Improving the Compilation of Prolog to C Using Type Information: Preliminary Results*

J. Morales
jfran@clip.dia.fi.upm.es

M. Carro
mcarro@fi.upm.es

Computer Science School
Technical University of Madrid
Boadilla del Monte, E-28660, Spain

Abstract. We describe the current status and preliminary results of a compiler of Prolog to C. This compiler can use high-level information on the initial Prolog program in order to optimize the resulting C code, which is then fed into a off-the-shelf C compiler. The basic translation process basically mimics the unfolding of a C-coded bytecode emulator with respect to the bytecode corresponding to the Prolog program. This allows reusing a sizeable amount of the associated machinery: ancillary pieces of C code, data definitions, memory management routines and areas, etc. We evaluate the performance of programs compiled both with and without type information.

1 Introduction

Several techniques for implementing Prolog have been devised since the interpreter originally developed by Colmerauer and Roussel [Col93], many of them aimed at achieving more speed. A good survey of part of this work can be found in [Van94]. A rough classification of implementation techniques for Prolog (extensible to other languages) is the following:

- Interpreters (such as C-Prolog [Per87] and others), where a slight preprocessing or translation might be done before program execution, but the bulk of the work is done at runtime by the interpreter.
- Compilers to *bytecode* and their interpreters (often called emulators). The compiler produces a relatively low level code in a special purpose language, but an interpreter of such code is still needed. Most emulators are based on the Warren Abstract Machine (WAM) [War83,AK91], but other proposals exist [Tay91,KB95]. Highly optimized emulators [CDRA00] offer very good performance.
- Compilers to a lower-level language, which generate an output requiring little or no additional support to be executed. Ideally, the compiler should generate directly machine code. Examples of this are the Aquarius system [VD92], the SICStus Prolog [Swe99] compiler (for some architectures), the latest

* The authors have been partially supported by the Spanish MCYT Project TIC 99-1151 *EDIPIA* and the EU ESPRIT Project 2001-34717 *Amos*

BimProlog compilers [VDW87,Mar93], the Gnu Prolog compiler [DC01], and the Mercury. [SHC96] compiler[1]

Each solution has its advantages and disadvantages. Generation of low level code promises faster programs at the expense of using more resources during the compilation phase. Interpreters have smaller load/compilation time and are a good solution due to their simplicity when speed is not a priority; executing the same Prolog code in differente architectures boils down (in principle) to recompiling the interpreter. Compilers are more complex than interpreters, and the difference is much more acute if some form of code analysis is performed as part of the compilation, which impacts development time. Emulators place themselves in some intermediate point, retaining the portability of interpreters, since only the emulator has to be recompiled for every target architecture (bytecode is usually architecture-independent).

In this paper we will summarily describe work on progress on a compiler of Prolog to C, together with a scheme to optimize the resulting code using higher-level information on the source program. These optimizations can be used to tackle lower-level issues, and therefore exceed what can be expressed solely by means of Prolog-to-Prolog transformations. Note that the selection of C as target (low-level) language does not, in practice, prevent portability, as C compilers exist for most architectures. Besides, C is low-level enough as to apply optimizations to its generation which will eventually make into the final executable code in a form known beforehand, therefore offering a good compromise between speed and portability.

2 Issues on Compiling Prolog to Lower Level Languages

Making as much work as possible at compile time in order to avoid run-time overhead is expected to bring more speed to a system: native code has all the odds to be faster than C, C has the same relationship with a bytecode emulator, and a bytecode emulator with an interpreter. Additionally, code optimization can be put to work at all levels —e.g., Prolog itself [Win89,PGH97], WAM code [FD99], lower-level code, and native code. However, optimizations performed at a higher language level are implicitly carried over onto lower levels, while new optimizations can be introduced as we approach native code level.

A practical matter is that compilers to native code need architecture-dependent back-ends. This may make porting and maintaining them a non-trivial task. Systems as Gnu Prolog try to avoid the mousetrap by using an intermediate “mini-assembler” code, easy to translate into machine code for different architectures. But it requires, anyhow, different back-ends for different architectures. Besides, recent performance evaluations [DC01] show that well-tuned emulator-based Prolog systems can beat, at least in some cases, Prolog compilers which generate machine code directly.

[1] Although Mercury is not a Prolog compiler, the source language is close enough as to be mentioned here.

A practical reason to compile to C is the availability of good C compilers for most architectures, which will tackle the task of generating executable code. In our case, the possibility of reusing components of an already emulator-based existing system[2] is a practical advantage: by adopting the same scheme for memory areas, data tagging, etc., existing fragments of C code (builtins, low-level file and stream management, memory management and garbage collection routines, etc.) can be used by the new compiler, which only has to replace the WAM emulator.

The difference with other, similar systems which compile to C comes from using compile-time information regarding determinacy, types, instantiation modes, etc. This information is expressed by means of a well-defined assertion language [PBH00], and provided either by the user or by automatic global analysis tools [HBPLG99]. For example, `wamcc` (a Gnu Prolog forerunner), which generated C, did not use extensive analysis information (but it included clever tricks which in practice tied it to a single C compiler, `gcc`); Aquarius [VD92] used analysis information at several compilation stages, but it generated directly machine code, and it was therefore difficult to port and maintain. Notwithstanding, it proved the power of using global information in a Prolog compiler.

A drawback of putting more burden on the compiler is that compile times grow, and compiler complexity increases. While this can turn out to a problem in extreme cases (specially if global analysis is made), incremental analysis and the aid of a module system [CH00] can help to alleviate it in practice. Moreover, global analysis is, in our proposal, not mandatory, and can be left to generate final executables. We expect that, as the system matures, the Prolog-to-C compiler itself (now in a prototype stage) will not be slower than a Prolog-to-bytecode compiler.

Another common issue in compiling to lower-level languages is the size of the final object code files, usually bigger than their bytecode counterparts, since single bytecode instructions correspond to several machine code instructions. Global information can be used to reduce this size difference by specializing C code, but additional means to reduce code size should be adopted in the generation of C code itself.

3 An Overview of the Compiler

The compilation process starts by a preprocessing phase which canonizes clauses (removing aliasing and structure unification from the head), and expands disjunctions, negations and if-then-else constructs. It also replaces `is/2` by explicit calls to arithmetic builtins and executes a simple, local analysis which gathers information about the type and freeness state of variables. Having this analysis in the compiler helps to improve the code even in the case that no external information is available. The next steps include the traslation of Prolog to WAM-

[2] Ciao Prolog [HBC+99] (`http://clip.dia.fi.upm.es/Software/Ciao`), a SICStus Prolog 0.5 derivative which we are using as development platform.

put_variable(I,J)	⟨uninit,I⟩ = ⟨uninit,J⟩
put_value(I,J)	⟨init,I⟩ = ⟨uninit,J⟩
get_variable(I,J)	⟨uninit,I⟩ = ⟨init,J⟩
get_value(I,J)	⟨init,I⟩ = ⟨init,J⟩
unify_variable(I[, J])	⟨uninit,I⟩ = ⟨modedep,J⟩
unify_value(I[, J])	⟨init,I⟩ = ⟨modedep,J⟩

Table 1. Representation of some WAM unification instructions with types

based instructions (also used by the Ciao Prolog emulator), splitting these WAM instructions into an intermediate low level code, and final traslation to C.

3.1 Typing WAM Instructions

WAM instructions dealing with data are internally handled with an enriched representation which encodes the possible instantiation state of its arguments. This helps in using type information, and also in generating and propagating low-level information regarding the abstract machine type and instantiation/initialization state of the variables (which is not seen at a higher level). Each unification instruction is represented as ⟨TypeX,MemX⟩ = ⟨TypeY,MemY⟩, where *TypeX* and *TypeY* refer to the classification of WAM-level types (see Figure 1), and *MemX* and *MemY* refer to the registers where these variables live.

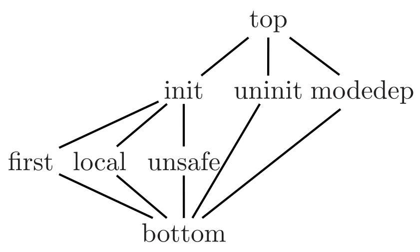

Fig. 1. Lattice of WAM types

Table 1 summarizes the aforementioned representation for some selected cases. The registers taken as arguments are the temporary registers (*x(I)*), the stack registers (*y(I)*) and the register for structure arguments (*n(I)*). The last one can be seen as the second argument which is implicit in the *unify_** WAM instructions. A number of other registers (`ok`, `temp`, ...) are available. **_constant*, **_nil*, **_list* and **_structure* WAM instructions are represented similarly.

The advantage of this representation is that it is more uniform than WAM instructions In particular, as more information is known about the variables, the associated (low level) types can be refined in order to generate more specific code. Using a richer lattice and initial information (Section 4), a more descriptive intermediate code is generated and used in the back-end.

3.2 Generation of Intermediate Low Level Language

WAM instructions are split into simpler ones, which are more suitable for optimizations and to simplify the generation of the final C code (and probably also the generation of code in languages of similar level). The deegre of complexity of the low-level code is similar to the one proposed in the BAM [VR90]. Table 2

no_choice	Mark that there is no alternative
first_choice(Arity, Alt)	Create a choicepoint
middle_choice(Arity, Alt)	Change the alternative
last_choice(Arity)	Remove the alternative
complete_choice(Arity)	Complete the choice point
cut_choice(Chp)	Cut to a given choice point
push_frame	Allocate a frame on top of the stack
complete_frame(FrameSize)	Complete the stack frame
modify_frame(NewSize)	Change the size of the frame
pop_frame	Deallocate the last frame
recover_frame	Recover after returning from a call
ensure_heap(CS, Amount, Arity)	Ensure that enough heap is allocated. (CS indicates completion status of the choice point)

Table 2. Choice, stack and heap management instructions

```
while (code != NULL)
   code = ((Continuation (*)(State *))code)(state);
```

```
Continuation foo(State *state) {
   ...
   state->cont = &foo_cont;
   return &foo2;
}
```

```
Continuation foo_cont(State *state) {
  ...
  return state->cont;
}
```

Fig. 2. The C execution loop and blocks scheme

shows the low level instructions related to the management of the stacks, which, at the moment, are very similar to those in the WAM.

Table 3 shows special control and data instructions. The *Type* argument which appears in several instructions is intended to reflect the type of the instruction arguments: for example, in the instruction *bind*, *Type* used to specify if the arguments contain a variable (and, if this is known, wether it lives in the heap, in the stack, etc.) or not. For the unification of structures, the use of write and read modes is avoided using a two-stream scheme (see [Van94] for an explanation and references). This scheme requires explicit control instructions, hence the existence of jump instructions (*jump*, *cjump*, and *ijump*). Jumps are performed to labels, marked as *global* (when they have to be stored in global data structures, such as the next alternative in a choicepoint) or *local*. For efficient indexing, the WAM instructions *switch_on_term*, *switch_on_cons* and *switch_on_functor* are also included, although the C back-end does not exploit them fully at the moment, resorting to a linear search in some cases. Failing is done by jumping to the special label *fail*. Builtins return an exit state in one argument, which is used to decide whether to backtrack or not.

Scheme of the Compilation to C The compilation produces C code which corresponds roughly to an unfolding of the initial bytecode emulator loop with

Data	
load(X, Type)	Load X with a term
trail_if_conditional(A)	Trail if A is a conditional variable
bind(TypeX, X, TypeY, Y)	Bind X and Y
read(Type, X)	Begin read of the structure arguments of X
deref(X, Y)	Dereference X into Y
move(X, Y)	Copy X to Y
globalize_if_unsafe(X, Y)	Copy X to Y ensuring safeness
globalize_to_arg(X, Y)	Copy X to argument register Y ensuring safeness
function(N, Is, O, H, Live)	Call a function
builtin(N, Is, Success)	Call a builtin
Control	
ijump(X)	Jump to the address stored in X
jump(Label)	Jump to *Label*
cjump(Cond, Label)	Jump to *Label* if *Cond* is true
switch_on_type(X, Var, Str, List, Cons)	Jump to the label that matchs the type of X
switch_on_functor(X, Table, Else)	
switch_on_cons(X, Table, Else)	
Conditions	
not(Cond)	Negate the *Cond* condition
test(Type, X)	True if X matchs *Type*
equal(X, Y)	True if X and Y are equal
erroneous(X)	True if X has an erroneous value

Table 3. Control and data instructions

respect to the bytecode. In the points where the emulated program counter changes a continuation passing using pointers to functions is used. Each block of bytecode, which begins in a label and ends in a instruction involving a possible jump, is translated to a C function with the state of the abstract machine as input argument and the next continuation as output argument. Schemes of the execution loop and of the functions code blocks are compiled into are shown in Figure 2. Additionally, we have implemented an optimization which reduces the function calling overhead, by using `goto` statements for jumps to local labels which are located in the same code block.

This scheme does not require using machine-dependent options of the C compiler or extensions to the ANSI C language (although machine-dependent optimizations can of course be given to the C compiler). Other systems, as [CDRA00] or [SHC96], take advantage of machine-dependent and non-portable constructs to obtain very good performance. However, one of the goals of our system is studying optimizations of a fixed compilation scheme with the use of program information, and portability and code cleanliness is given a high priority.

3.3 An Example: the `fact/2` Predicate

We will illustrate summarily the compilation stages with a sample implementation of the well-known factorial program (Figure 3). We have chosen it due to its simplicity, since the performance gain is not very high in this case. The code after canonizing and rewriting is shown in Figure 4. The WAM code corresponding to the recursive clause is in the leftmost column of Table 4, and the internal representation of this code appears in the same table, in the middle column. Note how variables are annotated using information which can be deduced from local inspection of the clause.

This WAM-like representation is translated to the low-level code shown in Figure 5 (ignore, at the moment, the shadowed and framed regions; they will be further discussed in Section 4). This code, which is quite low level now, is finally translated to C.

Executing `fact(100, N)` 20000 times took 3.32 seconds using the bytecode emulator, and 2.84 seconds with the C-compiled code C without external type information (a speedup of 1.16). We will see in the next section how this performance can be improved with the use of type information.

```
fact(0, 1).
fact(X, Y) :-
    X > 0,
    X0 is X - 1,
    fact(X0, Y0),
    Y is X * Y0.
```

Fig. 3. Factorial, initial code

```
fact(A, B) :-
    0 = A,
    1 = B.
fact(A, B) :-
    A > 0,
    builtin__sub1_1(A, C),
    fact(C, D),
    builtin__times_2(A, D, B).
```

Fig. 4. Factorial, after preprocessing

4 Improving Code Generation

The code generation seen so far can be greatly improved by using more information regarding, e.g., types, modes, determinacy, etc., as done in several other compilers. In the current version of the compiler we are using type information, which is expressed by means of the assertion language of Ciao Prolog [PBH00], as mentioned in Section 2. An example of assertions can be seen in the example of Section 4.2.

The generation of low-level code using additional type information uses an extended type lattice obtained by replacing the *init* element in the lattice in Figure 1 with the type domain in Figure 6. This information enriches the *Type* parameter of the low-level code.

4.1 Using Information inside the Compiler

During the compilation to low level code the information about the types of the variables is used to avoid unnecesary tests. The standard WAM compilation performs also some optimization, but it is based on ad-hoc per-clause implicit analysis, and it does not carry, e.g., information deduced from arithmetical builtins. By using richer type/mode information, a number of further optimizations can be done:

WAM code	Without Types	With Types
put_constant(0,2)	0 = ⟨uninit,x(2)⟩	0 = ⟨uninit,x(2)⟩
builtin_2(37,0,2)	⟨init,x(0)⟩ > ⟨int(0),x(2)⟩	⟨int,x(0)⟩ > ⟨int(0),x(2)⟩
allocate	builtin__push_frame	builtin__push_frame
get_y_variable(0,1)	⟨uninit,y(0)⟩ = ⟨init,x(1)⟩	⟨uninit,y(0)⟩ = ⟨var,x(1)⟩
get_y_variable(2,0)	⟨uninit,y(2)⟩ = ⟨init,x(0)⟩	⟨uninit,y(2)⟩ = ⟨int,x(0)⟩
init([1])	⟨uninit,y(1)⟩ = ⟨uninit,y(1)⟩	⟨uninit,y(1)⟩ = ⟨uninit,y(1)⟩
true(3)	builtin__complete_frame(3)	builtin__complete_frame(3)
function_1(2,0,0)	builtin__sub1_1(⟨init,x(0)⟩, ⟨uninit,x(0)⟩)	builtin__sub1_1(⟨int,x(0)⟩, ⟨uninit,x(0)⟩)
put_y_value(1,1)	⟨var,y(1)⟩ = ⟨uninit,x(1)⟩	⟨var,y(1)⟩ = ⟨uninit,x(1)⟩
call(fac/2,3)	builtin__modify_frame(3) fact(⟨init,x(0)⟩, ⟨init,x(1)⟩)	builtin__modify_frame(3) fact(⟨init,x(0)⟩, ⟨var,x(1)⟩)
put_y_value(2,0)	⟨init,y(2)⟩ = ⟨uninit,x(0)⟩	⟨int,y(2)⟩ = ⟨uninit,x(0)⟩
put_y_value(2,1)	⟨init,y(1)⟩ = ⟨uninit,x(1)⟩	⟨number,y(1)⟩ = ⟨uninit,x(1)⟩
function_2(9,0,0,1)	builtin__times_2(⟨init,x(0)⟩, ⟨init,x(1)⟩,⟨uninit,x(0)⟩)	builtin__times_2(⟨int,x(0)⟩, ⟨number,x(1)⟩, ⟨uninit,x(0)⟩)
get_y_value(0,0)	⟨init,y(0)⟩ = ⟨init,x(0)⟩	⟨var,y(0)⟩ = ⟨init,x(0)⟩
deallocate	builtin__pop_frame	builtin__pop_frame
execute(true/0)	builtin__proceed	builtin__proceed

Table 4. WAM code and internal representation without and with external types information. Underlined instruction changed due to additional information.

```
global(fact/2):
  first_choice(2,V1)
  ensure_heap(incompleted_choice,callpad,2)
  deref(x(0),x(0))
  cjump(not(test(var,x(0))),local(V3))
  load(temp2,int(0))
  bind(var,x(0),nonvar,temp2)
  jump(local(V4))
local(V3):
  cjump(not(test(int(0),x(0))),fail)
local(V4):
  deref(x(1),x(1))
  cjump(not(test(var,x(1))),local(V5))
  load(temp2,int(1))
  jump(local(V6))
local(V5):
  cjump(not(test(int(1),x(1))),fail)
local(V6):
  complete_choice(2)
  ijump(continuation)
global(V1):
  last_choice(2)
  load(x(2),int(0))
  builtin(numgt_2,[x(0),x(2)],ok)
  cjump(not(ok),fail)
  push_frame
  move(x(1),y(0))
  move(x(0),y(2))
  load(y(1),var(stack))
  complete_frame(3)
  function(sub1_1,[x(0)],x(0),0,1)
  cjump(erroneous(x(0)),fail)
  move(y(1),x(1))
  modify_frame(3)
  load(continuation,global(V0))
  jump(global(fact/2))
global(V0):
  recover_frame
  move(y(2),x(0))
  move(y(1),x(1))
  function(times_2,[x(0),x(1)],x(0),0,2)
  cjump(erroneous(x(0)),fail)
  deref(y(0),temp)
  deref(x(0),x(0))
  builtin(unify,[temp,x(0)],ok)
  cjump(not(ok),fail)
  pop_frame
  ijump(continuation)
```

Fig. 5. Low level code for the `fact/2` example (see also Section 4)

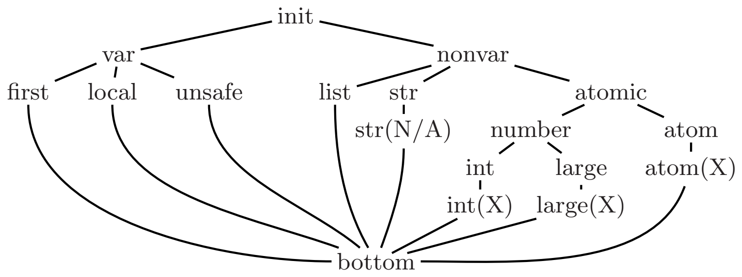

Fig. 6. Extended *init* subdomain

Unify instructions A call to the general *unify* builtin can be replaced by the more specialized *bind* instruction if one or both arguments are known to store variables. When arguments are known to be constants, a simple comparision instruction can be emitted instead.

Two-Stream Unification The unification of a register with a structure or constant needs some tests for determining the unification mode (read or write). Also, in read mode, an additional test is required to compare the register value with the constant or the structure functor. These tests can often be reduced to true or false if enough information is known about the variable.

Index Tree Generation Type information is also used to optimize the generation of index trees, used as part of the clause selection. An index tree is generated by selecting some literals from the beginning of the clause, mostly builtins and unifications, which give some amount of type/mode information. This is used to construct a decision tree on the types of the first argument.[3] When type information is available, the indexing tree can be optimized by removing some of the tests in the nodes.

Avoiding Unnecesary Variable Safeness Tests Another optimization made in the low level code with type information is the replacement of globalizing instructions for unsafe variables by explicit dereferences. When the type of a variable is nonvar, its globalization is equivalent to a dereference, which is faster.

Selecting Optimized Builtins Calls to builtins can also be optimized in the presence of type information. While the code of some of them is currently written in C and is external to the compiler, specialized versions can exist and be selected using the call patterns deduced from the type information. Currently, only arithmetic builtins are (partly) specialized, but this gives good speedups in many cases.

4.2 An Example: the `fact/2` Predicate with program information

Let us assume that it is known that `fact/2` (Figure 3) is always called with its first argument instantiated to a small integer (an integer which fits into a tagged

[3] This can of course be extended to other arguments.

word of the internal representation) and its second argument is a free variable. This information can be written with the assertion language as:

```
:- entry fact(X, Y) : ( t_int(X), t_var(Y) ).
```

which reflects the call types / modes. The propagation of that information through the canonized predicate gives the annotated program shown in Figure 7.

```
fact(A, B) :-
    true(t_int(A)),
    0 = A,
    true(t_var(B)),
    1 = B.
```

```
fact(A, B) :-
    true(t_int(A)),
    A > 0,
    true(t_int(A)), true(t_var(C)),
    builtin__sub1_1(A, C),
    true(t_any(C)), true(t_var(D)),
    fact(C, D),
    true(t_int(A)), true(t_number(D)),
    true(t_var(B)),
    builtin__times_2(A, D, B).
```

Fig. 7. Annotated factorial (using type information)

The WAM code generated for this example is shown in the rightmost column of Table 4. Underlined instructions were made more specific due to the initial information — note, however, that the representation is homogeneous with respect to the "no information" case.

The impact of type information in the generation of low-level code can be seen in Figure 5. Instructions in the shaded regions are **removed** when type information is available, and the (arithmetical) builtins enclosed in rectangles are replaced by calls to versions specialized to work with small integers and which do not perform type/mode testing.

The optimized `fact/2` program took 2.360 seconds with the same call as in Section 3.3: a 40% speedup with respect to the bytecode version and a 20% speedup over the compilation without type information.

5 Performance Measurements

We have evaluated the performance behavior of the executables generated with our compiler with respect to the emulated bytecode on a set of selected benchmarks. The benchmarks are not real-life programs, and some of them have been executed up to 10.000 times in order to obtain reasonable execution times. All the measurements have been made in a Pentium 4 @ 1.7GHz with a 256KB cache and 256MB of RAM, running Linux with a 2.4 kernel and using gcc 3.0.4 as C compiler.

The summary of the results appears in Table 5; the second, third, and fourth columns correspond, respectively, to the execution times of programs compiled to bytecode, to C, and to C optimized using information on the program. The

Program	Bytecode	C Code	Opt. C	Bytecode/C	Bytecode/Opt. C
`queens(11)`	780	550	260	1.41	3.00
`crypt`	1770	1260	920	1.40	1.92
`tak`	1120	1050	640	1.06	1.75
`qsort`	610	450	370	1.35	1.65
`primes`	1240	1110	820	1.11	1.51
`knights`	720	660	600	1.09	1.20
`poly`	510	520	440	0.98	1.15
`exp`	561	530	540	1.06	1.03
`fib`	350	420	380	0.83	0.92
Average	851	727	552	1.14	1.57

Table 5. Bytecode emulation vs. unoptimized and optimized compilation to C

next two columns show the speedup of programs compiled to C and to optimized C with respect to the emulated bytecode version.

The performance gain in the *naïve* translation to C is not impressive, and there are some programs which even show some slowdown. We have traced this to be due to several factors:

– The simple compilation scheme generates C code as clean and portable as possible, avoiding tricks which would speed the programs up. The profile execution is also very near to what the emulator would make.
– The C execution loop (Figure 2) is slightly more costly (for a few assembler instructions) than the fetch/switch loop of the emulator. We have traced this to be the cause of the slowdown of the `fib` benchmark. We want to improve this point in a future.
– The increment in size of the program (Table 6) may also cause more cache misses. We still have to investigate this point in more detail.

As expected, the performance obtained by using compile-time information is much better. The best speedups are obtained in benchmarks using arithmetic builtins (the only ones which have optimized versions), in which several groundness and type checks can be removed from the C code. This is, for example, the case of `queens`, in which it is known that all the numbers involved are small integers (i.e., no need for unbound length number arithmetic is needed). Besides avoiding checks, the functions which implement the arithmetic operations for small integers are simple enough as to be inlined by the C compiler. This is an example of an added benefit which comes for free from compiling to an intermediate language (C, in this case) and using tools designed for it.

Table 6 compares object size of the bytecode and of the schemes of compilation to C. As mentioned in Section 2, due to the different granularity of instructions, larger object files and executables are expected when compiling to C. The ratio is, however, not excessive: the worst case yields a tenfold increase with respect to the bytecode, the average case being below five times the bytecode size; some cases do not reach a threefold increase. In general, the ratio improves when optimizing information is used (rightmost column), since several

Program	Bytecode	C Code	Opt. C	C/Bytecode	Opt. C/Bytecode
`queens(11)`	7157	22432	19776	3.13	2.76
`crypt`	10632	69860	71588	6.57	6.73
`primes`	6398	23368	18000	3.65	2.81
`tak`	5434	15732	16120	2.89	2.96
`poly`	13531	81444	69660	6.01	5.14
`qsort`	6972	71788	58824	10.2	8.43
`exp`	6453	20536	20808	3.18	3.22
`fib`	5323	11852	11868	2.22	2.22
`knights`	7801	28496	28388	3.65	3.63
Average	7744	38389	35003	4.61	4.21

Table 6. Compared size of object files (bytecode vs. C)

tests are removed from the program. In some cases the size of the C code grows, however, when using more compile time information. The reason is that in the optimized version the *bind* instruction and arithmetic operations are inlined by the compiler generating slightly larger code.

Some of the optimizations used in the compilation to C do not give comparable results when applied directly to a bytecode emulator. For example, a version of the bytecode emulator hand-coded to work only with small integers (which can be boxed into a tagged word) gave a lower performance than that obtained doing the same with compilation to C. That means that when the overheads of calling to builtins is reduced, as is the case of the compilation to C, some minor optimizations for emulated systems acquire greater importance.

6 Conclusions and Future Work

We have reported on the scheme and performance of a Prolog-to-C compiler which uses type analysis information to improve final code by removing type and mode checks and by making calls to specialized versions of some builtins. The compiler is still on a preliminary version, but it shows promising results.

The compilation uses internally a simplified and more homogeneous representation for WAM code, which is then translated to a lower-level intermediate code. This step uses the type information available at compile time. This code is finally translated into C by the compiler back-end; using the intermediate code, as is done in other similar compilers, eases the final translation step, and would allow developing more easily back-ends to other target languages.

We have found using the same information to optimize a WAM bytecode emulator to be more difficult and to give less speedup, due to the greater granularity of the bytecode instructions (which aims at reducing the cost of fetching them). The same result has been reported elsewhere [Van94], although some recent work tries to improve WAM code by means of local analysis [FD02,FD99].

We expect to be able to use more information (e.g., determinacy information) to improve also clause selection, as well as to generate a better indexing scheme at the C level by using hashing on constants, instead of the linear search performed now. Also, we want to study which other optimizations can be added to the generation of C code without breaking its portability, and how the intermediate representation can be used to generate code for other back-ends (for example, GCC RTL, CIL, Java bytecode, etc.).

We want also to connect the analysis results given by CiaoPP with the compiler itself, so that the user does not have to enter type information but in the cases where automatic analysis fails to deduce the more concrete types.

References

[AK91] Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

[CDRA00] V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. *The Yap Prolog User's Manual*, 2000. Available from `http://www.ncc.up.pt/~vsc/Yap`.

[CH00] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[Col93] A. Colmerauer. The Birth of Prolog. In *Second History of Programming Languages Conference*, ACM SIGPLAN Notices, pages 37–52, March 1993.

[DC01] D. Diaz and P. Codognet. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming*, 2001(6), October 2001.

[FD99] M. Ferreira and L. Damas. Multiple Specialization of WAM Code. In *Practical Aspects of Declarative Languages*, number 1551 in LNCS. Springer, January 1999.

[FD02] Michel Ferreira and Luís Damas. Wam local analysis. In Bart Demoen, editor, *Proceedings of CICLOPS 2002*, pages 13–25, Copenhagen, Denmark, June 2002. Department of Computer Science, Katholieke Universiteit Leuven.

[HBC+99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[HBPLG99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[KB95] Andreas Krall and Thomas Berger. The $\mathrm{VAM}_{\mathrm{AI}}$ - an abstract machine for incremental global dataflow analysis of Prolog. In Maria Garcia de la Banda, Gerda Janssens, and Peter Stuckey, editors, *ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages*, pages 80–91, Tokyo, 1995. Science University of Tokyo.

[Mar93] André Mariën. *Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine*. PhD thesis, Katholieke Universiteit Leuven, September 1993.

[PBH00] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[Per87] F. Pereira. *C-Prolog User's Manual, Version 1.5.* University of Edinburgh, 1987.

[PGH97] G. Puebla, J. Gallagher, and M. Hermenegildo. Towards Integrating Partial Evaluation in a Specialization Framework based on Generic Abstract Interpretation. In M. Leuschel, editor, *Proceedings of the ILPS'97 Workshop on Specialization of Declarative Programs*, October 1997. Post ILPS'97 Workshop.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.

[Swe99] Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog 3.8 User's Manual*, 3.8 edition, October 1999. Available from `http://www.sics.se/sicstus/`.

[Tay91] A. Taylor. *High-Performance Prolog Implementation.* PhD thesis, Basser Department of Computer Science, Unversity of Sidney, June 1991.

[Van94] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.

[VD92] P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[VDW87] P. Van Roy, B. Demoen, and Y. D. Willems. Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection, and Determinism. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.

[VR90] P.L. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Univ. of California Berkeley, 1990. Report No. UCB/CSD 90/600.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[Win89] W. Winsborough. Path-dependent reachability analysis for multiple specialization. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.