('a','b',X,Y).
'b2a'(X,Y) :- as('b','a',X,Y).
```



We will provide extensive use cases for these combinators as we populate our group of isomorphisms. Given that $[Nat]$ has been chosen as the root, we will define our finite function data type *fun* simply as the identity isomorphism on sequences in $[Nat]$.

```
fun(Iso) :-itself(Iso).
```

3 Extending the Group of Isomorphisms

We will now populate our group of isomorphisms with combinators based on a few primitive converters.

3.1 An Isomorphism to Finite Sets of Natural Numbers

The isomorphism is specified with two bijections `set2fun` and `fun2set`.

```
set(iso(set2fun,fun2set)).
```

While finite sets and sequences share a common representation $[Nat]$, sets are subject to the implicit constraint that all their elements are distinct[1]. This suggest that a set like $\{7,1,4,3\}$ could be represented by first ordering it as $\{1,3,4,7\}$ and then compute the differences between consecutive elements. This gives $[1,2,1,3]$, with the first element 1 followed by the increments $[2,1,3]$. To turn it into a bijection, including 0 as a possible member of a sequence, another adjustment is needed: elements in the sequence of increments should be replaced by their predecessors. This gives $[1,1,0,2]$ as implemented by `set2fun`:

```
set2fun([],[]).
set2fun([X|Xs],[X|Fs]):-
  sort([X|Xs],[_|Ys]),
  set2fun(Ys,X,Fs).

set2fun([],_,[]).
set2fun([X|Xs],Y,[A|As]):-A is (X-Y)-1,set2fun(Xs,X,As).
```

It can now be verified easily that incremental sums of the successors of numbers in such a sequence, return the original set in sorted form, as implemented by `fun2set`:

[1] Such constraints can be regarded as *laws/assertions* that we assume holding for a given data type, when needed, restricting it to the appropriate domain of the underlying mathematical concept.

```
fun2set([],[]).
fun2set([A|As],Xs):-findall(X,prefix_sum(A,As,X),Xs).

prefix_sum(A,As,R):-append(Ps,_,As),length(Ps,L),
  sumlist(Ps,S),R is A+S+L.
```

The resulting Encoder (`set`) is now ready to interoperate with another Encoder:

```
?- as(set,fun,[0, 1, 0, 0, 4],S).
S = [0, 2, 3, 4, 9].

?- as(fun,set,[0, 2, 3, 4, 9],F).
F = [0, 1, 0, 0, 4].
```

As the example shows, this encoding maps arbitrary lists of natural numbers representing finite functions to strictly increasing sequences of natural numbers representing sets.

3.2 Folding Sets into Natural Numbers

We can fold a set, represented as a list of distinct natural numbers into a single natural number, reversibly, by observing that it can be seen as the list of exponents of 2 in the number's base 2 representation.

```
nat_set(iso(nat2set,set2nat)).

nat2set(N,Xs):-nat2elements(N,Xs,0).

nat2elements(0,[],_K).
nat2elements(N,NewEs,K1):-N>0,
  B is /\(N,1),N1 is N>>1,K2 is K1+1,
  add_el(B,K1,Es,NewEs),
  nat2elements(N1,Es,K2).

add_el(0,_,Es,Es).
add_el(1,K,Es,[K|Es]).

set2nat(Xs,N):-set2nat(Xs,0,N).

set2nat([],R,R).
set2nat([X|Xs],R1,Rn):-R2 is R1+(1<<X),set2nat(Xs,R2,Rn).
```

We will standardize this pair of operations as an *Encoder* for a natural number using our Root as a mediator:

```
nat(Iso):-nat_set(NatSet),set(Set),compose(NatSet,Set,Iso).
```

The resulting Encoder (`nat`) is now ready to interoperate with any other Encoder:

```
?- as(fun,nat,42,F).
F = [1, 1, 1]
```

```
?- as(set,nat,42,F).
F = [1, 3, 5]

?- as(fun,nat,2008,F).
F = [3, 0, 1, 0, 0, 0, 0]

?- as(set,nat,2008,S).
S = [3, 4, 6, 7, 8, 9, 10]

?- lend(nat,reverse,2008,R).
R = 1135 % different, sequence depends on order

?- lend(nat_set,reverse,2008,R).
R = 2008 % same, set is order independent

?- as(set,nat,42,S).
S = [1, 3, 5]

?- fit(length,nat,42,L).
L = 3

?- retrofit(succ,nat_set,[1,3,5],N).
N = 43
```

The reader might notice at this point that we have already made full circle - as finite sets can be seen as instances of finite sequences. Injective functions that are not surjections with wider and wider gaps can be generated using the fact that one of the representations is information theoretically "denser" than the other, for a given range:

```
?- as(set,fun,[0,1,2,3],S1).
S1 = [0, 2, 5, 9].

?- as(set,fun,[0,2,5,9],S2).
S2 = [0, 3, 9, 19].

?- as(set,fun,[0,3,9,19],S3).
S3 = [0, 4, 14, 34].
```

4 Generic Unranking and Ranking Hylomorphisms

The *ranking problem* for a family of combinatorial objects is finding a unique natural number associated to it, called its *rank*. The inverse *unranking problem* consists of generating a unique combinatorial object associated to each natural number.

4.1 Pure Hereditarily Finite Data Types

The unranking operation is seen here as an instance of a generic *anamorphism* mechanism (an *unfold* operation), while the ranking operation is seen as an instance of the corresponding catamorphism (a *fold* operation) [6, 7]. Together they form a mixed transformation called *hylomorphism*.

We will use such hylomorphisms to lift isomorphisms between lists and natural numbers to isomorphisms between a derived "self-similar" tree data type and natural numbers. In particular we will derive Ackermann's encoding from Hereditarily Finite Sets to Natural Numbers.

The data type T representing hereditarily finite structures will be a generic multiway tree with a single leaf type `[]`.

The two sides of our hylomorphism are parameterized by two transformations `f` and `g` forming an isomorphism `Iso f g`:

```
unrank(F,N,R):-call(F,N,Y),unranks(F,Y,R).
unranks(F,Ns,Rs):-maplist(unrank(F),Ns,Rs).

rank(G,Ts,Rs):-ranks(G,Ts,Xs),call(G,Xs,Rs).
ranks(G,Ts,Rs):-maplist(rank(G),Ts,Rs).
```

Both combinators can be seen as a form of "structured recursion" that propagate a simpler operation guided by the structure of the data type. For instance, the size of a tree of type T is obtained as:

```
tsize1(Xs,N):-sumlist(Xs,S),N is S+1.

tsize(T,N) :- rank(tsize1,T,N).
```

Note also that `unrank` and `rank` work on trees in cooperation with `unranks` and `ranks` working on lists of trees.

We can now combine an anamorphism+catamorphism pair into an isomorphism `hylo` defined with `rank` and `unrank` on the corresponding hereditarily finite data types:

```
hylo(IsoName,iso(rank(G),unrank(F))):-call(IsoName,iso(F,G)).

hylos(IsoName,iso(ranks(G),unranks(F))):-call(IsoName,iso(F,G)).
```

Hereditarily Finite Sets Hereditarily Finite Sets will be represented as an Encoder for the tree type `T`:

```
hfs(Iso):-hylo(nat_set,Hylo),nat(Nat),
  compose(Hylo,Nat,Iso).
```

The `hfs` Encoder can now borrow operations from sets or natural numbers as follows:

```
hfs_succ(H,R):-borrow(nat_hfs,succ,H,R).
nat_hfs(Iso):-nat(Nat),hfs(HFS),with(Nat,HFS,Iso).
```

```
?- hfs_succ([],R).
R = [[]] ;
```

Otherwise, hylomorphism induced isomorphisms work as usual with our embedded transformation language:

```
?- as(hfs,nat,42,H).
H = [[[]], [[], [[]]], [[], [[[]]]]]
```

One can notice that we have just derived as a "free algorithm" Ackermann's encoding [8, 9], from Hereditarily Finite Sets to Natural Numbers:

$$f(x) = \texttt{if } x = \{\} \texttt{ then } 0 \texttt{ else } \sum_{a \in x} 2^{f(a)}$$

together with its inverse:

```
ackermann(N,H):-as(nat,hfs,N,H).
inverse_ackermann(H,N):-as(hfs,nat,H,N).
```

Hereditarily Finite Functions The same tree data type can host a hylomorphism derived from finite functions instead of finite sets:

```
hff(Iso) :-
  hylo(nat,Hylo),nat(Nat),
  compose(Hylo,Nat,Iso).
```

The `hff` Encoder can be seen as another "free algorithm", providing data compression/succinct representation for Hereditarily Finite Sets. Note, for instance, the significantly smaller tree size in:

```
?- as(hff,nat,42,H).
H = [[[]], [[]], [[]]]
```

As the cognoscenti might observe this is explained by the fact that `hff` provides higher information density than `hfs`, by incorporating order information that matters in the case of sequence and is ignored in the case of a set.

5 Pairing/Unpairing

A *pairing* function is an isomorphism $f : Nat \times Nat \to Nat$. Its inverse is called *unpairing*.

We will introduce here an unusually simple pairing function (also mentioned in [10], p.142).

The function `bitpair` works by splitting a number's big endian bitstring representation into odd and even bits.

```
bitpair(p(I,J),P):-
  evens(I,Es),odds(J,Os),
  append(Es,Os,Ps),set2nat(Ps,P).

evens(X,Es):-nat2set(X,Ns),maplist(double,Ns,Es).
```

```
odds(X,Os):-evens(X,Es),maplist(succ,Es,Os).
double(N,D):-D is 2*N.
```

The inverse function `bitunpair` blends the odd and even bits back together.

```
bitunpair(N,p(E,O)):-nat2set(N,Ns),
  split_evens_odds(Ns,Es,Os),
  set2nat(Es,E),set2nat(Os,O).

split_evens_odds([],[],[]).
split_evens_odds([X|Xs],[E|Es],Os):-
  X mod 2 =:= 0,E is X // 2,
  split_evens_odds(Xs,Es,Os).
split_evens_odds([X|Xs],Es,[O|Os]):-
  X mod 2 =:= 1,O is X // 2,
  split_evens_odds(Xs,Es,Os).
```

The transformation of the bitlists is shown in the following example with bitstrings aligned:

```
?-bitunpair(2008,R)
R = p(60,26)

% 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
%   60:[      0,    1,    1,    1,    1]
%   26:[   0,    1,    0,    1,    1   ]
```

We can derive the following Encoder:

```
nat2(Iso):-nat(Nat),
  compose(iso(bitpair,bitunpair),Nat,Iso).
```

working as follows:

```
?- as(nat2,nat,2008,Pair).
Pair = p(60, 26)

?- as(nat,nat2,p(60,26),N).
N = 2008
```

6 Directed Graphs and Hypergraphs

We will now show that more complex data types like digraphs and hypergraphs have extremely simple encoders. This shows once more the importance of compositionality in the design of our embedded transformation language.

6.1 Encoding Directed Graphs

We can find a bijection from directed graphs (with no isolated vertices, corresponding to their view as binary relations), to finite sets by fusing their list of ordered pair representation into finite sets with a pairing function:

```
digraph2set(Ps,Ns) :- maplist(bitpair,Ps,Ns).
set2digraph(Ns,Ps) :- maplist(bitunpair,Ns,Ps).
```

The resulting Encoder is:

```
digraph(Iso):-set(Set),
  compose(iso(digraph2set,set2digraph),Set,Iso).
```

working as follows:

```
?- as(digraph,nat,2008,D),as(nat,digraph,D,N).
D = [p(1, 1), p(2, 0), p(2, 1), p(3, 1), p(0, 2), p(1, 2), p(0, 3)],
N = 2008
```

6.2 Encoding Hypergraphs

Definition 1 *A hypergraph (also called* set system*) is a pair $H = (X, E)$ where X is a set and E is a set of non-empty subsets of X.*

We can easily derive a bijective encoding of *hypergraphs*, represented as sets of sets:

```
set2hypergraph(S,G) :- maplist(nat2set,S,G).
hypergraph2set(G,S) :- maplist(set2nat,G,S).
```

The resulting Encoder is:

```
hypergraph(Iso):-set(Set),
  compose(iso(hypergraph2set,set2hypergraph),Set,Iso).
```

working as follows

```
?- as(hypergraph,nat,2008,G),as(nat,hypergraph,G,N).
G = [[0, 1], [2], [1, 2], [0, 1, 2], [3], [0, 3], [1, 3]],
N = 2008
```

7 Applications

Besides their utility as a uniform basis for a general purpose data conversion library, let us point out some specific applications of our isomorphisms.

7.1 Combinatorial Generation

A free combinatorial generation algorithm (providing a constructive proof of recursive enumerability) for a given structure is obtained simply through an isomorphism from *nat*:

```
nth(Thing,N,X) :- as(Thing,nat,N,X).

stream_of(Thing,X) :- nat_stream(N),nth(Thing,N,X).

nat_stream(0).
nat_stream(N):-nat_stream(N1),succ(N1,N).
```

```
?- nth(set,42,S).
S = [1, 3, 5]

?- stream_of(hfs,H).
H = [] ;
H = [[]] ;
H = [[[]]] ;
H = [[], [[]]] ;
H = [[[[]]]] ;
H = [[], [[[]]]] ;
...
```

7.2 Random Generation

Combining `nth` with a random generator for *nat* provides free algorithms for random generation of complex objects of customizable size:

```
random_gen(Thing,Max,Len,X):-
  random_fun(Max,Len,Ns),
  as(Thing,fun,Ns,X).

random_fun(Max,Len,Ns):-
  length(Ns,Len),
  maplist(random_nat(Max),Ns).

random_nat(Max,N):-random(X),N is integer(Max*X).

?- random_gen(set,100,4,R).
R = [16, 39, 118, 168].

?- random_gen(fun,100,4,R).
R = [92, 60, 47, 76].

?- random_gen(nat,100,4,R).
R = 269599466671506412912446917138642189142104131263755679205821010411 52.

?- random_gen(hfs,4,3,R).
R = [[[]], [[], [[[]]]], [[[]], [[], [[]]]]]

?- random_gen(hff,4,3,R).
R = [[], [], [[]]]
```

Besides providing arbitrary precision random numbers as a "free algorithm" on top of a builtin limited precision floating point generator, one can see that this technique can be used to implement elegantly random test generators in tools like QuickCheck [11] without having to write data structure specific scripts.

7.3 Succinct Representations

Depending on the information theoretical density of various data representations as well as on the constant factors involved in various data structures, significant

data compression can be achieved by choosing an alternate isomorphic representation, as shown in the following examples:

```
?- as(hff,hfs,[[[]], [[], [[]]], [[], [[[]]]]],HFF).
HFF = [[[]], [[]], [[]]]

?- as(nat,hff,[[[]], [[]], [[]]],N).
N = 42
```

In particular, mapping to efficient arbitrary length integer implementations (usually C-based libraries), can provide more compact representations or improved performance for isomorphic higher level data representations. We can compare representations sharing a common datatype to conjecture about their asymptotic information density.

7.4 Experimental Mathematics

For instance, after defining:

```
length_as(Thing,X,Len) :-nat(Nat),
  call(Thing,T),with(Nat,T,Iso),
  fit_iso(length,Iso,X,Len).

sum_as(Thing,X,Len) :-nat(Nat),
  call(Thing,T),with(Nat,T,Iso),
  fit_iso(sumlist,Iso,X,Len).

size_as(Thing,X,Len) :-nat(Nat),
  call(Thing,T),with(Nat,T,Iso),
  fit_iso(tsize,Iso,X,Len).
```

one can conjecture that finite functions are more compact than sets asymptotically

```
?- length_as(fun,12345678901234567890123456789O,L).
L = 54

?- length_as(set,12345678901234567890123456789O,L).
L = 54

?- length_as(fun,12345678901234567890123456789O,L).
L = 54

?- sum_as(set,12345678901234567890123456789O,L).
L = 2690

?- sum_as(fun,12345678901234567890123456789O,L).
L = 43
```

and then observe that the same trend applies also to their hereditarily finite derivatives:

```
?- size_as(hfs,12345678901234567890123456789O,L).
L = 627

?- size_as(hff,12345678901234567890123456789O,L).
L = 91
```

7.5 A surprising "free algorithm": strange_sort

A simple isomorphism like `nat_set` can exhibit interesting properties as a building block of more intricate mappings like Ackermann's encoding, but let's also note a (surprising to us) "free algorithm" – sorting a list of distinct elements without explicit use of comparison operations:

```
strange_sort(Unsorted,Sorted):-
  nat_set(Iso),
  to(Iso,Unsorted,Ns),
  from(Iso,Ns,Sorted).


?- strange_sort([2,9,3,1,5,0,7,4,8,6],Sorted).
Sorted = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This algorithm emerges as a consequence of the commutativity of addition and the unicity of the decomposition of a natural number as a sum of powers of 2. The cognoscenti might notice that such surprises are not totally unexpected. In a functional programming context, they go back as early as Wadler's Free Theorems [12].

7.6 Other Applications

A fairly large number of useful algorithms in fields ranging from data compression, coding theory and cryptography to compilers, circuit design and computational complexity involve bijective functions between heterogeneous data types. Their systematic encapsulation in a generic API that coexists well with strong typing can bring significant simplifications to various software modules with the added benefits of reliability and easier maintenance. In a Genetic Programming context [13] the use of isomorphisms between bitvectors/natural numbers on one side, and trees/graphs representing HFSs, HFFs on the other side, looks like a promising phenotype-genotype connection. Mutations and crossovers in a data type close to the problem domain are transparently mapped to numerical domains where evaluation functions can be computed easily. In the context of Software Transaction Memory implementations (like Haskell's STM [14]), encodings through isomorphisms are subject to efficient shortcuts, as undo operations in case of transaction failure can be performed by applying inverse transformations without the need to save the intermediate chain of data structures involved.

8 Related work

This work can be seen as part of a larger effort to cover in a declarative programming paradigm some fundamental combinatorial generation algorithms along the lines of Donald Knuth's recent work [15].

The closest reference on encapsulating bijections as a data type is [16] and Connan Eliot's composable bijections Haskell module [17], where, in a more complex setting, Arrows [18] are used as the underlying abstractions. While our `Iso` data type is similar to the *Bij* data type in [17] and BiArrow concept of [16], the techniques for using such isomorphisms as building blocks of an embedded composition language centered around encodings as Natural Numbers are new.

Ranking functions can be traced back to Gödel numberings [1, 2] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [19, 15, 20, 21]. However the generic view of such transformations as hylomorphisms obtained compositionally from simpler isomorphisms, as described in this paper, is new.

Natural Number encodings of Hereditarily Finite Sets have triggered the interest of researchers in fields ranging from Axiomatic Set Theory and Foundations of Logic to Complexity Theory and Combinatorics [22–27]. Computational and Data Representation aspects of Finite Set Theory have been described in logic programming and theorem proving contexts in [9, 28].

Pairing functions have been used in work on decision problems as early as [29, 30]. A typical use in the foundations of mathematics is [31]. An extensive study of various pairing functions and their computational properties is presented in [32].

9 Conclusion

We have shown the expressiveness of Prolog as a metalanguage for executable mathematics, by describing encodings for functions and finite sets in a uniform framework as data type isomorphisms with a group structure. Prolog's higher order predicates and recursion patterns have helped the design of an embedded data transformation language. Using higher order combinators a simplified random testing mechanism has been implemented as an empirical correctness test. The framework has been extended with hylomorphisms providing generic mechanisms for encoding Hereditarily Finite Sets and Hereditarily Finite Functions. In the process, a few surprising "free algorithms" have emerged, including Ackermann's encoding from Hereditarily Finite Sets to natural numbers. We plan to explore in depth in the near future, some of the results that are likely to be of interest in fields ranging from combinatorics to data compression and arbitrary precision numerical computations. While we have not explicitly provided a complexity analysis for various isomorphisms, it is clear from the actual code that our transformations typically work in time and space proportional to the overall size of the representation. In particular, when natural numbers are the source or the target, complexity is $O(log(N))$, given that $log(N)$ is the bitsize of the representation of N.

10 Acknowledgment

The author thanks the anonymous reviewers of CICLOPS'08 for their constructive criticism, substantial comments and suggestions.

References

1. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
2. Hartmanis, J., Baker, T.P.: On simple goedel numberings and translations. In Loeckx, J., ed.: ICALP. Volume 14 of Lecture Notes in Computer Science., Springer (1974) 301–316
3. Warren, D.H.D.: Higher-order extensions to Prolog – are they needed? In Michie, D., Hayes, J., Pao, Y.H., eds.: Machine Intelligence 10. Ellis Horwood (1981)
4. Tarau, P.: Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell (2008) http://arXiv.org/abs/0808.2953.
5. Gibbons, J., Lester, D., Bird, R.: Enumerating the rationals. Journal of Functional Programming **16**(4) (2006)
6. Hutton, G.: A Tutorial on the Universality and Expressiveness of Fold. J. Funct. Program. **9**(4) (1999) 355–372
7. Meijer, E., Hutton, G.: Bananas in Space: Extending Fold and Unfold to Exponential Types. In: FPCA. (1995) 324–333
8. Ackermann, W.F.: Die Widerspruchsfreiheit der allgemeinen Mengenlhere. Mathematische Annalen (114) (1937) 305–315
9. Piazza, C., Policriti, A.: Ackermann Encoding, Bisimulations, and OBDDs. TPLP **4**(5-6) (2004) 695–718
10. Pigeon, S.: Contributions à la compression de données. Ph.d. thesis, Université de Montréal, Montréal (2001)
11. Claessen, K., Hughes, J.: Testing monadic code with quickcheck. SIGPLAN Notices **37**(12) (2002) 47–59
12. Wadler, P.: Theorems for free! In: FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture, New York, NY, USA, ACM (1989) 347–359
13. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
14. Harris, T., Marlow, S., Jones, S.L.P., Herlihy, M.: Composable memory transactions. Commun. ACM **51**(8) (2008) 91–100
15. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
16. Alimarine, A., Smetsers, S., van Weelden, A., van Eekelen, M., Plasmeijer, R.: There and back again: arrows for invertible programming. In: Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM Press (2005) 86–97
17. Connan Eliot: Data.Bijections Haskell Module. http://haskell.org/haskellwiki/TypeCompose.
18. Hughes, J.: Generalizing Monads to Arrows Science of Computer Programming 37, pp. 67-111, May 2000.

19. Martinez, C., Molinero, X.: Generic algorithms for the generation of combinatorial objects. In Rovan, B., Vojtas, P., eds.: MFCS. Volume 2747 of Lecture Notes in Computer Science., Springer (2003) 572–581
20. Ruskey, F., Proskurowski, A.: Generating binary trees by transpositions. J. Algorithms **11** (1990) 68–84
21. Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Information Processing Letters **79** (2001) 281–284
22. Takahashi, M.o.: A Foundation of Finite Mathematics. Publ. Res. Inst. Math. Sci. **12**(3) (1976) 577–708
23. Kaye, R., Wong, T.L.: On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic Volume **48**(4) (2007) 497–510
24. Abian, A., Lamacchia, S.: On the consistency and independence of some set-theoretical constructs. Notre Dame Journal of Formal Logic **X1X**(1) (1978) 155–158
25. Avigad, J.: The Combinatorics of Propositional Provability. In: ASL Winter Meeting, San Diego (January 1997)
26. Kirby, L.: Addition and multiplication of sets. Math. Log. Q. **53**(1) (2007) 52–65
27. Leontjev, A., Sazonov, V.Y.: Capturing LOGSPACE over Hereditarily-Finite Sets. In Schewe, K.D., Thalheim, B., eds.: FoIKS. Volume 1762 of Lecture Notes in Computer Science., Springer (2000) 156–175
28. Paulson, L.C.: A Concrete Final Coalgebra Theorem for ZF Set Theory. In Dybjer, P., Nordström, B., Smith, J.M., eds.: TYPES. Volume 996 of Lecture Notes in Computer Science., Springer (1994) 120–139
29. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
30. Robinson, J.: Finite generation of recursively enumerable sets. Proceedings of the American Mathematical Society **19**(6) (dec 1968) 1480–1486
31. Cégielski, P., Richard, D.: Decidability of the theory of the natural integers with the cantor pairing function and the successor. Theor. Comput. Sci. **257**(1-2) (2001) 51–77
32. Rosenberg, A.L.: Efficient pairing functions - and why you should care. International Journal of Foundations of Computer Science **14**(1) (2003) 3–17

Precise Garbage Collection in Prolog

Jan Wielemaker[1] and Ulrich Neumerkel[2]

[1] Universiteit van Amsterdam, The Netherlands
J.Wielemaker@uva.nl
[2] Technische Universität Wien, Austria
ulrich@complang.tuwien.ac.at

Abstract. In this paper we present a series of tiny programs that verify that a Prolog heap garbage collector can find specific forms of garbage. Only 2 out of our tested 7 Prolog systems pass all tests. Comparing memory usage on realistic programs dealing with finite datastructures using both poor and precise garbage collection shows only a small difference, providing a plausible explanation why many Prolog implementors did not pay much attention to this issue. Attributed variables allow for creating infinite lazy datastructures. We prove that such datastructures have great practical value and their introduction requires 'precise' garbage collection. The Prolog community knows about three techniques to reach at precise garbage collection. We summarise these techniques and provide more details on scanning virtual machine instructions to infer reachability in a case study.

1 Introduction

All modern Prolog systems come with a heap garbage collector, no longer limiting the programmer to revert to failure driven loops or **findall/3** to free unneeded memory through backtracking. For this article, we define a 'precise' garbage collector as a garbage collector that reclaims all data that can no longer be reached considering all possible execution paths from the current state without considering semantics. I.e. in `1==2, A=ok`, A is unreachable due to the semantics of ==/2, but we consider all parts of a conjunction reachable and therefore A is considered reachable. Our survey of 7 popular Prolog systems (Sect. 3) reveals that only two satisfy this definition. We compared the memory requirements between the poorest and best performance of GC on 5 very different real-world programs (Tab. 2). The comparison indicates that precise GC is unimportant for many programs, which provides a plausible explanation why precise GC is not widespread.

Precise GC becomes important for processing infinite datastructures, in this case distinguished from *cyclic* structures. A truly infinite structure clearly never fits into finite physical memory. We are concerned with datastructures that grow due to further instantiation while (older) parts of the datastructure become unreachable after processing and can be reclaimed by the garbage collector. A typical example is processing input using a list: the list is expanded as new

input becomes available, while the head of the list becomes unreachable after being processed deterministically. This approach is used in [1], where infinite lists are used for communication between concurrent processes. Similar consideration motivated improvements in functional languages [2].

This article is organised as follows. First, in Sect. 2 we make a case for the practical value of infinite lazy datastructures and the requirement of precise GC. In Sect. 3 we identify possible leaks and test 7 Prolog implementations for them, 5 of which exhibit two or more leaks. This is followed by a survey of known existing techniques to reach precise GC and the description and evaluation of a case study adding precise GC to SWI-Prolog.[3]

2 A case for infinite lazy datastructures: pure input

Prolog DCG and other parsing techniques are based on processing lists. Unfortunately, the data that needs to be parsed is often provided as a Prolog stream that accesses data from the outside world. This problem has been identified long ago and many implementations of DCG provide a hook 'C'/3 to read an input character. This hook is of little practical use, notably due to the poor combination of non-determinism and side-effects. The current proposal for an ISO standard on DCGs [3] no longer mentions 'C'/3. Fortunately, extended unification [4–7] using attributed variables as found in many modern Prolog systems provides a straightforward mechanism to remedy this problem.

Figure 1 presents the simple algorithm to apply a grammar rule on input from a file as it appears in the SWI-Prolog library `pure_input.pl`. Besides standard ISO predicates, the implementation depends on **freeze**(*Var, Goal*), which delays *Goal* until *Var* becomes instantiated (coroutining); **call_cleanup**(*Goal, Cleanup*) which allows for closing the input handle when *Goal* becomes inaccessible due to deterministic termination, an exception or pruning of a choicepoint and finally **read_pending_input**(*Handle, Head, Tail*) which reads a block of buffered input into the difference-list `Head\Tail`. Freeze or a substitute is available in all systems with attributed variables. Call_cleanup is available in multiple Prolog implementations and has been discussed for inclusion in the upcoming revision of Part I of the ISO Prolog standard.[4] A block-read operation is not defined by the ISO standard but trivial to implement while it provides a very significant speedup (12× in SWI-Prolog 5.6.59) because it only needs to validate and lock the stream handle once.

The **phrase_from_file**(*:DCG, +File*) definition in Fig. 1 allows for applying an arbitrary non-deterministic DCG completely transparently on the content of a file while, given precise GC, the memory usage is independent from the size of this file. We compared the use of a DCG on a file with a carefully handcrafted program to count words in a text-file. We summarise the key results in the table below and conclude that the DCG version is much easier to read and very comparable in performance.

[3] http://www.swi-prolog.org

[4] Inclusion is stalled because the precise semantics prove hard to describe.

```
read_to_input_stream(Handle, Pos1, Stream0) :-
    set_stream_position(Handle, Pos1),
    (   at_end_of_stream(Handle)
    ->  Stream0 = []
    ;   read_pending_input(Handle, Stream0, Stream1),
        stream_property(Handle, position(Pos2)),
        freeze(Stream1, read_to_input_stream(Handle, Pos2, Stream1))
    ).

phrase_from_file(Phrase, File) :-
    open(File, read, Handle),
    stream_property(Handle, position(Pos)),
    freeze(Stream, read_to_input_stream(Handle, Pos, Stream)),
    call_cleanup(phrase(Phrase, Stream), close(Handle)).
```

Fig. 1. Implementation of input streams.

	traditional	**DCG on file**
Code size (lines)	31	22
Time (sec., 25MB file)	16.1	17.1
GC time (sec.)	0.9	1.4

From the above, we conclude that infinite (lazy) terms have great practical value and it is therefore desirable that garbage collection is capable of reclaiming the no-longer-accessible part of the term.

3 State of the art

Can pure input as described above be used in current Prolog systems with coroutining? We reviewed 7 Prolog implementations. The first obvious requirement is that there is no memory leak after a deterministic wakeup of a delayed goal (Sect. 3.1). The other requirements are about reclaiming unneeded parts of the input list within and-control and or-control. I.e. we must be able to create a list of arbitrary size if there are no references to the entire list. The simplest form is the test below. Predicate **f/1** builds a list, but as nobody uses it, GC reclaims it and **run/0** runs forever in constant space.

```
run :- f(_).
f([f|X]) :- f(X).
```

This is the simplest case, where the initial list is created through a singleton variable. In WAM-based systems with registers, the list resides in a register that is overwritten in each recursion. On virtual machines such as the ZIP [8, 9] and ATOAM [10] that pass arguments over the stack, last-call optimization overwrites the arguments, making the head inaccessible.

We will now go systematically through requirements to deal with infinite (lazy) datastructures. The first property validates that deterministic instantiation of an attributed variable does not leak. The remaining properties validate that various scenarios where the head of the list becomes inaccessible are detected by the garbage collector. Each test case considers a situation that requires special attention in one or more virtual machines, based on our understanding of, notably, the WAM and ZIP. As the number of possible virtual machines is unbounded, it is not possible to be sure that these cases cover all cases in all possible virtual machines. Each property is accompanied by a program that must run forever in constant space. A test is considered 'failed' if the system aborts or memory usage exceeds 1Gb. The given programs are very simple, using a fact **dummy/1** to pretend access to a variable. We assume that **dummy/1** cannot be optimized away by the compiler, otherwise a more complex replacement is needed.

3.1 Property 1: Permanent removal of attributes

Attributed variables that have been unified deterministically with a non-variable term must be reclaimed completely. This property can be tested using the program below. It creates delayed goals and executes them through deterministic binding. Note that for most constraint solvers, complete reclamation of attributed variables is not strictly necessary. Most CLP(FD) programs are concerned with finding solutions nondeterministically via a labeling procedure, thus most volatility stems from backtracking and not from forward recursion.

```
run :- run(_).
run(X) :- freeze(X, dummy(X)), X = 1, run(T).
dummy(_).
```

3.2 Property 2: And-control (head variables)

Variables appearing in the head of a rule and in the body must be discarded as soon as possible. We test this using the following which, like the previous test, must run forever in bounded memory. The call to **dummy/2** ensures $L0$ is not made inaccessible due to last call optimization.

```
run :- run(_,_).
run(L0, L) :- f(L0, L1), dummy(L1, L).
f([g|X], Y) :- f(X, Y).
dummy(Xs, Xs).
```

3.3 Property 3: And-control (existential variables)

Existential variables that occur in several goals, but not the last one. Ideally such variables should be covered by environment trimming [11] in the WAM. Careful environment trimming avoids more complex treatment.

```
run :- run(_,_).
run(L0, L) :- dummy(L0, L1), f(L1, L2), dummy(L2, L).
f([f|X], Y) :- f(X, Y).
dummy(Xs, Xs).
```

3.4 Property 4: Or-control

Or-control covers the case where a variable is only accessible from a choicepoint. A well behaved garbage collector will reset such variables and discard their current value (early-reset, [12]). This situation arises in disjunctions in grammars. E.g. `(...,"a"|...,"b")`, where .../0 is defined to match an unbounded string.

```
run :- run(_).
run(X) :- f(X).
run(X) :- X == [].
f([f|X]) :- f(X).
```

3.5 Property 5: Branching inside a clause

Branching (*A*;*B* and *If*`->`*Then*`;`*Else*) using different ordering of the variables in both branches cannot be handled optimally with the WAM environment trimming as the branches require different environment layout. This test is only of interest for systems that open code disjunctions, avoiding an auxiliary internal definition.

```
run(Z) :- p(_,_,Z).
p(X,Y,Z) :- (Z > 0 ->  f(X), g(Y), dummy ; g(Y), f(X), dummy).
f([f|X]) :- f(X).
g([g|X]) :- g(X).
dummy.
```

3.6 Conclusion from our survey

	1	2	3	4	5_0	5_1	VM
SICStus 3.12.5	ok	ok	ok	ok	ok	ok	WAM
Ciao 1.10p8	ok	ok	ok	ok	ok	ok	WAM
YAP 5.0.1	n	ok	ok	ok	ok	n	WAM
ECLiPSe 5.10	ok	ok	n	ok	n	n	WAM
SWI 5.6.54	n	n	n	n	n	n	ZIP
BProlog 7.1	n	ok	n	n	n	n	ATOAM
XSB 3.1	n	ok	n	n	n	n	WAM

Table 1. Evaluation of GC in some popular Prolog systems with coroutining. The numbers correspond to the properties. Property 5 is tested for both branches.

In the above sections we have provided tests for the main properties of Prolog coroutining and garbage collection needed to be able deal with infinite lazy datastructures. The results are shown in Tab. 1. As detailed descriptions of GC in these systems is either not in the literature or the description is likely to be outdated and we do not have access to the source code of all these systems we have not examined *why* tests succeed or fail. We merely conclude that precise GC has not been given much attention by the respective developers. Table 2 justifies this behaviour in the absence of infinite datastructures.

To the best of our knowledge, SICStus[5] and the derived Ciao system [1] reach a precise result using WAM registers, environment trimming and the implementation of in-clause alternative execution paths using anonymous predicates. The YAP VM uses virtual machine instructions for in-clause alternative execution paths, which cannot be handled perfectly with only environment trimming as explained in Sect. 4. It is hard to explain the behaviour of the other systems.

Our study started with providing a pure input library for SWI-Prolog. SWI-Prolog *design* was ok for property 1, but the implementation was proven flawed. As the SWI-Prolog virtual machine passes arguments over the stack and does not use environment trimming, it failed on all test.

4 Related work on data reachability in Prolog

Prolog systems discard data during backtracking. During forwards execution, discarding data is achieved by the heap garbage collector. The garbage collector preserves all data that is accessible through a set of *root pointers* [13]. The precise set of root pointers depends on the Virtual Machine (VM) architecture, where we distinguish between VMs that pass arguments in registers (WAM) and VMs that pass arguments using the stack (ZIP, ATOAM). The current stack frame and choice point are always root pointers. Registers and global variables are other examples. There are several mechanisms by which data becomes inaccessible from the set of root pointers that are part of the normal Prolog (forward) execution:

- Temporary variables allocated in registers become inaccessible when they are overwritten.
- Arguments (on machines passing arguments over the stack) and environment slots become inaccessible if the frame is discarded due to last-call optimization.
- Environment trimming (see below) shrinks the environment, discarding unneeded parts as the execution of the clause progresses.

Environment trimming [11] allocates variables in the environments ordered by the last subgoal that references the variable. Each call to a subgoal has an additional numeric argument that states that the first N variables of the

[5] www.sics.se/sicstus/ explained to one of the authors by Mats Carlsson.

environment are still valid. Together with registers for argument passing and last-call optimization, environment trimming reaches a precise result if there are *no alternative execution paths* in the VM instructions. This implies that disjunction (*A*;*B*) and *If*`->`*Then*`;`*Else* must be translated into pure (anonymous) predicates with some additional machinery to deal with proper scoping of the cut. This technique is used by SICStus Prolog and Ciao (see Sect. 3.6.

Many virtual machines realise disjunction and if-then-else using branch instructions in the VM. As different subgoal ordering in the alternate execution paths may require different ordering of variables in the environment (Sect. 3.5), there is no longer a perfect order and garbage collection that scans the entire environment will mark data that is no longer reachable because there is no instruction that refers to some variable. Table 1 suggests this is the status in YAP 5.0.1.

Environment trimming cannot deal with arguments that are passed over the stack as their order is determined by the calling convention and, analogous to in-clause branching, different clauses of the predicate generally require a different ordering.

VMs that pass arguments over the stack as well as VMs that use branching instructions to code in-clause alternate execution paths need additional measures to regain precise GC. Two techniques to achieve this have been part of the Prolog folklore for some time.[6] One scans the VM instructions from the continuation points to find the accessible variables. It was used by old versions of BIM Prolog. With native code this became very hard to maintain. The other uses compiler generated bitmaps for each possible continuation point that represent all reachable variables. This is used by BIM Prolog and hProlog.

In systems based on 'Binary Prolog' [14], continuations take the place of environments. They are represented by ordinary Prolog terms and therefore profit from the same data representations [15]. Garbage collection in such systems [16] do not require any special treatment. On the other hand, Binary Prolog requires more space for representing variables within continuations than traditional implementations. Every occurrence of a variable is now represented separately, while traditional environments represent each variable only once.

5 Our case study: SWI-Prolog

SWI-Prolog is based on the ZIP VM which passes arguments over the stack and uses branching instructions inside a clause. Like most today's Prolog systems, the VM is emulated. We briefly examine these properties under the assumption that the optimal choice depends on the specific setting: desired performance, portability, transparency for debugging, simplicity and speed of the compiler.

- *Argument passing*
  The use of registers for argument passing as the WAM has some clear advantages. It keeps the environment small and simplifies last-call optimization.

[6] according to Bart Demoen

This comes at a price: the compiler is more complicated and it is harder to provide a (graphical) debugger that provides access to variables in the parent frames.

- *Branching instructions*
  Using branching instructions to code disjunction and if-then-else prohibits precise trimming of the environment as we have seen in Sect. 4. On the other hand, execution is generally faster as no environment needs to be created for the anonymous predicates that otherwise replace different in-clause execution paths. We have no information on the implementation effort associated with these approaches.
- *Emulated VM vs. native code*
  An emulated VM is clearly easier to implement and if the VM is written in a portable language, portability of the system comes for free. In addition, it allows for simple decompilation [17] and simplifies two tasks in GC: identify not-yet-initialized variables in the environment and identify variables that can still be accessed from a given program counter (PC) location. SICStus has dropped native code in release 4[7]

The above observations make it clear that scanning VM instructions to remedy the reachability problem is the most obvious approach for SWI-Prolog. Because most todays Prolog implementation use an emulated VM and Tab. 1 proves that several systems still need to realise precise GC we believe a description of our case study will help persuading other implementors to implement precise GC and will help them to take the correct decisions right away.

6 Implementation

The SWI-Prolog VM differs considerable from the much more widely adopted WAM. SWI-Prolog's garbage collector however closely follows the SICStus Prolog garbage collector, which is described excellently in [12]. The fact that our GC closely follows a GC for a WAM-based system gives some confidence that our findings are applicable to a wider range of Prolog implementations. This section only concentrates on the modifications to the algorithm described in [12] and cannot be understood without detailed understanding of this paper.

Our modifications only affect the *marking* phase of GC. The modified algorithm is provided in pseudo code in Fig. 2 and discussed below. Added lines and deleted lines are marked with `+/-` at the start of the line.

First, initialize_and_mark() marks all data that is accessible from the continuation PC and at the same time initialises variables for which it finds a 'first-access' instruction, finishing the initialization of the environment. All environments are marked as 'seen'. This is the same as in [12], except

[7] Mats Carlsson has confirmed that SICStus 4 uses VM code scanning to deal with uninitialized variables in the environment. See also http://www.sics.se/sicstus/docs/latest4/pdf/relnotes.pdf

```
procedure mark_environments(env, PC)
   while ( env )
       if ( not_seen(env) )
           set_seen(env)
-          initialize(env, PC)
+          initialize_and_mark(env, PC)
           PC = env->PC
           env = env->parent
       else
+          mark(env, PC)
           return

procedure mark_choices(ch)
    env = ch->environment
    early_reset_trail()
    while ( ch )
        if ( pc_choice(ch) )
            mark_environments(env, ch->PC)
        else if ( alt_clause(ch) )
+           unmarked = count_unmarked_arguments(env)
+           while ( unmarked > 0 && clause )
+               mark_arguments(env, clause->code)
+               clause = next_visible(clause)
            if ( not_seen(env) )
                set_seen(env)
                mark_environments(env->parent, env->PC),
+       else if ( foreign_choice(ch) )
+           mark_all_arguments(env);

procedure mark_stacks(env, ch, PC)
    mark_environments(env, PC)
    mark_choices(ch)
```

Fig. 2. Pseudo code for the marking algorithm

- Mark variables in the environment that are referred to by instructions reachable from the PC instead of all variables in the environment.
- If we find a reference pointer to a parent environment, we mark the pointer and continue marking the referenced destination. In the traditional algorithm the variable in the parent is marked if we mark the parent environment. Now we must cover the case where the corresponding variable is accessed in this frame, but not in the parent frame.
- When called from mark_choices(), that marking is normally aborted if the frame has already been seen. Now we must continue to mark the first seen environment as this continuation may have a different PC and thus access to different variables. There is no need to continue with the parent frame as that has already been marked using the same PC.

Marking choicepoints is also similar to [12]. It resets trail entries that point to garbage cells (early reset, dealing with property 4) and then marks the associated environment. As SWI-Prolog passes arguments over the stack, if an alternate clause is encountered we need to keep all arguments that are used by the remainder of the clause list (possibly reduced due to indexing). Simply scanning the code of each clause could scan a lot of code on, for example, predicates with many facts. We avoid this by computing the number of unmarked arguments and abort the scan if all arguments are marked. Note that a clause without singleton variables in the head accesses all arguments and thus stops the search. Ground facts are a common example. Finally, as we have no information on how a foreign predicate accesses its arguments we must mark all arguments as accessible.

Sweeping an environment has been changed slightly. In [12], all heap references in the environment are inserted into relocation chains. Now, we first check whether the heap reference is marked. If so, we put it into a relocation chain as before, otherwise we assign the atom `'<garbage_collected>'` to the variable. This ensures consistency of the environment variable after heap relocation and is needed by the debugger if execution switches from normal mode to debug mode after a user interrupt or explicit call to **trace/0** inside code running in no-debug mode. In such cases, the debugger may show arguments of parent goals that were executed in normal mode as `'<garbage_collected>'` and the graphical debugger may show variables from the environment this way.

Note that if the program was started in debug mode, all data remains accessible through extra 'debug' choicepoints that also facilitate 'retry' at goals that were started deterministically. Figure 6 illustrates the problem using an explicit call to **garbage_collect/0** and **trace/0**. Explicitly calling **trace/0** is common practice to start debugging in a very specific state. The explicit call to **garbage_collect/0** is there only to illustrate what happens if GC was invoked at that specific point, while the system still operates in no-debug mode.

```
test :-
        read_line_to_codes(user_input, List),
        all_spaces(List).

all_spaces([]).
all_spaces([0' |T]) :- !, all_spaces(T).
all_spaces(_) :- garbage_collect, trace, fail.

1 ?- test.
|:   xx.
   Call: (11) fail ? goals
     [11] fail
     [10] all_spaces('<garbage_collected>')
      [1] '$toplevel'
   Call: (11) fail ?
```

Fig. 3. The debugger showing a garbage collected argument

7 Evaluation

Our evaluation considers four aspects: time, space, implementation effort and maintenance. In the tradition of SWI-Prolog, we consider mainly real and large applications. We selected the following applications because of diversity, size and the amount of garbage collection involved: CHAT80 (Pereira & Warren, 1986) running its test-suite in a forward chaining loop to force GC, Back52 (Thomas Hoppe et all., 1993) running its test suite, CHR compiler (Tom Schrijvers) compiling itself, k123.pl (Peter Vanbroekhoven) and pgolf.pl (Mats Carlsson).

The results are shown in Tab. 2. The first set of columns describe the overall timing, the last set describes characteristics of the code scanning version only and is discussed in Sect. 7.3. All timings are executed on an AMD Athlon X2 5400+; 64-bit Linux 2.6 using the 64-bit development version of SWI-Prolog based on 5.6.55. Reported time is in seconds. Frequency stepping was disabled during the tests.

7.1 Time evaluation

Table 2 shows that the overall execution time is only slightly affected by our changes. Note that the logic to trigger GC depends on the amount of memory that is accessible after the previous GC and therefore different effectiveness of GC leads to unpredictable overall behaviour of the program in terms of time and number of garbage collections.

We obtained a detailed breakdown of the garbage collector using valgrind [18] with the *callgrind* tool and *kcachegrind* to explore the results. The overhead of analysing instructions is approximately 1% of the garbage collector marking

Test	Time	#GC	GCLeft	GCTime	AvgScan	AvgCls	AvgInstr
Without code scanning							
k123	8.88	164	1,594,534	1.35			
chat80	2.56	109	18,661	0.10			
back52	2.31	406	5,589	0.17			
pgolf	13.22	53	7,328,689	3.44			
chr	6.41	36	3,466,387	1.17			
With code scanning							
k123	8.71	209	1,111,646	1.20	1.51	0.09	12.10
chat80	2.42	111	12,301	0.08	1.68	0.60	14.52
back52	2.21	420	3,360	0.15	1.56	0.12	11.19
pgolf	11.06	53	7,151,304	3.19	1.42	0.01	12.37
chr	6.29	38	3,265,471	1.15	1.91	0.32	14.52

Table 2. Effects of code scanning. *Time* is the total execution time (including GC time); *#GC* the number of garbage collections; *GCLeft* the average amount of memory (heap+trail) immediately after GC and *GCTime* the time spent on GC. *AvgScan* is the average number of continuation points that must be explored for an environment; *AvgCls* the average number of additional clauses scanned; *AvgInstr* the average number of instructions scanned before reaching the end of the clause.

time. These timing are slightly distorted because gcc's inline function optimization needs to be disabled to analyse the breakdown of execution time over the various functions.

7.2 Space evaluation

Our approach based on marking accessible data by scanning the VM instructions obviously reaches the 'precise' result as defined in the introduction for the heap and trail stack. It does not provide the optimal result for the environment stack. Only the approach as taken by SICStus is optimal here in the sense that the stack contains no variables that are not accessible, while using our marking approach the variables remain in the environment, bound to `'<garbage_collected>'`. Environment stack usage is in practice rarely a bottleneck and our deficiency is a constant amount rather than the difference between finite and infinite stack usage.

Table 2 also explains why precise GC is not widespread. Except for memory usage of the k123.pl test, we find no noticeable differences in the memory usage after GC. The k123 program is a small program (75 lines after cleanup of unreachable code). The central predicate **mmul/3** in Fig. 4 is deterministic. Lacking temporary registers and environment trimming, the old SWI-Prolog, could not dispose the intermediate matrices.

Implementation and maintenance Only the code for marking environments and clearing uninitialised variables was extended from originally 150 lines (C), to 557 including comment and debugging statements. Total implementation effort was

```
mmul(M, M6) :-
        mmul(M, M, M1),   mmul(M1, M1, M2), mmul(M2, M2, M3),
        mmul(M3, M3, M4), mmul(M4, M4, M5), mmul(M5, M5, M6).
```

Fig. 4. Main routine of k123.pl

4 days. One of the problems associated with VM instruction interpretation is maintenance that results from changing the instruction set. SWI-Prolog maintains information of the instruction format for each instruction. This is used to list VM instructions, deal with saving and loading and simplifies VM instruction scanning as it allows enumerating the instructions using a generic loop. Four instructions have variable length data associated with them (packed string and unbounded integer) and need (uniform) special attention.

In addition to the generic code walking, 36 out of 89 instructions require special attention as described in table Tab. 3. The table states the number of instructions the marking algorithm needs to understand, the number of groups of instructions that require different treatment (especially the variable accessing functions are often handled using the same code) and the number of lines of C-code involved.

Description	instructions	groups	lines
Identify flow control	6	5	44
Realise initialization of uninitialised variables	3	1	10
Identify variable access for marking (body)	14	6	30
Identify variable access for marking (head)	13	6	27

Table 3. Statistics on interpreting VM instructions

7.3 Discussion

Before we arrived at the current implementation we had two worries: prohibitive costs of multiple scans of the same code from different continuations and prohibitive scans of code from multiple clauses to identify the still-reachable arguments. Column *AvgCls* of Tab. 2 (page 135) indicates that scanning alternative clauses is cheap, while the value of equal GC behaviour between in-clauses disjunctions and alternative clauses is obvious.

Our first prototype avoided multiple scans of the same code from different continuations. Not correctly dealing with early-reset, this code was flawed and abandoned. Nevertheless, it executed the above programs correctly and we obtained statistics on its effectiveness. On the above test cases, multiple scans increase the number of scanned instructions by 0, 58%, 7%, 7% and 3% (same order as Tab. 2). As the scanning itself is responsible for less than 1% of the

time of the mark phase, it is considered neglectable. This conclusion can also be drawn from column *AvgScan* and *AvgInstr* together with the 1% time spent on code scanning.

8 Conclusions

We have defined a set of five properties, each of which accompanied with a very simple test case, that must be satisfied to deal with infinite (lazy) datastructures in Prolog. We have proven that such datastructures are of significant practical value as they can be used to realise processing a repositionable input stream using the full power of non-deterministic grammar rules (DCGs). The majority of Prolog implementations that provide the required attributed variables to realise a lazy datastructure does not provide the required precise garbage collector. Precise GC can be realised using a VM that uses registers to pass arguments, implements environment trimming and codes in-clauses disjunction using anonymous predicates. Our case study indicates that other virtual machines can be remedied by scanning virtual machine instructions to identify reachable variables in the environment. This technique is viable for any Prolog system based on emulating virtual machine instructions. Next to supporting infinite datastructure, the approximately 1% extra cost in the marking phase is more than compensated for in the compacting phase of the garbage collector.

The current version of SWI-Prolog is shipped with the described enhancements to the garbage collector and a library to use DCGs on repositionable input streams.

Acknowledgements

We would like to thank Mats Carlsson for explaining to one of the authors how the reachability problem is solved in SICStus Prolog, Bart Demoen for explaining some folklore and the bitmap technique and Paulo Moura for investigating the state of the art in some popular Prolog systems as shown in Tab. 1. Vitor Santos Costa has confirmed property 1 for YAP, which is planned to be fixed soon.

References

1. Hermenegildo, M.V., Gras, D.C., Carro, M.: Using attributed variables in the implementation of concurrent and parallel logic programming systems. In: ICLP. (1995) 631–645
2. Wadler, P.L.: Fixing some space leaks with a garbage collector. Software Practice and Experience **17** (1987) 595–609
3. et. al., P.M.: Prolog (2006) ISO/IEC DTR 132113:2006.
4. Neumerkel, U.: Extensible unification by metastructures. In Bruynooghe, M., ed.: Proceedings of META90, Workshop on Meta-Programming in Logic, Leuven, Belgium (1990)
5. Huitouze, S.L.: A new data structure for implementing extensions to prolog. In: PLILP. Volume 456., Springer-Verlag (1990) 136–150 LNCS 456.

6. Holzbaur, C.: Metastructures versus attributed variables in the context of extensible unification. In: PLILP. Volume 631., Springer-Verlag (1992) 260–268 LNCS 631.
7. Demoen, B.: Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium (2002) URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.
8. Bowen, D.L., Byrd, L.M., Clocksin, W.: A portable Prolog compiler. In Pereira, L.M., ed.: Proceedings of the Logic Programming Workshop 1983, Lisabon, Portugal, Universidade nova de Lisboa (1983)
9. Neumerkel, U.: The binary WAM, a simplified Prolog engine. Technical report, Technische Universität Wien (1993) http://www.complang.tuwien.ac.at/ulrich/papers/PDF/binwam-nov93.pdf.
10. Zhou, N.F.: Garbage collection in B-Prolog. In: Proc. of the First Workshop on Memory Management in Logic Programming Implementations. (2000)
11. Castro, L.F., Costa, V.S.: Understanding memory management in Prolog systems. In Codognet, P., ed.: ICLP. Volume 2237 of Lecture Notes in Computer Science., Springer (2001) 11–26
12. Appleby, K., Carlsson, M., Haridi, S., Sahlin, D.: Garbage collection for Prolog based on WAM. Communications of the ACM **31** (1988) 719–741
13. Bekkers, Y., Ridoux, O., Ungaro, L.: Dynamic memory management for sequential logic programming languages. In: Workshop on Memory Management. (1992) LNCS 627.
14. Tarau, P., Boyer, M.: Elementary logic programs. In: PLILP, Springer-Verlag (1990) 365–381 LNCS 456.
15. Tarau, P., Neumerkel, U.: A novel term compression scheme and data representation in the binwam. In: PLILP, Springer-Verlag (1994) 73–87 LNCS 844.
16. Demoen, B., Tarau, P., Engels, G.: Segment order preserving copying garbage collection for wam based prolog. In: Symposion on Applied Computing (SAC), ACM (1996) 380–386
17. Buettner, K.A.: Fast decompilation of compiled prolog clauses. In Shapiro, E.Y., ed.: ICLP. Volume 225 of Lecture Notes in Computer Science., Springer (1986) 663–670
18. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In Ferrante, J., McKinley, K.S., eds.: PLDI, ACM (2007) 89–100

Pairing Functions, Boolean Evaluation and Binary Decision Diagrams in Prolog

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
E-mail: tarau@cs.unt.edu

Abstract. A "pairing function" J associates a unique natural number z to any two natural numbers x,y such that for two "unpairing functions" K and L, the equalities K(J(x,y))=x, L(J(x,y))=y and J(K(z),L(z))=z hold. Using pairing functions on natural number representations of truth tables, we derive an encoding for Binary Decision Diagrams with the unique property that its boolean evaluation faithfully mimics its structural conversion to a a natural number through recursive application of a matching pairing function. We then use this result to derive *ranking* and *unranking* functions for BDDs and reduced BDDs. The paper is organized as a self-contained literate Prolog program, available at `http://logic.csci.unt.edu/tarau/research/2008/pBDD.zip`.
Keywords: logic programming and computational mathematics, pairing/unpairing functions, encodings of boolean functions, binary decision diagrams, natural number representations of truth tables

1 Introduction

This paper is an exploration with logic programming tools of *ranking* and *unranking* problems on Binary Decision Diagrams. The practical expressiveness of logic programming languages (in particular Prolog) are put at test in the process. The paper is part of a larger effort to cover in a declarative programming paradigm, arguably more elegantly, some fundamental combinatorial generation algorithms along the lines of [1]. However, our main focus is by no means "yet another implementation of BDDs in Prolog". The paper is more about fundamental isomorphisms between logic functions and their natural number representations, in the tradition of [2], with the unusual twist that everything is expressed as a literate Prolog program, and therefore automatically testable by the reader. One could put such efforts under the generic umbrella of an emerging research field that we would like to call *executable theoretical computer science*. Nevertheless, we also hope that the more practically oriented reader will be able to benefit from this approach by being able to experiment with, and reuse our Prolog code in applications.

The paper is organized as follows: Sections 2 and 3 overview efficient evaluation of boolean formulae in Prolog using bitvectors represented as arbitrary length integers and Binary Decision Diagrams (BDDs).

Section 4 discusses classic pairing and unpairing operations and introduces pairing/unpairing predicates acting directly on bitlists.

Section 5 introduces a novel BDD encoding (based on our unpairing functions) and discusses the surprising equivalence between boolean evaluation of BDDs and the inverse of our encoding, the main result of the paper.

Section 6 describes *ranking* and *unranking* functions for BDDs and reduced BDDs.

Sections 7 and 8 discuss related work, future work and conclusions.

The code in the paper, embedded in a literate programming LaTeX file, is entirely self contained and has been tested under *SWI-Prolog*.

2 Parallel Evaluation of Boolean Functions with Bitvector Operations

Evaluation of a boolean function can be performed one value at a time as in the predicate `if_then_else/4`

```
if_then_else(X,Y,Z,R):-
  bit(X),bit(Y),bit(Z),
  ( X==1->R=Y
  ; R=Z
  ).

bit(0).
bit(1).
```

resulting in a *truth table*[1]

```
?- if_then_else(X,Y,Z,R),write([X,Y,Z]:R),nl,fail;nl.
[0, 0, 0]:0
[0, 0, 1]:1
[0, 1, 0]:0
[0, 1, 1]:1
[1, 0, 0]:0
[1, 0, 1]:0
[1, 1, 0]:1
[1, 1, 1]:1
```

Clearly, this does not take advantage of the ability of modern hardware to perform such operations one word a time - with the instant benefit of a speed-up proportional to the word size. An alternate representation, adapted from [1] uses integer encodings of 2^n bits for each boolean variable $X_0, \ldots, X_{n-1}$. Bitvector operations evaluate all value combinations at once.

[1] One can see that if the number of variables is fixed, we can ignore the bitsrings in the brackets. Thus, the truth table can be identified with the natural number, represented in binary form by the last column.

Proposition 1 *Let x_k be a variable for $0 \leq k < n$ where n is the number of distinct variables in a boolean expression. Then column k in the matrix representation of the inputs in the the truth table represents, as a bitstring, the natural number:*

$$x_k = (2^{2^n} - 1)/(2^{2^{n-k-1}} + 1) \tag{1}$$

For instance, if $n = 2$, the formula computes $x_0 = 3 = [0, 0, 1, 1]$ and $x_1 = 5 = [0, 1, 0, 1]$.

The following predicates, working with arbitrary length bitstrings are used to evaluate variables x_k with $k \in [0..n-1]$ with formula 1 and map the constant boolean function `1` to the bitstring of length 2^n, `111..1`, representing $2^{2^n} - 1$

```
% maps variable K in [0..NbOfBits-1] to Xk
var_to_bitstring_int(NbOfBits,K,Xk):-
  all_ones_mask(NbOfBits,Mask),
  var_to_bitstring_int(NbOfBits,Mask,K,Xk).

var_to_bitstring_int(NbOfBits,Mask,K,Xk):-
  NK is NbOfBits-(K+1),
  D is (1≪(1≪NK))+1,
  Xk is Mask//D.

% represents constant 1 as 11...1 build with NbOfBits bits
all_ones_mask(NbOfBits,Mask):-Mask is (1≪(1≪NbOfBits))-1.
```

We have used in `var_to_bitstring_int` an adaptation of the efficient bitstring-integer encoding described in the Boolean Evaluation section of [1]. Intuitively, it is based on the idea that one can look at n variables as bitstring representations of the n columns of the truth table.

Variables representing such bitstring-truth tables (seen as *projection functions*) can be combined with the usual bitwise integer operators, to obtain new bitstring truth tables, encoding all possible value combinations of their arguments. Note that the constant 0 is represented as 0 while the constant 1 is represented as $2^{2^n} - 1$, corresponding to a column in the truth table containing ones exclusively.

3 Binary Decision Diagrams

We have seen that Natural Numbers in $[0..2^{2^n} - 1]$ can be used as representations of truth tables defining n-variable boolean functions. A binary decision diagram (BDD) [3] is an ordered binary tree obtained from a boolean function, by assigning its variables, one at a time, to `0` (left branch) and `1` (right branch). In virtually all practical applications BDDs are represented as DAGs after detecting shared nodes. We safely ignore this here as they represent the same logic function, which is all we care about at this point. Typically in the early literature, the acronym ROBDD is used to denote reduced ordered BDDs. Because

this optimization is now so prevalent, the term BDD is frequently use to refer to ROBDDs. Strictly speaking, BDD in this paper will stand for *ordered BDD with reduction of identical branches but without node sharing*.

The construction deriving a BDD of a boolean function f is known as Shannon expansion [4], and is expressed as

$$f(x) = (\bar{x} \wedge f[x \leftarrow 0]) \vee (x \wedge f[x \leftarrow 1]) \tag{2}$$

where $f[x \leftarrow a]$ is computed by uniformly substituting a for x in f. Note that by using the more familiar boolean if-the-else function Shannon expansion can also be expressed as:

$$f(x) = if\ x\ then\ f[x \leftarrow 1]\ else\ f[x \leftarrow 0] \tag{3}$$

We represent a *BDD* in Prolog as a binary tree with constants `0` and `1` as leaves, marked with the function symbol `c/1`. Internal *if-then-else* nodes marked with `ite/3` are controlled by variables, ordered identically in each branch, as first arguments of `ite/1`. The two other arguments are subtrees representing the `Then` and `Else` branches. Note that, in practice, reduced, canonical DAG representations are used instead of binary tree representations.

Alternatively, we observe that the Shannon expansion can be directly derived from a 2^n size truth table, using bitstring operations on encodings of its n variables. Assuming that the first column of a truth table corresponds to variable x, $x = 0$ and $x = 1$ mask out, respectively, the upper and lower half of the truth table.

```
% splits a truth table of NV variables in 2 tables of NV-1 variables
shannon_split(NV,X, Hi,Lo):-
  all_ones_mask(NV,M),
  NV1 is NV-1,
  all_ones_mask(NV1,LM),
  HM is xor(M,LM),
  Lo is /\(LM,X),
  H is /\(HM,X),
  Hi is H>>(1<<NV1).
```

Note that the operation `shannon_split` can be reversed as follows:

```
% fuses 2 truth tables of NV-1 variables into one of NV variables
shannon_fuse(NV,Hi,Lo, X):-
  NV1 is NV-1,
  H is Hi<<(1<<NV1),
  X is \/(H,Lo).

?- shannon_split(2, 7, X,Y),shannon_fuse(2, X,Y, Z).
X = 1,
Y = 3,
Z = 7.

?- shannon_split(3, 42, X,Y),shannon_fuse(3, X,Y, Z).
```

```
X = 2,
Y = 10,
Z = 42.
```

Another way to look at these two operations (for a fixed value of NV), is as bijections associating a pair of natural numbers to a natural number, i.e. as *pairing* functions.

4 Pairing and Unpairing Functions

Definition 1 *A* pairing function *is a bijection* $f : Nat \times Nat \rightarrow Nat$. *An* unpairing function *is a bijection* $g : Nat \rightarrow Nat \times Nat$.

Following Julia Robinson's notation [5], given a pairing function J, its left and right inverses K and L are such that

$$J(K(z), L(z)) = z \tag{4}$$

$$K(J(x, y)) = x \tag{5}$$

$$L(J(x, y)) = y \tag{6}$$

We refer to [6] for a typical use in the foundations of mathematics and to [7] for an extensive study of various pairing functions and their computational properties.

4.1 Cantor's Pairing Function

Starting from Cantor's pairing function

```
cantor_pair(K1,K2,P):-P is (((K1+K2)*(K1+K2+1))//2)+K2.
```

bijections from $Nat \times Nat$ to Nat have been used for various proofs and constructions of mathematical objects [5, 6].

For $X, Y \in \{0, 1, 2, 3\}$ the sequence of values of this pairing function is:

```
?- findall(R,(between(0,3,A),between(0,3,B),cantor_pair(A,B,R)),Rs).
Rs = [0, 2, 4, 6, 1, 5, 9, 13, 3, 11, 19, 27, 7, 23, 39, 55]
```

Note however, that the inverse of Cantor's pairing function involves potentially expensive floating point operations that are also likely to loose precision for arbitrary length integers.

4.2 The Pepis-Kalmar Pairing Function

Another pairing function that can be implemented using only elementary integer operations is the following:

$$f(x,y) = 2^x(2y+1) - 1 \tag{7}$$

The predicates `pepis_pair/3` and `pepis_unpair/3` are derived from the function **pepis_J** and its left and right unpairing companions **pepis_K** and **pepis_L** that have been used, by Pepis, Kalmar and Robinson in some fundamental work on recursion theory, decidability and Hilbert's Tenth Problem in [8–10]:

```
pepis_pair(X,Y,Z):-pepis_J(X,Y,Z).

pepis_unpair(Z,X,Y):-pepis_K(Z,X),pepis_L(Z,Y).

pepis_J(X,Y, Z):-Z is ((1≪X)∗((Y≪1)+1))-1.
pepis_K(Z, X):-Z1 is Z+1,two_s(Z1,X).
pepis_L(Z, Y):-Z1 is Z+1,no_two_s(Z1,N),Y is (N-1)≫1.

two_s(N,R):-even(N),!,H is N≫1,two_s(H,T),R is T+1.
two_s(_,0).

no_two_s(N,R):-two_s(N,T),R is N // (1≪T).

even(X):- 0 =:= /\(1,X).
odd(X):- 1 =:= /\(1,X).
```

This pairing function is asymmetrically growing (faster growth on the first argument). It works as follows:

```
?- pepis_pair(1,10,R).
R = 41.

?- pepis_unpair(10,1,R).
R = 3071.

?- findall(R,(between(0,3,A),between(0,3,B),pepis_pair(A,B,R)),Rs).
Rs=[0, 2, 4, 6, 1, 5, 9, 13, 3, 11, 19, 27, 7, 23, 39, 55]
```

4.3 Pairing/Unpairing operations acting directly on bitlists

We will describe here pairing operations, that are expressed exclusively as bitlist transformations of `bitmerge_unpair` and its inverse `bitmerge_pair`, and are therefore likely to be easily hardware implementable. As we have found out recently, they turn out to be the same as the functions defined in Steven Pigeon's PhD thesis on Data Compression [11], page 114).

The predicate `bitmerge_pair` implements a bijection from $Nat \times Nat$ to Nat that works by splitting a number's big endian bitstring representation into odd

and even bits, while its inverse `to_pair` blends the odd and even bits back together. The helper predicates `to_rbits` and `from_rbits`, given in the Appendix, convert to/from integers to bitlists.

```
bitmerge_pair(X,Y,P):-
  to_rbits(X,Xs),
  to_rbits(Y,Ys),
  bitmix(Xs,Ys,Ps),!,
  from_rbits(Ps,P).

bitmerge_unpair(P,X,Y):-
  to_rbits(P,Ps),
  bitmix(Xs,Ys,Ps),!,
  from_rbits(Xs,X),
  from_rbits(Ys,Y).

bitmix([X|Xs],Ys,[X|Ms]):-!,bitmix(Ys,Xs,Ms).
bitmix([],[X|Xs],[0|Ms]):-!,bitmix([X|Xs],[],Ms).
bitmix([],[],[]).
```

The transformation of the bitlists, done by the bidirectional predicate bitmix is shown in the following example with bitstrings aligned:

```
?- bitmerge_unpair(2008,X,Y),bitmerge_pair(X,Y,Z).
X = 60,
Y = 26,
Z = 2008

% 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
%   60:[      0,    1,    1,    1,    1]
%   26:[   0,    1,    0,    1,    1   ]
```

Note that we represent numbers with bits in reverse order (least significant on the left). Like in the case of Cantor's pairing function, we can see similar growth in both arguments:

```
?- between(0,15,N),bitmerge_unpair(N,A,B),
   write(N:(A,B)),write(' '),fail;nl.
0: (0, 0) 1: (1, 0) 2: (0, 1) 3: (1, 1)
4: (2, 0) 5: (3, 0) 6: (2, 1) 7: (3, 1)
8: (0, 2) 9: (1, 2) 10: (0, 3) 11: (1, 3)
12: (2, 2) 13: (3, 2) 14: (2, 3) 15: (3, 3)

?- between(0,3,A),between(0,3,B),bitmerge_pair(A,B,N),
   write(N:(A,B)),write(' '),fail;nl.
0: (0, 0) 2: (0, 1) 8: (0, 2) 10: (0, 3)
1: (1, 0) 3: (1, 1) 9: (1, 2) 11: (1, 3)
4: (2, 0) 6: (2, 1) 12: (2, 2) 14: (2, 3)
5: (3, 0) 7: (3, 1) 13: (3, 2) 15: (3, 3)
```

It is also convenient sometimes to see pairing/unpairing as one-to-one functions from/to the underlying language's ordered pairs, i.e. `X-Y` in Prolog :

```
bitmerge_pair(X-Y,Z):-bitmerge_pair(X,Y,Z).

bitmerge_unpair(Z,X-Y):-bitmerge_unpair(Z,X,Y).
```

5 Encodings of Binary Decision Diagrams

We will build a BDD by applying `bitmerge_unpair` recursively to a Natural Number `TT`, seen as an N-variable 2^N bit truth table. This results in a complete binary tree of depth N. As we will show later, this binary tree represents a BDD that returns `TT` when evaluated applying its boolean operations.

```
% NV=number of varibles, TT=a truth table, BDD the result
plain_bdd(NV,TT, bdd(NV,BDD)):-
  Max is (1<<(1<<NV)),
  TT<Max,
  isplit(NV,TT, BDD).

% recurses to depth NV, splitting TT into pairs
isplit(0,TT,c(TT)).
isplit(NV,TT,R):-NV>0,
  NV1 is NV-1,
  bitmerge_unpair(TT,Hi,Lo),
  isplit(NV1,Hi,H),
  isplit(NV1,Lo,L),
  ite(NV1,H,L)=R.
```

The following examples show the results returned by `plain_bdd` for all 2^{2^k} truth tables associated to k variables, with $k = 2$.

```
?- between(0,15,TT),plain_bdd(2,TT,BDD),write(TT:BDD),nl,fail;nl
0:bdd(2, ite(1, ite(0, c(0), c(0)), ite(0, c(0), c(0))))
1:bdd(2, ite(1, ite(0, c(1), c(0)), ite(0, c(0), c(0))))
2:bdd(2, ite(1, ite(0, c(0), c(0)), ite(0, c(1), c(0))))
...
13:bdd(2, ite(1, ite(0, c(1), c(1)), ite(0, c(0), c(1))))
14:bdd(2, ite(1, ite(0, c(0), c(1)), ite(0, c(1), c(1))))
15:bdd(2, ite(1, ite(0, c(1), c(1)), ite(0, c(1), c(1))))
```

5.1 Reducing the $BDDs$

The predicate `bdd_reduce` reduces a BDD by trimming identical left and right subtrees, and the predicate `bdd` associates this reduced form to $N \in Nat$.

```
bdd_reduce(BDD,bdd(NV,R)):-nonvar(BDD),BDD=bdd(NV,X),bdd_reduce1(X,R).

bdd_reduce1(c(TT),c(TT)).
bdd_reduce1(ite(_,A,B),R):-A==B,bdd_reduce1(A,R).
bdd_reduce1(ite(X,A,B),ite(X,RA,RB)):-A\==B,
```

```
  bdd_reduce1(A,RA),bdd_reduce1(B,RB).

bdd(NV,TT, ReducedBDD):-
  plain_bdd(NV,TT, BDD),
  bdd_reduce(BDD,ReducedBDD).
```

Note that we omit here the reduction step consisting in sharing common subtrees, as it is obtained easily by replacing trees with DAGs. The process is facilitated by the fact that our unique encoding provides a perfect hashing key for each subtree. The following examples show the results returned by `bdd` for `NV=2`.

```
?- between(0,15,TT),bdd(2,TT,BDD),write(TT:BDD),nl,fail;nl
0:bdd(2, c(0))
1:bdd(2, ite(1, ite(0, c(1), c(0)), c(0)))
2:bdd(2, ite(1, c(0), ite(0, c(1), c(0))))
3:bdd(2, ite(0, c(1), c(0)))
...
13:bdd(2, ite(1, c(1), ite(0, c(0), c(1))))
14:bdd(2, ite(1, ite(0, c(0), c(1)), c(1)))
15:bdd(2, c(1))
```

5.2 From BDDs to Natural Numbers

One can "evaluate back" the binary tree representing the BDD, by using the pairing function `bitmerge_pair`. The inverse of `plain_bdd` is implemented as follows:

```
plain_inverse_bdd(bdd(_,X),TT):-plain_inverse_bdd1(X,TT).

plain_inverse_bdd1(c(TT),TT).
plain_inverse_bdd1(ite(_,L,R),TT):-
  plain_inverse_bdd1(L,X),
  plain_inverse_bdd1(R,Y),
  bitmerge_pair(X,Y,TT).


?- plain_bdd(3,42, BDD),plain_inverse_bdd(BDD,N).
BDD = bdd(3,
          ite(2,
              ite(1,
                  ite(0, c(0), c(0)),
                  ite(0, c(0), c(0))),
              ite(1,
                  ite(0, c(1), c(1)),
                  ite(0, c(1), c(0))))),
N = 42
```

Note however that `plain_inverse_bdd/2` does not act as an inverse of `bdd/3`, given that the *structure* of the BDD tree is changed by reduction.

5.3 Boolean Evaluation of BDDs

This raises the obvious question: how can we recover the original truth table from a reduced BDD? The obvious answer is: by evaluating it as a boolean function! The predicate `ev/2` describes the *BDD* evaluator:

```
ev(bdd(NV,B),TT):-
  all_ones_mask(NV,M),
  eval_with_mask(NV,M,B,TT).

evc(0,_,0).
evc(1,M,M).

eval_with_mask(_,M,c(X),R):-evc(X,M,R).
eval_with_mask(NV,M,ite(X,T,E),R):-
  eval_with_mask(NV,M,T,A),
  eval_with_mask(NV,M,E,B),
  var_to_bitstring_int(NV,M,X,V),
  ite(V,A,B,R).
```

The predicate `ite/4` used in `eval_with_mask` implements the boolean function `if X then T else E` using arbitrary length bitvector operations:

```
ite(X,T,E, R):-R is xor(/\(X,xor(T,E)),E).
```

Note that this equivalent formula for `ite` is slightly more efficient than the obvious one with $\wedge$ and $\vee$ as it requires only 3 boolean operations. We will use `ite/4` as the basic building block for implementing a boolean evaluator for BDDs.

5.4 The Equivalence

A surprising result is that boolean evaluation and structural transformation with repeated application of *pairing* produce the same result, i.e. the predicate `ev/2` also acts as an inverse of `bdd/2` and `plain_bdd/2`.
As the following example shows, boolean evaluation `ev/2` *faithfully emulates* `plain_inverse_bdd/2`*, on both plain and reduced BDDs.*

```
?- plain_bdd(3,42,BDD),ev(BDD,N).
BDD = bdd(3,
        ite(2,
            ite(1,
                ite(0, c(0), c(0)),
                ite(0, c(0), c(0))),
            ite(1,
                ite(0, c(1), c(1)),
                ite(0, c(1), c(0))))),
N = 42

?- bdd(3,42,BDD),ev(BDD,N).
BDD = bdd(3,
```

```
         ite(2,
            c(0),
            ite(1,
                c(1),
                ite(0, c(1), c(0)))),
N = 42
```

The main result of this subsection can now be summarized as follows:

Proposition 2 *Let B be the complete binary tree of depth N, obtained by recursive applications of* `bitmerge_unpair` *on a truth table T, as described by the predicate* `plain_bdd(N,T,B)`.

Then for any NV and any T, when B is interpreted as an (unreduced) BDD, the result V of its boolean evaluation using the predicate $ev(N, B, V)$ and the result R obtained by applying $plain\_inverse\_bdd(N, B, R)$ are both identical to T. Moreover, the operation $ev(N, B, V)$ reverses the effects of both `plain_bdd` *and* `bdd` *with an identical result.*

Proof: The predicate `plain_bdd` builds a binary tree by splitting the bitstring $tt \in [0..2^N - 1]$ up to depth N. Observe that this corresponds to the Shannon expansion [4] of the formula associated to the truth table, using variable order $[n-1, ..., 0]$. Observe that the effect of `bitstring_unpair` is the same as

- the effect of `var_to_bitstring_int(N,M,(N-1),R)` acting as a mask selecting the left branch
- and the effect of its complement, acting as a mask selecting the right branch.

Given that 2^N is the double of 2^{N-1}, the same invariant holds at each step, as the bitstring length of the truth table reduces to half. On the other hand, it is clear that ev reverses the action of both `plain_bdd` and `bdd` as BDDs and reduced BDDs represent the same boolean function [3].

This result can be seen as a yet another intriguing isomorphism between boolean, arithmetic and symbolic computations.

6 Ranking and Unranking of BDDs

One more step is needed to extend the mapping between $BDDs$ with N variables to a bijective mapping from/to Nat: we will have to "shift toward infinity" the starting point of each new block of BDDs in Nat as BDDs of larger and larger sizes are enumerated.

First, we need to know by how much - so we compute the sum of the counts of boolean functions with up to N variables.

```
bsum(0,0).
bsum(N,S):-N>0,N1 is N-1,bsum1(N1,S).

bsum1(0,2).
bsum1(N,S):-N>0,N1 is N-1,bsum1(N1,S1),S is S1+(1<<(1<<N)).
```

The stream of all such sums can now be generated as usual:

```
bsum(S):-nat(N),bsum(N,S).

nat(0).
nat(N):-nat(N1),N is N1+1.
```

What we are really interested in, is decomposing `N` into the distance to the last `bsum` smaller than N, `N_M` and the index of that generates the sum, `K`.

```
to_bsum(N, X,N_M):-
  nat(X),bsum(X,S),S>N,!,
  K is X-1,
  bsum(K,M),
  N_M is N-M.
```

Unranking of an arbitrary BDD is now easy - the index `K` determines the number of variables and `N_M` determines the rank. Together they select the right BDD with `plain_bdd` and `bdd/3`.

```
nat2plain_bdd(N,BDD):-to_bsum(N, K,N_M),plain_bdd(K,N_M,BDD).

nat2bdd(N,BDD):-to_bsum(N, K,N_M),bdd(K,N_M,BDD).
```

Ranking of a BDD is even easier: we first compute its `NumberOfVars` and its rank `Nth`, then we shift the rank by the `bsums` up to `NumberOfVars`, enumerating the ranks previously assigned.

```
plain_bdd2nat(bdd(NumberOfVars,BDD),N) :-
  B=bdd(NumberOfVars,BDD),
  plain_inverse_bdd(B,Nth),
  K is NumberOfVars-1,
  bsum(K,S),N is S+Nth.

bdd2nat(bdd(NumberOfVars,BDD),N) :-
  B=bdd(NumberOfVars,BDD),
  ev(B,Nth),
  K is NumberOfVars-1,
  bsum(K,S),N is S+Nth.
```

As the following example shows, `nat2plain_bdd/2` and `plain_bdd2nat/2` implement inverse functions.

```
?- nat2plain_bdd(42,BDD),plain_bdd2nat(BDD,N).
BDD = bdd(4,
          ite(3,
              ite(2,
                  ite(1,
                      ite(0, c(0), c(0)),
                      ite(0, c(1), c(0))),
                  ite(1,
                      ite(0, c(1), c(0)),
                      ite(0, c(0), c(0)))),
```

```
              ite(2,
                  ite(1,
                      ite(0, c(0), c(0)),
                      ite(0, c(0), c(0))),
                  ite(1, ite(0, c(0), c(0)),
                         ite(0, c(0), c(0)))))),
N = 42
```

The same applies to `nat2bdd/2` and its inverse `bdd2nat/2`.

```
?- nat2bdd(42,BDD),bdd2nat(BDD,N).
BDD = bdd(4,
          ite(3,
              ite(2,
              ite(1, c(0),
                     ite(0, c(1), c(0))),
                     ite(1,
                         ite(0, c(1),c(0)), c(0))),
                         c(0))),
N = 42
```

We can now generate infinite streams of BDDs as follows:

```
plain_bdd(BDD):-nat(N),nat2plain_bdd(N,BDD).

bdd(BDD):-nat(N),nat2bdd(N,BDD).

?- plain_bdd(BDD).
BDD = bdd(1, ite(0, c(0), c(0))) ;
BDD = bdd(1, ite(0, c(1), c(0))) ;
BDD = bdd(2, ite(1, ite(0, c(0), c(0)), ite(0, c(0), c(0)))) ;
BDD = bdd(2, ite(1, ite(0, c(1), c(0)), ite(0, c(0), c(0)))) ;
...
?- bdd(BDD).
BDD = bdd(1, c(0)) ;
BDD = bdd(1, ite(0, c(1), c(0))) ;
BDD = bdd(2, c(0)) ;
BDD = bdd(2, ite(1, ite(0, c(1), c(0)), c(0))) ;
BDD = bdd(2, ite(1, c(0), ite(0, c(1), c(0)))) ;
BDD = bdd(2, ite(0, c(1), c(0))) ;
...
```

7 Related work

Pairing functions have been used in work on decision problems as early as [8, 9, 5]. *Ranking* functions can be traced back to Gödel numberings [2, 12] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [13, 1]. Binary Decision Diagrams are the dominant boolean function representation in the field of circuit design automation [14]. BDDs have been used in a Genetic Programming context [15, 16] as

a representation of evolving individuals subject to crossovers and mutations expressed as structural transformations and recently in a machine learning context for compressing probabilistic Prolog programs [17] representing candidate theories. Other interesting uses of BDDs in a logic and constraint programming context are related to representations of finite domains. In [18] an algorithm for finding minimal reasons for inferences is given.

8 Conclusion and Future Work

The surprising connection of pairing/unpairing functions and BDDs, is the indirect result of implementation work on a number of practical applications. Our initial interest has been triggered by applications of the encodings to combinational circuit synthesis in a logic programming framework [19, 20]. We have found them also interesting as uniform blocks for Genetic Programming applications of Logic Programming. In a Genetic Programming context [21], the bijections between bitvectors/natural numbers on one side, and trees/graphs representing BDDs on the other side, suggest exploring the mapping and its action on various transformations as a phenotype-genotype connection. Given the connection between BDDs to boolean and finite domain constraint solvers it would be interesting to explore in that context, efficient succinct data representations derived from our BDD encodings.

References

1. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
2. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691
4. Shannon, C.E.: Claude Elwood Shannon: collected papers. IEEE Press, Piscataway, NJ, USA (1993)
5. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
6. Cégielski, P., Richard, D.: Decidability of the theory of the natural integers with the cantor pairing function and the successor. Theor. Comput. Sci. **257**(1-2) (2001) 51–77
7. Rosenberg, A.L.: Efficient pairing functions - and why you should care. International Journal of Foundations of Computer Science **14**(1) (2003) 3–17
8. Pepis, J.: Ein verfahren der mathematischen logik. The Journal of Symbolic Logic **3**(2) (jun 1938) 61–76
9. Kalmar, L.: On the reduction of the decision problem. first paper. ackermann prefix, a single binary predicate. The Journal of Symbolic Logic **4**(1) (mar 1939) 1–9
10. Robinson, J.: An introduction to hyperarithmetical functions. The Journal of Symbolic Logic **32**(3) (sep 1967) 325–342

11. Pigeon, S.: Contributions à la compression de données. Ph.d. thesis, Université de Montréal, Montréal (2001)
12. Hartmanis, J., Baker, T.P.: On simple goedel numberings and translations. In Loeckx, J., ed.: ICALP. Volume 14 of Lecture Notes in Computer Science., Springer (1974) 301–316
13. Martinez, C., Molinero, X.: Generic algorithms for the generation of combinatorial objects. In Rovan, B., Vojtas, P., eds.: MFCS. Volume 2747 of Lecture Notes in Computer Science., Springer (2003) 572–581
14. Drechsler, R., Shi, J., Fey, G.: Synthesis of fully testable circuits from bdds. IEEE Trans. on CAD of Integrated Circuits and Systems **23**(3) (2004) 440–443
15. Sakanashi, H., Higuchi, T., Iba, H., Kakazu, Y.: Evolution of binary decision diagrams for digital circuit design using genetic programming. In Higuchi, T., Iwata, M., Liu, W., eds.: ICES. Volume 1259 of Lecture Notes in Computer Science., Springer (1996) 470–481
16. Chen, S.T., Lin, S.S., Huang, L.T., Wei, C.J.: Towards the exact minimization of bdds-an elitism-based distributed evolutionary algorithm. J. Heuristics **10**(3) (2004) 337–355
17. Raedt, L.D., Kersting, K., Kimmig, A., Revoredo, K., Toivonen, H.: Compressing probabilistic prolog programs. Machine Learning **70**(2-3) (2008) 151–168
18. Hawkins, P., Stuckey, P.J.: A hybrid bdd and sat finite domain constraint solver. In Hentenryck, P.V., ed.: PADL. Volume 3819 of Lecture Notes in Computer Science., Springer (2006) 103–117
19. Tarau, P., Luderman, B.: Exact combinational logic synthesis and non-standard circuit design. In: CF '08: Proceedings of the 2008 conference on Computing frontiers, New York, NY, USA, ACM (2008) 179–188
20. Tarau, P., Luderman, B.: A Logic Programming Framework for Combinational Circuit Synthesis. In: 23rd International Conference on Logic Programming (ICLP), LNCS 4670, Porto, Portugal, Springer (September 2007) 180–194
21. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)

Appendix

To make the code in the paper fully self contained, we list here some auxiliary functions.

```
% converts an int to a list of bits, least significant first
to_rbits(0,[]).
to_rbits(N,[B|Bs]):-N>0,B is N mod 2, N1 is N//2,
  to_rbits(N1,Bs).

% converts a list of bits (least significant first) into an int
from_rbits(Rs,N):-nonvar(Rs),from_rbits(Rs,0,0,N).

from_rbits([],_,N,N).
from_rbits([X|Xs],E,N1,N3):-NewE is E+1,N2 is X<<E+N1,
  from_rbits(Xs,NewE,N2,N3).
```

Confidence based Work Stealing in Parallel Constraint Programming

Geoffrey Chu[1], Christian Schulte[2], and Peter J. Stuckey[1]

[1] NICTA Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au
[2] KTH – Royal Institute of Technology, Sweden
cschulte@kth.se

Abstract. In parallel constraint solving, work stealing not only allows for dynamic load balancing, but also determines which parts of the search tree are searched next. Thus the place from where work is stolen has a dramatic effect on the efficiency of a parallel search algorithm. In this paper we examine quantitatively how optimal work stealing can be performed given an estimate of the relative solution densities of the subtrees at each node in the search tree and show how this is related to the branching heuristic strength. We propose an adaptive work stealing algorithm that automatically performs different work stealing strategies based on the strength of the branching heuristic at each node. Many parallel depth-first search patterns arise naturally from our algorithm. Our algorithm is able to produce near perfect or super linear algorithmic efficiencies on all problems tested. Real speedups using 8 threads ranges from 4-5 times speedup to super linear speedup.

1 Introduction

In parallel constraint solving, work stealing has often been seen only as a mechanism for keeping processors occupied. Analysis of work stealing schemes often assume that the amount of work to be done is fixed and independent of the work stealing scheme, e.g. [1]. While this is true for certain kinds of problems, e.g. finding all solutions, proving unsatisfiability, it is not true for others, e.g. finding the first solution, finding the optimal solution. Such analyses fail to account for the fact that the place from which work is stolen determines the search strategy and can have a dramatic effect on the efficiency of the parallel algorithm. Many systems choose to steal from as close to the root of the search tree as possible, e.g. [2], as this tends to give the greatest granularity. However, this is not always the best place to steal from in terms of the efficiency of the algorithm.

We illustrate how work stealing from different places can have different effects on efficiency with two examples. Let us consider a relatively simple framework for parallel search. One thread begins with ownership of the entire search tree. When a thread finishes searching the subtree it was responsible for, it will pick an unexplored part of the search tree and steal that subtree off its current owner. This continues until a solution is found, or the entire search tree has been searched in the case of unsatisfiability or optimization.

Example 1. The first problem we will consider is the Travelling Salesman Problem. In our experiments, with 8 threads, stealing left and low (as deep in the tree as possible) requires visiting a number of nodes equal to the sequential algorithm, while stealing high (near the root) requires visiting ∼30% more nodes on average (see Table 1).

The explanation for this is simple. Let us examine the details of one particular instance. In this instance, with the sequential algorithm, the optimal solution is found after 47 seconds of CPU time, after which the algorithm spends another ∼300 seconds proving that no better solution exists. When work stealing is done as left and low as possible in the parallel search, all of the threads are working towards finding that leftmost optimal solution, and the optimal solution is found in 47 seconds of total CPU time as before (wall clock time ∼6 seconds). After this, the search takes another 300 seconds of CPU time to conclude. Thus we have perfect linear speedup both in finding the optimal solution, and in proving that no better solution exist.

If we steal high however, only 1 of the threads is actually exploring the leftmost part of the search tree and working towards that leftmost optimal solution. The other 7 threads are off searching other parts of the search tree, unfruitfully in this case. This time, the optimal solution is found in 47 seconds of wall clock time (376 seconds of CPU time!). The algorithm then spends another 200 seconds of CPU time proving that no better solution exists. What has happened is that we got no speedup whatsoever for finding the optimal solution, but linear speedup for proving that no better solution exists. Since we found the optimal solution so much later in the search (376 seconds CPU time instead of 47 seconds), the threads spent an enormous amount of CPU time searching without the pruning benefits of the optimal solution, thus the total number of nodes searched in this instance is dramatically increased, leading to a great loss of efficiency. Clearly, this effect gets worse as a higher number of threads is used. □

It may appear from this example that stealing left and low would be efficient for all problems. However, such a strategy can produce at best linear speedup.

Example 2. The second problem we will consider is the n-Queens problem. The search tree is very deep and a top level mistake will not be recovered from for hours. Stealing low in parallel search solves the instance within the time limit if and only if the sequential depth first search solved it within the time limit. This only occurred when a solution falls in the very leftmost part of the search tree (only 4 instances out of 100 tested, see Table 2). Stealing high, in contrast, allows many areas of the search tree to be explored, so a poor choice at the root of the search tree is not as important. Stealing high results in solving 100 out of 100 instances tested. This is clearly far more robust than stealing low, producing greatly super-linear speedup. □

Veron *et al* [3] claims that linear and super linear speedups can be expected for branch and bound problems, but they fail to note that finding the optimal solution does not parallelize trivially as shown by Example 1. Rao and Kumar [4] (and others) show that super linear speedup ought to be consistently attainable for finding the first or the optimal solution for certain types of problems. Their analysis is valid if the search tree is random (i.e. we have no idea how the solutions

are distributed), but is not valid in systems where the branching heuristic orders the branches based on their likelihood of yielding a solution. The presence of such a branching heuristic makes linear speedup in finding solutions non-trivial. Gendron and Crainic [5] describe the issue and provide a description how the issue is handled in several systems. In general, the solutions utilise some kind of best-first criterion to guide how the problem is split up (see e.g. [6, 7]).

Our contributions in this paper are as follows. We perform a quantitative analysis of how different work stealing strategies affect the total amount of work performed and explain the relationship between branching heuristic strength and the optimal search strategy. We propose an adaptive work stealing algorithm that, when provided with a user given *confidence*, which is the estimated ratio of solution densities between the left and right subtrees at each node, will work steal in a near optimal manner. We show that confidence based work stealing leads to very good algorithmic efficiencies, i.e. it does not visit many more nodes, and sometimes much less, than sequential DFS (Depth First Search).

Although our analysis is done in the context of work stealing in parallel constraint programming systems, the analysis is actually about the relationship between branching heuristic strength and the optimal search order in the search tree created by that branching heuristic. Thus the analysis actually applies to all complete tree search algorithms whether sequential or parallel. As we will show later, when the assumptions about branching heuristic strength that lie behind standard sequential algorithms such as DFS, Interleaved Depth First Search (IDFS), Limited Discrepancy Search (LDS) or Depth-bounded Discrepancy Search (DDS) is given to our algorithm as *confidence* estimates, our algorithm automatically produces the exact same search patterns used in those algorithms. Thus our analysis and algorithm provides a framework which explains/unifies/produces all those standard search strategies. In contrast to the standard sequential algorithms which are based on rather simplistic assumptions about how branching heuristic strength varies in different parts of the search tree, our algorithm can adapt to branching heuristic strength on a node by node basis, potentially producing search patterns that are vastly superior to the standard ones. Our algorithm is also fully parallel and thus we have automatically parallelised DFS, IDFS, LDS and DDS as well.

The layout of the paper is as follows. In section 2 we perform a quantitative analysis of optimal work stealing. In section 3 we describe our adaptive work stealing algorithm. In section 4 we give examples of the behaviour of our algorithm. In section 5 we present our experimental evaluation. Finally in section 6 we conclude.

2 Analysis of Work Allocation

In this section we show quantitatively that the strength of the branching heuristic determines the optimal place to work steal from. We will concentrate on the case of solving a satisfaction problem. The case for optimization is related since it is basically a series of satisfaction problems.

Preliminary definitions. A *constraint state* (C, D) consists system of constraints C over variables V with initial domain D assigning possible values $D(v)$ to

each variables $v \in V$. The *propagation solver*, solv repeatedly removes values from the domains of variables that cannot take part in the solution of some constraint $c \in C$, until it cannot detect any new values that can be removed. It obtains a new domain $\text{solv}(C, D) = D'$. If D' assigns a variable the empty set the resulting state is a *failure state*. If D' assigns each variable a single value ($|D(v)| = 1, v \in V$) then the resulting state is a *solution state*. Failure states and solution states are *final states*.

Finite domain propagation interleaves propagation solving with search. Given a current (non-final) state (C, D) where $D = \text{solv}(C, D)$ the search process chooses a *search disjunction* $\vee_{i=1}^{n} c_i$ which is consequence of the current state $C \wedge D$. The child states of this state are calculated as $(C \wedge c_i, \text{solv}(C \wedge c_i, D)), 1 \leq i \leq n$. Given a root state (C, D), this defines a *search tree* of states, where each non-final state is an internal node with children defined by the search disjunction and final states are leaves.

The *solution density* of a search tree T with x nodes and y solution state nodes is y/x. The solution density is the inverse of the *mean nodes to solution* of T defined as x/y.

Optimal split for binary nodes For simplicity, assume that the cost of visiting each node in the search tree is roughly equal. Intuitively, the optimal way to perform a search is to assign all of our threads to the most promising parts of the search tree at each stage. These places are the parts of the search tree where the mean nodes to solution is lowest, or in other words where the solution density is highest. Assuming an oracle that could give us accurate solution density information, work stealing from nodes whose subtrees have the highest solution densities will be optimal. In practice however, the solution density estimates will not be perfect, thus we have to take various other factors into account. Namely:

1. Any estimate of the solution density of a subtree will have a very high error, with a substantial chance that the solution density is actually zero.
2. The real solution densities, and hence the errors in the estimate, are highly correlated between subtrees that are close together, as they share decision constraints from higher up in the tree, and these constraints may already have made solutions impossible or plentiful.
3. The solution density estimate of a subtree should decrease as nodes in that tree are examined without finding a solution. This is caused by two factors.
   (a) As the most fruitful parts of the subtree are searched, the average solution density of the remaining nodes decrease.
   (b) The correlation between solution densities between nearby subtrees mean that the more nodes have failed in that subtree, the more likely the remaining nodes are to fail as well.

We have to take these issues into account when utilizing solutions densities to determine where to work steal.

Example 3. Let T be a search tree with a binary decision at the root. Let $A = 0.6$ and $B = 0.4$ be the solution density estimates for the left and right branches of T. Assume also that the two subtrees have the same number of nodes. If we

had 8 threads, what is the optimal division of threads between the two branches such that the expected time to find a solution is lowest?

If the solution density estimate was perfect, we would simply send all 8 threads down the left branch. However, according to (1) the estimate has a very large error. Further according to (2) the solution density of the subtrees down the left branch are highly correlated. If we send all 8 threads down the left branch, and it turns out that the real solution density is small or zero, then all 8 threads end up stuck. Because of (1) and (2), it is actually better to send some of the threads down the right branch as well as long as B is not far smaller than A, for example, for values of $A = 0.6, B = 0.4$, we may wish to send 6 threads down the left branch and 2 threads down the right branch. □

Given the actual solution density probability distribution for the two branches, we can calculate the expected number of nodes searched to find a solution. We derive the expression for a simple case. Suppose the solution density probability distribution is uniform, i.e. has equal probability of being any value between 0 and S where S is the solution density estimate. Let A and B be the solution density estimates for the left and right branch respectively, and assume a proportion p and $(1 - p)$ of the processing power is sent down the left and right branch respectively. Then the expected number of nodes to be searched is given by the hybrid function (see Appendix A for the details of the calculation):

$$f(A, B, p) = \begin{cases} \frac{1}{pA}(2 + \ln(\frac{pA}{(1-p)B})) & \text{for } pA > (1-p)B \\ \frac{1}{(1-p)B}(2 + \ln(\frac{(1-p)B}{pA})) & \text{otherwise} \end{cases} \quad (1)$$

The shape of this function does not depend on the absolute values of A and B (which only serves to scale the function), but on their ratio, thus the shape is fixed for any fixed value of $r = A/(A + B)$. The value of p which minimizes this function for a given value of r is shown in Figure 1. This graph tells us the optimal way to divide up our processing power so that we have the lowest expected number of nodes to search.

As can be seen, although not linear, the optimal values of p are well approximated by the straight line $p = r$. In fact the value of the f function at $p = r$ is no more than 2% higher than the true minimum for any r over the range of $0.1 \leq r < 0.9$. For simplicity we will make this approximation from now on. This means that it is near optimal to divide the amount of processing power according to the ratio of the solution density estimate for the two branches. For example, if $r = 0.9$, which means that A is 9 times as high as B, then it is near optimal to send 0.9 of our processing power down the left branch and 0.1 of our processing power down the right. Or if $r = 0.5$, which means that $A = B$, then it is near optimal to send equal amounts of processing power down the two branches.

Define the *confidence* of a branching heuristic at each node as the ratio $r = A/(A + B)$. The branching heuristic can be considered *strong* when $r \to 1$, that is the solution density estimate of the left branch is far greater than for the right branch, or in other words, the heuristic is really good at shaping the search tree so that solutions are near the left. In this case, our analysis shows that since r is close to 1, we should allocate almost all our processing power to the left branch everytime. This is equivalent to stealing as left and as low as possible. The

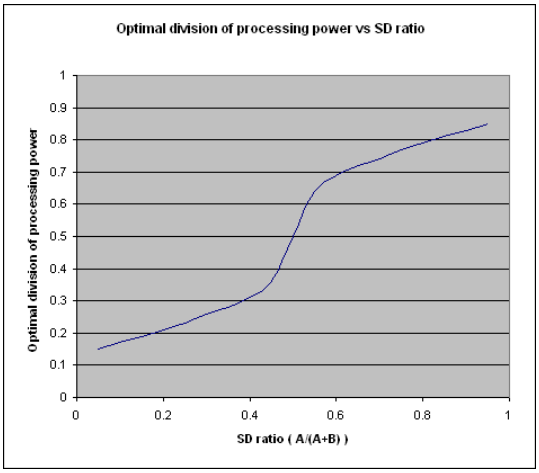


Fig. 1. Optimal division of processing power based on solution density ratio

branching heuristic is *weak* when $r \approx 0.5$, that is the solution density estimate of the left branch and right branch are similar because the branching heuristic has no clue where the solutions are. In this case, our analysis shows that since $r = 0.5$, the processing power should be distributed evenly between left and right branches at each node. This is equivalent to stealing as high as possible.[3]

3 Adaptive work stealing

Our analysis shows that the optimal work stealing strategy is dependant on the strength of the branching heuristic. Since we have a quantitative understanding of how optimal work stealing is related to branching heuristic strength, we can design a search algorithm that can automatically adapt and produce "optimal" search patterns when given some indication of the strength of the branching heuristic by the problem model. In this section, we flesh out the theory and discuss the implementation details of the algorithm in Gecode [8].

3.1 Dynamically updating solution density estimates

Now we examine how solution density estimates should be updated during search as more information becomes available.

First we need to relate the solution density estimate of a subtree with root (C, D) with the solution density estimate of its child subtrees (the subtrees rooted at its child states $(C \wedge c_i, \text{solv}(C \wedge c_i, D))$). Consider an n-ary node. Let the subtree have solution density estimate S. Let the child subtree at the ith branch have solution density estimate A_i and have size (number of nodes) x_i. If S and A_i

[3] We ignore the possibility of an *anti-heuristic* where the right branch is preferable to the left.

are estimates of average solution density, then clearly: $S = \sum_{i=1}^{n} A_i x_i / \sum_{i=1}^{n} x_i$, i.e. the average solution density of the subtree is the weighted average of the solution densities of its child subtrees.

Assuming no correlation between the solution densities of subtrees, we have that if the first k child subtrees have been searched unsuccessfully, then the updated solution density estimate is $S = \sum_{i=k+1}^{n} A_i x_i / \sum_{i=k+1}^{n} x_i$. Assuming that x_i are all approximately equal, then the expression simplifies to: $S = \sum_{i=k+1}^{n} A_i/(n-k)$. For example, suppose $A_1 = 0.3, A_2 = 0.2, A_3 = 0.1$, then initially, $S = (0.3 + 0.2 + 0.1)/3 = 0.2$. After branch 1 is searched, we have $S = (0.2 + 0.1)/2 = 0.15$, and after branch 2 is searched, we have $S = (0.1)/1 = 0.1$. This has the effect of reducing S as the branches with the highest values of A_i are searched, as the average of the remaining branches will decrease.

Now we consider the case where there are correlations between the solution density estimates of the child subtrees. The correlation is likely since all of the nodes in a subtree share the constraint C of the root state. Since the correlation is difficult to model we pick a simple generic model. Suppose the solution density estimates for each child subtree is given by $A_i = \rho A'_i$, where ρ represents the effect on the solution density due to the constraint added at the root node, and A'_i represents the effect on the solution density due to constraints added within branch i. Then ρ is a common factor in the solution density estimates for each branch and represents the correlation between them. We have that:

$$S = \frac{\sum_{i=1}^{n} A_i x_i}{\sum_{i=1}^{n} x_i} = \rho \frac{\sum_{i=1}^{n} A'_i x_i}{\sum_{i=1}^{n} x_i}.$$

Suppose that when k out of n of the branches have been searched without finding a solution, the value of ρ is updated to $\rho\frac{n-k}{n}$. This models the idea that the more branches have failed, the more likely it is that the constraint C added at the root node has already made solutions unlikely or impossible. Then when k branches have been searched, we have: $S = \rho\frac{n-k}{n} \sum_{i=k+1}^{n} A'_i x_i / \sum_{i=k+1}^{n} x_i$. Assuming that x_i are all approximately equal again, then the expression simplifies to: $S = \rho\frac{n-k}{n} \sum_{i=k+1}^{n} A'_i/(n-k) = \frac{\rho}{n} \sum_{i=k+1}^{n} A'_i = \sum_{i=k+1}^{n} A_i/n$. Equivalently, we can write it as:

$$S = \frac{\sum_{i=1}^{n} A_i}{n} \tag{2}$$

where we update A_i to 0 when branch i fails. The formula can be recursively applied to update the solution density estimates of any node in the tree given a change in solution density estimate in one of its subtrees.

In all of our results, the actual values of the solution densities are not required. We can formulate everything using *confidence*, the ratio between the solution densities of the different branches at each node. In terms of confidence, when a subtree is searched and fails the confidence values should be updated as follows:

Let r_i be the confidence value of the node i levels above the root of the failed subtree and r'_i be the updated confidence value. Let $\bar{r}_i = r_i, \bar{r}'_i = r'_i$ if the failed subtree is in the left branch of the node ith levels above the root of the failed subtree and $\bar{r}_i = 1 - r_i, \bar{r}'_i = 1 - r'_i$ otherwise. Then:

$$\bar{r}'_i = (\bar{r}_i - \prod_{k=1}^{i} \bar{r}_i)/(1 - \prod_{k=1}^{i} \bar{r}_i) \quad (3)$$

3.2 Confidence model

Given a confidence at each node, we now know how to work steal "optimally", and how to update confidences as search proceeds. But how do we get an initial confidence at each node. Ideally, the problem modeller, with expert knowledge about the problem and the branching heuristic can develop a solution density heuristic that gives us a confidence value at each node. However, this may not always happen, perhaps due to a lack of time or expertise. We can simplify things by using general confidence models. For example, we could assume that the confidence takes on an equal value *conf* for all nodes. This is sufficient to model general ideas like: the heuristic is strong or the heuristic is weak. Or we could have a confidence model that assigns $r = 0.5$ to the top d levels and $r = 0.99$ for the rest. This can model ideas like the heuristic is weak for the first d levels, but very strong after that, much like the assumptions used in DDS.

3.3 The algorithm

Given that we have a confidence value at each node, our confidence based search algorithm will work as follows. The number of threads down each branch of a node is updated as the search progresses. When a job is finished, the confidence values of all nodes above the finished subtree is updated as described in (3).

When work stealing is required, we start at the root of the tree, and use the number of threads down each branch, the confidence value, and the optimal division derived in Section 2 to work out whether the thread should be assigned to the left branch or the right branch. We then move on to that node and repeat. We continue until we find an unexplored node, at which point we steal the subtree with that unexplored node as root.

There is an exception to this. Although we may sometimes want to steal as low as possible, we cannot steal too low, as then the granularity would become too small and communication costs will dominate the runtime. Thus we dynamically determine a granularity bound under which threads are not allowed to steal, e.g. 15 levels above the average fail depth. If the work stealing algorithm guides the work stealing to the level of the granularity bound, then the last unexplored node above the granularity bound is stolen instead. The granularity bound is dynamically adjusted to maintain a minimum average job size so that work stealing does not occur more often than a certain threshold.

Since the confidence values are constantly updated, the optimal places to search next changes as search progresses. In order for our algorithm to adapt quickly, we do not require a thread to finish the entire subtree it stole before stealing again, as this could take exponential time [9]. Instead, after a given *restart* time has passed, the thread returns the unexplored parts of its subtree to the master and work steals again from the top. This is similar to the idea used in interleaving DFS [10].

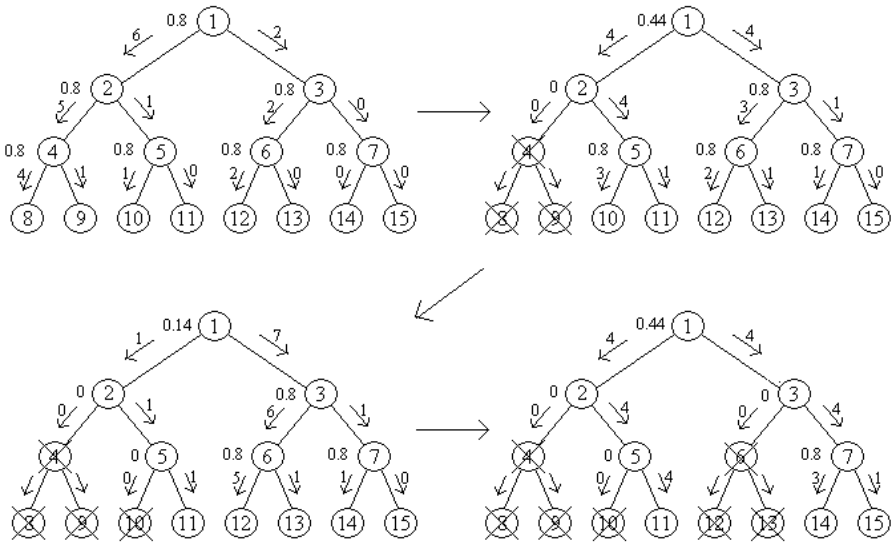


Fig. 2. Example 1

4 Sample behaviour of adaptive work stealing algorithm

In this section, we go through some examples of how the work stealing and confidence updating works in our algorithm. For the first example, suppose we know that the branching heuristic is reasonably strong, but not perfect. We may use $conf = 0.8$. Refer to Figure 2 in the explanation.

Let's suppose we have 8 threads. Initially, all the confidence values are 0.8. When the 8 threads attempt to work steal at the root, the first thread will go down the left hand side. The second thread will go down the left hand side as well. The 3rd thread will go down the right hand side. The fourth thread will go down the left hand side, etc, until we end up with 6 threads down the left and 2 threads down the right. At node 2, we will have 5 threads down the left and 1 thread down the right. At node 3, we will have 2 threads down the left, and so on. The work stealing has strongly favored sending threads towards the left side of each node because of the reasonably high confidence values of 0.8.

Suppose as search progresses the subtree starting at node 4 finishes without producing a solution. Then we need to update the confidence values. Using (3), the confidence value at node 2 becomes 0, and the confidence value at node 1 becomes 0.44. Now when the threads work steal from the root, things are different. Since one of the most fruitful parts of the left branch has been completely searched without producing a solution, it has become much less likely that there is a solution down the left branch. The updated confidence value reflects this. Now the threads will be distributed such that 4 threads are down the left branch and 4 threads are down the right branch. Next, perhaps the subtree starting at node 10 finishes. The confidence value at node 5 then becomes 0, the confidence value at node 2 remains 0 and the confidence value at node 1 becomes 0.14.

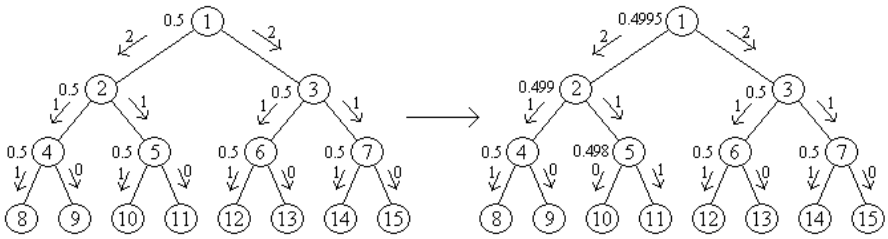


Fig. 3. Example 2

The vast majority of the fruitful places in the left branch has been exhausted without finding a solution, and the confidence value at the root has been updated to strongly favor the right branch. The threads will now be distributed such that 7 threads go down the right and 1 go down the left. Next, suppose the subtree starting at node 6 finishes. The confidence value at node 3 becomes 0 and the confidence value at node 1 becomes 0.44. Since the most fruitful part of the right branch has also failed, the confidence value now swings back to favor the left branch more. This kind of confidence updating and redistribution of threads will continue on, distributing the threads according to the current best solution density estimates. In our explanation here, for simplicity we only updated the confidence values very infrequently. In the actual implementation, confidence values are updated after every job is finished and thus occur much more frequently and in much smaller sized chunks.

For the second example, suppose we knew that the heuristic was very bad and was basically random. We may use $conf = 0.5$, i.e. the initial solution density estimates down the left and right branch are equal. Refer to Figure 3 in the explanation.

Let's suppose we have 4 threads. Initially, all the confidence values are 0.5. When the 4 threads attempt to work steal at the root, the first thread will go left, then left, then left, etc. The second thread will go right, then left, then left, etc. The third thread will go left, then right, then left, etc, and the fourth thread will go right, then right, then left, etc. This distributes the threads as far away from each other as possible which is exactly what we want. However, if the search tree is deep, and the first few decisions that the threads made within its own subtree are wrong, they may still all get stuck and never find a solution. This is where the interleaving limit kicks in. After a certain time threshold is reached, the threads abandon their current search and begin work stealing from the root again. Since the confidence values are updated when they abandon their current job, they take a different path when they next work steal. For example, if the thread down node 5 abandons after having finished a subtree with root node at depth 10, then the confidence at node 5 becomes 0.498, the confidence at node 2 become 0.499, and the confidence at node 1 becomes 0.4995. Then when the thread work steals from the root, it will again go left, then right. When it gets to node 5 however, the confidence value is 0.498 and there are no threads down either branch, thus it will go right at this node instead of left like last time. The updated confidence

values has guided the thread to an unexplored part of the search tree that is as different from those already searched as possible. This always happens because solution density estimates are decremented whenever part of a subtree is found to have failed, so the confidence will always be updated to favour the unexplored parts of the search tree.

As some other examples, we briefly mention what confidence models leads to some standard search patterns. DFS: $conf = 1$, $restart = \infty$. IDFS: $conf = 1$, $restart = 1000$. LDS: $conf = 1\text{-}\epsilon$, $restart = 1$ node. DDS: $conf = 0.5$ if depth $<$ d, $1\text{-}\epsilon$ if depth $\geq$ d, $restart = 1$ node.

5 Experimental evaluation

The confidence based work stealing is implemented in Gecode 2.1.1 [8]. The benchmarks are run on a Dell PowerEdge 6850 with 4x 3.0 Ghz Xeon Dual Core Pro 7120 CPUs. 8 threads are used for the parallel search algorithm. We use a time limit of 20 min CPU time (so 2.5 min wall clock time for 8 threads), a restart time of 5 seconds, and a dynamic granularity bound that adjusts itself to try to steal no more than once every 0.5 seconds. We collected the following data: wall clock runtime, CPU utilization, communication overhead, number of steals, total number of nodes searched and number of nodes explored to find the optimal solution.

In our first set of experiments we examine the efficiency of our algorithm for two optimization problems from Gecode's example problems. The problems are: Travelling Salesman Problem (TSP), Photo and Queens-Armies. A description of these problems can be found at [8]. We use the given search heuristic (in the Gecode example file) for each, except for TSP where we try both a strong heuristic based on maximising cost reduction and a weak heuristic that just picks variables and values in order. For both Photo and TSP, we randomly generated many instances of an appropriate size for benchmarking. Only the size 9 and size 10 instances of Queen-Armies are of an appropriate size for benchmarking. We use the simple confidence model with $conf = 1$, 0.66 and 0.5. The results are given in Table 1.

It is apparent from our experiments that the hardware/OS we experimented on is highly non-ideal and does not in fact give us a linear increase in real processing speed when more processors are used. We suspect this is due to issues such as cache contention, memory contention, context switching, etc. The effect causes threads to slow down by up to 40% at 8 threads. In view of this, the primary statistics we will look at in our analysis of our algorithm will be algorithmic efficiency and the communication cost. Algorithmic efficiency minus the communication cost represents the theoretical efficiency on an ideal parallel computer. The runtime efficiency represents what you may get on a real world, non-ideal parallel computer.

It is clear that in all of our problems, runtime is essentially proportional to the number of nodes searched, and it is highly correlated to the amount of time taken to find the optimal solution. The quicker the optimal solution is found, the fewer the nodes searched and the lower the total runtime. The communication cost, which includes all work stealing and synchronisation overheads, is less than

Table 1. Experimental results for optimization problems with simple confidence model. The results show: number of problems solved in the time limit (Solved), wall clock runtime in seconds (Runtime), speedup relative to the sequential version (Speedup), and runtime efficiency (RunE) which is Speedup/8, CPU utilization (CPU%), communication overhead (Comm%), number of steals (Steals), total number of nodes explored (Nodes), the algorithmic efficiency (AlgE) the total number of nodes explored in the parallel version versus the sequential version, the number of nodes explored to find the optimal solution (Onodes), and the solution finding efficiency (SFE) the total number of nodes explored in the parallel version to find the optimal versus the sequential version. Values for Runtime, CPU%, Comm%, Steals, Nodes, and Onodes are the geometric mean of the instances solved by all 4 versions.

TSP with strong heuristic, 200 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	181	56.0	—	—	99.8%	0.0%	—	1240k	—	180k	—
1	172	11.3	4.95	0.62	94.7%	2.8%	447	1222k	1.01	218k	0.82
0.66	170	13.3	4.20	0.53	94.6%	0.5%	370	1517k	0.82	580k	0.31
0.5	160	16.2	3.45	0.43	94.2%	1.3%	533	1564k	0.80	658k	0.27

TSP with weak heuristic, 200 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	189	78.6	—	—	99.8%	0.0%	—	1.99M	—	1.59M	—
1	186	17.7	4.45	0.56	96.5%	4.0%	686	1.99M	1.00	1.59M	1.00
0.66	186	17.7	4.46	0.56	96.3%	0.4%	319	1.97M	1.01	1.60M	1.00
0.5	184	15.7	5.01	0.63	95.5%	0.8%	287	1.73M	1.15	1.39M	1.15

Photo, 200 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	173	63.9	—	—	99.9%	0.0%	—	5.01M	—	622k	—
1	152	15.5	4.12	0.52	98.0%	1.7%	636	4.93M	1.02	542k	1.15
0.66	153	15.5	4.12	0.52	97.5%	0.4%	388	4.91M	1.02	467k	1.33
0.5	152	15.4	4.15	0.52	97.7%	0.4%	253	4.90M	1.02	492k	1.26

Queen Armies, 2 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	2	1146	—	—	99.7%	0.0%	—	27.1M	—	800k	—
1	2	219	5.24	0.65	98.7%	1.1%	2519	28.8M	0.94	1669k	0.48
0.66	2	213	5.38	0.67	98.2%	0.5%	1924	28.4M	0.96	1781k	0.45
0.5	2	217	5.29	0.66	98.3%	0.4%	1631	28.6M	0.95	1902k	0.42

1% for most problems, but goes up to around 3-4% for some steal low strategies. For algorithmic efficiency, we will examine each the problem in turn.

The strong heuristic in TSP is quite strong. Using $conf = 1$ achieves near perfect algorithmic efficiency. Other values of *conf* clearly cause an algorithmic slowdown. The optimal solution is found on average 2.7 and 3.0 times slower for $conf = 0.66$ and 0.5 respectively, resulting in an algorithmic efficiency of 0.82 and 0.80 respectively. The opposite is true when the weak heuristic is used. Using $conf = 1$ or 0.66 allows us to find the leftmost optimal solution in approximately the same number of nodes as the sequential algorithm, but using $conf = 0.5$ to reflect that the heuristic is weak allows the algorithm to find the optimal solution even faster, producing an algorithmic efficiency of 1.15 compared to the sequential algorithm.

Table 2. Experimental results for satisfaction problems with simple confidence model

n-Queens, 100 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE
Seq	4	2.9	—	—	99.9%	0.0%	—	1859	—
1	4	10.4	—	—	99.0%	86.6%	2	1845	—
0.66	29	18.0	—	—	81.6%	0.3%	9	15108	—
0.5	100	2.9	—	—	65.5%	1.6%	8	14484	—

Knights, 40 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE
Seq	7	0.22	—	—	99.9%	0.0%	—	1212	—-
1	7	0.26	—	—	68.1%	59.7%	2	1150	—
0.66	13	0.50	—	—	48.0%	4.7%	8	8734	—
0.5	21	0.66	—	—	35.2%	6.0%	8	8549	—

Perfect-Square, 100 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE
Seq	15	483.1	—	—	99.9%	0.0%	—	213k	—
1	13	72.3	6.68	0.83	98.0%	19.1%	419	216k	0.99
0.66	14	71.2	6.78	0.85	86.4%	2.9%	397	218k	0.98
0.5	82	8.9	54.02	6.75	89.0%	4.8%	21	32k	6.64

The branching heuristic in Photo is designed to minimize the size of the search tree, rather than to place the solutions on the left side of the tree, hence, it is a "weak" heuristic as far as our analysis is concerned. Using $conf = 0.66$ and 0.5 to reflect this clearly produce higher solution finding efficiency than $conf = 1$, giving 1.33 and 1.26 vs 1.15 respectively. However, for the Photo problem, there are so many optimal solutions in the search tree that one gets found extremely quickly regardless of which strategy is used, and hence finding the optimal solution faster has no real effect on total runtime.

The results for Queens-Armies show little difference depending on confidence. Clearly the heuristic is better than random at finding an optimal solution, and solution finding efficiency degrades slightly as we ignore the heuristic. But the overall nodes searched are almost identical for all confidence values.

In our second set of experiments we examine the efficiency of our algorithm for three satisfaction problems from Gecode's examples [8]. The problems are: *n*-Queens, Knights, and Perfect-Square.

The sequential version solved very few instances of *n*-Queens and Knights. Furthermore, all those solves are extremely fast (< 3 sec) and are caused by the search engine finding a solution at the very leftmost part of the search tree. Most of the time spent in those runs is from travelling down to the leaf of the search tree rather than actual search and is not parallelizable, thus comparison of the statistics for the parallel vs sequential algorithms on those instances is not meaningful as there is very little work to parallelize. The number of instances solved is the more interesting statistic and is a better means of comparison. The parallel algorithm beats the sequential algorithm by an extremely large margin in terms of the number of instances solved.

n-Queens and Knights both have very deep subtrees and thus once the sequential algorithm fails to find a solution in the leftmost subtree, it will often end up stuck effectively forever. Modelling the fact that the branching heuristic is

Table 3. Experimental results using accurate confidence values, where we follow the confidence value to degree α.

Golomb-Ruler 12

α	Nodes	AlgE
Seq	5.31M	—
1	2.24M	2.37
0.5	3.48M	1.53
0	4.27M	1.24
-0.5	10.8M	0.49
-1	10.6M	0.50

Golomb-Ruler 13

α	Nodes	AlgE
Seq	71.0M	—
1	53.2M	1.34
0.5	57.6M	1.23
0	61.9M	1.15
-0.5	74.8M	0.95
-1	111M	0.64

very weak at the top by using $conf = 0.5$ clearly produce a super linear speedup. The parallel algorithm solves 100 out of 100 instances of n-Queens compared to 4 out of 100 instances for the sequential algorithm or the parallel algorithm with $conf = 1$. The speedup cannot be measured as the sequential algorithm does not terminate for days when it fails to find a solution quickly. Similarly the parallel algorithm with $conf = 0.5$ solved 21 instances of Knights compared to 7 for the sequential and the parallel version with $conf = 1$.

Perfect Square's heuristic is better than random, but is still terribly weak. Using $conf = 0.5$ to model this once again produces super linear speedup, solving 82 instances out of 100 compared to 15 out of 100 for the sequential algorithm. We can compare runtimes for this problem as the sequential version solved a fair number of instances and those solves actually require some work (483 sec on average). The speedup in this case is 54 using 8 threads.

So far, we have tested the efficiency of our algorithm using simple confidence models where the confidence value is the same for all nodes. This is the most primitive way to use our algorithm and does not really illustrate its full power. We expect that our algorithm should perform even better when confidence values specific to each node are provided, so that we can actually encode and utilise information like, the heuristic is confident at this node but not confident at that node, etc. In our third set of experiments, we examine the efficiency of our algorithm when node specific confidence values are provided.

Due to our lack of domain knowledge, we will not attempt to write a highly accurate confidence heuristic. Rather, we will simulate one by first performing an initial full search of the search tree to find all solutions, then produce confidence estimates for the top few levels of the search tree using several strategies like, follow the measured solution density exactly, follow it approximately, ignore it, go against it, etc, to see what effect this has on runtime. Let α quantify how closely we follow the measured confidence value and let $conf$ be the measured confidence value. Then we use the following formula for our confidence estimate: $conf' = \alpha \times conf + (1 - \alpha) \times 0.5$. If $\alpha = 1$, then we follow it exactly. If $\alpha = -1$, we go against it completely, etc. We use the Golomb-Ruler problem (see [8]) for our experiment as the full search tree is small enough to enumerate completely. The results are shown in Table 3.

The results show that using confidence values that are even a little biased towards the real value is sufficient to produce super linear speedup. And not surprisingly, going against the real value will result in substantial slowdowns.

6 Conclusion

By analysing work stealing schemes using a model based on solution density, we were able to quantitatively relate the strength of the branching heuristic with the optimal place to work steal from. This leads to an adaptive work stealing algorithm that can utilise confidence estimates to automatically produce "optimal" work stealing patterns. The algorithm produced near perfect or better than perfect algorithmic efficiency on all the problems we tested. In particular, by adapting to a steal high, interleaving search pattern, it is capable of producing super linear speedup on several problem classes. The real efficiency is lower than the algorithmic efficiency due to hardware effects, but is still quite good at a speedup of at least 4-5 at 8 threads. Communication costs are negligible on all problems even at 8 threads.

References

1. Kumar, V., Rao, V. N.: Parallel depth first search. Part II. Analysis. In: International Journal of Parallel Programming, 16(6):501–519, (1987)
2. Schulte, C.: Parallel search made simple. In: Beldiceanu, N., Harvey, W., Henz, M., Laburthe, F., Monfroy, E., Müller, T. (eds.) Proceedings of TRICS: Techniques for Implementing Constraint Programming Systems, a post-conference workshop of CP 2000, Singapore, September (2000)
3. Veron, A., Schuerman, K., Reeve, M., Li, L.: Why and how in the ElipSys Or-Parallel CLP system. In: Proceedings of PARLE'93, pp. 291–302. Springer-Verlag (1993)
4. Rao, V. N., Kumar, V.: Superlinear Speedup in State-Space Search, Technical Report, AI Lab TR88-80, University of Texas at Austin June (1988)
5. Gendron, B., Crainic, T.G.: Parallel branch-and-bound algorithms: Survey and Synthesis. In: Operations Research 42, pp. 1042–1066. (1994)
6. Quinn, M. J.: Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer. In: IEEE Trans. Comp. 39(3), pp. 384–387 (1990)
7. Mohan, J.: Performance of Parallel Programs: Model and Analyses. Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Penn. (1984)
8. Gecode. `www.gecode.org`
9. Perron, L.: Search procedures and parallelism in constraint programming. In: Jaffar, J., editor, Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (1999)
10. Meseguer, P.: Interleaved depth-first search. In: International Joint Conference on Artificial Intelligence, volume 2, pp. 1382–1387, August (1997)

A Expected Search Time Calculation

Calculating the expected search time requires a double integration of a hybrid function. Since this is rather difficult we will pick a simple solution density probability distribution. Suppose the solution density probability distribution is uniform, i.e. when the solution density estimate is A, there is an equal chance of the actual value being anything from 0 to A. The real solution density values are actually discrete as there can only be a whole number of solutions, but for ease of integration we leave it as a continuous function. This probability distribution satisfies our criteria that there is a large error in the estimate and that there is a substantial chance that it is actually 0. Suppose the solution density estimates for the left and right branch are A and B respectively, and that they each have n nodes. Suppose also the solutions are randomly distributed among the n nodes. We know that when there are m items randomly located in n locations, the expected number of locations to look before we find one of them is given by $\frac{n+1}{m+1}$. Thus if a is the real solution density in the left branch, the expected time to find a solution in the left branch is $\frac{n+1}{an+1} \approx \frac{n}{an+1} = \frac{1}{a+\frac{1}{n}}$. Suppose we divide up our processing power such that p units is sent down the left branch, and $(1-p)$ units is sent down the right branch. The expected number of nodes searched will depend on which of the branches yield a solution first, thus for real solution density values of a and b for the left and right branch, it is given by the hybrid function:

$$\min\left(\frac{1}{p(a+\frac{1}{n})}, \frac{1}{(1-p)(b+\frac{1}{n})}\right) \tag{4}$$

The expected number of nodes to be searched for solution density estimates A and B for the left and right branch respectively, given a uniform solution density probability distribution will then be given by:

$$\frac{1}{AB}\int_0^A \int_0^B \min\left(\frac{1}{p(a+\frac{1}{n})}, \frac{1}{(1-p)(b+\frac{1}{n})}\right) db\, da \tag{5}$$

To evaluate this, we need to split the integral into two domains corresponding to the two halves of the hybrid function. The boundary of the hybrid function is given by:

$$\begin{aligned} & p(a+\tfrac{1}{n}) = (1-p)(b+\tfrac{1}{n}) \\ \Rightarrow \quad & a = \tfrac{1-p}{p}(b+\tfrac{1}{n}) - \tfrac{1}{n} \\ \text{or} \quad & b = \tfrac{p}{1-p}(a+\tfrac{1}{n}) - \tfrac{1}{n} \end{aligned}$$

There are four cases depending on whether the boundary of the hybrid function intersects the a or the b axis and whether it intersects the $a = A$ line or the $b = B$ line. For $p > 0.5$ and $p(A+\frac{1}{n}) > (1-p)(B+\frac{1}{n})$, which corresponds to intersecting the b axis and the $b = B$ line, we have:

$$
\begin{aligned}
&\tfrac{1}{AB}\int_0^A\int_0^B \min(\frac{1}{p(a+\frac{1}{n})},\frac{1}{(1-p)(b+\frac{1}{n})})dbda\\
=\ &\tfrac{1}{AB}[\int_{\frac{p}{(1-p)n}-\frac{1}{n}}^{B}\int_0^{\frac{1-p}{p}(b+\frac{1}{n})-\frac{1}{n}}\frac{1}{(1-p)(b+\frac{1}{n})}dadb+\\
&\int_0^{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}}\int_0^{\frac{p}{1-p}(a+\frac{1}{n})-\frac{1}{n}}\frac{1}{p(a+\frac{1}{n})}dbda+\\
&\int_{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}}^{A}\int_0^{B}\frac{1}{p(a+\frac{1}{n})}dbda\\
=\ &\tfrac{1}{AB}[\int_{\frac{p}{(1-p)n}-\frac{1}{n}}^{B}\frac{1}{p}-\frac{1}{(1-p)n(b+\frac{1}{n})}db+\\
&\int_0^{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}}\frac{1}{1-p}-\frac{1}{pn(a+\frac{1}{n})}da+\\
&\int_{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}}^{A}\frac{B}{p(a+\frac{1}{n})}da\\
=\ &\tfrac{1}{AB}[\,[\frac{B}{p}-\frac{1}{(1-p)n}\ln(B+\frac{1}{n})-\frac{1}{(1-p)n}+\frac{1}{pn}+\frac{1}{(1-p)n}\ln(\frac{p}{(1-p)n})]+\\
&[\frac{B+\frac{1}{n}}{p}-\frac{1}{(1-p)n}-\frac{1}{pn}\ln(\frac{1-p}{p}(B+\frac{1}{n}))+\frac{1}{pn}\ln(\frac{1}{n})]+\\
&[\frac{B}{p}\ln(A+\frac{1}{n})-\frac{B}{p}\ln(\frac{1-p}{p}(B+\frac{1}{n}))]]\\
=\ &\tfrac{1}{AB}[\,\frac{2(B+\frac{1}{n})}{p}-\frac{1}{n}\ln(\frac{1-p}{p}(nB+1))+\\
&\frac{B}{p}\ln(\frac{p}{1-p}\frac{(nA+1)}{(nB+1)})-\frac{2}{(1-p)n}-\frac{1}{(1-p)n}\ln(\frac{1-p}{p}(Bn+1))]
\end{aligned}
$$

If we are reasonably high up in the search tree, which is where the results of this calculation is most important, then we can assume that we are expecting a potentially large number of solutions down each branch, e.g. $An, Bn \gg 1$. In that case, all of the terms containing $1/n$ are much smaller than the terms containing A or B and the expression simplifies to:

$$\frac{1}{AB}[2\frac{B+\frac{1}{n}}{p} - \frac{1}{n}\ln(\frac{1-p}{p}(nB+1)) + \tag{6}$$
$$\frac{B}{p}\ln(\frac{p}{1-p}\frac{(nA+1)}{(nB+1)}) - \frac{2}{(1-p)n} - \frac{1}{(1-p)n}\ln(\frac{1-p}{p}(Bn+1))]$$
$$= \frac{1}{AB}[\frac{2B}{p} + \frac{B}{p}\ln(\frac{pA}{(1-p)B})]$$
$$= \frac{1}{pA}(2+\ln(\frac{pA}{(1-p)B})) \tag{7}$$

The calculation for the case $p < 0.5$ and $p(A+\frac{1}{n}) > (1-p)(B+\frac{1}{n})$ is similar, and after the simplification, yields the same equation as (6). Since the problem is symmetric with respect to A and B, and p and $(1-p)$, we can trivially derive the equation for the other two cases, which is:

$$\frac{1}{(1-p)B}(2+\ln(\frac{(1-p)B}{pA})) \tag{8}$$

Thus the full function for calculating the expected number of nodes searched given A, B and p is given by the hybrid function:

$$f(A,B,p) = \begin{cases} \frac{1}{pA}(2+\ln(\frac{pA}{(1-p)B})) & \text{for } pA > (1-p)B \\ \frac{1}{(1-p)B}(2+\ln(\frac{(1-p)B}{pA})) & \text{otherwise} \end{cases}$$

An Efficient Term Representation for CHR Indexing

Beata Sarna-Starosta[1] and Tom Schrijvers*[2]

[1] LogicBlox Inc., Atlanta, Georgia, USA
bss@logicblox.com
[2] Department of Computer Science, K.U.Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract. The overhead of matching CHR's multi-headed rules is alleviated by constraint store indexing. The attributed variable interface provides efficient means of indexing on logical variables. Current state-of-the-art indexing strategies for ground terms use hash tables. However, the hash tables incur considerable performance overhead, especially when frequently computing hash values for large terms.
We propose a high-level approach which improves the efficiency of ground term indexing. In this approach, we introduce a new data representation for ground terms, inspired by attributed variables, that avoids the overhead of hash-table indexing. The experimental evaluation establishes the usefulness of our representation, but indicates a high cost of mapping between this representation and Prolog's standard terms. Thus, we reuse previously implemented post-processing program transformations to compensate for this overhead. We compare our approach with the current state of the art, and give measurements of its effectiveness in the K.U.Leuven CHR system.
keywords: Constraint Handling Rules, indexing, program transformation, term representation, attributed variables

1 Introduction

Constraint Handling Rules (CHR) [4] is a high-level rule-based declarative programming language, usually embedded in a host language such as Prolog or Haskell. Typical applications of CHR include scheduling [1] and type checking [14]. CHR features *multi-headed rules*, i.e., rules with multiple predicates on the left-hand side (the *head*), which sets it apart from conventional declarative languages, e.g., Prolog or Haskell, where a rule's head admits only one predicate or function.

Multi-headed rules afford much of CHR's expressive power by allowing to easily combine information from distinct constraints via matching. However, as the matching procedure significantly affects the complexity of rule evaluation [5], this source of expressiveness often leads to performance bottlenecks. This effect is

* Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen)

borne out by the approximative complexity formula of [5], where the multiplicity of rule's head appears in the exponent.

Aware of this problem, CHR developers have built data structures supporting efficient indexing on variables (attributed variables [7]) and ground data (search trees [8]). With [12] came the realization that $\mathcal{O}(1)$ indexing is essential for implementing CHR algorithms with optimal complexity, which led to the use of hash tables for indexing ground data, and the general result that the complexity of CHR systems equals that of RAM machines [13]. CHRd [9] has slimmed the original attributed variable indexing for faster evaluation of the class of direct-indexed CHR and use in a tabulated environment.

In this paper we advance the research on CHR indexing with a high-level approach to efficient indexing on ground terms. Specifically, we make the following contributions:

- propose an alternative to hash tables for indexing ground data, which does not suffer from amortization-related overhead (Section 3),
- reuse previously developed post-processing program transformations [10] to reduce the disadvantages of the new approach (Section 4),
- demonstrate the measurements of the usefulness of the presented technique in K.U.Leuven CHR system (Section 5), and
- provide an implementation of the presented techniques (available online at `http://www.cs.kuleuven.be/~toms/CHR/AttributedData/`).

The presentation begins with an overview of CHR and indexing in Section 2. Section 3 describes our new representation for ground terms, the conversions between the new representation and Prolog terms, and the program transformation for introducing these conversions. Section 4 discusses the overhead of the conversions, and treats it with the post-processing program transformation. Section 5 presents the experimental evaluation of the proposed transformations, Section 6 relates our approach to other work, and Section 7 concludes.

2 Preliminaries

CHR is a language of multi-headed rewriting rules that is particularly well-suited for specifying custom constraint solvers at a high-level. A CHR program prescribes the transformations of a *constraint store* (a collection of *user-defined* constraints), based on the *built-in* constraints of the host language. For the purpose of this paper we consider Prolog as the host language; the built-in constraints are Prolog predicates and equations (unifications) of Herbrand terms.

CHR Syntax. A CHR program is a finite set of rules of the form:

label `@` *Head* `?=>` *Guard* `|` *Body*

The *label* names the rule and may be omitted along with the trailing `@`. The arrow `?=>` denotes the kind of transformation a rule defines, and may be either `<=>` or `==>` (we use `?=>` as a shorthand notation for both forms). There are

```
:- chr_constraint arrow/2, merge/2.

pick @ merge(N,A), merge(N,B) <=> A<B | M is N+1, arrow(A,B), merge(M,A).
join @ arrow(X,A) \ arrow(X,B) <=> A<B | arrow(A,B).
```

Table 1. An example CHR program encoding the merge-sort algorithm

three kinds of CHR rules. The most general are *simpagation* rules of the form: $H_1 \setminus H_2$ `<=>` $G \mid B$, where H_1 and H_2 are sequences of user-defined constraint terms (the d constraint terms. A rule specifies that when constraints in the store match H_1 and H_2 and the guard G holds, the constraints that match H_2 can be *replaced* by the corresponding constraints in B. The literal `true` represents an empty sequence of constraint terms. The guard part, $G \mid$, may be omitted when G is empty.

A *simplification* rule, which has the form: H_2 `<=>` $G \mid B$, specifies that when the stored constraints match the head, and the guard holds, the head constraints can be replaced by the body constraints. A rule of this form can be represented by a simpagation rule: `true` $\setminus H_2$ `<=>` $G \mid B$.

A *propagation* rule, which has the form: H_1 `==>` $G \mid B$, specifies that when the stored constraints match the head, and the guard holds, the body constraints can be *added* to the store. A rule of this form can be represented by a simpagation rule: $H_1 \setminus$ `true` `<=>` $G \mid B$.

Example 1. Consider the CHR program in Table 1. The simplification rule `pick` states that each pair of stored constraints matching `merge(N,A)` and `merge(N,B)` such that `A<B` should be replaced with the pair of constraints `arrow(A,B)` and `merge(M,A)` where `M=N+1`. The simpagation rule `join` states that, in the presence of two constraints `arrow(X,A)` and `arrow(X,B)` such that `A<B`, the constraint `arrow(X,B)` should be replaced by `arrow(A,B)`.

The program, by Thom Frühwirth, encodes the classical merge-sort algorithm. The algorithm is executed in the bottom-up fashion: the `pick` rule selects two sublists of elements at the same level for merging, whereas the `join` rule merges two selected sublists together.

CHR Semantics. CHR has a well-defined declarative as well as operational semantics [4, 3, 9]. The declarative interpretation of a CHR program is given by the set of universally quantified formulas corresponding to the CHR rules, and an underlying consistent constraint theory.

The original operational interpretation of a CHR program [4] is a non-deterministic transition system. The transitions are made when an unsolved constraint is added to the store, or by firing any applicable program rule.

The refined operational semantics [3][3] defines a more deterministic transition system, specifying, among others, that rules are tried in textual order. An ex-

[3] followed by most CHR implementations

	$\langle[\underline{\texttt{merge}(1,80)},\texttt{merge}(1,40),\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \emptyset\rangle$	(1)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(1,40)},\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \{\texttt{merge}(1,80)\}\rangle$	(2)
$\rightarrowtail_{\texttt{pick}}$	$\langle[\underline{\texttt{arrow}(40,80)},\texttt{merge}(2,40),\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \emptyset\rangle$	(3)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(2,40)},\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \{\texttt{arrow}(40,80)\}\rangle$	(4)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(1,50)},\texttt{merge}(1,70)],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40)\}\rangle$	(5)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(1,70)}],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40),\texttt{merge}(1,50)\}\rangle$	(6)
$\rightarrowtail_{\texttt{pick}}$	$\langle[\underline{\texttt{arrow}(50,70)},\texttt{merge}(2,50)],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40)\}\rangle$	(7)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(2,50)}],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40),\texttt{arrow}(50,70)\}\rangle$	(8)
$\rightarrowtail_{\texttt{pick}}$	$\langle[\underline{\texttt{arrow}(40,50)},\texttt{merge}(3,40)],\ \{\texttt{arrow}(40,80),\texttt{arrow}(50,70)\}\rangle$	(9)
$\rightarrowtail_{\texttt{join}}$	$\langle[\underline{\texttt{arrow}(50,80)},\texttt{merge}(3,40)],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50)\}\rangle$	(10)
$\rightarrowtail_{\texttt{join}}$	$\langle[\underline{\texttt{arrow}(70,80)},\texttt{merge}(3,40)],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50)\}\rangle$	(11)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(3,40)}],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50),\texttt{arrow}(70,80)\}\rangle$	(12)
$\rightarrowtail^*$	$\langle[\,],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50),\texttt{arrow}(70,80),\texttt{merge}(3,40)\}\rangle$	(13)

Table 2. An example derivation for the merge-sort program

tended version of the same transition system is used by the set-based operational semantics [9].

Example 2. The merge-sort program from Example 1 constructs a sorted list from a collection of sorted sublists. The head of a sorted sublist is given by means of a `merge(L,N)` constraint, where $2^{\texttt{L}-1}$ is the sublist's length and `N` is the sublist's first element. The `arrow/2` constraints model the edges between the nodes of a sorted sublist.

Table 2 outlines an example derivation for the program under the refined operational semantics. For the clarity of the presentation, the irrelevant transitions and the parts of the execution state not affected by the derivation have been omitted. For each presented derivation step, the table shows the current goal, with the active constraint underlined, and the contents of the constraint store. In the initial goal each sublist consists of a single element, and hence all sublists have the same length (equal to 2^{1-1}). The nodes are collected in the constraint store until two same-length nodes match the head of the `pick` rule. The rule transforms such two nodes into a sorted sublist and increments the length. The `join` rule sorts the nodes within each individual sublist. At the end of the derivation, the constraint store contains a collection of `arrow/2` constraints representing the sorted list.

CHR Indexing. Indexing in CHR facilitates retrieval of suspended constraints to match partner constrains in rule heads. Efficient (constant-time) constraint store indexing has been traditionally implemented by means of attributed variables [6], which provide a way to associate Prolog variables with mutable data represented as arbitrary terms. In the context of CHR, a variable's attribute corresponds to those stored constraints, in which the variable is involved. The

attribute term has the form: `attr(`$Index_1$`,...,`$Index_n$`)`, where each $Index_i$ is a data structure, typically a list, that contains all constraints on the variable with a particular constraint symbol. The presence of all variable's constraints in its attribute expedites matching when the variable is shared among the constraints in the heads of the rules.

Constraint store indexing based on attributed variables is efficient, but not always practical—for example, it is not feasible for ground constraints, in which no variables are involved. For that reason, in addition to using variable attributes, early implementations of CHR accumulated constraints in global, unordered lists. This representation supported $\mathcal{O}(1)$-time insertion of the constraints, however, constraint lookup and deletion were—in the worst case—linear in the store size. The introduction of hash tables [12] facilitated indexing on ground data, yielding amortized constant time complexity for all operations. A hash-table constraint store is defined as an array, in which every element represents a set of colliding constraints (i.e., constraints that evaluate to the same value of the hash function). The table is initialized to a small size, and dynamically expanded whenever the number of constraints exceeds given threshold. The expansion involves replacing the current array with an array of doubled size, and re-evaluating the hash function for all elements. Frequent evaluation of the hash function, the number of colliding constraints, and the resizing operation incur constant, but potentially considerable, overhead on processing the hash tables, which makes them altogether slower than attributed variables.

3 Attributed Data

In this section, we consider constraints containing arguments that are ground terms. If such arguments are matched against each other in rule heads, then constant-time matching is realized by means of a hash-table index on these ground arguments.

As an alternative to hash tables, we propose *attributed data*, which provide $\mathcal{O}(1)$ indexing with constant factors closer to those of attributed variables. The key insight underlying our approach is that the CHR run time can internally use an attributed-variable–like representation for externally provided ground terms.

3.1 Indexing Key Declarations

In our approach, ground arguments of the constraints that are matched against each other in rule heads—and hence serve as indexing keys—are internally represented using a special data type *key type*. The programmers indicate such constraint arguments using the new annotation '`as_chr_key`'. The specifier '`+`*type* `as_chr_key` *keytype*' states that the argument in question is ground (`+`), and uses *type* as its external representation and *keytype* as its internal representation. The abstract key type for a given indexing key in a CHR program is generated automatically by the CHR compiler based on the occurrence pattern of that key in the heads of the program rules.

Example 3. In the merge-sort program from Example 1, since the second argument of `merge/2` as well as both arguments of `arrow/2` are always ground and correspond to the numbers being sorted, a programmer may decide to capture all of them using the same internal representation. Denoted as `elem_key`, this representation is declared as follows:

```
:- chr_constraint
     merge(+int,+int as_chr_key elem_key),
     arrow(+int as_chr_key elem_key,+int as_chr_key elem_key).
```

3.2 Indexing Key Representation

The instances of the new data type resemble the attribute terms of attributed variables. The key type representation, however, does not include the actual variables to avoid unnecessary indirection.

The internal representation $\mathcal{I}$ of a ground indexing key in a CHR program is a term:

$$\mathcal{I} = \texttt{key}(\mathcal{E}, \mathit{Index}_1, \ldots, \mathit{Index}_n)$$

where each Index_i is an index on an argument position of that key in a head constraint of some program rule, and $\mathcal{E}$ is the key's original external value.

The number and form of the indexes in the internal representation for a particular key is orthogonal to the use of attributed data, and is determined by the CHR compiler based on the form of the rule heads and the subset of head constraints available when looking for a matching partner. For a detailed discussion of this issue we refer the reader to Section 3.2 of [8].

For the purpose of this paper we assume that the default representation of argument indexes Index_i is a flat list of constraint suspensions, with predefined operations for adding and removing the constraints. The main structure itself can be updated (e.g. for replacing an old index with a new one) by the destructive argument update predicate `setarg/3` implemented by most Prolog systems.

Example 4. Since two of the three argument positions declared as indexing keys in Example 3 are never used to retrieve partner constraints, the CHR compiler decides that only one index—for the first argument of `arrow/2`— will be exploited to speed-up the matching of the `join` rule.

Hence, given the number 80 as the external representation, the corresponding internal representation, assuming that the single index is empty, is `key(80,[])`.

Definition 1 (Conversion Functions). *For a ground indexing key type t, the injective* conversion function ϕ *maps an external value t_ε of t onto the internal representation $t_\mathcal{I}$ of t:*

$$\phi(t_\varepsilon) = \begin{cases} h[t_\varepsilon] & \text{if } h[t_\varepsilon] \text{ is defined} \\ t_\mathcal{I} & \text{otherwise} \\ & \quad \text{such that } t_\mathcal{I} = \texttt{key}(t_\varepsilon, \emptyset_1, \ldots, \emptyset_n) \\ & \qquad \text{and } h := h[t_\varepsilon \rightarrow t_\mathcal{I}] \end{cases}$$

where h is a global hash table relating the external values of ground indexing keys to their known internal representations. The injective conversion function $\psi = \phi^{-1}$ *maps the internal representations $t_\mathcal{I}$ onto the external values $t_\mathcal{E}$:*

$$\psi(\texttt{key}(t_\mathcal{E}, Index_1, \ldots, Index_n)) = t_\mathcal{E}$$

Example 5. The following internal representations are initially computed for the list of numbers given in the query in Example 1: ϕ`(80) = key(80,[])`, ϕ`(40) = key(40,[])`, ϕ`(50) = key(50,[])`, ϕ`(70) = key(70,[])`. Figure 1 depicts these internal representations, as well as the example hash table (with a linked list of buckets) underlying ϕ.

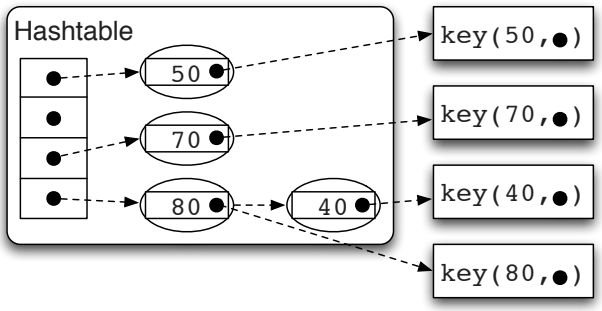


Fig. 1. Internal representation of 80, 40, 50 and 70, and hash table for ϕ.

3.3 Source-to-Source Transformation

In this section we define a source-to-source transformation for mapping between the external and internal representations of ground indexing keys. Without loss of generality, we only formalize the transformation for a single key type. Multiple keys are easily supported by repeated application of the transformation, while making sure to avoid name clashes.

The conversion rule Φ applies the conversion function ϕ at run time:

Definition 2 (Conversion Rule). *The* conversion rule Φ *replaces the external value of a ground indexing key argument t_i in a constraint term c/n with its internal representation $t' = \phi(t)$:*

$$c(t_1,\ldots,t_i,\ldots,t_n) \texttt{ <=> } t'_i = \phi(t_i),\ c'(t_1,\ldots,t'_i,\ldots,t_n).$$

Example 6. The dynamic conversion rule for the `arrow/2` constraint from the merge-sort program is of the form:

`arrow(X,Ne) <=> Ni = `ϕ`(Ne), arrow'(X,Ni).`

Definition 3 (Converted Rule). *The* converted CHR rule *is defined as:*

$$\phi(H \texttt{ ?=> } G \mid B) \;=\; H' \texttt{ ?=> } G', G \mid B$$

where

- *H' differs from H in that any constraint $c(t_1, \ldots, t_i, \ldots, t_n)$ is replaced by its converted form $c'(t_1, \ldots, x_i, \ldots, t_n)$, where x_i is a fresh variable.*
- *the new guard G' relates the original arguments of each constraint to the new ones: G' contains one t_i `=` $\psi(x_i)$ for each converted argument.*

Example 7. The converted `join` rule from the merge-sort program is of the form:

```
join' @ arrow'(X1,AI) \ arrow'(X2,BI) <=>
            X = ψ(X1), X = ψ(X2),
            A = ψ(AI), B = ψ(BI), A<B |
            arrow(A,B).
```

Definition 4 (Converted Program). *The* converted CHR program $\phi(P)$ *is defined as the set of converted rules $\overline{R}$ comprising the original program, the functions ϕ and ψ, and the encoding of Φ:*

$$\phi(P) = \phi(\overline{R}) \cup \phi \cup \psi \cup \Phi$$

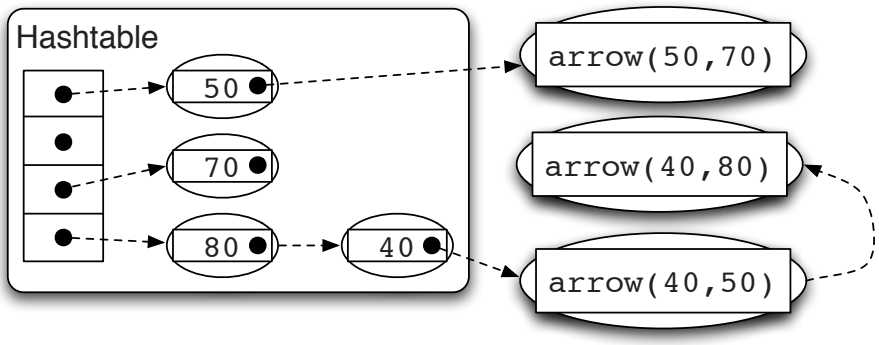
3.4 Elaborated Example

Consider the merge-sort program, and the query

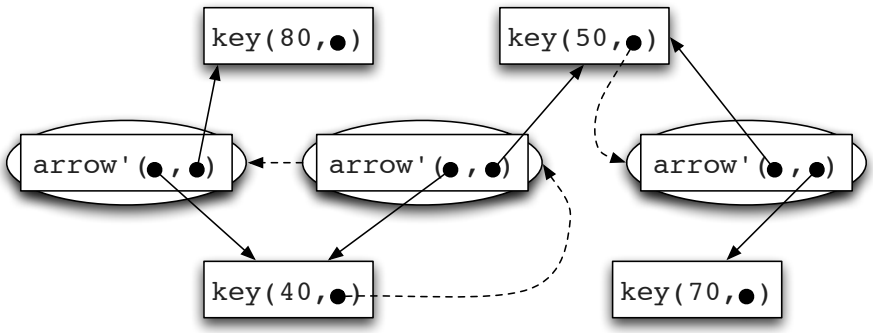
```
?- merge(1,80), merge(1,40), merge(1,50), merge(1,70).
```

evaluated as shown in Table 2. In the execution state (9), `arrow(40,50)` is the active constraint, whereas `arrow(40,80)` and `arrow(50,70)` are suspended in the constraint store. In the following derivation step, the `join` rule is triggered, and `arrow(40,80)` is retrieved from the store to serve as the partner constraint to match the rule's head.

Figure 2 illustrates two instances of this situation: (a) with indexing based on a hash table, and (b) with indexing based on attributed data. In the former case, retrieving the required partner constraint involves hashing the number 40 into the table, traversing the bucket list to find the appropriate bucket, and locating the constraint within the bucket. In the latter case, the internal representation `key(40,L)` provides direct access to the linked list containing `arrow(40,80)`. Clearly, using attributed data avoids the overhead of hashing into the table and of traversing the bucket list.



(a) Hashtable

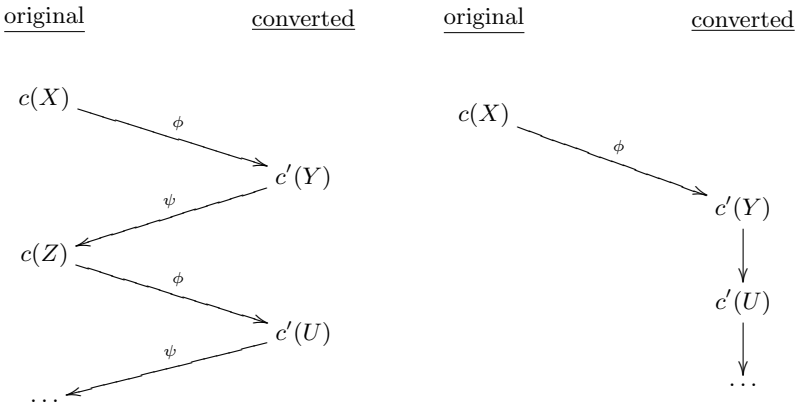


(b) Attributed Data

Fig. 2. Situation during the merge-sorting of 80, 40, 50 and 70.

4 Post-Processing

The experimental results in Section 5 indicate that the performance improvement obtained by better indexing is offset, or in some cases even surpassed, by the run-time overhead of applying the conversion functions. In this section we outline a transformation that statically eliminates most of this overhead; it was previously used to avoid similar performance issues in other transformations based on term flattening [10]. The effectiveness of the transformation is borne out by the benchmarks in Section 5.



(a) Actual Situation (b) Ideal Situation

Fig. 3. Transitions between the original and converted constraints

Alternating the conversions between the internal and external argument representations is a major source of runtime overhead. In a typical scenario (Figure 3(a)), an external value is converted into the internal representation and matched in a head of a rule, then it is converted back in the rule's body for calling a new constraint, converted again to match another rule, and so on. To avoid this overhead, the transformed rules should operate solely on the internal representation of the arguments, whereas the external values should be used only by the queries external to the programs. We propose a four-step rewriting procedure that aims to trigger this ideal scenario (Figure 3(b)). Execution of a program enhanced with the procedure consists of two phases:

(1) conversion of an argument's external value to the internal representation, and
(2) processing of the internal representation.

For all but the most trivial programs, we expect the runtime cost of (1) to be marginal with respect to the cost of (2).

Our rewriting procedure comprises the following steps.

Step 1: Make conversion explicit.
Unfold constraint calls according to the conversion rules.

Example 8. Consider the `join` rule from the program in Table 1:

```
arrow(X,A) \ arrow(X,B) <=> A < B | arrow(A,B).
```

After conversion, the rule has the form:

```
arrow'(XI1,AI) \ arrow'(XI2,BI) <=>
     X = ψ(XI1), X = ψ(XI2),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow(A,B).
```

By applying Step 1 to the above rule we obtain:

```
arrow'(XI1,AI) \ arrow'(XI2,BI) <=>
     X = ψ(XI1), X = ψ(XI2),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(φ(A),φ(B)).
```

We refer the reader to the work of Tacchella et al. [15] for the formal definition and correctness proof of unfolding of CHR rules.

Step 2: Eliminate identity conversion.
Apply the following equation from left to right:

$$\forall \bar{t} : \phi \circ \psi(\bar{t}) = \bar{t}$$

The transformation is valid based on the property that ϕ is the inverse of ψ.

Example 9. Applying Step 2 to the last rule in Example 8 yields:

```
arrow'(XI1,AI) \ arrow'(XI2,BI) <=>
     X = ψ(XI1), X = ψ(XI2),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(AI,BI).
```

Step 3: Convert external values of matchings to the internal representations.

Apply the equivalence from left to right:

$$\forall \overline{t_1}, \overline{t_2} : \psi(t_1) = \psi(t_2) \Leftrightarrow \overline{t_1} = \overline{t_2}$$

The transformation is valid based on the property that ψ is injective.

Example 10. Applying Step 3 to `X` in the rule from Example 9 yields:

```
arrow'(XI,AI) \ arrow'(XI,BI) <=>
     X = ψ(XI),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(AI,BI).
```

Step 4: Clean up.
Drop unused conversion guards and refold the unfolded constraint calls that could not be simplified.

Example 11. Applying Step 4 to the rule in Example 10 yields:

```
arrow'(XI,AI) \ arrow'(XI,BI) <=>
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(AI,BI).
```

In general, these rewriting steps are not sufficient to enforce the ideal scenario of Figure 3(b). However, as the results in Section 5 show, they have good practical effects.

5 Evaluation

We implemented our approach in K.U.Leuven CHR [11] on SWI-Prolog [16]. The implementation consists of two components: (1) a pre-processor, which transforms a CHR program with key annotations into its converted form, and (2) the actual code generator of the CHR compiler, which generates attributed data indexing instructions and emits definitions for the conversion functions. Note that the pre-processor performs the transformations for all keys simultaneously rather than sequentially. In doing so, it avoids generating multiple intermediate conversion rules for constraints involving more than one key type.

We have evaluated our implementation on several standard CHR benchmarks. All run times, given in seconds for the original programs and relative to the original for the transformed versions, were measured on a MacBook Pro Intel Core Duo 1.83 GHz, with 1 GB RAM. Our benchmark suite includes the following programs:

- `chrg`, a CHRg parser with an exponential number of parses
- `dijkstra`, Dijkstra's shortest path algorithm
- `fib`, computation of fibonacci numbers, with effective memoing
- `fib2`, computation of fibonacci numbers, with ineffective memoing
- `mergesort`, mergesort algorithm
- `flat_ram`, RAM machine interpreter, flattened by symbol specialization [10]
- `reverse`, reversing chain of list cells
- `turing`, Turing machine simulator, running the *copy* program
- `uf_opt`, optimal union-find algorithm

	index representation				
benchmark	hash table	attr. data	relative	post-processed	relative
`chrg`	2.17	2.10	96.8%	1.58	72.8 %
`flat_ram`	4.69	4.31	91.9%	2.50	53.3%
`mergesort`	3.33	4.89	146.8%	1.85	55.6 %
`reverse`	2.55	3.25	127.4%	1.92	75.3%
`uf_opt`	0.34	0.38	111.8%	0.25	73.5%
`turing`	1.50	1.31	87.3%	1.19	79.3%
`wfs`	1.32	0.88	66.7%	0.85	64.4%
`fib`	1.24	1.53	123.4%	1.52	122.6%
`fib2`	1.61	1.30	80.7%	1.05	65.2%
`dijkstra`	2.26	4.52	200.0%	3.53	156.2%

Table 3. K.U.Leuven CHR run times (in sec.) for attributed data benchmarks

– `wfs`, well-founded semantics algorithm.

For each benchmark, we have manually added the `as_chr_key` annotations for the argument positions according to the following prioritized guidelines:

1. If two head constraints share more than one variable, we do not annotate the corresponding argument positions of those variables, because they are better served by multi-argument indexing. For instance, consider a rule head of the form `c(X,Y), d(Y,X)`. Although indexing on a single argument, i.e., using either `X` or `Y`, does work, indexing on the combination of both arguments is usually more efficient.
2. If two head constraints share exactly one variable, we annotate the corresponding argument positions of that variable with the same key.
3. If no variables are shared, no index is required.

Most benchmarks require a single key type. The exceptions are `ram_flat` and `turing`, each using two key spaces to represent instruction labels/states and data addresses, and `wfs` with separate key spaces for atoms and clause identifiers.

Table 3 lists the run-time results of exploiting attributed data in K.U.Leuven CHR, measured for plain hash tables, plain attributed data, and attributed data with post-processed rule bodies.

The first block of seven benchmarks clearly shows the positive effects of our approach. Although, the attributed data used alone causes a slow-down (up to about 50% for `mergesort`), when augmented with post-processing, it improves the run time by 20% to 50%.

The second block illustrates two cases of slow-downs incurred by the use of attributed data. The first benchmark, `fib`, performs one hash-table lookup per new constraint, and the initial attributed data conversion preserves that count. Hence, the attributed data manipulation is pure overhead (25%). The second benchmark, `fib2`, modifies the simpagation rule of `fib`:

```
fib(N,F1) \ fib(N,F2) <=> F1 = F2.
```

into a simplification rule:

```
fib(N,F1), fib(N,F2) <=> F1 = F2, fib(N,F1).
```

This modification causes the parameter `N` to be reused in the new call in the rule's body. As a consequence, attributed data requires only one hash-table lookup for every two new constraints, which results in a visible speed-up.

The second slow-down, in `dijkstra`, results form a limitation of our current implementation, which does not allow multi-argument indices involving attributed data arguments. For this benchmark, such a multi-argument index would be more efficient than a single-argument attributed data index.

6 Related Work

Several programming languages define features that resemble our concept of attributed data. The `as_chr_key` annotation is related to (primary, secondary and foreign) keys in database tables and indexing declarations in some Prolog systems.

The conversion function ϕ relates to *hash consing*—a technique, originated in Lisp, for mapping to and representing terms by unique (hash) values. Although the main aim of hash consing is to reduce memory consumption by increased sharing, it is also used to speed up equality tests.

The *solver types* facility of Mercury [2] also imposes a dual view of constraint arguments. The internal representation type is defined by the library programmer, rather than generated automatically. Externally, the solver type is abstract, but coercion functions should be provided for external representations. Finally, a folklore optimization technique in C/C++ adds (pointer) fields to structures to compactly represent lists (and other data types) that contain them.

7 Conclusion

We have presented attributed data—a new term representation that facilitates improving the efficiency of CHR indexing at a high level. A complementary post-processing procedure compensates for possible overhead of conversions between the new representation and the standard representation of Prolog terms.

Our technique has been implemented for the K.U.Leuven CHR system on SWI-Prolog. Evaluation on a set of benchmarks shows that using attributed data enables performance improvement, and that post-processing is critical to fully realize this potential.

As a further optimization of the approach, we could directly expose the abstract key types in the situations when there is no preference for the external argument representation. For example, programmers often use variables and integers as identifiers in CHR constraints. The nature of the data type is of no concern, as long as it supports unique value creation and value comparison. The appropriate choice of the abstract key type could eliminate unnecessary indirections of attributed variables or hash tables.

Two other interesting avenues for future work involve introducing support for automated inference of key type annotations, and extending attributed-data indexing to combinations of multiple arguments.

Acknowledgments

We are grateful for the helpful comments of the anonymous reviewers.

References

1. Slim Abdennadher and Michael Marte. University course timetabling using constraint handling rules. *Applied Artificial Intelligence*, 14(4):311–325, 2000.
2. Ralph Becket et al. Adding constraint solving to Mercury. In *8th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2006.
3. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *20th International Conference on Logic Programming (ICLP)*, pages 90–104, 2004.
4. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
5. Thom Frühwirth. As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs. *Electronic Notes in Theoretical Computer Science*, 59(3), 2002.
6. Christian Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. Technical Report TR-92-23, Austrian Research Institute for Artificial Intelligence, Vienna, Austria, 1992.
7. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), April 2000.
8. Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming*, 5(Issue 4 & 5):503–531, 2005.
9. Beata Sarna-Starosta and C.R. Ramakrishnan. Compiling Constraint Handling Rules for Efficient Tabled Evaluation. In *9th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2007.
10. Beata Sarna-Starosta and Tom Schrijvers. Transformation-based indexing techniques for constraint handling rules. In T. Schrijvers, F. Raiser, and T. Frühwirth, editors, *CHR '08*, RISC Report Series 08-10, University of Linz, Austria, pages 3–18, Hagenberg, Austria, July 2008.
11. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In *1st Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.
12. Tom Schrijvers and Thom Frühwirth. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.
13. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules (CHR)*, pages 3–17, 2005.
14. Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transactions on Programming Languages and Systems*, 27(6):1216–1269, 2005.
15. Paolo Tacchella, Maurizio Gabbrielli, and Maria Chiara Meo. Unfolding in chr. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 179–186, 2007.
16. Jan Wielemaker. SWI-Prolog release 5.6.0, 2006. http://www.swi-prolog.org/.

About Redundant Sudoku Rules

Bart Demoen[1], María García de la Banda[2]

[1] Department of Computer Science, K.U.Leuven, Belgium
[2] Monash University, Australia

Abstract. The rules of Sudoku are often specified by twenty seven `all_different` constraints, which we will call *big* rules. It is shown that many subsets of six of these big rules are redundant, and that six is maximal. Any `all_different` constraint can be specified as a (quadratic sized) set of binary inequalities, which we will call *small* rules. The redundancy of small rules is also investigated.

1 Introduction

A very common formulation of the 3x3 Sudoku [1] rules is one in which (a) all numbers in the puzzle are said to be in [1..9] and (b) the numbers in each column, row and box are said to be different. A CLP-program would typically code the latter as 27 `all_different` constraints of 9 variables each: we will refer to these constraints as *the big rules*. We will use often the word *Sudoku* in italics as an abbreviation of the 27 big rules.

Any `all_different` constraint can also be formulated in terms of binary inequality constraints. For example, `all_different([A,B,C,D])` is the conjunction of the constraints `A` $\neq$ `B`, `A` $\neq$ `C`, `A` $\neq$ `D`, `B` $\neq$ `C`, `B` $\neq$ `D` and `C` $\neq$ `D`. We will refer to these binary $\neq$-constraints as the *small rules*. As we will see, *Sudoku* can be rewritten as 810 different small rules.

For most people it is intuitively clear that some of the small rules must be *redundant*, i.e., implied by the others. It might be less obvious which ones are redundant, let alone how many. On the other hand, often, the same people are convinced that not a single big rule is redundant. These two issues form the topic of this paper: what is the largest redundant set of big rules, and the largest redundant set of small rules.

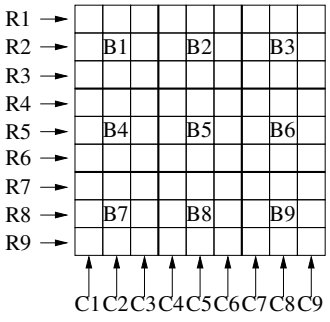
The paper proceeds as follows. We start by revising some Sudoku terminology in Section 2. In Section 3 we introduce a pictorial representation of big Sudoku rules that will make proofs much easier. In Section 4 we prove two positive lemmas that can be used to easily reason about the redundancy of subsets of the 6 big rules. In Section 5 we describe a Prolog program that systematically applies the two positive lemmas to find *all* redundant sets of six big rules. While doing this we detect 7 *negative* lemmas. This results in a complete classification. In Section 6 we turn to the study of sets of seven big rules and show that none of them are redundant. Again, our Prolog program discovers a new negative lemma, whose proof is also presented. In Section 7, we show that at least 20% of the small rules can be redundant, and conjecture that no more are possible. Finally, in Section 8 we conclude and provide some historical notes.

2 Terminology

The usual formulation of Sudoku refers to 27 regions on the board:

- the 9 rows, denoted as R1 ... R9
- the 9 columns, denoted as C1 ... C9
- the 9 boxes, denoted as B1 ... B9

as shown in the picture below. By an abuse notation we also write R3 if we mean that the `all_different` constraint on row R3 is enforced or true.



Individual cells of a puzzle are denoted as A11, A12 ... A99. We use the word horizontal (vertical) *chute* to refer to three horizontal (vertical) boxes - the usual term is band (stack). For instance, {B2, B5, B8} denotes a vertical chute. In the usual specification of Sudoku, each cell is involved in 20 small rules: 8 in the same box, 6 more in the same row and 6 more in the same column. Since there are 81 cells, and each rule is posted twice, there are in total 810 different small rules. When a set S of constraints is equivalent to the conjunction of all the Sudoku constraints, we use the short phrase: S is *Sudoku*.

3 Representing Sets of Sudoku Rules

Given the above notation, we could easily represent sets of rules as, for example, $\{R1, R2, R3, B1, B5, B9\}$. However, this only works well for small sets. Since we will be dealing mostly with sets of more than 20 big rules, we develop a graphical representation. Our representation always shows the borders of the boxes of a Sudoku board. A missing column, row or box rule will appear as a shaded column, row or box, respectively. Figure 1 shows an example.

Fig. 1. The left shows a Sudoku board with all 27 rules, the right shows one with only 22: $\{C1, C3, C4, C5, C6, C7, C8, C9, R1, R2, R3, R4, R6, R7, R8, R9, B1, B3, B4, B6, B8, B9\}$

The pictures provide a quick and intuitive insight into which rules are present and which are not. Note that the absence of a rule does not mean it is violated, simply that it has not been specified in the associated model.

We will also use $Missing(n)$ to denote the set $\{S \subseteq Sudoku | \#S = 27 - n\}$. For example, every element of $Missing(5)$ has 22 big rules.

We can in a similar way show the set of rules defined only for a given a chute: this is illustrated in Figure 2.

Fig. 2. $\{R1, R2, R3, B1, B2, B3\}$, $\{R1, R2, R3, B1, B3\}$ and $\{R1, R3, B1, B3\}$

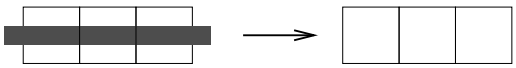
4 Two Constructive Lemma's

Lemma 1.

Proof The pictured lemma says that the set of rules $\{R1, R2, R3, B1, B3\}$ implies also $B2$. The proof can be given by trying to fill the chute with 27 numbers, so that  is fulfilled (and of course with all numbers being in [1..9]). Consider first where we can place a 5. There must be exactly one 5 in R1, one 5 in R2 and one 5 in R3, so there are in total three 5's in the chute. There is also exactly one 5 in B1, and one 5 in B3, so the remaining 5 must be in B2. And this holds also for the other numbers, so B2 is satisfied. ■

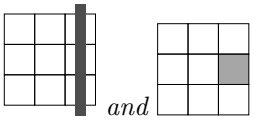
The dual of Lemma 1 is Lemma 2: we leave its proof to the reader.

Lemma 2.



The lemma  is clearly trivial and we can state the following corollary by composing the above two lemmas:

Corollary 1.

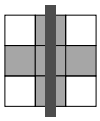
 *and* *are both Sudoku.*

Proof Glue together twice the trivial lemma with Lemma 1 or Lemma 2, and obtain the result immediately. ■

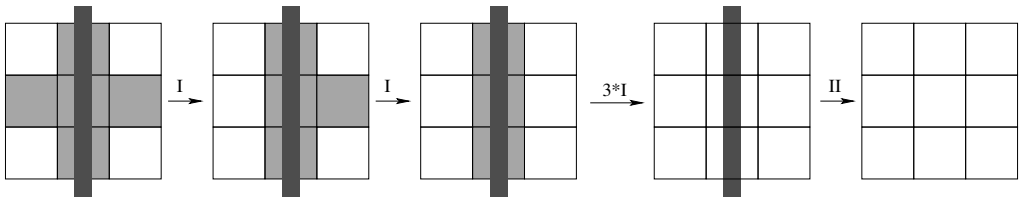
It is now clear that every single big rule is, by itself, redundant !

The two lemma's really are *constructive*: they show how to derive one new big rule from a set of big rules. The following two theorems exploit that constructive power to reason about redundancy.

Theorem 1.

 *is Sudoku.*

Proof We prove this by repeatedly using Lemma 1 and 2 as follows:



where the labels of the arcs indicate which lemma is used, and how many times it is used. ■

The proof of Theorem 2 is also left to the reader.

Theorem 2.

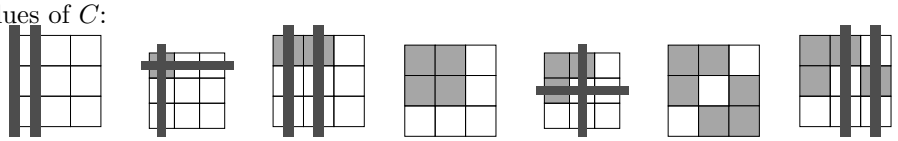
 *is Sudoku.*

The theorems show that at least two elements of $Missing(6)$ are large enough for representing *Sudoku*. Note that there are many symmetric versions of the theorems, but we have of course chosen the ones that are visually most pleasing. In the next section we will investigate all elements of $Missing(6)$ that are redundant.

5 A Full Classification of $Missing(6)$

Lemmas I and II give us a way to increase the number of big rules, as shown in the proof of Theorem 1. We use this in the algorithm of Figure 3 (where n is a parameter of the algorithm) to determine all elements of $Missing(6)$ that are redundant (i.e., those for which the algorithm will output *S is Sudoku*).

While the number of elements in $Missing(6)$ is relatively small (296,010), it is much smaller if we eliminate from $Missing(6)$ those elements that can be obtained from the spatial symmetries of Sudoku. We have programmed Algorithm I in Prolog (of course) and run it over the (reduced) set of $Missing(n)$ for values of n in 2..6. To our surprise, the algorithm only got stuck for the following seven values of C:



```
for each S ∈ Missing(n) do
        C ← copy(S)
        while Lemma 1 or Lemma 2 is applicable to C
                apply it to C
        if C == Sudoku
                then output S is Sudoku
                else output S got stuck in C
```

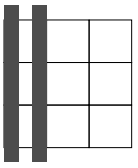
Fig. 3. Algorithm I

It is clear that if a C is not *Sudoku*, then any subset of C is not *Sudoku* either, i.e., the S from which C was derived by lemma application, is not *Sudoku*. So we set off to prove that the above sets of big rules are not *Sudoku*. This resulted in the seven negative lemmas provided in the next section.

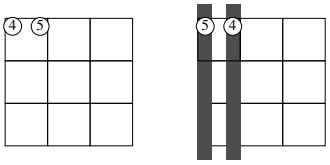
5.1 Seven *Negative* Lemmas

For each of the configurations C above, we can prove the negative result that C is not *Sudoku*. The proof of each lemma consists of a simple picture whose details we explain for the first proof. We expect the reader to work out the details for the others.

Lemma 3.


is not Sudoku.

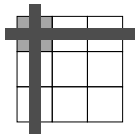
Proof



Explaining the proof: consider a completely filled out Sudoku puzzle that satisfies the full set of big rules, and which has a 4 in A11 and a 5 in A13. This situation is depicted in the left part of the proof. If we swap the 4 and 5 we obtain the picture on the right, where the shadows indicate the only two big rules that are violated by the swap. Clearly, the filled out puzzle with the two numbers swapped is not a valid solution to the full set of rules, but it only violates C1 and C3. That proves the lemma.

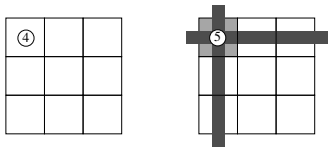
The statements and proofs of the other six negative lemmas are similar: we always start from a completely filled out Sudoku puzzle, with numbers 4 and 5 at particular places. It is easy to check that such an initial puzzle indeed exists.

Lemma 4.



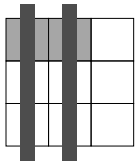
is not Sudoku.

Proof



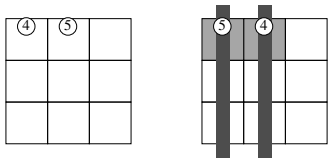
■

Lemma 5.



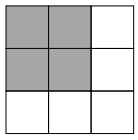
is not Sudoku.

Proof



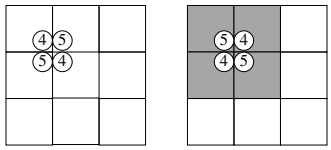
■

Lemma 6.



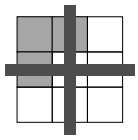
is not Sudoku.

Proof



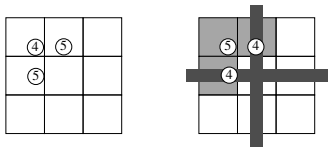
■

Lemma 7.



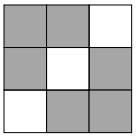
is not Sudoku.

Proof



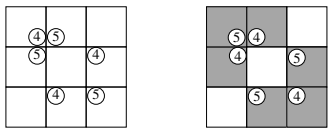
■

Lemma 8.



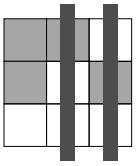
is not Sudoku.

Proof



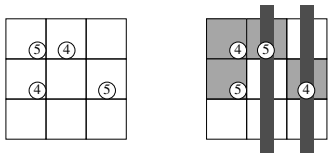
■

Lemma 9.



is not Sudoku.

Proof



■

5.2 Using the Negative Lemmas

We can increase the accuracy of our first algorithm by noticing that subsets of non-*Sudoku* are also non-*Sudoku*:

```
for each S ∈ Missing(n) do
        C ← copy(S)
        while Lemma 1 or Lemma 2 applicable to C
                apply a Lemma to C
        if C == Sudoku
                then output S is Sudoku
                else output S is not Sudoku
```

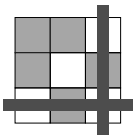
Fig. 4. Algorithm II

We have run the algorithm with $n = 6$ and, for each element S of (the reduced) $Missing(6)$, we have modified the program to generate a picture with some annotations. These are provided in the Appendix. It turns out there are 40 different elements in (the reduced) $Missing(6)$ that are *Sudoku*.

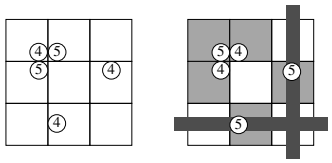
6 No Element in $Missing(7)$ is *Sudoku*

When run with $n = 7$, Algorithm I gets stuck in only one new set of big rules. This results in one more negative lemma:

Lemma 10.


is not Sudoku.

Proof



Running Algorithm II for $n = 7$ shows that no element in $Missing(7)$ is *Sudoku*. This means that six is the maximal size of a redundant set of big rules.

7 Redundant Sets of Small Rules

Recall that *Sudoku* can also be specified by 810 small rules, which are obtained by expanding the big rules to binary inequalities. We will denote the set of all small rules $Sudoku_{small}$. In this section we will briefly study the redundancy of sets of the small rules. In analogy with $Missing(n)$ which was meant for big rules, we introduce the notation $Missing_{small}(n) = \{S \subseteq Sudoku_{small} | \#S = 810-n\}$.

The output of Algorithm II for $n = 6$ (shown in the Appendix) indicates that the largest n for which an element of $Missing_{small}(n)$ is known to be $Sudoku$ is 162: indeed, the big rules of Theorem 2 give rise directly to 648 small rules. We name this set of small rules $Small_{648}$. Theorem 2 now can be read as: $Small_{648}$ *is Sudoku*.

We have used $Small_{648}$ in an experiment which needed two more ingredients. The first is a large set of difficult Sudoku puzzles. For this we took the set from Gordon Royle's website [2] who has collected a large set (more than 50.000) distinct and *minimal* Sudoku puzzles with 17 given entries. Here minimal means that while with the 17 givens the puzzle has a unique solution, if any one given is removed the puzzle has more than one solution. We refer to this set as GR.

The second ingredient is a way to transform a given Sudoku solver P to take into account less small rules. Because of the symmetries, this needs to be done only 11 times, so we did that by hand. Our P was adapted from an example CLP(FD) program from the B-Prolog [3] distribution and run under B-Prolog. Since we did not know in advance how many examples we would run, we wanted a fast CLP(FD) system. However, the programs also run in e.g., SICStus Prolog.

for each $s \in Small_{648}$ **do**
    $S \leftarrow Small_{648} \setminus \{s\}$
    transform P to take into account only S
    run P on every problem $p \in GR$
    if *some* p has more than one solution
        then output S *is not Sudoku*
        else output S *maybe is Sudoku*

Fig. 5. Algorithm III

It turns out that the algorithm could always decide that S is not $Sudoku$. This proves that the set $Small_{648}$ forms a locally minimal set of small rules equivalent to Sudoku. Another way to phrase this result is: *Sudoku only needs 80% of its small rules.*

Since the number of example problems that needed to be tried before the modified program P finds more than one second solution is so small, we dare to conjecture the following:

Conjecture 1. No element of $Missing_{small}(n)$ is $Sudoku$ for $n > 162$.

It is clear that this conjecture should not be attacked with blind and brute force.

8 Discussion and Conclusion

On 18 May 2008, in rec.puzzles, the following message was posted:

```
Quick question that I though someone here might know the
answer to - or be able to suggest a different forum.
```

```
If you have a completed sudoku grid, you supposedly need
to check all 9 rows, then all 9 columns, then all 9 boxes
to validate that it has been completed correctly. But it's
pretty obvious that the grid can be validated with somewhat
less checking. For instance, if each of the boxes has been
checked and the first 2 rows are checked, there's no need
to check the 3rd row.

So what's the minimum amount of checking that needs to be
done to show that a completed 9x9 grid is valid?
```

Before we even saw this post[1], other people tried to answer, but it was clear that none had the full picture presented here. Still, the original poster had figured out our Theorem 2 on his own, but got stuck there. It was quite satisfactory that our research started out of curiosity and ended up being of use to someone !

Redundant constraints are often important for a solver to be able to find a solution efficiently. So it might seem a futile exercise to find out whether a particular constraint satisfaction problem has redundant constraints. However, understanding better redundant Sudoku rules might give insight in why the 16-17 problem is so hard. Also, studying redundant Sudoku constraints is interesting in itself, because it seems not generally known that so many of the big and small Sudoku rules are redundant. On the other hand, it is difficult to *add* rules and stay *Sudoku*: it is clear that any additional small inequality rule changes the game.

During our discussion, one particular constraint was considered sacred: all cells have a value in $1..Max$, with $Max = 9$. It is clear that one cannot maintain any big constraint for Max strictly smaller than 9. But it seems worthwhile to investigate $Max = 10$ (or more) for the usual Sudoku constraints and for particular givens: the uniqueness of the solutions under such circumstances could result in a better understanding of Sudoku. It is also clear that our techniques can be readily applied to the investigation of Sudoku puzzles of different sizes. In particular, the generalization of our lemmas 1 and 2 to other sizes is not difficult, and the algorithms remain correct. Still, for large sizes, they might be not as helpful.

Acknowledgements Lots of this work was done while the first author was on a research visit at Monash University, and enjoying the Stuckey hospitality in Apollo Bay and Elwood, Australia. Many thanks for an enjoyable stay.

References

1. Wikipedia: Sudoku (2008) http://en.wikipedia.org/wiki/Sudoku.
2. Royle, G.: Minimal sudokus with 17 givens (2008) http://people.csse.uwa.edu.au/gordon/sudokumin.php.
3. Zhou, N.F.: B-Prolog users manual, version 6.8. Technical report, Afany Software (2005)

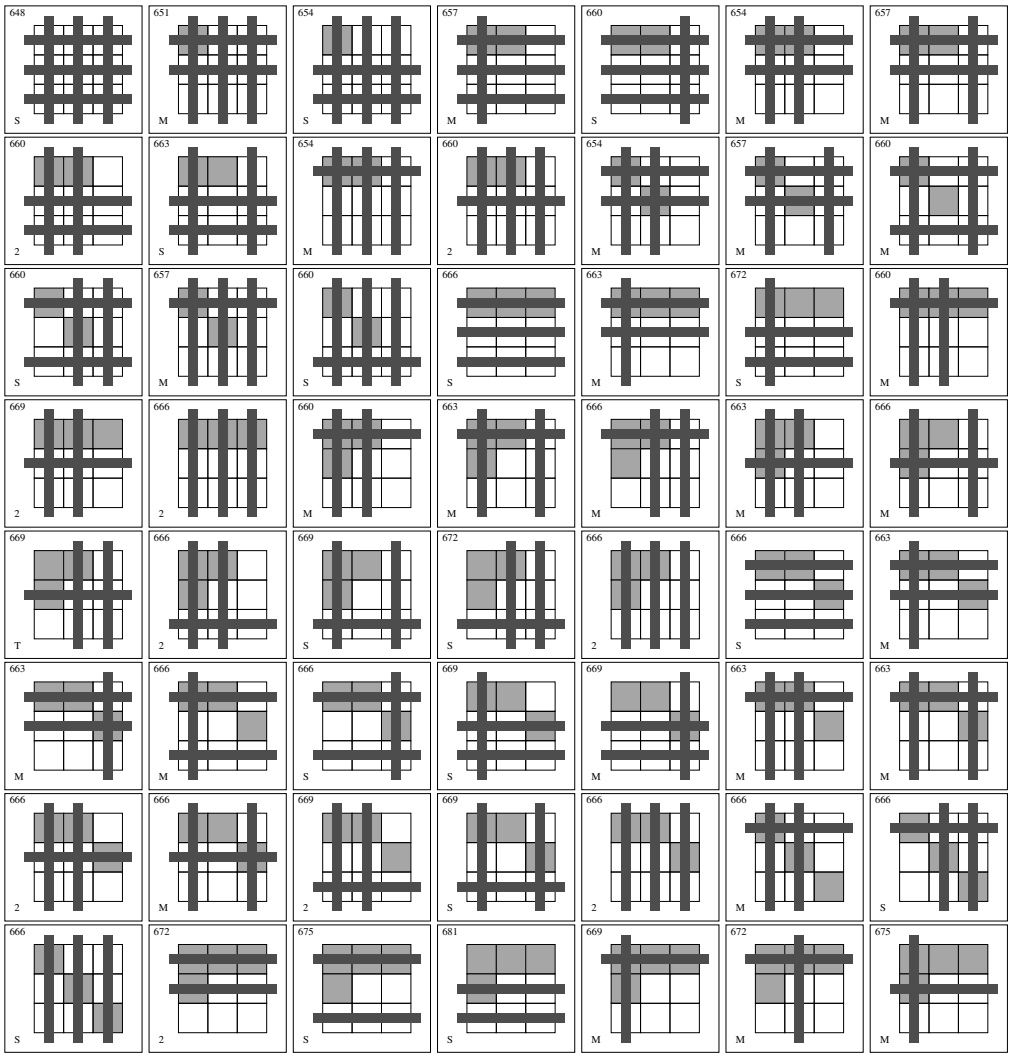
[1] We don't read rec.puzzles, but someone who knew about our results made us aware of the particular post.

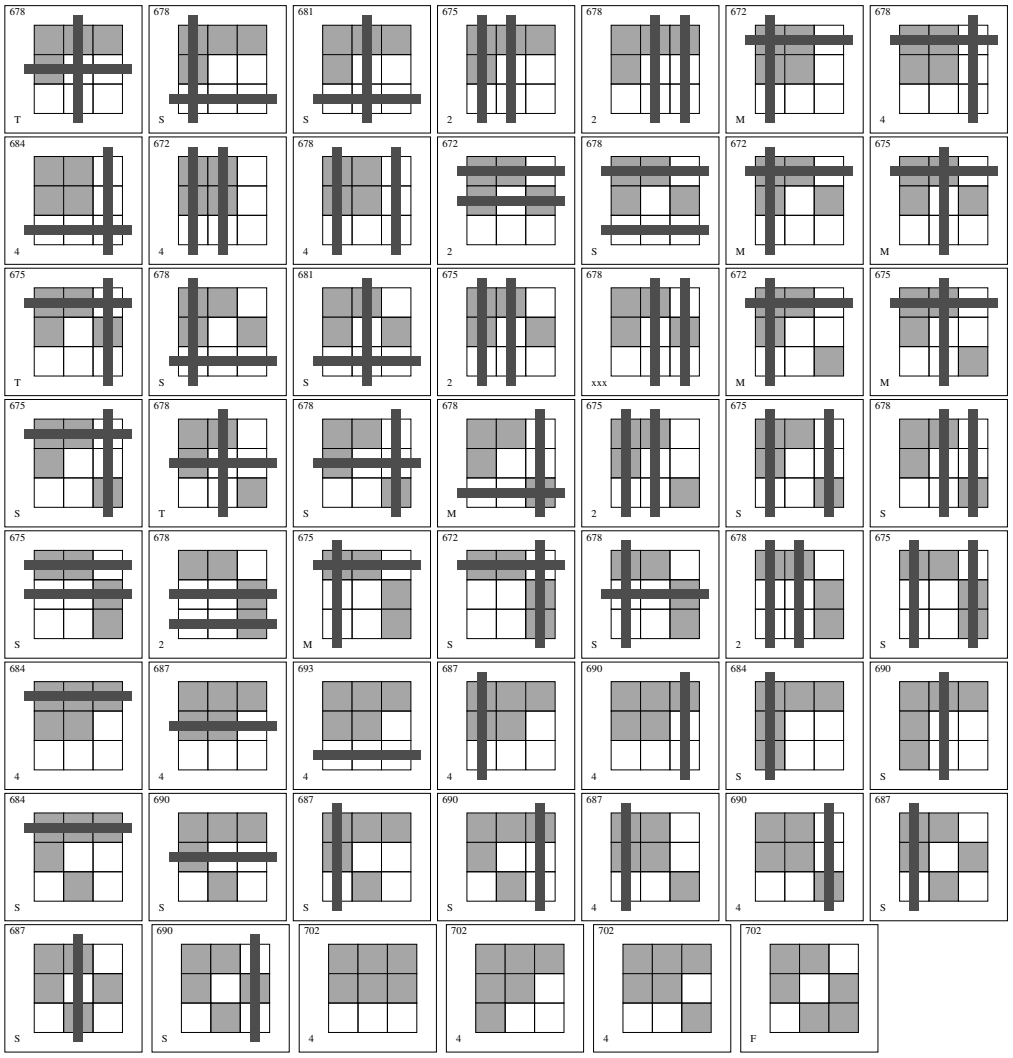
Appendix

All Elements of $Missing(6)$

Each element of $Missing(6)$ is annotated as follows:

- upper left corner: indicates the number of small rules that result from expanding the big rules displayed
- lower left corner: S means $Sudoku$; the other characters indicate in which configuration algorithm I got stuck; M corresponds to Lemma 4, 2 corresponds to Lemma 5, 4 corresponds to Lemma 6, T corresponds to Lemma 7, xxx corresponds to Lemma 9, and F corresponds to Lemma 8,




678 T
678 S
681 S
675 2
678 2
672 M
678 4
684 4
672 4
678 4
672 2
678 S
672 M
675 M
675 T
678 S
681 S
675 2
678 xxx
672 M
675 M
675 S
678 T
678 S
678 M
675 2
675 S
678 S
675 S
678 2
675 M
672 S
678 S
678 2
675 S
684 4
687 4
693 4
687 4
690 4
684 S
690 S
684 S
690 S
687 S
690 S
687 S
690 4
687 4
690 S
687 S
690 S
702 4
702 4
702 4
702 F

Proceedings of

CICLOPS 2008

8th International Colloquium on Implementation of Constraint and LOgic Programming Systems

Udine, 12th-13th December, 2008

Organizers:

Manuel Carro Liñares

UPM

Bart Demoen

KUL

Preface

As previous editions of CICLOPS, the 2008 version in Udine brings together researchers interested in the sequential and parallel implementation of logic and constraint programming languages and systems. CICLOPS promotes the free exchange of ideas and early dissemination of potentially premature and promising ideas. CICLOPS 2008 continues a tradition of successful workshops on Implementations of Logic Programming Systems, previously held with considerable success in Budapest (1993) and Ithaca (1994), the Compulog Net workshops on Parallelism and Implementation, and CICLOPS-es in Paphos (2001), Copenhagen (2002), (2003), St.-Malo (2004), Sitges (2005), Seattle (2006), and Porto (2007). With thirteen papers, the program is full, varied and interesting, and it shows both the need for this workshop and the potential for the future: you should feel very sorry if you cannot attend the workshop. The program committee did a great job in reviewing the papers and they have already given valuable feedback to the authors. Together with the interaction during the workshop, which is by tradition informal, lively and open, this CICLOPS again truly serves as a credible stepping stone to formal publication. We thank all authors, reviewers, and attendees for their efforts and interest.

Manuel Carro
Bart Demoen
Madrid and Leuven, November 2008

Program Committee

Slim Abdennadher (German University in Cairo)
Roberto Bagnara (University of Parma)
Amadeo Casas (Microsoft Co.)
Henning Christiansen (Roskilde University)
Gregory Duck (NICTA)
Hai-Feng Guo (University of Nebraska at Omaha)
Remy Haemmerle (Technical University of Madrid)
José Morales (Complutense University of Madrid)
Ulrich Neumerkel (Technische Universität Wien)
Phuong-Lan Nguyen (IMA, Université Catholique de l'Ouest)
Ricardo Rocha (University of Porto)
Tom Schrijvers (K.U. Leuven)
Paul Tarau (University of North Texas)
Jan Wielemaker (Universiteit van Amsterdam)
Manuel Carro (Technical University of Madrid — Co-chair)
Bart Demoen (K.U. Leuven — Co-chair)

Table of Contents

Implementing Thread Cancellation in Multithreaded Prolog Systems

Atef Suleiman and John Miller

School of Electrical Engineering and Computer Science
Washington State University (Tri-Cities)
West 201B, Richland, WA 99354-2125, USA
{asuleiman,jhmiller}@tricity.wsu.edu

Abstract. The Prolog primitive `thread_cancel/1`, which simply cancels a thread as recommended in ISO/IEC DTR 13211-5:2007, is conspicuously absent in well-maintained, widely used multithreaded Prolog systems. The ability to cancel a thread is useful for application development and is critical to Prolog embeddability. The difficulty of cancelling a thread is due to the instant mapping of Prolog multithreading primitives to the native-machine thread methods. This paper reports on an attempt to implement thread cancellation using self-blocking threads. A thread blocks at the same safe execution point where the state of the underlying virtual machine is defined. A blocked thread awaits a notification to resume or terminate. A resumed thread may be redirected to self-block by a blocking primitive. Experimental results based on a prototype implementation show that using self-blocking threads not only simplifies the implementation of thread cancellation but also improves the performance of message-passing primitives.

Key words: Prolog, concurrency, threads

1 Introduction

Explicit expressions of concurrency advance Prolog's standing as a practical programming language capable of exploiting modern multiprocessor computers. Prolog programs consist largely of static code, knowledge expressed as facts and rules, accessible to any number of execution threads running concurrently, in parallel or otherwise. Additionally, due to their declarative and high-level nature, Prolog programs retain and expose opportunities for parallel execution unparalleled in conventional programming languages. To facilitate expressions of concurrency, a thread model is proposed in ISO/IEC DTR 13211-5:2007 [1], variants of which are implemented in well-maintained, widely used Prolog systems, such as Ciao [2], SWI-Prolog [3], XSB [4], Yap [5] and others. The model includes a set of low-level primitives for thread creation, synchronization and communication. In addition to sharing the static database on a read-only basis, Prolog threads may modify and share the dynamic database in a mutually exclusive manner. Recent research in definition and implementation of high-level

parallelism primitives shows that a relevant speedup is obtainable by exploiting parallelism expressly at the source-language level [6, 7]. As this research activity illustrates, there are situations in which a need to cancel a thread arises after the thread has already started.

The need for cancelling a thread is illustrated by the high-level primitive `threaded/1` defined in [6]. Given a conjunction of well-formed goals, this primitive simulates an and-parallel operator executing the goals concurrently using a dedicated thread for each goal. The primitive succeeds if all goals succeed; otherwise, if a goal fails or raises an exception, it fails. Hence, once a thread at some point returns a failure result for a goal it has executed, the remaining threads should be cancelled since they serve no purpose at that point. A similar need for cancelling a thread arises when a threaded goal executes successfully as part of a deterministic disjunction executing concurrently. These and other practical examples, such as an asynchronously generated cancel condition initiated by a user request to exit a running program, show that thread cancellability is a desirable method of Prolog threads.

The option of cancelling a Prolog thread is provided by the primitive `thread_cancel/1`, described in [1] as follows:

> *`thread_cancel/1` cancels a thread. Any mutexes held by the thread shall be automatically released. The main Prolog thread cannot be cancelled. Other than this, any thread can cancel any other thread. It is expected that all the resources consumed by the thread be released upon thread cancellation.*

Prolog systems, however, implement `thread_cancel/1` in a variable way. XSB shares the responsibility for cancelling a thread with the programmer, whereas SWI-Prolog defers the implementation of `thread_cancel/1` altogether to the programmer, with the insight that the primitive is best implemented depending on the thread model of the problem at hand. In Ciao, the outcome of cancelling a thread is partly defined and depends wholly on the state of the target thread. The implementation of `thread_cancel/1` in these and other otherwise-compliant Prolog systems suggests that the above description for `thread_cancel/1` may be easier said than done.

The difficulty of cancelling a thread is due to blocking functions. Standard library functions, such as `read`, `accept`, `wait`, are subject to blocking as they are dependent on external events, e.g., the availability of input, establishment of a network connection, occurrence of a specified event. A thread attempting to cancel a blocked thread must be able to interact with the function inside which the target thread is blocked. The interaction may be initiated by either the cancelling thread, by means of signalling, or the cancelled thread, by means of polling. The latter method is adapted by POSIX threads [8], on which the majority of Prolog systems is based.

Also referred to as *Pthreads*, POSIX threads is a set of C functions for managing threads, mutual exclusion and condition variables.[1] A Prolog thread is directly mapped to a POSIX thread, running a Prolog engine within a multi-engined Prolog virtual machine. Cancelling a Prolog thread in the context of POSIX threads is a well-defined process insofar as the semantics of the latter is concerned.

POSIX specifies a subset of blocking functions as *cancellation points*. A blocking function designated as cancellation-point is expected to call an internal or external Pthreads function, e.g., `pthread_testcancel`, at sufficient intervals and be prepared for the possibility that the function may not return due to the thread being cancelled. Consequently, any function calling a cancellation-point function must be equally prepared to give up control without further notice. A function prepares for the possible loss of control by registering thread-specific cleanup functions to be executed in the event of thread cancellation. The process of cancelling a Prolog thread may, thus, proceed as follows. Given a proper accounting of consumed resources using `pthread_cleanup_push` and `pthread_cleanup_pop` within every lexical scope containing a cancellation point, a thread cancels another thread asynchronously by calling `pthread_cancel`, which flags the target thread as cancelled and returns immediately. If the target thread is active, the Prolog engine traps the thread at a safe execution point and destroys it by exiting the thread startup function. Otherwise, if the target thread is blocked or is to block, Pthreads takes over control at the next cancellation point and begins the actual cancellation process by calling the thread cleanup functions in a last-in-first-out order. Apart from excluding certain blocking functions, most notably `pthread_mutex_lock`, from the standard list of cancellation points, the process of cancelling a POSIX thread seems transparent enough to support an orderly cancellation of the adjoining Prolog thread.

However, as evident by the lack of support for `thread_cancel/1` in well-maintained Prolog systems, the direct mapping approach to thread cancellation faces implementation issues related, in part, to Prolog signals and garbage collection. As recommended in [1], a Prolog thread should be able to signal another thread to execute a goal as a soft interrupt at safe points, including, for example, the point at which a Prolog thread is suspended waiting for a message from a message queue. At that point, neither POSIX signals nor Pthreads cancellation points provide a mechanism for processing Prolog signals. While a Prolog thread can process a POSIX signal, thus receive a Prolog signal, it can not execute the signal, while the thread is blocked by a cancellation-point function. Similarly, memory and atom garbage collection algorithms require a high level of cooperation among Prolog threads incompatible with low-level mapping of Prolog threads to Pthreads. For example, when an active Prolog thread triggers atom garbage-collection, all other threads must suspend and produce their list of atoms. Here, again, a Prolog thread blocked by a cancellation-point function can not be guaranteed to heed a garbage-collection interrupt in any specifiable

[1] Condition variables are synchronization objects that allow threads to wait for certain events (conditions) to occur.

manner. In order to effect a working level of cooperation among Prolog threads, a high-level mapping of Prolog threads to Pthreads is required.

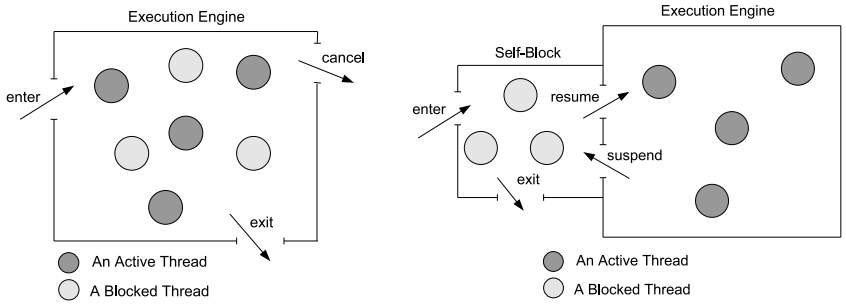
This paper reports on an attempt to implement thread cancellation using self-blocking threads. A self-blocking thread blocks at the same safe execution point where the state of the underlying Prolog engine is defined. A blocked thread awaits a notification to resume or terminate. A resumed thread may be reinstructed to self-block by a blocking primitive. Experimental results based on a proof-of-concept implementation show that using self-blocking threads is a viable approach for creating Prolog threads with the provision of facilitating their cancellation at any point during execution.

Section 2 presents the approach of self-blocking threads in the context of enabling synchronous cancellation of active and blocked threads. Section 3 includes implementation notes related to select blocking primitives. Section 4 presents the results of a performance comparison between self-blocking and directly-mapped threads. Section 5 briefly describes existing implementations of `thread_cancel/1`. Section 6 concludes with a summary of the cost-benefits of self-blocking threads.

2 An Execution Engine and Self-Block

Cancelling an active thread is a straightforward task. The thread is simply tagged as cancelled and the actual cancellation takes place upon the thread reaching a safe execution point. Cancelling a blocked thread, on the other hand, is a complex task requiring the consent and cooperation of the blocking function. Figure 1(a) shows a conceptual depiction of active and blocked threads inside the execution engine of a Prolog virtual machine. The difficulty of cancelling a thread lies with those threads that are blocked as a result of calling blocking functions. Figure 1(b) depicts the same threads in a new formation: active threads continue to be active; blocked threads are blocked on their own accord, using a self-blocking mechanism. The blocking functions are replaced by their cooperative counterparts, which are asynchronous, persistent and capable of instructing threads to block (suspend) or unblock (resume) as it may be warranted during execution. The task of cancelling a blocked thread is specifiable independent of the cancel method of the underlying native thread.

A blocking function directs a calling thread to self-block by returning a code indicating a *pending result*, based on which the thread self-blocks (suspends) waiting to be resumed or cancelled. A blocked thread is resumed by notifying the thread to continue execution from the point at which it was suspended, and is cancelled by notifying the thread to exit using the same control path used by an active thread exiting normally. A blocked thread may also be notified to perform atom-garbage collection or execute a goal as an interrupt. Multiple notifications are serialized using mutual exclusion. A notifying thread acquires exclusive control of the target thread prior to notification, with the caveat that control is granted only if the thread is suspended. A thread is suspended using



(a) Direct mapping (b) Self-blocking

Fig. 1. Graphical depiction of active and blocked threads.

the interrupt-vector mechanism commonly used in single-threaded systems to break into the top-level loop.

2.1 Implementing the Self-Block

The self-block is implemented using a standard synchronization composite of a *mutex*, *condition variable* and *counter*. Each thread is associated with a composite instance, which is initialized upon thread creation *in sync* with the creating thread. A blocking thread atomically unlocks the mutex and waits for the condition variable to be signalled by another thread. A signalling thread locks the mutex momentarily and signals the condition variable of the target thread. A blocked thread whose condition variable has been signalled re-locks its mutex, increments the counter and resumes execution. In addition to its standard role of preventing a race condition, the mutex is used to query the status of a thread. A thread queries the status of another thread by attempting to lock its mutex. If the attempt is successful, the thread is idle; otherwise, it is running. The counter is intended to be used in a test-yield loop to compel a signalled thread to assume ownership of its mutex.

The start-up algorithm for self-blocking threads is outlined in Figure 2. The algorithm takes a Prolog engine as input and proceeds as follows. First, it initializes a synchronization composite and swaps a reference to it with that of the temporary composite initialized by the creating thread for synchronizing with the current, newly created, thread (Lines 1-3). Second, it momentarily locks the mutex and signals the condition variable of the creating thread so that the latter may proceed (Line 4). Third, the algorithm iteratively suspends and resumes calling the execution engine for as often as the latter indicates a pending result (Lines 6-10). Lastly, the synchronization composite is destroyed and the native thread of control exits detaching from the adjoining Prolog engine.

```
    Input : A Prolog engine self
 1: initialize (composite = {mutex,condition,counter})
 2: lock (mutex)
 3: swap (self.composite, composite)
 4: lock (mutex), signal (condition), unlock (mutex)
 5: composite ← self.composite
 6: do
 7:     wait (condition, mutex)
 8:     counter ← counter + 1
 9:     execution_engine (self)
10: while self.result is a pending result
11: terminate (composite)
```

Fig. 2. Start-up algorithm of self-blocking threads.

2.2 Implementing `thread_cancel/1`

Cancelling a thread involves first suspending the thread, then destroying it. Since suspending and destroying a thread are well-defined tasks, they are implemented by the predicates `thread_suspend/1` and `thread_destroy/1`. With a negligible risk of raising an unintended exception, `thread_cancel/1` is defined as follows:

```
thread_cancel(Thread) :-
    thread_suspend(Thread),
    thread_destroy(Thread).
```

The algorithms for implementing *thread_suspend* and *thread_destroy* are listed in Figure 3 and 4, respectively. Both algorithms begin by decoding the target thread *thread* from the current actual arguments of the calling thread $self$. It is assumed that access to shared resources, such as the list of existing threads *list_of_threads*, is serialized using a locking mechanism.

The algorithm for *thread_suspend* starts by locking the list of existing threads and performing a series of tests, including whether the target thread is non-existent (Lines 3-6), referenced by other threads (Lines 7-10) or itself the calling thread (Lines 11-14), in which cases it throws an appropriate error-term or returns a pending result; otherwise, it increments the reference counter of the target thread and unlocks the list of existing threads (Lines 15-16). Next, the algorithm suspends the target thread by first setting its interrupt vector, then locking its mutex momentarily (Lines 17-22). Since it is possible that the target thread suspends for a reason other than having been interrupted, the thread interrupt vector is reset based on the return result. Lastly, the algorithm decrements the reference counter of the target thread and continues execution with the following instruction (Lines 23-26). Chances are that the next instruction to be executed corresponds to *thread_destroy*. In a like manner, *thread_destroy* algorithm destroys a target thread, provided the thread exists, is idle, different from the calling thread and unreferenced by any other threads.

```
1:  thread ← decode (self,1)
2:  lock (thread_resource)
3:  if thread ∉ list_of_threads then
4:      unlock (thread_resource)
5:      throw existence_error
6:  end
7:  if thread.reference > 0 then
8:      unlock (thread_resource)
9:      throw permission_error
10: end
11: if thread = self then
12:     unlock (thread_resource)
13:     thread.signal ← thread.signal ∨
        suspend_signal
14:     return signal_result
15: end
16: thread.reference ← thread.reference + 1
17: unlock (thread_resource)
18: thread.signal ← thread.signal ∨
    suspend_signal
19: lock (thread.mutex)
20: if thread.result ≠ signal_result then
21:     thread.signal ← thread.signal ∧
        ¬suspend_signal
22: end
23: unlock (thread.mutex)
24: lock (thread_resource)
25: thread.reference ← thread.reference − 1
26: unlock (thread_resource)
27: goto next_instruction
```

Fig. 3. *thread_suspend* algorithm

```
1:  thread ← decode (self,1)
2:  lock (thread_resource)
3:  if thread ∉ list_of_threads then
4:      unlock (thread_resource)
5:      throw existence_error
6:  end
7:  if thread = self ∨ thread.reference ≠ 0
    ∨ ¬ trylock(thread.mutex) then
8:      unlock (thread_resource)
9:      throw permission_error
10: end
11: destroy (thread)
12: unlock (thread_resource)
13: goto next_instruction
```

Fig. 4. *thread_destroy* algorithm

3 Implementing Thread-Blocking Predicates

Blocking predicates, be they built-in or user-defined, i.e., foreign, block by instructing the calling thread to self-block. For Prolog systems that provide a foreign-language interface, blocking foreign code communicates its blocking instructions by calling an appropriate interface function. The following are implementation notes related to select blocking predicates.

`get_code(+Stream, ?Code)` gets the character code of a single character from the (non-standard) input stream *Stream* and unifies it with the term *Code*. The predicate behaves like the standard built-in `get_code/2`, except that if the stream position of *Stream* is *end-of-stream* and *eof_action(suspend)* is a property of *Stream*, then the calling thread suspends, with the expectation that the foreign module that created *Stream* (e.g., an embedding application or shared library) will call an appropriate interface function to resume the calling thread when new characters become available.

`thread_get_message(+Queue, ?Term)` searches the message queue *Queue* for a term unifiable with the term *Term*. If a term is found, the term is unified with *Term* and deleted from *Queue*. Otherwise, if a term is not found, the calling thread is added to a waiting list associated with *Queue* and instructed to

block (suspend). The search, deletion and addition are performed in a mutually exclusive manner.

`thread_send_message(+Queue, @Term)` searches the waiting list of the message queue *Queue* for a thread whose receiving term is unifiable with the term *Term*. If a thread is found, then the thread is deleted from the waiting list, the receiving term is unified with *Term*, and the thread is instructed to unblock (resume). Otherwise, if a receiving thread is not found, *Term* is added to *Queue*. The search, deletion and addition are performed in a mutually exclusive manner.

`mutex_lock(+Mutex)` acquires the Prolog mutex *Mutex* blocking if necessary. If *Mutex* is already acquired by a thread other than the calling thread, then the calling thread is added to a waiting list associated with *Mutex* and instructed to suspend. If Mutex is previously acquired by the calling thread, then the recursion counter of *Mutex* is incremented. Otherwise, if *Mutex* is free, the calling thread acquires *Mutex*. The conditionals and corresponding actions are performed in a mutually exclusive manner.

`mutex_unlock(+Mutex)` releases the Prolog mutex *Mutex*. If *Mutex* is acquired by the calling thread and the recursion counter of *Mutex* is greater than zero, then the recursion counter is decremented. If *Mutex* is acquired by the calling thread and the recursion counter of *Mutex* is zero, then *Mutex* is first released, then acquired by the first thread, if any, on the waiting list of *Mutex* and the thread is instructed to resume. The conditionals and corresponding actions are performed in a mutually exclusive manner.

`sleep(+Interval)` suspends execution of the calling thread for the interval *Interval*. If *Interval* is an integer greater than zero, then the calling thread *Self* is suspended immediately and resumed after *Interval* is elapsed as follows. If an alarm is already set for a thread *Thread* and is expected to set off after interval $Interval_{thread}$ is elapsed, and $Interval > Interval_{thread}$, then the pair (*Self*, *Interval* - $Interval_{thread}$) is inserted into list *List*, containing ordered pairs of alarms to be set and threads to be resumed. Otherwise, if $Interval < Interval_{thread}$, then the alarm is cancelled, a new alarm is created to set off after *Interval* is elapsed, and the pair (*Thread*, $Interval_{thread}$ - *Interval*) is inserted into *List*. The insertion and cancellation are performed in a mutually exclusive manner. The alarm is a special thread that sleeps synchronously for and on behalf of the intervals and threads in *List*.

4 Performance Evaluation

A prototype Prolog implementation was developed to assess the performance of self-blocking threads on two popular operating systems: Linux and Windows. The prototype is a simple compiler and emulator comparable in performance to SWI-Prolog [3]. A select number of multithreading primitives were implemented using the self-blocking method, as described in Section 3, and the direct mapping method, as implemented in SWI-Prolog. The method in effect is determined at build time using conditional compilation. Three performance parameters were

measured: thread-creation time, message-passing time and synchronization time. The latter parameters were also measured using SWI-Prolog. All measurements were obtained by averaging ten runs per input per program. The computing environment is comprised of a single computer, equipped with Intel Core 2 Quad processor (2.5GHz), 3GB RAM (800MHz), dual-bootable with Linux Debian version 4.0 and Windows Vista (32-bit).

It should be noted that although both Linux and Windows use one-to-one mapping between user threads and kernel threads, Linux threads appear to be considerably more lightweight than Windows threads, possibly due in part to the Windowing system of Windows being an integral part of Windows kernel. The objective of this evaluation is to compare the performance of self-blocking threads to that of directly mapped threads. A thread performance comparison between Linux and Windows is outside the scope of this paper, let alone the interests of its authors.

4.1 Thread Creation

As described in Section 2.1, the procedure for creating a self-blocking thread requires that the calling thread blocks until the newly created thread initializes its self-blocking mechanism. The thread-creation time parameter is intended to quantify the overhead incurred by self-blocking threads during thread creation.

The execution time of thread creation of self-blocking and directly mapped threads was measured directly using two simple programs written in C. The first program measures the execution time of thread creation of directly mapped threads. It trivially creates a variable number of threads by calling the function `pthread_create`, tracking the wall time elapsed using the function `clock`. The second program measures the execution time of thread creation of self-blocking threads. It has the structure of the first program except that the call to `pthread_create` is embedded in a new function responsible for synchronizing the calling thread with the thread to be created. The new function initializes a temporary synchronization composite comprised of a mutex and condition variable, and calls `pthread_create`, passing a reference to the composite. It then calls `pthread_cond_wait` and blocks waiting for the composite to be signalled by the newly created thread. Meanwhile, the new thread first initializes its self-blocking mechanism, then signals the composite of the calling thread so that the latter may proceed.

As shown in Table 1, self-blocking threads are more expensive to create than directly mapped threads. The average execution time of thread creation of a self-blocking thread is about twice that of a directly mapped thread on both Linux and Windows. On Linux, the execution time of thread creation increases as the number of threads increases, approaching a measurable value when the number of threads equals or exceeds $1,000$. On Windows, the execution time of

thread creation is stable, around 200 μs per self-blocking thread and 100 μs per directly mapped thread, regardless of the number of threads.[2]

Table 1. Comparison of average execution time of thread creation (μs *per thread*).

# of threads	Linux		Windows	
	Direct mapping	Self-blocking	Direct mapping	Self-blocking
100	0	0	107	205
200	0	0	106	207
500	0	0	106	206
1000	2	6	107	205
2000	7	12	106	204
4000	10	15	106	204

4.2 Message Passing

The message-passing time parameter was first measured for the case of a single sender/receiver, where neither implementation method has an apparent advantage over the other. Here, passing a message involves sending the message and waking up the receiving thread. The time measurements were obtained using the program described in [9]. The program involves passing a message between N threads M times. The threads are linked in a ring structure. The message is an integer specifying the number of times the message is to be passed. Upon receiving the integer-message, a thread decrements the integer and passes it to the next thread. The message passing between threads continues until the integer becomes less than zero, at which point a thread simply exits. The program is listed in Figure 5. The message-passing time measurements were estimated for select numbers of threads performing message passing $1,000,000$ times. The results are presented in Table 2.[3]

Overall, the performance of self-blocking threads and directly mapped threads are comparable on both Linux and Windows. On a closer examination, however, the self-blocking approach is consistently, albeit slightly, faster than the direct mapping approach as implemented in both the prototype and SWI-Prolog. The

[2] On Windows, according to spawn-time measurement results obtained from Prototype and SWI-Prolog, the execution time of POSIX thread creation is the dominant component of the execution time of Prolog thread creation.

[3] For assurance and sheer curiosity, the time measurements of Java threads were also obtained and presented. On Linux, Java threads perform simple message passing twice as fast as Prolog threads using either approach. The Java speedup is likely due to Prolog's need to validate, in a mutually exclusive manner, the existence of a thread prior to accessing its message queue. The question as to why Java threads were unable to maintain a similar speedup factor on Windows is outstanding.

```
start(N, M) :-
    N1 is N - 1,
    thread_self(Thread),
    setup(N1, Thread, NextThread),
    thread_send_message(NextThread, M),
    catch(process(NextThread), _, true).

process(Thread) :-
    repeat,
        thread_get_message(M),
        M1 is M - 1,
        thread_send_message(Thread, M1),
        M1 < 0,
    !.

setup(0, Thread, Thread) :- !.
setup(N, Thread, NextThread) :-
    Goal = process(Thread),
    thread_create(Goal, NewThread, [detached(true)]),
    N1 is N - 1,
    setup(N1, NewThread, NextThread).
```

Fig. 5. Program for measuring simple message-passing time.

number of threads that can be created in SWI-Prolog is limited to less than 100 threads, thus the time measurements corresponding to numbers of threads equal or exceeding 100 are unobtainable. The simple message-passing time is relatively stable, around 4 μs on Linux, 12 μs on Windows, per message, for a range of 10 to 400 threads. However, this parameter is likely to increase as the number of threads increases due in part to cache exhaustion due, in turn, to the uncommon memory requirements of Prolog threads.

Table 2. Comparison of average execution time of threads performing simple message-passing (μs *per message*).

(a) Average execution time on Linux

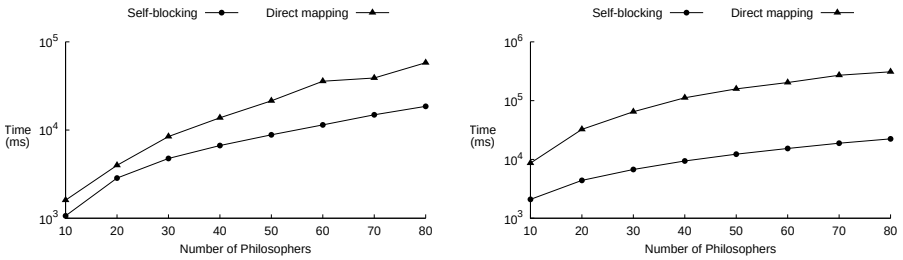
# of threads	self-blocking	direct mapping	SWI-Prolog 5.6.61	Java 1.6.0_06
10	5.86	5.90	5.99	3.03
20	4.36	5.26	4.73	2.94
40	4.26	4.78	4.58	2.91
80	4.02	4.53	4.94	3.21
100	4.15	4.38	–	3.24
200	4.12	4.38	–	3.35

(b) Average execution time on Windows

# of threads	self-blocking	direct mapping	SWI-Prolog 5.6.61	Java 1.6.0_06
10	11.75	13.21	14.54	11.75
20	11.95	12.20	13.71	11.75
40	11.95	12.73	13.29	11.22
80	11.95	12.48	13.38	11.26
100	12.04	12.83	–	11.39
200	12.78	13.51	–	11.39

The message-passing time parameter was, second, measured for the case of multiple senders/receivers, where self-blocking threads have a decisive advantage over directly-mapped threads. Here, message passing may involve a series of time-consuming operations, including adding (copying) a sender's message to a message queue, searching a list of waiting receivers for one whose skeletal message matches a newly added message, searching a message queue for a message matching a receiver's skeletal message, waking up potential receivers or just a matching receiver, and adding a new receiver to a list of waiting receivers.

The classic concurrency problem of the dining philosophers was used to illustrate the speed advantage of self-blocking threads in programs that require extensive message passing. The solution found in [10] was adapted to obtain wall time measurements for a variable number of philosophers. The measurements are depicted graphically in Figure 6.



(a) Linux (b) Windows

Fig. 6. The Dining Philosophers benchmark (10,000 eat-think cycle per philosopher.)

As expected, self-blocking threads outperform directly mapped threads, by a factor of 2 on Linux and by an order of magnitude on Windows. The source of the speedup is transparent. In the self-blocking approach, a new sender signals at most one potential receiver, whereas in the direct-mapping approach, the sender must signal all waiting receivers, even though only one of which might succeed in getting the sender's message while the other receivers will attempt in vain to unify their skeletal messages with the old messages of previous senders. In addition to performing needless unification, the majority of receivers effects needless task-switches performed by the operating system at the behest of unassuming senders.

4.3 Synchronization

The synchronization time parameter was measured using a simple program, which creates a variable number of threads, each of which updates a shared resource 10,000 times. Mutual exclusion is achieved using a global mutex and the synchronization primitives `mutex_lock/1` and `mutex_unlock/1`. The average

execution time per mutual exclusion was estimated by subtracting the wall time required to execute an equal number of updates sequentially. The results are presented in Table 3.

Table 3. Comparison of average execution time of threads updating a shared resource (*μs per mutual exclusion*).

(a) Average execution time on Linux

# of threads	self-blocking	direct mapping (compliant)	direct mapping (incompliant)	SWI-Prolog 5.6.61
10	7.53	7.97	0.56	0.86
20	7.90	8.01	0.75	0.95
40	7.47	8.52	0.91	1.02
80	7.53	8.70	0.96	1.08
100	6.88	8.78	0.99	–
200	7.89	8.93	1.01	–

(b) Average execution time on Windows

# of threads	self-blocking	direct mapping (compliant)	direct mapping (incompliant)	SWI-Prolog 5.6.61
10	11.06	16.10	1.53	11.31
20	10.90	17.04	1.49	11.91
40	10.46	17.37	1.51	12.52
80	10.89	17.39	1.49	12.49
100	10.57	17.52	1.49	–
200	10.75	18.18	1.48	–

The performance of self-blocking and directly mapped threads in programs that require extensive synchronization varies depending on the implementation of Prolog mutex. For implementations potentially compliant with [1], self-blocking threads compare favorably to directly mapped threads on Linux. On Windows, the former (self-blocking) threads outperform the latter threads by a factor as high as 1.7. Moreover, on Windows, the prototype's compliant implementation using self-blocking threads outperforms SWI-Prolog incompliant implementation using directly mapped threads. The criteria for compliance, for the purpose of this comparison, is that a Prolog mutex is indestructible while it is in use, e.g., one or more threads are blocked attempting to acquire the mutex. As shown in Table 3, lifting this requirement of indestructibility can result in a synchronization speed characteristic of low-level programming languages, however, to the negation of the premise of using self-blocking threads, which is to provide a safe and user-friendly Prolog multithreaded environment.

5 Related work

While Prolog systems agree on how to create threads, they differ widely on how to cancel them.

SWI-Prolog [3] and Yap [5] defer the implementation of `thread_cancel/1` to the programmer with the insight that thread cancellation is best implemented based on the thread model of the application at hand. In the boss/worker thread model, for example, `thread_cancel/1` may be implemented by communicating to the thread to be cancelled a specially coded message instructing the thread to exit or abort. In a computation-intensive application, for another example, cancelling a thread may be achieved by signalling the thread to execute a goal quoting a control primitive, such as `thread_exit(cancelled)`.

In XSB [4], thread cancelation is a joint responsibility of the system and the application. The latter initiates the process of canceling a thread by calling `thread_cancel/1`, giving the thread to be cancelled as an argument. For its part, XSB internally flags the given thread as canceled and waits for the thread execution to reach a call or execute port, at which point XSB throws a cancelation error ending its role in the thread cancelation process. The target thread, henceforth, is expected to catch the error, release any allocated resources and exit voluntarily.

Ciao [2] provides a primitive named `eng_kill/1`, which attempts to cancel the thread associated with a given goal identifier. The attempt may succeed, fail, block or render the system in an unstable state, depending on whether, irrespectively, the thread to be cancelled is trappable at a standard port, the goal identifier is valid, the thread is blocked by a system call, or other noted, however unspecified, situations.

Other Prolog systems, such as BinProlog and Qu-Prolog, provide other variations on the theme of thread cancellation. However, the primitives tasked with cancelling a thread are summarily documented. Attempts to learn of the internals of these primitives, through haphazard queries written with ill intents, showed that thread cancellation in these systems is problematic.

6 Conclusion

This paper presented an experimental implementation approach for creating Prolog threads with the provision of facilitating their destruction at any point during execution. The approach is based on self-blocking threads, a common implementation technique for managing thread interactions in multithreaded applications. The ability to cancel a thread safely and synchronously improves Prolog's standing as a useful programming language, capable of expressing variable solutions to complex concurrent problems for prototyping or production purposes. Additionally, it preserves the integrity of Prolog's traditional top-level loop program and improves Prolog's embeddability into multi-paradigm, multi-language applications.

Thread cancellability with self-blocking threads increases the complexity of system and extension development, as might be expected of features of high-level programming languages. Standard library functions, such as `seek`, `sleep`, `select`, may not be used directly to implement built-in and library predicates. Instead, these functions are reemployed within newly designed, more complex functions which are reentrant, persistent, asynchronous and able to communicate intermediate results. This added complexity may be viewed as a fair price, paid at the right layer in the right currency, C, in exchange for preserving Prolog's dictum of combining simplicity and power at the user level.

Although native in their own right, self-blocking threads exhibit the programmability of green threads,[4] as they are at most one standard port away from relinquishing processor control and one wake-up call from regaining it. As such, they are fit to yield the main benefits of both native and green multithreaded environments, namely parallelism and portability. Used in this capacity, the self-blocking approach constitutes a cost-efficient compromise between using native preemptive threads [11] and nonnative cooperative threads [12].

The performance of self-blocking threads compares favorably to that of directly mapped threads, despite that the time cost of creating a self-blocking thread is twice that of a directly mapped thread, due to the initial cost of the former's self-blocking mechanism. Self-blocking threads support a wide range of algorithms for implementing message passing, a primary means of thread communication and synchronization [1]. For programs that require extensive message passing, experimental results showed that execution times vary by up to an order of magnitude, depending on the operating system and the algorithm used for matching the messages of senders and receivers. Given that directly mapped threads can hardly do without a message queue and message passing, the run time advantage of self-blocking threads should offset the initial cost of their self-blocking mechanisms.

The utility of self-blocking threads extends beyond simplifying thread cancellation to enabling the implementation of high-level features, such as the separation of thread creation and execution, the implementation of suspend and resume primitives, backtracking, multiple executions and execution modes. The ability to separate thread creation from execution, proposed in passing in [13], facilitates the implementation of a high level API, which subsumes the one recommended in [1], which in turn facilitates the implementation of yet higher-level parallel operators analogous to those introduced in [6] and [7]. Experiments are being conducted to evaluate the merits of new multithreading primitives in terms of simplicity and expressiveness, as well as performance.

Acknowledgments. The authors thank Jan Wielemaker and Richard O'Keefe for their insightful, differing views. This research was supported by the Office of Science (BER), U.S. Department of Energy, Grant No. DE-FG02-05ER64105.

[4] Green threads are threads that are scheduled by a virtual machine instead of natively by the underlying operating system.

References

1. Wielemaker, J., Moura, P., Nunes, P., Robinson, P., Marques, R., Swift, T.: ISO/IEC DTR 13211-5:2007 Prolog Multi-threading Predicates. (2007)
2. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., Lopez-Garcıa, P., Puebla, G.: The Ciao Prolog System. Reference Manual (v1. 8). The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of Madrid (UPM), May (2002)
3. Wielemaker, J.: SWI Prolog 5.6 Reference Manual. Department of Social Science Informatics, University of Amsterdam, Amsterdam, Marz (2006)
4. Sagonas, K., Swift, T., Warren, D., Freire, J., Rao, P.: XSB Prolog. The XSB System Version 3.1 Volume 1: Programmers Manual (2007)
5. Santos-Costa, V., Damas, L., Reis, R., Azevedo, R.: The Yap Prolog Users Manual. Universidade do Porto and COPPE Sistemas (2006)
6. Moura, P., Crocker, P., Nunes, P.: High-Level Multi-threading Programming in Logtalk. In Hudak, P., Warren, D.S., eds.: PADL. Volume 4902 of Lecture Notes in Computer Science., Springer (2008) 265–281
7. Casas, A., Carro, M., Hermenegildo, M.: Towards a high-level implementation of flexible parallelism primitives for symbolic languages. Proceedings of the 2007 international workshop on Parallel symbolic computation (2007) 93–94
8. The IEEE and The Open Group: 1003.1 Standard for Information Technology-Portable Operating System Interface (Posix) System Interfaces, Issue 6. IEEE Std 1003.1-2001. System Interfaces, Issue 6 (2001)
9. Halen, J., Karlsson, R., Nilsson, M.: Performance measurements of threads in Java and processes in Erlang. Webpage, Last visit January (2006)
10. de Bosschere, K., Tarau, P.: Blackboard-based extensions in Prolog. Software—Practice & Experience **26** (1996) 49–69
11. Wielemaker, J.: Native preemptive threads in SWI-prolog. In Palamidessi, C., ed.: ICLP. Volume 2916 of Lecture Notes in Computer Science., Springer (2003) 331–345
12. Eskilson, J., Carlsson, M., Palamidessi, C., Glaser, H., Meinke, K.: SICStus MT—A Multithreaded Execution Environment for SICStus Prolog. Programming Languages: Implementations, Logics, and Programs (**1490**) 36–53
13. Carro, M., Hermenegildo, M.: Concurrency in Prolog Using Threads and a Shared Database. Logic Programming: Proceedings of the 1999 International Conference on Logic Programming (1999)

Interactors: Logic Engine Interoperation with Pure Prolog Semantics

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
E-mail: tarau@cs.unt.edu

Abstract. We introduce a new programming language construct, *Interactors*, supporting the agent-oriented view that programming is a dialog between simple, self-contained, autonomous building blocks.
We define *Interactors* as an abstraction of answer generation and refinement in *Logic Engines* resulting in expressive language extension and metaprogramming patterns.
As a first step toward a declarative semantics, we sketch a pure Prolog specification showing that Interactors can be expressed at source level, in a relatively simple and natural way.
Interactors extend language constructs like Ruby, Python and `C#`'s multiple coroutining block returns through *yield* statements and they can emulate the action of fold operations and monadic constructs in functional languages.
Using the Interactor API, we describe at source level, language extensions like dynamic databases and algorithms involving combinatorial generation and infinite answer streams.
Keywords*: Prolog language extensions, logic engines, semantics of metaprogramming constructs, generalized iterators, agent oriented programming language constructs*

1 Introduction

Interruptible Iterators are a new Java extension described in [1]. The underlying construct is the `yield` statement providing multiple returns and resumption of iterative blocks, i.e. for instance, a `yield` statement in the body of a `for` loop will return a result for each value of the loop's index.

The `yield` statement has been integrated in newer Object Oriented languages like `Ruby` [2, 3] `C#` [4] and `Python` [5] but it goes back to the *Coroutine Iterators* introduced in older languages like CLU [6] and ICON [7].

A natural generalization of Iterators, is the more radical idea of allowing clients to communicate to/from inside blocks of arbitrary recursive computations. The challenge is to achieve this without the fairly complex interrupt based communication protocol between the iterator and its client described in [1]. This suggests some form of structured two-way communication between a client and the usually autonomous service the client requires from a given language construct, often encapsulating an independent component.

Agent programming constructs have influenced design patterns at "macro level", ranging from interactive Web services to mixed initiative computer human interaction. *Performatives* in Agent communication languages [8] have made these constructs reflect explicitly the intentionality, as well as the negotiation process involved in agent interactions. At a more theoretical level, it has been argued that *interactivity*, seen as fundamental computational paradigm, can actually expand computational expressiveness and provide new models of computation [9].

In a logic programming context, the Jinni agent programming language [10] and the BinProlog system [11] have been centered around logic engine constructs providing an API that supported reentrant instances of the language processor. This has naturally led to a view of logic engines as instances of a generalized family of iterators called *Fluents* [12], that have allowed the separation of the first-order language interpreters from the multi-threading mechanism, while providing a very concise source-level reconstruction of Prolog's built-ins.

Building upon the *Fluents* API described in [12], this paper will focus on bringing interaction-centered, agent oriented constructs from software design frameworks and design patterns to programming language level.

The resulting language constructs, that we shall call *Interactors*, will express control, metaprogramming and interoperation with stateful objects and external services. They complement pure Horn Clause Prolog with a significant boost in expressiveness, to the point where they allow emulating at source level virtually all Prolog builtins, including dynamic database operations.

2 First Class Logic Engines

Our *Interactor API* is a natural extension of the *Logic Engine API* introduced in [12]. An *Engine* is simply a language processor reflected through an API that allows its computations to be controlled interactively from another *Engine* very much the same way a programmer controls Prolog's interactive toplevel loop: launch a new goal, ask for a new answer, interpret it, react to it.

A *Logic Engine* is an *Engine* running a Horn Clause Interpreter with LD-resolution [13] on a given clause database, together with a set of built-in operations. The command

```
new_engine(AnswerPattern,Goal,Interactor)
```

creates a new Horn Clause solver, uniquely identified by `Interactor`, which shares code with the currently running program and is initialized with `Goal` as a starting point. `AnswerPattern` is a term, usually a list of variables occurring in `Goal`, of which answers returned by the engine will be instances. Note however that `new_engine/3` acts like a typical constructor, no computations are performed at this point, except for allocating data areas. In our actual implementation, with all data areas dynamic, engines are lightweight and engine creation is extremely fast.

The `get/2` operation is used to retrieve successive answers generated by an Interactor, on demand. It is also responsible for actually triggering computations in the engine.

```
get(Interactor,AnswerInstance)
```

It tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`. If an answer is found, it is returned as `the(AnswerInstance)`, otherwise the atom `no` is returned. As in the case of the `Maybe` Monad in Haskell, returning distinct functors in the case of success and failure, allows further case analysis in a pure Horn Clause style, without needing Prolog's CUT or if-then-else operation.

Note that bindings are not propagated to the original `Goal` or `AnswerPattern` when `get/2` retrieves an answer, i.e. `AnswerInstance` is obtained by first standardizing apart (renaming) the variables in `Goal` and `AnswerPattern`, and then backtracking over its alternative answers in a separate Prolog interpreter. Therefore, backtracking in the caller interpreter does not interfere with the new Interactor's iteration over answers. Backtracking over the Interactor's creation point, as such, makes it unreachable and therefore subject to garbage collection.

An Interactor is stopped with the `stop/1` operation that might or might not reclaim resources held by the engine. In our actual implementation we are using a fully automated memory management mechanism where unreachable engines are automatically garbage collected.
So far, these operations provide a minimal *Coroutine Iterator API*, powerful enough to switch tasks cooperatively between an engine and its client and emulate key Prolog built-ins like `if-then-else` and `findall` [12], as well as higher order operations like *fold* and *best_of*.

3 From Fluents to Interactors

We will now describe the extension of the *Fluents* API of [12] that provides a minimal bidirectional communication API between interactors and their clients.

The following operations provide a "mixed-initiative" interaction mechanism, allowing more general data exchanges between an engine and its client.

3.1 A yield/return operation

First, like the `yield return` construct of `C#` and the `yield operation` of Ruby and Python, our `return/1` operation

```
return(Term)
```

will save the state of the engine and transfer *control* and a *result* `Term` to its client. The client will receive a copy of `Term` simply by using its `get/2` operation. Similarly to Ruby's `yield`, our `return` operation suspends and returns data from arbitrary computations (possibly involving recursion) rather than from specific language constructs like a `while` or `for` loop.

Note that an Interactor returns control to its client either by calling `return/1` or when a computed answer becomes available. By using a sequence of `return/get` operations, an engine can provide a stream of *intermediate/final results* to its client, without having to backtrack. This mechanism is powerful enough to implement a complete exception handling mechanism (see [12]) simply by defining

```
throw(E):-return(exception(E)).
```

When combined with a `catch(Goal,Exception,OnException)`, on the client side, the client can decide, upon reading the exception with `get/2`, if it wants to handle it or to throw it to the next level.

3.2 Interactors and Coroutining

The operations described so far allow an engine to return answers from any point in its computation sequence. The next step is to enable an engine's client to *inject* new goals (executable data) to an arbitrary inner context of an engine. Two new primitives are needed:

```
to_engine(Engine,Data)
```

used to send a client's data to an Engine, and

```
from_engine(Data)
```

used by the engine to receive a client's Data.

A typical use case for the *Interactor API* looks as follows:

1. the *client* creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*, parameterized as follows:
   (a) the *client* passes some data and a new goal to the *engine* and issues a `get` operation that passes control to it
   (b) the *engine* starts a computation from its initial goal or the point where it has been suspended and runs (a copy of) the new goal received from its *client*
   (c) the *engine* returns (a copy of) the answer, then suspends and returns control to its *client*
3. the *client* interprets the answer and proceeds with its next computation step
4. the process is fully reentrant and the *client* may repeat it from an arbitrary point in its computation

Using a metacall mechanism like `call/1` (which can also be emulated in terms of engine operations [12]) or directly through a source level transformation [14], one can implement a close equivalent of Ruby's `yield` statement as follows:

```
ask_engine(Engine,(Answer:-Goal), Result):-
  to_engine(Engine,(Answer:-Goal)),
  get(Engine,Result).

engine_yield(Answer):-
  from_engine((Answer:-Goal)),
  call(Goal),
  return(Answer).
```

where `ask_engine` sends a goal (possibly built at runtime) to an engine, which in turn, executes it and returns a result with an `engine_yield` operation.

As the following example shows, this allows the client to use from outside the (infinite) recursive loop of an engine as a form of *updatable persistent state*.

```
sum_loop(S1):-engine_yield(S1⇒S2),sum_loop(S2).

inc_test(R1,R2):-
   new_engine(_,sum_loop(0),E),
   ask_engine(E,(S1⇒S2:-S2 is S1+2),R1),
   ask_engine(E,(S1⇒S2:-S2 is S1+5),R2).

?- inc_test(R1,R2).
R1=the(0 ⇒ 2),
R2=the(2 ⇒ 7)
```

Note also that after parameters (the increments 2 and 5) are passed to the engine, results dependent on its state (the sums so far 2 and 7) are received back. Moreover, note that an arbitrary goal is injected in the local context of the engine where it is executed, with access to the engine's *state variables* `S1` and `S2`. As engines have separate garbage collectors (or in simple cases as a result of tail recursion), their infinite loops run in constant space, provided that no unbounded size objects are created.

4 A (mostly) Pure Prolog Specification

At a first look, Interactors deviate from the usual Horn Clause semantics of pure Prolog programs. A legitimate question arises: are they not just another procedural extension, say, like assert/retract, setarg, global variables etc.?

We will show here that the semantic gap between pure Prolog and its extension with Interactors is much narrower than one would expect. The techniques that we will describe can be seen as an executable specification of Interactors within the well understood semantics of logic programs (SLDNF resolution).

Toward this end, we will sketch an emulation, in pure Prolog, of the key constructs involved in defining Interactors.

There are four distinct concepts to be emulated:

1. we need to eliminate backtracking to be able to access multiple answers at a time
2. we need to emulate `copy_term` as different search branches and multiple uses of a given clause require fresh instances of terms, with variables standardized apart
3. we need to emulate suspending and resuming an engine
4. engines should be able to receive and return Prolog terms

We will focus here on the first two, that are arguably less obvious, by providing actual implementations. After that, we will briefly discuss the feasibility of the last two.

4.1 Metainterpreting Backtracking

First, let's define a clause representation, that can be obtained easily with a source-to-source translator. Clauses in the database are represented with difference-list terms, structurally isomorphic to the binarization transformation described in [14]. The code of a classic Prolog naive reverse + permutation generator program becomes:

```
:-op(1150,xfx,<=).

clauses([
        [app([],A,A)|B]<=B,
        [app([C|D],E,[C|F])|G]<=[app(D,E,F)|G],

        [nrev([],[])|H]<=H,
        [nrev([I|J],K)|L]<=[nrev(J,M),app(M,[I],K)|L],

        [perm([],[])|N]<=N,
        [perm([O|P],Q)|R]<=[perm(P,S),ins(O,S,Q)|R],

        [ins(T,U,[T|U])|V]<=V,
        [ins(W,[X|Y],[X|Z])|XO]<=[ins(W,Y,Z)|XO]
]).
```

Note that we can assume that variables are local to each clause and therefore they have been standardized apart accordingly[1].

First, let's define the basic inference step (equivalent to an LD-resolution step, [13]) as a simple "arrow composition" operation:

```
compose(F1,F2,A<=C):-copy_term(F1,A<=B),copy_term(F2,B<=C).
```

We can now add a new "arrow" to a list of existing arrows, provided that the composition succeeds:

```
match_one(F1,F2,Fs,[NewF|Fs]):-compose(F1,F2,F3),!,NewF=F3.
match_one(_,_,Fs,Fs).
```

and lift this to have an arrow (seen as representing the current goal), select from a *list* of clauses the ones that match:

```
match_all([],_,Fs,Fs).
match_all([Clause|Cs],Arrow,Fs1,Fs3):-
  match_one(Arrow,Clause,Fs1,Fs2),
  match_all(Cs,Arrow,Fs2,Fs3).
```

We can add a stopping condition to mark the success of an LD-derivation as matching an arrow of the form `Answer<=[]`

```
derive_one(Answer<=[],_,Fs,Fs,As,[Answer|As]).
derive_one(Answer<=[G|Gs],Cs,Fs,NewFs,As,As):-
  match_all(Cs,Answer<=[G|Gs],Fs,NewFs).
```

[1] Allowing shared variables would bring a different, but nevertheless interesting semantics, with "inter-clausal variables" seen as write-once global variables.

With these building blocks in place, an LD-derivation of *all answer instances* of a query can be defined as:

```
all_instances(AnswerPattern,Goal,Clauses,Answers):-
  Gs=[AnswerPattern⇐[Goal]],
  derive_all(Gs,Clauses,[],Answers).
```

where `derive_all` lifts the derivation process to progressively solve all existing and newly generated goals:

```
derive_all([],_,As,As).
derive_all([Arrow|Fs],Cs,OldAs,NewAs):-
  derive_one(Arrow,Cs,Fs,NewFs,OldAs,As),
  derive_all(NewFs,Cs,As,NewAs).
```

Finally, we can integrate the clause database:

```
all_answers(X,G,R):-clauses(Cs),all_instances(X,G,Cs,R).
```

and try out a few goals:

```
?- all_answers(Xs+Ys,app(Xs,Ys,[1,2,3]),Rs).
Rs = [[]+[1, 2, 3], [1]+[2, 3], [1, 2]+[3], [1, 2, 3]+[]]

?- all_answers(P,perm([1,2,3],P),Ps).
Ps = [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

Note, that for non-ground queries, answers computed this way keep variable equalities as expected:

```
?- List=[A,B,B,A],all_answers(R,nrev(List,R),Rs).
List = [A, B, B, A],
Rs = [[_A, _B, _B, _A]]
```

Note that, except for relying on `copy_term` and a cut that can be replaced with a negation as failure, the metainterpreter is entirely written in pure Prolog.

4.2 Emulating `copy_term`

We can emulate the effect of `copy_term` in the previously described metainterpreter by observing that a logical variable can be "split" into two new ones and consequently a Prolog term can be recursively deconstructed and rebuilt as two fresh terms, identical to it up to uniform variable renamings.

```
fork_term('$v'(T1,T2), R1,R2):-R1=T1,R2=T2.
fork_term(T, T1,T2):-
  nonvar(T),functor(T,F,N),(F/N) \== ('$v'/2),
  functor(T1,F,N),functor(T2,F,N),
  fork_args(N,T,T1,T2).

fork_args(0,_,_,_).
fork_args(I,T,T1,T2):-I>0,
  I1 is I-1,arg(I,T,X),
```

```
  fork_term(X,A,B),
  arg(I,T1,A),arg(I,T2,B),
  fork_args(I1,T,T1,T2).
```

One can see that this produces indeed two fresh copies of the original term:

```
?- fork_term(f(A,B,g(B,A)),T1,T2).
A = '$v'(_A1, _A2),
B = '$v'(_B1, _B2),
T1 = f(_A1, _B1, g(_B1, _A1)),
T2 = f(_A2, _B2, g(_B2, _A2)).
```

Note that `functor` and `arg` can be seen as generic abbreviations for predicates describing the building/decomposition operations for each function symbol occurring in the program and `$v/2` can be assumed to be any function symbol not occurring in the program. Along the lines of [15] one can see that this functionality can be also expressed through a simple program transformation provided that `nonvar/1` can be expressed using negation as failure as

```
  nonvar(X):- not(X=0),not(X=1).
```

We will obtain a slightly different definition of composition, that would require replacing both the clause and the resolvent with one of the copies while using the other pair of copies for the arrow compositions.

```
compose(F1,F2, A<=C, NewF1,NewF2):-
   fork_term(F1,A<=B,NewF1),
   fork_term(F2,B<=C,NewF2).
```

One can now see that after propagating the extra arguments through the clauses of the metainterpreter described in subsection 4.1, together with the source level transformations we just mentioned, a metainterpreter that does not require `copy_term` can be derived.

4.3 Implementing suspend/resume and term/exchanges

The metainterpreter described in subsection 4.1 can be easily modified to return the current goal list when observing a `return(X)` instruction and then be resumed at will, by adding a clause similar to the one handling the case `Answer<=[]`. At this point, data exchange operations and `to_engine` and `from_engine` can be implemented through an extra argument added to the metainterpreter.

5 Interactors and Higher Order Constructs

As a first glimpse at the expressiveness of the Interactor API, we will implement, in the tradition of higher order functional programming, a *fold* operation [16] connecting results produced by independent branches of a backtracking Prolog engine:

```
efoldl(Engine,F,R1,R2):-
  get(Engine,X),
  efoldl_cont(X,Engine,F,R1,R2).

efoldl_cont(no,_Engine,_F,R,R).
efoldl_cont(the(X),Engine,F,R1,R2):-
  call(F,R1,X,R),
  efoldl(Engine,F,R,R2).
```

Classic functional programming idioms like *reverse as fold* are then implemented simply as:

```
reverse(Xs,Ys):-
  new_engine(X,member(X,Xs),E),
  efoldl(E,reverse_cons,[],Ys).

reverse_cons(Y,X,[X|Y]).
```

Note also the automatic *deforestation* effect [17] of this programming style - no intermediate list structures need to be built, if one wants to aggregate the values retrieved from an arbitrary generator engine with an operation like sum or product.

6 Emulating Dynamic Databases with Interactors

The gain in expressiveness coming directly from the view of logic engines as answer generators is significant. We refer to [12] for source level implementations of virtually all essential Prolog built-ins (exceptions included). The notable exception is Prolog's dynamic database, requiring the bidirectional communication provided by interactors.

The key idea for implementing dynamic database operations with Interactors is to use a logic engine's state in an infinite recursive loop, similar to the coinductive programming style advocated in [18], to emulate state changes in its client engine.

First, a simple difference-list based infinite server loop is built:

```
queue_server:-queue_server(Xs,Xs).

queue_server(Hs1,Ts1):-
  from_engine(Q),
    server_task(Q,Hs1,Ts1,Hs2,Ts2,A),
  return(A),
  queue_server(Hs2,Ts2).
```

Next we provide the queue operations, needed to maintain the state of the database.

```
server_task(add_element(X),Xs,[X|Ys],Xs,Ys,yes).
server_task(push_element(X),Xs,Ys,[X|Xs],Ys,yes).
server_task(queue,Xs,Ys,Xs,Ys,Xs-Ys).
```

```
server_task(delete_element(X),Xs,Ys,NewXs,Ys,YesNo):-
  server_task_delete(X,Xs,NewXs,YesNo).
```

Then we implement the auxiliary predicates supporting various queue operations:

```
server_task_remove(Xs,NewXs,YesNo):-
  nonvar(Xs),Xs=[X|NewXs],!,
  YesNo=yes(X).
server_task_remove(Xs,Xs,no).

server_task_delete(X,Xs,NewXs,YesNo):-
  select_nonvar(X,Xs,NewXs),!,
  YesNo=yes(X).
server_task_delete(_,Xs,Xs,no).

select_nonvar(X,XXs,Xs):-nonvar(XXs),XXs=[X|Xs].
select_nonvar(X,YXs,[Y|Ys]):-nonvar(YXs),YXs=[Y|Xs],
  select_nonvar(X,Xs,Ys).
```

Finally, we put it all together, as a dynamic database API:

We can create a new engine server providing Prolog database operations:

```
new_edb(Engine):-new_engine(done,queue_server,Engine).
```

We can add new clauses to the database

```
edb_assertz(Engine,Clause):-
  ask_engine(Engine,add_element(Clause),the(yes)).

edb_asserta(Engine,Clause):-
  ask_engine(Engine,push_element(Clause),the(yes)).
```

and we can return fresh instances of asserted clauses

```
edb_clause(Engine,Head,Body):-
  ask_engine(Engine,queue,the(Xs-[])),
  member((Head:-Body),Xs).
```

or remove them from the the database

```
edb_retract1(Engine,Head):-Clause=(Head:-_Body),
  ask_engine(Engine,
    delete_element(Clause),the(yes(Clause))).
```

Finally, the database can be discarded by discarding the engine that hosts it:

```
edb_delete(Engine):-stop(Engine).
```

The following example shows how the database generates the equivalent of `clause/2`, ready to be passed to a Prolog metainterpreter.

```
test_clause(Head,Body):-
  new_edb(Db),
    edb_assertz(Db,(a(2):-true)),
```

```
    edb_asserta(Db,(a(1):-true)),
    edb_assertz(Db,(b(X):-a(X))),
  edb_clause(Db,Head,Body).
```

As a side note, combining this emulation with the metainterpreter described in section 4, provides an executable specification of Prolog's dynamic database operations in pure Prolog, worth investigating in depth, as future work.

Externally implemented dynamic databases can also be made visible as Interactors and reflection of the interpreter's own handling of the Prolog database becomes possible. As an additional benefit, multiple databases can be provided. This simplifies adding module, object or agent layers at source level. By combining database and communication Interactors, software abstractions like mobile code and autonomous agents can be built as shown in [19]. Encapsulating external stateful objects like file systems or external database or Web service interfaces as Interactors can provide a uniform interfacing mechanism and reduce programmer learning curves in practical applications of Prolog.

Moreover, Prolog operations traditionally captive to predefined list based implementations (like DCGs) can be made generic and mapped to work directly on Interactors encapsulating file, URL and socket Readers.

7 Refining control: a backtracking if-then-else

Modern Prolog implementations (SWI, SICStus, BinProlog, Jinni) also provide a variant of `if-then-else` that either backtracks over multiple answers of its `then` branch or switches to the `else` branch if no answers in the `then` branch are found. With the same API, we can implement it at source level as follows:

```
if_any(Cond,Then,Else):-
  new_engine(Cond,Cond,Engine),
  get(Engine,Answer),
  select_then_or_else(Answer,Engine,Cond,Then,Else).

select_then_or_else(no,_,_,_,Else):-Else.
select_then_or_else(the(BoundCond),Engine,Cond,Then,_):-
  backtrack_over_then(BoundCond,Engine,Cond,Then).

backtrack_over_then(Cond,_,Cond,Then):-Then.
backtrack_over_then(_,Engine,Cond,Then):-
  get(Engine,the(NewBoundCond)),
  backtrack_over_then(NewBoundCond,Engine,Cond,Then).
```

8 Simplifying Algorithms: Interactors and Combinatorial Generation

Various combinatorial generation algorithms have elegant backtracking implementations. However, it is notoriously difficult (or inelegant, through the use of impure side effects) to compare answers generated by different OR-branches of Prolog's search tree.

8.1 Comparing Alternative Answers

Such optimization problems can easily be expressed as follows:

- running the generator in a separate logic engine
- collecting and comparing the answers in a client controlling the engine

The second step can actually be automated, provided that the comparison criterion is given as a predicate

```
compare_answers(First,Second,Best)
```

to be applied to the engine with an `efold` operation

```
best_of(Answer,Comparator,Generator):-
  new_engine(Answer,Generator,E),
  efoldl(E,
    compare_answers(Comparator),no,
  Best),
  Answer=Best.

compare_answers(Comparator,A1,A2,Best):-
  if((A1\==no,call(Comparator,A1,A2)),
    Best=A1,
    Best=A2
  ).

?-best_of(X,>,member(X,[2,1,4,3])).
X=4
```

8.2 Counting Answers without Accumulating

Problems as simple as counting the number of solutions of a combinatorial generation problem can become tricky in Prolog (unless one uses impure side effects) as one might run out of space by having to generate all solutions as a list, just to be able to count them. The following example shows how this can be achieved using an `efold` operation on an integer partition generator:

```
integer_partition_of(N,Ps):-
  positive_ints(N,Is),
  split_to_sum(N,Is,Ps).

split_to_sum(0,_,[]).
split_to_sum(N,[K|Ks],R):-N>0,sum_choice(N,K,Ks,R).

sum_choice(N,K,Ks,[K|R]):-
  NK is N-K,split_to_sum(NK,[K|Ks],R).
sum_choice(N,_,Ks,R):-split_to_sum(N,Ks,R).

positive_ints(1,[1]).
positive_ints(N,[N|Ns]):-N>1,N1 is N-1,
```

```
  positive_ints(N1,Ns).

% counts partitions by running
% the generator on an engine that returns
% 1 for each answer that is found
count_partitions(N,R):-
  new_engine(1,
    integer_partition_of(N,_),Engine),
  efoldl(Engine,+,0,R).
```

8.3 Encapsulating Infinite Computations Streams

An infinite stream of natural numbers is implemented as:

```
loop(N):-return(N),N1 is N+1,loop(N1).
```

The following example shows a simple space efficient generator for the infinite stream of prime numbers:

```
prime(P):-prime_engine(E),element_of(E,P).

prime_engine(E):-new_engine(_,new_prime(1),E).

new_prime(N):-N1 is N+1,
  if(test_prime(N1),true,return(N1)),
  new_prime(N1).

test_prime(N):-
  M is integer(sqrt(N)),between(2,M,D),N mod D =:=0
```

Note that the program has been wrapped, using the `element_of` predicate defined in [12], to provide one answer at a time through backtracking. Alternatively, a forward recursing client can use the `get(Engine)` operation to extract primes one at a time from the stream.

9 Applications of Interactors and Practical Language Extensions

Interactors and Multi-Threading As a key difference with typical multi-threaded Prolog implementations like Ciao-Prolog and SWI-Prolog [20, 21], our Interactor API is designed up front with a clear separation between *engines* and *threads* as we prefer to see them as orthogonal language constructs.

While one can build a self-contained lightweight multi-threading API solely by switching control among a number of cooperating engines, with the advent of multi-core CPUs as the norm rather than the exception, the need for *native* multi-threading constructs is justified on both performance and expressiveness grounds. Assuming a dynamic implementation of a logic engine's stacks, Interactors provide lightweight independent computation states that can be easily mapped to the underlying native threading API.

A minimal native Interactor based multi-threading API, has been implemented in the Jinni Prolog system [10] on top of a simple thread launching built-in

```
run_bg(Engine,ThreadHandle)
```

This runs a new Thread starting from the engine's `run()` predicate and returns a handle to the Thread object. To ensure that access to the Engine's state is safe and synchronized, we hide the engine handle and provide a simple producer/consumer data exchanger object, called a `Hub`. Some key components of the multi-threading API, partly designed to match Java's own threading API are:

- `bg(Goal)`: launches a new Prolog thread on its own engine starting with `Goal`.
- `hub_ms(Timeout,Hub)`: constructs a new `Hub` - a synchronization device on which `N` consumer threads can wait with `collect(Hub,Data)` (similar to a synchronized `from_engine` operation) for data produced by `M` producers providing data with `put(Hub,Data)` (similar to a synchronized `from_engine` operation.

Interactor Pools `Thread Pools` have been in use either at kernel level or user level in various operating system and language implementations to avoid costly allocation and deallocation of resources required by Threads. Likewise, for Interactor implementations that cannot avoid high creation/initialization costs, it makes sense to build *Interactor Pools*. An Interactor Pool is maintained by a dedicated Logic Engine that keeps track of the state of various Interactors and provides recently freed handles, when available, to `new_engine` requests.

Associative Interactors The message passing style interaction shown in the previous sections between engines and their clients, can be easily generalized to associative communication through a unification based blackboard interface [22]. Exploring this concept in depth promises more flexible interaction patterns, as out of order `ask_engine` and `engine_yield` operations would become possible, matched by association patterns.

10 Interactors Beyond Logic Programming Languages

We will now compare Interactors with similar constructs in other programming paradigms.

10.1 Interactors in Object Oriented Languages

Extending Interactors to mainstream Object Oriented languages is definitely of practical importance, given the gain in expressiveness. An elegant open source Prolog engine `Yield Prolog` has been recently implemented in terms of Python's

yield and `C#`'s *yield return* primitives [23]. Extending Yield Prolog to support our Interactor API only requires adding the communication operations `from_engine` and `to_engine`. In older languages like `Java`, `C++` or `Objective C` one needs to implement a more complex API, including a `yield return` emulation.

10.2 Interactors and similar constructs in Functional Languages

Interactors based on logic engines encapsulate future computations that can be unrolled on demand. This is similar to lazy evaluation mechanisms in languages like Haskell [24]. Interactors share with Monads [25, 26] the ability to sequentialize functional computations and encapsulate state information. With higher order functions, monadic computations can pass functions to inner blocks. On the other hand, our `ask_engine` / `engine_yield` mechanism, like Ruby's `yield`, is arguably more flexible, as it provides arbitrary switching of control (coroutining) between an Interactor and its client. The ability to define Prolog's `findall` construct as well as `fold` operations in terms of Interactors, is similar to definition of comprehensions [26] in terms of Monads.

11 Conclusion

We have shown that Logic Engines encapsulated as Interactors can be used to build on top of pure Prolog a practical Prolog system, including dynamic database operations, entirely at source level. We have also provided a sketch of an executable semantics for Logic Engine operations in pure Prolog. This shows that, in principle, their exact specification can be expressed declaratively.

In a broader sense, Interactors can be seen as a starting point for rethinking fundamental programming language constructs like Iterators and Coroutining in terms of language constructs inspired by *performatives* in agent oriented programming.

Beyond applications to logic-based language design, we hope that our language constructs will be reusable in the design and implementation of new functional and object oriented languages.

References

1. Liu, J., Kimball, A., Myers, A.C.: Interruptible iterators. In Morrisett, J.G., Jones, S.L.P., eds.: POPL, ACM (2006) 283–294
2. Matsumoto, Y.: The Ruby Programming Language. (June 2000)
3. Sasada, K.: YARV: yet another RubyVM: innovating the ruby interpreter. In Johnson, R., Gabriel, R.P., eds.: OOPSLA Companion, ACM (2005) 158–159
4. Microsoft Corp.: Visual `C#`. Project URL http://msdn.microsoft.com/vcsharp.
5. van Rossum, G.: A Tour of the Python Language. In: TOOLS (23), IEEE Computer Society (1997) 370
6. Liskov, B., Atkinson, R.R., Bloom, T., Moss, J.E.B., Schaffert, C., Scheifler, R., Snyder, A.: CLU Reference Manual. Volume 114 of Lecture Notes in Computer Science. Springer (1981)

7. Griswold, R.E., Hanson, D.R., Korb, J.T.: Generators in Icon. ACM Trans. Program. Lang. Syst. **3**(2) (1981) 144–161
8. Mayfield, J., Labrou, Y., Finin, T.W.: Evaluation of KQML as an Agent Communication Language. In Wooldridge, M., Müller, J.P., Tambe, M., eds.: ATAL. Volume 1037 of Lecture Notes in Computer Science., Springer (1995) 347–360
9. Wegner, P., Eberbach, E.: New Models of Computation. Comput. J. **47**(1) (2004) 4–9
10. Tarau, P.: Orthogonal Language Constructs for Agent Oriented Logic Programming. In Carro, M., Morales, J.F., eds.: Proceedings of CICLOPS 2004, Fourth Colloquium on Implementation of Constraint and Logic Programming Systems, Saint-Malo, France (September 2004)
11. Tarau, P.: BinProlog 11.x Professional Edition: Advanced BinProlog Programming and Extensions Guide. Technical report, BinNet Corp. (2006)
12. Tarau, P.: Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In Lloyd, J., ed.: Computational Logic–CL 2000: First International Conference, London, UK (July 2000) LNCS 1861, Springer-Verlag.
13. Tarau, P., Boyer, M.: Nonstandard Answers of Elementary Logic Programs. In Jacquet, J., ed.: Constructing Logic Programs. J.Wiley (1993) 279–300
14. Tarau, P., Boyer, M.: Elementary Logic Programs. In Deransart, P., Maluszyński, J., eds.: Proceedings of Programming Language Implementation and Logic Programming. Number 456 in Lecture Notes in Computer Science, Springer (August 1990) 159–173
15. Warren, D.H.D.: Higher-order extensions to Prolog – are they needed? In Michie, D., Hayes, J., Pao, Y.H., eds.: Machine Intelligence 10. Ellis Horwood (1981)
16. Bird, R.S., de Moor, O.: Solving optimisation problems with catamorphism. In Bird, R.S., Morgan, C., Woodcock, J., eds.: MPC. Volume 669 of Lecture Notes in Computer Science., Springer (1992) 45–66
17. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2) (1990) 231–248
18. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-logic programming: Extending logic programming with coinduction. In Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A., eds.: ICALP. Volume 4596 of Lecture Notes in Computer Science., Springer (2007) 472–483
19. Tarau, P., Dahl, V.: High-Level Networking with Mobile Code and First Order AND-Continuations. Theory and Practice of Logic Programming **1**(3) (May 2001) 359–380 Cambridge University Press.
20. Carro, M., Hermenegildo, M.V.: Concurrency in prolog using threads and a shared database. In: ICLP. (1999) 320–334
21. Wielemaker, J.: Native preemptive threads in swi-prolog. In Palamidessi, C., ed.: ICLP. Volume 2916 of Lecture Notes in Computer Science., Springer (2003) 331–345
22. De Bosschere, K., Tarau, P.: Blackboard-based Extensions in Prolog. Software — Practice and Experience **26**(1) (January 1996) 49–69
23. Jeff Thompson: Yield Prolog. Project URL http://yieldprolog.sourceforge.net.
24. Peyton Jones, S.L., ed.: Haskell 98 Language and Libraries: The Revised Report. (September 2002) http://haskell.org/definition/haskell98-report.pdf.
25. Moggi, E.: Notions of computation and monads. Information and Computation **93** (1991) 55–92
26. Wadler, P.: Comprehending monads. In: ACM Conf. Lisp and Functional Programming, Nice, France, ACM Press (1990) 61–78

Secure Implementation of Meta-predicates *

Paulo Moura

Dep. of Computer Science, University of Beira Interior, Portugal
Center for Research in Advanced Computing Systems, INESC–Porto, Portugal
pmoura@di.ubi.pt

Abstract. This paper identifies potential security loopholes in the implementation of support for meta-predicates. Closing these loopholes depends on three conditions: a clear distinction between closures and goals, support for an extended meta-predicate directive that allows the specification of closures, and the availability of the `call/2-N` family of built-in meta-predicates. These conditions provide the basis for a set of simple safety rules that allows meta-predicates to be securely supported. These safety rules are currently implemented by Logtalk, an object-oriented logic programming language, and may also be applied in the context of Prolog predicate-based module systems. Experimental results illustrate how these rules can prevent several security problems, including accidental or malicious changes to the original meta-predicate arguments and bypassing of predicate scope rules and predicate scope directives.

Keywords: logic-programming, meta-predicates, security

1 Introduction

Prolog and Logtalk [1, 2] meta-predicates are predicates with one or more arguments that are called as goals on the body of a predicate clause. A typical example is the `findall/3` predicate whose second argument is used for generating solutions that are collected into a list. Meta-arguments may also be closures. In the context of this paper, a closure is defined as a callable term used to construct a goal by appending one or more arguments. The archetypal example is a list mapping predicate that succeeds when a closure can be successfully applied to each element in the list. Meta-predicates are particularly useful in the presence of an encapsulation mechanism such as a module system or an object-oriented extension. Defining an exported or public meta-predicate within a module or an object allows client modules and objects to reuse predicates customized by calls to local predicates.

Meta-predicates require special care in the context of Prolog module systems and object-oriented extensions as meta-calls must be executed in the meta-predicate *calling context* and not in the meta-predicate *definition context*.

* This work is partially supported by the FCT research project MOGGY (PTDC/EIA/70830/2006).

A recent paper [3] showed that the implementation of meta-predicates found in most Prolog *predicate-based* module systems allows a module to call non-exported predicates of another module, thus breaking encapsulation. This problem is usually absent from *atom-based* module systems such as XSB [4] where atoms, including predicate functors, are internally tagged with the definition module. The lack of enforcement of module encapsulation can, however, be thought as a consequence of the original design goals of module systems. Traditional Prolog module systems never aimed to fulfill any security role, being designed instead as a simple solution for partitioning code in different namespaces. Moreover, in most Prolog module systems, any module predicate can be called by using explicit module qualification (Ciao [5, 6] and ECLiPSe [7] are notable exceptions, only allowing calls to exported module predicates). Prolog extensions such as Logtalk, however, are designed to enforce encapsulation and predicate scope rules. In this case, meta-predicates must be properly supported without the danger of providing the means of accidental or malicious bypassing of predicate scope directives. The same paper also exposed flaws in the Logtalk support of meta-predicates which allowed bypassing of predicate scope directives. These flaws resulted from clever use of closures and from unsafe handling of goal execution context in the presence of meta-calls. During our research to correct these problems, we uncovered other meta-predicate implementation flaws that are not necessarily related to bypassing of predicate scope directives. In fact, potential loopholes exist that may allow accidental or carefully crafted meta-predicate definitions to change the original meta-predicate call. These changes may allow calling a different predicate in the calling context or calling the intended predicate with corrupted arguments. Calling a predicate different from the one specified in the original meta-predicate call is always a flaw, even when the called predicate is public or exported. Corrupting the original meta-predicate arguments can be done conditionally, resulting in hard to find problems as only specific usage patterns will lead to compromised results.

Correcting these flaws can be accomplished by finding and implementing a set of safety rules that ensures secure compilation and use of meta-predicates. Although our research takes place in the context of the Logtalk programming language, these safety rules are equally relevant in the context of predicate-based Prolog module systems (the proposed safety rules are not tied to the semantic differences between objects and modules). These safety rules are useful even in the context of module systems that allow the `:/2` control construct to bypass predicate scope rules, promoting better coding standards for meta-predicate definitions.

This paper is organized as follows. Section 2 describes an extended meta-predicate declaration directive, which supports the specification of both goals and closures as meta-arguments. Section 3 discusses how meta-calls can be constructed from closures. Section 4 enumerates potential loopholes in the implementation of meta-predicate support. Section 5 presents and discusses the safety rules applied by Logtalk to compile and execute meta-predicates. Section 6 identifies limitations imposed by our safety rules on meta-predicate definitions. Sec-

tion 7 presents experimental results in testing common Prolog module systems for the loopholes discussed in this paper. Section 8 presents our conclusions on safe compilation and use of meta-predicates, together with some remarks on the importance of increasing the awareness of security issues among the Logic Programming community.

2 Extended Meta-predicate Directive

User meta-predicates are declared using *meta-predicate directives*. These directives use a meta-predicate template to specify which arguments are *meta-arguments*, i.e. which arguments will be used as goals or closures in the body of the meta-predicate clauses. In plain Prolog, meta-predicate directives are optional and primarily useful for cross-reference tools. When module or object systems are present, meta-predicates directives are required for proper compilation of meta-predicates. An example of a Logtalk meta-predicate directive where the meta-arguments are goals is:

```
:- meta_predicate(findall(*, ::, *)).
```

In meta-predicate templates, the atom `::` represents a meta-argument that will be called as a goal. Normal arguments are represented by the atom `*`. This is similar to the declaration of meta-predicates found in most Prolog compilers and in the ISO Prolog standard for modules [8] (the atom `::` is used instead of the atom `:` for consistency with the Logtalk message sending operators). A positive integer, `N`, specifies a closure that will be used to construct a call by appending `N` arguments. For example:

```
| ?- map(double, [1, 2, 3], L).
L = [2, 4, 6]
yes
```

The corresponding `meta_predicate/1` directive would be:

```
:- meta_predicate(map(2, *, *)).
```

The first argument in the `map/3` template specifies that the meta-argument is a closure that will be used to construct a meta-call by appending two arguments. In the example above, this requires the existence of a `double/2` predicate in the calling context of the meta-predicate.

The use of non-negative integers to specify closures was first introduced in Quintus Prolog [9] for providing information to predicate cross-reference tools. A description of this usage can also be found on a recent Prolog standardization proposal [10]. Other Prolog compilers, such as SICStus Prolog [11] and YAP [12], also accept this notation for compatibility with existing code. As discussed later in this paper, the support for specifying closures in meta-predicate directives is essential to ensure safe compilation and use of meta-predicates. The Ciao Prolog system defines an alternative but equivalent syntax for specifying closures, using a compound term `pred(I)` where `I` is the number of extra arguments.

3 From Closures to Meta-calls

Given a closure and its additional arguments, the corresponding meta-call is constructed by appending the extra arguments to the existing ones. Although it is always possible to use the standard predicate `=../2` and a list append predicate to construct the meta-call, the preferable and simpler solution is to use the `call/N` family of built-in meta-predicates found in Logtalk and in most Prolog compilers. The first argument of these predicates must be a closure, with the remaining arguments being interpreted as the closure extra arguments. For example, the query `call(integer, 3)` is equivalent to the query `integer(3)`. These predicates provide improved performance when compared with the explicit construction of meta-calls (which requires building temporary lists).

As discussed later in the paper, the use of the `call/N` family of built-in meta-predicates is mandatory when working with closures as they avoid the introduction of new variables to explicitly represent the constructed meta-calls.

4 Potential Meta-predicate Loopholes

When reasoning about meta-predicate semantics, it is helpful to define a set of terms which helps us visualize how and where meta-calls take place:

Definition context This is the object or module containing the meta-predicate definition.

Calling context This is the object or module from which a meta-predicate is called. This can be the object or module where the meta-predicate is defined in the case of a local call or another object or module assuming that the meta-predicate is within scope.

Execution context This comprises both the calling context and the definition context. It includes all the information needed for the language runtime to execute a meta-predicate call.

Our research is focused on three potential loopholes when implementing meta-predicate support. The first loophole can be exploited to corrupt the original meta-arguments when a meta-predicate is executed:

Making malicious changes to meta-arguments Using unification with the meta-arguments may allow a meta-predicate to test for specific goals and closures and modify them before making the corresponding meta-calls. This potential loophole can be exploited by testing only for some very specific usage patterns, thus making its detection harder.

The two following loopholes can be exploited to bypass predicate scope directives or to break predicate scope rules. In the case of Logtalk, predicate scope rules are supported using predicate scope directives (object predicates are private by default). In the case of Prolog module systems, it should not be possible to call non-exported predicates from client modules.

Hijacking of the predicate execution context Hijacking a predicate execution context may allow a meta-predicate to gain access to predicates within the calling context other than the ones specified in the meta-predicate call.

Using closures for constructing unintended meta-calls A potential loophole exists when appending additional arguments to a closure in order to construct a meta-call. This loophole can be exploited by constructing a call to a predicate with the same functor of the closure but with an arity different to that intended by the caller of the meta-predicate.

5 Compiling Meta-predicates for Safety

This section describes four safety rules, illustrated with examples,[1] intended to close the loopholes discussed above in the context of predicate-based encapsulation module and object systems. The ideal rules would allow catching all problems at compile time. Unfortunately, as we will illustrate in this section, this is not always possible. Some deceiving meta-predicates definitions constitute perfectly valid code; the potential for trouble resulting only from the use of such definitions. For these cases, the compiler can still print a warning. At runtime, our safety rules ensure that any inappropriate use of a meta-predicate definition is caught by generating an appropriate exception.

The first two rules check for the context for meta-predicate calls. The last two rules check for the consistency of meta-predicate directives and the consistency between meta-predicate directives and meta-calls. The rules presentation is conceptual: actual implementations may choose to combine the first and second rules and combine the third and fourth rules. The first three rules are expected to be implemented at the compiler level. The fourth rule may be implemented instead in a programming code style or policy checker.

(a) The meta-arguments on a meta-predicate clause head must be variables.

This simple rule helps to prevent a meta-predicate from modifying the original arguments of a meta-call. By testing and acting upon the actual meta-arguments, a meta-predicate could try to make a meta-call different from the original one to be executed in the calling context. Consider the following example `(a)`:

```
:- object(library).

    :- public(map/3).
    :- meta_predicate(map(*, 2, *)).
    map(In, scale(_), Out) :-
        !, map_(In, scale(3), Out).
    map(In, Closure, Out) :-
        map_(In, Closure, Out).
```

[1] These examples use Logtalk objects. Converting them to Prolog modules requires replacing object directives with module directives, removing the explicit predicate scope directives, and rewriting the meta-predicate directives.

```
    :- meta_predicate(map_(*, 2, *)).
    map_([], _, []).
    map_([X| Xs], Closure, [Y| Ys]) :-
        call(Closure, X, Y),
        map_(Xs, Closure, Ys).

:- end_object.
```

The `map/3` meta-predicate in this `library` object behaves as expected except when the closure argument unifies with the term `scale(_)`. In this case, the original predicate argument is simply ignored and replaced by a fixed value. Assume now that we define the following client object:

```
:- object(client).

    :- public(double/2).
    double(Ints, Doubles) :-
        library::map(Ints, scale(2), Doubles).

    scale(Scale, X, Xscaled) :-
        Xscaled is X*Scale.

:- end_object.
```

In the absence of this safety rule, the compromised behavior of the `map/3` meta-predicate could be illustrated by the following goal:

```
| ?- client::double([1,2,3], Doubles).
Doubles = [3,6,9]
yes
```

By implementing this safety rule, Logtalk generates a compile time error[2] for the first clause of the `map/3` predicate in the `library` object:

```
type_error(variable, scale(_))
```

This rule is, however, easy to circumvent by simply moving the unification from the meta-predicate clause head into the clause body. The meta-predicate `map/3` in the example above can be easily rewritten as:

```
map(In, Closure, Out) :-
    (   Closure = scale(_) ->
        map_(In, scale(3), Out)
    ;   map_(In, Closure, Out)
    ).
```

Despite this weakness, there are three reasons to include this rule. First, it provides a necessary condition for the second safety rule, described next. Second, rule violations result in compile time errors, which are always preferable to runtime errors. Third, it is trivial to implement: the compiler can apply it before any other rule by simply checking the meta-arguments in the clause heads.

[2] Arguably, this error is more of a representation error than a type error; nevertheless, we decided to follow the practice established by the current ISO Prolog standard.

(b) Meta-calls whose arguments are not variables appearing in meta-argument positions in the clause head must be compiled as calls to local predicates.

This rule applies to the compilation of both meta-predicates and normal predicates. It prevents hijacking of the execution context, which could otherwise be used to call predicates in the calling context not passed as meta-arguments. This problem can occur with e.g. a naive implementation of execution context passing from a clause head to the goals in the clause body.

This rule is trivial to implement when compiling clauses of normal predicates: any meta-call in a clause body must be compiled as a local meta-call. This rule is also easy to implement when compiling clauses of meta-predicates since the corresponding meta-predicate directive is mandatory.

As a consequence of this rule, when a meta-predicate calls a second meta-predicate, the meta-arguments executed in the calling context will be strictly the ones coming from the call to the first meta-predicate. That is, the programmer cannot use a second meta-predicate to construct a meta-call different from the one intended by the original caller of the meta-predicate. Consider the following example (`b1`):

```
:- object(library).

    :- public(meta/2).
    :- meta_predicate(meta(::, ::)).
    meta(Goal1, Goal2) :-
        call(Goal1), call(Goal2).

    :- public(meta/1).
    :- meta_predicate(meta(::)).
    meta(Goal1) :-
        meta(Goal1, local).

    local :-
        write('local predicate in object library'), nl.

:- end_object.
```

The rule requires that client calls to the `meta/1` predicate must result in the interpretation of `local/0` as a call to a local predicate, thus executed in the context of the object `library`. We use the following client object to illustrate the correct behavior:

```
:- object(client).

    :- public(test/0).
    test :-
        library::meta(goal).

    goal :-
        write('goal meta-argument in object client'), nl.
```

```
    local :-
        write('local predicate in object client'), nl.

:- end_object.
```

This safety rule will ensure the following result:

```
| ?- client::test.
goal meta-argument in object client
local predicate in object library
yes
```

Meta-calls can also appear in the body of normal predicates. This rule ensures that an object cannot hijack the execution context of the original, non meta-predicate call and use it through a local meta-predicate to construct arbitrary calls to predicates in the calling context. Therefore, we cannot convert a normal argument into a meta-argument by calling a local meta-predicate. Consider the following simplified version of an example found in [3] (`b2`):

```
:- object(library).

    :- meta_predicate(meta(::)).
    meta(Goal) :-
        call(Goal).

    :- public(normal/1).
    normal(Arg) :-
        meta(Arg).

:- end_object.
```

In this case, the argument in the meta-predicate call, `Arg`, must be interpreted as a local meta-call. Consider now the following client object:

```
:- object(client).

    :- public(test/0).
    test :-
        library::normal(term).

    term :-
        write('Some local, private predicate.').

:- end_object.
```

This safety rule will ensure the following result:

```
| ?- catch(client::test, E, write(E)).
E = error(existence_error(procedure,term), context(object,library,_))
yes
```

Therefore, the predicate `term/0` in the object `client` (which is the calling context for the `normal/1` predicate) will not be called.

Although the two examples above make use of additional user-defined meta-predicates whose meta-arguments are goals, the rule also applies when working with closures and when calling built-in meta-predicates. For example, consider the following tentative exploit (`b3`) using the `call/1` built-in meta-predicate and a meta-predicate definition that does not comply with the corresponding directive (as two arguments are appended to the closure instead of one):

```
:- object(library).

    :- public(m/2).
    :- meta_predicate(m(1, *)).
    m(Closure, Arg) :-
        Closure =.. List,
        list::append(List, [Arg, _], NewList),
        Call =.. NewList,
        call(Call).

:- end_object.
```

With this safety rule in place, the meta-call `call(Call)` above is compiled as a local meta-call since the variable `Call` does not occur in the head of the meta-predicate clause in a meta-argument position. The following definition of a simple client object illustrates the consequences of the meta-predicate definition above:

```
:- object(client).

:- public(test/1).
    test(X) :-
        library::m(a, X).

    a(1). a(2).

    a(3, three). a(4, four).

:- end_object.
```

After compiling and loading these two objects, an example test query would be:

```
?- catch(client::test(X), E, true).
E = error(existence_error(procedure, a/2), context(object, library, _))
yes
```

As the exception term shows, the meta-call is compiled and executed as a local call in the context of the `library` object. Without this safety rule in place, a faulty implementation would wrongly call the predicate `a/2` defined in the object `client`:

```
?- catch(client::test(X), E, true).
X = 3 ;
X = 4
yes
```

The above example shows that meta-predicates with meta-arguments that are closures cannot be defined using `call/1` calls as explicitly constructing the meta-call from the closure results in a new variable not occurring in the clause head. It follows that the use of the `call/2-N` built-in predicates is mandatory for defining meta-predicates that work with closures. This is subsumed by the third rule:

(c) Meta-predicate closures must be used within a `call/2-N` built-in predicate call that complies with the corresponding meta-predicate directive.

The number of additional arguments appended to a closure in a `call/2-N` call must comply with the meta-predicate declaration; simply ensuring that a closure is a variable occurring in a meta-argument position is not a sufficient condition. This rule ensures that a meta-predicate cannot construct a predicate call with the same functor but with a different arity of the original meta-argument. For example, a meta-predicate definition `(c)` such as:

```
:- meta_predicate(map(1, *)).
map(Closure, [Element| Rest]) :-
    ..., call(Closure, Element, Result), ...
```

would result in the following compile time error:

```
arity_mismatch(closure, call(map, Element, Result), map(1, *))
```

The `call/3` meta-call in this example does not comply with the meta-predicate specification, which requires a single additional argument. In fact, the actual meta-call would not be the one that the programmer intended when calling the meta-predicate. Moreover, the call could correspond either to a predicate in the calling context that is not within scope of the meta-predicate definition context or to a non-existing predicate (which would result in a runtime existence error).

(d) The meta-predicate arity should be equal to the sum of the extra arguments specified by each closure plus the number of normal, non meta-arguments.

Assume that we correct the meta-predicate directive used to illustrate the previous rule in order to be consistent with the `call/2-N` call by writing `(d)`:

```
:- meta_predicate(map(2, *)).
```

Trying to compile the updated code would result in the following error:

```
arity_mismatch(closure, map(Closure, [Element| Rest]), map(2, *))
```

This error results from the meta-predicate directive specifying a closure requiring two extra arguments while only one normal argument is declared. This is potentially misleading for a client that may expect the library meta-predicate to call a unary predicate based on the meta-predicate arity.

6 Known Limitations

6.1 Closures with a Variable Number of Arguments

The proposed safety rules and the extended meta-predicate directive do not support the specification of meta-predicates that allow a *variable* number of arguments to be appended to a closure. This restriction makes some common meta-predicates such as `apply/2` useless as a public or exported predicate. The usual definition of this predicate is:

```
apply(Closure, Args) :-
    Closure =.. List,
    append(List, Args, NewList),
    Call =.. NewList,
    call(Goal).
```

As the variable `Goal` is not a meta-argument in the clause head, the meta-call `call(Goal)` is compiled as a call to a local predicate (as per the second safety rule) and not as a call to a predicate in the calling context of the meta-predicate. This restriction is not considered, however, a serious limitation as the number of extra closure arguments is usually known *a priori*, therefore allowing the use of the `call/2-N` built-in meta-predicates.

6.2 Meta-predicates Implemented in Foreign Code

Prolog compilers often include libraries with predicates implemented using a foreign language interface. It is also possible to implement meta-predicates this way. A common example is the implementation of callbacks to Prolog code in the context of GUI extensions (see e.g. the SWI-Prolog XPCE package [13]). In this case, the verification of the safety rules described in the previous section would require manual verification of the source code in the foreign language. It should be noted, however, that the use of foreign language resources rises its own set of security issues that goes well beyond meta-predicates issues.

7 Prolog Module Systems

In this section, we test five Prolog compilers for the potential meta-predicates loopholes described earlier: Ciao 1.10#8, ECLiPSe 5.10#141, SICStus Prolog 4.0.2, SWI-Prolog 5.6.59, and YAP 5.1.3. Although there are other Prolog compilers supporting predicate-based module systems, we believe this is a representative set of module implementation solutions.

Our experiments are complicated by two problems. First, the details of the module versions of the examples in Section 4 differ for each compiler due to the lack of a de-facto standard for Prolog module systems.[3] In particular, the five

[3] The full source code used in the examples for both Logtalk and the tested Prolog compilers is available at `http://logtalk.org/papers/simp/mptests.tar.gz`.

tested systems provide three different materializations of a meta-predicate declaration directive. Second, the documentation of the Prolog module systems often forces us to resort to experimentation in order to find out the exact operational semantics of modules, meta-predicate directives, and meta-calls.

The experimental results are presented in Table 1. In this table, a value of `N/A` means that the `meta_predicate/1` directive or its equivalent does not support the specification of meta-predicate templates. The results for the example `(d)` indicate if a Prolog compiler checks for the consistency between meta-predicate directives and the number of extra arguments required by the declared closures. This consistency check should result, at least, in a compilation warning but it is not performed by any of the tested Prolog compilers.

Table 1. Experimental results for the safety rule examples.

Examples	Ciao	ECLiPSe	SICStus	SWI (mp)	SWI (mt)	YAP
(a1)	ok	wrong	ok	wrong	wrong	ok
(a2)	ok	ok	wrong	ok	ok	wrong
(b1)	ok	wrong	ok	wrong	wrong	ok
(b1)	ok	wrong	ok	wrong	wrong	ok
(b2)	ok	ok	ok	wrong	ok	ok
(b3)	ok	wrong	wrong	wrong	wrong	wrong
(c)	ok	N/A	wrong	wrong	N/A	wrong
(d)	wrong	wrong	wrong	wrong	wrong	wrong

The conversion of the Logtalk example `(a)` into Prolog module code rises an interesting issue with the module systems of SICStus Prolog and YAP. These systems expand meta-arguments in goals appearing in the body of meta-predicate clauses but not in the head of meta-predicate clauses. As a consequence, the first clause of the `map/3` is never used, making the test result for these Prolog compilers misleading. One workaround is to rewrite this clause using explicit module qualification, which allows all the clauses to be used. Although this rewrite defeats the purpose of the meta-predicate directive, it is also a possible exploit vector. Therefore, we chose to split the example `(a)` in two tests. Test `(a1)` uses the same exact clauses as in example `(a)`. Test `(a2)` uses explicit module qualification for the `scale/1` arguments in the first clause of the meta-predicate `map/3`.

The results for test `(a2)` are interesting and a bit surprising. While the results for SICStus Prolog and YAP are expected, the changes in test `(a2)` allow both ECLiPSe and SWI-Prolog to return correct results, reversing the bad score in test `(a1)` (it is worth noting that the module systems of ECLiPSe and SWI-Prolog are distinct). The Ciao compiler is not fooled by these tricks.

Another interesting result concerns the `(b2)` and `(b3)` examples of our second security rule, `(b)`. All compilers behaved correctly in example `(b2)`. However, with the exception of Ciao, all compilers provided a wrong answer for example `(b3)`, allowing access to a private predicate, `a/2`, in the client module, instead of restricting the access to the predicate `a/1` used as argument in the meta-predicate call. In this case, these Prolog compilers acted properly when meta-arguments are goals but not when the meta-arguments are closures.

Some brief, Prolog compiler-specific comments about the results follow:

Ciao. This is the only tested Prolog compiler that disallows writing meta-predicate directives inconsistent with the meta-predicate definitions. It is also the Prolog compiler that scored the best test results (as expected, giving the emphasis by Ciao developers in static code analysis). The test results for the third example of our second security rule (`b3`) are particularly interesting. The Ciao compiler correctly catches our attempts to specify a closure with a single extra argument while, at the same time, defining the meta-predicate to call the closure with two extra arguments.[4] Correcting the meta-predicate directive to specify a closure with two extra arguments, however, results in the definition of a meta-predicate that only allows a single extra argument to be passed. The Ciao compiler fails to warn the user of this potential problem when compiling the example (`d`).

ECLiPSE. This compiler does not provide a `meta_predicate/1` directive, relying instead on a proprietary `tool/2` directive whose arguments are predicate indicators. Thus, this directive does not allow the programmer to define meta-predicate templates. The test examples are modified to use the `tool/2` directive and the built-in predicate `@/2` as suggested in the ECLiPSe documentation.

SICStus Prolog. This compiler allows the specification of closures in the directive `meta_predicate/1` but only for compatibility with existing code. Correcting the directive in the test example (`b3`) to make it consistent with the meta-predicate definition does not lead to a correct answer.

SWI-Prolog. We present two sets of results for SWI-Prolog. The first set, *mp*, uses an emulation of the `meta_predicate/1` directive provided in the compatibility libraries distributed with SWI-Prolog. The second set, *mt*, uses the SWI-Prolog native directive `module_transparent/1` whose argument is a predicate indicator. Therefore, it does not allow the programmer to define meta-predicate templates. We are discussing with the main SWI-Prolog developer the possible implementation of our safety rules as a component of a general style or policy checker, integrated with the current cross-referencer tool. This would allow existing code to be checked for possible violations without the danger of breaking it.

YAP. Similarly to SICStus Prolog, YAP accepts the specification of closures in the `meta_predicate/1` directive but only for compatibility with existing code. Correcting the directive in the example (`b3`) to match the meta-predicate definition does not result in a correct answer. The safety rules described in this paper are expected to be implemented in a forthcoming version of YAP. Their use is expected to be optional, enabled by a Prolog compiler flag.

[4] There is a typo in the Ciao documentation of the meta-predicate specification for closures. The notation `pred(N)` indicates the number of extra arguments, with the closure being used within a `call/N+1` predicate, not within a `call/N` predicate as described in the documentation.

8 Discussion and Conclusions

The safety rules described in this paper fix all known flaws on the Logtalk support for meta-predicates.[5] These rules may also be adapted and applied in the context of predicate-based Prolog module systems in order to correct the flaws uncovered by our experiments. However, given the syntactic and semantic differences among the implementations of Prolog modules systems, the existence of other loopholes is to be expected. Nevertheless, the lack of a formal guarantee that the proposed rules close all loopholes in current implementations should not excuse not fixing the known loopholes.

The safety rules are easy to implement and computationally inexpensive, as exemplified in the current Logtalk compiler implementation. These rules enjoy the nice property of all the required computations being performed at compile time. In the worst case, some of the rules imply that the use of a flawed meta-predicate definition results in a runtime exception due to the meta-calls being compiled as calls to local predicates and not as calls in the meta-predicate calling context. This is an unfortunate consequence of the fact that some safety violations only occur when using meta-predicate definitions that, per se, constitute perfectly valid code. It follows that the worst case cannot be improved by finding stronger compiler checking rules. At best, the compiler could issue a warning when compiling a public meta-predicate whose meta-calls are compiled as a local calls for safety reasons.

The extended `meta_predicate/1` directive described in this paper provides essential information for preventing misuse of closures. We show that specifying closures using positive integers is not just an optional feature, useful for cross-reference and documenting tools or for compatibility reasons, but a necessary feature for safe compilation and use of meta-predicates.

Calls constructed from closures must be made by using the `call/2-N` built-in predicates. This allows the consistency between the meta-predicate directives and definitions to be checked at compile time, preventing loopholes when appending arguments to a closure in order to construct a meta-call. The `call/2-N` family of built-in predicates is already provided by most Prolog compilers and is included in the current draft of the ISO Prolog Core revision standardization proposal.[6]

There is currently no formal proof that the described safety rules are sufficient to prevent highjacking of predicate execution context and the misuse of closures in the context of Logtalk. In the case of Prolog module systems each module system needs a proof, as there is no de-facto standard. These proofs would need to be based on formal descriptions of the module systems, to be provided by their authors; these descriptions are beyond the scope of this paper.

[5] All the safety rules are implemented by the Logtalk compiler since version 2.30.6.

[6] In the case of Logtalk, although its current version uses a Prolog system as a back-end compiler, its implementation of the `call/2-N` built-in predicate does not depend on the availability of the `call/2-N` Prolog built-in predicates.

The problems described in this paper are representative of what can go wrong when using meta-predicates in field applications where security is a basic requirement. It is worth noting that the flaws described in this paper are not always evident from a quick inspection of compromised source code (which, by itself, assumes its availability). Despite existing research on improving module systems (see e.g. [3, 6]), security concerns are often overlooked by Prolog implementors and programmers. Secure implementation of meta-predicates is just one of the topics where compilers and language runtimes must perform securely. In a scenario of increasing industrial use of Prolog-based solutions, either in embedded form or as stand-alone applications, preemptive thinking about security issues is necessary. In this regard, the Prolog community is still far from the security mindset found in other programing communities.

Acknowledgements. We are grateful to Rémy Haemmerlé and François Fages for bringing to our attention flaws on the implementation of meta-predicates in earlier versions of Logtalk. We thank also Jan Wielemaker and Sara Madeira for their comments and help in revising this paper.

References

1. Moura, P.: Logtalk 2.33.0 User Manual. (September 2008)
2. Moura, P.: Logtalk – Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal (September 2003)
3. Haemmerlé, R., Fages, F.: Modules for Prolog Revisited. In Etalle, S., Truszczyński, M., eds.: International Conference on Logic Programming 2006. Number 4079 in LNCS, Springer-Verlag (August 2006) 41–55
4. Group, T.X.R.: The XSB Programmer's Manual: version 3.1. (2007)
5. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López, P., Puebla, G.: The Ciao Prolog System. Technical Report CLIP 3/97.1, The CLIP Group, School of Computer Science, Technical University of Madrid (December 2002)
6. Gras, D.C., Hermenegildo, M.V.: A New Module System for Prolog. In: CL'00: Proceedings of the First International Conference on Computational Logic, London, UK, Springer-Verlag (2000) 131–148
7. Cheadle, A.M., Harvey, W., Sadler, A.J., Schimpf, J., Shen, K., Wallace, M.G.: ECLiPSe: A tutorial introduction. Technical Report IC-Parc-03-1, IC-Parc, Imperial College, London (2003)
8. ISO/IEC: International Standard ISO/IEC 13211-2 Information Technology — Programming Languages — Prolog — Part II: Modules. ISO/IEC (2000)
9. for Computer Science, S.I.: Quintus Prolog 3.5 User's Manual. (2003)
10. O'Keefe, R.: An Elementary Prolog Library. `http://www.cs.otago.ac.nz/staffpriv/ok/pllib.htm`
11. for Computer Science, S.I.: SICStus Prolog 4.0.2 User Manual. (2007)
12. Costa, V.S.: The YAP User's Manual: version 5.1.3. (2008)
13. Wielemaker, J., Anjewierden, A.: An Architecture for Making Object-Oriented Systems Available from Prolog. In: Proceedings of the 12th International Workshop on Logic Programming Environments. (2002) 97–110

Tabling Logic Programs in a Common Global Trie

Jorge Costa and Ricardo Rocha

DCC-FC & CRACS
University of Porto, Portugal
c0607002@alunos.dcc.fc.up.pt ricroc@dcc.fc.up.pt

Abstract. The performance of tabled evaluation largely depends on the implementation of the table space. Arguably, the most successful data structure for tabling is tries. However, while tries are efficient for variant based tabled evaluation, they are limited in their ability to recognize and represent repeated answers for different calls. In this paper, we propose a new design for the table space where terms in a tabled subgoal call or/and answer are stored in a common global trie instead of being spread over several different tries. Our preliminary experiments using the YapTab tabling system show very promising reductions on memory usage.

Keywords: Tabling, Table Space, Implementation.

1 Introduction

Tabling [1–3] is an implementation technique where intermediate answers for subgoals are stored and then reused whenever a repeated call appears. The performance of tabled evaluation largely depends on the implementation of the table space – being called very often, fast lookup and insertion capabilities are mandatory. Applications can make millions of different calls, hence compactness is also required. Arguably, the most successful data structure for tabling is *tries* [4]. Tries meet the previously enumerated criteria of efficiency and compactness.

Used in applications that pose many queries, possibly with a large number of answers, tabling can build arbitrarily many and/or very large tables, quickly filling up memory. A possible solution for this problem is to dynamically abolish some of the tables. This can be done using explicit tabling primitives or using a memory management strategy that automatically recovers space among the least recently used tables when memory runs out [5]. An alternative approach is to store tables externally in a relational database management system and then reload them back only when necessary [6].

A complementary approach to the previous problem is to study how less redundant, more compact and more efficient data structures can be used to better represent the table space. While tries are efficient for variant based tabled evaluation, they are limited in their ability to recognize and represent repeated

answers for different calls. In [7], Rao *et al.* proposed a table organization using *Dynamic Threaded Sequential Automata* (DTSA) which recognizes reusable subcomputations for subsumption based tabling. In [8], Johnson *et al.* proposed an alternative to DTSA, called *Time-Stamped Trie* (TST), which not only maintains the time efficiency of the DTSA but has better space efficiency.

In this paper, we propose a different approach. We propose a new design for the table space where all terms in a tabled subgoal call or/and answer are stored in a *common global trie* instead of being spread over several different trie data structures. Our approach resembles the *hash-consing* technique [9], as it tries to share data that is structurally equal. An obvious goal is to save memory usage by reducing redundancy in term representation to a minimum. We will focus our discussion on a concrete implementation, the YapTab system [10, 11], but our proposals can be easy generalized and applied to other tabling systems.

The remainder of the paper is organized as follows. First, we briefly introduce some background concepts about tries and the table space. Next, we describe YapTab's new design for the table space organization using the common global trie and then, we describe how we have extended YapTab to provide engine support for our approach. At last, we present some preliminary experimental results and we end by outlining some conclusions.

2 Table Space

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for tabled subgoals in an appropriate data space, called the *table space*. Whenever a repeated tabled call is found, the subgoal's answers are recalled from the table space instead of being re-evaluated against the program clauses. The table space may be accessed in a number of ways: **(i)** to find out if a subgoal is in the table and, if not, insert it; **(ii)** to verify whether a newly found answer is already in the table and, if not, insert it; and **(iii)** to load answers to variant subgoals. With these requirements, YapTab implements its table space using *tries* [12] which is regarded a very efficient way to implement tables [4].

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term. Each root-to-leaf path represents a term described by the tokens labelling the nodes traversed. Two terms with common prefixes will branch off from each other at the first distinguishing token. For example, the tokenized form of the term $p(X, q(Y, X), Z)$ is the stream of 6 tokens: $p/3, VAR_0, q/2, VAR_1, VAR_0, VAR_2$. Variables are represented using the formalism proposed by Bachmair *et al.* [13], where each variable in a term is represented as a distinct constant. Formally, this corresponds to a function, $numbervar()$, from the set of variables in a term t to the sequence of constants $VAR_0, ..., VAR_N$, such that $numbervar(X) < numbervar(Y)$ if X is encountered before Y in the left-to-right traversal of t.

Internally, the trie nodes are 4-field data structures. The first field stores the node's token, the second field stores a pointer to the node's first child, the third field stores a pointer to the node's parent and the fourth field stores a pointer

to the node's next sibling. Each node's outgoing transitions may be determined by following the child pointer to the first child node and, from there, continuing through the list of sibling pointers. To increase performance, YapTab enforces the *substitution factoring* [4] mechanism and implements tables using two levels of tries - one for subgoal calls, the other for computed answers. More specifically, the table space of YapTab is organized in the following way:

- each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the predicate's *subgoal trie*.
- each different subgoal call is represented as a unique path in the subgoal trie, starting at the predicate's table entry and ending in a *subgoal frame* data structure, with the argument terms being stored within the path's nodes.
- the subgoal frame data structure acts as an entry point to the *answer trie*.
- each different subgoal answer is represented as a unique path in the answer trie. Oppositely to subgoal tries, answer trie paths hold just the substitution terms for the free variables which exist in the argument terms of the corresponding subgoal call.
- the leaf's child pointer of answers is used to point to the next available answer, a feature that enables answer recovery in insertion order. The subgoal frame has internal pointers that point respectively to the first and last answer on the trie. Whenever a variant subgoal starts consuming answers, it sets a pointer to the first leaf node. To consume the remaining answers, it must follow the leaf's linked list, setting the pointer as it consumes answers along the way. Answers are loaded by traversing the answer trie nodes bottom-up.

An example for a tabled predicate `t/2` is shown in Figure 1. Initially, the subgoal trie is empty. Then, the subgoal `t(a(1),X)` is called and three trie nodes are inserted: one for the functor `a/1`, a second for the constant `1` and one last for variable `X`. The subgoal frame is inserted as a leaf, waiting for the answers. Next, the subgoal `t(a(2),X)` is also called. It shares one common node with `t(a(1),X)` but, having `a/1` a different argument, two new trie nodes and a new subgoal frame are inserted. At the end, the answers for each subgoal are stored in the corresponding answer trie as their values are computed. Note that, for this particular example, the completed answer trie for both subgoal calls is exactly the same.

3 Common Global Trie

We next describe YapTab's new design for the table space organization. In this new design, all terms in a tabled subgoal call or/and answer are now stored in a common global trie (GT) instead of being spread over several different trie data structures. The GT data structure still is a tree structure where each different path through the trie nodes corresponds to a term. However, here a term can end at any internal trie node and not necessarily at a leaf trie node.

The previous subgoal trie and answer trie data structures are now represented by a unique level of trie nodes that point to the corresponding terms in the GT

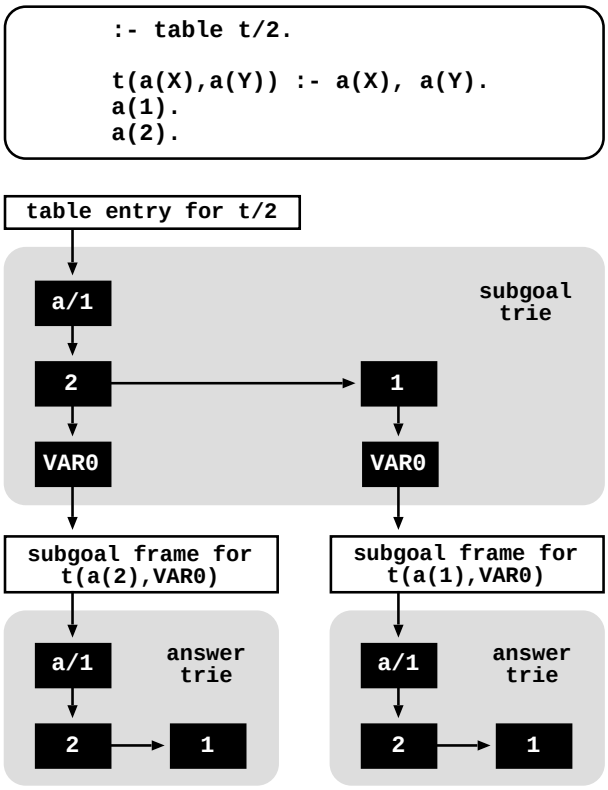


Fig. 1. YapTab's original table design

(see Figure 2 for details). For the subgoal tries, each node now represents a different subgoal call where the node's token is the pointer to the unique path in the GT that represents the argument terms for the subgoal call. The organization used in the subgoal tries to maintain the list of sibling nodes and to access the corresponding subgoal frames remains unaltered. For the answer tries, each node now represents a different subgoal answer where the node's token is the pointer to the unique path in the GT that represents the substitution terms for the free variables which exist in the argument terms. The organization used in the answer tries to maintain the list of sibling nodes and to enable answer recovery in insertion order remains unaltered. With this organization, answers are now loaded by following the pointer in the node's token and then by traversing the corresponding GT's nodes bottom-up.

On completion of a subgoal, a strategy exists that avoids answer recovery using bottom-up unification and performs instead what is called a *completed table optimization* [4]. This optimization implements answer recovery by top-down traversing the completed answer trie and by executing specific WAM-like code from the answer trie nodes. With our new design, the nodes in the GT can

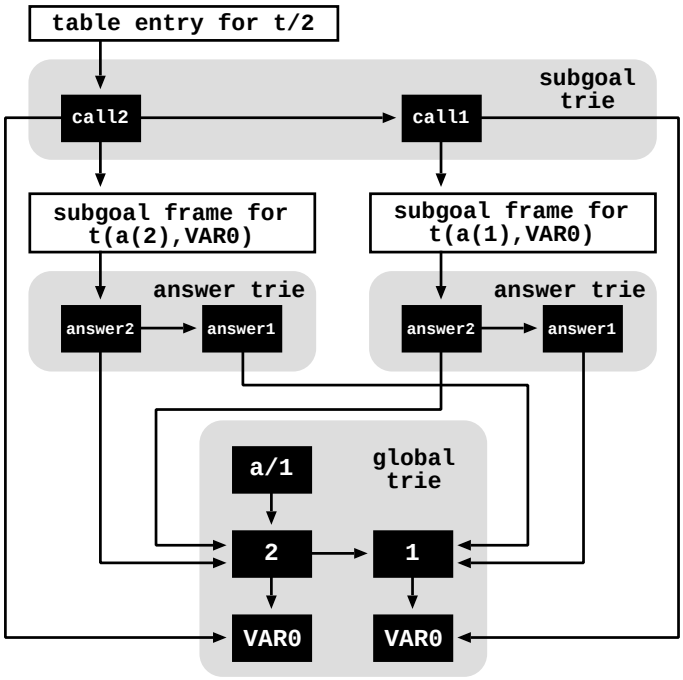


Fig. 2. YapTab's new table design

belong to several different subgoal/answer tries, and thus this optimization is no longer possible.

Figure 2 uses again the example from Figure 1 to illustrate how the GT's design works. Initially, the subgoal trie and the GT are empty. Then, the first subgoal `t(a(1),X)` is called and three nodes are inserted on the GT: one to represent the functor `a/1`, another for the constant `1` and a last one for variable `X`. Next, a node representing the path inserted on the GT is stored in the subgoal trie (node labeled `call1`). The token field for the `call1` node is made to point to the leaf node of the GT's inserted path and the child field is made to point to a new subgoal frame. For the second subgoal call, `t(a(2),X)`, we start again by inserting the call in the GT and then we store a node in the subgoal trie (node labeled `call2`) to represent the path inserted on the GT.

As we saw in the previous example, for each subgoal call we have two answers: the terms `a(1)` and `a(2)`. However, as these terms are already represented on the GT, we need to store only two nodes, in each answer trie, to represent them (nodes labeled `answer1` and `answer2`). The token field for these answer trie nodes are made to point to the corresponding term representation on the GT. With this example we can see that terms in the GT can end at any internal trie node (and not necessarily at a leaf trie node) and that a common path on the GT can simultaneously represent different subgoal and answer terms.

4 Implementation Details

We then describe in more detail the data structures and algorithms for YapTab's new table design based on the GT. We start with Figure 3 showing in more detail the table organization previously presented in Figure 2.

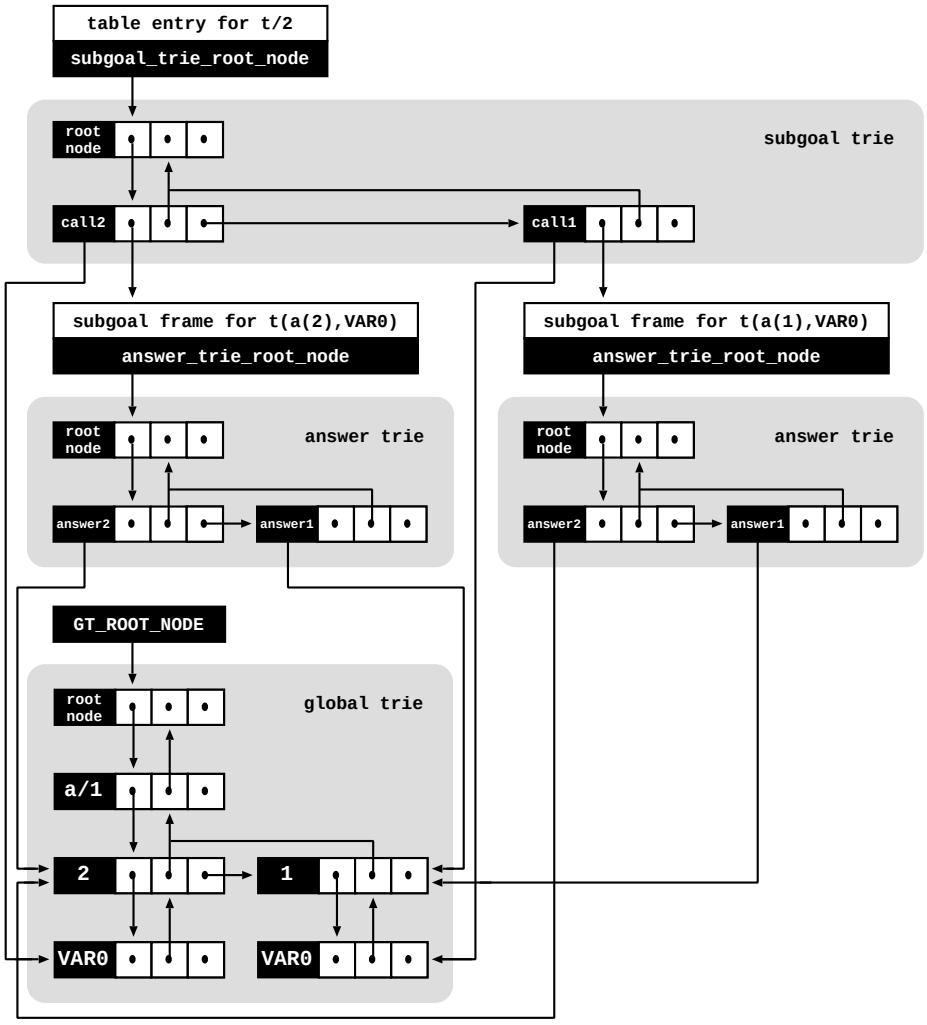


Fig. 3. Implementation details for YapTab's new table design

Internally, tries are represented by a top *root node*, acting as the entry point for the corresponding subgoal, answer or global trie data structure. For the subgoal tries, the root node is stored in the corresponding table entry's

`subgoal_trie_root_node` data field. For the answer tries, the root node is stored in the corresponding subgoal frame's `answer_trie_root_node` data field. For the global trie, the root node is stored in the `GT_ROOT_NODE` global variable.

Regarding the trie nodes, remember that they are internally implemented as 4-field data structures. The first field (`token`) stores the token for the node and the second (`child`), third (`parent`) and fourth (`sibling`) fields store pointers, respectively, to the first child node, to the parent node, and to the sibling node.

Traversing a trie to check/insert for new calls or for new answers is implemented by repeatedly invoking a `trie_node_check_insert()` procedure for each token that represents the call/answer being checked. Given a trie node `parent` and a token `t`, the `trie_node_check_insert()` procedure returns the child node of `parent` that represents the given token `t`. Figure 4 shows the pseudo-code for this procedure.

```
trie_node_check_insert(TRIE_NODE parent, TOKEN t) {
  child = parent->child
  if (child == NULL) {              // the list of sibling nodes is empty
    child = new_trie_node(t, NULL, parent, NULL)
    parent->child = child
  } if (is_not_a_hash_table(child)) {    // sibling nodes without hashing
    sibling_nodes = 0             // to count the number of sibling nodes
    do {           // check if token t is already in the list of siblings
      if (child->token == t)
        return child
      sibling_nodes++
      child = child->sibling
    } while (child)
    child = new_trie_node(t, NULL, parent, parent->child)
    if (sibling_nodes > MAX_SIBLING_NODES_PER_LEVEL) {  // alloc new hash
      hash = new_hash_table(child)
      parent->child = hash
    } else
      parent->child = child
  } else {                                  // sibling nodes with hashing
    hash = child
    bucket = hash_function(hash, t)    // get the hash bucket for token t
    child = bucket
    sibling_nodes = 0
    while (child) {     // check if token t is already in the hash bucket
      if (child->token == t)
        return child
      sibling_nodes++
      child = child->sibling
    }
    child = new_trie_node(t, NULL, parent, bucket)
    if (sibling_nodes > MAX_SIBLING_NODES_PER_BUCKET)      // expand hash
      expand_hash_table(hash)
  }
  return child
}
```

Fig. 4. Pseudo-code for the `trie_node_check_insert()` procedure

Initially, the procedure checks if the list of sibling nodes is empty. If this is the case, a new trie node representing the given token `t` is initialized and inserted as the first child of the given `parent` node. To initialize new trie nodes, we use a `new_trie_node()` procedure with four arguments, each one corresponding to the initial values to be stored respectively in the `token`, `child`, `parent` and `sibling` fields of the new trie node.

Otherwise, if the list of sibling nodes is not empty, the procedure checks if they are being indexed through a hash table. Searching through a list of sibling nodes is initially done sequentially. This could be too expensive if we have hundreds of siblings. A threshold value (`MAX_SIBLING_NODES_PER_LEVEL`) controls whether to dynamically index the nodes through a hash table, hence providing direct node access and optimizing search. Further hash collisions are reduced by dynamically expanding the hash tables when a second threshold value (`MAX_SIBLING_NODES_PER_BUCKET`) is reached for a particular hash bucket.

If not using hashing, the procedure then traverses sequentially the list of sibling nodes and checks for one representing the given token `t`. If such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the list. If reaching the threshold value `MAX_SIBLING_NODES_PER_LEVEL`, a new hash table is initialized and inserted as the first child of the given `parent` node.

If using hashing, the procedure first calculates the hash bucket for the given token `t` and then, it traverses sequentially the list of sibling nodes in the bucket checking for one representing `t`. Again, if such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the bucket list. If reaching the threshold value `MAX_SIBLING_NODES_PER_BUCKET`, the current hash table is expanded.

To manipulate tries we use two interface procedures. For traversing a trie to check/insert for new calls or for new answers we use the

```
trie_check_insert(TRIE_NODE root, TERM term)
```

procedure, where `root` is the root node of the trie to be used and `term` is the call/answer term to be inserted. The `trie_check_insert()` procedure invokes repeatedly the previous `trie_node_check_insert()` procedure for each token that represents the given `term` and returns the reference to the leaf node representing its path. Note that inserting a term requires in the worst case allocating as many nodes as necessary to represent its complete path. On the other hand, inserting repeated terms requires traversing the trie structure until reaching the corresponding leaf node, without allocating any new node.

To load a term from a trie back to the Prolog engine we use the

```
trie_load(TRIE_NODE leaf)
```

procedure, where `leaf` is the reference to the leaf node of the term to be returned. When loading a term, the trie nodes are traversed in bottom-up order.

When inserting terms in the table space we need to distinguish two situations: **(i)** inserting tabled calls in a subgoal trie structure; and **(ii)** inserting

answers in a particular answer trie structure. The former situation is handled by the `subgoal_check_insert()` procedure as shown in Figure 5 and the latter situation is handled by the `answer_check_insert()` procedure as shown in Figure 6.

```
subgoal_check_insert(TABLE_ENTRY te, SUBGOAL_CALL call) {
  st_root_node = te->subgoal_trie_root_node
  if (GT_ROOT_NODE) {                                  // new table design
    leaf_gt_node = trie_check_insert(GT_ROOT_NODE, call)
    leaf_st_node = trie_node_check_insert(st_root_node, leaf_gt_node)
  } else {                                            // original table design
    leaf_st_node = trie_check_insert(st_root_node, call)
  }
  return leaf_st_node
}
```

Fig. 5. Pseudo-code for the `subgoal_check_insert()` procedure

In the original table design, the `subgoal_check_insert()` procedure simply uses the `trie_check_insert()` procedure to check/insert the given `call` in the subgoal trie corresponding to the given table entry `te`. In the new design based on the GT, the `subgoal_check_insert()` procedure now first checks/inserts the given `call` in the GT. Then, it uses the reference to the GT's leaf node representing `call` (`leaf_gt_node` in Figure 5) as the token to be checked/inserted in the subgoal trie corresponding to the given table entry `te`. Note that this is done by calling the `trie_node_check_insert()` procedure, thus if the list of sibling nodes in the subgoal trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, then a new hash table is initialized as described before.

```
answer_check_insert(SUBGOAL_FRAME sf, ANSWER answer) {
  at_root_node = sf->answer_trie_root_node
  if (GT_ROOT_NODE) {                                  // new table design
    leaf_gt_node = trie_check_insert(GT_ROOT_NODE, answer)
    leaf_at_node = trie_node_check_insert(at_root_node, leaf_gt_node)
  } else {                                            // original table design
    leaf_at_node = trie_check_insert(at_root_node, answer)
  }
  return leaf_at_node
}
```

Fig. 6. Pseudo-code for the `answer_check_insert()` procedure

The `answer_check_insert()` procedure works similarly. In the original table design, it checks/inserts the given `answer` in the answer trie corresponding to the given subgoal frame `sf`. In the new design based on the GT, it first checks/inserts the given `answer` in the GT and, then, it uses the reference to the GT's leaf node representing `answer` (`leaf_at_node` in Figure 6) as the token to be checked/inserted in the answer trie corresponding to the given sub-

goal frame `sf`. Again, if the list of sibling nodes in the answer trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, a new hash table is initialized.

Finally, the `answer_load()` procedure is used to consume answers. Figure 7 shows the pseudo-code for it. In the original table design, it simply uses the `trie_load()` procedure to load from the answer trie the answer given by the trie node `leaf_at_node`. In the new design based on the GT, the `answer_load()` procedure first accesses the GT's leaf node represented in the `token` field of the given trie node `leaf_at_node` (`leaf_gt_node` in Figure 7). Then, it uses the `trie_load()` procedure to load from the GT back to the Prolog engine the answer represented by the obtained GT's leaf node.

```
answer_load(ANSWER_TRIE_NODE leaf_at_node) {
  if (GT_ROOT_NODE) {                                  // new table design
    leaf_gt_node = leaf_at_node->token
    answer = trie_load(leaf_gt_node)
  } else {                                          // original table design
    answer = trie_load(leaf_at_node)
  }
  return answer
}
```

Fig. 7. Pseudo-code for the `answer_load()` procedure

5 Preliminary Experimental Results

We next present some preliminary experimental results comparing YapTab with and without support for the common global trie data structure. The environment for our experiments was an AMD Athlon XP 2800+ with 1 GByte of main memory and running the Linux kernel 2.6.24-19.

To evaluate the impact of our proposal, we have defined a tabled predicate `t/5` that simply stores in the table space terms defined by `term/1` facts, and then we used a top query goal `test/0` to recursively call `t/5` with all combinations of one and two free variables in the arguments. An example of such code for functor terms of arity 1 (500 terms in total) is shown next.

```
:- table t/5.
t(A,B,C,D,E) :- term(A), term(B), term(C), term(D), term(E).

test :- t(A,f(1),f(1),f(1),f(1)), fail.        term(f(1)).
...                                            term(f(2)).
test :- t(f(1),f(1),f(1),f(1),A), fail.        ...
test :- t(A,B,f(1),f(1),f(1)), fail.           term(f(499)).
...                                            term(f(500)).
test :- t(f(1),f(1),f(1),A,B), fail.
test.
```

We experimented the `test/0` predicate with 7 different kinds of 500 `term/1` facts: integers, atoms and functor terms of arity 1 to 5. Table 1 shows the memory

usage, in KBytes, and the running times, in milliseconds, to store to the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers for YapTab with (column ***YapTab+GT***) and without (column ***YapTab***) support for the common global trie data structure.

Terms	*YapTab (a)*			*YapTab+GT (b)*			*Ratio (b)/(a)*		
	Mem	*Store*	*Load*	*Mem*	*Store*	*Load*	*Mem*	*Store*	*Load*
500 int	49074	490	155	52803	738	164	1.08	1.51	1.06
500 atom	49074	508	158	52803	770	167	1.08	1.52	1.06
500 f/1	49172	693	242	52811	1029	243	1.07	1.48	1.00
500 f/2	98147	842	314	56725	1298	310	**0.58**	1.54	**0.99**
500 f/3	147122	1098	377	60640	1562	378	**0.41**	1.42	1.00
500 f/4	196097	1258	512	64554	1794	435	**0.33**	1.43	**0.85**
500 f/5	245072	1418	691	68469	2051	619	**0.28**	1.45	**0.90**

Table 1. Memory usage (in KBytes) and store/load times (in milliseconds) for YapTab with and without support for the common global trie data structure

The results show that GT support can reduce memory usage proportionally to the depth and redundancy of the terms stored in the GT. In particular, for functor terms of arity 2 to 5, the results show an increasing and very significant reduction on memory usage. The results for integer and atoms terms are also very interesting as they show that the cost of representing only atomic terms in the GT (between 7% and 8% in these experiments) can be manageable when we increase redundancy. Note that integers and atoms terms are represented by a single node in the original YapTab design, and by an extra node (therefore requiring two nodes) if using the GT approach.

On the other hand, these results seem to indicate that memory reduction comes at a price in execution time. With GT support, we need to navigate in two tries when checking/inserting a term. Moreover, in some situations, the cost of inserting a new term in an empty/small trie can be less than the cost of navigating in the GT, even when the term is already stored in the GT. However, our results seem to suggest that this cost decreases also proportionally to the depth and redundancy of the terms stored in the GT.

The results obtained for loading terms do not suggest significant differences. However and surprisingly, the GT approach showed to outperform the original YapTab design in some experiments.

6 Conclusions and Further Work

We have presented a new design for the table space organization that uses a common global trie to store terms in tabled subgoal calls and answers. Our goal is to reduce redundancy in term representation, thus saving memory by sharing data that is structurally equal. Our preliminary experiments showed very significant reductions on memory usage.

Further work will include exploring the impact of applying our proposal to real-world applications that pose many subgoal queries, possibly with a large number of redundant answers, such as ILP applications, seeking real-world experimental results allowing us to improve and expand our current implementation. In particular, we intend to study how alternative designs for the table space organization can further reduce redundancy in term representation.

Acknowledgements

This work has been partially supported by the research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/ EIA/66924/2006) and by Fundação para a Ciência e Tecnologia.

References

1. Michie, D.: Memo Functions and Machine Learning. Nature **218** (1968) 19–22
2. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
3. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
4. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (1999) 31–54
5. Rocha, R.: On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In: International Symposium on Practical Aspects of Declarative Languages. Number 4354 in LNCS, Springer-Verlag (2007) 155–169
6. Costa, P., Rocha, R., Ferreira, M.: Tabling Logic Programs in a Database. In: Workshop on (Constraint) Logic Programming. (2007) 125–135
7. Rao, P., Ramakrishnan, C.R., Ramakrishnan, I.V.: A Thread in Time Saves Tabling Time. In: Joint International Conference and Symposium on Logic Programming, The MIT Press (1996) 112–126
8. Johnson, E., Ramakrishnan, C.R., Ramakrishnan, I.V., Rao, P.: A Space Efficient Engine for Subsumption-Based Tabled Evaluation of Logic Programs. In: Fuji International Symposium on Functional and Logic Programming. Number 1722 in LNCS, Springer-Verlag (1999) 284–300
9. Goto, E.: Monocopy and Associative Algorithms in Extended Lisp. Technical Report TR 74-03, University of Tokyo (1974)
10. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
11. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming **5**(1 & 2) (2005) 161–205
12. Fredkin, E.: Trie Memory. Communications of the ACM **3** (1962) 490–499
13. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: International Joint Conference on Theory and Practice of Software Development. Number 668 in LNCS, Springer-Verlag (1993) 61–74

Efficient Evaluation of Deterministic Tabled Calls

Miguel Areias and Ricardo Rocha

DCC-FC & CRACS
University of Porto, Portugal
c0507028@alunos.dcc.fc.up.pt ricroc@dcc.fc.up.pt

Abstract. The execution model in which most tabling engines are based allocates a choice point whenever a new tabled subgoal is called. This happens even when the call is deterministic. However, some of the information from the choice point is never used when evaluating deterministic tabled calls with batched scheduling. Thus, if tabling is applied to a long deterministic computation, the system may end up consuming a huge amount of memory inadvertently. In this paper, we propose a solution that reduces this memory overhead to a minimum. Our results show that, for deterministic tabled calls with batched scheduling, it is possible not only to reduce the memory usage overhead, but also the running time of the evaluation.

Keywords: Tabling, Deterministic Calls, Implementation.

1 Introduction

Tabling [1, 2] is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. Implementations of tabling are now widely available in systems like XSB Prolog [3], Yap Prolog [4], B-Prolog [5], ALS-Prolog [6], Mercury [7] and more recently Ciao Prolog [8]. Actual implementations differ in the execution rule, in the data structures used to implement tabling, and in the changes to the underlying Prolog engine. Arguably, the SLG-WAM [9] is the most popular execution rule, but even here several issues require careful research, such as engine integration, execution data structures, termination detection, and scheduling support.

The increasing interest in tabling technology led to further developments and proposals that improve some practical deficiencies of current tabling execution models in key aspects of tabled evaluation like re-computation [10, 11], scheduling [12] and memory recovery [13]. The discussion we address in this work also results from practical deficiencies that we have found in the execution data structures used to evaluate deterministic tabled calls if applying batched scheduling [14].

The execution model in which most tabling engines are based allocates a choice point whenever a new tabled subgoal is called. This happens even when the call is deterministic, i.e., defined by a single matching clause. This is necessary

since the information from the choice point is crucial to correctly implement some tabling operations. However, some of this information is never used when evaluating deterministic tabled calls with batched scheduling. Thus, if tabling is applied to a long deterministic computation, the system may end up consuming a huge amount of memory inadvertently. In this paper, we propose a solution that reduces this memory overhead to a minimum. We will focus our discussion on a concrete implementation, the YapTab system [4], an efficient suspension-based tabling engine that extends the state-of-the-art Yap Prolog system [15] to support tabled evaluation for definite programs, but our proposal can be generalized and applied to other tabling engines.

The remainder of the paper is organized as follows. First, we briefly introduce the main background concepts about tabled evaluation. Next, we discuss in more detail how YapTab compiles and dynamically indexes deterministic tabled calls. We then describe how we have extended YapTab to provide engine support to efficiently deal with deterministic tabled calls. At last, we present some preliminary experimental results and we end by outlining some conclusions.

2 Basic Tabling Concepts

Tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears[1]. Whenever a tabled subgoal is first called, a new entry is allocated in an appropriated data space, the *table space*. Table entries are used to collect the answers found for their corresponding subgoals. Moreover, they are also used to verify whether calls to subgoals are repeated. Repeated calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all repeated calls. Within this model, the nodes in the search space are classified as either: *generator nodes*, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to repeated calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals.

The YapTab design follows the seminal SLG-WAM design [9]: it extends WAM's execution model [16] with a new data area, the *table space*; a new set of registers, the *freeze registers*; an extension of the standard trail, the *forward trail*; and four new operations for definite programs:

Tabled Subgoal Call: this operation is a call to a tabled subgoal. It checks if the subgoal is in the table space. If so, it allocates a consumer node and starts consuming the available answers. If not, it adds a new entry to the table space, and allocates a new generator node. When the call is deterministic, the tabled subgoal call operation is implemented by the `table_try_single` WAM-like instruction.

[1] We say that a subgoal repeats a previous subgoal if they are the same up to variable renaming.

New Answer: this operation verifies whether a newly found answer is already in the table, and if not, inserts the answer. Otherwise, the operation fails.

Answer Resolution: this operation verifies whether extra answers are available for a particular consumer node and, if so, consumes the next one. If no answers are available, it suspends the current computation and schedules a possible resolution to continue the execution.

Completion: this operation determines whether a tabled subgoal is completely evaluated. A subgoal is said to be complete when no more answers can be generated, that is, when its set of stored answers represent all the conclusions that can be inferred from the set of facts and rules in the program. If the subgoal has been completely evaluated, the operation closes the subgoal's table entry and reclaims stack space. Otherwise, control moves to a consumer with unconsumed answers.

During tabled evaluation, at several points, we can choose between continuing forward execution, backtracking to interior nodes, returning answers to consumer nodes, or performing completion. The decision on which operation to perform is determined by the *scheduling strategy*. Different strategies may have a significant impact on performance, and may lead to a different ordering of solutions to the query goal. Arguably, the two most successful tabling scheduling strategies are batched scheduling and local scheduling [14]. YabTab supports both batched scheduling, local scheduling and the dynamic intermixing of batched and local scheduling at the subgoal level [12]. Local scheduling does not have any relevance for this work, so we will not consider it.

Batched scheduling schedules the program clauses in a depth-first manner as does the WAM. It favors forward execution first, backtracking next, and consuming answers or completion last. It thus tries to delay the need to move around the search tree by batching the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues. For some situations, this results in creating dependencies to older subgoals, therefore enlarging the current SCC (*Strongly Connected Component*) and delaying the completion point to an older generator node. By default in YapTab, tabled predicates are evaluated using batched scheduling [12].

3 Deterministic Tabled Calls in YapTab

In this section we discuss how tabled predicates are compiled in YapTab and, in particular, we show how YapTab uses the Yap compiler to generate compiled and indexed code for deterministic tabled calls.

3.1 Compilation of Tabled Predicates

Tabled predicates defined by several clauses are compiled using the `table_try_me`, `table_retry_me` and `table_trust_me` WAM-like instructions in a similar manner to the generic `try_me`/`retry_me`/`trust_me` WAM sequence. The `table_try_me`

instruction extends the WAM's `try_me` instruction to support the tabled subgoal call operation. The `table_retry_me` and `table_trust_me` differ from the generic WAM instructions in that they restore a generator choice point rather than a standard WAM choice point. Tabled predicates defined by a single clause are compiled using the `table_try_single` WAM-like instruction. This instruction optimizes the `table_try_me` instruction for the case when the tabled predicate is defined by a single clause. Figure 1 shows the YapTab's compiled code for a tabled predicate `t/1` defined by a single clause and for a tabled predicate `t/3` defined by several clauses.

```
% predicate definitions
:- table t/1.
t(X) :- ...

:- table t/3.
t(a1,b1,c1) :- ...
t(a2,b2,c2) :- ...
t(a2,b1,c3) :- ...
t(a2,b3,c1) :- ...
t(a3,b1,c2) :- ...


% compiled code generated by YapTab for predicate t/1
t1_1:  table_try_single t1_1a
t1_1a: 'WAM code for clause t(X) :- ...'


% compiled code generated by YapTab for predicate t/3
t3_1:  table_try_me t3_2
t3_1a: 'WAM code for clause t(a1,b1,c1) :- ...'
t3_2:  table_retry_me t3_3
t3_2a: 'WAM code for clause t(a2,b2,c2) :- ...'
t3_3:  table_retry_me t3_4
t3_3a: 'WAM code for clause t(a2,b1,c3) :- ...'
t3_4:  table_retry_me t3_5
t3_4a: 'WAM code for clause t(a2,b3,c1) :- ...'
t3_5:  table_trust_me
t3_5a: 'WAM code for clause t(a3,b1,c2) :- ...'
```

Fig. 1. Compilation of tabled predicates in YapTab

As `t/1` is a deterministic tabled predicate, the `table_try_single` instruction will be executed for every call to this predicate. On the other hand, `t/3` is a non-deterministic tabled predicate, but some calls to this predicate can be deterministic, i.e., defined by a single matching clause. Consider, for example, the previous definition of `t/3` and the calls `t(a3,X,Y)` and `t(X,Y,c3)`. These two calls are deterministic as they only match with a single `t/3` clause, respectively, the 5th and 3rd clause. We next show how YapTab uses the demand-driven indexing mechanism of Yap to dynamically generate `table_try_single` instructions for this kind of deterministic calls.

3.2 Demand-Driven Indexing

Yap implements *demand-driven indexing* (or *just-in-time indexing*) [17] since version 5. The idea behind it is to generate flexible multi-argument indexing of Prolog clauses during program execution based on actual demand. This feature is implemented for static code, dynamic code and the internal database. All indexing code is generated on demand for all and only for the indices required. This is done by building an indexing tree using similar building blocks to the WAM but it generates indices based on the instantiation on the current goal, and expands indices given different instantiations for the same goal.

This powerful optimization provides that YapTab can execute calls to non-deterministic tabled predicates like deterministic tabled predicates. This happens when Yap's indexing scheme finds that for a particular call to a non-deterministic tabled predicate, there is only a single clause that matches the call. Figure 2 shows an example illustrating the indexed code generated for a non-deterministic call and two deterministic calls to the previous `t/3` tabled predicate.

```
% indexed code generated by YapTab for call t(a2,X,Y)
table_try   t3_2a
table_retry t3_3a
table_trust t3_4a

% indexed code generated by YapTab for call t(a3,X,Y)
table_try_single t3_5a

% indexed code generated by YapTab for call t(X,Y,c3)
table_try_single t3_3a
```

Fig. 2. Demand-driven indexing of tabled predicates in YapTab

The call `t(a2,X,Y)` is non-deterministic as it matches the 2nd, 3rd and 4th clauses of `t/3`, so a `table_try`/`table_retry`/`table_trust` sequence is generated. The other two calls, `t(a3,X,Y)` and `t(X,Y,c3)`, are both deterministic as they only match a single `t/3` clause, so a `table_try_single` instruction can be generated. Note however, that there are situations where a call can be deterministic, but Yap's indexing scheme cannot detect it as deterministic in order to generate the appropriate `table_try_single` instruction. In such cases, we cannot benefit directly from our approach, but we can take advantage of the similarities between the `table_try_single` instruction and the *last matching clause* of a non-deterministic tabled call to apply our approach later.

3.3 Last Matching Clause

When evaluating a tabled predicate, the last matching clause of a call to the predicate is implemented either by the `table_trust_me` instruction or by the `table_trust` instruction. The former situation occurs when we have a generic

call to the predicate (all the arguments of the call are unbound variables) and the latter situation occurs when we have a more specific call (some of the arguments are at least partially instantiated) optimized by indexing code.

In a WAM-based implementation [16], the last matching clause of a call is implemented by first restoring all the necessary information from the current choice point (usually pointed to by the WAM's `B` register) and then, by discarding the current choice point by updating `B` to its predecessor. In a tabled implementation, the `table_trust_me` and `table_trust` instructions also restore all the necessary information from the current choice point `B`, but instead of updating `B` to its predecessor, they update the next clause field of `B` to the `completion` instruction. By doing that, they force completion detection when the computation backtracks again to `B`, i.e., whether the clauses for the subgoal call at hand are all exploited.

Hence, the computation state that we have when executing a `table_trust_me` or `table_trust` instruction is similar to that one of a `table_try_single` instruction, that is, in both cases the current clause can be seen as deterministic as it is the last (or single) matching clause for the subgoal call at hand. Thus, we can view the `table_trust_me` and `table_trust` instructions as a special case of the `table_try_single` instruction. This means that the approach used for the `table_try_single` instruction to efficiently deal with deterministic tabled calls can be applied to the `table_trust_me` and `table_trust` instructions. We discuss the implementation details for these instructions in the next section.

4 Implementation Details

In this section, we describe in detail how we have extended YapTab to provide engine support to efficiently deal with deterministic tabled calls.

4.1 Generator Nodes

In YapTab, a generator node is implemented as a WAM choice point extended with some extra fields. The format of a generic generator choice point of YapTab is depicted in Figure 3. Fields that are not found in standard WAM choice points are coloured gray. A generator choice point is divided in three sections. The top section contains the usual WAM fields needed to restore the computation on backtracking plus two extra fields [12]: `cp_dep_fr` is a pointer to the corresponding dependency frame, used by local scheduling for fixpoint check, and `cp_sg_fr` is a pointer to the associated subgoal frame where answers should be stored. The middle section contains the argument registers of the subgoal and the bottom section contains the *substitution factor*, i.e., the set of free variables which exist in the terms in the argument registers. The substitution factor is an optimization that allows the new answer operation to store in the table space only the substitutions for the free variables in the subgoal call [18].

If we now turn our attention to how generator choice points are handled during evaluation, we find that some of this information is never used when evaluating deterministic tabled calls with batched scheduling. This happens mainly

cp_b	Failure continuation CP
cp_ap	Next unexploit alternative
cp_tr	Top of trail
cp_cp	Success continuation PC
cp_h	Top of global stack
cp_env	Current Environment
cp_dep_fr	Dependency frame
cp_sg_fr	Subgoal frame

An	Argument Register n
⋮	⋮
A1	Argument Register 1

m	Number of Substitution Vars
Vm	Substitution Variable m
⋮	⋮
V1	Substitution Variable 1

Fig. 3. Format of a generic generator choice point in YapTab

because, with batched scheduling, the computation is never resumed in a deterministic generator choice point. This allow us to remove the argument registers and the standard `cp_cp`, `cp_h` and `cp_env` fields. The `cp_dep_fr` field can also be removed because it is only necessary with local scheduling [12], which is never the case. Figure 4 shows the new format of YapTab's deterministic generator choice point with the strictly necessary fields.

The `cp_b` field is needed for failure continuation; the `cp_ap` and `cp_tr` are required when backtracking to the choice point; the `cp_sg_fr` is required by the new answer and completion operations; and the substitution factor fields are required by the new answer operation. In order to avoid extra overheads when manipulating the different kinds of choice points that can coexist in an evaluation, we have rearranged all kinds of choice points in such a way that the top three fields are now the same as the ones for a deterministic generator choice point: the `cp_b`, `cp_ap` and `cp_tr` fields.

The memory reduction obtained with the new representation for deterministic generator choice points increases when the number of argument registers (the arity of the predicate being called) and the number of substitution variables are, respectively, bigger and smaller. Considering that A is the number of arguments registers and that S is the number of substitution variables, the percentage of memory saved with the new representation can be expressed as follows:

cp_b	Failure continuation CP
cp_ap	Next unexploit alternative
cp_tr	Top of trail
cp_sg_fr	Subgoal frame

m	Number of Substitution Vars
Vm	Substitution Variable m
⋮	⋮
V1	Substitution Variable 1

Fig. 4. Format of a deterministic generator choice point in YapTab

$$1 - \frac{4 + 1 + S}{8 + A + 1 + S}$$

4.2 Tabling Operations

In order to deal with the new representation for deterministic generator choice points, this required small changes to the tabled subgoal call, new answer and completion operations. Figures 5, 6, 7 and 8 show in more detail the changes (blocks of code marked with comment '`// new`') made to the `table_try_single`, `table_trust_me`[2], `new_answer` and `completion` instructions. Figure 9 shows the pseudo-code for the auxiliary procedure `is_deterministic_generator_cp()`. We assume that memory addresses grow downwards and that the choice point stack grows upwards.

```
table_try_single(TABLED_CALL tc) {
  sg_fr = subgoal_check_insert(tc)   // sg_fr is the subgoal frame for tc
  if (new_tabled_subgoal_call(sg_fr)) {
    if (evaluation_mode(tc) == batched_scheduling)                // new
      store_deterministic_generator_node(sg_fr)
    else                                                 // local scheduling
      store_generic_generator_node(sg_fr)
    ...
    goto next_instruction()
  }
  ...
}
```

Fig. 5. Pseudo-code for the `table_try_single` instruction

[2] The changes made to the `table_trust` instruction are identical to the ones made to the `table_trust_me` instruction.

The table_try_single instruction now tests whenever the subgoal being called is to be evaluated using batched or local scheduling. If batched, it allocates a deterministic generator choice point. If local, it proceeds as before and allocates a generic generator choice point.

```
table_trust_me(TABLED_CALL tc) {
  // the B register points to the current choice point
  restore_generic_generator_node(B, COMPLETION)
  if (evaluation_mode(tc) == batched_scheduling &&
      not_in_a_frozen_segment(B) {                                  // new
    subs_factor = B + sizeof(generic_generator_cp) + arity(tc)
    gen_cp = subs_factor - sizeof(deterministic_generator_cp)
    gen_cp->cp_sg_fr = B->cp_sg_fr
    gen_cp->cp_tr = B->cp_tr
    gen_cp->cp_ap = B->cp_ap
    gen_cp->cp_b  = B->cp_b
    B = gen_cp
  }
  ...
}
```

Fig. 6. Pseudo-code for the table_trust_me instruction

The table_trust_me instruction now tests if the current tabled call is being evaluated using batched scheduling and if the current choice point is not in a frozen segment[3]. If these two conditions hold, we can recover some memory space by transforming the current generator choice point into a deterministic generator choice point. To do that, we need to copy the cp_sg_fr, cp_tr, cp_ap and cp_b fields in the current choice point to their new position, just above the substitution factor variables.

```
new_answer(TABLED_CALL tc, ANSWER ans) {
  if (is_deterministic_generator_cp(B)) {                           // new
    gen_cp = deterministic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
    subs_factor = gen_cp + sizeof(deterministic_generator_cp)
  } else {                              // generic generator choice point
    gen_cp = generic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
    subs_factor = gen_cp + sizeof(generic_generator_cp) + arity(tc)
  }
  ...
}
```

Fig. 7. Pseudo-code for the new_answer instruction

[3] The YapTab system uses frozen segments to protect the stacks of suspended computations [4]. Thus, if the current choice point is trapped in a frozen segment it is worthless to try to recover memory from it using our approach.

```
completion() {
  ...                                                  // fixpoint check loop
  // subgoal completely evaluated
  if (is_deterministic_generator_cp(B)) {                          // new
    gen_cp = deterministic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
  } else {
    gen_cp = generic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
  }
  complete_subgoal(sg_fr)
  ...
}
```

Fig. 8. Pseudo-code for the `completion` instruction

For the new answer and completion operations, since both generator types have different sizes, we need a way to correctly identify which is the type of the generator in order to correctly access the required fields on each structure. To do that, we use the `is_deterministic_generator_cp()` auxiliary procedure to test if a generator choice point is deterministic or not. Figure 9 shows the pseudo-code for it.

The `is_deterministic_generator_cp()` procedure assumes that, by default, we have a generic generator choice point and we check if the `cp_h` field (which is aligned with the field representing the number of substitution variables in a deterministic generator choice point) is less than the maximum number of allowed substitution variables (`MAX_SUBSTITUTION_VARS`). If this is case, then we know that we have a deterministic generator choice point.

```
is_deterministic_generator_cp(CHOICE_POINT cp) {
  gen_cp = generic_generator_cp(cp)
  if (gen_cp->cp_h <= MAX_SUBSTITUTION_VARS)
    return TRUE
  else
    return FALSE
}
```

Fig. 9. Pseudo-code for the `is_deterministic_generator_cp()` procedure

5 Preliminary Experimental Results

We next present some preliminary experimental results comparing YapTab with and without support for deterministic tabled calls. The environment for our experiments was a AMD Athlon(tm) 64 Processor 3200+ processor with 2 GByte of main memory and running the Linux kernel 2.6.24-19 with YapTab 5.1.3.

To evaluate the impact of our proposal, first we have defined three deterministic tabled predicates, respectively with arities 5, 11 and 17, that simply call themselves recursively:

```
:- table t/5, t/11, t/17.

t(N,A2,A3,A4,A5) :-
  N > 0, N1 is N - 1,
  t(N1,A2,A3,A4,A5).

t(N,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11) :-
  N > 0, N1 is N - 1,
  t(N1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11).

t(N,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17) :-
  N > 0, N1 is N - 1,
  t(N1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17).
```

The first argument `N` controls the number of times the predicate is executed. It thus defines the number of generator choice points to be allocated (we used a value of 100,000 in our experiments). In order to have specific combinations of argument registers and substitution variables, we have ran each predicate with three different sets of free variables in the arguments:

```
:- t(100000,A2,A3,A4,A5).
:- t(100000,A2,A3,0,0).
:- t(100000,0,0,0,0).

:- t(100000,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11).
:- t(100000,A2,A3,A4,A5,A6,0,0,0,0,0).
:- t(100000,0,0,0,0,0,0,0,0,0,0).

:- t(100000,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17).
:- t(100000,A2,A3,A4,A5,A6,A7,A8,A9,0,0,0,0,0,0,0,0).
:- t(100000,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0).
```

These experiments are a kind of best-case scenario as they only allocate generator choice points and they do not store permanent variables for *environment frames* [16]. Table 1 shows the memory usage, in KBytes, for the local stack[4] and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls. A third column ***Ratio (1–b/a)*** shows the memory and running time ratio between both approaches. For the memory ratio, we show in parentheses the percentage of memory saved if using the formula presented at the end of section 4.1.

The results in Table 1 indicate that YapTab with support for deterministic tabled calls can decrease, on average, memory usage by 48% and running time by 23%. These results also confirm that memory reduction increases when the number of argument registers is bigger and the number of substitution variables is smaller. This is coherent with the formula presented in section 4.1. The small difference between our experiments and the values obtained when using the formula came from the fact that, in the formula, we are considering a local stack without environment frames.

[4] In YapTab, the local stack contains both choice points and environment frames. Other systems, like XSB Prolog, have separate choice point and environment stacks.

Args	***Subs***	***YapTab (a)***		***YapTab+Det (b)***		***Ratio (1–b/a)***	
		Memory	***Time***	***Memory***	***Time***	***Memory***	***Time***
5	4	9,376	82	5,860	70	0.37 (0.50)	0.15
5	2	8,594	78	5,079	66	0.41 (0.57)	0.15
5	0	7,813	80	4,297	65	0.45 (0.64)	0.19
11	10	14,063	137	8,204	96	0.42 (0.50)	0.30
11	5	12,110	136	6,251	89	0.48 (0.60)	0.35
11	0	10,157	124	4,297	108	0.58 (0.75)	0.13
17	16	18,751	173	10,547	129	0.44 (0.50)	0.25
17	8	15,626	164	7,422	109	0.53 (0.62)	0.34
17	0	12,501	153	4,297	114	0.66 (0.81)	0.25
Average						0.48 (0.61)	0.23

Table 1. Memory usage (in KBytes) and running times (in milliseconds) for YapTab without and with the new support for deterministic tabled calls

Next, we tested our approach with the *sequence comparisons* problem [19]. In this problem, we have two sequences A and B, and we want to determine the minimal number of operations needed to turn A into B. We used the original tabled program from [19] and a transformed tabled program that forces all calls to use the `table_try_single` instruction. We experimented these two versions with sequences of length 500, 1000, 1500 and 2000. Table 2 shows the memory usage, in KBytes, for the local stack and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls. A third column ***Ratio (1–b/a)*** shows the memory and running time ratio between both approaches.

Version	***Length***	***YapTab (a)***		***YapTab+Det (b)***		***Ratio (1–b/a)***	
		Memory	***Time***	***Memory***	***Time***	***Memory***	***Time***
Original	500	51,774	1,548	44,938	1,264	0.13	0.18
	1000	207,063	13,548	179,719	11,212	0.13	0.17
	1500	465,868	60,475	404,344	50,631	0.13	0.16
	2000	828,188	189,647	718,813	157,213	0.13	0.17
Transformed	500	45,915	1,172	39,051	848	0.15	0.28
	1000	183,625	10,024	156,227	8,460	0.15	0.16
	1500	413,133	45,874	351,528	36,106	0.15	0.21
	2000	734,438	140,068	624,953	113,011	0.15	0.19
Average						0.14	0.19

Table 2. Memory usage (in KBytes) and running times (in milliseconds) for YapTab without and with the new support for deterministic tabled calls

In general, for memory usage, the results in Table 2 are slightly different from the previous results obtained in Table 1. For both version of the *sequence comparisons* program, YapTab with support for deterministic tabled calls can

decrease, on average, memory usage by 14%. This reduction on memory saving, compared with the results on Table 1, happens mainly because of the existence of permanent variables in the body of the clauses in the *sequence comparisons* program. On the other hand, for the running times, the results in Table 2 confirm the previous results obtained in Table 1.

The results in Table 2 also show very similar memory and running time ratios for both versions of the *sequence comparisons* program. This suggests that we can take advantage of our approach by using the last matching clause optimization and not only when a program contains deterministic tabled predicates.

Finally, we tested our approach with a *path* program that computes the transitive closure of a NxN grid using a right recursive algorithm:

```
:- table path/2.

path(X,Z) :- edge(X,Z).
path(X,Z) :- edge(X,Y), path(Y,Z).
```

Regarding the `edge/2` facts, we used four grid configuration with 30x30, 40x40, 50x50 and 60x60 nodes. Table 3 shows the memory usage, in KBytes, for the local stack and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls. Again, a third column ***Ratio (1–b/a)*** shows the memory and running time ratio between both approaches.

Grid	***YapTab (a)***		***YapTab+Det (b)***		***Ratio (1–b/a)***	
	Memory	***Time***	***Memory***	***Time***	***Memory***	***Time***
30x30	119	1,304	98	1,464	0.18	-0.12
40x40	211	4,400	175	4,024	0.17	0.09
50x50	330	11,208	273	10,996	0.17	0.02
60x60	476	28,509	393	28,213	0.17	0.01
Average					0.17	0.00

Table 3. Memory usage (in KBytes) and running times (in milliseconds) for YapTab without and with the new support for deterministic tabled calls

The *path* program confirms tendency to memory reduction, this case in 17%, on average. Running time gets sightly worse, thought comparison between both approaches remains in positive territory in three cases. Note however, that our approach was mainly designed to achieve a reduction on memory usage by paying a small cost on running time due to the extra code needed to deal with the new data structures and algorithms. Despite of this fact, on average, our approach showed a very good performance in all experiments.

6 Conclusions and Further Work

We have presented a proposal for the efficient evaluation of deterministic tabled calls with batched scheduling. A well-known aspect of tabling is the overhead in terms of memory usage compared with standard Prolog. This raised us the question of whether it was possible to minimize this overhead when evaluating deterministic tabled computations. Our preliminary results are quite promising, they suggest that, for deterministic tabled calls with batched scheduling, it is possible not only to reduce the memory usage overhead, but also the running time of the evaluation for certain class of applications.

Further work will include exploring the impact of applying our proposal to more complex problems, seeking real-world experimental results allowing us to improve and expand our current implementation.

Acknowledgements

This work has been partially supported by the research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/ EIA/66924/2006) and by Fundação para a Ciência e Tecnologia.

References

1. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
2. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
3. Rao, P., Sagonas, K., Swift, T., Warren, D.S., Freire, J.: XSB: A System for Efficiently Computing Well-Founded Semantics. In: International Conference on Logic Programming and Non-Monotonic Reasoning. Number 1265 in LNCS, Springer-Verlag (1997) 431–441
4. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming **5**(1 & 2) (2005) 161–205
5. Zhou, N.F., Sato, T., Shen, Y.D.: Linear Tabling Strategies and Optimizations. Theory and Practice of Logic Programming **8**(1) (2008) 81–109
6. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 181–196
7. Somogyi, Z., Sagonas, K.: Tabling in Mercury: Design and Implementation. In: International Symposium on Practical Aspects of Declarative Languages. Number 3819 in LNCS, Springer-Verlag (2006) 150–167
8. Chico, P., Carro, M., Hermenegildo, M.V., Silva, C., Rocha, R.: An Improved Continuation Call-Based Implementation of Tabling. In: International Symposium on Practical Aspects of Declarative Languages. Number 4902 in LNCS, Springer-Verlag (2008) 197–213

9. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20**(3) (1998) 586–634
10. Sagonas, K., Stuckey, P.: Just Enough Tabling. In: ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM Press (2004) 78–89
11. Saha, D., Ramakrishnan, C.R.: Incremental Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 235–249
12. Rocha, R., Silva, F., Santos Costa, V.: Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 250–264
13. Rocha, R.: On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In: International Symposium on Practical Aspects of Declarative Languages. Number 4354 in LNCS, Springer-Verlag (2007) 155–169
14. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: International Symposium on Programming Language Implementation and Logic Programming. Number 1140 in LNCS, Springer-Verlag (1996) 243–258
15. Santos Costa, V., Damas, L., Reis, R., Azevedo, R.: YAP User's Manual. Available from `http://www.dcc.fc.up.pt/~vsc/Yap`.
16. Aït-Kaci, H.: Warren's Abstract Machine – A Tutorial Reconstruction. The MIT Press (1991)
17. Santos Costa, V., Sagonas, K., Lopes, R.: Demand-Driven Indexing of Prolog Clauses. In: International Conference on Logic Programming. Number 4670 in LNCS, Springer-Verlag (2007) 395–409
18. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (1999) 31–54
19. Warren, D.S.: Programming in Tabled Prolog. Technical report, Department of Computer Science, State University of New York (1999) Available from `http://www.cs.sunysb.edu/~warren/xsbbook/book.html`

A Program Transformation for Continuation Call-Based Tabled Execution

Pablo Chico de Guzman[1] Manuel Carro[1] Manuel V. Hermenegildo[1,2]
pchico@clip.dia.fi.upm.es {mcarro,herme}@fi.upm.es

[1] School of Computer Science, Univ. Politecnica de Madrid, Spain
[2] IMDEA Software, Spain

Abstract. The advantages of tabled evaluation regarding program termination and reduction of complexity are well known —as are the significant implementation, portability, and maintenance efforts that some proposals (especially those based on suspension) require. This implementation effort is reduced by program transformation-based continuation call techniques, at some efficiency cost. However, the traditional formulation of this proposal by Ramesh and Cheng limits the interleaving of tabled and non-tabled predicates and thus cannot be used as-is for arbitrary programs. In this paper we present a complete translation for the continuation call technique which, using the runtime support needed for the traditional proposal, solves these problems and makes it possible to execute arbitrary tabled programs. We present performance results which show that CCall offers a useful tradeoff that can be competitive with state-of-the-art implementations.
Keywords: Tabled logic programming, Continuation-call tabling, Implementation, Performance, Program transformation.

1 Introduction

Tabling [18, 19, 4] is a strategy for executing logic programs which uses *memoization* of already processed calls and their answers to improve several of the limitations of SLD resolution. It brings termination for bounded term-size programs and improves efficiency in programs which perform repeated computations and has been successfully applied to deductive databases [14], program analysis [20, 5], reasoning in the semantic Web [23], model checking [13], etc.

However, tabling also has certain drawbacks, including that predicates to be tabled have to be selected carefully[3] in order not to incur in undesired slowdowns and, specially relevant to our discussion, that its efficient implementation is generally complex. In *suspension-based tabling* the computation state of suspended tabled subgoals has to be preserved to avoid backtracking over them. This is done either by *freezing* the stacks, as in XSB [17], by copying to another

[3] XSB includes an `auto_table` declaration which triggers a conservative analysis to detect which predicates are to be tabled in order to ensure termination. However, more predicates than needed can be selected.

area, as in CAT [8], or by using an intermediate solution as in CHAT [9]. *Linear tabling* maintains instead a single execution tree without requiring suspension and resumption of sub-computations. The computation of the (local) fixpoint is performed by making subgoals "loop" in their alternatives until no more solutions are found. This may make some computations to be repeated. Examples of this method are the linear tabling of B-Prolog [22, 21] and the DRA scheme [10]. Suspension-based mechanisms achieve very good performance but, in general, require deeper changes to the underlying implementation. Linear mechanisms, on the other hand, can usually be implemented on top of existing sequential engines without major modifications.

The Continuation Call (`CCall`) approach to tabling [15, 16] tries to combine the best of both worlds: it is a reasonably efficient suspension-based mechanism which requires relatively simple additions to the Prolog implementation / compiler,[4] thus making maintenance and porting much easier. In [6] we proposed a number of optimizations to the `CCall` approach and showed that with such optimizations performance could be competitive with traditional implementations. However, this was only partially satisfactory since the `CCall` tabling approach is restricted to programs with a certain interleaving of tabled and non-tabled predicate calls (see Figure 3 and Section 3.1), and thus cannot execute general tabled programs.

In this paper we present an extension of the `CCall` translation which, using the same runtime support of the traditional proposal, overcomes the problems pointed out above. In Section 5 we present a complexity comparison of the proposed approach with CHAT. Finally, we present performance results from our implementation. These results show that our approach offers a useful tradeoff which can be competitive with state of the art implementations, while keeping implementation efforts relatively low.

2 The Continuation Call Technique

We sketch now how tabled evaluation [4, 17] works from a user point of view and we briefly describe the Continuation Call technique, on which we base our work.

2.1 Tabling Basics

We will use as example the program in Figure 1, whose purpose is to determine the reachability of nodes in a graph. If the graph contains cycles, there will be queries which will make the program loop forever under the standard SLD resolution strategy, regardless of the order of the clauses. Tabling changes the operational semantics for predicates marked with the `:- table` declaration, which forces the compiler and runtime system to distinguish the first occurrence of a tabled goal (the *generator*) and subsequent calls which are identical up to variable renaming (the *consumers*). The generator applies resolution using the

[4] As an example, no modification to the underlying engine is needed.

program clauses to derive answers for the goal. Consumers *suspend* the current execution path (using implementation-dependent means) and start execution on a different branch corresponding to another clause of the predicate within which the execution was suspended. When such an alternative branch finally succeeds, the answer generated for the initial query (the generator) is inserted in a table associated with that generator. This makes it possible to reactivate consumers and to continue execution at the point where they were stopped. Thus, consumers do not use SLD resolution, but obtain instead the answers from the table where they were previously inserted by the generator. Predicates not marked as tabled are executed according to SLD resolution, hopefully with minimal overhead due to the availability of tabling. This can be graphically seen as the ability to suspend execution in a part of the tree which cannot progress (because it enters a loop) and continue it somewhere else, where a solution for the looping goal can be produced.

2.2 `CCall` by Example

`CCall` implements tabling by a combination of program transformation and side effects in the form of insertions into and retrievals from a table which relates calls, answers, and the continuation code to be executed after consumers read answers from the table. We will now sketch how the mechanism works using the `path/2` example (Figure 1). The original code is transformed into the program in Figure 2 which is the one actually executed.

Roughly speaking, the transformation for tabling is as follows: an auxiliary predicate (`slg_path/2`) for `path/2` is introduced so that calls to `path/2` made from regular (SLD) Prolog execution do not need to be aware of the fact that `path/2` is being tabled. The primitive `slg/1` will make sure that its argument is executed to completion and will return, on backtracking, all the solutions found for the tabled predicate. To this end, `slg/1` checks if the call has already been executed. If so, all its answers are returned by backtracking. Otherwise, control is passed to a new predicate (`slg_path/2` in this case).[5] `slg_path/2` receives in its first argument the original call to `path/2` and in the second argument the identifier of its generator, which is used to relate operations on the table with this initial call. Each clause of `slg_path/2` is derived from a clause of the original `path/2` predicate by:

- Adding an `answer/2` primitive at the end of each clause of the original tabled predicate. `answer/2` is responsible for inserting answers in the table after checking for redundancy.
- Instrumenting calls to tabled predicates using the `slgcall/1` primitive. If this tabled call is a consumer, `path_cont/3`, along with its arguments, is recorded as (one of) the continuation(s) of its generator. If the tabled call is a generator, it is associated with a new call identifier and execution follows using the `slg_path/2` program clauses to derive new answers (as done

[5] The unique name has been created for simplicity by prepending `slg_` to the predicate name –any safe means of constructing a unique predicate symbol can be used.

```
:- table path/2.

path(X, Z):-
    edge(X, Y),
    path(Y, Z).
path(X, Z):-
    edge(X, Z).
```

Fig. 1. A sample program.

```
path(X, Y):- slg(path(X, Y)).
slg_path(path(X, Y), Id):-
    edge(X, Y),
     slgcall (path_cont(Id, [X], path(Y, Z))).
slg_path(path(X, Y), Id):-
    edge(X, Y),
    answer(Id, path(X, Y)).

path_cont(Id, [X], path(Y, Z)):-
    answer(Id, path(X, Z)).
```

Fig. 2. The program in Figure 1 after being transformed for tabled execution.

by `slg/1`). Besides, `path_cont/3` will be recorded as a continuation of the generator identified by `Id` if the tabled call cannot be completed (there were dependencies on previous generators). The `path_cont/3` continuation will be called consuming found answers or erased upon completion of its generator.
– Encoding the remaining of the clause body of `path/2` after the recursive call by using `path_cont/3`. It is constructed similarly to `slg_path/2`, i.e., applying the same transformation as for the initial clauses and calling `slgcall/1`.

The second argument of `path_cont/3` is a list of bindings needed to recover the environment of the continuation call. Note that, in the program in Figure 1, an answer to a query such as `?- path(X, Y)` may need to bind variable `X`. This variable does not appear in the recursive call to `path/2`, and hence it does not appear in the `path/2` term passed on to `slgcall/1` either. In order for the body of `path_cont/3` to insert in the table the answer corresponding to the initial query, variable `X` (and, in general, any other necessary variable) has to be passed down to `answer/2`. This is done with the list `[X]`, which is inserted in the table as well and completes the environment needed for the continuation `path_cont/3` to resume the previously suspended call.

A safe approximation of the variables which should appear in this list is the set of variables which appear in the clause before the tabled goal and which are used in the continuation, including the `answer/2` primitive. Variables appearing in the tabled call itself do not need to be included, as they will be passed along anyway. This list of bindings corresponds to the frame of the parent call if the `answer/2` primitive is added to the end of the body being translated. More details about `CCall` approach and their primitives can be found at [15].

Key Contribution of CCall: a new predicate name is created for all points where suspension can happen. Suspension is performed by saving this predicate name, a list of bindings, and a generator identifier. Resumption is performed by constructing a Prolog goal with the information saved on suspension plus the answer which raised the resumption. It is clear that this is significantly simpler

```
:- table t/1.

t(A):-
    p(B),
    A is B + 1.
t(0).

p(B):- t(B), B < 1.
```

Fig. 3. A program for which the original `CCall` transformation fails.

```
t(A):- slg(t(A)).
slg_t(t(A), Id):-
    p(B), A is B + 1,
    answer(Id, t(A)).

slg_t(t(0), Id):-
    answer(Id, t(0)).

p(B):- t(B), B < 1.
```

Fig. 4. The program in Figure 3 after being transformed for tabled execution.

to implement than other approaches as XSB or CHAT, where changes in the abstract machine have to be introduced. Consequently, porting and maintainability are simpler too, since `CCall` is independent of the compiler and how to create a Prolog term on the heap is the only one low level operation to implement.

3 Mixing Tabled and Non-Tabled Predicates

A continuation is the way `CCall` tabling preserves both the environment and the code of a consumer to be resumed. The list of bindings contains the same variables as the frame of the predicate where the `slgcall/1` primitive is executed, taking into account the `answer/2` primitive added at the end of the clause. However, the `CCall` approach to tabling, as originally proposed, has a problem when Prolog predicates appear between generators and consumers: the environments created by the non-tabled predicates are not taken into account, and they may be needed to correctly suspend and resume tabled predicates, as the example in the following section shows.

3.1 An Ill-Behaved Transformation

Figure 3 shows an example of a tabled program, where tabled and non-tabled execution (`t/1` and `p/1`) are mixed. The translation of the program is shown in Figure 4, taking into account the rules in Section 2.2.

The execution of the program with the query `t(A)` is shown in Figure 5. The execution is correct until `slg/1` is called again by `p/1`. At that point execution should suspend (and later resume), but `slg/1` does not have any associated continuation, and it does not have any pointer to the code to be executed on resumption (partially in `p/1` and partially in `slg_t/2`): `B < 1, A is B + 1, answer(Id,t(A))` is lost on backtracking and it is not reachable when resuming. Consequently, the second answer to the query, `t(1)`, is lost.

The call to `t(B)` made by `p(B)` could have been translated as if it were in the body of a tabled clause, but in that case the piece of code `A is B + 1` in

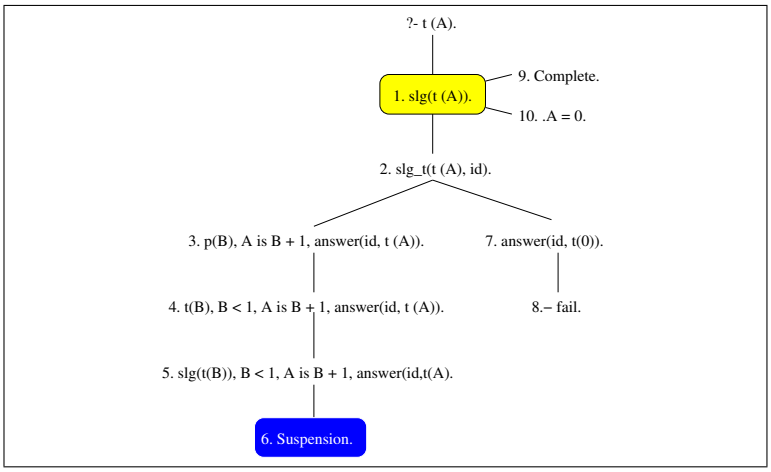


Fig. 5. Tabling execution of example of Figure 1.

the first clause of `t/1` would be lost anyway. This is an example of why all the frames between a consumer and its nearest generator have to be saved when suspending, and it is not enough to save just the last one, as in the original `CCall` proposal [15], which does work, however, when all the calls to the tabled predicates appear in the body of the clause of a tabled predicate. In that case, it is enough to save the last frame with the associated continuation code. Note that all the suspension-based tabling approaches preserve the frames / environments from the consumer until the corresponding generator.

To solve this problem, we have extended the translation to take into account a new kind of predicates, named *bridges*. A bridge predicate is a non-tabled Prolog predicate whose clauses generate frames which have to be saved in the continuation of a consumer. In the example of Figure 3, `p/1` is a bridge predicate.

3.2 Marking Predicates as Bridges

Bridge predicates are all the non-tabled predicates which can appear in the execution tree of a query between a generator and each of its consumers, i.e., the predicates whose environments are in the local stack between the environment of the generator and the environment of each of its consumers. Note that tabled predicates do not need to be included as bridge predicates as their environment will be already saved by the translation. Additionally, only recursive calls which can lead to infinite loops under SLD resolution have to actually be taken into account, because these are the only ones which can suspend and later be resumed. Programs for which tabling merely speeds up already terminating computations are not subject to the problem outlined above, and therefore do not benefit from the improved translation shown herein.

Thus, in order to determine a minimal set of bridge predicates, B_{min}, we need to determine before the minimum set of tabled predicates, T_{min}, which ensures

Make a graph G with an edge $(p1/n1,\ p2/n2) \Leftrightarrow p2/n2$ **is** called from $p1/n1$
$Bridges = \emptyset$
FOR each predicate T in TABLED PREDICATES
  $Forward =$ All predicates reached from T in G
  $Backward =$ All predicates from which T **is** reached in G
  $Bridges = Bridges \cup (Forward \cap Backward)$
$Bridges = Bridges -$ TABLED PREDICATES

Fig. 6. Safe approximation to look for bridge predicates.

termination. When T_{min} is found, B_{min} is the set of non-tabled predicates which are "in the middle" of two calls to predicates belonging to T_{min}. Since looking for T_{min} is undecidable (because it implies detecting infinite failures), looking for B_{min} is also undecidable and a *safe approximation*, which may mark as bridge some predicates which do not need to be, is needed.

As we will see in Section 4.2, the only disadvantage of such an over-approximation is that some code will be duplicated (to accept a new argument for the case where a bridge predicate is called from a tabled execution), and that bridge predicates, having an extra argument, can be called when this is not needed. The algorithm we have implemented (Figure 6) only looks for tabled predicates which can recursively call themselves. For the examples used for performance evaluation in Section 6, using the safe approximation algorithm produces an average slowdown of only 3% with respect to a perfect characterization of bridge predicates.

4 A General Translation for Tabled Programs

In this section we present program transformation rules which take into account bridge predicates. This transformation assumes that the safe approximation algorithm for bridge predicates has already been run, and all the bridge predicates have been marked by adding a `:- bridge P/N` declaration in the program.

As seen in Section 2.2, a continuation is the way to save an environment, because the predicate name is the same as the PC counter of the environment and the list of bindings is the same as the variables that a environment saves. Consequently, the goal of the new translation is to associate a continuation with each of the bridge predicates to save their associated environment. These continuations receive a new argument (the continuation to be executed) which is used to push a pointer (i.e., the name of a predicate) to the code to continue with, in a way similar to environments in local stacks.

4.1 Translation Rules

The rules for the original translation have three different goals: to maintain the interface with the rest of the code, to manage tabled calls which appear in the body of the clauses of a tabled predicate, and to insert answers at the end of

```
trans(C, C) :- \+ table(C), \+ bridge(C).
trans(( :- table P/N ), ( P(X1..Xn) :- slg(P(X1..Xn)) )).
trans(( Head :- Body ), LC) :-
    table(Head),
    Head_tr =.. ['slg_' ∘ Head, Head, Id],
    End = answer(Id, Head),
    transBody(Head_tr, Body, Id, [], End, LC).
trans(( Head :- Body ), ( Head :- Body ) ∘ LC) :-
    bridge(Head),
    Head_tr =.. [Head ∘ '_bridge', Head, Id, Cont],
    End = call(Cont),
    transBody(Head_tr, Body, Id, Cont, End, LC).

transBody([], [], _, _, [], []).
transBody(Head, Body, Id, ContPrev, End, ( Head :- Body_tr ) ∘ RestBody_tr) :-
    following(Body, Pref, Pred, Suff),
    getLBinds(Pref, Suff, LBinds),
    updateBody(Pred, End, Id, Pref, LBinds, ContPrev, Cont, Body_tr),
    transBody(Cont, Suff, Id, ContPrev, End, RestBody_tr).

following(Body, Pref, Pred, Suff) :-
    member(Body, Pred),
    (table(Pred); bridge(Pred)), !,
    Body = Pref ∘ Pred ∘ Suff.

updateBody([], End, _Id, Pref, _LBinds, _ContPrev, [], Pref ∘ End).
updateBody(Pred, _End, Id, Pref, LBinds, ContPrev, Cont, Pref ∘ slgcall(Cont)) :-
    table(Pred),
    getNameCont(NameCont),
    Cont = NameCont(Id, LBinds, Pred, ContPrev).
updateBody(Pred, _End, Id, Pref, LBinds, ContPrev, Cont, Pref ∘ Bridge_call) :-
    bridge(Pred),
    getNameCont(NameCont),
    Cont = NameCont(Id, LBinds, Pred, ContPrev),
    Bridge_call =.. [Pred ∘ '_bridge', Pred, Id, Cont] .
```

Fig. 7. The Prolog code of the translation rules.

the evaluation of each clause. The same points have to be addressed for bridge clauses, taking into account that a tabled or bridge call has to be translated if it appears in the body of a tabled predicate or a bridge predicate.

The rules for the new translation, which uses the same primitives as the original `CCall` proposal, are shown in Figure 7, where for conciseness we have used a sugared Prolog-like language. For example, a functional syntax is implicitly assumed where needed, and infix '∘' is a general `append` function which joins either (linear) structures or, when applied to atoms, concatenates them. It may appear in an output head position with the expected semantics.

The `trans/2` predicate receives a clause to be translated and returns the list of clauses resulting from the translation. Its first clause ensures that predicates which are non-tabled and non-bridge are not transformed.[6] The second one is to generate the interface of table predicates with the rest of the code: if there is a tabled declaration, the interface is generated. The third clause translates clauses of tabled predicates, and the fourth one translates clauses of bridge predicates, where the original one is maintained in case it is called outside a tabled call (this is in order to preserve the interface with non-tabled code). They generate the new head of the clause, `Head_tr`, and the code which has to be appended at the end of the body, `End`, before calling `transBody/6` with these arguments. `End` can be the `answers/2` primitive for tabled clauses or `call(Cont)`, which invokes the following pushed continuation, stored in the fourth argument.

`transBody/6` generates, in its last argument, the translation of the body of a clause by taking care, in each iteration, of the code until the next tabled or bridge call, or until the end the clause, and appending the translation of the rest of the clause to this partial translation. In other words, it calls `updateBody/8` to translate tabled or bridge calls and continues translating the rest of the body.

The `following/4` splits a clause body in three parts: a prefix, until the first time a tabled or bridge call appears, the tabled or bridge call itself, and a suffix from this call until the end of the clause. `getLBinds/3` obtains the list of variables which have to be saved to recover the environment of the consumer, based on the ideas of Section 2.2.

The `updateBody/8` predicate completes the body prefix until the next tabled or bridge call. Its first six arguments are inputs, the seventh one is the head of the continuation for the suffix of the body, and the last argument is the new translation for the prefix. The first clause takes care of the base case, when there are no calls to bridge or tabled predicates left, the second clause generates code for a call to a tabled predicate, and the last one does the same with a bridge predicate. That `getNameCont/1` generates a unique name for the continuation.

We will now use the example in Figure 3, adding a `:- bridge p/1` declaration, to exemplify how a translation would take place.

4.2 The Previous Example with the Correct Transformation

The translation of the first clause of `t/1` is done by the third clause of `trans/2`, which makes the head of the translated clause to be `slg_t(t(A), Id)` and states that the final call of that clause has to be `answer(Id, t(A))` —i.e., when the clause successfully finishes, it adds the answer to the table.

`transBody/6` takes care then of the rest of the body, which identifies which environment variables (`A`, in this case) have to be saved and matches `Pref`, `Pred`, and `Suff` with the goals before the call to the bridge predicate (none —

[6] The predicates `table/1` and `bridge/1` are dynamically generated by the compiler from the corresponding declaration. They check if their argument is a clause of a tabled or bridge predicate, or if their argument is a functor corresponding to a tabled or bridge predicate, respectively.

```
t(A) :- slg(t(A)).
slg_t(t(A), Id) :-
    p_bridge(p(B), Id, slg_t0(Id, [A], p(B), [])).

slg_t(t(0), Id) :- answer(Id, t(0)).

slg_t0(Id, [A], p(B), []) :-
    A is B + 1,
    answer(Id, t(A)).

p(B) :- t(B), B < 1.
p_bridge(p(B), Id, Cont) :-
    slgcall(p_bridge0(Id, [], t(B), Cont)).

p_bridge0(Id, [], t(B), Cont) :-
    B < 1,
    call(Cont).
```

Fig. 8. The program in Figure 3 after being transformed for tabled execution.

and empty conjunction), the call to the bridge predicate (`p(B)`), and the goals after this call (`A is B + 1`). The third clause of `updateBody/8` generates the body of `Head_tr`, to give the first clause of `slg_t/2`. A continuation is generated for the rest of the body; the code of the continuation is a predicate whose head is `slg_t0/3` and its body is generated by the first clause of `updateBody/8`.

The translation of the second clause of `t/1` is simpler, as it only has to add `answer(Id, t(0))` at the end of the body of the new predicate.

The clause for `p/1` is kept to maintain its interface when it is not called from inside a another tabled execution. The translation for the clause of `p/1` is made by the fourth clause of `trans/2` where `Head_tr` is unified with `p_bridge(p(B), Id, Cont)`. `End` is unified with `call(Cont)` — a call to the continuation code to be resumed by the following pushed continuation. `transBody/6` finds an empty list of environment variables and unifies `Pref`, `Pred` and `Suff` with `[]`, `t(B)` and `B < 1`, respectively. The second clause of `updateBody/8` generates the body for the new predicate `p_bridge/3`. A continuation is generated to execute the rest of the body, whose head is `p_bridge0/3` and whose body is generated by the first clause of `updateBody/8`. As we can see, bridge predicates are pushing continuations which are sequentially called when consumers are resumed.

4.3 Execution of the Transformed Program

The execution tree of the transformed program is shown in Figure 9. It is similar to that in Figure 5, but a continuation `slg_t0(id, [A], p(B), [])` is passed to the transformed clause of `p/1`. This continuation contains the code to be executed after the execution of `p(B)` and the list `[A]` needed to recover its environment. Consequently, there are two continuations associated with the suspension: one continuation to execute the rest of the code of `p(B)` and another one to execute the rest of the code of `t(A)`.

After the first answer is found, this double continuation is resumed. It is executed as a normal Prolog and the second answer, `t(1)`, is found.

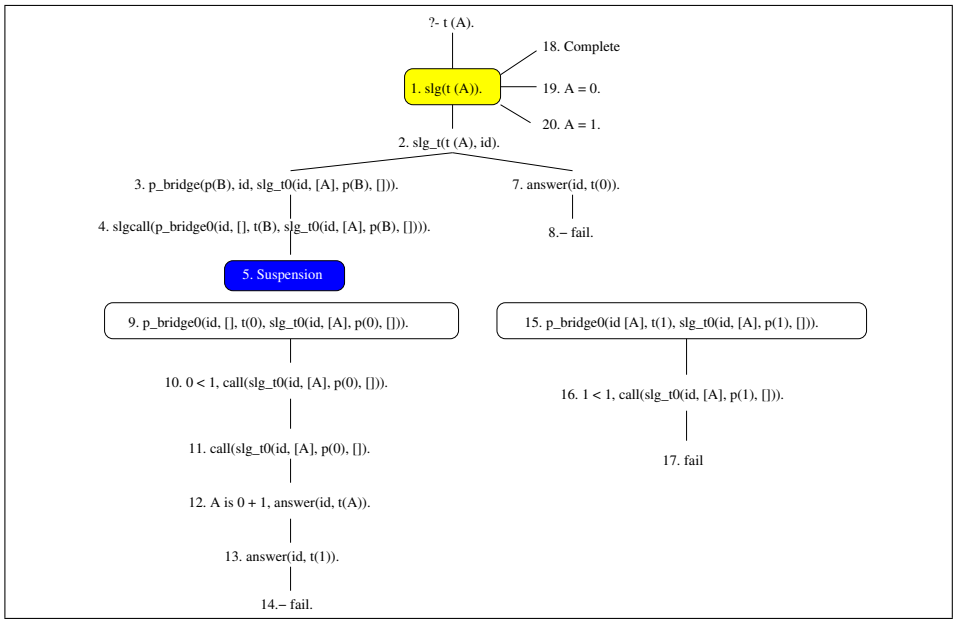


Fig. 9. New `CCall` tabling execution.

5 Θ(CHAT) is not comparable with Θ(CCall)

In this section we present a comparative analysis of the complexity of `CCall` and CHAT, which is an efficient implementation of tabling with a comparatively simple machinery. Since it is known that Θ(CHAT) is Θ(`SLG-WAM`) [7], the comparative analysis applies to the `SLG-WAM` as well.

The complexity analysis focuses on the operations of suspension and resumption. The environment of a consumer has to be protected when suspending to reinstall it when resuming. `CCall` achieves that by copying the continuation associated with the consumer in a special memory area to be protected on backtracking. In the original implementation [15] this continuation is copied from the heap to a separate table (when suspending) and back (when resuming). As proposed in [6], continuations can be saved in a special memory area with the same data format as the heap. This makes it possible to use WAM instructions and additional machinery on them and, when resuming, they can be used as normal Prolog data and code, without being recopied each time a consumer is resumed.

On the other hand, CHAT freezes the heap and the frame stack when resuming. The heap and frame stack are frozen by traversing the choice point stack. For all the choice points between the consumer choice point and its generator, the pointer to the end of the heap and frame stack are changed to the values of the consumer choice point values. By doing that, heap and frame stack are protected on backtracking. However, the consumer choice point has to be

copied to a special memory area as well as the segment trail (with its associated values) between the consumer and the generator, to reinstall the values of the bound variables at the time of suspension which backtracking will unbind. In consequence, when resuming the trail values have to be reinstalled as well as the consumer choice point.

Each consumer is suspended only once, and it can be resumed several times. The rest of the operations, i.e., checking if a tabled call is a generator or a consumer, are not analyzed, because they are common to both systems. In addition, we will ignore the cost of working at the Prolog level, since this is an orthogonal issue: `CCall` primitives could be compiled to WAM instructions and working at Prolog level does not increase the system complexity.

Θ(`CCall`): when suspending, `CCall` has to copy all the environments until the last generator and the structures in the heap which hang from them. If we name `E` the size of all the environments and `H` the size of the structures in the heap, the time consumption when suspending is: Θ(`E + H`).

When resuming, `CCall` just has to perform pattern matching of the continuation against its clause. The time taken by the pattern matching depends on the size of the list of bindings, which is known to be Θ(`E`). Since each consumer can be resumed `N` times, the time consumption of resuming consumers is Θ(`N`$\times$`E`).

Θ(CHAT): when suspending, CHAT has to traverse the frame and choicepoint stacks, but with the improvements presented in [7], the time this takes can be neglected because a choice point is only traversed once for all the consumers. The trail and the last choice point have to be copied. If we call `T` the size of the trail and `C` the size of the choice point, which is bound by a constant for a given program, the time consumption when suspending is: Θ(`T`).

When resuming, CHAT has to reinstall the values of the frame and the choice point. Since each consumer can be resumed `N` times, the time consumption of resuming is Θ(`N`$\times$`T`).

Analyzing the worst cases of both systems: we can conclude `E + H` $\geq$ `T`, because each variable can only be once in the trail, and then `CCall` is worse than CHAT when suspending. On the other hand, in case that `E < T`, `CCall` is better than CHAT when resuming. Consequently, for a plausible general case, the more resumptions there are, the better `CCall` behaves in comparison with CHAT, and conversely. In any case, the worst and best cases for each implementation are different, which makes them difficult to compare. For example, if there is a very large structure pointed to from the environments, and none of its elements are pointed to from the trail, `CCall` is slower than CHAT, since it has to copy all the structure in a different memory area when suspending and CHAT does nothing both when suspending and when resuming.

On the other hand, if all the elements of the structure are pointed to from the trail, `CCall` has to copy all the structure on suspension in a different memory area

to protect it on backtracking, but it is ready to be resumed without any other operation (just a unification with the pointer to the structure). CHAT has to copy all the structure on suspension too, because all the structure is in the trail. In addition, each time the consumer is resumed, all the elements of the structure have to be reinstalled using the trail, and CHAT has to perform more operations than `CCall`, and then, the more resumptions there are, the worse CHAT would be in comparison with `CCall`. Anyway, as the trail is usually much smaller than the heap, in general cases, CHAT will have an advantage over `CCall`.

6 Performance Evaluation

We have implemented the proposed technique as an extension of the Ciao system [1]. Tabled evaluation is provided to the user as a loadable *package* that implements the new directives and user-level predicates, performs the program transformations, and links in the low-level support for tabling. We have implemented `CCall` tabling with the efficiency improvements presented in [6] and the new translation for general programs explained in this paper.

Table 1 aims at determining how the proposed implementation of tabling compares with state-of-the-art systems —namely, the latest available versions of XSB, YapTab, and B-Prolog, at the time of writing, using the typical benchmarks which appear in other performance evaluations of tabling approaches.[7] In this table we provide, for several benchmarks, the raw time (in milliseconds) taken to execute them using tabling. Measurements have been made with Ciao-1.13, using the standard, unoptimized bytecode-based compilation, and with the `CCall` extensions loaded, as well as in XSB 3.0.1, YapTab 5.1.1, and B-Prolog 7.0. Note that we did not compare with CHAT, which was available as a configuration option in the XSB system and which was removed in recent XSB versions. CHAT can be expected to be at least as fast (if not slightly faster) than XSB.

All the executions were performed using local scheduling and disabling garbage collection; in the end this did not impact execution times very much. We used `gcc 4.1.1` to compile all the systems, and we executed them on a machine with Fedora Core Linux, kernel 2.6.9, and an Intel Xeon DESCHUTES processor.

The first benchmark is `path`, the same as Figure 1, which has been executed with a chain-shaped graph. Since this is a tabling-intensive program with no consumers in its execution, the difference with other systems is mainly due to having large parts of the execution done at Prolog level. The following five benchmarks, until `atr2`, are also tabling intensive. As their associated environments are very small, `CCall` is far from its worst case (see Section 5), and the difference with other systems is similar to that in `path` and for a similar reason. The worst case in this set is `tcn` because there are two calls to `slgcall/1` per generator, and the overhead of working at the Prolog level is duplicated.

B-Prolog, which uses a linear tabling approach, suffers if costly predicates have to be recomputed: this is what happens in benchmarks from `pg` until `peep`,

[7] This is in contrast to [6] where, due to the limitations of the `CCall` approach the benchmarks presented did not need the use of bridge predicates.

Program	CCall	XSB	YapTab	BProlog	# Bridges
path	517.92	231.4	151.12	206.26	0
tcl	96.93	59.91	39.16	51.60	0
tcr	315.44	106.91	90.13	96.21	0
tcn	485.77	123.21	85.87	117.70	0
sgm	3151.8	1733.1	1110.1	1474.0	0
atr2	689.86	602.03	262.44	320.07	0
pg	15.240	13.435	8.5482	36.448	6
kalah	23.152	19.187	13.156	28.333	20
gabriel	23.500	19.633	12.384	40.753	12
disj	18.095	15.762	9.2131	29.095	15
cs_o	34.176	27.644	18.169	85.719	14
cs_r	66.699	55.087	34.873	170.25	15
peep	68.757	58.161	37.124	150.14	10

Table 1. Comparing Ciao+CCall with XSB, YapTab, and B-Prolog.

where tabled and non-tabled execution is mixed. This is a well-known disadvantage of linear tabling techniques which does not affect suspension-based approaches. It has to be noted, however, that latest versions of B-Prolog implement an optimized variant of its original linear tabling mechanism [21] which tries to avoid reevaluation of looping subgoals.

In order to compare our implementation with XSB and YapTab, we must take into account that the speeds of XSB, and YapTab[8] are different, at least in those cases where the program execution is large enough to be really significant (between 1.8 and 2 times slower in the case of XSB and 1.5 times faster in the case of YapTab).

In non-trivial benchmarks, from `pg` until `peep`, which at least in principle should reflect more accurately what one might expect in larger applications using tabling, execution times are in the end very competitive when comparing with XSB or YapTab. This is probably due to the fact that the raw speed of the basic engine in Ciao is higher than in XSB and closer to YapTab, rather than to factors related to tabling execution, but it also implies that the overhead of the approach to tabling used is reasonable after the proposed optimizations in [6]. In this context it should be noted that in these experiments we have used the baseline, bytecode-based compilation and abstract machine. Turning on global analysis and using optimizing compilers and abstract machines [11, 3, 12] can further improve the speed of the SLD part of the computation.

[8] Note that we are comparing the tabled-enabled version of Yap, which is somewhat slower than the regular Yap.

7 Conclusions

We have presented an extension of the continuation call technique which does not have the limitations of the original continuation call approach regarding the interleaving of tabled and non-tabled predicates. This approach has the advantage of being easier to implement and maintain than other techniques which require non-trivial modifications to low-level machinery. Although there is an overhead imposed by executing at Prolog level, we expect the speed of the source (Prolog) language to gradually improve by using global analysis, optimizing compilers, and better abstract machines. Accordingly, we expect the performance of `CCall` to improve in the future and thus gradually gain ground in the comparisons.

Although a non optimal tabled execution is obviously a disadvantage, it is worth noting that, since our implementation introduces only minimal changes in the WAM and none in the associated Prolog compiler, the speed at which regular Prolog is executed remains unchanged. In addition to this, the modular design of our approach gives better chances of making it easier to port to other systems. In our case, executables which do not need tabling have very little tabling-related code, as the data structures (for tries, etc.) are handled by dynamic libraries loaded on demand, and only stubs are needed in the regular engine. The program transformation is taken care of by a plugin for the Ciao compiler [2] (a "package," in Ciao's terms) which is loaded and active only at compile time, and which does not remain in the final executable.

References

1. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.
2. D. Cabeza and M. Hermenegildo. The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
3. M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo. High-Level Languages for Small Devices: A Case Study. In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.
4. Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
5. S. Dawson, C.R. Ramakrishnan, and D.S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In *Proceedings of PLDI'96*, pages 117–126, New York, USA, 1996. ACM Press.
6. P. Chico de Guzmán, M. Carro, M. Hermenegildo, Claudio Silva, and Ricardo Rocha. An Improved Continuation Call-Based Implementation of Tabling. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 198–213. Springer-Verlag, January 2008.

7. Bart Demoen and K. Sagonas. CHAT is θ(SLG-WAM). In D. Mc. Allester H. Ganzinger and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning*, volume 1705 of *Lectures Notes in Computer Science*, pages 337–357. Springer, September 1999.
8. Bart Demoen and Konstantinos Sagonas. CAT: The Copying Approach to Tabling. In *Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 21–35. Springer-Verlag, 1998.
9. Bart Demoen and Konstantinos F. Sagonas. Chat: The copy-hybrid approach to tabling. In *Practical Applications of Declarative Languages*, pages 106–121, 1999.
10. Hai-Feng Guo and Gopal Gupta. A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In *International Conference on Logic Programming*, pages 181–196, 2001.
11. J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.
12. J. Morales, M. Carro, and M. Hermenegildo. Comparing Tag Scheme Variations Using an Abstract Machine Generator. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 32–43. ACM Press, July 2008.
13. Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 143–154. Springer Verlag, 1997.
14. Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
15. R. Ramesh and Weidong Chen. Implementation of tabled evaluation with delaying in prolog. *IEEE Trans. Knowl. Data Eng.*, 9(4):559–574, 1997.
16. R. Rocha, C. Silva, and R. Lopes. On Applying Program Transformation to Implement Suspension-Based Tabling in Prolog. In V. Dahl and I. Niemelä, editors, *23rd International Conference on Logic Programming*, number 4670 in LNCS, pages 444–445, Porto, Portugal, September 2007. Springer-Verlag.
17. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, May 1998.
18. H. Tamaki and M. Sato. OLD resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, 1986. Lecture Notes in Computer Science, Springer-Verlag.
19. D.S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.
20. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.
21. Neng-Fa Zhou, T. Sato, and Yi-Dong Shen. Linear Tabling Strategies and Optimizations. *Theory and Practice of Logic Programming*, 8(1):81–109, 2008.
22. Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming*, 2001(10), October 2001.
23. Youyong Zou, Tim Finin, and Harry Chen. F-OWL: An Inference Engine for Semantic Web. In *Formal Approaches to Agent-Based Systems*, volume 3228 of *Lecture Notes in Computer Science*, pages 238–248. Springer Verlag, January 2005.

Extending Tabled Logic Programming with Multi-Threading: A Systems Perspective

Rui Marques[1], Terrance Swift[2], and José Cunha[1]

[1] CITI, Dep. Informática – FCT, Universidade Nova de Lisboa
[2] CENTRIA — Universidade Nova de Lisboa

Abstract. Tabled Logic Programming (TLP) has proven a useful paradigm for application areas such as natural language grammars, program analysis, model checking, ontology management, collaborative agents, and the semantic web. The benefits of TLP arise from the fact that tabling factors out redundant subcomputations when evaluating a goal, leading to powerful termination and complexity properties. While the design and implementation of sequential TLP systems has been heavily studied, multi-threaded TLP systems are much newer. Tabling can be integrated with multi-threading in a variety of ways. Different threads may use *private* tables to support their own computations, while *shared* tables can be used as a basis of communication among threads to amortize repeated queries and to exploit a measure of parallelism from a computation. This paper discusses multi-threaded TLP in the context of XSB, a leading open-source Prolog whose tabling engine has recently been extended for multi-threading, including tabled negation, tabled constraints, and subsumptive tabling.

Tabled Logic Programming (TLP) has proven to be an important area of Logic Programming (LP) over the last decade, with research and commercial use in such areas as natural language grammars, program analysis, model checking, ontology management, collaborative agents, and the semantic web. Following the initial implementation of tabling in XSB, various forms of tabling have been added to other open-source Prologs including B-Prolog, YAP, Mercury, ALS and Ciao. There are a number of reasons for the adoption of tabling. TLP is more declarative than LP: it ensures termination and polynomial complexity for logic programs with negation that have the *bounded term size* property – i.e. those for which the size of terms constructed during an evaluation is bounded. Tabling can also evaluate negation according to the Well-Founded Semantics, which among other advantages allows an integration of Prolog-style systems with ASP systems that solve combinatorial problems. Finally, tabling can be closely integrated with Prolog systems so that constraints, cuts, and exceptions are supported, and implemented through extensions of Prolog's virtual machine.

Multi-threaded Prolog has been developed as a research activity for many years (cf. e.g [1]) and a draft ISO standard is available [2]. Many Prologs, including Ciao, SWI, YAP, Qu-Prolog and XSB, support multi-threaded programming, allowing programmers to benefit from parallelism by manually decompos-

ing queries. They provide a sophisticated environment for a number of applications, but most of them do not support multi-threaded TLP (MT-TLP).

This paper presents the approach to MT-TLP taken by XSB Prolog, which supports a wide variety of TLP features, including tabled negation, tabled constraints, call subsumption, answer subsumption, incremental recomputation of tables, tabled dynamic code and garbage collection of abolished tables. When multi-threading is added, tables may be *private* to a thread, or *shared* among threads, leading to several design goals:

- Any tabling function should be available to any active thread using tables that are private to a thread.
- Any tabling function should be available to any active thread using tables that are shared among threads.
- Private tables should be highly scalable up to the number of cores available.
- For problems that support large amounts of parallelism, shared tables should be able to provide speedup proportional to the number of cores available.

Although these goals are ambitious, many are already supported in Version 3.2 of XSB [3]. We first review aspects of TLP in Section 1. We then describe MT-TLP in XSB Version 3.2, including a high-level description of algorithms for multi-threaded computations that share tables. In order to illustrate how MT-TLP functions can be used in practice, Section 3 presents extended examples of its application to various types of Petri-Net formalisms. Finally, Section 4 discusses performance of these and other examples.

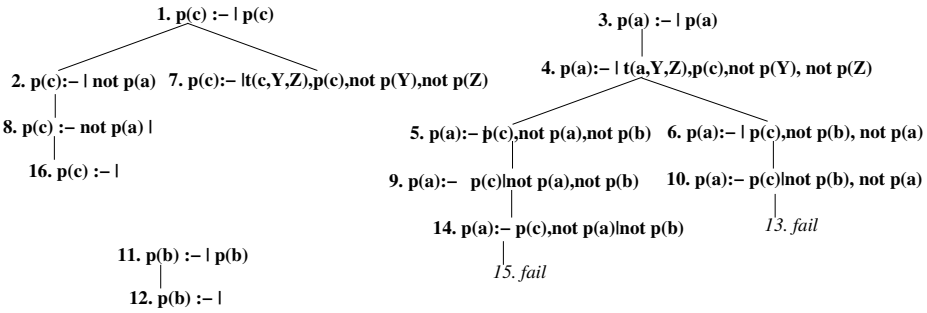
1 Tabling

We review aspects of tabling relevant to our presentation. Due to space limitations, the presentation is informal: references for formalisms of tabling and for proofs of its complexity and termination properties can be found in e.g. [3].

Example 1. Figure 1 shows a program P_1 along with the tabled evaluation of the goal `?- p(c)` represented as a forest of trees. In Figure 1, the number beside each node indicates the order of its creation. Accordingly, the evaluation begins with a tree for `p(c)` with root node labeled with the clause `p(c):- |p(c)`. The head of each node label represents the bindings, if any, to the subgoal made during the course of evaluation. The body of a node label has two parts: the literals on the right of the `|` symbol represent unresolved goals, which in this paper are selected in a fixed left-to-right order; the literals on the left represent *delayed* literals. PROGRAM CLAUSE RESOLUTION is used to produce the children of the root of a tree: this operation produces node 1, with selected literal `not p(a)`. The atom `p(a)` is new to the evaluation, so a NEW SUBGOAL operation creates a tree rooted at node 3 for `p(a)`. PROGRAM CLAUSE RESOLUTION creates node 4, and because the selected literal of node 4 `t(a,Y,Z)` is not tabled, PROGRAM

[3] Version 3.2 has not yet been released, but all features mentioned in this paper are available in the CVS version of XSB available at `xsb.sourceforge.net`.

CLAUSE RESOLUTION creates node 5 also. Since the selected literal of node 5, `p(c)` has a tree in the forest, but has no answers, the evaluation backtracks to node 4 and produces node 6. Once again, there are no answers to return so the evaluation backtracks to the tree for `p(c)` and produces node 7. At this stage none of the selected literals for nodes 1-7 have answers, nor unresolved program clauses. However, SLG provides a DELAYING operation which allows a selected negative literal to be moved to the left of the `|` symbol. This operation creates node 8, which is considered to be an answer, since it has no unresolved goals; at the same time, since node 8 has a delayed literal it is a *conditional* answer. This conditional answer can be used to resolve against the selected literals of nodes 5 and 6, producing nodes 9 and 10. Note that using a conditional answer for resolution causes the resolved goal to be delayed. The atom `p(b)` of the selected literal of node 10 is new to the evaluation, so a new tree is created for `p(b)` which produces an unconditional answer causing the derivation path from node 10 to be failed as indicated by the failure node 13. At this point, node 9 has `not p(a)` as its selected goal. Another DELAYING operation is performed to produce node 14, which again has `not p(b)` as its selected literal and produces a failure node. At this point, all operations have been performed on the selected literals of all nodes. The trees are *completely evaluated* and can be marked as *complete*. Once they are completed and it is determined that `p(a)` has no (conditional or unconditional) answers, a SIMPLIFICATION operation removes `not p(a)` from the delay list of node 8 to produce the unconditional answer in node 16.




```
:– table p/1.

p(c):– not p(a).        p(b).        p(X):– t(X,Y,Z),p(c),not p(Y),not p(Z).

t(a,b,a).               t(a,a,b).
```

Fig. 1. The program P_1 and tabled evaluation of goal `?- p(c)` to P_1

Example 1 illustrates a number of operational aspects of tabling. First, a tabled evaluation needs to be able to *suspend* and later *resume* a computation path, as when the path to node 5 is suspended and later resumed to produce node 9. Next, since non-completed subgoals require execution stack space while completed subgoals require only table space to store their answers, a practical tabled evaluation must be able to *incrementally complete* tabled subgoals to ensure space efficiency. For instance, the tree for `p(b)` can be completed immediately after node 12 is produced.

However, there are aspects of tabled evaluation that Example 1 does not explicitly demonstrate. Example 1 implicitly uses *call variance* – a NEW SUBGOAL operation is performed on a tabled subgoal S if no variant of S has previously been encountered. In some evaluations, it can be useful to restrict NEW SUBGOAL operations to occur only if no *subsuming* call for S had been encountered. Call subsumption can be efficient for applications that can exploit it – for instance computing a bottom-up fixed point for program analysis or for RDF inference. However call subsumption introduces overheads when it is not used (about 20% in XSB). Furthermore, there may be situations in which it is important to maintain the call patterns of an evaluation, as in tabling a meta-interpreter: such patterns are preserved by call variance, but not by call subsumption. In addition, Example 1 does not make use of a tabling feature called *answer subsumption*. Rather than returning every answer for a (perhaps completed) table, it may be best to return answers that are optimal according to some partial order. Similarly, answer subsumption may return an answer that is a function of other answers. For instance, an answer may be resolved against a consuming subgoal only if it is the join of other answers [4], or if the answer is the summation of independently derived probabilities [5]. As shown by the Petri Net example in Section 3 that uses ω-sequences, answer subsumption also can be useful for ensuring termination by abstracting answers.

Example 1 also does not use dynamic code, which interacts with tabling in two ways. First, dynamic predicates can be tabled in XSB, a handy feature for applications that generate tabled code. Second, when a tabled evaluation relies on a dynamic predicate D_p information in the table may become out of date as clauses of D_p are asserted or retracted. By using suitable declarations, *incremental recomputation* can automatically maintain the consistency of tables with dynamic code. While in Version 3.2 of XSB incremental recomputation is restricted to definite programs, it has proven useful in applications [6].

The ability to maintain constrained variables in the subgoals and answers of tables is useful for the analysis of temporal systems (see for instance [7]). A final critical, but often overlooked feature of tabling systems is the ability to abolish tables and reclaim their space. Version 3.2 of XSB allows reclamation of table space for abolished completed tables at the predicate and subgoal level. It may not be safe to immediately reclaim the space of an abolished table, as choice points may point into the table's code. Thus, in a manner similar to reclaiming retracted dynamic clauses, a pointer to the predicate or subgoal is put on a list of elements to later garbage collect when it is safe to do so.

Scheduling Strategies Two popular strategies for performing tabling operations are Local evaluation (the underlying strategy of Example 1) and Batched evaluation. Local evaluation is based on a *Subgoal Dependency Graph (SDG)* constructed from a forest of trees, $\mathcal{F}$ (cf. Figure 1). This graph has as its vertices each non-completed tabled subgoal in the forest, and has a link (S_1, S_2) if a node in the tree for subgoal S_1 has subgoal S_2 in its selected literal or in a delayed literal. Since $SDG(\mathcal{F})$ is a directed graph, a Strongly Connected Component (SCC) can be defined; As terminology a maximal SCC is an SCC that is contained in no other SCC, while an independent SCC $\mathcal{S}$ is an SCC such that there is no edge from a vertex in $\mathcal{S}$ to a vertex not contained in $\mathcal{S}$. In Local evaluation, tabling operations are performed only in trees whose subgoals are in an independent maximal SCC. Because of this restriction, Local evaluations have a behavior similar to a depth-first search. As a result, a given state of a Local evaluation generally has few uncompleted subgoals, and so is space efficient. In addition, Local evaluation prevents the return of an answer to a node in a tree that is not in an independent SCC. Along with other scheduling constraints, including ensuring that all SIMPLIFICATION operations are performed as early as possible, Local evaluation can guarantee that if a conditional answer with head A is returned to a node N outside of an independent SCC, then no unconditional answer with head A will ever be available to be returned to N. In Example 1 this would mean that if `p(a)` were part of a larger evaluation, its conditional answer (node 8) would never be returned outside of its SCC: only the unconditional answer (node 16) would be thus returned. Local evaluation is also advantageous for answer subsumption since it returns only the best answers (according to a given ordering) outside of an SCC.

However Local evaluation, is not useful for applications that require a single answer, or for applications where a table produces an answer that can be concurrently consumed by some other thread. For these purposes, Batched Evaluation is superior. Batched evaluation treats bindings made by answer resolution in the same way substitutions are treated in Prolog: the binding is propagated to all ancestor environments, thus "returning" an answer to its calling environment immediately. Answers are also returned to consuming nodes upon backtracking. Upon backtracking to the oldest subgoal in an SCC $\mathcal{S}$, $\mathcal{S}$ is either completed or a backtracking chain is created to return unresolved answers to consuming nodes for subgoals in $\mathcal{S}$. In this manner, answers are scheduled for return a batch at a time. For left recursion, Batched evaluation is about 10-20% faster than Local evaluation. The decision of whether to use Batched or Local evaluation is thus application dependent. XSB must be configured with one or the other strategy, but YAP allows dynamic mixing of the strategies [8].

2 Multi-Threaded Tabling

A multi-threaded tabling engine [9] was first made available in Version 3.0 of XSB, and has been substantially refined and extended since then. The simplest execution model is based on private tables, where each thread keeps its own copy of tabled information. This model has several advantages:

- Private tables use sequential tabling algorithms. The main implementation problems are to make the tabling engine reentrant with a low overhead, to allow each thread to reclaim its own table space and to ensure that allocation of table space does not affect scalability. Private tables in XSB support all tabling features that were present at the time of implementation, including tabled negation, tabled constraints, and call and answer subsumption.
- Private tables generally require no synchronization among threads above the level of memory allocation.
- Private tables are suitable to ensure query completeness or to support a particular semantics. Tables are automatically reclaimed when the thread that computed them exits. This reclamation includes not only subgoal and answer tries, but the delay lists and supporting structures used to compute the Well-Founded Semantics.

Shared tables tables are also important:

- If different threads require the same tables, memory usage for shared tables will be significantly lower than for private tables.
- Shared tables amortize execution time for (sub-)queries that are repeated by more than one thread.
- Shared tables allow the decomposition of a program, so that a set of threads computes a set of tables, partially supporting Table-Parallelism [10].

Execution Models for Shared Tables In [9] two models for shared tables, *Concurrent Local Evaluation* and *Concurrent Batched Evaluation* were proposed and implemented. In these models, the SLG forest is dynamically partitioned among threads, each thread evaluating a set of subgoals. In Concurrent Local Evaluation, which relies on Local Scheduling, when a thread T encounters a tabled subgoal S that has not been encountered by any thread, T evaluates S. Other threads are only allowed to use the table for S after T has completed S. Concurrency control for tables mainly arises when more than one thread evaluates different tabled subgoals in the same SCC at the same time. In this case, a deadlock will occur, which the engine detects and resolves, so that a single thread assumes computation of all tabled subgoals in the SCC. In Figure 1 such as situation would occur if a thread T_1 called `p(a)` and another called `p(c)` before it was called by T_1. Tabled subgoals that are computed by a new thread must have their answers recomputed. It is shown in [9] that recomputation does not add to the abstract complexity of the Well-Founded Semantics. Just as Local evaluation is the default scheduling strategy for sequential XSB and for thread-private tables, Concurrent Local Evaluation is the default scheduling strategy for thread-shared tables.

Because it is a type of Local Evaluation, Concurrent Local Evaluation does not allow a consuming node to use answers produced by a subgoal outside of its SCC until the table for the answers is completed – a restriction that prevents producer-consumer models of parallelism. This limitation is overcome by *Concurrent Batched Evaluation* which allows several threads to compute (inter-)dependent tabled subgoals in parallel. As with Concurrent Local Evaluation,

each subgoal can be computed by only one thread. However, a given thread may consume answers as they are produced by another thread. Within XSB, the implementation of Concurrent Batched Evaluation extends the implementation of sequential Batched Evaluation. In sequential Batched Evaluation, when the engine backtracks to the oldest subgoal in an SCC, it schedules the return of unconsumed answers for each consuming node in the SCC by creating a chain of choice points, and then backtracks into the newly created chain. This is extended to a multi-threaded context as follows. If different threads compute different SCCs, they can work independently, and can consume answers from other threads as they become available. However, let $\mathcal{S}$ be an SCC computed by multiple threads. All threads concurrently consume answers and perform other operations while they have work to do. Suppose a thread T_1 computing subgoals in $\mathcal{S}$ backtracks to the oldest subgoal that it "owns" in $\mathcal{S}$. If any other thread computing $\mathcal{S}$ is active, T_1 will suspend and will be re-awakened when a thread performs batch scheduling for $\mathcal{S}$; otherwise if T_1 is the last unsuspended thread computing subgoals in $\mathcal{S}$, T_1 itself will perform a fixed point check and batched scheduling and awaken the other threads computing $\mathcal{S}$ — either to return further answers or to complete their tables. As implemented in XSB, Concurrent Batched Evaluation thus allows parallel computation of subgoals, but has a sequential fixpoint check that synchronizes multiple threads when they compute the same SCC.

Implementation Status The status of MT-TLP in XSB Version 3.2 is shown in Table 1. Private tables support all features except for incremental recomputation (cf. Section 1, which was introduced after the multi-threaded engine was introduced into XSB. Concurrent Local Evaluation supports most features, but does not yet support call subsumption. In addition, it only partially supports space reclamation since shared tables can be abolished, but their space will not be reclaimed until there is only a single active thread in the engine. Both private tables and shared tables under Concurrent Local Evaluation have been heavily tested. XSB can also be configured to use Concurrent Batched Evaluation, however this model has been less thoroughly tested than Concurrent Local Evaluation and should be considered experimental. Nonetheless, Concurrent Batched Completion supports a number of tabling features, but is currently restricted to left-to-right dynamically stratified programs.

Related Work The approach to MT-TLP in XSB can be contrasted to that of OptYap [11]. OptYap extends an Or-parallel Prolog system with tabling, while XSB extends a Tabled Prolog system to allow multi-threading. The different starting points lead to different strengths in the current implementation of each system. In OptYap, different workers can collaborate to solve the same goal – leading to impressive speedups even in programs using left recursion. As will be shown in Section 4, shared tables in XSB can be used to speed up evaluations, but only for problems that are easily decomposable. Thus for definite programs, to which OptYap is currently restricted, OptYap can exploit much more parallelism than can XSB. On the other hand, XSB's implementation supports more tabling features within multi-threading, and integrates multi-threaded tabling more thoroughly with other system features, such as dynamic code and space

Feature	Private Tables	Shared Tables (Local)	Shared Tables (Batched-β)
Tabled constraints	Supported	Supported	Supported
Answer subsumption	Supported	Supported	Supported
Tabled Dynamic Code	Supported	Supported	Supported
Tabled negation	Supported	Supported	Partially Supported
Space reclamation	Supported	Partially Supported	Partially Supported
Call subsumption	Supported	Not supported	Not Supported
Incremental recomputation	Not supported	Not supported	Not Supported

Table 1. Multi-threaded functionality in XSB v. 3.2

reclamation. As a result, XSB can multi-thread computations that OptYap cannot (currently) evaluate, including several examples from Section 3.

3 Analysis of Petri Nets and Workflow Nets

The analysis of process logics in the style of Petri Nets illustrates a use of various tabled evaluations can exploit multi-threading. Reachability is a central problem for Petri Net analysis, to which problems such as liveness, deadlock-freedom, and the existence of homes states can be reduced. While we have taken care that the programs shown are correct and motivated by use cases, we stress that the methods described in this section are intended primarily to illustrate MT-TLP and to support the performance studies of Section 4, but do not represent fully developed analysis systems for Petri or Workflow Nets [4].

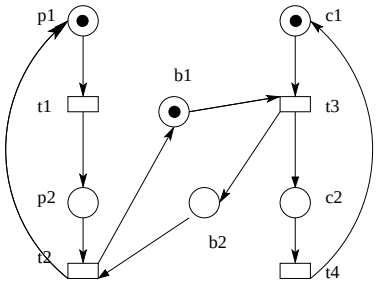


Fig. 2. A Simple Producer-Consumer Net

Using Tabling for Elementary Petri Nets Elementary Petri Nets (EPNs) or 1-safe Petri Nets (cf. [12]) are particularly simple to analyze. Consider the EPN shown in Figure 2, which depicts a simple producer consumer system. An EPN allows a place to contain at most 1 token; thus a finite EPN will have

[4] All programs can be obtained via `http://xsb.cvs.sourceforge.net/xsb/mttests/benches`.

only a finite number of configurations so that determining reachability of an EPN configuration is decidable. Our encoding represents the configuration of an EPN by a list of its marked places: thus the configuration in Figure 2 is represented as the list `[b1,c1,p1]`. Next, a transition T is represented by a list of places with input arcs to T ($\bullet T$) and output arcs from T ($T\bullet$). Predicate `trans/3` in Figure 3 shows each transition of Figure 2 represented as a Prolog fact, and that the transitions use XSB's trie indexing to obtain full indexing on list elements. Figure 3 shows a program for determining reachability in an EPN; so that solutions to the goal `reachable([b1,c1,p1],X)` are configurations reachable from the EPN in Figure 2. For efficiency the reachability program assumes that the lists in all transitions and configurations are sorted. For a transition T to have concession in a configuration C of an EPN, every place in $\bullet T$ must be marked, and no place in $T\bullet$ can be marked. These conditions are checked by the predicate `hasTransition/2` in Figure 3 which recurses through the places in the current configuration (`Conf`) to find sets of transitions that might have concession. This recursion (in `get_trans_for_conf_1/3`) allows indexed calls to transitions to be made based on each place in the input configuration. Each set of possible transitions is then filtered to include only those transitions that actually have concession in `Conf`, using operations on ordered sets (via `check_concession/2`). `hasTransition/2` succeeds when the first of these transitions is applied; further transitions are applied upon backtracking.

Based on `hasTransition/2`, a tabled reachability predicate can be written as a simple left-recursion. Tabling `reachable/2` is useful in two ways: it prevents looping when a given configuration is reachable from itself; and it also filters out redundant paths to a reachable configuration. By using the left recursive form of `reachable/2`, a typical call such as `reachable([b1,c1,p1],X)` with first argument bound and second free, would require a single tabled subgoal, and would have as answers all configurations reachable from `[b1,c1,p1]`. XSB's use of tries to represent tabled subgoals and their answers, allows efficient checking of answers and efficient use of memory, since the trie data structure factors out common list prefixes. If `reachable/2` is made thread-shared, then various threads can access the table to determine useful transitions, isolated places, and other information. Reachability analysis can exploit multi-threading if there is more than one initial configuration of interest or if a Petri Net is coarsely decomposable.

Using Petri Nets to Model Workflows The analysis and verification workflows is a promising direction for MT-TLP. Petri net-based formalisms, called Workflow Nets, are suitable to represent control and data flows, such as loops, I/O preconditions, if/then clauses and other synchronization dependencies between workflow units. To model reachability in a Workflow Net, the EPN is first extended to allow multiple tokens in a given place, and to change the representation of marked place from a constant such as `p1` to a Prolog term that is marked with a given instance and perhaps other information, e.g. `p1(instance(7))`. Transitions are then extended with functionality to dynamically evaluate guard conditions, to create sub-instances, to check for the absence of tokens in given places (which allows merging of dynamically created paths

```
% Prolog representation of the Producer-Consumer Net
:- index(trans/2,trie).
trans([p1],[p2],t1).          trans([b2,p2],[p1,b1],t2).
trans([b1,c1],[b2,c2],t3).    trans([c2],[c1],t4).

% Program to determine reachability of an elementary net
:- table reachable/2.
reachable(InConf,NewConf):-
   reachable(InConf,Conf),
   hasTransition(Conf,NewConf).
reachable(InConf,NewConf):-
   hasTransition(InConf,NewConf).

hasTransition(Conf,NewConf):-
   get_trans_for_conf(Conf,AllTrans),
   member(Trans,AllTrans),
   apply_trans_to_conf(Trans,Conf,NewConf).

get_trans_for_conf(Conf,Flattrans):-
   get_trans_for_conf_1(Conf,Conf,Trans),
   flatten(Trans,Flattrans).

get_trans_for_conf_1([],_Conf,[]).
get_trans_for_conf_1([H|T],Conf,[Trans1|RT]):-
   findall(trans([H|In],Out,Tran),trans([H|In],Out,Tran),Trans),
   check_concession(Trans,Conf,Trans1),
   get_trans_for_conf_1(T,Conf,RT).

check_concession([],_,[]).
check_concession([trans(In,Out,Name)|T],Input,[trans(In,Out,Name)|T1]):-
   ord_subset(In,Input),
   ord_disjoint(Out,Input),!,
   check_concession(T,Input,T1).
check_concession([_Trans|T],Input,T1):-
   check_concession(T,Input,T1).

apply_trans_to_conf(trans(In,Out_Name),Conf,NewConf):-
   ord_subtract(Conf,In,Diff),
   flatten([Out|Diff],Temp),
   sort(Temp,NewConf).
```

Fig. 3. TLP Program for analyzing Elementary Petri Nets

through the net), and to delete tokens from places if a transition is taken (which allows cancellation). Transitions for Workflow Net have the abstract form

```
trans(InConf,OutConf,dyn(Conditions,Effects))
```

where the last argument contains dynamic conditions that must be satisfied before the transition can be taken, and dynamic effects to be applied upon taking the transition (e.g. cancellation). The Workflow Net evaluator based on this syntax is approximately twice the size of that of Figure 3, and can emulate nearly all common workflow control patterns [13]. In fact, the emulator has been used with MT-TLP to analyze health workflows based on clinical care guidelines.

Using Answer Subsumption for ω Sequences Workflow nets are an extension of Place/Transition Petri Nets, which do not distinguish between tokens, but do allow a place to hold more than one token. Reachability is decidable in Place/Transition Nets, and can be determined using a method called ω-sequences (see e.g. [14]). The main idea in determining ω sequences is to define a partial order $\geq_\omega$ as follows. If configurations C_1 and C_2 are both reachable, C_1 and C_2 have tokens in the same set Pl of places, and there exists a non-empty $PL_{sub} \subseteq PL$, such that for each $pl \in Pl_{sub}$ C_1 has strictly more tokens than C_2, then $C_1 >_\omega C_2$. When evaluating reachability, if C_2 is reached first, and then C_1 was subsequently reached, C_1 is abstracted by marking each place in PL_{sub} with the special token ω which is taken to be greater than any integer. If C_1 was reached first and then C_2, C_2 is treated as having already been seen.

From the viewpoint of TLP, ω-abstractions form an example of answer subsumption. To compute reachability with ω abstractions, when each solution S to `reachable/2` is obtained, the solution S is compared to answers in the table. If some answer in the table is greater than or equal to S in $\geq_\omega$ then S is not added to the table; however if S is greater than some set $\mathcal{S}_A$ of answers, the answers $\mathcal{S}_A$ are removed from the table and the ω abstraction of S with respect to $\mathcal{S}_A$ is added. The main top-level change to Figure 3 needed for implementation is the use of the XSB library predicate `filterPOA/5` as the top-level call and in the first clause of `reachable/2`.

```
reachable(InConf,NewConf):-
   filterPOA(reachable(InConf),Conf,gte_omega,omega_abstr,call_abstr),
   hasTransition(Conf,NewConf).
```

`filterPOA/5` takes the call and argument to which answer subsumption is to be applied as its first two arguments, while the third argument, `gte_omega` is the name of the partial order itself. The forth argument is the name of the predicate to use to perform ω-abstraction of answers. Finally, the fifth argument, is the name of the predicate to compare a candidate solution Sol to answers in the table. The predicate, `call_abstr/2` abstracts Sol to form a call to the table so that only a small set of answers will be compared with Sol to determine if Sol should be added to the table and possibly ω-abstracted [5]. In other words, upon derivation of Sol, a term $Call_{Sol}$ is created using `call_abs/2` and all answers in

[5] `filterPOA/5` is itself tabled and uses thread-private tables; for shared tables `shared_filterPOA/5` is used.

the current table that unify with $Call_{Sol}$ are collected. Each of these is compared to Sol using `gte_omega/2`. If one of the answers $>_\omega$ than Sol, the predicate fails; otherwise the set $\mathcal{A}$ of answers that Sol is $>_\omega$ than is collected, and if non-empty, the abstraction of Sol with respect to $\mathcal{A}$, Sol_{abs} is taken; the answers in $\mathcal{A}$ deleted from the table, and Sol_{abs} added.

Extending nets with Constraint-based Reasoning A variety of formalisms extend Place/Transition Nets to add conditions that must be evaluated for a transition to fire and effects that must occur upon its firing. In the Workflow nets described above conditions and effects were Prolog predicates, but there is no reason why a condition could not be the entailment of a formula in a given constraint domain, and the effect the propagation of new constraints to variables associated with given places in the net. Using such an approach, constraint-based reasoning can be incorporated into workflow or other process specifications. The top-level change required to implement constraint nets occurs when actually applying a transition to a configuration, in `apply_trans_to_conf/3`:

```
apply_trans_to_conf(trans(In,Entailment,Out),Conf,NewConf):-
   unify_for_entailment(In,Conf,MidConf),
   entailed(Entailment),
   call_new_constraints(Out,OutPlaces),
   flatsort([OutPlaces|MidConf],NewConf).
```

First, variables in the transition are unified with those of the configuration to produce a new constraint store. If the formula `Entailment` is entailed by the constraint store, new constraints from the transition are placed on the output variables via calling the constraints in the list `Out`. Note that this extension is not specific to a given constraint domain, but its use for reachability does depend on tabled constraints.

Using Tabled Negation for Preferences on Nets Preferences can be combined with Workflow nets so that if more than one transition is possible for a given configuration C of a workflow instance, only preferred transitions from C are taken. This has two practical uses. First, the preferences may check run-time information from a database or other store to determine what transitions to avoid: in fact, since the preference relation is simply a (tabled) Prolog predicate the preference relation may perform sophisticated run-time look-aheads. Second, since preferences can be dynamic, they may be used to fine-tune a general workflow to local policies – for instance adjusting a clinical workflow system to policies of a given hospital, medical department, or ward. Adapting the methodology of [15], the top-level change to the code of Figure 3 is to the `hasTransition/2` predicate

```
hasTransition(Conf,NewConf):-
   get_trans_for_conf(Conf,AllTrans),
   member(Trans,AllTrans),
   sk_not(unpreferred(Trans,AllTrans,Conf)),
   apply_trans_to_conf(Trans,Conf,NewConf).
```

`sk_not/1` is an XSB predicate that soundly evaluates non-ground tabled negation by skolemizing variables, ensuring here that only preferred transitions are taken. Since the basis for preferences is the well-founded semantics, if a transition is

preferred to itself at a given configuration, `hasTransition/2` will produce an answer that is neither true nor false.

Summary: Tabling for Petri Nets Tabling provides a concise means for coding reachablility (and other analysis problems) for a variety of Petri-net formalisms. At the same time, tabling may not be the best approach for all such problems. Reachability in EPNs is in *PSPACE* [16], while in the worst case, tabling requires 2^N states for an EPN with N places. At the same time, algorithms for reachability that stay in *PSPACE* (e.g. by using loop-checking) will in the worst case require time proportional to the number of traces (paths) rather than to the number of states, as required by tabling.

4 Performance Results

Table 2 shows the performance results for benchmarks on a machine with a 4 core AMD 64 processor running Debian Linux. All times were taken as the best of three runs and are presented in seconds. The programs `Elementary`, `Workflow`, `Omega`, `Constraint`, and `Preferences` were all discussed in the previous section. `Dynamic Elementary` is the same as `Elementary` except that it uses tabled dynamic code for `reachable/2`. The nets tested vary with each type of benchmark. For (Dynamic) `Elementary`, the underlying nets are designed to capture the effects of repeatedly locking and unlocking mutexes, while in `Workflow` the net is designed to use a number of standard workflow control patterns from [13]. The net for `Omega` was synthesized to have a relatively small number of places in which ω-abstractions were necessary, although the *check* for whether an ω-abstraction was needed was necessary in all places. For `Constraints`, the network was designed so that places compete for a shared resource represented by a term with variables constrained using CLP(R). Once a place obtains a resource, various transitions fire to constrain the variables of a resource until they entail the guard of a transition that moves the term to another place along a path, and eventually back to the initial configuration. The net for `Preferences` extends a workflow net to prefer those transitions from a given configuration that cannot lead to proscribed configurations: the preferences thus model look-ahead within a workflow state. Due to differingzd limitations on the sizes of shared and of multiple copies of private tables, the sizes of the nets differ between private and shared versions of each benchmark, resulting in different performance numbers.

The benchmark `Call Subsumption` does not use a Petri Net formalism, but rather evaluates the goal `?- ranc(A,B)` to the tabled predicate

```
ranc(X,Y):- edge(X,Y).          ranc(X,Y):- edge(X,Z),ranc(Z,Y).
```

where `ranc/2` uses call subsumption and `edge/2` is a chain of 2048 vertices.

Table 2 presents the results of the benchmarks; however two other features of the benchmarks must be be explained before evaluating the results. First, the sizes of the underlying nets vary greatly from test to test, as do the number of reachable states – thus the absolute times should not be used to compare the benchmark of one kind of net to another. Second, shared table benchmarks test the time for N threads to each traverse 4/N identical nets. Thus, the shared

Programs Using Private Tables (Local Evaluation)					
N. threads	1	2	Overhead	4	Overhead
`Private Elementary`	5.94	6.23	4.8%	6.25	5.2%
`Private Dynamic Elementary`	6.03	6.03	0%	6.03	0%
`Private Workflow`	19.21	19.68	2.4%	19.95	3.8%
`Private Omega`	7.18	8.33	16.0%	10.3	46.0%
`Private Omega Specialized`	6.37	6.37	0%	6.37	0.0%
`Private Constraint`	2.75	2.84	3.2%	2.85	3.6%
`Private Preferences`	3.74	3.77	0.8%	3.82	2.1%
`Call Subsumption`	.86	1.04	20.0%	1	43%

Programs Using Shared Tables (Local Evaluation)					
N. threads	1	2	Speedup	4	Speedup
`Shared Elementary`	25.12	13.00	1.93	6.55	3.83
`Shared Dynamic Elemtary`	24.8	13.02	1.90	6.59	3.76
`Shared Workflow`	41.25	20.78	1.98	10.58	3.89
`Shared Omega`	19.58	10.38	1.88	5.57	3.51
`Shared Constraint`	11.13	5.56	2.00	2.83	3.93
`Shared Preferences`	3.73	1.86	1.99	0.95	3.92

Table 2. Scalability Results for Private and Shared Tables (Local Evaluation)

benchmarks test a "best case" situation for exploiting parallelism by shared tables. In Table 2 the scalability for both private and shared tables is usually linear to 4 cores, and the times for `Dynamic Elementary` are nearly the same as for `Elementary`. The first exception is the `Omega` benchmark using private tables. The slowdown in `Omega` was determined to arise from the use of `call/[2,3]` in the library predicate `filterPOA/5` (See Section 3). This use caused contention for the mutex protecting XSB's predicate table. When `filterPOA/5` was specialized to avoid `call/[2,3]` in `Omega Specialized`, the contention disappears, and the benchmark becomes scalable. The second exception to scalability is `Call Subsumption`. Executing `Call Subsumption` requires a large amount of space to be allocated for 2049 tabled calls and over $2k * 1k$ answers. While other benchmarks, such as `Elementary` also have a large number of answers, `Call Subsumption` spends nearly all of its time doing tabling operations — and memory management. Although XSB manages memory for private tables within a thread and so reduces contention for process-level memory managers, it cannot eliminate this contention. As a result, the high proportion of time spent on memory management in `Call Subsumption` reduces its scalability on `ranc/2`.

5 Discussion

We have described the approach to MT-TLP in XSB and shown how it can be used to evaluate sophisticated process and workflow formalisms in a simple and direct manner. The goals stated in Section 1 are ambitious: still, they are

largely met. Except for incremental recomputation, all the features in Table 1 are supported by private tables, while and nearly all except incremental recomputation and call subsumption are at least partially supported by shared tables. When supported, the tabled features can be almost always be made to scale linearly to the number of cores available for our benchmarking. Several existing XSB applications will benefit from the MT-TLP model as described in this paper. These include the ontology management system CDF [17], the object-logic system Flora-2 [18] and the model-checking system XMC [19]. The first two of these applications rely on tabled negation, while applications of XMC to real-time systems and security protocols rely on tabled constraints. For these and other applications, the MT-TLP model can increase availability and speed.

Acknowledgements The authors thank David S. Warren for help in implementing private tables and tabled dynamic code. Performance results we obtained on hardware supported with CITI FCTMCTES Plurianual Funding.

References

1. Wielemaker, J.: Native preemptive threads in SWI-Prolog. In: Practical Aspects of Declarative Languages. (2003) 331–345 LNCS 2916.
2. Moura, P.: Prolog multi-threading support. ISO/IEC. (5 2007) DTR 13211.
3. XSB: The XSB Programmer's Manual: Vols. 1 and 2. (2007) Available via `http://xsb.sourceforge.net`.
4. Swift, T.: Tabling for non-monotonic programming. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 201–240
5. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In: ICLP. (2004)
6. Ramakrishnan, C., Ramakrishnan, I., Warren, D.S.: XcelLog: A deductive spreadsheet system. Knowledge Engineering Review **22**(3) (2007) 269–279
7. Sarna-Starosta, B.: Constraint-based Analysis of Security Protocols. PhD thesis, SUNY Stony Brook (2005)
8. Rocha, R., Silva, F., Costa, V.S.: Dynamic mixed-strategy evaluation of tabled logic programs. In: ICLP. (2005) 250264
9. Marques, R.: Concurrent Tabling: Algorithms and Implementation. PhD thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa (2007) Available from `http://asc.di.fct.unl.pt/~rfm/research.html`.
10. Freire, J., Hu, R., Swift, T., Warren, D.S.: Parallelizing tabled evaluation. In: 7th International PLILP Symposium, Springer-Verlag (1995) 115–132
11. Rocha, R., Silva, F., Costa, V.S.: On applying or-parallelism and tabling to logic programs. TPLP **5**(1 & 2) (2005) 161–205
12. Rozenberg, G., Engelfriet, J.: Elemenary net systems. In: Lectures on Petri Nets I: Basic Models. Springer LNCS 1491 (1998) 12–121
13. van der Aalst, W., ter Hofstede, A.: YAWL: Yet another workflow language (revised version). Technical report, Queensland University of Technology (2003)
14. Desel, J., Reisig, W.: Place/transition Petri nets. In: Lectures on Petri Nets I: Basic Models. Springer LNCS 1491 (1998) 122–174
15. Cui, B., Swift, T.: Preference logic grammars: Fixed-point semantics and application to data standardization. Artificial Intelligence **138** (2002) 117–147

16. Esparza, J., Nielsen, M.: Decidability issues for Petri nets. J. Inform Process. Cybernet **30**(3) (1994) 143–160
17. Swift, T., Warren, D.S.: The meaning of cold dead fish. Available via `http://www.cs.sunysb.edu/~tswift` (2003)
18. Yang, G., Kifer, M.: Flora: Implementing an efficient dood system using a tabling logic engine. In: DOOD. (2000)
19. Ramakrishnan, C., Ramakrishnan, I., Smolka, S., Dong, Y., Du, X., Roychoudhury, A., Venkatakrishnan, V.: XMC: A logic-programming-based verification toolset. In: CAV. (2000) 576–590

Declarative Combinatorics in Prolog: Shapeshifting Data Objects with Isomorphisms and Hylomorphisms

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
E-mail: tarau@cs.unt.edu

Abstract. This paper is an exploration in a logic programming framework of isomorphisms between elementary data types (natural numbers, sets, finite functions, graphs, hypergraphs) and their extension to hereditarily finite universes through *hylomorphisms* derived from *ranking/unranking* and *pairing/unpairing* operations.
An embedded higher order combinator language provides any-to-any encodings automatically.
A few examples of "free algorithms" obtained by transferring operations between data types are shown. Other applications range from stream iterators on combinatorial objects to succinct data representations and generation of random instances.
The self-contained source code of the paper, as generated from a literate Prolog program, is available at `http://logic.csci.unt.edu/tarau/research/2008/pISO.zip`
Keywords: Prolog data representations, computational mathematics, ranking/unranking, Ackermann encoding, hereditarily finite sets and functions, pairing/unpairing

1 Introduction

Data structures in imperative languages have traditionally been designed with *mutability* in mind and therefore with space saving strategies based on in-place updates. On the contrary, the dominance of *immutable* data structures in declarative languages suggests *sharing* "equivalent" immutable components as an effective space saving alternative.

Moreover, in the presence of higher order constructs, function sharing among heterogeneous data objects, is also appealing, as a way to borrow or lend "free algorithms".

The closest analogy to this, drawn from everyday thinking, is ... *analogy*. Analogical/metaphoric thinking routinely shifts entities and operations from a field to another hoping to uncover similarities in representation or use.

However, this rises the question: what guaranties do we have that doing this between data types is useful and safe?

Also sharing heterogeneous data objects faces two problems:

- some form of equivalence needs to be proven between two objects A and B before A can replace B in a data structure, a possibly tedious and error prone task
- the fast growing diversity of data types makes harder and harder to recognize sharing opportunities.

The techniques introduced in this paper provide a generic solution to these problems, through isomorphic mappings between heterogeneous data types, such that unified internal representations make equivalence checking and sharing possible. The added benefit of these "shapeshifting" data types is that the functors transporting their data content will also transport their operations, resulting in shortcuts that provide, for free, implementations of interesting algorithms. The simplest instance is the case of isomorphisms – reversible mappings that also transport operations. In their simplest form such isomorphisms show up as *encodings* – to some simpler and easier to manipulate representation – for instance natural numbers.

Such encodings can be traced back to Gödel numberings [1, 2] associated to formulae, but a wide diversity of common computer operations, ranging from wireless data transmissions to cryptographic codes qualify.

Encodings between data types provide a variety of services ranging from free iterators and random objects to data compression and succinct representations. Tasks like serialization and persistence are facilitated by simplification of reading or writing operations without the need of special purpose parsers. Sensitivity to internal data representation format or size limitations can be circumvented without extra programming effort.

2 An Embedded Data Transformation Language

It is important to organize such encodings as a flexible embedded language to accommodate any-to-any conversions without the need to write one-to-one converters. Toward this end we will organize our encodings as a group of isomorphisms within a (mildly) category theory-inspired design.

We will start by designing an embedded transformation language as a set of operations on this group of isomorphisms. We will then extend it with a set of higher order combinators mediating the composition of encodings and the transfer of operations between data types.

2.1 The Group of Isomorphisms

We implement an isomorphism between two objects X and Y as a Prolog data type (a term with functor `iso/2`) `iso(F,G)`, encapsulating a bijection F and its inverse G.

$$X \underset{g = f^{-1}}{\overset{f = g^{-1}}{\rightleftarrows}} Y$$

As a well-known mechanism to embed higher order functions in Prolog [3], we will use `iso/2` as a *closure* (higher order predicate) to be applied to an input argument and an output argument. We assume the presence of Prolog's `call/N` predicate that applies a closure to `N-1` extra arguments and `maplist/N` that applies a closure to `N-1` extra list arguments. We can organize the *group* of isomorphisms as follows.

First we define the group structure as a set of isomorphism transformers:

```
compose(iso(F,G),iso(F1,G1),iso(fcompose(F1,F),fcompose(G,G1))).
itself(iso(id,id)).
invert(iso(F,G),iso(G,F)).
```

Then, we provide evaluators for isomorphisms, that apply their left or right functions to actual arguments. Note that like `iso/2`, compose/3 is a closure to be applied to 2 extra arguments with `call/2` or `maplist/2`.

```
fcompose(G,F,X,Y):-call(F,X,Z),call(G,Z,Y).
id(X,X).
from(iso(F,_),X,Y):-call(F,X,Y).
to(iso(_,G),X,Y):-call(G,X,Y).
```

The *from* function extracts the first component (a *section* in category theory parlance) and the *to* function extracts the second component (a *retraction*) defining the isomorphism. We can now formulate *laws* about isomorphisms that can be used to test correctness of implementations.

Proposition 1 *The data type* `iso/2` *specifies a group structure, i.e. the* compose *operation is associative,* itself *acts as an identity element and* invert *computes the inverse of an isomorphism.*

It is convenient to give a name to each isomorphism as a unary predicate

```
<name>(iso(From,To)).
```

We can transport operations from an object to another with *borrow* and *lend* combinators defined as follows:

```
borrow(IsoName,H,X,Y):-call(IsoName,iso(F,G)),
  fcompose(F,fcompose(H,G),X,Y).

lend(IsoName,H,X,Y):-call(IsoName,Iso),
  invert(Iso,iso(F,G)),
  fcompose(F,fcompose(H,G),X,Y).
```

The combinators `fit` and `retrofit` just transport an object `x` through an isomorphism and apply to it an operation `op` available on the other side:

```
fit(Op,IsoName,X,Y):-
  call(IsoName,Iso),fit_iso(Op,Iso,X,Y).

fit_iso(Op,Iso,X,Y):-
  from(Iso,X,Z),call(Op,Z,Y).
```

```
retrofit(Op,IsoName,X,Y):-call(IsoName,Iso),
  retrofit_iso(Op,Iso,X,Y).

retrofit_iso(Op,Iso,X,Y):-
  to(Iso,X,Z),call(Op,Z,Y).
```

We can see the combinators `from, to, compose, itself, invert, borrow, lend, fit etc.` as part of an *embedded data transformation language*. Various examples for their use will be given as soon as we populate our universe with interesting isomorphisms.

2.2 Choosing a Root

To avoid defining $n(n-1)/2$ isomorphisms between n objects, we choose a *Root* object to/from which we will actually implement isomorphisms. We will extend our embedded combinator language using the group structure of the isomorphisms to connect any two objects through isomorphisms to/from the *Root*.

Choosing a *Root* object is somewhat arbitrary, but it makes sense to pick a representation that is relatively easy convertible to various others, efficiently implementable and, last but not least, scalable to accommodate large objects up to the runtime system's actual memory limits.

We will choose as our *Root* object *Finite Sequences of Natural Numbers*. They can be seen as as finite functions from an initial segment of Nat, say $[0..n]$, to Nat. We will represent them as lists i.e. their Prolog type is $[Nat]$. Alternatively, an array representation can be chosen. Note that in the case of a Prolog not supporting arbitrary precision integers or rationals, such lists could be used, in principle, to emulate them at source level, through the use of isomorphisms mapping them to natural numbers, signed integers and then rational numbers, following the techniques described in [4, 5].

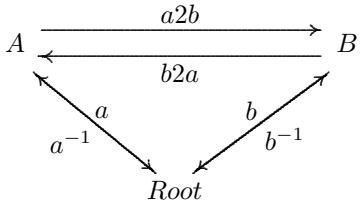
We can now define an *Encoder* as an isomorphism connecting an object to *Root* together with the combinators *with* and *as* providing an *embedded transformation language* for routing isomorphisms through two *Encoders*.

```
with(Iso1,Iso2,Iso):-invert(Iso2,Inv2),
  compose(Iso1,Inv2,Iso).

as(That,This,X,Y):-
  call(That,ThatF),call(This,ThisF),
  with(ThatF,ThisF,Iso),
  to(Iso,X,Y).
```

The combinator `with` turns two Encoders into an arbitrary isomorphism, i.e. acts as a connection hub between their domains. The combinator `as` adds a more convenient syntax such that converters between "a" and "b" can be designed as:

```
'a2b'(X,Y) :- as('a','b',X,Y).
'b2a'(X,Y) :- as('b','a',X,Y).
```



We will provide extensive use cases for these combinators as we populate our group of isomorphisms. Given that $[Nat]$ has been chosen as the root, we will define our finite function data type *fun* simply as the identity isomorphism on sequences in $[Nat]$.

```
fun(Iso) :-itself(Iso).
```

3 Extending the Group of Isomorphisms

We will now populate our group of isomorphisms with combinators based on a few primitive converters.

3.1 An Isomorphism to Finite Sets of Natural Numbers

The isomorphism is specified with two bijections `set2fun` and `fun2set`.

```
set(iso(set2fun,fun2set)).
```

While finite sets and sequences share a common representation $[Nat]$, sets are subject to the implicit constraint that all their elements are distinct[1]. This suggest that a set like $\{7,1,4,3\}$ could be represented by first ordering it as $\{1,3,4,7\}$ and then compute the differences between consecutive elements. This gives $[1,2,1,3]$, with the first element 1 followed by the increments $[2,1,3]$. To turn it into a bijection, including 0 as a possible member of a sequence, another adjustment is needed: elements in the sequence of increments should be replaced by their predecessors. This gives $[1,1,0,2]$ as implemented by `set2fun`:

```
set2fun([],[]).
set2fun([X|Xs],[X|Fs]):-
  sort([X|Xs],[_|Ys]),
  set2fun(Ys,X,Fs).

set2fun([],_,[]).
set2fun([X|Xs],Y,[A|As]):-A is (X-Y)-1,set2fun(Xs,X,As).
```

It can now be verified easily that incremental sums of the successors of numbers in such a sequence, return the original set in sorted form, as implemented by `fun2set`:

[1] Such constraints can be regarded as *laws/assertions* that we assume holding for a given data type, when needed, restricting it to the appropriate domain of the underlying mathematical concept.

```
fun2set([],[]).
fun2set([A|As],Xs):-findall(X,prefix_sum(A,As,X),Xs).

prefix_sum(A,As,R):-append(Ps,_,As),length(Ps,L),
  sumlist(Ps,S),R is A+S+L.
```

The resulting Encoder (`set`) is now ready to interoperate with another Encoder:

```
?- as(set,fun,[0, 1, 0, 0, 4],S).
S = [0, 2, 3, 4, 9].

?- as(fun,set,[0, 2, 3, 4, 9],F).
F = [0, 1, 0, 0, 4].
```

As the example shows, this encoding maps arbitrary lists of natural numbers representing finite functions to strictly increasing sequences of natural numbers representing sets.

3.2 Folding Sets into Natural Numbers

We can fold a set, represented as a list of distinct natural numbers into a single natural number, reversibly, by observing that it can be seen as the list of exponents of 2 in the number's base 2 representation.

```
nat_set(iso(nat2set,set2nat)).

nat2set(N,Xs):-nat2elements(N,Xs,0).

nat2elements(0,[],_K).
nat2elements(N,NewEs,K1):-N>0,
  B is /\(N,1),N1 is N>>1,K2 is K1+1,
  add_el(B,K1,Es,NewEs),
  nat2elements(N1,Es,K2).

add_el(0,_,Es,Es).
add_el(1,K,Es,[K|Es]).

set2nat(Xs,N):-set2nat(Xs,0,N).

set2nat([],R,R).
set2nat([X|Xs],R1,Rn):-R2 is R1+(1<<X),set2nat(Xs,R2,Rn).
```

We will standardize this pair of operations as an *Encoder* for a natural number using our Root as a mediator:

```
nat(Iso):-nat_set(NatSet),set(Set),compose(NatSet,Set,Iso).
```

The resulting Encoder (`nat`) is now ready to interoperate with any other Encoder:

```
?- as(fun,nat,42,F).
F = [1, 1, 1]
```

```
?- as(set,nat,42,F).
F = [1, 3, 5]

?- as(fun,nat,2008,F).
F = [3, 0, 1, 0, 0, 0, 0]

?- as(set,nat,2008,S).
S = [3, 4, 6, 7, 8, 9, 10]

?- lend(nat,reverse,2008,R).
R = 1135 % different, sequence depends on order

?- lend(nat_set,reverse,2008,R).
R = 2008 % same, set is order independent

?- as(set,nat,42,S).
S = [1, 3, 5]

?- fit(length,nat,42,L).
L = 3

?- retrofit(succ,nat_set,[1,3,5],N).
N = 43
```

The reader might notice at this point that we have already made full circle - as finite sets can be seen as instances of finite sequences. Injective functions that are not surjections with wider and wider gaps can be generated using the fact that one of the representations is information theoretically "denser" than the other, for a given range:

```
?- as(set,fun,[0,1,2,3],S1).
S1 = [0, 2, 5, 9].

?- as(set,fun,[0,2,5,9],S2).
S2 = [0, 3, 9, 19].

?- as(set,fun,[0,3,9,19],S3).
S3 = [0, 4, 14, 34].
```

4 Generic Unranking and Ranking Hylomorphisms

The *ranking problem* for a family of combinatorial objects is finding a unique natural number associated to it, called its *rank*. The inverse *unranking problem* consists of generating a unique combinatorial object associated to each natural number.

4.1 Pure Hereditarily Finite Data Types

The unranking operation is seen here as an instance of a generic *anamorphism* mechanism (an *unfold* operation), while the ranking operation is seen as an instance of the corresponding catamorphism (a *fold* operation) [6, 7]. Together they form a mixed transformation called *hylomorphism*.

We will use such hylomorphisms to lift isomorphisms between lists and natural numbers to isomorphisms between a derived "self-similar" tree data type and natural numbers. In particular we will derive Ackermann's encoding from Hereditarily Finite Sets to Natural Numbers.

The data type T representing hereditarily finite structures will be a generic multiway tree with a single leaf type `[]`.

The two sides of our hylomorphism are parameterized by two transformations `f` and `g` forming an isomorphism `Iso f g`:

```
unrank(F,N,R):-call(F,N,Y),unranks(F,Y,R).
unranks(F,Ns,Rs):-maplist(unrank(F),Ns,Rs).

rank(G,Ts,Rs):-ranks(G,Ts,Xs),call(G,Xs,Rs).
ranks(G,Ts,Rs):-maplist(rank(G),Ts,Rs).
```

Both combinators can be seen as a form of "structured recursion" that propagate a simpler operation guided by the structure of the data type. For instance, the size of a tree of type T is obtained as:

```
tsize1(Xs,N):-sumlist(Xs,S),N is S+1.

tsize(T,N) :- rank(tsize1,T,N).
```

Note also that `unrank` and `rank` work on trees in cooperation with `unranks` and `ranks` working on lists of trees.

We can now combine an anamorphism+catamorphism pair into an isomorphism `hylo` defined with `rank` and `unrank` on the corresponding hereditarily finite data types:

```
hylo(IsoName,iso(rank(G),unrank(F))):-call(IsoName,iso(F,G)).

hylos(IsoName,iso(ranks(G),unranks(F))):-call(IsoName,iso(F,G)).
```

Hereditarily Finite Sets Hereditarily Finite Sets will be represented as an Encoder for the tree type `T`:

```
hfs(Iso):-hylo(nat_set,Hylo),nat(Nat),
  compose(Hylo,Nat,Iso).
```

The `hfs` Encoder can now borrow operations from sets or natural numbers as follows:

```
hfs_succ(H,R):-borrow(nat_hfs,succ,H,R).
nat_hfs(Iso):-nat(Nat),hfs(HFS),with(Nat,HFS,Iso).
```

```
?- hfs_succ([],R).
R = [[]] ;
```

Otherwise, hylomorphism induced isomorphisms work as usual with our embedded transformation language:

```
?- as(hfs,nat,42,H).
H = [[[]], [[], [[]]], [[], [[[]]]]]
```

One can notice that we have just derived as a "free algorithm" Ackermann's encoding [8, 9], from Hereditarily Finite Sets to Natural Numbers:

$$f(x) = \texttt{if } x = \{\} \texttt{ then } 0 \texttt{ else } \sum_{a \in x} 2^{f(a)}$$

together with its inverse:

```
ackermann(N,H):-as(nat,hfs,N,H).
inverse_ackermann(H,N):-as(hfs,nat,H,N).
```

Hereditarily Finite Functions The same tree data type can host a hylomorphism derived from finite functions instead of finite sets:

```
hff(Iso) :-
  hylo(nat,Hylo),nat(Nat),
  compose(Hylo,Nat,Iso).
```

The `hff` Encoder can be seen as another "free algorithm", providing data compression/succinct representation for Hereditarily Finite Sets. Note, for instance, the significantly smaller tree size in:

```
?- as(hff,nat,42,H).
H = [[[]], [[]], [[]]]
```

As the cognoscenti might observe this is explained by the fact that `hff` provides higher information density than `hfs`, by incorporating order information that matters in the case of sequence and is ignored in the case of a set.

5 Pairing/Unpairing

A *pairing* function is an isomorphism $f : Nat \times Nat \to Nat$. Its inverse is called *unpairing*.

We will introduce here an unusually simple pairing function (also mentioned in [10], p.142).

The function `bitpair` works by splitting a number's big endian bitstring representation into odd and even bits.

```
bitpair(p(I,J),P):-
  evens(I,Es),odds(J,Os),
  append(Es,Os,Ps),set2nat(Ps,P).

evens(X,Es):-nat2set(X,Ns),maplist(double,Ns,Es).
```

```
odds(X,Os):-evens(X,Es),maplist(succ,Es,Os).
double(N,D):-D is 2*N.
```

The inverse function `bitunpair` blends the odd and even bits back together.

```
bitunpair(N,p(E,O)):-nat2set(N,Ns),
  split_evens_odds(Ns,Es,Os),
  set2nat(Es,E),set2nat(Os,O).

split_evens_odds([],[],[]).
split_evens_odds([X|Xs],[E|Es],Os):-
  X mod 2 =:= 0,E is X // 2,
  split_evens_odds(Xs,Es,Os).
split_evens_odds([X|Xs],Es,[O|Os]):-
  X mod 2 =:= 1,O is X // 2,
  split_evens_odds(Xs,Es,Os).
```

The transformation of the bitlists is shown in the following example with bitstrings aligned:

```
?-bitunpair(2008,R)
R = p(60,26)

% 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
%   60:[      0,    1,    1,    1,    1]
%   26:[   0,    1,    0,    1,    1   ]
```

We can derive the following Encoder:

```
nat2(Iso):-nat(Nat),
  compose(iso(bitpair,bitunpair),Nat,Iso).
```

working as follows:

```
?- as(nat2,nat,2008,Pair).
Pair = p(60, 26)

?- as(nat,nat2,p(60,26),N).
N = 2008
```

6 Directed Graphs and Hypergraphs

We will now show that more complex data types like digraphs and hypergraphs have extremely simple encoders. This shows once more the importance of compositionality in the design of our embedded transformation language.

6.1 Encoding Directed Graphs

We can find a bijection from directed graphs (with no isolated vertices, corresponding to their view as binary relations), to finite sets by fusing their list of ordered pair representation into finite sets with a pairing function:

```
digraph2set(Ps,Ns) :- maplist(bitpair,Ps,Ns).
set2digraph(Ns,Ps) :- maplist(bitunpair,Ns,Ps).
```

The resulting Encoder is:

```
digraph(Iso):-set(Set),
  compose(iso(digraph2set,set2digraph),Set,Iso).
```

working as follows:

```
?- as(digraph,nat,2008,D),as(nat,digraph,D,N).
D = [p(1, 1), p(2, 0), p(2, 1), p(3, 1), p(0, 2), p(1, 2), p(0, 3)],
N = 2008
```

6.2 Encoding Hypergraphs

Definition 1 *A hypergraph (also called* set system*) is a pair $H = (X, E)$ where X is a set and E is a set of non-empty subsets of X.*

We can easily derive a bijective encoding of *hypergraphs*, represented as sets of sets:

```
set2hypergraph(S,G) :- maplist(nat2set,S,G).
hypergraph2set(G,S) :- maplist(set2nat,G,S).
```

The resulting Encoder is:

```
hypergraph(Iso):-set(Set),
  compose(iso(hypergraph2set,set2hypergraph),Set,Iso).
```

working as follows

```
?- as(hypergraph,nat,2008,G),as(nat,hypergraph,G,N).
G = [[0, 1], [2], [1, 2], [0, 1, 2], [3], [0, 3], [1, 3]],
N = 2008
```

7 Applications

Besides their utility as a uniform basis for a general purpose data conversion library, let us point out some specific applications of our isomorphisms.

7.1 Combinatorial Generation

A free combinatorial generation algorithm (providing a constructive proof of recursive enumerability) for a given structure is obtained simply through an isomorphism from *nat*:

```
nth(Thing,N,X) :- as(Thing,nat,N,X).

stream_of(Thing,X) :- nat_stream(N),nth(Thing,N,X).

nat_stream(0).
nat_stream(N):-nat_stream(N1),succ(N1,N).
```

```
?- nth(set,42,S).
S = [1, 3, 5]

?- stream_of(hfs,H).
H = [] ;
H = [[]] ;
H = [[[]]] ;
H = [[], [[]]] ;
H = [[[[]]]] ;
H = [[], [[[]]]] ;
...
```

7.2 Random Generation

Combining `nth` with a random generator for *nat* provides free algorithms for random generation of complex objects of customizable size:

```
random_gen(Thing,Max,Len,X):-
  random_fun(Max,Len,Ns),
  as(Thing,fun,Ns,X).

random_fun(Max,Len,Ns):-
  length(Ns,Len),
  maplist(random_nat(Max),Ns).

random_nat(Max,N):-random(X),N is integer(Max*X).

?- random_gen(set,100,4,R).
R = [16, 39, 118, 168].

?- random_gen(fun,100,4,R).
R = [92, 60, 47, 76].

?- random_gen(nat,100,4,R).
R = 269599466671506412912446917138642189142104131263755679205821010411 52.

?- random_gen(hfs,4,3,R).
R = [[[]], [[], [[[]]]], [[[]], [[], [[]]]]]

?- random_gen(hff,4,3,R).
R = [[], [], [[]]]
```

Besides providing arbitrary precision random numbers as a "free algorithm" on top of a builtin limited precision floating point generator, one can see that this technique can be used to implement elegantly random test generators in tools like QuickCheck [11] without having to write data structure specific scripts.

7.3 Succinct Representations

Depending on the information theoretical density of various data representations as well as on the constant factors involved in various data structures, significant

data compression can be achieved by choosing an alternate isomorphic representation, as shown in the following examples:

```
?- as(hff,hfs,[[[]], [[], [[]]], [[], [[[]]]]],HFF).
HFF = [[[]], [[]], [[]]]

?- as(nat,hff,[[[]], [[]], [[]]],N).
N = 42
```

In particular, mapping to efficient arbitrary length integer implementations (usually C-based libraries), can provide more compact representations or improved performance for isomorphic higher level data representations. We can compare representations sharing a common datatype to conjecture about their asymptotic information density.

7.4 Experimental Mathematics

For instance, after defining:

```
length_as(Thing,X,Len) :-nat(Nat),
  call(Thing,T),with(Nat,T,Iso),
  fit_iso(length,Iso,X,Len).

sum_as(Thing,X,Len) :-nat(Nat),
  call(Thing,T),with(Nat,T,Iso),
  fit_iso(sumlist,Iso,X,Len).

size_as(Thing,X,Len) :-nat(Nat),
  call(Thing,T),with(Nat,T,Iso),
  fit_iso(tsize,Iso,X,Len).
```

one can conjecture that finite functions are more compact than sets asymptotically

```
?- length_as(fun,12345678901234567890123456789O,L).
L = 54

?- length_as(set,12345678901234567890123456789O,L).
L = 54

?- length_as(fun,12345678901234567890123456789O,L).
L = 54

?- sum_as(set,12345678901234567890123456789O,L).
L = 2690

?- sum_as(fun,12345678901234567890123456789O,L).
L = 43
```

and then observe that the same trend applies also to their hereditarily finite derivatives:

```
?- size_as(hfs,12345678901234567890123456789O,L).
L = 627

?- size_as(hff,12345678901234567890123456789O,L).
L = 91
```

7.5 A surprising "free algorithm": strange_sort

A simple isomorphism like `nat_set` can exhibit interesting properties as a building block of more intricate mappings like Ackermann's encoding, but let's also note a (surprising to us) "free algorithm" – sorting a list of distinct elements without explicit use of comparison operations:

```
strange_sort(Unsorted,Sorted):-
  nat_set(Iso),
  to(Iso,Unsorted,Ns),
  from(Iso,Ns,Sorted).


?- strange_sort([2,9,3,1,5,0,7,4,8,6],Sorted).
Sorted = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This algorithm emerges as a consequence of the commutativity of addition and the unicity of the decomposition of a natural number as a sum of powers of 2. The cognoscenti might notice that such surprises are not totally unexpected. In a functional programming context, they go back as early as Wadler's Free Theorems [12].

7.6 Other Applications

A fairly large number of useful algorithms in fields ranging from data compression, coding theory and cryptography to compilers, circuit design and computational complexity involve bijective functions between heterogeneous data types. Their systematic encapsulation in a generic API that coexists well with strong typing can bring significant simplifications to various software modules with the added benefits of reliability and easier maintenance. In a Genetic Programming context [13] the use of isomorphisms between bitvectors/natural numbers on one side, and trees/graphs representing HFSs, HFFs on the other side, looks like a promising phenotype-genotype connection. Mutations and crossovers in a data type close to the problem domain are transparently mapped to numerical domains where evaluation functions can be computed easily. In the context of Software Transaction Memory implementations (like Haskell's STM [14]), encodings through isomorphisms are subject to efficient shortcuts, as undo operations in case of transaction failure can be performed by applying inverse transformations without the need to save the intermediate chain of data structures involved.

8 Related work

This work can be seen as part of a larger effort to cover in a declarative programming paradigm some fundamental combinatorial generation algorithms along the lines of Donald Knuth's recent work [15].

The closest reference on encapsulating bijections as a data type is [16] and Connan Eliot's composable bijections Haskell module [17], where, in a more complex setting, Arrows [18] are used as the underlying abstractions. While our `Iso` data type is similar to the *Bij* data type in [17] and BiArrow concept of [16], the techniques for using such isomorphisms as building blocks of an embedded composition language centered around encodings as Natural Numbers are new.

Ranking functions can be traced back to Gödel numberings [1, 2] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [19, 15, 20, 21]. However the generic view of such transformations as hylomorphisms obtained compositionally from simpler isomorphisms, as described in this paper, is new.

Natural Number encodings of Hereditarily Finite Sets have triggered the interest of researchers in fields ranging from Axiomatic Set Theory and Foundations of Logic to Complexity Theory and Combinatorics [22–27]. Computational and Data Representation aspects of Finite Set Theory have been described in logic programming and theorem proving contexts in [9, 28].

Pairing functions have been used in work on decision problems as early as [29, 30]. A typical use in the foundations of mathematics is [31]. An extensive study of various pairing functions and their computational properties is presented in [32].

9 Conclusion

We have shown the expressiveness of Prolog as a metalanguage for executable mathematics, by describing encodings for functions and finite sets in a uniform framework as data type isomorphisms with a group structure. Prolog's higher order predicates and recursion patterns have helped the design of an embedded data transformation language. Using higher order combinators a simplified random testing mechanism has been implemented as an empirical correctness test. The framework has been extended with hylomorphisms providing generic mechanisms for encoding Hereditarily Finite Sets and Hereditarily Finite Functions. In the process, a few surprising "free algorithms" have emerged, including Ackermann's encoding from Hereditarily Finite Sets to natural numbers. We plan to explore in depth in the near future, some of the results that are likely to be of interest in fields ranging from combinatorics to data compression and arbitrary precision numerical computations. While we have not explicitly provided a complexity analysis for various isomorphisms, it is clear from the actual code that our transformations typically work in time and space proportional to the overall size of the representation. In particular, when natural numbers are the source or the target, complexity is $O(log(N))$, given that $log(N)$ is the bitsize of the representation of N.

10 Acknowledgment

The author thanks the anonymous reviewers of CICLOPS'08 for their constructive criticism, substantial comments and suggestions.

References

1. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
2. Hartmanis, J., Baker, T.P.: On simple goedel numberings and translations. In Loeckx, J., ed.: ICALP. Volume 14 of Lecture Notes in Computer Science., Springer (1974) 301–316
3. Warren, D.H.D.: Higher-order extensions to Prolog – are they needed? In Michie, D., Hayes, J., Pao, Y.H., eds.: Machine Intelligence 10. Ellis Horwood (1981)
4. Tarau, P.: Declarative Combinatorics: Isomorphisms, Hylomorphisms and Hereditarily Finite Data Types in Haskell (2008) http://arXiv.org/abs/0808.2953.
5. Gibbons, J., Lester, D., Bird, R.: Enumerating the rationals. Journal of Functional Programming **16**(4) (2006)
6. Hutton, G.: A Tutorial on the Universality and Expressiveness of Fold. J. Funct. Program. **9**(4) (1999) 355–372
7. Meijer, E., Hutton, G.: Bananas in Space: Extending Fold and Unfold to Exponential Types. In: FPCA. (1995) 324–333
8. Ackermann, W.F.: Die Widerspruchsfreiheit der allgemeinen Mengenlhere. Mathematische Annalen (114) (1937) 305–315
9. Piazza, C., Policriti, A.: Ackermann Encoding, Bisimulations, and OBDDs. TPLP **4**(5-6) (2004) 695–718
10. Pigeon, S.: Contributions à la compression de données. Ph.d. thesis, Université de Montréal, Montréal (2001)
11. Claessen, K., Hughes, J.: Testing monadic code with quickcheck. SIGPLAN Notices **37**(12) (2002) 47–59
12. Wadler, P.: Theorems for free! In: FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture, New York, NY, USA, ACM (1989) 347–359
13. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
14. Harris, T., Marlow, S., Jones, S.L.P., Herlihy, M.: Composable memory transactions. Commun. ACM **51**(8) (2008) 91–100
15. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
16. Alimarine, A., Smetsers, S., van Weelden, A., van Eekelen, M., Plasmeijer, R.: There and back again: arrows for invertible programming. In: Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM Press (2005) 86–97
17. Connan Eliot: Data.Bijections Haskell Module. http://haskell.org/haskellwiki/TypeCompose.
18. Hughes, J.: Generalizing Monads to Arrows Science of Computer Programming 37, pp. 67-111, May 2000.

19. Martinez, C., Molinero, X.: Generic algorithms for the generation of combinatorial objects. In Rovan, B., Vojtas, P., eds.: MFCS. Volume 2747 of Lecture Notes in Computer Science., Springer (2003) 572–581
20. Ruskey, F., Proskurowski, A.: Generating binary trees by transpositions. J. Algorithms **11** (1990) 68–84
21. Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Information Processing Letters **79** (2001) 281–284
22. Takahashi, M.o.: A Foundation of Finite Mathematics. Publ. Res. Inst. Math. Sci. **12**(3) (1976) 577–708
23. Kaye, R., Wong, T.L.: On Interpretations of Arithmetic and Set Theory. Notre Dame J. Formal Logic Volume **48**(4) (2007) 497–510
24. Abian, A., Lamacchia, S.: On the consistency and independence of some set-theoretical constructs. Notre Dame Journal of Formal Logic **X1X**(1) (1978) 155–158
25. Avigad, J.: The Combinatorics of Propositional Provability. In: ASL Winter Meeting, San Diego (January 1997)
26. Kirby, L.: Addition and multiplication of sets. Math. Log. Q. **53**(1) (2007) 52–65
27. Leontjev, A., Sazonov, V.Y.: Capturing LOGSPACE over Hereditarily-Finite Sets. In Schewe, K.D., Thalheim, B., eds.: FoIKS. Volume 1762 of Lecture Notes in Computer Science., Springer (2000) 156–175
28. Paulson, L.C.: A Concrete Final Coalgebra Theorem for ZF Set Theory. In Dybjer, P., Nordström, B., Smith, J.M., eds.: TYPES. Volume 996 of Lecture Notes in Computer Science., Springer (1994) 120–139
29. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
30. Robinson, J.: Finite generation of recursively enumerable sets. Proceedings of the American Mathematical Society **19**(6) (dec 1968) 1480–1486
31. Cégielski, P., Richard, D.: Decidability of the theory of the natural integers with the cantor pairing function and the successor. Theor. Comput. Sci. **257**(1-2) (2001) 51–77
32. Rosenberg, A.L.: Efficient pairing functions - and why you should care. International Journal of Foundations of Computer Science **14**(1) (2003) 3–17

Precise Garbage Collection in Prolog

Jan Wielemaker[1] and Ulrich Neumerkel[2]

[1] Universiteit van Amsterdam, The Netherlands
J.Wielemaker@uva.nl
[2] Technische Universität Wien, Austria
ulrich@complang.tuwien.ac.at

Abstract. In this paper we present a series of tiny programs that verify that a Prolog heap garbage collector can find specific forms of garbage. Only 2 out of our tested 7 Prolog systems pass all tests. Comparing memory usage on realistic programs dealing with finite datastructures using both poor and precise garbage collection shows only a small difference, providing a plausible explanation why many Prolog implementors did not pay much attention to this issue. Attributed variables allow for creating infinite lazy datastructures. We prove that such datastructures have great practical value and their introduction requires 'precise' garbage collection. The Prolog community knows about three techniques to reach at precise garbage collection. We summarise these techniques and provide more details on scanning virtual machine instructions to infer reachability in a case study.

1 Introduction

All modern Prolog systems come with a heap garbage collector, no longer limiting the programmer to revert to failure driven loops or **findall/3** to free unneeded memory through backtracking. For this article, we define a 'precise' garbage collector as a garbage collector that reclaims all data that can no longer be reached considering all possible execution paths from the current state without considering semantics. I.e. in `1==2, A=ok`, A is unreachable due to the semantics of ==/2, but we consider all parts of a conjunction reachable and therefore A is considered reachable. Our survey of 7 popular Prolog systems (Sect. 3) reveals that only two satisfy this definition. We compared the memory requirements between the poorest and best performance of GC on 5 very different real-world programs (Tab. 2). The comparison indicates that precise GC is unimportant for many programs, which provides a plausible explanation why precise GC is not widespread.

Precise GC becomes important for processing infinite datastructures, in this case distinguished from *cyclic* structures. A truly infinite structure clearly never fits into finite physical memory. We are concerned with datastructures that grow due to further instantiation while (older) parts of the datastructure become unreachable after processing and can be reclaimed by the garbage collector. A typical example is processing input using a list: the list is expanded as new

input becomes available, while the head of the list becomes unreachable after being processed deterministically. This approach is used in [1], where infinite lists are used for communication between concurrent processes. Similar consideration motivated improvements in functional languages [2].

This article is organised as follows. First, in Sect. 2 we make a case for the practical value of infinite lazy datastructures and the requirement of precise GC. In Sect. 3 we identify possible leaks and test 7 Prolog implementations for them, 5 of which exhibit two or more leaks. This is followed by a survey of known existing techniques to reach precise GC and the description and evaluation of a case study adding precise GC to SWI-Prolog.[3]

2 A case for infinite lazy datastructures: pure input

Prolog DCG and other parsing techniques are based on processing lists. Unfortunately, the data that needs to be parsed is often provided as a Prolog stream that accesses data from the outside world. This problem has been identified long ago and many implementations of DCG provide a hook 'C'/3 to read an input character. This hook is of little practical use, notably due to the poor combination of non-determinism and side-effects. The current proposal for an ISO standard on DCGs [3] no longer mentions 'C'/3. Fortunately, extended unification [4–7] using attributed variables as found in many modern Prolog systems provides a straightforward mechanism to remedy this problem.

Figure 1 presents the simple algorithm to apply a grammar rule on input from a file as it appears in the SWI-Prolog library `pure_input.pl`. Besides standard ISO predicates, the implementation depends on **freeze**(*Var, Goal*), which delays *Goal* until *Var* becomes instantiated (coroutining); **call_cleanup**(*Goal, Cleanup*) which allows for closing the input handle when *Goal* becomes inaccessible due to deterministic termination, an exception or pruning of a choicepoint and finally **read_pending_input**(*Handle, Head, Tail*) which reads a block of buffered input into the difference-list `Head\Tail`. Freeze or a substitute is available in all systems with attributed variables. Call_cleanup is available in multiple Prolog implementations and has been discussed for inclusion in the upcoming revision of Part I of the ISO Prolog standard.[4] A block-read operation is not defined by the ISO standard but trivial to implement while it provides a very significant speedup (12× in SWI-Prolog 5.6.59) because it only needs to validate and lock the stream handle once.

The **phrase_from_file**(*:DCG, +File*) definition in Fig. 1 allows for applying an arbitrary non-deterministic DCG completely transparently on the content of a file while, given precise GC, the memory usage is independent from the size of this file. We compared the use of a DCG on a file with a carefully handcrafted program to count words in a text-file. We summarise the key results in the table below and conclude that the DCG version is much easier to read and very comparable in performance.

[3] http://www.swi-prolog.org

[4] Inclusion is stalled because the precise semantics prove hard to describe.

```
read_to_input_stream(Handle, Pos1, Stream0) :-
    set_stream_position(Handle, Pos1),
    (   at_end_of_stream(Handle)
    ->  Stream0 = []
    ;   read_pending_input(Handle, Stream0, Stream1),
        stream_property(Handle, position(Pos2)),
        freeze(Stream1, read_to_input_stream(Handle, Pos2, Stream1))
    ).

phrase_from_file(Phrase, File) :-
    open(File, read, Handle),
    stream_property(Handle, position(Pos)),
    freeze(Stream, read_to_input_stream(Handle, Pos, Stream)),
    call_cleanup(phrase(Phrase, Stream), close(Handle)).
```

Fig. 1. Implementation of input streams.

	traditional	**DCG on file**
Code size (lines)	31	22
Time (sec., 25MB file)	16.1	17.1
GC time (sec.)	0.9	1.4

From the above, we conclude that infinite (lazy) terms have great practical value and it is therefore desirable that garbage collection is capable of reclaiming the no-longer-accessible part of the term.

3 State of the art

Can pure input as described above be used in current Prolog systems with coroutining? We reviewed 7 Prolog implementations. The first obvious requirement is that there is no memory leak after a deterministic wakeup of a delayed goal (Sect. 3.1). The other requirements are about reclaiming unneeded parts of the input list within and-control and or-control. I.e. we must be able to create a list of arbitrary size if there are no references to the entire list. The simplest form is the test below. Predicate **f/1** builds a list, but as nobody uses it, GC reclaims it and **run/0** runs forever in constant space.

```
run :- f(_).
f([f|X]) :- f(X).
```

This is the simplest case, where the initial list is created through a singleton variable. In WAM-based systems with registers, the list resides in a register that is overwritten in each recursion. On virtual machines such as the ZIP [8, 9] and ATOAM [10] that pass arguments over the stack, last-call optimization overwrites the arguments, making the head inaccessible.

We will now go systematically through requirements to deal with infinite (lazy) datastructures. The first property validates that deterministic instantiation of an attributed variable does not leak. The remaining properties validate that various scenarios where the head of the list becomes inaccessible are detected by the garbage collector. Each test case considers a situation that requires special attention in one or more virtual machines, based on our understanding of, notably, the WAM and ZIP. As the number of possible virtual machines is unbounded, it is not possible to be sure that these cases cover all cases in all possible virtual machines. Each property is accompanied by a program that must run forever in constant space. A test is considered 'failed' if the system aborts or memory usage exceeds 1Gb. The given programs are very simple, using a fact **dummy/1** to pretend access to a variable. We assume that **dummy/1** cannot be optimized away by the compiler, otherwise a more complex replacement is needed.

3.1 Property 1: Permanent removal of attributes

Attributed variables that have been unified deterministically with a non-variable term must be reclaimed completely. This property can be tested using the program below. It creates delayed goals and executes them through deterministic binding. Note that for most constraint solvers, complete reclamation of attributed variables is not strictly necessary. Most CLP(FD) programs are concerned with finding solutions nondeterministically via a labeling procedure, thus most volatility stems from backtracking and not from forward recursion.

```
run :- run(_).
run(X) :- freeze(X, dummy(X)), X = 1, run(T).
dummy(_).
```

3.2 Property 2: And-control (head variables)

Variables appearing in the head of a rule and in the body must be discarded as soon as possible. We test this using the following which, like the previous test, must run forever in bounded memory. The call to **dummy/2** ensures $L0$ is not made inaccessible due to last call optimization.

```
run :- run(_,_).
run(L0, L) :- f(L0, L1), dummy(L1, L).
f([g|X], Y) :- f(X, Y).
dummy(Xs, Xs).
```

3.3 Property 3: And-control (existential variables)

Existential variables that occur in several goals, but not the last one. Ideally such variables should be covered by environment trimming [11] in the WAM. Careful environment trimming avoids more complex treatment.

```
run :- run(_,_).
run(L0, L) :- dummy(L0, L1), f(L1, L2), dummy(L2, L).
f([f|X], Y) :- f(X, Y).
dummy(Xs, Xs).
```

3.4 Property 4: Or-control

Or-control covers the case where a variable is only accessible from a choicepoint. A well behaved garbage collector will reset such variables and discard their current value (early-reset, [12]). This situation arises in disjunctions in grammars. E.g. `(...,"a"|...,"b")`, where .../0 is defined to match an unbounded string.

```
run :- run(_).
run(X) :- f(X).
run(X) :- X == [].
f([f|X]) :- f(X).
```

3.5 Property 5: Branching inside a clause

Branching (*A*;*B* and *If*`->`*Then*`;`*Else*) using different ordering of the variables in both branches cannot be handled optimally with the WAM environment trimming as the branches require different environment layout. This test is only of interest for systems that open code disjunctions, avoiding an auxiliary internal definition.

```
run(Z) :- p(_,_,Z).
p(X,Y,Z) :- (Z > 0 ->  f(X), g(Y), dummy ; g(Y), f(X), dummy).
f([f|X]) :- f(X).
g([g|X]) :- g(X).
dummy.
```

3.6 Conclusion from our survey

	1	2	3	4	5_0	5_1	VM
SICStus 3.12.5	ok	ok	ok	ok	ok	ok	WAM
Ciao 1.10p8	ok	ok	ok	ok	ok	ok	WAM
YAP 5.0.1	n	ok	ok	ok	ok	n	WAM
ECLiPSe 5.10	ok	ok	n	ok	n	n	WAM
SWI 5.6.54	n	n	n	n	n	n	ZIP
BProlog 7.1	n	ok	n	n	n	n	ATOAM
XSB 3.1	n	ok	n	n	n	n	WAM

Table 1. Evaluation of GC in some popular Prolog systems with coroutining. The numbers correspond to the properties. Property 5 is tested for both branches.

In the above sections we have provided tests for the main properties of Prolog coroutining and garbage collection needed to be able deal with infinite lazy datastructures. The results are shown in Tab. 1. As detailed descriptions of GC in these systems is either not in the literature or the description is likely to be outdated and we do not have access to the source code of all these systems we have not examined *why* tests succeed or fail. We merely conclude that precise GC has not been given much attention by the respective developers. Table 2 justifies this behaviour in the absence of infinite datastructures.

To the best of our knowledge, SICStus[5] and the derived Ciao system [1] reach a precise result using WAM registers, environment trimming and the implementation of in-clause alternative execution paths using anonymous predicates. The YAP VM uses virtual machine instructions for in-clause alternative execution paths, which cannot be handled perfectly with only environment trimming as explained in Sect. 4. It is hard to explain the behaviour of the other systems.

Our study started with providing a pure input library for SWI-Prolog. SWI-Prolog *design* was ok for property 1, but the implementation was proven flawed. As the SWI-Prolog virtual machine passes arguments over the stack and does not use environment trimming, it failed on all test.

4 Related work on data reachability in Prolog

Prolog systems discard data during backtracking. During forwards execution, discarding data is achieved by the heap garbage collector. The garbage collector preserves all data that is accessible through a set of *root pointers* [13]. The precise set of root pointers depends on the Virtual Machine (VM) architecture, where we distinguish between VMs that pass arguments in registers (WAM) and VMs that pass arguments using the stack (ZIP, ATOAM). The current stack frame and choice point are always root pointers. Registers and global variables are other examples. There are several mechanisms by which data becomes inaccessible from the set of root pointers that are part of the normal Prolog (forward) execution:

- Temporary variables allocated in registers become inaccessible when they are overwritten.
- Arguments (on machines passing arguments over the stack) and environment slots become inaccessible if the frame is discarded due to last-call optimization.
- Environment trimming (see below) shrinks the environment, discarding unneeded parts as the execution of the clause progresses.

Environment trimming [11] allocates variables in the environments ordered by the last subgoal that references the variable. Each call to a subgoal has an additional numeric argument that states that the first N variables of the

[5] www.sics.se/sicstus/ explained to one of the authors by Mats Carlsson.

environment are still valid. Together with registers for argument passing and last-call optimization, environment trimming reaches a precise result if there are *no alternative execution paths* in the VM instructions. This implies that disjunction (*A*;*B*) and *If*`->`*Then*`;`*Else* must be translated into pure (anonymous) predicates with some additional machinery to deal with proper scoping of the cut. This technique is used by SICStus Prolog and Ciao (see Sect. 3.6.

Many virtual machines realise disjunction and if-then-else using branch instructions in the VM. As different subgoal ordering in the alternate execution paths may require different ordering of variables in the environment (Sect. 3.5), there is no longer a perfect order and garbage collection that scans the entire environment will mark data that is no longer reachable because there is no instruction that refers to some variable. Table 1 suggests this is the status in YAP 5.0.1.

Environment trimming cannot deal with arguments that are passed over the stack as their order is determined by the calling convention and, analogous to in-clause branching, different clauses of the predicate generally require a different ordering.

VMs that pass arguments over the stack as well as VMs that use branching instructions to code in-clause alternate execution paths need additional measures to regain precise GC. Two techniques to achieve this have been part of the Prolog folklore for some time.[6] One scans the VM instructions from the continuation points to find the accessible variables. It was used by old versions of BIM Prolog. With native code this became very hard to maintain. The other uses compiler generated bitmaps for each possible continuation point that represent all reachable variables. This is used by BIM Prolog and hProlog.

In systems based on 'Binary Prolog' [14], continuations take the place of environments. They are represented by ordinary Prolog terms and therefore profit from the same data representations [15]. Garbage collection in such systems [16] do not require any special treatment. On the other hand, Binary Prolog requires more space for representing variables within continuations than traditional implementations. Every occurrence of a variable is now represented separately, while traditional environments represent each variable only once.

5 Our case study: SWI-Prolog

SWI-Prolog is based on the ZIP VM which passes arguments over the stack and uses branching instructions inside a clause. Like most today's Prolog systems, the VM is emulated. We briefly examine these properties under the assumption that the optimal choice depends on the specific setting: desired performance, portability, transparency for debugging, simplicity and speed of the compiler.

- *Argument passing*
  The use of registers for argument passing as the WAM has some clear advantages. It keeps the environment small and simplifies last-call optimization.

[6] according to Bart Demoen

This comes at a price: the compiler is more complicated and it is harder to provide a (graphical) debugger that provides access to variables in the parent frames.

- *Branching instructions*
  Using branching instructions to code disjunction and if-then-else prohibits precise trimming of the environment as we have seen in Sect. 4. On the other hand, execution is generally faster as no environment needs to be created for the anonymous predicates that otherwise replace different in-clause execution paths. We have no information on the implementation effort associated with these approaches.
- *Emulated VM vs. native code*
  An emulated VM is clearly easier to implement and if the VM is written in a portable language, portability of the system comes for free. In addition, it allows for simple decompilation [17] and simplifies two tasks in GC: identify not-yet-initialized variables in the environment and identify variables that can still be accessed from a given program counter (PC) location. SICStus has dropped native code in release 4[7]

The above observations make it clear that scanning VM instructions to remedy the reachability problem is the most obvious approach for SWI-Prolog. Because most todays Prolog implementation use an emulated VM and Tab. 1 proves that several systems still need to realise precise GC we believe a description of our case study will help persuading other implementors to implement precise GC and will help them to take the correct decisions right away.

6 Implementation

The SWI-Prolog VM differs considerable from the much more widely adopted WAM. SWI-Prolog's garbage collector however closely follows the SICStus Prolog garbage collector, which is described excellently in [12]. The fact that our GC closely follows a GC for a WAM-based system gives some confidence that our findings are applicable to a wider range of Prolog implementations. This section only concentrates on the modifications to the algorithm described in [12] and cannot be understood without detailed understanding of this paper.

Our modifications only affect the *marking* phase of GC. The modified algorithm is provided in pseudo code in Fig. 2 and discussed below. Added lines and deleted lines are marked with `+/-` at the start of the line.

First, initialize_and_mark() marks all data that is accessible from the continuation PC and at the same time initialises variables for which it finds a 'first-access' instruction, finishing the initialization of the environment. All environments are marked as 'seen'. This is the same as in [12], except

[7] Mats Carlsson has confirmed that SICStus 4 uses VM code scanning to deal with uninitialized variables in the environment. See also http://www.sics.se/sicstus/docs/latest4/pdf/relnotes.pdf

```
procedure mark_environments(env, PC)
   while ( env )
       if ( not_seen(env) )
           set_seen(env)
-          initialize(env, PC)
+          initialize_and_mark(env, PC)
           PC = env->PC
           env = env->parent
       else
+          mark(env, PC)
           return

procedure mark_choices(ch)
    env = ch->environment
    early_reset_trail()
    while ( ch )
        if ( pc_choice(ch) )
            mark_environments(env, ch->PC)
        else if ( alt_clause(ch) )
+           unmarked = count_unmarked_arguments(env)
+           while ( unmarked > 0 && clause )
+               mark_arguments(env, clause->code)
+               clause = next_visible(clause)
            if ( not_seen(env) )
                set_seen(env)
                mark_environments(env->parent, env->PC),
+       else if ( foreign_choice(ch) )
+           mark_all_arguments(env);

procedure mark_stacks(env, ch, PC)
    mark_environments(env, PC)
    mark_choices(ch)
```

Fig. 2. Pseudo code for the marking algorithm

- Mark variables in the environment that are referred to by instructions reachable from the PC instead of all variables in the environment.
- If we find a reference pointer to a parent environment, we mark the pointer and continue marking the referenced destination. In the traditional algorithm the variable in the parent is marked if we mark the parent environment. Now we must cover the case where the corresponding variable is accessed in this frame, but not in the parent frame.
- When called from mark_choices(), that marking is normally aborted if the frame has already been seen. Now we must continue to mark the first seen environment as this continuation may have a different PC and thus access to different variables. There is no need to continue with the parent frame as that has already been marked using the same PC.

Marking choicepoints is also similar to [12]. It resets trail entries that point to garbage cells (early reset, dealing with property 4) and then marks the associated environment. As SWI-Prolog passes arguments over the stack, if an alternate clause is encountered we need to keep all arguments that are used by the remainder of the clause list (possibly reduced due to indexing). Simply scanning the code of each clause could scan a lot of code on, for example, predicates with many facts. We avoid this by computing the number of unmarked arguments and abort the scan if all arguments are marked. Note that a clause without singleton variables in the head accesses all arguments and thus stops the search. Ground facts are a common example. Finally, as we have no information on how a foreign predicate accesses its arguments we must mark all arguments as accessible.

Sweeping an environment has been changed slightly. In [12], all heap references in the environment are inserted into relocation chains. Now, we first check whether the heap reference is marked. If so, we put it into a relocation chain as before, otherwise we assign the atom `'<garbage_collected>'` to the variable. This ensures consistency of the environment variable after heap relocation and is needed by the debugger if execution switches from normal mode to debug mode after a user interrupt or explicit call to **trace/0** inside code running in no-debug mode. In such cases, the debugger may show arguments of parent goals that were executed in normal mode as `'<garbage_collected>'` and the graphical debugger may show variables from the environment this way.

Note that if the program was started in debug mode, all data remains accessible through extra 'debug' choicepoints that also facilitate 'retry' at goals that were started deterministically. Figure 6 illustrates the problem using an explicit call to **garbage_collect/0** and **trace/0**. Explicitly calling **trace/0** is common practice to start debugging in a very specific state. The explicit call to **garbage_collect/0** is there only to illustrate what happens if GC was invoked at that specific point, while the system still operates in no-debug mode.

```
test :-
        read_line_to_codes(user_input, List),
        all_spaces(List).

all_spaces([]).
all_spaces([0' |T]) :- !, all_spaces(T).
all_spaces(_) :- garbage_collect, trace, fail.

1 ?- test.
|:   xx.
   Call: (11) fail ? goals
     [11] fail
     [10] all_spaces('<garbage_collected>')
      [1] '$toplevel'
   Call: (11) fail ?
```

Fig. 3. The debugger showing a garbage collected argument

7 Evaluation

Our evaluation considers four aspects: time, space, implementation effort and maintenance. In the tradition of SWI-Prolog, we consider mainly real and large applications. We selected the following applications because of diversity, size and the amount of garbage collection involved: CHAT80 (Pereira & Warren, 1986) running its test-suite in a forward chaining loop to force GC, Back52 (Thomas Hoppe et all., 1993) running its test suite, CHR compiler (Tom Schrijvers) compiling itself, k123.pl (Peter Vanbroekhoven) and pgolf.pl (Mats Carlsson).

The results are shown in Tab. 2. The first set of columns describe the overall timing, the last set describes characteristics of the code scanning version only and is discussed in Sect. 7.3. All timings are executed on an AMD Athlon X2 5400+; 64-bit Linux 2.6 using the 64-bit development version of SWI-Prolog based on 5.6.55. Reported time is in seconds. Frequency stepping was disabled during the tests.

7.1 Time evaluation

Table 2 shows that the overall execution time is only slightly affected by our changes. Note that the logic to trigger GC depends on the amount of memory that is accessible after the previous GC and therefore different effectiveness of GC leads to unpredictable overall behaviour of the program in terms of time and number of garbage collections.

We obtained a detailed breakdown of the garbage collector using valgrind [18] with the *callgrind* tool and *kcachegrind* to explore the results. The overhead of analysing instructions is approximately 1% of the garbage collector marking

Test	Time	#GC	GCLeft	GCTime	AvgScan	AvgCls	AvgInstr
Without code scanning							
k123	8.88	164	1,594,534	1.35			
chat80	2.56	109	18,661	0.10			
back52	2.31	406	5,589	0.17			
pgolf	13.22	53	7,328,689	3.44			
chr	6.41	36	3,466,387	1.17			
With code scanning							
k123	8.71	209	1,111,646	1.20	1.51	0.09	12.10
chat80	2.42	111	12,301	0.08	1.68	0.60	14.52
back52	2.21	420	3,360	0.15	1.56	0.12	11.19
pgolf	11.06	53	7,151,304	3.19	1.42	0.01	12.37
chr	6.29	38	3,265,471	1.15	1.91	0.32	14.52

Table 2. Effects of code scanning. *Time* is the total execution time (including GC time); *#GC* the number of garbage collections; *GCLeft* the average amount of memory (heap+trail) immediately after GC and *GCTime* the time spent on GC. *AvgScan* is the average number of continuation points that must be explored for an environment; *AvgCls* the average number of additional clauses scanned; *AvgInstr* the average number of instructions scanned before reaching the end of the clause.

time. These timing are slightly distorted because gcc's inline function optimization needs to be disabled to analyse the breakdown of execution time over the various functions.

7.2 Space evaluation

Our approach based on marking accessible data by scanning the VM instructions obviously reaches the 'precise' result as defined in the introduction for the heap and trail stack. It does not provide the optimal result for the environment stack. Only the approach as taken by SICStus is optimal here in the sense that the stack contains no variables that are not accessible, while using our marking approach the variables remain in the environment, bound to `'<garbage_collected>'`. Environment stack usage is in practice rarely a bottleneck and our deficiency is a constant amount rather than the difference between finite and infinite stack usage.

Table 2 also explains why precise GC is not widespread. Except for memory usage of the k123.pl test, we find no noticeable differences in the memory usage after GC. The k123 program is a small program (75 lines after cleanup of unreachable code). The central predicate **mmul/3** in Fig. 4 is deterministic. Lacking temporary registers and environment trimming, the old SWI-Prolog, could not dispose the intermediate matrices.

Implementation and maintenance Only the code for marking environments and clearing uninitialised variables was extended from originally 150 lines (C), to 557 including comment and debugging statements. Total implementation effort was

```
mmul(M, M6) :-
        mmul(M, M, M1),   mmul(M1, M1, M2), mmul(M2, M2, M3),
        mmul(M3, M3, M4), mmul(M4, M4, M5), mmul(M5, M5, M6).
```

Fig. 4. Main routine of k123.pl

4 days. One of the problems associated with VM instruction interpretation is maintenance that results from changing the instruction set. SWI-Prolog maintains information of the instruction format for each instruction. This is used to list VM instructions, deal with saving and loading and simplifies VM instruction scanning as it allows enumerating the instructions using a generic loop. Four instructions have variable length data associated with them (packed string and unbounded integer) and need (uniform) special attention.

In addition to the generic code walking, 36 out of 89 instructions require special attention as described in table Tab. 3. The table states the number of instructions the marking algorithm needs to understand, the number of groups of instructions that require different treatment (especially the variable accessing functions are often handled using the same code) and the number of lines of C-code involved.

Description	instructions	groups	lines
Identify flow control	6	5	44
Realise initialization of uninitialised variables	3	1	10
Identify variable access for marking (body)	14	6	30
Identify variable access for marking (head)	13	6	27

Table 3. Statistics on interpreting VM instructions

7.3 Discussion

Before we arrived at the current implementation we had two worries: prohibitive costs of multiple scans of the same code from different continuations and prohibitive scans of code from multiple clauses to identify the still-reachable arguments. Column *AvgCls* of Tab. 2 (page 135) indicates that scanning alternative clauses is cheap, while the value of equal GC behaviour between in-clauses disjunctions and alternative clauses is obvious.

Our first prototype avoided multiple scans of the same code from different continuations. Not correctly dealing with early-reset, this code was flawed and abandoned. Nevertheless, it executed the above programs correctly and we obtained statistics on its effectiveness. On the above test cases, multiple scans increase the number of scanned instructions by 0, 58%, 7%, 7% and 3% (same order as Tab. 2). As the scanning itself is responsible for less than 1% of the

time of the mark phase, it is considered neglectable. This conclusion can also be drawn from column *AvgScan* and *AvgInstr* together with the 1% time spent on code scanning.

8 Conclusions

We have defined a set of five properties, each of which accompanied with a very simple test case, that must be satisfied to deal with infinite (lazy) datastructures in Prolog. We have proven that such datastructures are of significant practical value as they can be used to realise processing a repositionable input stream using the full power of non-deterministic grammar rules (DCGs). The majority of Prolog implementations that provide the required attributed variables to realise a lazy datastructure does not provide the required precise garbage collector. Precise GC can be realised using a VM that uses registers to pass arguments, implements environment trimming and codes in-clauses disjunction using anonymous predicates. Our case study indicates that other virtual machines can be remedied by scanning virtual machine instructions to identify reachable variables in the environment. This technique is viable for any Prolog system based on emulating virtual machine instructions. Next to supporting infinite datastructure, the approximately 1% extra cost in the marking phase is more than compensated for in the compacting phase of the garbage collector.

The current version of SWI-Prolog is shipped with the described enhancements to the garbage collector and a library to use DCGs on repositionable input streams.

Acknowledgements

We would like to thank Mats Carlsson for explaining to one of the authors how the reachability problem is solved in SICStus Prolog, Bart Demoen for explaining some folklore and the bitmap technique and Paulo Moura for investigating the state of the art in some popular Prolog systems as shown in Tab. 1. Vitor Santos Costa has confirmed property 1 for YAP, which is planned to be fixed soon.

References

1. Hermenegildo, M.V., Gras, D.C., Carro, M.: Using attributed variables in the implementation of concurrent and parallel logic programming systems. In: ICLP. (1995) 631–645
2. Wadler, P.L.: Fixing some space leaks with a garbage collector. Software Practice and Experience **17** (1987) 595–609
3. et. al., P.M.: Prolog (2006) ISO/IEC DTR 132113:2006.
4. Neumerkel, U.: Extensible unification by metastructures. In Bruynooghe, M., ed.: Proceedings of META90, Workshop on Meta-Programming in Logic, Leuven, Belgium (1990)
5. Huitouze, S.L.: A new data structure for implementing extensions to prolog. In: PLILP. Volume 456., Springer-Verlag (1990) 136–150 LNCS 456.

6. Holzbaur, C.: Metastructures versus attributed variables in the context of extensible unification. In: PLILP. Volume 631., Springer-Verlag (1992) 260–268 LNCS 631.
7. Demoen, B.: Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium (2002) URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.
8. Bowen, D.L., Byrd, L.M., Clocksin, W.: A portable Prolog compiler. In Pereira, L.M., ed.: Proceedings of the Logic Programming Workshop 1983, Lisabon, Portugal, Universidade nova de Lisboa (1983)
9. Neumerkel, U.: The binary WAM, a simplified Prolog engine. Technical report, Technische Universität Wien (1993) http://www.complang.tuwien.ac.at/ulrich/papers/PDF/binwam-nov93.pdf.
10. Zhou, N.F.: Garbage collection in B-Prolog. In: Proc. of the First Workshop on Memory Management in Logic Programming Implementations. (2000)
11. Castro, L.F., Costa, V.S.: Understanding memory management in Prolog systems. In Codognet, P., ed.: ICLP. Volume 2237 of Lecture Notes in Computer Science., Springer (2001) 11–26
12. Appleby, K., Carlsson, M., Haridi, S., Sahlin, D.: Garbage collection for Prolog based on WAM. Communications of the ACM **31** (1988) 719–741
13. Bekkers, Y., Ridoux, O., Ungaro, L.: Dynamic memory management for sequential logic programming languages. In: Workshop on Memory Management. (1992) LNCS 627.
14. Tarau, P., Boyer, M.: Elementary logic programs. In: PLILP, Springer-Verlag (1990) 365–381 LNCS 456.
15. Tarau, P., Neumerkel, U.: A novel term compression scheme and data representation in the binwam. In: PLILP, Springer-Verlag (1994) 73–87 LNCS 844.
16. Demoen, B., Tarau, P., Engels, G.: Segment order preserving copying garbage collection for wam based prolog. In: Symposion on Applied Computing (SAC), ACM (1996) 380–386
17. Buettner, K.A.: Fast decompilation of compiled prolog clauses. In Shapiro, E.Y., ed.: ICLP. Volume 225 of Lecture Notes in Computer Science., Springer (1986) 663–670
18. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In Ferrante, J., McKinley, K.S., eds.: PLDI, ACM (2007) 89–100

Pairing Functions, Boolean Evaluation and Binary Decision Diagrams in Prolog

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
E-mail: tarau@cs.unt.edu

Abstract. A "pairing function" J associates a unique natural number z to any two natural numbers x,y such that for two "unpairing functions" K and L, the equalities K(J(x,y))=x, L(J(x,y))=y and J(K(z),L(z))=z hold. Using pairing functions on natural number representations of truth tables, we derive an encoding for Binary Decision Diagrams with the unique property that its boolean evaluation faithfully mimics its structural conversion to a a natural number through recursive application of a matching pairing function. We then use this result to derive *ranking* and *unranking* functions for BDDs and reduced BDDs. The paper is organized as a self-contained literate Prolog program, available at `http://logic.csci.unt.edu/tarau/research/2008/pBDD.zip`.
Keywords: logic programming and computational mathematics, pairing/unpairing functions, encodings of boolean functions, binary decision diagrams, natural number representations of truth tables

1 Introduction

This paper is an exploration with logic programming tools of *ranking* and *unranking* problems on Binary Decision Diagrams. The practical expressiveness of logic programming languages (in particular Prolog) are put at test in the process. The paper is part of a larger effort to cover in a declarative programming paradigm, arguably more elegantly, some fundamental combinatorial generation algorithms along the lines of [1]. However, our main focus is by no means "yet another implementation of BDDs in Prolog". The paper is more about fundamental isomorphisms between logic functions and their natural number representations, in the tradition of [2], with the unusual twist that everything is expressed as a literate Prolog program, and therefore automatically testable by the reader. One could put such efforts under the generic umbrella of an emerging research field that we would like to call *executable theoretical computer science*. Nevertheless, we also hope that the more practically oriented reader will be able to benefit from this approach by being able to experiment with, and reuse our Prolog code in applications.

The paper is organized as follows: Sections 2 and 3 overview efficient evaluation of boolean formulae in Prolog using bitvectors represented as arbitrary length integers and Binary Decision Diagrams (BDDs).

Section 4 discusses classic pairing and unpairing operations and introduces pairing/unpairing predicates acting directly on bitlists.

Section 5 introduces a novel BDD encoding (based on our unpairing functions) and discusses the surprising equivalence between boolean evaluation of BDDs and the inverse of our encoding, the main result of the paper.

Section 6 describes *ranking* and *unranking* functions for BDDs and reduced BDDs.

Sections 7 and 8 discuss related work, future work and conclusions.

The code in the paper, embedded in a literate programming LaTeX file, is entirely self contained and has been tested under *SWI-Prolog*.

2 Parallel Evaluation of Boolean Functions with Bitvector Operations

Evaluation of a boolean function can be performed one value at a time as in the predicate `if_then_else/4`

```
if_then_else(X,Y,Z,R):-
  bit(X),bit(Y),bit(Z),
  ( X==1->R=Y
  ; R=Z
  ).

bit(0).
bit(1).
```

resulting in a *truth table*[1]

```
?- if_then_else(X,Y,Z,R),write([X,Y,Z]:R),nl,fail;nl.
[0, 0, 0]:0
[0, 0, 1]:1
[0, 1, 0]:0
[0, 1, 1]:1
[1, 0, 0]:0
[1, 0, 1]:0
[1, 1, 0]:1
[1, 1, 1]:1
```

Clearly, this does not take advantage of the ability of modern hardware to perform such operations one word a time - with the instant benefit of a speed-up proportional to the word size. An alternate representation, adapted from [1] uses integer encodings of 2^n bits for each boolean variable $X_0, \ldots, X_{n-1}$. Bitvector operations evaluate all value combinations at once.

[1] One can see that if the number of variables is fixed, we can ignore the bitsrings in the brackets. Thus, the truth table can be identified with the natural number, represented in binary form by the last column.

Proposition 1 *Let x_k be a variable for $0 \leq k < n$ where n is the number of distinct variables in a boolean expression. Then column k in the matrix representation of the inputs in the the truth table represents, as a bitstring, the natural number:*

$$x_k = (2^{2^n} - 1)/(2^{2^{n-k-1}} + 1) \tag{1}$$

For instance, if $n = 2$, the formula computes $x_0 = 3 = [0, 0, 1, 1]$ and $x_1 = 5 = [0, 1, 0, 1]$.

The following predicates, working with arbitrary length bitstrings are used to evaluate variables x_k with $k \in [0..n-1]$ with formula 1 and map the constant boolean function `1` to the bitstring of length 2^n, `111..1`, representing $2^{2^n} - 1$

```
% maps variable K in [0..NbOfBits-1] to Xk
var_to_bitstring_int(NbOfBits,K,Xk):-
  all_ones_mask(NbOfBits,Mask),
  var_to_bitstring_int(NbOfBits,Mask,K,Xk).

var_to_bitstring_int(NbOfBits,Mask,K,Xk):-
  NK is NbOfBits-(K+1),
  D is (1≪(1≪NK))+1,
  Xk is Mask//D.

% represents constant 1 as 11...1 build with NbOfBits bits
all_ones_mask(NbOfBits,Mask):-Mask is (1≪(1≪NbOfBits))-1.
```

We have used in `var_to_bitstring_int` an adaptation of the efficient bitstring-integer encoding described in the Boolean Evaluation section of [1]. Intuitively, it is based on the idea that one can look at n variables as bitstring representations of the n columns of the truth table.

Variables representing such bitstring-truth tables (seen as *projection functions*) can be combined with the usual bitwise integer operators, to obtain new bitstring truth tables, encoding all possible value combinations of their arguments. Note that the constant 0 is represented as 0 while the constant 1 is represented as $2^{2^n} - 1$, corresponding to a column in the truth table containing ones exclusively.

3 Binary Decision Diagrams

We have seen that Natural Numbers in $[0..2^{2^n} - 1]$ can be used as representations of truth tables defining n-variable boolean functions. A binary decision diagram (BDD) [3] is an ordered binary tree obtained from a boolean function, by assigning its variables, one at a time, to `0` (left branch) and `1` (right branch). In virtually all practical applications BDDs are represented as DAGs after detecting shared nodes. We safely ignore this here as they represent the same logic function, which is all we care about at this point. Typically in the early literature, the acronym ROBDD is used to denote reduced ordered BDDs. Because

this optimization is now so prevalent, the term BDD is frequently use to refer to ROBDDs. Strictly speaking, BDD in this paper will stand for *ordered BDD with reduction of identical branches but without node sharing*.

The construction deriving a BDD of a boolean function f is known as Shannon expansion [4], and is expressed as

$$f(x) = (\bar{x} \wedge f[x \leftarrow 0]) \vee (x \wedge f[x \leftarrow 1]) \tag{2}$$

where $f[x \leftarrow a]$ is computed by uniformly substituting a for x in f. Note that by using the more familiar boolean if-the-else function Shannon expansion can also be expressed as:

$$f(x) = if\ x\ then\ f[x \leftarrow 1]\ else\ f[x \leftarrow 0] \tag{3}$$

We represent a *BDD* in Prolog as a binary tree with constants `0` and `1` as leaves, marked with the function symbol `c/1`. Internal *if-then-else* nodes marked with `ite/3` are controlled by variables, ordered identically in each branch, as first arguments of `ite/1`. The two other arguments are subtrees representing the `Then` and `Else` branches. Note that, in practice, reduced, canonical DAG representations are used instead of binary tree representations.

Alternatively, we observe that the Shannon expansion can be directly derived from a 2^n size truth table, using bitstring operations on encodings of its n variables. Assuming that the first column of a truth table corresponds to variable x, $x = 0$ and $x = 1$ mask out, respectively, the upper and lower half of the truth table.

```
% splits a truth table of NV variables in 2 tables of NV-1 variables
shannon_split(NV,X, Hi,Lo):-
  all_ones_mask(NV,M),
  NV1 is NV-1,
  all_ones_mask(NV1,LM),
  HM is xor(M,LM),
  Lo is /\(LM,X),
  H is /\(HM,X),
  Hi is H>>(1<<NV1).
```

Note that the operation `shannon_split` can be reversed as follows:

```
% fuses 2 truth tables of NV-1 variables into one of NV variables
shannon_fuse(NV,Hi,Lo, X):-
  NV1 is NV-1,
  H is Hi<<(1<<NV1),
  X is \/(H,Lo).

?- shannon_split(2, 7, X,Y),shannon_fuse(2, X,Y, Z).
X = 1,
Y = 3,
Z = 7.

?- shannon_split(3, 42, X,Y),shannon_fuse(3, X,Y, Z).
```

```
X = 2,
Y = 10,
Z = 42.
```

Another way to look at these two operations (for a fixed value of NV), is as bijections associating a pair of natural numbers to a natural number, i.e. as *pairing* functions.

4 Pairing and Unpairing Functions

Definition 1 *A* pairing function *is a bijection* $f : Nat \times Nat \rightarrow Nat$. *An* unpairing function *is a bijection* $g : Nat \rightarrow Nat \times Nat$.

Following Julia Robinson's notation [5], given a pairing function J, its left and right inverses K and L are such that

$$J(K(z), L(z)) = z \tag{4}$$

$$K(J(x, y)) = x \tag{5}$$

$$L(J(x, y)) = y \tag{6}$$

We refer to [6] for a typical use in the foundations of mathematics and to [7] for an extensive study of various pairing functions and their computational properties.

4.1 Cantor's Pairing Function

Starting from Cantor's pairing function

```
cantor_pair(K1,K2,P):-P is (((K1+K2)*(K1+K2+1))//2)+K2.
```

bijections from $Nat \times Nat$ to Nat have been used for various proofs and constructions of mathematical objects [5, 6].

For $X, Y \in \{0, 1, 2, 3\}$ the sequence of values of this pairing function is:

```
?- findall(R,(between(0,3,A),between(0,3,B),cantor_pair(A,B,R)),Rs).
Rs = [0, 2, 4, 6, 1, 5, 9, 13, 3, 11, 19, 27, 7, 23, 39, 55]
```

Note however, that the inverse of Cantor's pairing function involves potentially expensive floating point operations that are also likely to loose precision for arbitrary length integers.

4.2 The Pepis-Kalmar Pairing Function

Another pairing function that can be implemented using only elementary integer operations is the following:

$$f(x,y) = 2^x(2y+1) - 1 \tag{7}$$

The predicates `pepis_pair/3` and `pepis_unpair/3` are derived from the function **pepis_J** and its left and right unpairing companions **pepis_K** and **pepis_L** that have been used, by Pepis, Kalmar and Robinson in some fundamental work on recursion theory, decidability and Hilbert's Tenth Problem in [8–10]:

```
pepis_pair(X,Y,Z):-pepis_J(X,Y,Z).

pepis_unpair(Z,X,Y):-pepis_K(Z,X),pepis_L(Z,Y).

pepis_J(X,Y, Z):-Z is ((1≪X)∗((Y≪1)+1))-1.
pepis_K(Z, X):-Z1 is Z+1,two_s(Z1,X).
pepis_L(Z, Y):-Z1 is Z+1,no_two_s(Z1,N),Y is (N-1)≫1.

two_s(N,R):-even(N),!,H is N≫1,two_s(H,T),R is T+1.
two_s(_,0).

no_two_s(N,R):-two_s(N,T),R is N // (1≪T).

even(X):- 0 =:= /\(1,X).
odd(X):- 1 =:= /\(1,X).
```

This pairing function is asymmetrically growing (faster growth on the first argument). It works as follows:

```
?- pepis_pair(1,10,R).
R = 41.

?- pepis_unpair(10,1,R).
R = 3071.

?- findall(R,(between(0,3,A),between(0,3,B),pepis_pair(A,B,R)),Rs).
Rs=[0, 2, 4, 6, 1, 5, 9, 13, 3, 11, 19, 27, 7, 23, 39, 55]
```

4.3 Pairing/Unpairing operations acting directly on bitlists

We will describe here pairing operations, that are expressed exclusively as bitlist transformations of `bitmerge_unpair` and its inverse `bitmerge_pair`, and are therefore likely to be easily hardware implementable. As we have found out recently, they turn out to be the same as the functions defined in Steven Pigeon's PhD thesis on Data Compression [11], page 114).

The predicate `bitmerge_pair` implements a bijection from $Nat \times Nat$ to Nat that works by splitting a number's big endian bitstring representation into odd

and even bits, while its inverse `to_pair` blends the odd and even bits back together. The helper predicates `to_rbits` and `from_rbits`, given in the Appendix, convert to/from integers to bitlists.

```
bitmerge_pair(X,Y,P):-
  to_rbits(X,Xs),
  to_rbits(Y,Ys),
  bitmix(Xs,Ys,Ps),!,
  from_rbits(Ps,P).

bitmerge_unpair(P,X,Y):-
  to_rbits(P,Ps),
  bitmix(Xs,Ys,Ps),!,
  from_rbits(Xs,X),
  from_rbits(Ys,Y).

bitmix([X|Xs],Ys,[X|Ms]):-!,bitmix(Ys,Xs,Ms).
bitmix([],[X|Xs],[0|Ms]):-!,bitmix([X|Xs],[],Ms).
bitmix([],[],[]).
```

The transformation of the bitlists, done by the bidirectional predicate bitmix is shown in the following example with bitstrings aligned:

```
?- bitmerge_unpair(2008,X,Y),bitmerge_pair(X,Y,Z).
X = 60,
Y = 26,
Z = 2008

% 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
%   60:[      0,    1,    1,    1,    1]
%   26:[   0,    1,    0,    1,    1   ]
```

Note that we represent numbers with bits in reverse order (least significant on the left). Like in the case of Cantor's pairing function, we can see similar growth in both arguments:

```
?- between(0,15,N),bitmerge_unpair(N,A,B),
   write(N:(A,B)),write(' '),fail;nl.
0: (0, 0) 1: (1, 0) 2: (0, 1) 3: (1, 1)
4: (2, 0) 5: (3, 0) 6: (2, 1) 7: (3, 1)
8: (0, 2) 9: (1, 2) 10: (0, 3) 11: (1, 3)
12: (2, 2) 13: (3, 2) 14: (2, 3) 15: (3, 3)

?- between(0,3,A),between(0,3,B),bitmerge_pair(A,B,N),
   write(N:(A,B)),write(' '),fail;nl.
0: (0, 0) 2: (0, 1) 8: (0, 2) 10: (0, 3)
1: (1, 0) 3: (1, 1) 9: (1, 2) 11: (1, 3)
4: (2, 0) 6: (2, 1) 12: (2, 2) 14: (2, 3)
5: (3, 0) 7: (3, 1) 13: (3, 2) 15: (3, 3)
```

It is also convenient sometimes to see pairing/unpairing as one-to-one functions from/to the underlying language's ordered pairs, i.e. `X-Y` in Prolog :

```
bitmerge_pair(X-Y,Z):-bitmerge_pair(X,Y,Z).

bitmerge_unpair(Z,X-Y):-bitmerge_unpair(Z,X,Y).
```

5 Encodings of Binary Decision Diagrams

We will build a BDD by applying `bitmerge_unpair` recursively to a Natural Number `TT`, seen as an N-variable 2^N bit truth table. This results in a complete binary tree of depth N. As we will show later, this binary tree represents a BDD that returns `TT` when evaluated applying its boolean operations.

```
% NV=number of varibles, TT=a truth table, BDD the result
plain_bdd(NV,TT, bdd(NV,BDD)):-
  Max is (1<<(1<<NV)),
  TT<Max,
  isplit(NV,TT, BDD).

% recurses to depth NV, splitting TT into pairs
isplit(0,TT,c(TT)).
isplit(NV,TT,R):-NV>0,
  NV1 is NV-1,
  bitmerge_unpair(TT,Hi,Lo),
  isplit(NV1,Hi,H),
  isplit(NV1,Lo,L),
  ite(NV1,H,L)=R.
```

The following examples show the results returned by `plain_bdd` for all 2^{2^k} truth tables associated to k variables, with $k = 2$.

```
?- between(0,15,TT),plain_bdd(2,TT,BDD),write(TT:BDD),nl,fail;nl
0:bdd(2, ite(1, ite(0, c(0), c(0)), ite(0, c(0), c(0))))
1:bdd(2, ite(1, ite(0, c(1), c(0)), ite(0, c(0), c(0))))
2:bdd(2, ite(1, ite(0, c(0), c(0)), ite(0, c(1), c(0))))
...
13:bdd(2, ite(1, ite(0, c(1), c(1)), ite(0, c(0), c(1))))
14:bdd(2, ite(1, ite(0, c(0), c(1)), ite(0, c(1), c(1))))
15:bdd(2, ite(1, ite(0, c(1), c(1)), ite(0, c(1), c(1))))
```

5.1 Reducing the $BDDs$

The predicate `bdd_reduce` reduces a BDD by trimming identical left and right subtrees, and the predicate `bdd` associates this reduced form to $N \in Nat$.

```
bdd_reduce(BDD,bdd(NV,R)):-nonvar(BDD),BDD=bdd(NV,X),bdd_reduce1(X,R).

bdd_reduce1(c(TT),c(TT)).
bdd_reduce1(ite(_,A,B),R):-A==B,bdd_reduce1(A,R).
bdd_reduce1(ite(X,A,B),ite(X,RA,RB)):-A\==B,
```

```
  bdd_reduce1(A,RA),bdd_reduce1(B,RB).

bdd(NV,TT, ReducedBDD):-
  plain_bdd(NV,TT, BDD),
  bdd_reduce(BDD,ReducedBDD).
```

Note that we omit here the reduction step consisting in sharing common subtrees, as it is obtained easily by replacing trees with DAGs. The process is facilitated by the fact that our unique encoding provides a perfect hashing key for each subtree. The following examples show the results returned by `bdd` for `NV=2`.

```
?- between(0,15,TT),bdd(2,TT,BDD),write(TT:BDD),nl,fail;nl
0:bdd(2, c(0))
1:bdd(2, ite(1, ite(0, c(1), c(0)), c(0)))
2:bdd(2, ite(1, c(0), ite(0, c(1), c(0))))
3:bdd(2, ite(0, c(1), c(0)))
...
13:bdd(2, ite(1, c(1), ite(0, c(0), c(1))))
14:bdd(2, ite(1, ite(0, c(0), c(1)), c(1)))
15:bdd(2, c(1))
```

5.2 From BDDs to Natural Numbers

One can "evaluate back" the binary tree representing the BDD, by using the pairing function `bitmerge_pair`. The inverse of `plain_bdd` is implemented as follows:

```
plain_inverse_bdd(bdd(_,X),TT):-plain_inverse_bdd1(X,TT).

plain_inverse_bdd1(c(TT),TT).
plain_inverse_bdd1(ite(_,L,R),TT):-
  plain_inverse_bdd1(L,X),
  plain_inverse_bdd1(R,Y),
  bitmerge_pair(X,Y,TT).


?- plain_bdd(3,42, BDD),plain_inverse_bdd(BDD,N).
BDD = bdd(3,
          ite(2,
              ite(1,
                  ite(0, c(0), c(0)),
                  ite(0, c(0), c(0))),
              ite(1,
                  ite(0, c(1), c(1)),
                  ite(0, c(1), c(0))))),
N = 42
```

Note however that `plain_inverse_bdd/2` does not act as an inverse of `bdd/3`, given that the *structure* of the BDD tree is changed by reduction.

5.3 Boolean Evaluation of BDDs

This raises the obvious question: how can we recover the original truth table from a reduced BDD? The obvious answer is: by evaluating it as a boolean function! The predicate `ev/2` describes the *BDD* evaluator:

```
ev(bdd(NV,B),TT):-
  all_ones_mask(NV,M),
  eval_with_mask(NV,M,B,TT).

evc(0,_,0).
evc(1,M,M).

eval_with_mask(_,M,c(X),R):-evc(X,M,R).
eval_with_mask(NV,M,ite(X,T,E),R):-
  eval_with_mask(NV,M,T,A),
  eval_with_mask(NV,M,E,B),
  var_to_bitstring_int(NV,M,X,V),
  ite(V,A,B,R).
```

The predicate `ite/4` used in `eval_with_mask` implements the boolean function `if X then T else E` using arbitrary length bitvector operations:

```
ite(X,T,E, R):-R is xor(/\(X,xor(T,E)),E).
```

Note that this equivalent formula for `ite` is slightly more efficient than the obvious one with $\wedge$ and $\vee$ as it requires only 3 boolean operations. We will use `ite/4` as the basic building block for implementing a boolean evaluator for BDDs.

5.4 The Equivalence

A surprising result is that boolean evaluation and structural transformation with repeated application of *pairing* produce the same result, i.e. the predicate `ev/2` also acts as an inverse of `bdd/2` and `plain_bdd/2`.
As the following example shows, boolean evaluation `ev/2` *faithfully emulates* `plain_inverse_bdd/2`*, on both plain and reduced BDDs.*

```
?- plain_bdd(3,42,BDD),ev(BDD,N).
BDD = bdd(3,
        ite(2,
            ite(1,
                ite(0, c(0), c(0)),
                ite(0, c(0), c(0))),
            ite(1,
                ite(0, c(1), c(1)),
                ite(0, c(1), c(0))))),
N = 42

?- bdd(3,42,BDD),ev(BDD,N).
BDD = bdd(3,
```

```
         ite(2,
            c(0),
            ite(1,
                c(1),
                ite(0, c(1), c(0)))),
N = 42
```

The main result of this subsection can now be summarized as follows:

Proposition 2 *Let B be the complete binary tree of depth N, obtained by recursive applications of* `bitmerge_unpair` *on a truth table T, as described by the predicate* `plain_bdd(N,T,B)`.

Then for any NV and any T, when B is interpreted as an (unreduced) BDD, the result V of its boolean evaluation using the predicate $ev(N, B, V)$ and the result R obtained by applying $plain\_inverse\_bdd(N, B, R)$ are both identical to T. Moreover, the operation $ev(N, B, V)$ reverses the effects of both `plain_bdd` *and* `bdd` *with an identical result.*

Proof: The predicate `plain_bdd` builds a binary tree by splitting the bitstring $tt \in [0..2^N - 1]$ up to depth N. Observe that this corresponds to the Shannon expansion [4] of the formula associated to the truth table, using variable order $[n-1, ..., 0]$. Observe that the effect of `bitstring_unpair` is the same as

- the effect of `var_to_bitstring_int(N,M,(N-1),R)` acting as a mask selecting the left branch
- and the effect of its complement, acting as a mask selecting the right branch.

Given that 2^N is the double of 2^{N-1}, the same invariant holds at each step, as the bitstring length of the truth table reduces to half. On the other hand, it is clear that ev reverses the action of both `plain_bdd` and `bdd` as BDDs and reduced BDDs represent the same boolean function [3].

This result can be seen as a yet another intriguing isomorphism between boolean, arithmetic and symbolic computations.

6 Ranking and Unranking of BDDs

One more step is needed to extend the mapping between $BDDs$ with N variables to a bijective mapping from/to Nat: we will have to "shift toward infinity" the starting point of each new block of BDDs in Nat as BDDs of larger and larger sizes are enumerated.

First, we need to know by how much - so we compute the sum of the counts of boolean functions with up to N variables.

```
bsum(0,0).
bsum(N,S):-N>0,N1 is N-1,bsum1(N1,S).

bsum1(0,2).
bsum1(N,S):-N>0,N1 is N-1,bsum1(N1,S1),S is S1+(1<<(1<<N)).
```

The stream of all such sums can now be generated as usual:

```
bsum(S):-nat(N),bsum(N,S).

nat(0).
nat(N):-nat(N1),N is N1+1.
```

What we are really interested in, is decomposing `N` into the distance to the last `bsum` smaller than N, `N_M` and the index of that generates the sum, `K`.

```
to_bsum(N, X,N_M):-
  nat(X),bsum(X,S),S>N,!,
  K is X-1,
  bsum(K,M),
  N_M is N-M.
```

Unranking of an arbitrary BDD is now easy - the index `K` determines the number of variables and `N_M` determines the rank. Together they select the right BDD with `plain_bdd` and `bdd/3`.

```
nat2plain_bdd(N,BDD):-to_bsum(N, K,N_M),plain_bdd(K,N_M,BDD).

nat2bdd(N,BDD):-to_bsum(N, K,N_M),bdd(K,N_M,BDD).
```

Ranking of a BDD is even easier: we first compute its `NumberOfVars` and its rank `Nth`, then we shift the rank by the `bsums` up to `NumberOfVars`, enumerating the ranks previously assigned.

```
plain_bdd2nat(bdd(NumberOfVars,BDD),N) :-
  B=bdd(NumberOfVars,BDD),
  plain_inverse_bdd(B,Nth),
  K is NumberOfVars-1,
  bsum(K,S),N is S+Nth.

bdd2nat(bdd(NumberOfVars,BDD),N) :-
  B=bdd(NumberOfVars,BDD),
  ev(B,Nth),
  K is NumberOfVars-1,
  bsum(K,S),N is S+Nth.
```

As the following example shows, `nat2plain_bdd/2` and `plain_bdd2nat/2` implement inverse functions.

```
?- nat2plain_bdd(42,BDD),plain_bdd2nat(BDD,N).
BDD = bdd(4,
          ite(3,
              ite(2,
                  ite(1,
                      ite(0, c(0), c(0)),
                      ite(0, c(1), c(0))),
                  ite(1,
                      ite(0, c(1), c(0)),
                      ite(0, c(0), c(0)))),
```

```
              ite(2,
                  ite(1,
                      ite(0, c(0), c(0)),
                      ite(0, c(0), c(0))),
                  ite(1, ite(0, c(0), c(0)),
                         ite(0, c(0), c(0)))))),
N = 42
```

The same applies to `nat2bdd/2` and its inverse `bdd2nat/2`.

```
?- nat2bdd(42,BDD),bdd2nat(BDD,N).
BDD = bdd(4,
          ite(3,
              ite(2,
              ite(1, c(0),
                     ite(0, c(1), c(0))),
                     ite(1,
                         ite(0, c(1),c(0)), c(0))),
                         c(0))),
N = 42
```

We can now generate infinite streams of BDDs as follows:

```
plain_bdd(BDD):-nat(N),nat2plain_bdd(N,BDD).

bdd(BDD):-nat(N),nat2bdd(N,BDD).

?- plain_bdd(BDD).
BDD = bdd(1, ite(0, c(0), c(0))) ;
BDD = bdd(1, ite(0, c(1), c(0))) ;
BDD = bdd(2, ite(1, ite(0, c(0), c(0)), ite(0, c(0), c(0)))) ;
BDD = bdd(2, ite(1, ite(0, c(1), c(0)), ite(0, c(0), c(0)))) ;
...
?- bdd(BDD).
BDD = bdd(1, c(0)) ;
BDD = bdd(1, ite(0, c(1), c(0))) ;
BDD = bdd(2, c(0)) ;
BDD = bdd(2, ite(1, ite(0, c(1), c(0)), c(0))) ;
BDD = bdd(2, ite(1, c(0), ite(0, c(1), c(0)))) ;
BDD = bdd(2, ite(0, c(1), c(0))) ;
...
```

7 Related work

Pairing functions have been used in work on decision problems as early as [8, 9, 5]. *Ranking* functions can be traced back to Gödel numberings [2, 12] associated to formulae. Together with their inverse *unranking* functions they are also used in combinatorial generation algorithms [13, 1]. Binary Decision Diagrams are the dominant boolean function representation in the field of circuit design automation [14]. BDDs have been used in a Genetic Programming context [15, 16] as

a representation of evolving individuals subject to crossovers and mutations expressed as structural transformations and recently in a machine learning context for compressing probabilistic Prolog programs [17] representing candidate theories. Other interesting uses of BDDs in a logic and constraint programming context are related to representations of finite domains. In [18] an algorithm for finding minimal reasons for inferences is given.

8 Conclusion and Future Work

The surprising connection of pairing/unpairing functions and BDDs, is the indirect result of implementation work on a number of practical applications. Our initial interest has been triggered by applications of the encodings to combinational circuit synthesis in a logic programming framework [19, 20]. We have found them also interesting as uniform blocks for Genetic Programming applications of Logic Programming. In a Genetic Programming context [21], the bijections between bitvectors/natural numbers on one side, and trees/graphs representing BDDs on the other side, suggest exploring the mapping and its action on various transformations as a phenotype-genotype connection. Given the connection between BDDs to boolean and finite domain constraint solvers it would be interesting to explore in that context, efficient succinct data representations derived from our BDD encodings.

References

1. Knuth, D.: The Art of Computer Programming, Volume 4, draft (2006) http://www-cs-faculty.stanford.edu/∼knuth/taocp.html.
2. Gödel, K.: Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für Mathematik und Physik **38** (1931) 173–198
3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691
4. Shannon, C.E.: Claude Elwood Shannon: collected papers. IEEE Press, Piscataway, NJ, USA (1993)
5. Robinson, J.: General recursive functions. Proceedings of the American Mathematical Society **1**(6) (dec 1950) 703–718
6. Cégielski, P., Richard, D.: Decidability of the theory of the natural integers with the cantor pairing function and the successor. Theor. Comput. Sci. **257**(1-2) (2001) 51–77
7. Rosenberg, A.L.: Efficient pairing functions - and why you should care. International Journal of Foundations of Computer Science **14**(1) (2003) 3–17
8. Pepis, J.: Ein verfahren der mathematischen logik. The Journal of Symbolic Logic **3**(2) (jun 1938) 61–76
9. Kalmar, L.: On the reduction of the decision problem. first paper. ackermann prefix, a single binary predicate. The Journal of Symbolic Logic **4**(1) (mar 1939) 1–9
10. Robinson, J.: An introduction to hyperarithmetical functions. The Journal of Symbolic Logic **32**(3) (sep 1967) 325–342

11. Pigeon, S.: Contributions à la compression de données. Ph.d. thesis, Université de Montréal, Montréal (2001)
12. Hartmanis, J., Baker, T.P.: On simple goedel numberings and translations. In Loeckx, J., ed.: ICALP. Volume 14 of Lecture Notes in Computer Science., Springer (1974) 301–316
13. Martinez, C., Molinero, X.: Generic algorithms for the generation of combinatorial objects. In Rovan, B., Vojtas, P., eds.: MFCS. Volume 2747 of Lecture Notes in Computer Science., Springer (2003) 572–581
14. Drechsler, R., Shi, J., Fey, G.: Synthesis of fully testable circuits from bdds. IEEE Trans. on CAD of Integrated Circuits and Systems **23**(3) (2004) 440–443
15. Sakanashi, H., Higuchi, T., Iba, H., Kakazu, Y.: Evolution of binary decision diagrams for digital circuit design using genetic programming. In Higuchi, T., Iwata, M., Liu, W., eds.: ICES. Volume 1259 of Lecture Notes in Computer Science., Springer (1996) 470–481
16. Chen, S.T., Lin, S.S., Huang, L.T., Wei, C.J.: Towards the exact minimization of bdds-an elitism-based distributed evolutionary algorithm. J. Heuristics **10**(3) (2004) 337–355
17. Raedt, L.D., Kersting, K., Kimmig, A., Revoredo, K., Toivonen, H.: Compressing probabilistic prolog programs. Machine Learning **70**(2-3) (2008) 151–168
18. Hawkins, P., Stuckey, P.J.: A hybrid bdd and sat finite domain constraint solver. In Hentenryck, P.V., ed.: PADL. Volume 3819 of Lecture Notes in Computer Science., Springer (2006) 103–117
19. Tarau, P., Luderman, B.: Exact combinational logic synthesis and non-standard circuit design. In: CF '08: Proceedings of the 2008 conference on Computing frontiers, New York, NY, USA, ACM (2008) 179–188
20. Tarau, P., Luderman, B.: A Logic Programming Framework for Combinational Circuit Synthesis. In: 23rd International Conference on Logic Programming (ICLP), LNCS 4670, Porto, Portugal, Springer (September 2007) 180–194
21. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)

Appendix

To make the code in the paper fully self contained, we list here some auxiliary functions.

```
% converts an int to a list of bits, least significant first
to_rbits(0,[]).
to_rbits(N,[B|Bs]):-N>0,B is N mod 2, N1 is N//2,
  to_rbits(N1,Bs).

% converts a list of bits (least significant first) into an int
from_rbits(Rs,N):-nonvar(Rs),from_rbits(Rs,0,0,N).

from_rbits([],_,N,N).
from_rbits([X|Xs],E,N1,N3):-NewE is E+1,N2 is X<<E+N1,
  from_rbits(Xs,NewE,N2,N3).
```

Confidence based Work Stealing in Parallel Constraint Programming

Geoffrey Chu[1], Christian Schulte[2], and Peter J. Stuckey[1]

[1] NICTA Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au
[2] KTH – Royal Institute of Technology, Sweden
cschulte@kth.se

Abstract. In parallel constraint solving, work stealing not only allows for dynamic load balancing, but also determines which parts of the search tree are searched next. Thus the place from where work is stolen has a dramatic effect on the efficiency of a parallel search algorithm. In this paper we examine quantitatively how optimal work stealing can be performed given an estimate of the relative solution densities of the subtrees at each node in the search tree and show how this is related to the branching heuristic strength. We propose an adaptive work stealing algorithm that automatically performs different work stealing strategies based on the strength of the branching heuristic at each node. Many parallel depth-first search patterns arise naturally from our algorithm. Our algorithm is able to produce near perfect or super linear algorithmic efficiencies on all problems tested. Real speedups using 8 threads ranges from 4-5 times speedup to super linear speedup.

1 Introduction

In parallel constraint solving, work stealing has often been seen only as a mechanism for keeping processors occupied. Analysis of work stealing schemes often assume that the amount of work to be done is fixed and independent of the work stealing scheme, e.g. [1]. While this is true for certain kinds of problems, e.g. finding all solutions, proving unsatisfiability, it is not true for others, e.g. finding the first solution, finding the optimal solution. Such analyses fail to account for the fact that the place from which work is stolen determines the search strategy and can have a dramatic effect on the efficiency of the parallel algorithm. Many systems choose to steal from as close to the root of the search tree as possible, e.g. [2], as this tends to give the greatest granularity. However, this is not always the best place to steal from in terms of the efficiency of the algorithm.

We illustrate how work stealing from different places can have different effects on efficiency with two examples. Let us consider a relatively simple framework for parallel search. One thread begins with ownership of the entire search tree. When a thread finishes searching the subtree it was responsible for, it will pick an unexplored part of the search tree and steal that subtree off its current owner. This continues until a solution is found, or the entire search tree has been searched in the case of unsatisfiability or optimization.

Example 1. The first problem we will consider is the Travelling Salesman Problem. In our experiments, with 8 threads, stealing left and low (as deep in the tree as possible) requires visiting a number of nodes equal to the sequential algorithm, while stealing high (near the root) requires visiting ∼30% more nodes on average (see Table 1).

The explanation for this is simple. Let us examine the details of one particular instance. In this instance, with the sequential algorithm, the optimal solution is found after 47 seconds of CPU time, after which the algorithm spends another ∼300 seconds proving that no better solution exists. When work stealing is done as left and low as possible in the parallel search, all of the threads are working towards finding that leftmost optimal solution, and the optimal solution is found in 47 seconds of total CPU time as before (wall clock time ∼6 seconds). After this, the search takes another 300 seconds of CPU time to conclude. Thus we have perfect linear speedup both in finding the optimal solution, and in proving that no better solution exist.

If we steal high however, only 1 of the threads is actually exploring the leftmost part of the search tree and working towards that leftmost optimal solution. The other 7 threads are off searching other parts of the search tree, unfruitfully in this case. This time, the optimal solution is found in 47 seconds of wall clock time (376 seconds of CPU time!). The algorithm then spends another 200 seconds of CPU time proving that no better solution exists. What has happened is that we got no speedup whatsoever for finding the optimal solution, but linear speedup for proving that no better solution exists. Since we found the optimal solution so much later in the search (376 seconds CPU time instead of 47 seconds), the threads spent an enormous amount of CPU time searching without the pruning benefits of the optimal solution, thus the total number of nodes searched in this instance is dramatically increased, leading to a great loss of efficiency. Clearly, this effect gets worse as a higher number of threads is used. □

It may appear from this example that stealing left and low would be efficient for all problems. However, such a strategy can produce at best linear speedup.

Example 2. The second problem we will consider is the n-Queens problem. The search tree is very deep and a top level mistake will not be recovered from for hours. Stealing low in parallel search solves the instance within the time limit if and only if the sequential depth first search solved it within the time limit. This only occurred when a solution falls in the very leftmost part of the search tree (only 4 instances out of 100 tested, see Table 2). Stealing high, in contrast, allows many areas of the search tree to be explored, so a poor choice at the root of the search tree is not as important. Stealing high results in solving 100 out of 100 instances tested. This is clearly far more robust than stealing low, producing greatly super-linear speedup. □

Veron *et al* [3] claims that linear and super linear speedups can be expected for branch and bound problems, but they fail to note that finding the optimal solution does not parallelize trivially as shown by Example 1. Rao and Kumar [4] (and others) show that super linear speedup ought to be consistently attainable for finding the first or the optimal solution for certain types of problems. Their analysis is valid if the search tree is random (i.e. we have no idea how the solutions

are distributed), but is not valid in systems where the branching heuristic orders the branches based on their likelihood of yielding a solution. The presence of such a branching heuristic makes linear speedup in finding solutions non-trivial. Gendron and Crainic [5] describe the issue and provide a description how the issue is handled in several systems. In general, the solutions utilise some kind of best-first criterion to guide how the problem is split up (see e.g. [6, 7]).

Our contributions in this paper are as follows. We perform a quantitative analysis of how different work stealing strategies affect the total amount of work performed and explain the relationship between branching heuristic strength and the optimal search strategy. We propose an adaptive work stealing algorithm that, when provided with a user given *confidence*, which is the estimated ratio of solution densities between the left and right subtrees at each node, will work steal in a near optimal manner. We show that confidence based work stealing leads to very good algorithmic efficiencies, i.e. it does not visit many more nodes, and sometimes much less, than sequential DFS (Depth First Search).

Although our analysis is done in the context of work stealing in parallel constraint programming systems, the analysis is actually about the relationship between branching heuristic strength and the optimal search order in the search tree created by that branching heuristic. Thus the analysis actually applies to all complete tree search algorithms whether sequential or parallel. As we will show later, when the assumptions about branching heuristic strength that lie behind standard sequential algorithms such as DFS, Interleaved Depth First Search (IDFS), Limited Discrepancy Search (LDS) or Depth-bounded Discrepancy Search (DDS) is given to our algorithm as *confidence* estimates, our algorithm automatically produces the exact same search patterns used in those algorithms. Thus our analysis and algorithm provides a framework which explains/unifies/produces all those standard search strategies. In contrast to the standard sequential algorithms which are based on rather simplistic assumptions about how branching heuristic strength varies in different parts of the search tree, our algorithm can adapt to branching heuristic strength on a node by node basis, potentially producing search patterns that are vastly superior to the standard ones. Our algorithm is also fully parallel and thus we have automatically parallelised DFS, IDFS, LDS and DDS as well.

The layout of the paper is as follows. In section 2 we perform a quantitative analysis of optimal work stealing. In section 3 we describe our adaptive work stealing algorithm. In section 4 we give examples of the behaviour of our algorithm. In section 5 we present our experimental evaluation. Finally in section 6 we conclude.

2 Analysis of Work Allocation

In this section we show quantitatively that the strength of the branching heuristic determines the optimal place to work steal from. We will concentrate on the case of solving a satisfaction problem. The case for optimization is related since it is basically a series of satisfaction problems.

Preliminary definitions. A *constraint state* (C, D) consists system of constraints C over variables V with initial domain D assigning possible values $D(v)$ to

each variables $v \in V$. The *propagation solver*, solv repeatedly removes values from the domains of variables that cannot take part in the solution of some constraint $c \in C$, until it cannot detect any new values that can be removed. It obtains a new domain $\text{solv}(C, D) = D'$. If D' assigns a variable the empty set the resulting state is a *failure state*. If D' assigns each variable a single value ($|D(v)| = 1, v \in V$) then the resulting state is a *solution state*. Failure states and solution states are *final states*.

Finite domain propagation interleaves propagation solving with search. Given a current (non-final) state (C, D) where $D = \text{solv}(C, D)$ the search process chooses a *search disjunction* $\vee_{i=1}^{n} c_i$ which is consequence of the current state $C \wedge D$. The child states of this state are calculated as $(C \wedge c_i, \text{solv}(C \wedge c_i, D)), 1 \leq i \leq n$. Given a root state (C, D), this defines a *search tree* of states, where each non-final state is an internal node with children defined by the search disjunction and final states are leaves.

The *solution density* of a search tree T with x nodes and y solution state nodes is y/x. The solution density is the inverse of the *mean nodes to solution* of T defined as x/y.

Optimal split for binary nodes For simplicity, assume that the cost of visiting each node in the search tree is roughly equal. Intuitively, the optimal way to perform a search is to assign all of our threads to the most promising parts of the search tree at each stage. These places are the parts of the search tree where the mean nodes to solution is lowest, or in other words where the solution density is highest. Assuming an oracle that could give us accurate solution density information, work stealing from nodes whose subtrees have the highest solution densities will be optimal. In practice however, the solution density estimates will not be perfect, thus we have to take various other factors into account. Namely:

1. Any estimate of the solution density of a subtree will have a very high error, with a substantial chance that the solution density is actually zero.
2. The real solution densities, and hence the errors in the estimate, are highly correlated between subtrees that are close together, as they share decision constraints from higher up in the tree, and these constraints may already have made solutions impossible or plentiful.
3. The solution density estimate of a subtree should decrease as nodes in that tree are examined without finding a solution. This is caused by two factors.
   (a) As the most fruitful parts of the subtree are searched, the average solution density of the remaining nodes decrease.
   (b) The correlation between solution densities between nearby subtrees mean that the more nodes have failed in that subtree, the more likely the remaining nodes are to fail as well.

We have to take these issues into account when utilizing solutions densities to determine where to work steal.

Example 3. Let T be a search tree with a binary decision at the root. Let $A = 0.6$ and $B = 0.4$ be the solution density estimates for the left and right branches of T. Assume also that the two subtrees have the same number of nodes. If we

had 8 threads, what is the optimal division of threads between the two branches such that the expected time to find a solution is lowest?

If the solution density estimate was perfect, we would simply send all 8 threads down the left branch. However, according to (1) the estimate has a very large error. Further according to (2) the solution density of the subtrees down the left branch are highly correlated. If we send all 8 threads down the left branch, and it turns out that the real solution density is small or zero, then all 8 threads end up stuck. Because of (1) and (2), it is actually better to send some of the threads down the right branch as well as long as B is not far smaller than A, for example, for values of $A = 0.6, B = 0.4$, we may wish to send 6 threads down the left branch and 2 threads down the right branch. □

Given the actual solution density probability distribution for the two branches, we can calculate the expected number of nodes searched to find a solution. We derive the expression for a simple case. Suppose the solution density probability distribution is uniform, i.e. has equal probability of being any value between 0 and S where S is the solution density estimate. Let A and B be the solution density estimates for the left and right branch respectively, and assume a proportion p and $(1 - p)$ of the processing power is sent down the left and right branch respectively. Then the expected number of nodes to be searched is given by the hybrid function (see Appendix A for the details of the calculation):

$$f(A, B, p) = \begin{cases} \frac{1}{pA}(2 + \ln(\frac{pA}{(1-p)B})) & \text{for } pA > (1-p)B \\ \frac{1}{(1-p)B}(2 + \ln(\frac{(1-p)B}{pA})) & \text{otherwise} \end{cases} \quad (1)$$

The shape of this function does not depend on the absolute values of A and B (which only serves to scale the function), but on their ratio, thus the shape is fixed for any fixed value of $r = A/(A + B)$. The value of p which minimizes this function for a given value of r is shown in Figure 1. This graph tells us the optimal way to divide up our processing power so that we have the lowest expected number of nodes to search.

As can be seen, although not linear, the optimal values of p are well approximated by the straight line $p = r$. In fact the value of the f function at $p = r$ is no more than 2% higher than the true minimum for any r over the range of $0.1 \leq r < 0.9$. For simplicity we will make this approximation from now on. This means that it is near optimal to divide the amount of processing power according to the ratio of the solution density estimate for the two branches. For example, if $r = 0.9$, which means that A is 9 times as high as B, then it is near optimal to send 0.9 of our processing power down the left branch and 0.1 of our processing power down the right. Or if $r = 0.5$, which means that $A = B$, then it is near optimal to send equal amounts of processing power down the two branches.

Define the *confidence* of a branching heuristic at each node as the ratio $r = A/(A + B)$. The branching heuristic can be considered *strong* when $r \to 1$, that is the solution density estimate of the left branch is far greater than for the right branch, or in other words, the heuristic is really good at shaping the search tree so that solutions are near the left. In this case, our analysis shows that since r is close to 1, we should allocate almost all our processing power to the left branch everytime. This is equivalent to stealing as left and as low as possible. The

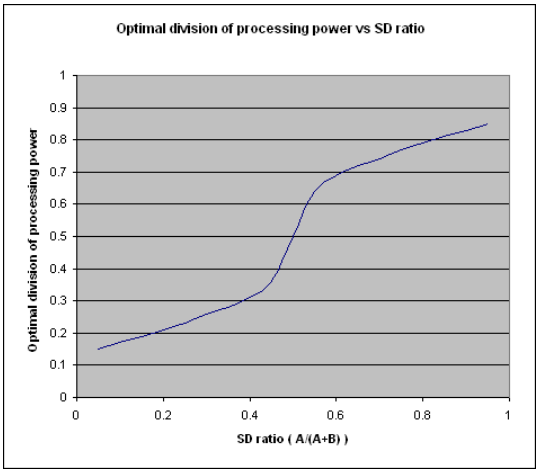


Fig. 1. Optimal division of processing power based on solution density ratio

branching heuristic is *weak* when $r \approx 0.5$, that is the solution density estimate of the left branch and right branch are similar because the branching heuristic has no clue where the solutions are. In this case, our analysis shows that since $r = 0.5$, the processing power should be distributed evenly between left and right branches at each node. This is equivalent to stealing as high as possible.[3]

3 Adaptive work stealing

Our analysis shows that the optimal work stealing strategy is dependant on the strength of the branching heuristic. Since we have a quantitative understanding of how optimal work stealing is related to branching heuristic strength, we can design a search algorithm that can automatically adapt and produce "optimal" search patterns when given some indication of the strength of the branching heuristic by the problem model. In this section, we flesh out the theory and discuss the implementation details of the algorithm in Gecode [8].

3.1 Dynamically updating solution density estimates

Now we examine how solution density estimates should be updated during search as more information becomes available.

First we need to relate the solution density estimate of a subtree with root (C, D) with the solution density estimate of its child subtrees (the subtrees rooted at its child states $(C \wedge c_i, \text{solv}(C \wedge c_i, D))$). Consider an n-ary node. Let the subtree have solution density estimate S. Let the child subtree at the ith branch have solution density estimate A_i and have size (number of nodes) x_i. If S and A_i

[3] We ignore the possibility of an *anti-heuristic* where the right branch is preferable to the left.

are estimates of average solution density, then clearly: $S = \sum_{i=1}^{n} A_i x_i / \sum_{i=1}^{n} x_i$, i.e. the average solution density of the subtree is the weighted average of the solution densities of its child subtrees.

Assuming no correlation between the solution densities of subtrees, we have that if the first k child subtrees have been searched unsuccessfully, then the updated solution density estimate is $S = \sum_{i=k+1}^{n} A_i x_i / \sum_{i=k+1}^{n} x_i$. Assuming that x_i are all approximately equal, then the expression simplifies to: $S = \sum_{i=k+1}^{n} A_i/(n-k)$. For example, suppose $A_1 = 0.3, A_2 = 0.2, A_3 = 0.1$, then initially, $S = (0.3 + 0.2 + 0.1)/3 = 0.2$. After branch 1 is searched, we have $S = (0.2 + 0.1)/2 = 0.15$, and after branch 2 is searched, we have $S = (0.1)/1 = 0.1$. This has the effect of reducing S as the branches with the highest values of A_i are searched, as the average of the remaining branches will decrease.

Now we consider the case where there are correlations between the solution density estimates of the child subtrees. The correlation is likely since all of the nodes in a subtree share the constraint C of the root state. Since the correlation is difficult to model we pick a simple generic model. Suppose the solution density estimates for each child subtree is given by $A_i = \rho A'_i$, where ρ represents the effect on the solution density due to the constraint added at the root node, and A'_i represents the effect on the solution density due to constraints added within branch i. Then ρ is a common factor in the solution density estimates for each branch and represents the correlation between them. We have that:

$$S = \frac{\sum_{i=1}^{n} A_i x_i}{\sum_{i=1}^{n} x_i} = \rho \frac{\sum_{i=1}^{n} A'_i x_i}{\sum_{i=1}^{n} x_i}.$$

Suppose that when k out of n of the branches have been searched without finding a solution, the value of ρ is updated to $\rho\frac{n-k}{n}$. This models the idea that the more branches have failed, the more likely it is that the constraint C added at the root node has already made solutions unlikely or impossible. Then when k branches have been searched, we have: $S = \rho\frac{n-k}{n} \sum_{i=k+1}^{n} A'_i x_i / \sum_{i=k+1}^{n} x_i$. Assuming that x_i are all approximately equal again, then the expression simplifies to: $S = \rho\frac{n-k}{n} \sum_{i=k+1}^{n} A'_i/(n-k) = \frac{\rho}{n} \sum_{i=k+1}^{n} A'_i = \sum_{i=k+1}^{n} A_i/n$. Equivalently, we can write it as:

$$S = \frac{\sum_{i=1}^{n} A_i}{n} \tag{2}$$

where we update A_i to 0 when branch i fails. The formula can be recursively applied to update the solution density estimates of any node in the tree given a change in solution density estimate in one of its subtrees.

In all of our results, the actual values of the solution densities are not required. We can formulate everything using *confidence*, the ratio between the solution densities of the different branches at each node. In terms of confidence, when a subtree is searched and fails the confidence values should be updated as follows:

Let r_i be the confidence value of the node i levels above the root of the failed subtree and r'_i be the updated confidence value. Let $\bar{r}_i = r_i, \bar{r}'_i = r'_i$ if the failed subtree is in the left branch of the node ith levels above the root of the failed subtree and $\bar{r}_i = 1 - r_i, \bar{r}'_i = 1 - r'_i$ otherwise. Then:

$$\bar{r}'_i = (\bar{r}_i - \prod_{k=1}^{i} \bar{r}_i)/(1 - \prod_{k=1}^{i} \bar{r}_i) \quad (3)$$

3.2 Confidence model

Given a confidence at each node, we now know how to work steal "optimally", and how to update confidences as search proceeds. But how do we get an initial confidence at each node. Ideally, the problem modeller, with expert knowledge about the problem and the branching heuristic can develop a solution density heuristic that gives us a confidence value at each node. However, this may not always happen, perhaps due to a lack of time or expertise. We can simplify things by using general confidence models. For example, we could assume that the confidence takes on an equal value *conf* for all nodes. This is sufficient to model general ideas like: the heuristic is strong or the heuristic is weak. Or we could have a confidence model that assigns $r = 0.5$ to the top d levels and $r = 0.99$ for the rest. This can model ideas like the heuristic is weak for the first d levels, but very strong after that, much like the assumptions used in DDS.

3.3 The algorithm

Given that we have a confidence value at each node, our confidence based search algorithm will work as follows. The number of threads down each branch of a node is updated as the search progresses. When a job is finished, the confidence values of all nodes above the finished subtree is updated as described in (3).

When work stealing is required, we start at the root of the tree, and use the number of threads down each branch, the confidence value, and the optimal division derived in Section 2 to work out whether the thread should be assigned to the left branch or the right branch. We then move on to that node and repeat. We continue until we find an unexplored node, at which point we steal the subtree with that unexplored node as root.

There is an exception to this. Although we may sometimes want to steal as low as possible, we cannot steal too low, as then the granularity would become too small and communication costs will dominate the runtime. Thus we dynamically determine a granularity bound under which threads are not allowed to steal, e.g. 15 levels above the average fail depth. If the work stealing algorithm guides the work stealing to the level of the granularity bound, then the last unexplored node above the granularity bound is stolen instead. The granularity bound is dynamically adjusted to maintain a minimum average job size so that work stealing does not occur more often than a certain threshold.

Since the confidence values are constantly updated, the optimal places to search next changes as search progresses. In order for our algorithm to adapt quickly, we do not require a thread to finish the entire subtree it stole before stealing again, as this could take exponential time [9]. Instead, after a given *restart* time has passed, the thread returns the unexplored parts of its subtree to the master and work steals again from the top. This is similar to the idea used in interleaving DFS [10].

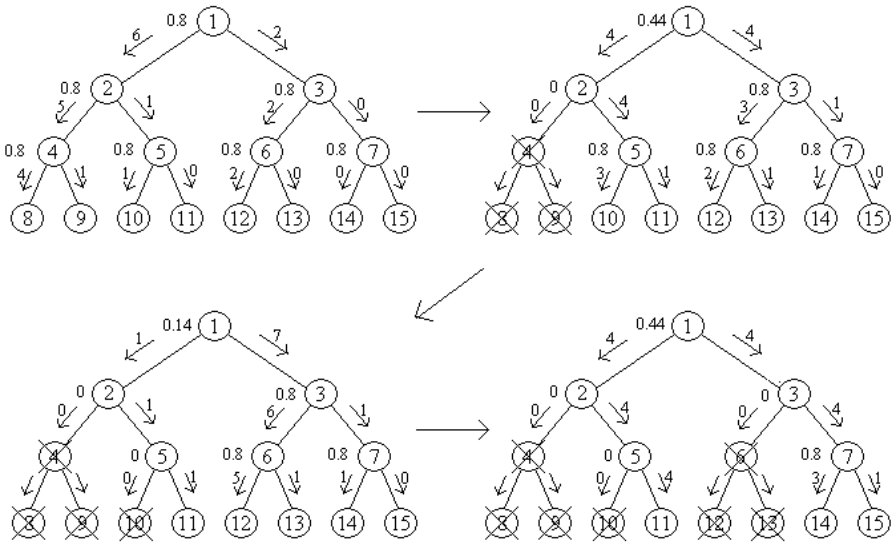


Fig. 2. Example 1

4 Sample behaviour of adaptive work stealing algorithm

In this section, we go through some examples of how the work stealing and confidence updating works in our algorithm. For the first example, suppose we know that the branching heuristic is reasonably strong, but not perfect. We may use $conf = 0.8$. Refer to Figure 2 in the explanation.

Let's suppose we have 8 threads. Initially, all the confidence values are 0.8. When the 8 threads attempt to work steal at the root, the first thread will go down the left hand side. The second thread will go down the left hand side as well. The 3rd thread will go down the right hand side. The fourth thread will go down the left hand side, etc, until we end up with 6 threads down the left and 2 threads down the right. At node 2, we will have 5 threads down the left and 1 thread down the right. At node 3, we will have 2 threads down the left, and so on. The work stealing has strongly favored sending threads towards the left side of each node because of the reasonably high confidence values of 0.8.

Suppose as search progresses the subtree starting at node 4 finishes without producing a solution. Then we need to update the confidence values. Using (3), the confidence value at node 2 becomes 0, and the confidence value at node 1 becomes 0.44. Now when the threads work steal from the root, things are different. Since one of the most fruitful parts of the left branch has been completely searched without producing a solution, it has become much less likely that there is a solution down the left branch. The updated confidence value reflects this. Now the threads will be distributed such that 4 threads are down the left branch and 4 threads are down the right branch. Next, perhaps the subtree starting at node 10 finishes. The confidence value at node 5 then becomes 0, the confidence value at node 2 remains 0 and the confidence value at node 1 becomes 0.14.

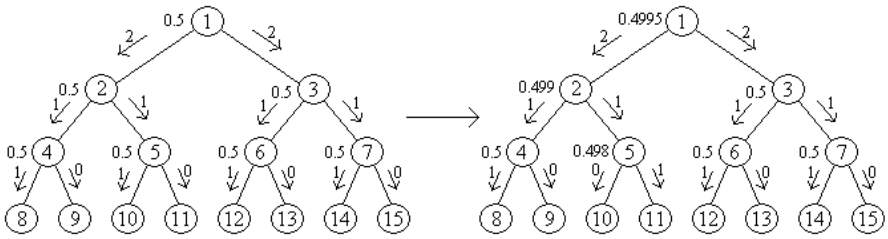


Fig. 3. Example 2

The vast majority of the fruitful places in the left branch has been exhausted without finding a solution, and the confidence value at the root has been updated to strongly favor the right branch. The threads will now be distributed such that 7 threads go down the right and 1 go down the left. Next, suppose the subtree starting at node 6 finishes. The confidence value at node 3 becomes 0 and the confidence value at node 1 becomes 0.44. Since the most fruitful part of the right branch has also failed, the confidence value now swings back to favor the left branch more. This kind of confidence updating and redistribution of threads will continue on, distributing the threads according to the current best solution density estimates. In our explanation here, for simplicity we only updated the confidence values very infrequently. In the actual implementation, confidence values are updated after every job is finished and thus occur much more frequently and in much smaller sized chunks.

For the second example, suppose we knew that the heuristic was very bad and was basically random. We may use $conf = 0.5$, i.e. the initial solution density estimates down the left and right branch are equal. Refer to Figure 3 in the explanation.

Let's suppose we have 4 threads. Initially, all the confidence values are 0.5. When the 4 threads attempt to work steal at the root, the first thread will go left, then left, then left, etc. The second thread will go right, then left, then left, etc. The third thread will go left, then right, then left, etc, and the fourth thread will go right, then right, then left, etc. This distributes the threads as far away from each other as possible which is exactly what we want. However, if the search tree is deep, and the first few decisions that the threads made within its own subtree are wrong, they may still all get stuck and never find a solution. This is where the interleaving limit kicks in. After a certain time threshold is reached, the threads abandon their current search and begin work stealing from the root again. Since the confidence values are updated when they abandon their current job, they take a different path when they next work steal. For example, if the thread down node 5 abandons after having finished a subtree with root node at depth 10, then the confidence at node 5 becomes 0.498, the confidence at node 2 become 0.499, and the confidence at node 1 becomes 0.4995. Then when the thread work steals from the root, it will again go left, then right. When it gets to node 5 however, the confidence value is 0.498 and there are no threads down either branch, thus it will go right at this node instead of left like last time. The updated confidence

values has guided the thread to an unexplored part of the search tree that is as different from those already searched as possible. This always happens because solution density estimates are decremented whenever part of a subtree is found to have failed, so the confidence will always be updated to favour the unexplored parts of the search tree.

As some other examples, we briefly mention what confidence models leads to some standard search patterns. DFS: $conf = 1$, $restart = \infty$. IDFS: $conf = 1$, $restart = 1000$. LDS: $conf = 1\text{-}\epsilon$, $restart = 1$ node. DDS: $conf = 0.5$ if depth $<$ d, $1\text{-}\epsilon$ if depth $\geq$ d, $restart = 1$ node.

5 Experimental evaluation

The confidence based work stealing is implemented in Gecode 2.1.1 [8]. The benchmarks are run on a Dell PowerEdge 6850 with 4x 3.0 Ghz Xeon Dual Core Pro 7120 CPUs. 8 threads are used for the parallel search algorithm. We use a time limit of 20 min CPU time (so 2.5 min wall clock time for 8 threads), a restart time of 5 seconds, and a dynamic granularity bound that adjusts itself to try to steal no more than once every 0.5 seconds. We collected the following data: wall clock runtime, CPU utilization, communication overhead, number of steals, total number of nodes searched and number of nodes explored to find the optimal solution.

In our first set of experiments we examine the efficiency of our algorithm for two optimization problems from Gecode's example problems. The problems are: Travelling Salesman Problem (TSP), Photo and Queens-Armies. A description of these problems can be found at [8]. We use the given search heuristic (in the Gecode example file) for each, except for TSP where we try both a strong heuristic based on maximising cost reduction and a weak heuristic that just picks variables and values in order. For both Photo and TSP, we randomly generated many instances of an appropriate size for benchmarking. Only the size 9 and size 10 instances of Queen-Armies are of an appropriate size for benchmarking. We use the simple confidence model with $conf = 1$, 0.66 and 0.5. The results are given in Table 1.

It is apparent from our experiments that the hardware/OS we experimented on is highly non-ideal and does not in fact give us a linear increase in real processing speed when more processors are used. We suspect this is due to issues such as cache contention, memory contention, context switching, etc. The effect causes threads to slow down by up to 40% at 8 threads. In view of this, the primary statistics we will look at in our analysis of our algorithm will be algorithmic efficiency and the communication cost. Algorithmic efficiency minus the communication cost represents the theoretical efficiency on an ideal parallel computer. The runtime efficiency represents what you may get on a real world, non-ideal parallel computer.

It is clear that in all of our problems, runtime is essentially proportional to the number of nodes searched, and it is highly correlated to the amount of time taken to find the optimal solution. The quicker the optimal solution is found, the fewer the nodes searched and the lower the total runtime. The communication cost, which includes all work stealing and synchronisation overheads, is less than

Table 1. Experimental results for optimization problems with simple confidence model. The results show: number of problems solved in the time limit (Solved), wall clock runtime in seconds (Runtime), speedup relative to the sequential version (Speedup), and runtime efficiency (RunE) which is Speedup/8, CPU utilization (CPU%), communication overhead (Comm%), number of steals (Steals), total number of nodes explored (Nodes), the algorithmic efficiency (AlgE) the total number of nodes explored in the parallel version versus the sequential version, the number of nodes explored to find the optimal solution (Onodes), and the solution finding efficiency (SFE) the total number of nodes explored in the parallel version to find the optimal versus the sequential version. Values for Runtime, CPU%, Comm%, Steals, Nodes, and Onodes are the geometric mean of the instances solved by all 4 versions.

TSP with strong heuristic, 200 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	181	56.0	—	—	99.8%	0.0%	—	1240k	—	180k	—
1	172	11.3	4.95	0.62	94.7%	2.8%	447	1222k	1.01	218k	0.82
0.66	170	13.3	4.20	0.53	94.6%	0.5%	370	1517k	0.82	580k	0.31
0.5	160	16.2	3.45	0.43	94.2%	1.3%	533	1564k	0.80	658k	0.27

TSP with weak heuristic, 200 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	189	78.6	—	—	99.8%	0.0%	—	1.99M	—	1.59M	—
1	186	17.7	4.45	0.56	96.5%	4.0%	686	1.99M	1.00	1.59M	1.00
0.66	186	17.7	4.46	0.56	96.3%	0.4%	319	1.97M	1.01	1.60M	1.00
0.5	184	15.7	5.01	0.63	95.5%	0.8%	287	1.73M	1.15	1.39M	1.15

Photo, 200 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	173	63.9	—	—	99.9%	0.0%	—	5.01M	—	622k	—
1	152	15.5	4.12	0.52	98.0%	1.7%	636	4.93M	1.02	542k	1.15
0.66	153	15.5	4.12	0.52	97.5%	0.4%	388	4.91M	1.02	467k	1.33
0.5	152	15.4	4.15	0.52	97.7%	0.4%	253	4.90M	1.02	492k	1.26

Queen Armies, 2 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE	Onodes	SFE
Seq	2	1146	—	—	99.7%	0.0%	—	27.1M	—	800k	—
1	2	219	5.24	0.65	98.7%	1.1%	2519	28.8M	0.94	1669k	0.48
0.66	2	213	5.38	0.67	98.2%	0.5%	1924	28.4M	0.96	1781k	0.45
0.5	2	217	5.29	0.66	98.3%	0.4%	1631	28.6M	0.95	1902k	0.42

1% for most problems, but goes up to around 3-4% for some steal low strategies. For algorithmic efficiency, we will examine each the problem in turn.

The strong heuristic in TSP is quite strong. Using $conf = 1$ achieves near perfect algorithmic efficiency. Other values of *conf* clearly cause an algorithmic slowdown. The optimal solution is found on average 2.7 and 3.0 times slower for $conf = 0.66$ and 0.5 respectively, resulting in an algorithmic efficiency of 0.82 and 0.80 respectively. The opposite is true when the weak heuristic is used. Using $conf = 1$ or 0.66 allows us to find the leftmost optimal solution in approximately the same number of nodes as the sequential algorithm, but using $conf = 0.5$ to reflect that the heuristic is weak allows the algorithm to find the optimal solution even faster, producing an algorithmic efficiency of 1.15 compared to the sequential algorithm.

Table 2. Experimental results for satisfaction problems with simple confidence model

n-Queens, 100 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE
Seq	4	2.9	—	—	99.9%	0.0%	—	1859	—
1	4	10.4	—	—	99.0%	86.6%	2	1845	—
0.66	29	18.0	—	—	81.6%	0.3%	9	15108	—
0.5	100	2.9	—	—	65.5%	1.6%	8	14484	—

Knights, 40 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE
Seq	7	0.22	—	—	99.9%	0.0%	—	1212	—-
1	7	0.26	—	—	68.1%	59.7%	2	1150	—
0.66	13	0.50	—	—	48.0%	4.7%	8	8734	—
0.5	21	0.66	—	—	35.2%	6.0%	8	8549	—

Perfect-Square, 100 instances

conf	Solved	Runtime	Speedup	RunE	CPU%	Comm%	Steals	Nodes	AlgE
Seq	15	483.1	—	—	99.9%	0.0%	—	213k	—
1	13	72.3	6.68	0.83	98.0%	19.1%	419	216k	0.99
0.66	14	71.2	6.78	0.85	86.4%	2.9%	397	218k	0.98
0.5	82	8.9	54.02	6.75	89.0%	4.8%	21	32k	6.64

The branching heuristic in Photo is designed to minimize the size of the search tree, rather than to place the solutions on the left side of the tree, hence, it is a "weak" heuristic as far as our analysis is concerned. Using $conf = 0.66$ and 0.5 to reflect this clearly produce higher solution finding efficiency than $conf = 1$, giving 1.33 and 1.26 vs 1.15 respectively. However, for the Photo problem, there are so many optimal solutions in the search tree that one gets found extremely quickly regardless of which strategy is used, and hence finding the optimal solution faster has no real effect on total runtime.

The results for Queens-Armies show little difference depending on confidence. Clearly the heuristic is better than random at finding an optimal solution, and solution finding efficiency degrades slightly as we ignore the heuristic. But the overall nodes searched are almost identical for all confidence values.

In our second set of experiments we examine the efficiency of our algorithm for three satisfaction problems from Gecode's examples [8]. The problems are: *n*-Queens, Knights, and Perfect-Square.

The sequential version solved very few instances of *n*-Queens and Knights. Furthermore, all those solves are extremely fast (< 3 sec) and are caused by the search engine finding a solution at the very leftmost part of the search tree. Most of the time spent in those runs is from travelling down to the leaf of the search tree rather than actual search and is not parallelizable, thus comparison of the statistics for the parallel vs sequential algorithms on those instances is not meaningful as there is very little work to parallelize. The number of instances solved is the more interesting statistic and is a better means of comparison. The parallel algorithm beats the sequential algorithm by an extremely large margin in terms of the number of instances solved.

n-Queens and Knights both have very deep subtrees and thus once the sequential algorithm fails to find a solution in the leftmost subtree, it will often end up stuck effectively forever. Modelling the fact that the branching heuristic is

Table 3. Experimental results using accurate confidence values, where we follow the confidence value to degree α.

Golomb-Ruler 12

α	Nodes	AlgE
Seq	5.31M	—
1	2.24M	2.37
0.5	3.48M	1.53
0	4.27M	1.24
-0.5	10.8M	0.49
-1	10.6M	0.50

Golomb-Ruler 13

α	Nodes	AlgE
Seq	71.0M	—
1	53.2M	1.34
0.5	57.6M	1.23
0	61.9M	1.15
-0.5	74.8M	0.95
-1	111M	0.64

very weak at the top by using $conf = 0.5$ clearly produce a super linear speedup. The parallel algorithm solves 100 out of 100 instances of n-Queens compared to 4 out of 100 instances for the sequential algorithm or the parallel algorithm with $conf = 1$. The speedup cannot be measured as the sequential algorithm does not terminate for days when it fails to find a solution quickly. Similarly the parallel algorithm with $conf = 0.5$ solved 21 instances of Knights compared to 7 for the sequential and the parallel version with $conf = 1$.

Perfect Square's heuristic is better than random, but is still terribly weak. Using $conf = 0.5$ to model this once again produces super linear speedup, solving 82 instances out of 100 compared to 15 out of 100 for the sequential algorithm. We can compare runtimes for this problem as the sequential version solved a fair number of instances and those solves actually require some work (483 sec on average). The speedup in this case is 54 using 8 threads.

So far, we have tested the efficiency of our algorithm using simple confidence models where the confidence value is the same for all nodes. This is the most primitive way to use our algorithm and does not really illustrate its full power. We expect that our algorithm should perform even better when confidence values specific to each node are provided, so that we can actually encode and utilise information like, the heuristic is confident at this node but not confident at that node, etc. In our third set of experiments, we examine the efficiency of our algorithm when node specific confidence values are provided.

Due to our lack of domain knowledge, we will not attempt to write a highly accurate confidence heuristic. Rather, we will simulate one by first performing an initial full search of the search tree to find all solutions, then produce confidence estimates for the top few levels of the search tree using several strategies like, follow the measured solution density exactly, follow it approximately, ignore it, go against it, etc, to see what effect this has on runtime. Let α quantify how closely we follow the measured confidence value and let $conf$ be the measured confidence value. Then we use the following formula for our confidence estimate: $conf' = \alpha \times conf + (1 - \alpha) \times 0.5$. If $\alpha = 1$, then we follow it exactly. If $\alpha = -1$, we go against it completely, etc. We use the Golomb-Ruler problem (see [8]) for our experiment as the full search tree is small enough to enumerate completely. The results are shown in Table 3.

The results show that using confidence values that are even a little biased towards the real value is sufficient to produce super linear speedup. And not surprisingly, going against the real value will result in substantial slowdowns.

6 Conclusion

By analysing work stealing schemes using a model based on solution density, we were able to quantitatively relate the strength of the branching heuristic with the optimal place to work steal from. This leads to an adaptive work stealing algorithm that can utilise confidence estimates to automatically produce "optimal" work stealing patterns. The algorithm produced near perfect or better than perfect algorithmic efficiency on all the problems we tested. In particular, by adapting to a steal high, interleaving search pattern, it is capable of producing super linear speedup on several problem classes. The real efficiency is lower than the algorithmic efficiency due to hardware effects, but is still quite good at a speedup of at least 4-5 at 8 threads. Communication costs are negligible on all problems even at 8 threads.

References

1. Kumar, V., Rao, V. N.: Parallel depth first search. Part II. Analysis. In: International Journal of Parallel Programming, 16(6):501–519, (1987)
2. Schulte, C.: Parallel search made simple. In: Beldiceanu, N., Harvey, W., Henz, M., Laburthe, F., Monfroy, E., Müller, T. (eds.) Proceedings of TRICS: Techniques for Implementing Constraint Programming Systems, a post-conference workshop of CP 2000, Singapore, September (2000)
3. Veron, A., Schuerman, K., Reeve, M., Li, L.: Why and how in the ElipSys Or-Parallel CLP system. In: Proceedings of PARLE'93, pp. 291–302. Springer-Verlag (1993)
4. Rao, V. N., Kumar, V.: Superlinear Speedup in State-Space Search, Technical Report, AI Lab TR88-80, University of Texas at Austin June (1988)
5. Gendron, B., Crainic, T.G.: Parallel branch-and-bound algorithms: Survey and Synthesis. In: Operations Research 42, pp. 1042–1066. (1994)
6. Quinn, M. J.: Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multicomputer. In: IEEE Trans. Comp. 39(3), pp. 384–387 (1990)
7. Mohan, J.: Performance of Parallel Programs: Model and Analyses. Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Penn. (1984)
8. Gecode. `www.gecode.org`
9. Perron, L.: Search procedures and parallelism in constraint programming. In: Jaffar, J., editor, Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (1999)
10. Meseguer, P.: Interleaved depth-first search. In: International Joint Conference on Artificial Intelligence, volume 2, pp. 1382–1387, August (1997)

A Expected Search Time Calculation

Calculating the expected search time requires a double integration of a hybrid function. Since this is rather difficult we will pick a simple solution density probability distribution. Suppose the solution density probability distribution is uniform, i.e. when the solution density estimate is A, there is an equal chance of the actual value being anything from 0 to A. The real solution density values are actually discrete as there can only be a whole number of solutions, but for ease of integration we leave it as a continuous function. This probability distribution satisfies our criteria that there is a large error in the estimate and that there is a substantial chance that it is actually 0. Suppose the solution density estimates for the left and right branch are A and B respectively, and that they each have n nodes. Suppose also the solutions are randomly distributed among the n nodes. We know that when there are m items randomly located in n locations, the expected number of locations to look before we find one of them is given by $\frac{n+1}{m+1}$. Thus if a is the real solution density in the left branch, the expected time to find a solution in the left branch is $\frac{n+1}{an+1} \approx \frac{n}{an+1} = \frac{1}{a+\frac{1}{n}}$. Suppose we divide up our processing power such that p units is sent down the left branch, and $(1-p)$ units is sent down the right branch. The expected number of nodes searched will depend on which of the branches yield a solution first, thus for real solution density values of a and b for the left and right branch, it is given by the hybrid function:

$$\min\left(\frac{1}{p(a+\frac{1}{n})}, \frac{1}{(1-p)(b+\frac{1}{n})}\right) \tag{4}$$

The expected number of nodes to be searched for solution density estimates A and B for the left and right branch respectively, given a uniform solution density probability distribution will then be given by:

$$\frac{1}{AB}\int_0^A \int_0^B \min\left(\frac{1}{p(a+\frac{1}{n})}, \frac{1}{(1-p)(b+\frac{1}{n})}\right) db\, da \tag{5}$$

To evaluate this, we need to split the integral into two domains corresponding to the two halves of the hybrid function. The boundary of the hybrid function is given by:

$$\begin{aligned} & p(a+\tfrac{1}{n}) = (1-p)(b+\tfrac{1}{n}) \\ \Rightarrow \quad & a = \tfrac{1-p}{p}(b+\tfrac{1}{n}) - \tfrac{1}{n} \\ \text{or} \quad & b = \tfrac{p}{1-p}(a+\tfrac{1}{n}) - \tfrac{1}{n} \end{aligned}$$

There are four cases depending on whether the boundary of the hybrid function intersects the a or the b axis and whether it intersects the $a = A$ line or the $b = B$ line. For $p > 0.5$ and $p(A+\frac{1}{n}) > (1-p)(B+\frac{1}{n})$, which corresponds to intersecting the b axis and the $b = B$ line, we have:

$$
\begin{aligned}
&\tfrac{1}{AB}\int_0^A\int_0^B \min(\frac{1}{p(a+\frac{1}{n})},\frac{1}{(1-p)(b+\frac{1}{n})})dbda\\
=\ &\tfrac{1}{AB}[\int_{\frac{p}{(1-p)n}-\frac{1}{n}}^{B}\int_0^{\frac{1-p}{p}(b+\frac{1}{n})-\frac{1}{n}}\frac{1}{(1-p)(b+\frac{1}{n})}dadb+\\
&\int_0^{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}}\int_0^{\frac{p}{1-p}(a+\frac{1}{n})-\frac{1}{n}}\frac{1}{p(a+\frac{1}{n})}dbda+\\
&\int_{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}}^{A}\int_0^{B}\frac{1}{p(a+\frac{1}{n})}dbda\\
=\ &\tfrac{1}{AB}[\int_{\frac{p}{(1-p)n}-\frac{1}{n}}^{B}\frac{1}{p}-\frac{1}{(1-p)n(b+\frac{1}{n})}db+\\
&\int_0^{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}}\frac{1}{1-p}-\frac{1}{pn(a+\frac{1}{n})}da+\\
&\int_{\frac{1-p}{p}(B+\frac{1}{n})-\frac{1}{n}}^{A}\frac{B}{p(a+\frac{1}{n})}da\\
=\ &\tfrac{1}{AB}[\,[\frac{B}{p}-\frac{1}{(1-p)n}\ln(B+\frac{1}{n})-\frac{1}{(1-p)n}+\frac{1}{pn}+\frac{1}{(1-p)n}\ln(\frac{p}{(1-p)n})]+\\
&[\frac{B+\frac{1}{n}}{p}-\frac{1}{(1-p)n}-\frac{1}{pn}\ln(\frac{1-p}{p}(B+\frac{1}{n}))+\frac{1}{pn}\ln(\frac{1}{n})]+\\
&[\frac{B}{p}\ln(A+\frac{1}{n})-\frac{B}{p}\ln(\frac{1-p}{p}(B+\frac{1}{n}))]]\\
=\ &\tfrac{1}{AB}[\,\frac{2(B+\frac{1}{n})}{p}-\frac{1}{n}\ln(\frac{1-p}{p}(nB+1))+\\
&\frac{B}{p}\ln(\frac{p}{1-p}\frac{(nA+1)}{(nB+1)})-\frac{2}{(1-p)n}-\frac{1}{(1-p)n}\ln(\frac{1-p}{p}(Bn+1))]
\end{aligned}
$$

If we are reasonably high up in the search tree, which is where the results of this calculation is most important, then we can assume that we are expecting a potentially large number of solutions down each branch, e.g. $An, Bn \gg 1$. In that case, all of the terms containing $1/n$ are much smaller than the terms containing A or B and the expression simplifies to:

$$\frac{1}{AB}[2\frac{B+\frac{1}{n}}{p} - \frac{1}{n}\ln(\frac{1-p}{p}(nB+1)) + \tag{6}$$
$$\frac{B}{p}\ln(\frac{p}{1-p}\frac{(nA+1)}{(nB+1)}) - \frac{2}{(1-p)n} - \frac{1}{(1-p)n}\ln(\frac{1-p}{p}(Bn+1))]$$
$$= \frac{1}{AB}[\frac{2B}{p} + \frac{B}{p}\ln(\frac{pA}{(1-p)B})]$$
$$= \frac{1}{pA}(2+\ln(\frac{pA}{(1-p)B})) \tag{7}$$

The calculation for the case $p < 0.5$ and $p(A+\frac{1}{n}) > (1-p)(B+\frac{1}{n})$ is similar, and after the simplification, yields the same equation as (6). Since the problem is symmetric with respect to A and B, and p and $(1-p)$, we can trivially derive the equation for the other two cases, which is:

$$\frac{1}{(1-p)B}(2+\ln(\frac{(1-p)B}{pA})) \tag{8}$$

Thus the full function for calculating the expected number of nodes searched given A, B and p is given by the hybrid function:

$$f(A,B,p) = \begin{cases} \frac{1}{pA}(2+\ln(\frac{pA}{(1-p)B})) & \text{for } pA > (1-p)B \\ \frac{1}{(1-p)B}(2+\ln(\frac{(1-p)B}{pA})) & \text{otherwise} \end{cases}$$

An Efficient Term Representation for CHR Indexing

Beata Sarna-Starosta[1] and Tom Schrijvers*[2]

[1] LogicBlox Inc., Atlanta, Georgia, USA
bss@logicblox.com
[2] Department of Computer Science, K.U.Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract. The overhead of matching CHR's multi-headed rules is alleviated by constraint store indexing. The attributed variable interface provides efficient means of indexing on logical variables. Current state-of-the-art indexing strategies for ground terms use hash tables. However, the hash tables incur considerable performance overhead, especially when frequently computing hash values for large terms.
We propose a high-level approach which improves the efficiency of ground term indexing. In this approach, we introduce a new data representation for ground terms, inspired by attributed variables, that avoids the overhead of hash-table indexing. The experimental evaluation establishes the usefulness of our representation, but indicates a high cost of mapping between this representation and Prolog's standard terms. Thus, we reuse previously implemented post-processing program transformations to compensate for this overhead. We compare our approach with the current state of the art, and give measurements of its effectiveness in the K.U.Leuven CHR system.
keywords: Constraint Handling Rules, indexing, program transformation, term representation, attributed variables

1 Introduction

Constraint Handling Rules (CHR) [4] is a high-level rule-based declarative programming language, usually embedded in a host language such as Prolog or Haskell. Typical applications of CHR include scheduling [1] and type checking [14]. CHR features *multi-headed rules*, i.e., rules with multiple predicates on the left-hand side (the *head*), which sets it apart from conventional declarative languages, e.g., Prolog or Haskell, where a rule's head admits only one predicate or function.

Multi-headed rules afford much of CHR's expressive power by allowing to easily combine information from distinct constraints via matching. However, as the matching procedure significantly affects the complexity of rule evaluation [5], this source of expressiveness often leads to performance bottlenecks. This effect is

* Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen)

borne out by the approximative complexity formula of [5], where the multiplicity of rule's head appears in the exponent.

Aware of this problem, CHR developers have built data structures supporting efficient indexing on variables (attributed variables [7]) and ground data (search trees [8]). With [12] came the realization that $\mathcal{O}(1)$ indexing is essential for implementing CHR algorithms with optimal complexity, which led to the use of hash tables for indexing ground data, and the general result that the complexity of CHR systems equals that of RAM machines [13]. CHRd [9] has slimmed the original attributed variable indexing for faster evaluation of the class of direct-indexed CHR and use in a tabulated environment.

In this paper we advance the research on CHR indexing with a high-level approach to efficient indexing on ground terms. Specifically, we make the following contributions:

- propose an alternative to hash tables for indexing ground data, which does not suffer from amortization-related overhead (Section 3),
- reuse previously developed post-processing program transformations [10] to reduce the disadvantages of the new approach (Section 4),
- demonstrate the measurements of the usefulness of the presented technique in K.U.Leuven CHR system (Section 5), and
- provide an implementation of the presented techniques (available online at `http://www.cs.kuleuven.be/~toms/CHR/AttributedData/`).

The presentation begins with an overview of CHR and indexing in Section 2. Section 3 describes our new representation for ground terms, the conversions between the new representation and Prolog terms, and the program transformation for introducing these conversions. Section 4 discusses the overhead of the conversions, and treats it with the post-processing program transformation. Section 5 presents the experimental evaluation of the proposed transformations, Section 6 relates our approach to other work, and Section 7 concludes.

2 Preliminaries

CHR is a language of multi-headed rewriting rules that is particularly well-suited for specifying custom constraint solvers at a high-level. A CHR program prescribes the transformations of a *constraint store* (a collection of *user-defined* constraints), based on the *built-in* constraints of the host language. For the purpose of this paper we consider Prolog as the host language; the built-in constraints are Prolog predicates and equations (unifications) of Herbrand terms.

CHR Syntax. A CHR program is a finite set of rules of the form:

label `@` *Head* `?=>` *Guard* `|` *Body*

The *label* names the rule and may be omitted along with the trailing `@`. The arrow `?=>` denotes the kind of transformation a rule defines, and may be either `<=>` or `==>` (we use `?=>` as a shorthand notation for both forms). There are

```
:- chr_constraint arrow/2, merge/2.

pick @ merge(N,A), merge(N,B) <=> A<B | M is N+1, arrow(A,B), merge(M,A).
join @ arrow(X,A) \ arrow(X,B) <=> A<B | arrow(A,B).
```

Table 1. An example CHR program encoding the merge-sort algorithm

three kinds of CHR rules. The most general are *simpagation* rules of the form: $H_1 \setminus H_2$ `<=>` $G \mid B$, where H_1 and H_2 are sequences of user-defined constraint terms (the d constraint terms. A rule specifies that when constraints in the store match H_1 and H_2 and the guard G holds, the constraints that match H_2 can be *replaced* by the corresponding constraints in B. The literal `true` represents an empty sequence of constraint terms. The guard part, $G \mid$, may be omitted when G is empty.

A *simplification* rule, which has the form: H_2 `<=>` $G \mid B$, specifies that when the stored constraints match the head, and the guard holds, the head constraints can be replaced by the body constraints. A rule of this form can be represented by a simpagation rule: `true` $\setminus H_2$ `<=>` $G \mid B$.

A *propagation* rule, which has the form: H_1 `==>` $G \mid B$, specifies that when the stored constraints match the head, and the guard holds, the body constraints can be *added* to the store. A rule of this form can be represented by a simpagation rule: $H_1 \setminus$ `true` `<=>` $G \mid B$.

Example 1. Consider the CHR program in Table 1. The simplification rule `pick` states that each pair of stored constraints matching `merge(N,A)` and `merge(N,B)` such that `A<B` should be replaced with the pair of constraints `arrow(A,B)` and `merge(M,A)` where `M=N+1`. The simpagation rule `join` states that, in the presence of two constraints `arrow(X,A)` and `arrow(X,B)` such that `A<B`, the constraint `arrow(X,B)` should be replaced by `arrow(A,B)`.

The program, by Thom Frühwirth, encodes the classical merge-sort algorithm. The algorithm is executed in the bottom-up fashion: the `pick` rule selects two sublists of elements at the same level for merging, whereas the `join` rule merges two selected sublists together.

CHR Semantics. CHR has a well-defined declarative as well as operational semantics [4, 3, 9]. The declarative interpretation of a CHR program is given by the set of universally quantified formulas corresponding to the CHR rules, and an underlying consistent constraint theory.

The original operational interpretation of a CHR program [4] is a non-deterministic transition system. The transitions are made when an unsolved constraint is added to the store, or by firing any applicable program rule.

The refined operational semantics [3][3] defines a more deterministic transition system, specifying, among others, that rules are tried in textual order. An ex-

[3] followed by most CHR implementations

	$\langle[\underline{\texttt{merge}(1,80)},\texttt{merge}(1,40),\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \emptyset\rangle$	(1)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(1,40)},\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \{\texttt{merge}(1,80)\}\rangle$	(2)
$\rightarrowtail_{\texttt{pick}}$	$\langle[\underline{\texttt{arrow}(40,80)},\texttt{merge}(2,40),\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \emptyset\rangle$	(3)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(2,40)},\texttt{merge}(1,50),\texttt{merge}(1,70)],\ \{\texttt{arrow}(40,80)\}\rangle$	(4)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(1,50)},\texttt{merge}(1,70)],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40)\}\rangle$	(5)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(1,70)}],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40),\texttt{merge}(1,50)\}\rangle$	(6)
$\rightarrowtail_{\texttt{pick}}$	$\langle[\underline{\texttt{arrow}(50,70)},\texttt{merge}(2,50)],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40)\}\rangle$	(7)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(2,50)}],\ \{\texttt{arrow}(40,80),\texttt{merge}(2,40),\texttt{arrow}(50,70)\}\rangle$	(8)
$\rightarrowtail_{\texttt{pick}}$	$\langle[\underline{\texttt{arrow}(40,50)},\texttt{merge}(3,40)],\ \{\texttt{arrow}(40,80),\texttt{arrow}(50,70)\}\rangle$	(9)
$\rightarrowtail_{\texttt{join}}$	$\langle[\underline{\texttt{arrow}(50,80)},\texttt{merge}(3,40)],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50)\}\rangle$	(10)
$\rightarrowtail_{\texttt{join}}$	$\langle[\underline{\texttt{arrow}(70,80)},\texttt{merge}(3,40)],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50)\}\rangle$	(11)
$\rightarrowtail^*$	$\langle[\underline{\texttt{merge}(3,40)}],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50),\texttt{arrow}(70,80)\}\rangle$	(12)
$\rightarrowtail^*$	$\langle[\,],\ \{\texttt{arrow}(50,70),\texttt{arrow}(40,50),\texttt{arrow}(70,80),\texttt{merge}(3,40)\}\rangle$	(13)

Table 2. An example derivation for the merge-sort program

tended version of the same transition system is used by the set-based operational semantics [9].

Example 2. The merge-sort program from Example 1 constructs a sorted list from a collection of sorted sublists. The head of a sorted sublist is given by means of a `merge(L,N)` constraint, where $2^{\texttt{L}-1}$ is the sublist's length and `N` is the sublist's first element. The `arrow/2` constraints model the edges between the nodes of a sorted sublist.

Table 2 outlines an example derivation for the program under the refined operational semantics. For the clarity of the presentation, the irrelevant transitions and the parts of the execution state not affected by the derivation have been omitted. For each presented derivation step, the table shows the current goal, with the active constraint underlined, and the contents of the constraint store. In the initial goal each sublist consists of a single element, and hence all sublists have the same length (equal to 2^{1-1}). The nodes are collected in the constraint store until two same-length nodes match the head of the `pick` rule. The rule transforms such two nodes into a sorted sublist and increments the length. The `join` rule sorts the nodes within each individual sublist. At the end of the derivation, the constraint store contains a collection of `arrow/2` constraints representing the sorted list.

CHR Indexing. Indexing in CHR facilitates retrieval of suspended constraints to match partner constrains in rule heads. Efficient (constant-time) constraint store indexing has been traditionally implemented by means of attributed variables [6], which provide a way to associate Prolog variables with mutable data represented as arbitrary terms. In the context of CHR, a variable's attribute corresponds to those stored constraints, in which the variable is involved. The

attribute term has the form: `attr(`$Index_1$`,...,`$Index_n$`)`, where each $Index_i$ is a data structure, typically a list, that contains all constraints on the variable with a particular constraint symbol. The presence of all variable's constraints in its attribute expedites matching when the variable is shared among the constraints in the heads of the rules.

Constraint store indexing based on attributed variables is efficient, but not always practical—for example, it is not feasible for ground constraints, in which no variables are involved. For that reason, in addition to using variable attributes, early implementations of CHR accumulated constraints in global, unordered lists. This representation supported $\mathcal{O}(1)$-time insertion of the constraints, however, constraint lookup and deletion were—in the worst case—linear in the store size. The introduction of hash tables [12] facilitated indexing on ground data, yielding amortized constant time complexity for all operations. A hash-table constraint store is defined as an array, in which every element represents a set of colliding constraints (i.e., constraints that evaluate to the same value of the hash function). The table is initialized to a small size, and dynamically expanded whenever the number of constraints exceeds given threshold. The expansion involves replacing the current array with an array of doubled size, and re-evaluating the hash function for all elements. Frequent evaluation of the hash function, the number of colliding constraints, and the resizing operation incur constant, but potentially considerable, overhead on processing the hash tables, which makes them altogether slower than attributed variables.

3 Attributed Data

In this section, we consider constraints containing arguments that are ground terms. If such arguments are matched against each other in rule heads, then constant-time matching is realized by means of a hash-table index on these ground arguments.

As an alternative to hash tables, we propose *attributed data*, which provide $\mathcal{O}(1)$ indexing with constant factors closer to those of attributed variables. The key insight underlying our approach is that the CHR run time can internally use an attributed-variable–like representation for externally provided ground terms.

3.1 Indexing Key Declarations

In our approach, ground arguments of the constraints that are matched against each other in rule heads—and hence serve as indexing keys—are internally represented using a special data type *key type*. The programmers indicate such constraint arguments using the new annotation '`as_chr_key`'. The specifier '`+`*type* `as_chr_key` *keytype*' states that the argument in question is ground (`+`), and uses *type* as its external representation and *keytype* as its internal representation. The abstract key type for a given indexing key in a CHR program is generated automatically by the CHR compiler based on the occurrence pattern of that key in the heads of the program rules.

Example 3. In the merge-sort program from Example 1, since the second argument of `merge/2` as well as both arguments of `arrow/2` are always ground and correspond to the numbers being sorted, a programmer may decide to capture all of them using the same internal representation. Denoted as `elem_key`, this representation is declared as follows:

```
:- chr_constraint
     merge(+int,+int as_chr_key elem_key),
     arrow(+int as_chr_key elem_key,+int as_chr_key elem_key).
```

3.2 Indexing Key Representation

The instances of the new data type resemble the attribute terms of attributed variables. The key type representation, however, does not include the actual variables to avoid unnecessary indirection.

The internal representation $\mathcal{I}$ of a ground indexing key in a CHR program is a term:

$$\mathcal{I} = \texttt{key}(\mathcal{E}, \mathit{Index}_1, \ldots, \mathit{Index}_n)$$

where each Index_i is an index on an argument position of that key in a head constraint of some program rule, and $\mathcal{E}$ is the key's original external value.

The number and form of the indexes in the internal representation for a particular key is orthogonal to the use of attributed data, and is determined by the CHR compiler based on the form of the rule heads and the subset of head constraints available when looking for a matching partner. For a detailed discussion of this issue we refer the reader to Section 3.2 of [8].

For the purpose of this paper we assume that the default representation of argument indexes Index_i is a flat list of constraint suspensions, with predefined operations for adding and removing the constraints. The main structure itself can be updated (e.g. for replacing an old index with a new one) by the destructive argument update predicate `setarg/3` implemented by most Prolog systems.

Example 4. Since two of the three argument positions declared as indexing keys in Example 3 are never used to retrieve partner constraints, the CHR compiler decides that only one index—for the first argument of `arrow/2`— will be exploited to speed-up the matching of the `join` rule.

Hence, given the number 80 as the external representation, the corresponding internal representation, assuming that the single index is empty, is `key(80,[])`.

Definition 1 (Conversion Functions). *For a ground indexing key type t, the injective* conversion function ϕ *maps an external value t_ε of t onto the internal representation $t_\mathcal{I}$ of t:*

$$\phi(t_\varepsilon) = \begin{cases} h[t_\varepsilon] & \text{if } h[t_\varepsilon] \text{ is defined} \\ t_\mathcal{I} & \text{otherwise} \\ & \quad \text{such that } t_\mathcal{I} = \texttt{key}(t_\varepsilon, \emptyset_1, \ldots, \emptyset_n) \\ & \qquad \text{and } h := h[t_\varepsilon \rightarrow t_\mathcal{I}] \end{cases}$$

where h is a global hash table relating the external values of ground indexing keys to their known internal representations. The injective conversion function $\psi = \phi^{-1}$ *maps the internal representations $t_\mathcal{I}$ onto the external values $t_\mathcal{E}$:*

$$\psi(\texttt{key}(t_\mathcal{E}, Index_1, \ldots, Index_n)) = t_\mathcal{E}$$

Example 5. The following internal representations are initially computed for the list of numbers given in the query in Example 1: ϕ`(80) = key(80,[])`, ϕ`(40) = key(40,[])`, ϕ`(50) = key(50,[])`, ϕ`(70) = key(70,[])`. Figure 1 depicts these internal representations, as well as the example hash table (with a linked list of buckets) underlying ϕ.

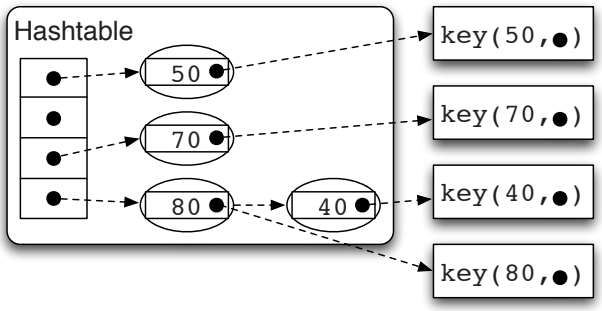


Fig. 1. Internal representation of 80, 40, 50 and 70, and hash table for ϕ.

3.3 Source-to-Source Transformation

In this section we define a source-to-source transformation for mapping between the external and internal representations of ground indexing keys. Without loss of generality, we only formalize the transformation for a single key type. Multiple keys are easily supported by repeated application of the transformation, while making sure to avoid name clashes.

The conversion rule Φ applies the conversion function ϕ at run time:

Definition 2 (Conversion Rule). *The* conversion rule Φ *replaces the external value of a ground indexing key argument t_i in a constraint term c/n with its internal representation $t' = \phi(t)$:*

$$c(t_1,\ldots,t_i,\ldots,t_n) \texttt{ <=> } t'_i = \phi(t_i),\ c'(t_1,\ldots,t'_i,\ldots,t_n).$$

Example 6. The dynamic conversion rule for the `arrow/2` constraint from the merge-sort program is of the form:

`arrow(X,Ne) <=> Ni = `ϕ`(Ne), arrow'(X,Ni).`

Definition 3 (Converted Rule). *The* converted CHR rule *is defined as:*

$$\phi(H \texttt{ ?=> } G \mid B) \;=\; H' \texttt{ ?=> } G', G \mid B$$

where

- *H' differs from H in that any constraint $c(t_1, \ldots, t_i, \ldots, t_n)$ is replaced by its converted form $c'(t_1, \ldots, x_i, \ldots, t_n)$, where x_i is a fresh variable.*
- *the new guard G' relates the original arguments of each constraint to the new ones: G' contains one t_i `=` $\psi(x_i)$ for each converted argument.*

Example 7. The converted `join` rule from the merge-sort program is of the form:

```
join' @ arrow'(X1,AI) \ arrow'(X2,BI) <=>
            X = ψ(X1), X = ψ(X2),
            A = ψ(AI), B = ψ(BI), A<B |
            arrow(A,B).
```

Definition 4 (Converted Program). *The* converted CHR program $\phi(P)$ *is defined as the set of converted rules $\overline{R}$ comprising the original program, the functions ϕ and ψ, and the encoding of Φ:*

$$\phi(P) = \phi(\overline{R}) \cup \phi \cup \psi \cup \Phi$$

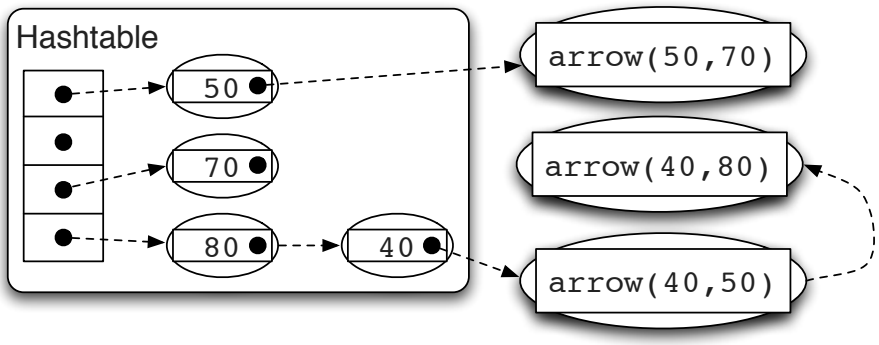
3.4 Elaborated Example

Consider the merge-sort program, and the query

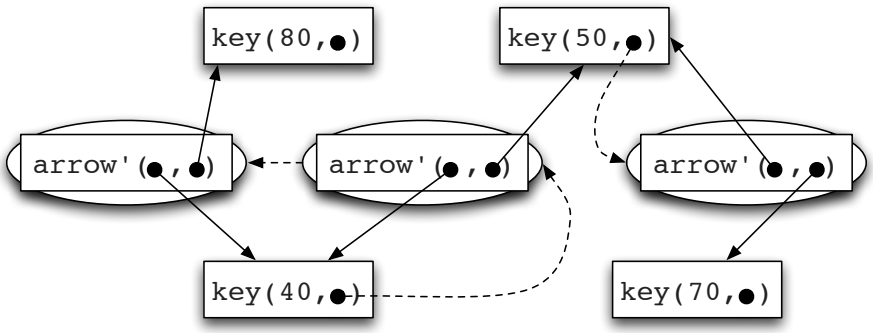
```
?- merge(1,80), merge(1,40), merge(1,50), merge(1,70).
```

evaluated as shown in Table 2. In the execution state (9), `arrow(40,50)` is the active constraint, whereas `arrow(40,80)` and `arrow(50,70)` are suspended in the constraint store. In the following derivation step, the `join` rule is triggered, and `arrow(40,80)` is retrieved from the store to serve as the partner constraint to match the rule's head.

Figure 2 illustrates two instances of this situation: (a) with indexing based on a hash table, and (b) with indexing based on attributed data. In the former case, retrieving the required partner constraint involves hashing the number 40 into the table, traversing the bucket list to find the appropriate bucket, and locating the constraint within the bucket. In the latter case, the internal representation `key(40,L)` provides direct access to the linked list containing `arrow(40,80)`. Clearly, using attributed data avoids the overhead of hashing into the table and of traversing the bucket list.



(a) Hashtable

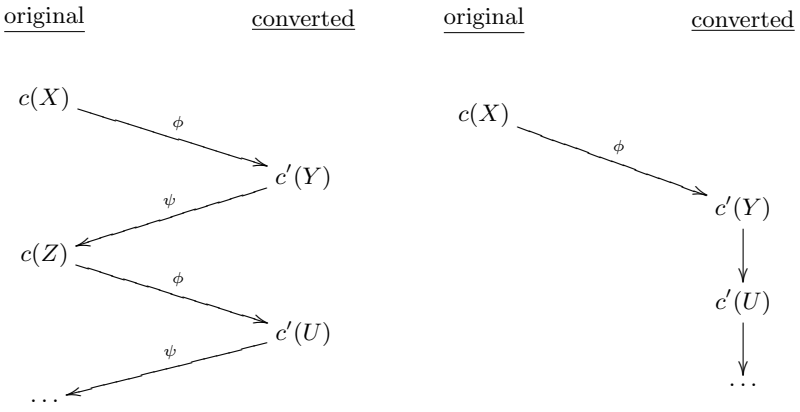


(b) Attributed Data

Fig. 2. Situation during the merge-sorting of 80, 40, 50 and 70.

4 Post-Processing

The experimental results in Section 5 indicate that the performance improvement obtained by better indexing is offset, or in some cases even surpassed, by the run-time overhead of applying the conversion functions. In this section we outline a transformation that statically eliminates most of this overhead; it was previously used to avoid similar performance issues in other transformations based on term flattening [10]. The effectiveness of the transformation is borne out by the benchmarks in Section 5.



(a) Actual Situation (b) Ideal Situation

Fig. 3. Transitions between the original and converted constraints

Alternating the conversions between the internal and external argument representations is a major source of runtime overhead. In a typical scenario (Figure 3(a)), an external value is converted into the internal representation and matched in a head of a rule, then it is converted back in the rule's body for calling a new constraint, converted again to match another rule, and so on. To avoid this overhead, the transformed rules should operate solely on the internal representation of the arguments, whereas the external values should be used only by the queries external to the programs. We propose a four-step rewriting procedure that aims to trigger this ideal scenario (Figure 3(b)). Execution of a program enhanced with the procedure consists of two phases:

(1) conversion of an argument's external value to the internal representation, and
(2) processing of the internal representation.

For all but the most trivial programs, we expect the runtime cost of (1) to be marginal with respect to the cost of (2).

Our rewriting procedure comprises the following steps.

Step 1: Make conversion explicit.
Unfold constraint calls according to the conversion rules.

Example 8. Consider the `join` rule from the program in Table 1:

```
arrow(X,A) \ arrow(X,B) <=> A < B | arrow(A,B).
```

After conversion, the rule has the form:

```
arrow'(XI1,AI) \ arrow'(XI2,BI) <=>
     X = ψ(XI1), X = ψ(XI2),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow(A,B).
```

By applying Step 1 to the above rule we obtain:

```
arrow'(XI1,AI) \ arrow'(XI2,BI) <=>
     X = ψ(XI1), X = ψ(XI2),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(φ(A),φ(B)).
```

We refer the reader to the work of Tacchella et al. [15] for the formal definition and correctness proof of unfolding of CHR rules.

Step 2: Eliminate identity conversion.
Apply the following equation from left to right:

$$\forall \bar{t} : \phi \circ \psi(\bar{t}) = \bar{t}$$

The transformation is valid based on the property that ϕ is the inverse of ψ.

Example 9. Applying Step 2 to the last rule in Example 8 yields:

```
arrow'(XI1,AI) \ arrow'(XI2,BI) <=>
     X = ψ(XI1), X = ψ(XI2),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(AI,BI).
```

Step 3: Convert external values of matchings to the internal representations.

Apply the equivalence from left to right:

$$\forall \overline{t_1}, \overline{t_2} : \psi(t_1) = \psi(t_2) \Leftrightarrow \overline{t_1} = \overline{t_2}$$

The transformation is valid based on the property that ψ is injective.

Example 10. Applying Step 3 to `X` in the rule from Example 9 yields:

```
arrow'(XI,AI) \ arrow'(XI,BI) <=>
     X = ψ(XI),
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(AI,BI).
```

Step 4: Clean up.
Drop unused conversion guards and refold the unfolded constraint calls that could not be simplified.

Example 11. Applying Step 4 to the rule in Example 10 yields:

```
arrow'(XI,AI) \ arrow'(XI,BI) <=>
     A = ψ(AI), B = ψ(BI),
     A < B | arrow'(AI,BI).
```

In general, these rewriting steps are not sufficient to enforce the ideal scenario of Figure 3(b). However, as the results in Section 5 show, they have good practical effects.

5 Evaluation

We implemented our approach in K.U.Leuven CHR [11] on SWI-Prolog [16]. The implementation consists of two components: (1) a pre-processor, which transforms a CHR program with key annotations into its converted form, and (2) the actual code generator of the CHR compiler, which generates attributed data indexing instructions and emits definitions for the conversion functions. Note that the pre-processor performs the transformations for all keys simultaneously rather than sequentially. In doing so, it avoids generating multiple intermediate conversion rules for constraints involving more than one key type.

We have evaluated our implementation on several standard CHR benchmarks. All run times, given in seconds for the original programs and relative to the original for the transformed versions, were measured on a MacBook Pro Intel Core Duo 1.83 GHz, with 1 GB RAM. Our benchmark suite includes the following programs:

- `chrg`, a CHRg parser with an exponential number of parses
- `dijkstra`, Dijkstra's shortest path algorithm
- `fib`, computation of fibonacci numbers, with effective memoing
- `fib2`, computation of fibonacci numbers, with ineffective memoing
- `mergesort`, mergesort algorithm
- `flat_ram`, RAM machine interpreter, flattened by symbol specialization [10]
- `reverse`, reversing chain of list cells
- `turing`, Turing machine simulator, running the *copy* program
- `uf_opt`, optimal union-find algorithm

	index representation				
benchmark	hash table	attr. data	relative	post-processed	relative
`chrg`	2.17	2.10	96.8%	1.58	72.8 %
`flat_ram`	4.69	4.31	91.9%	2.50	53.3%
`mergesort`	3.33	4.89	146.8%	1.85	55.6 %
`reverse`	2.55	3.25	127.4%	1.92	75.3%
`uf_opt`	0.34	0.38	111.8%	0.25	73.5%
`turing`	1.50	1.31	87.3%	1.19	79.3%
`wfs`	1.32	0.88	66.7%	0.85	64.4%
`fib`	1.24	1.53	123.4%	1.52	122.6%
`fib2`	1.61	1.30	80.7%	1.05	65.2%
`dijkstra`	2.26	4.52	200.0%	3.53	156.2%

Table 3. K.U.Leuven CHR run times (in sec.) for attributed data benchmarks

– `wfs`, well-founded semantics algorithm.

For each benchmark, we have manually added the `as_chr_key` annotations for the argument positions according to the following prioritized guidelines:

1. If two head constraints share more than one variable, we do not annotate the corresponding argument positions of those variables, because they are better served by multi-argument indexing. For instance, consider a rule head of the form `c(X,Y), d(Y,X)`. Although indexing on a single argument, i.e., using either `X` or `Y`, does work, indexing on the combination of both arguments is usually more efficient.
2. If two head constraints share exactly one variable, we annotate the corresponding argument positions of that variable with the same key.
3. If no variables are shared, no index is required.

Most benchmarks require a single key type. The exceptions are `ram_flat` and `turing`, each using two key spaces to represent instruction labels/states and data addresses, and `wfs` with separate key spaces for atoms and clause identifiers.

Table 3 lists the run-time results of exploiting attributed data in K.U.Leuven CHR, measured for plain hash tables, plain attributed data, and attributed data with post-processed rule bodies.

The first block of seven benchmarks clearly shows the positive effects of our approach. Although, the attributed data used alone causes a slow-down (up to about 50% for `mergesort`), when augmented with post-processing, it improves the run time by 20% to 50%.

The second block illustrates two cases of slow-downs incurred by the use of attributed data. The first benchmark, `fib`, performs one hash-table lookup per new constraint, and the initial attributed data conversion preserves that count. Hence, the attributed data manipulation is pure overhead (25%). The second benchmark, `fib2`, modifies the simpagation rule of `fib`:

```
fib(N,F1) \ fib(N,F2) <=> F1 = F2.
```

into a simplification rule:

```
fib(N,F1), fib(N,F2) <=> F1 = F2, fib(N,F1).
```

This modification causes the parameter `N` to be reused in the new call in the rule's body. As a consequence, attributed data requires only one hash-table lookup for every two new constraints, which results in a visible speed-up.

The second slow-down, in `dijkstra`, results form a limitation of our current implementation, which does not allow multi-argument indices involving attributed data arguments. For this benchmark, such a multi-argument index would be more efficient than a single-argument attributed data index.

6 Related Work

Several programming languages define features that resemble our concept of attributed data. The `as_chr_key` annotation is related to (primary, secondary and foreign) keys in database tables and indexing declarations in some Prolog systems.

The conversion function ϕ relates to *hash consing*—a technique, originated in Lisp, for mapping to and representing terms by unique (hash) values. Although the main aim of hash consing is to reduce memory consumption by increased sharing, it is also used to speed up equality tests.

The *solver types* facility of Mercury [2] also imposes a dual view of constraint arguments. The internal representation type is defined by the library programmer, rather than generated automatically. Externally, the solver type is abstract, but coercion functions should be provided for external representations. Finally, a folklore optimization technique in C/C++ adds (pointer) fields to structures to compactly represent lists (and other data types) that contain them.

7 Conclusion

We have presented attributed data—a new term representation that facilitates improving the efficiency of CHR indexing at a high level. A complementary post-processing procedure compensates for possible overhead of conversions between the new representation and the standard representation of Prolog terms.

Our technique has been implemented for the K.U.Leuven CHR system on SWI-Prolog. Evaluation on a set of benchmarks shows that using attributed data enables performance improvement, and that post-processing is critical to fully realize this potential.

As a further optimization of the approach, we could directly expose the abstract key types in the situations when there is no preference for the external argument representation. For example, programmers often use variables and integers as identifiers in CHR constraints. The nature of the data type is of no concern, as long as it supports unique value creation and value comparison. The appropriate choice of the abstract key type could eliminate unnecessary indirections of attributed variables or hash tables.

Two other interesting avenues for future work involve introducing support for automated inference of key type annotations, and extending attributed-data indexing to combinations of multiple arguments.

Acknowledgments

We are grateful for the helpful comments of the anonymous reviewers.

References

1. Slim Abdennadher and Michael Marte. University course timetabling using constraint handling rules. *Applied Artificial Intelligence*, 14(4):311–325, 2000.
2. Ralph Becket et al. Adding constraint solving to Mercury. In *8th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2006.
3. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *20th International Conference on Logic Programming (ICLP)*, pages 90–104, 2004.
4. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
5. Thom Frühwirth. As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs. *Electronic Notes in Theoretical Computer Science*, 59(3), 2002.
6. Christian Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. Technical Report TR-92-23, Austrian Research Institute for Artificial Intelligence, Vienna, Austria, 1992.
7. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), April 2000.
8. Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming*, 5(Issue 4 & 5):503–531, 2005.
9. Beata Sarna-Starosta and C.R. Ramakrishnan. Compiling Constraint Handling Rules for Efficient Tabled Evaluation. In *9th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2007.
10. Beata Sarna-Starosta and Tom Schrijvers. Transformation-based indexing techniques for constraint handling rules. In T. Schrijvers, F. Raiser, and T. Frühwirth, editors, *CHR '08*, RISC Report Series 08-10, University of Linz, Austria, pages 3–18, Hagenberg, Austria, July 2008.
11. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In *1st Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.
12. Tom Schrijvers and Thom Frühwirth. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.
13. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules (CHR)*, pages 3–17, 2005.
14. Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transactions on Programming Languages and Systems*, 27(6):1216–1269, 2005.
15. Paolo Tacchella, Maurizio Gabbrielli, and Maria Chiara Meo. Unfolding in chr. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 179–186, 2007.
16. Jan Wielemaker. SWI-Prolog release 5.6.0, 2006. http://www.swi-prolog.org/.

About Redundant Sudoku Rules

Bart Demoen[1], María García de la Banda[2]

[1] Department of Computer Science, K.U.Leuven, Belgium
[2] Monash University, Australia

Abstract. The rules of Sudoku are often specified by twenty seven `all_different` constraints, which we will call *big* rules. It is shown that many subsets of six of these big rules are redundant, and that six is maximal. Any `all_different` constraint can be specified as a (quadratic sized) set of binary inequalities, which we will call *small* rules. The redundancy of small rules is also investigated.

1 Introduction

A very common formulation of the 3x3 Sudoku [1] rules is one in which (a) all numbers in the puzzle are said to be in [1..9] and (b) the numbers in each column, row and box are said to be different. A CLP-program would typically code the latter as 27 `all_different` constraints of 9 variables each: we will refer to these constraints as *the big rules*. We will use often the word *Sudoku* in italics as an abbreviation of the 27 big rules.

Any `all_different` constraint can also be formulated in terms of binary inequality constraints. For example, `all_different([A,B,C,D])` is the conjunction of the constraints `A` $\neq$ `B`, `A` $\neq$ `C`, `A` $\neq$ `D`, `B` $\neq$ `C`, `B` $\neq$ `D` and `C` $\neq$ `D`. We will refer to these binary $\neq$-constraints as the *small rules*. As we will see, *Sudoku* can be rewritten as 810 different small rules.

For most people it is intuitively clear that some of the small rules must be *redundant*, i.e., implied by the others. It might be less obvious which ones are redundant, let alone how many. On the other hand, often, the same people are convinced that not a single big rule is redundant. These two issues form the topic of this paper: what is the largest redundant set of big rules, and the largest redundant set of small rules.

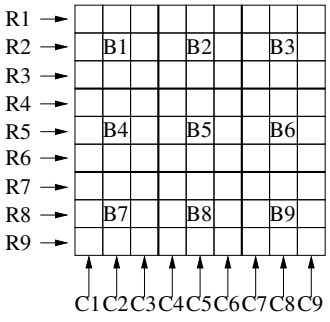
The paper proceeds as follows. We start by revising some Sudoku terminology in Section 2. In Section 3 we introduce a pictorial representation of big Sudoku rules that will make proofs much easier. In Section 4 we prove two positive lemmas that can be used to easily reason about the redundancy of subsets of the 6 big rules. In Section 5 we describe a Prolog program that systematically applies the two positive lemmas to find *all* redundant sets of six big rules. While doing this we detect 7 *negative* lemmas. This results in a complete classification. In Section 6 we turn to the study of sets of seven big rules and show that none of them are redundant. Again, our Prolog program discovers a new negative lemma, whose proof is also presented. In Section 7, we show that at least 20% of the small rules can be redundant, and conjecture that no more are possible. Finally, in Section 8 we conclude and provide some historical notes.

2 Terminology

The usual formulation of Sudoku refers to 27 regions on the board:

- the 9 rows, denoted as R1 ... R9
- the 9 columns, denoted as C1 ... C9
- the 9 boxes, denoted as B1 ... B9

as shown in the picture below. By an abuse notation we also write R3 if we mean that the `all_different` constraint on row R3 is enforced or true.



Individual cells of a puzzle are denoted as A11, A12 ... A99. We use the word horizontal (vertical) *chute* to refer to three horizontal (vertical) boxes - the usual term is band (stack). For instance, {B2, B5, B8} denotes a vertical chute. In the usual specification of Sudoku, each cell is involved in 20 small rules: 8 in the same box, 6 more in the same row and 6 more in the same column. Since there are 81 cells, and each rule is posted twice, there are in total 810 different small rules. When a set S of constraints is equivalent to the conjunction of all the Sudoku constraints, we use the short phrase: S is *Sudoku*.

3 Representing Sets of Sudoku Rules

Given the above notation, we could easily represent sets of rules as, for example, $\{R1, R2, R3, B1, B5, B9\}$. However, this only works well for small sets. Since we will be dealing mostly with sets of more than 20 big rules, we develop a graphical representation. Our representation always shows the borders of the boxes of a Sudoku board. A missing column, row or box rule will appear as a shaded column, row or box, respectively. Figure 1 shows an example.

Fig. 1. The left shows a Sudoku board with all 27 rules, the right shows one with only 22: $\{C1, C3, C4, C5, C6, C7, C8, C9, R1, R2, R3, R4, R6, R7, R8, R9, B1, B3, B4, B6, B8, B9\}$

The pictures provide a quick and intuitive insight into which rules are present and which are not. Note that the absence of a rule does not mean it is violated, simply that it has not been specified in the associated model.

We will also use $Missing(n)$ to denote the set $\{S \subseteq Sudoku | \#S = 27 - n\}$. For example, every element of $Missing(5)$ has 22 big rules.

We can in a similar way show the set of rules defined only for a given a chute: this is illustrated in Figure 2.

Fig. 2. $\{R1, R2, R3, B1, B2, B3\}$, $\{R1, R2, R3, B1, B3\}$ and $\{R1, R3, B1, B3\}$

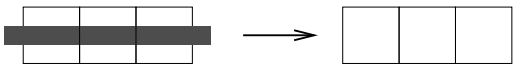
4 Two Constructive Lemma's

Lemma 1.

Proof The pictured lemma says that the set of rules $\{R1, R2, R3, B1, B3\}$ implies also $B2$. The proof can be given by trying to fill the chute with 27 numbers, so that  is fulfilled (and of course with all numbers being in [1..9]). Consider first where we can place a 5. There must be exactly one 5 in R1, one 5 in R2 and one 5 in R3, so there are in total three 5's in the chute. There is also exactly one 5 in B1, and one 5 in B3, so the remaining 5 must be in B2. And this holds also for the other numbers, so B2 is satisfied. ■

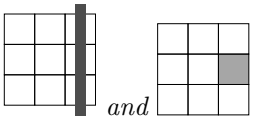
The dual of Lemma 1 is Lemma 2: we leave its proof to the reader.

Lemma 2.



The lemma  is clearly trivial and we can state the following corollary by composing the above two lemmas:

Corollary 1.

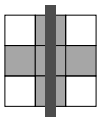
 *and* *are both Sudoku.*

Proof Glue together twice the trivial lemma with Lemma 1 or Lemma 2, and obtain the result immediately. ■

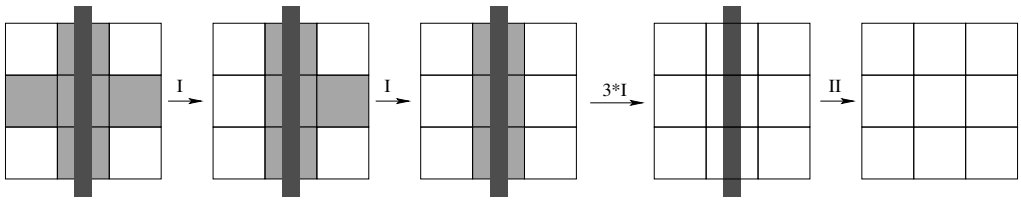
It is now clear that every single big rule is, by itself, redundant !

The two lemma's really are *constructive*: they show how to derive one new big rule from a set of big rules. The following two theorems exploit that constructive power to reason about redundancy.

Theorem 1.

 *is Sudoku.*

Proof We prove this by repeatedly using Lemma 1 and 2 as follows:



where the labels of the arcs indicate which lemma is used, and how many times it is used. ■

The proof of Theorem 2 is also left to the reader.

Theorem 2.

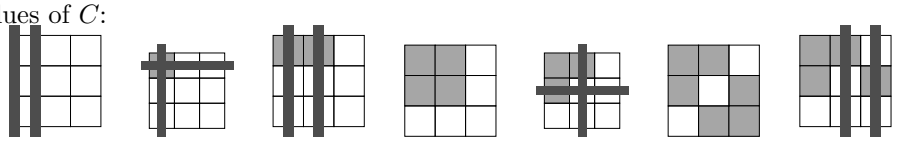
 *is Sudoku.*

The theorems show that at least two elements of $Missing(6)$ are large enough for representing *Sudoku*. Note that there are many symmetric versions of the theorems, but we have of course chosen the ones that are visually most pleasing. In the next section we will investigate all elements of $Missing(6)$ that are redundant.

5 A Full Classification of $Missing(6)$

Lemmas I and II give us a way to increase the number of big rules, as shown in the proof of Theorem 1. We use this in the algorithm of Figure 3 (where n is a parameter of the algorithm) to determine all elements of $Missing(6)$ that are redundant (i.e., those for which the algorithm will output *S is Sudoku*).

While the number of elements in $Missing(6)$ is relatively small (296,010), it is much smaller if we eliminate from $Missing(6)$ those elements that can be obtained from the spatial symmetries of Sudoku. We have programmed Algorithm I in Prolog (of course) and run it over the (reduced) set of $Missing(n)$ for values of n in 2..6. To our surprise, the algorithm only got stuck for the following seven values of C:



```
for each S ∈ Missing(n) do
        C ← copy(S)
        while Lemma 1 or Lemma 2 is applicable to C
                apply it to C
        if C == Sudoku
                then output S is Sudoku
                else output S got stuck in C
```

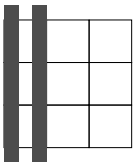
Fig. 3. Algorithm I

It is clear that if a C is not *Sudoku*, then any subset of C is not *Sudoku* either, i.e., the S from which C was derived by lemma application, is not *Sudoku*. So we set off to prove that the above sets of big rules are not *Sudoku*. This resulted in the seven negative lemmas provided in the next section.

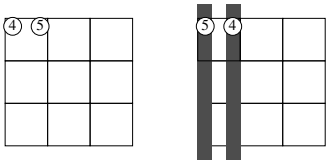
5.1 Seven *Negative* Lemmas

For each of the configurations C above, we can prove the negative result that C is not *Sudoku*. The proof of each lemma consists of a simple picture whose details we explain for the first proof. We expect the reader to work out the details for the others.

Lemma 3.


is not Sudoku.

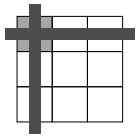
Proof



Explaining the proof: consider a completely filled out Sudoku puzzle that satisfies the full set of big rules, and which has a 4 in A11 and a 5 in A13. This situation is depicted in the left part of the proof. If we swap the 4 and 5 we obtain the picture on the right, where the shadows indicate the only two big rules that are violated by the swap. Clearly, the filled out puzzle with the two numbers swapped is not a valid solution to the full set of rules, but it only violates C1 and C3. That proves the lemma.

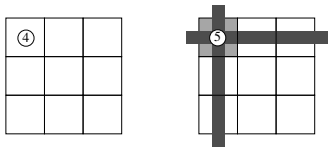
The statements and proofs of the other six negative lemmas are similar: we always start from a completely filled out Sudoku puzzle, with numbers 4 and 5 at particular places. It is easy to check that such an initial puzzle indeed exists.

Lemma 4.



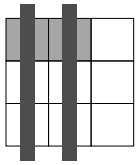
is not Sudoku.

Proof



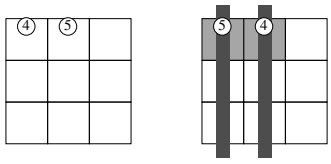
■

Lemma 5.



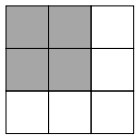
is not Sudoku.

Proof



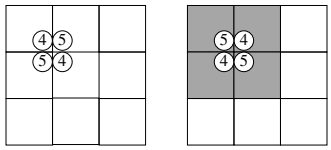
■

Lemma 6.



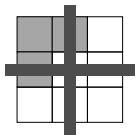
is not Sudoku.

Proof



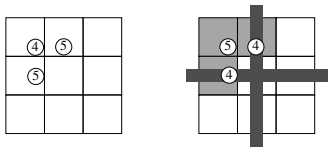
■

Lemma 7.



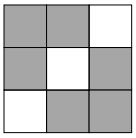
is not Sudoku.

Proof



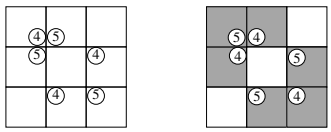
■

Lemma 8.



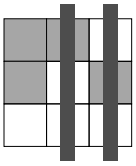
is not Sudoku.

Proof



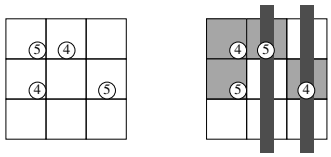
■

Lemma 9.



is not Sudoku.

Proof



■

5.2 Using the Negative Lemmas

We can increase the accuracy of our first algorithm by noticing that subsets of non-*Sudoku* are also non-*Sudoku*:

```
for each S ∈ Missing(n) do
        C ← copy(S)
        while Lemma 1 or Lemma 2 applicable to C
                apply a Lemma to C
        if C == Sudoku
                then output S is Sudoku
                else output S is not Sudoku
```

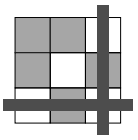
Fig. 4. Algorithm II

We have run the algorithm with $n = 6$ and, for each element S of (the reduced) $Missing(6)$, we have modified the program to generate a picture with some annotations. These are provided in the Appendix. It turns out there are 40 different elements in (the reduced) $Missing(6)$ that are *Sudoku*.

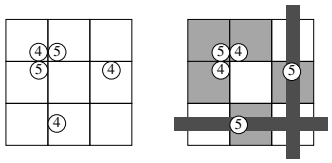
6 No Element in $Missing(7)$ is *Sudoku*

When run with $n = 7$, Algorithm I gets stuck in only one new set of big rules. This results in one more negative lemma:

Lemma 10.


is not Sudoku.

Proof



Running Algorithm II for $n = 7$ shows that no element in $Missing(7)$ is *Sudoku*. This means that six is the maximal size of a redundant set of big rules.

7 Redundant Sets of Small Rules

Recall that *Sudoku* can also be specified by 810 small rules, which are obtained by expanding the big rules to binary inequalities. We will denote the set of all small rules $Sudoku_{small}$. In this section we will briefly study the redundancy of sets of the small rules. In analogy with $Missing(n)$ which was meant for big rules, we introduce the notation $Missing_{small}(n) = \{S \subseteq Sudoku_{small} | \#S = 810-n\}$.

The output of Algorithm II for $n = 6$ (shown in the Appendix) indicates that the largest n for which an element of $Missing_{small}(n)$ is known to be $Sudoku$ is 162: indeed, the big rules of Theorem 2 give rise directly to 648 small rules. We name this set of small rules $Small_{648}$. Theorem 2 now can be read as: $Small_{648}$ *is Sudoku*.

We have used $Small_{648}$ in an experiment which needed two more ingredients. The first is a large set of difficult Sudoku puzzles. For this we took the set from Gordon Royle's website [2] who has collected a large set (more than 50.000) distinct and *minimal* Sudoku puzzles with 17 given entries. Here minimal means that while with the 17 givens the puzzle has a unique solution, if any one given is removed the puzzle has more than one solution. We refer to this set as GR.

The second ingredient is a way to transform a given Sudoku solver P to take into account less small rules. Because of the symmetries, this needs to be done only 11 times, so we did that by hand. Our P was adapted from an example CLP(FD) program from the B-Prolog [3] distribution and run under B-Prolog. Since we did not know in advance how many examples we would run, we wanted a fast CLP(FD) system. However, the programs also run in e.g., SICStus Prolog.

for each $s \in Small_{648}$ **do**
    $S \leftarrow Small_{648} \setminus \{s\}$
    transform P to take into account only S
    run P on every problem $p \in GR$
    if *some* p has more than one solution
        then output S *is not Sudoku*
        else output S *maybe is Sudoku*

Fig. 5. Algorithm III

It turns out that the algorithm could always decide that S is not $Sudoku$. This proves that the set $Small_{648}$ forms a locally minimal set of small rules equivalent to Sudoku. Another way to phrase this result is: *Sudoku only needs 80% of its small rules.*

Since the number of example problems that needed to be tried before the modified program P finds more than one second solution is so small, we dare to conjecture the following:

Conjecture 1. No element of $Missing_{small}(n)$ is $Sudoku$ for $n > 162$.

It is clear that this conjecture should not be attacked with blind and brute force.

8 Discussion and Conclusion

On 18 May 2008, in rec.puzzles, the following message was posted:

```
Quick question that I though someone here might know the
answer to - or be able to suggest a different forum.
```

```
If you have a completed sudoku grid, you supposedly need
to check all 9 rows, then all 9 columns, then all 9 boxes
to validate that it has been completed correctly. But it's
pretty obvious that the grid can be validated with somewhat
less checking. For instance, if each of the boxes has been
checked and the first 2 rows are checked, there's no need
to check the 3rd row.

So what's the minimum amount of checking that needs to be
done to show that a completed 9x9 grid is valid?
```

Before we even saw this post[1], other people tried to answer, but it was clear that none had the full picture presented here. Still, the original poster had figured out our Theorem 2 on his own, but got stuck there. It was quite satisfactory that our research started out of curiosity and ended up being of use to someone !

Redundant constraints are often important for a solver to be able to find a solution efficiently. So it might seem a futile exercise to find out whether a particular constraint satisfaction problem has redundant constraints. However, understanding better redundant Sudoku rules might give insight in why the 16-17 problem is so hard. Also, studying redundant Sudoku constraints is interesting in itself, because it seems not generally known that so many of the big and small Sudoku rules are redundant. On the other hand, it is difficult to *add* rules and stay *Sudoku*: it is clear that any additional small inequality rule changes the game.

During our discussion, one particular constraint was considered sacred: all cells have a value in $1..Max$, with $Max = 9$. It is clear that one cannot maintain any big constraint for Max strictly smaller than 9. But it seems worthwhile to investigate $Max = 10$ (or more) for the usual Sudoku constraints and for particular givens: the uniqueness of the solutions under such circumstances could result in a better understanding of Sudoku. It is also clear that our techniques can be readily applied to the investigation of Sudoku puzzles of different sizes. In particular, the generalization of our lemmas 1 and 2 to other sizes is not difficult, and the algorithms remain correct. Still, for large sizes, they might be not as helpful.

Acknowledgements Lots of this work was done while the first author was on a research visit at Monash University, and enjoying the Stuckey hospitality in Apollo Bay and Elwood, Australia. Many thanks for an enjoyable stay.

References

1. Wikipedia: Sudoku (2008) http://en.wikipedia.org/wiki/Sudoku.
2. Royle, G.: Minimal sudokus with 17 givens (2008) http://people.csse.uwa.edu.au/gordon/sudokumin.php.
3. Zhou, N.F.: B-Prolog users manual, version 6.8. Technical report, Afany Software (2005)

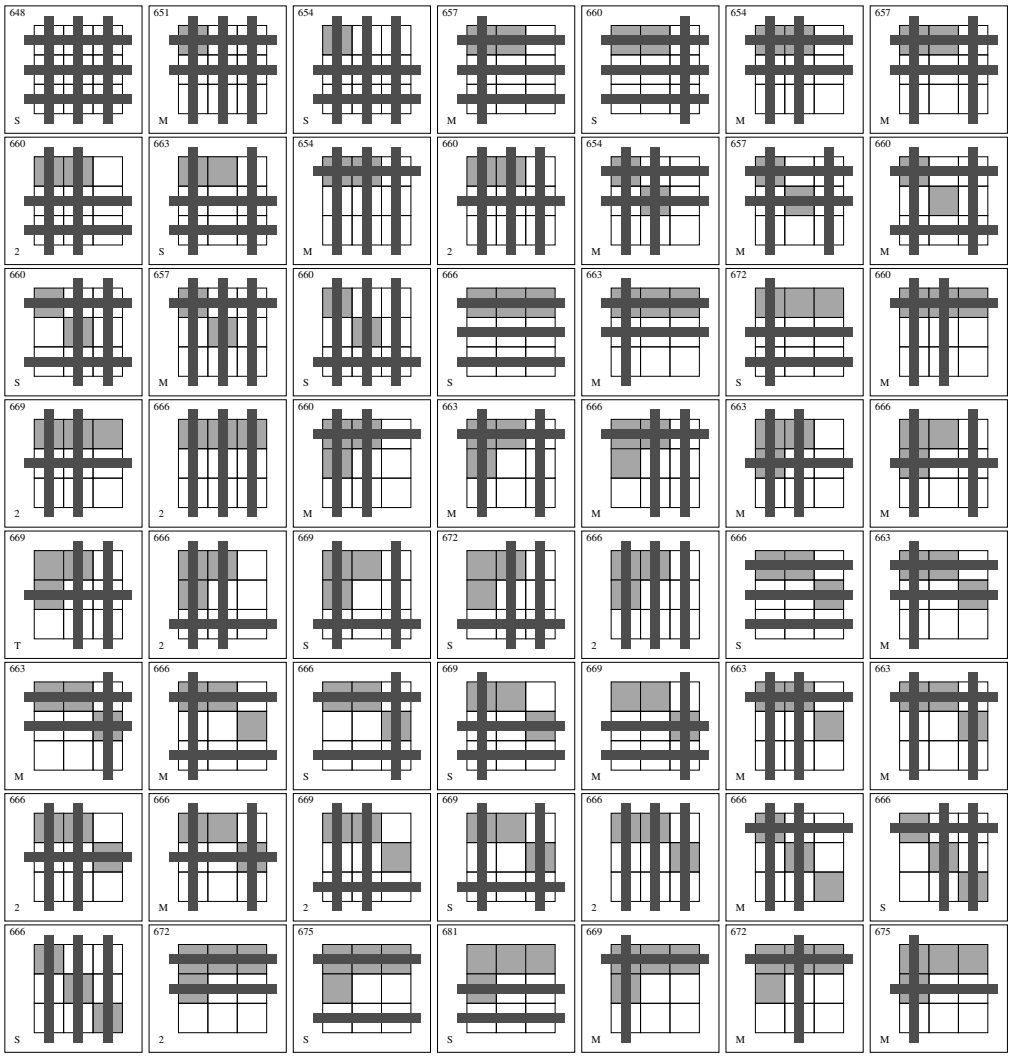
[1] We don't read rec.puzzles, but someone who knew about our results made us aware of the particular post.

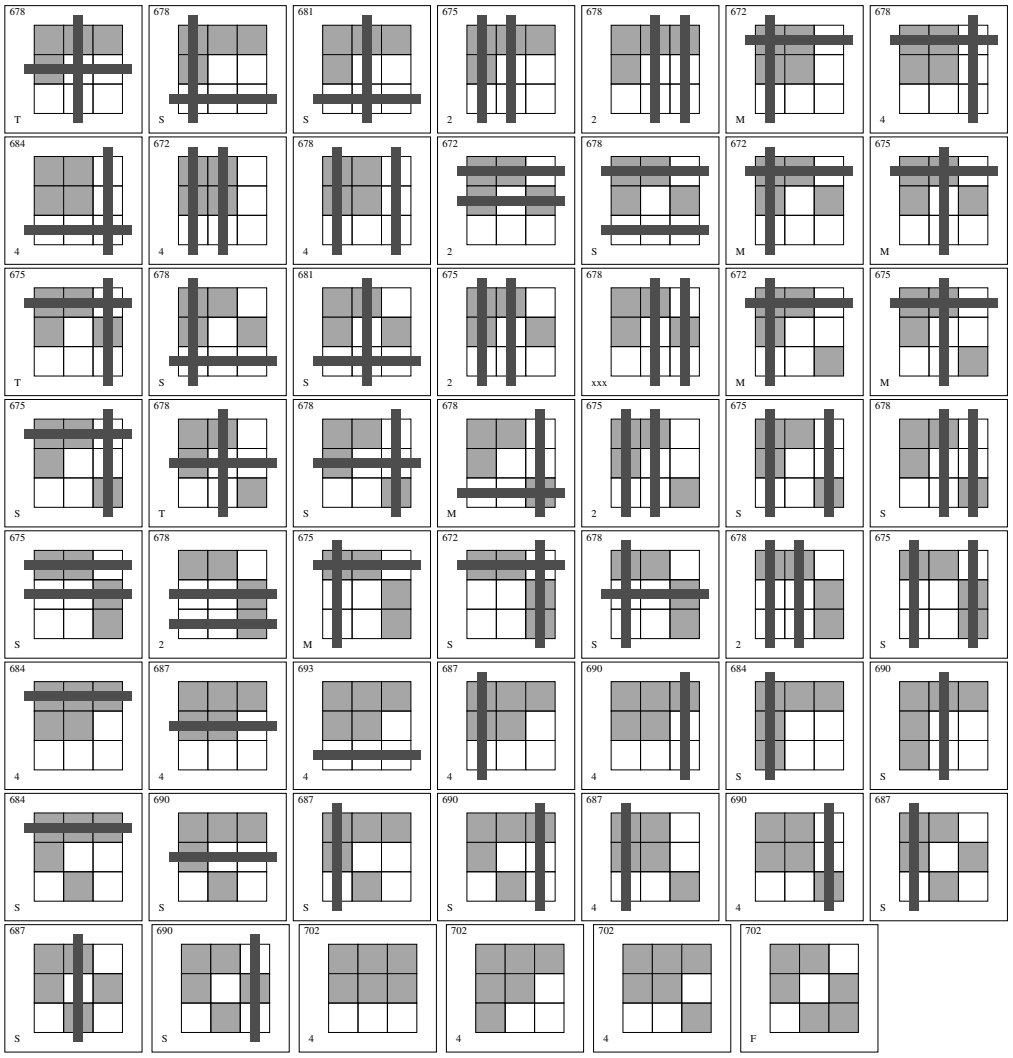
Appendix

All Elements of $Missing(6)$

Each element of $Missing(6)$ is annotated as follows:

- upper left corner: indicates the number of small rules that result from expanding the big rules displayed
- lower left corner: S means $Sudoku$; the other characters indicate in which configuration algorithm I got stuck; M corresponds to Lemma 4, 2 corresponds to Lemma 5, 4 corresponds to Lemma 6, T corresponds to Lemma 7, xxx corresponds to Lemma 9, and F corresponds to Lemma 8,




678 T
678 S
681 S
675 2
678 2
672 M
678 4
684 4
672 4
678 4
672 2
678 S
672 M
675 M
675 T
678 S
681 S
675 2
678 xxx
672 M
675 M
675 S
678 T
678 S
678 M
675 2
675 S
678 S
675 S
678 2
675 M
672 S
678 S
678 2
675 S
684 4
687 4
693 4
687 4
690 4
684 S
690 S
684 S
690 S
687 S
690 S
687 S
690 4
687 4
690 S
687 S
690 S
702 4
702 4
702 4
702 F