# ASAP

## IST-2001-38059

**Advanced Analysis and Specialization for Pervasive Systems**

# Efficient Off-line Specialization

| | |
|---|---|
| Deliverable number: | D5 |
| Workpackage: | Basic Specialization Techniques (WP3) |
| Preparation date: | 1 November 2003 |
| Due date: | 1 May 2003 |
| Classification: | Public |
| Lead participant: | Univ. of Southampton |
| Partners contributed: | Univ. of Southampton, Roskilde Univ |

**Project funded by the European Community under the "Information Society Technologies" (IST) Programme (1998–2002).**

**Short description:**

In the first part of this deliverable we give a gentle introduction to offline partial evaluation in general and our LOGEN system in particular. We then show how one can specialize interpreters using offline specialization, starting from simple interpreters on to more complicated ones. Experimental results are also presented, highlighting that the LOGEN system can be a good basis for generating compilers for high-level languages. Being able to deal with interpreters will be of major importance later in the project, especially for workpackage 5. This part of the deliverable has been accepted for publication as a book chapter (Logic-Based Program Synthesis and Transformation, LNCS, Springer-Verlag).

In the second part we describe how we have adapted the LOGEN partial evaluation system in particular in order to specialize modular programs (and specializing them in a modular fashion, i.e., allowing one to separately specialize different modules).Once the implementation of the modular specialisation is finished, we will produce a full paper and submit it to a conference.

The third part demonstrates that the $cogen$ approach is also applicable to the specialization of constraint logic programs and leads to effective specializers. We present the basic specialization technique for CLP(R) and CLP(Q) programs and show experimental results using the LOGEN system. This part has been presented at the PSI'03 conference and the paper will appear in the LNCS post-proceedings.

The fourth part shows how to derive a self-applicable partial evaluator from our LOGEN compiler generator system. Apart from academic curiosity, this allows one to easily generate more or less optimized specialized specializers, just by tuning the annotations. One can also easily generate debugging versions of the specialized specializers. It can also be used to obtain a binding-time analysis, applying the CiaoPP system on purposely generated specializers (generated for analysis purposes and not intended to be run). This part of the deliverable is based on a paper accepted for the FLOPS'04 symposium.

In the fifth part of the deliverable, we present the fully automatic BTA that will be part of our integrated tool, and which makes use of the various technologies developed by the various partners.

In the final part we present a polyvariant binding-time analysis for Mercury which is based on constraint solving, and can deal with higher-order features. This part has been accepted for publication as a book chapter (Logic-Based Program Synthesis and Transformation, LNCS, Springer-Verlag).

# Contents

# Part I

# Introduction and Specialisation of Interpreters

We present the latest version of the LOGEN partial evaluation system for logic programs. In particular we present new binding-types, and show how they can be used to effectively specialise a wide variety of interpreters. We show how to achieve Jones-optimality in a systematic way for several interpreters. Finally, we present and specialise a non-trivial interpreter for a small functional programming language. Experimental results are also presented, highlighting that the LOGEN system can be a good basis for generating compilers for high-level languages.

## 1 Introduction

Partial evaluation [59] is a source-to-source program transformation technique which specialises programs by fixing part of the input of some source program $P$ and then pre-computing those parts of $P$ that only depend on the known part of the input. The so-obtained transformed programs are less general than the original but can be much more efficient. The part of the input that is fixed is referred to as the *static* input, while the remainder of the input is called the *dynamic* input.

Partial evaluation has been especially useful when applied to interpreters. In that setting the static input is typically the object program being interpreted, while the actual call to the object program is dynamic. Partial evaluation can then produce a more efficient, specialised version of

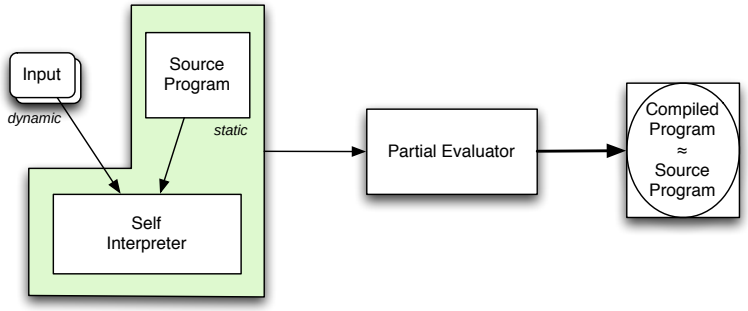



Figure 1: Jones Optimality

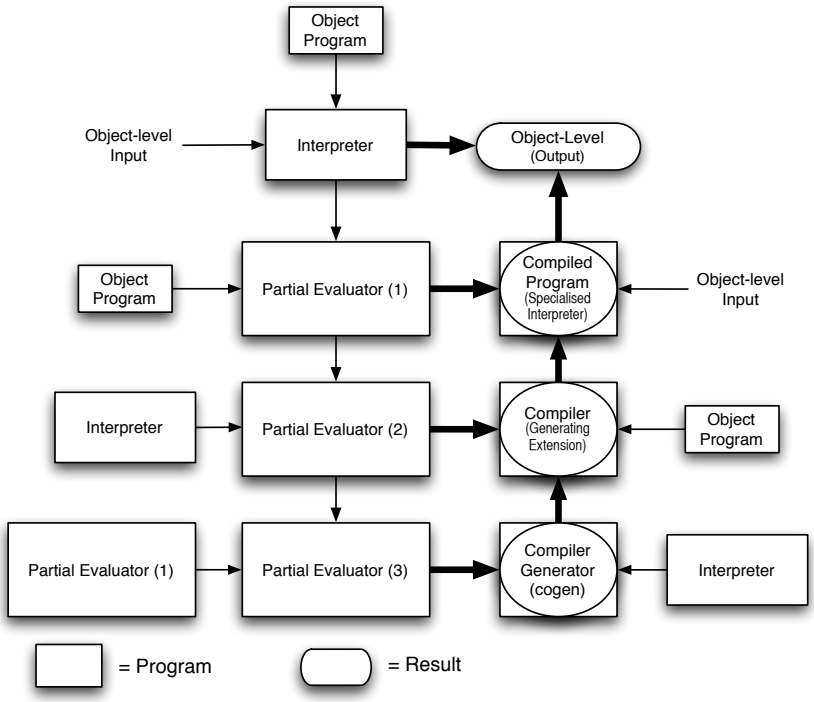



Figure 2: Illustrating the three Futamura projections

the interpreter, which is sometimes akin to a compiled version of the object program.

The ultimate goal in that setting is to achieve so-called *Jones optimality* [57, 59, 85], i.e., fully getting rid of a layer of interpretation (called the "optimality criterion" in [59]). More precisely, if we have a self-interpreter $I$ for a language $L$, i.e., an interpreter for $L$ written in that same language $L$, and then specialise $I$ for a particular object program $O$ we would like to obtain a specialised interpreter which is equivalent to $O$ (or better of course). This is illustrated in Figure 1.

In this work we study systematically how to specialise a wide variety of interpreters written in Prolog using so-called offline partial evaluation. We will illustrate this using the partial evaluation system LOGEN starting from very simple interpreters progressing towards more complicated interpreters. We will also show how we can actually achieve the goal of Jones optimality for a logic programming self-interpreter, as well as for a debugger derived from it; i.e., when specialising the debugger for an object program $O$ with none of its predicates being spyed on we will always get a specialised debugger equivalent to $O$. We believe this to be the first result of its kind in a logic programming setting. In fact, how to effectively specialise interpreters has been a matter of ongoing research for many years, and has been of big interest in the logic programming

community, see e.g., [105, 117, 114, 13, 21, 67, 123, 72] to mention just a few. However, despite these efforts, achieving Jones optimality in a systematic way has remained mainly a dream. To our knowledge, Jones optimiality has been achieved only for a simple Vanilla self-interpreter in [123], but the technique does not scale up to more involved interpreters. All of these works have mainly tried to tackle the problem using fully automatic online partial evaluation techniques, while here we are using the offline approach. Basically, an *online* specialiser takes all of its control decisions during the specialisation process itself, while an *offline* specialiser is guided by a preliminary *binding-time analysis*, which in our case will be (partially) done by hand. The basic reason we opt for the off-line approach is that it allows to steer the specialisation process far better than on-line techniques. This steering is of particular importance in the current setting, since all of the previous research using automatic on-line techniques has shown that specialising interpreters (in general and especially Jones optimality) is hard to achieve.

# 2 Offline Partial Evaluation and the Cogen Approach

## 2.1 The Futamura projections

A partial evaluation or deduction system is called *self-applicable* if it is able to effectively[1] specialise itself. The practical interests of such a capability are manifold. The most well-known lie with the so called second and third *Futamura projections* [30]. The general mechanism of the Futamura projections is depicted in Figure 2. The first Futamura projection consists of specialising an *interpreter* for a particular *object program*, thereby producing a specialised version of the interpreter which can be seen as a *compiled* version of the object program. If the partial evaluator is self-applicable then one can specialise the partial evaluator for performing the first Futamura projection, thereby obtaining a *compiler* for the interpreter under consideration. This process is called the second Futamura projection. The third Futamura projection now consists of specialising the partial evaluator to perform the second Futamura projection. By this process we obtain a *compiler generator* (cogen for short).

## 2.2 Offline Specialisation and the Cogen Approach

Guided by these Futamura projections a lot of effort, especially in the functional partial evaluation community, has been put into making systems self-applicable. First successful self-application was reported in [61], and later refined in [62] (see also [59]). The main idea which

[1] This implies some efficiency considerations, e.g. the system has to terminate within reasonable time constrains, using an appropriate amount of memory.

made this self-application possible was to separate the specialisation process into two phases, as depicted in Figure 3:

- first a *binding-time analysis* ($BTA$ for short) is performed which, given a program and an approximation of the input available for specialisation, approximates all values within the program and generates annotations that steer (or control) the specialisation process.
- a (simplified) *specialisation phase*, which is guided by the result of the $BTA$.

Such an approach is *off-line* because most, control decisions are taken beforehand. The interest for self-application lies with the fact that only the second, simplified phase has to be self-applied. On a more technical level, such an approach also avoids the generation of overly general compilers and compiler generators. We refer to [61, 62, 59] for further details. In the context of logic programming languages the off-line approach was used in [94] and to some extent also in [43].

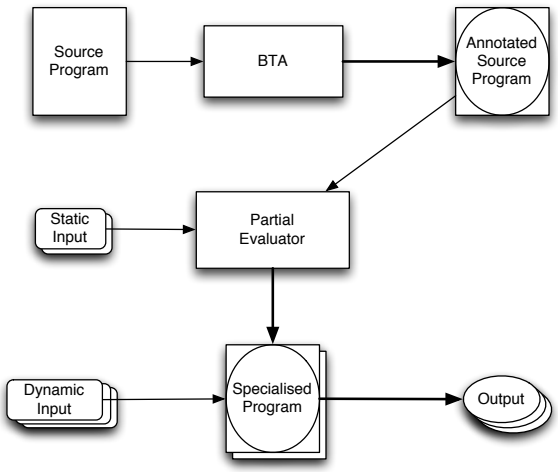



Figure 3: Offline Partial Evaluation

However, the actual creation of the $cogen$ according to the third Futamura projection is not of much interest to users since $cogen$ can be generated once and for all when a specialiser is given. Therefore, from a user's point of view, whether a $cogen$ is produced by self-application or not is of little importance; what is important is that it exists and that it is efficient and produces efficient, non-trivial specialised specialisers. This is the background behind the approach to program specialisation called the *cogen approach* [50, 52, 9, 3, 40, 119] (as opposed to the more traditional $mix$ approach): instead of trying to write a partial evaluation system $mix$ which is neither too inefficient nor too difficult to self-apply one simply writes a compiler generator directly.

## 2.3 Overview of LOGEN

The application of the cogen approach in a logic programming setting was leading to the LOGEN system [63, 78], which we describe in more detail in the next section.

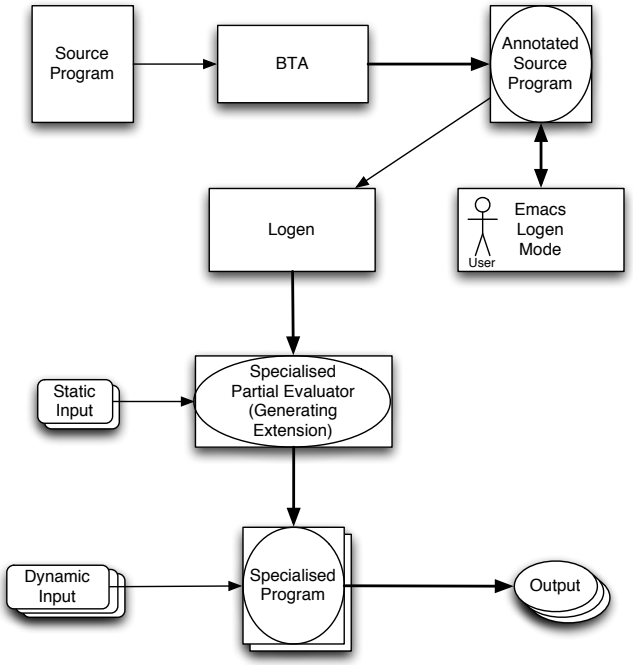



Figure 4: Illustrating the LOGEN system and the *cogen* approach

Figure 4 highlights the way the LOGEN system works. Typically, a user would proceed as follows:

– First the source program is annotated using the BTA, which produces an annotated source program. This annotated source program can be further edited, by using the LOGEN Emacs mode. This also allows an expert to inspect and manually refine the annotations to get better specialisation.

  The picture does not show that LOGEN now also contains a term expansion package (for SICStus and Ciao Prolog) that strips the annotations when loading the annotated source program, allowing the annotated source program to be run directly. Together with the Emacs mode, one can thus continue to develop, maintain and debug the source program together with its annotation (and one can forget the original un-annotated source program).
– Second, LOGEN is run on the annotated source program and produces a specialised specialiser, called a *generating extension*.

– This generating extension can now be used to specialise the source program for some static input. Note that the same generating extension can be run many times for different static inputs (i.e., there is no need to re-run LOGEN on the annotated source program unless the annotated source program itself changes).
– When the remainder of the input is known, the specialised program can now be run and will produce the same output as the original source program. Note again, that the same specialised program can be run for different dynamic inputs; one only has to re-generate the specialised program if the static input changes (or the original program itself changes).

# 3 Offline Partial Deduction of Logic Programs

We now try to formalise the process of offline partial evaluation of logic programs and give a better understanding on how LOGEN specialises its source programs.

Throughout this part of the delivarable, we suppose familiarity with basic notions in logic programming. We follow the notational conventions of [83]. In particular, in programs, we denote variables through strings starting with an upper-case symbol, while the notations of constants, functions and predicates begin with a lower-case character.

## 3.1 Partial Deduction

The term "partial deduction" has been introduced in [65] to replace the term partial evaluation in the context of pure logic programs (no side effects, no cuts). Though in some parts we briefly touch upon the consequences of impure language constructs, we adhere to this terminology because the word "deduction" places emphasis on the purely logical nature of most of the source programs. Before presenting partial deduction, we first present some aspects of the logic programming execution model.

Formally, executing a logic program $P$ for an atom $A$ consists of building a so-called *SLD-tree* for $P \cup \{\leftarrow A\}$ and then extracting the *computed answer substitutions* from every non-failing branch of that tree. Take for example the well-known append program:

```
append([],L,L).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

For example, the SLD-tree for `append([a,b],[c],R)` is presented on the left in Figure 5. The underlined atoms are called selected atoms. Here there is only one branch, and its computed answer is `R = [a,b,c]`.

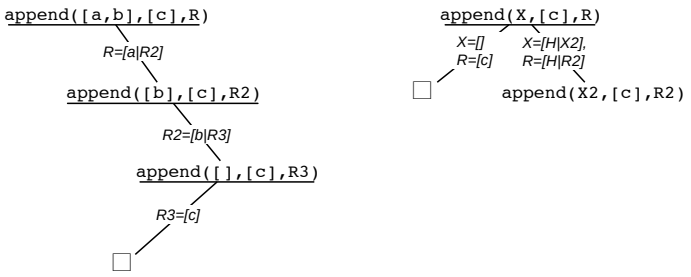



Figure 5: Complete and Incomplete SLD-trees for the `append` program

Partial evaluation builds upon this approach with two major differences:

- it is possible to *not* select a given atom, leading to so-called *incomplete* SLD-trees where the leaves are different from the empty goal. This is because the lack of the full input may cause the SLD-tree to have extra branches, in particular infinite ones. For example, in Figure 5 the rightmost tree is an incomplete SLD-tree for `append(X,[c],R)`, whose full SLD-tree would be infinite.

  The partial evaluator should not only avoid constructing infinite branches, but also other branches causing inefficiencies in the specialised program.

  Building such a tree is called *unfolding*. An *unfolding rule* tells us which atom to select at which point. Every branch of an incomplete tree now does not produce a computed answer, it rather produces a conditional answer which can be expressed as a program clause by taking the resultant of that branch defined below.

- because of the point above, we may have to build a series of SLD-trees, to ensure that every non-selected atom is covered by some root of some tree. The fact that every leaf is an instance of a root is called *closedness* (or sometimes also *coveredness*).

  In Figure 5 the leaf atom `append(X2,[c],R2)` is already an instance of its root atom, and so closedness is already ensured and there is for this example no need to build more trees.

**Definition 3.1** Let $P$ be a program, $G = \leftarrow Q$ a goal, $D$ a finite SLDNF-derivation of $P \cup \{G\}$ ending in $\leftarrow B$, and $\theta$ the composition of the *mgu*s in the derivation steps. Then the formula $Q\theta \leftarrow B$ is called the **resultant** of $D$.

E.g., the resultants of the derivations in the right tree of Figure 5 are:

```
append([],[c],[c]).
append([H|X2],[c],[H|R2]) :- append(X2,[c],R2).
```

Partial deduction starts from an initial set of atoms $A$ provided by the user that is chosen in such a way that all runtime queries of interest are closed, i.e., an instance of some atom in $A$. As we have seen, constructing a specialised program requires to construct an SLDNF-tree for each atom in $A$. Moreover, one can easily imagine that ensuring closedness may require revision of the set $A$. Hence, when controlling partial deduction, it is natural to separate the control into two components (as already pointed out in [33, 91]):

- The *local control* controls the construction of the finite SLD-tree for each atom in $A$ and thus determines *what* the residual clauses for the atoms in $A$ are.
- The *global control* controls the content of $A$, it decides *which* atoms are ultimately partially deduced (taking care that $A$ remains closed for the initial atoms provided by the user).

More details on exactly how to control partial deduction in general can be found, e.g., in [73]. In offline partial evaluation the local control is hardwired, in the form of annotations added to the source program. The global control is also partially hard-wired, by specifying which arguments to which predicate are dynamic and which ones are static.

## 3.2 An Offline Partial Deduction Algorithm

As already outlined earlier, an offline partial evaluator works on an annotated version of the source program. For offline partial deduction of logic programs there are usually two kinds of annotations:

– filter declarations, which indicate which arguments to which predicates are static and which ones dynamic. This influences the global control only.
– clause annotations, which mark every call in the body indicating how that call should be treated during unfolding. This thus influences the local control only. There is of course an interplay between these two annotations, and we return to this below.

For example, one could annotate the `append` example above by saying that the second argument of `append` is static, while the others are dynamic and we could mark the recursive call `append(X,Y,Z)` as not unfoldable. Given such annotations and a specialisation query `append(X,[c],Z)`, offline partial deduction would unfold exactly as depicted in the right tree of Figure 5 and produce the resultants above.

Based on such annotations, offline partial evaluation proceeds as follows:

**Algorithm 3.2 (offline partial deduction)**
**Input:** A program $P$ and an atom $A$

$M = \{A\}$
**repeat**
select an unmarked atom $A$ in $M$ and mark it
unfold $A$ by following the annotations in the annotated source program
**if** a selected atom $S$ is marked as `memo` **then**
generalise $S$ into $S'$ by replacing all arguments marked as dynamic (in the filter declarations) with a fresh variable
**if** no variant of $S'$ is in $M$ **then** add it to $M$
pretty print the specialised clauses of $A$
**until** all atoms in $M$ are marked

In practice, renaming transformations are also involved: Every atom in $M$ is assigned a new predicate name, whose arity is the number of arguments marked as dynamic (static arguments do not need to be passed around; they have already been built into the specialised code). For example, the resultants of the derivations in the right tree of Figure 5 would get transformed into the following, where the second static argument has been removed:

```
append__0([],[c]).
append__0([H|X2],[H|R2]) :- append__0(X2,R2).
```

To give a better picture, we present a Prolog version of the above algorithm. The code is runnable (using an implementation of `gensym`, see [115], to generate new predicate names). The full treatment in LOGEN is of course much more complicated, but this should give a good idea of how LOGEN specialises programs.

An atom $A$ is specialised by calling `memo(`$A$`,Res)` in the code below. The `memo/2` and `memo_table/2` predicates return in their second argument the call to the new specialised predicate where the static arguments are removed and the dynamic ones generalised. This generalisation and filtering is performed by the `generalise_and_filter/3` predicate that returns in its second argument the generalised original call (to be unfolded) with fresh variables and in its third argument the corresponding call to the specialised predicate. It uses the annotations as defined by the `filter/2` predicate to perform its task. The call `memo_table(X,ResX)` within the definition of `memo/2` simply binds `ResX` to the residual version of the call `X`. Note that `ResX` is different from `FX`, which is the residual version of the generalised call `GenX` which has fresh variables. For example, given the filter declaration for `app` below and for `X = app(X,[],X)`

we would get `GenX = app(Y,[],Z)`, and something like `FX = app__0(Y,Z)` and `ResX = app__0(X,X)`.

The predicate `unfold/2` computes the bodies of the specialised predicates. A call annotated as *memo* is replaced by a call to the specialised version. It is created, if it does not exist, by the call to `memo/2`. A call annotated as *unfolded* is further unfolded. To be able to deal with built-ins, we also add two more annotations: a call annotated as *call* is completely evaluated; finally, a call annotated as *rescall* is added to the residual code without modification (for built-ins that cannot be evaluated). These two annotations can also be useful for user-predicates (a user predicate marked as *call* is completely unfolded without further examination of the annotations, while the *rescall* annotation can be useful for predicates defined elsewhere or whose code is not annotated). All clauses defining the new predicate are collected using `findall/3` and pretty printed.

```
:- dynamic memo_table/2.
memo(X,ResX) :- (memo_table(X,ResX)
      -> true /* nothing to be done: already specialised */
      ; (generalise_and_filter(X,GenX,FX),
         assert(memo_table(GenX,FX)),
         findall((FX:-B),unfold(GenX,B),XClauses),
         pretty_print_clauses(XClauses),nl,
         memo_table(X,ResX) ) ).

unfold(X,Code) :- rule(X,B), body(B,Code).
body((A,B),(CA,CB)) :- body(A,CA), body(B,CB).
body(memo(X),ResX) :- memo(X,ResX).
body(unfold(X),ResCode) :- unfold(X,ResCode).
body(call(C),true) :- call(C).
body(rescall(C),C).

generalise_and_filter(Call,GCall,FCall) :- filter(Call,ArgTypes),
         Call =.. [P|Args],
         gen_filter(ArgTypes,Args,GenArgs,FiltArgs),
         GCall =.. [P|GenArgs],
         gensym(P,NewP), FCall =.. [NewP|FiltArgs].
gen_filter([],[],[],[]).
gen_filter([static|AT],[Arg|ArgT],[Arg|GT],FT) :-
       gen_filter(AT,ArgT,GT,FT).
gen_filter([dynamic|AT],[_|ArgT],[GenArg|GT],[GenArg|FT]) :-
```

```
        gen_filter(AT,ArgT,GT,FT).


/* the annotated source program: */
/* filter indicates how to generalise and filter */
filter(app(_,_,_),[dynamic,static,dynamic]).
/* rule annotates the source and indicates how to unfold */
rule(app([],L,L),call(true)).
rule(app([H|X],Y,[H|Z]),memo(app(X,Y,Z))).
```

Call: `memo(app(X,[b],Y))` gives:

```
app__1([],[b]):-true
app__1([_12855|_12856],[_12855|_12854]) :- app__1(_12856,_12854).
```

## 3.3 Local and global termination

Without proper annotations of the source program, the above partial evaluator may fail to terminate. There are essentially two reasons for nontermination.

- **local termination** The unfolding predicate `unfold/2` may fail to terminate or provide infinitely many answers.
- **global termination** Even if all calls to `unfold/2` terminate, we may still run into problems because the partial evaluator may try to build infinitely many specialised versions of some predicate for infinitely many different static values.[2]

To overcome the first problem, we may have to mark certain calls as `memo` rather than `unfold`. In the worst case, every call is marked as `memo`, which always ensures local termination (but means that little or no specialisation is performed).

To overcome global termination problems, we have to play with the filter declarations and mark more arguments as `dynamic` rather than `static`.

Another possible problem appears when built-ins lack enough input to behave as they do at run-time (either by triggering an error or by giving a different result). When this appears, we have to mark the offending call as `rescall` rather than `call`.

[2] One often tries to ensure that a static argument is of so-called *bounded static variation* [59], so that global termination is guaranteed.

# 4 Non-recursive Propositional Logic Interpreter

We first introduce a simple propositional logic interpreter to demonstrate the basic annotations. The interpreter will accept $and$, $or$, $not$, $implies$ and basic variables. The $int(Prog, Env, Result)$ predicate takes two input arguments, the propositional formula and the environment containing the variable mappings and produces the result. The environment is a list of values, $var(i)$ indexes the $i^{th}$ element in the environment.

```
not(true,false).
not(false,true).
and(true,true ,true).          or(true ,_ ,true).
and(false,_ ,false).           or(false,true,true).
and(true,false,false).         or(false,false,false).

int(true,_,true).              int(false,_,false).
int(implies(X,Y),Env, Z) :- int(or(not(X),Y),Env,Z).
int(and(X,Y),Env, Z) :- int(X,Env,R1),int(Y,Env,R2),and(R1,R2,Z).
int(or(X,Y),Env, Z) :- int(X,Env,R1),int(Y,Env,R2),or(R1,R2,Z).
int(not(X),Env, Z) :- int(X,Env,R1),not(R1,Z).
int(var(X),Env, Z) :- lookup(X,Env,Z).
lookup(0,[X|_],X).
lookup(N,[X|T],Y) :- N>0, N1 is N-1, lookup(N1,T,Y).
```

To be able to use LOGEN, one must first define the entry points and annotate the variables for the specialiser.

- **filter** annotates the arguments for residual predicates, using the following annotations
  - **static** the value of the argument is known at specialisation time.
  - **dynamic** the value of the argument is not necessarily known at specialisation time.

  Top level predicates that one intends to specialise must be declared in this way, as well as any subsidiary predicate which cannot be fully unfolded.

The syntax for LOGEN's filter declarations is more user-friendly than in the previous section. For example, for the above program we could declare:

```
:- filter int(static, dynamic, dynamic).
:- filter lookup(dynamic, dynamic, dynamic).
```

In other words, we assume that the propositional formula (the first argument of `int/3`) is known at specialisation time (**static**) but the environment will only be known at runtime (**dynamic**).

Next we must annotate the clauses in the original program to control the specialisation. The following constructs can be used to annotate clauses in a program:

- **unfold** for reducible predicates, they will be unravelled during specialisation,
- **memo** for non-reducible predicates, they will be added to the memoisation table and replaced with a generalised residual predicate,
- **call** the call will be made during specialisation. This is useful for built-in's or for user predicates which should be fully evaluated (without further intervention of the specialiser).
- **rescall** the call will be kept and will appear in the final specialised code. In contrast to the **memo** annotation, no specialised predicate definition is produced for the call. This annotation is especially useful for built-ins, but can also be useful for user predicates (e.g., because the code is not available at specialisation time). The example below will show the difference with the **memo** annotation.

As the propositional formula is known at specialisation time (**static**) all calls to `int/3` can be unfolded. As concerns the variable lookups, let us first be cautious and mark the call to `lookup` as a **rescall**:

```
int(var(X),Env, Z) :- lookup(X,Env,Z).
```
$\underbrace{\phantom{\texttt{lookup(X,Env,Z).}}}_{rescall}$

Let us specialise the interpreter for the logical formula:
$((var(0) \vee (var(1) \wedge \neg var(2))) \vee false) \wedge true$. The output from specialisation is a new version of the program representing the truth table for the formula; as the call to lookup was marked as **rescall** it appears in the specialised program:

```
int(and(or(or(var(0),and(not(var(1)),var(2))),false),true),Env,R) :-
                                    int__0(Env,R).
int__0(A,true) :-
    lookup(0,A,true),lookup(1,A,true),lookup(2,A,C).
int__0(A,false) :-
    lookup(0,A,false),lookup(1,A,true),lookup(2,A,C).
int__0(A,true) :-
    lookup(0,A,true),lookup(1,A,false),lookup(2,A,true).
int__0(A,true) :-
```

```
    lookup(0,A,false),lookup(1,A,false),lookup(2,A,true).
int__0(A,true) :-
    lookup(0,A,true),lookup(1,A,false),lookup(2,A,false).
int__0(A,false) :-
    lookup(0,A,false),lookup(1,A,false),lookup(2,A,false).
```

Observe that no specialised predicate has been produced for `lookup/3`, as we have used the **rescall** annotation. If we mark the call in `int/3` to `lookup/3` as **memo** rather than **rescall** and within the clauses of `lookup/3` we mark the built-in's as **rescall** and the recursive call as **memo**, we obtain the following very similar result:

```
int__0(A,true) :-
    lookup__1(0,A,true),lookup__1(1,A,true),lookup__1(2,A,B).
...
lookup__1(0,[B|C],B).
lookup__1(B,[C|D],E) :- B > 0, F is (B - 1), lookup__1(F,D,E).
```

The main difference is that the specialised program no-longer requires the original code of lookup to run, but apart from that it is almost identical to the previous result. One may notice that in all calls to `lookup/3` the first argument is actually static. One may thus think of changing the filter declaration for `lookup/3` into:

```
:- filter lookup(static, dynamic, dynamic).
```

Unfortunately, if we now run LOGEN we get a specialisation time error. Indeed, in the recursive call `lookup(N1,T,Y)` in second clause of `lookup/3` the variable `N1` will be unbound at specialisation time, and hence LOGEN will complain. The problem is that we have not evaluated the call `N1 is N-1` which binds `N1`. Indeed, what we need to do is to annotate the clause as follows:

`lookup(N,[X|T],Y) :-` $\underbrace{\texttt{N} > \texttt{0}}_{call}$, $\underbrace{\texttt{N1 is N} - \texttt{1}}_{call}$, $\underbrace{\texttt{lookup}(\texttt{N1}, \texttt{T}, \texttt{Y})}_{memo}$.

There is actually no need to **memo** the calls to lookup: given that we know the first argument we can annotate all calls to `lookup/3` as **unfold** and LOGEN will produce the following program:

```
int__0([true,true,B|C],true).
int__0([false,true,B|C],false).
int__0([true,false,true|B],true).
int__0([false,false,true|B],true).
int__0([true,false,false|B],true).
int__0([false,false,false|B],false).
```

It is actually possible to obtain an even better specialisation than this, by providing more information about the structure of the environment. For that we need more sophisticated filter annotations, which we introduce later in Section 7. As an indication and teaser, if we declare `:- filter int(static,list(dynamic), dynamic).` then LOGEN can now produce the following specialised program:

```
int__0(true,true,B,true).
int__0(false,true,B,false).
int__0(true,false,true,true).
int__0(false,false,true,true).
int__0(true,false,false,true).
int__0(false,false,false,false).
```

for the call:

```
int(and(or(or(var(0),and(not(var(1)),var(2))),false),true), [A,B,C],D)}
```

This program is more efficient as the environment list has vanished and no longer needs to be inspected.

# 5 Specialising the Vanilla Self-Interpreter

## 5.1 Background

A classical benchmark for partial evaluation has been the so-called (plain) *vanilla meta-interpreter* (see, e.g., [49, 5]), described by the following piece of Prolog code:

```
solve(empty).
solve(and(A,B)) :- solve(A), solve(B).
solve(X) :-  clause(X,Y), solve(Y).
clause(dapp(X,Y,Z,R),and(app(Y,Z,YZ),app(X,YZ,R))).
clause(app([],L,L),empty).
clause(app([H|X],Y,[H|Z]),app(X,Y,Z)).
```

The `clause/2` facts describe the object program to be interpreted, while `solve/1` is the meta-interpreter executing the object program. In practice, `solve` will often be instrumented so as to provide extra functionality for, e.g., debugging, analysis (e.g., using abstract unifications instead of concrete unification) or transformation. We will actually do so later in this section. However, even without these extensions the vanilla interpreter provides enough challenges for

partial evaluation. Indeed, we would like to specialise the interpreter so as to obtain a residual program equivalent to the object program being interpreted. For example, one would like to specialise our vanilla interpreter for the query `solve(dapp(X,Y,Z,R))` and obtain a specialised interpreter equivalent to:

```
dapp(X,Y,Z,R) :- app(Y,Z,YZ),app(X,YZ,R).
app([],L,L).
app([H|X],Y,[H|Z]) :- app(X,Y,Z).
```

As we have seen in the introduction (cf. Figure 1), achieving such a feat for every object program and query is called "Jones-optimality" [57, 85].

Online partial evaluators such as ECCE [79] or MIXTUS [106] come close to achieving Jones-optimality for many object programs. However, they will not do so for *all* object programs and we refer the reader to [88] (discussing the parsing problem) and the more recent [123] and [72] for more details. [123] presents a particular specialisation technique that can achieve Jones-optimality for the vanilla interpreter, but the technique is very specific to that interpreter and as far as we understand does not scale to extensions of it.

In the rest of this section we show how LOGEN *can* achieve Jones-optimality for the vanilla interpreter, and we show how we can then handle extensions of the basic interpreter.

## 5.2 The nonvar binding time annotation

First, we have to present a new feature of LOGEN which is useful when specialising interpreters. In addition to marking arguments to predicates as static or dynamic, LOGEN also supports the binding-type **nonvar**. This means that this argument is not a free variable and will have at least a top-level function symbol, but it is not necessarily ground. For generalisation, LOGEN will then keep the top-level function symbol but replace all its sub-arguments by fresh variables. For filtering, every sub-argument becomes a new argument of the residual predicate.

A small example will help to illustrate this annotation:

```
:- filter p(nonvar).
p(f(X)) :- p(g(a)).
p(g(X)) :- p(h(X)).
p(h(a)).
p(h(X)) :- p(f(X)).
```

If we mark no call as unfoldable (i.e., every call is marked **memo**), we get the following specialised program for the call p(f(Z)):

```
%%% p(f(A)) :- p__0(A).  p(g(A)) :- p__1(A). p(h(A)) :- p__2(A).
 p__0(B) :-   p__1(a).
 p__1(B) :-   p__2(B).
 p__2(a).
 p__2(B) :-  p__0(B).
```

If we mark everything as **unfold**, except the last call, we obtain:

```
%%% p(f(A)) :- p__0(A).
 p__0(B).
 p__0(B) :- p__0(a).
```

## 5.3 Jones-Optimality for Vanilla

The vanilla interpreter as shown above, is actually a badly written program as it mixes the control structures `and` and `empty` with the actual calls to predicates of the object program. This means that the vanilla interpreter will not behave correctly if the object program contains predicates `and/2` or `empty/0`. This fact also poses problems typing the program. Even more importantly for us, it also prevents one from annotating the program effectively for LOGEN. Indeed, statically there is no way to know whether any of the three recursive calls to `solve/1` has a control structure or a user call as its argument. For LOGEN this means that we can only mark the call `clause(X,Y)` as **unfold**. Indeed, if we mark any of the `solve/1` calls as **unfold** we may get into trouble, i.e., non-termination of the specialisation process. This also means that we cannot even mark the argument to `solve/1` as `nonvar`, as it may actually become a variable. Indeed, take the call `solve(and(p,q))`: it will be generalised into `solve(and(X,Y))` and after unfolding with the second clause we get the calls `solve(X)` and `solve(Y)`. We thus only obtain very little specialisation and we will not achieve Jones-optimality.

Two ways to solve this problem are as follows:

- assume that the control structures are used in a principled, predictable way that will allow us to produce a better annotation.
- rewrite the interpreter so that it is clearly typed, allowing us to produce an effective annotation as well as solving the problem with the name clashes between object program and control structures.

We will pursue these solutions in the remainder of this section. A third possible solution is to use more precise binding types which we introduce in later in Section 7. This will give some improvements, but not full Jones optimality, due to the bad way in which solve is written.

### 5.3.1 Structuring conjunctions

The first solution is to enforce a standard way of writing down conjunctions within `clause/2` facts by requesting that every conjuctions is either `empty` or is an `and` whose left part is an atom and the right hand a conjunction. For the example above, this means that we have to rewrite the `clause/2` facts as follows:

```
clause(dapp(X,Y,Z,R),and(app(Y,Z,YZ),and(app(X,YZ,R),empty))).
clause(app([],L,L),empty).
clause(app([H|X],Y,[H|Z]),and(app(X,Y,Z),empty)).
```

This allows us to predict what to find within the arguments of a conjunction and thus we can now annotate the interpreter more effectively, without risking non-termination:

```
:- filter solve(nonvar).
solve(empty).
solve(and(A,B)) :- solve(A), solve(B).
solve(X) :- clause(X,Y), solve(Y).
```

($solve(A)$: $memo$; $solve(B)$: $unfold$; $clause(X,Y)$: $unfold$; $solve(Y)$: $unfold$)

Given our assumption about the structure of conjunctions, the above annotation will still ensure termination of the generating extension:

- **local termination**:
  - the call to `clause(X,Y)` can be unfolded as before as `clause/2` is definied by facts
  - the calls `solve(B)` and `solve(Y)` can be unfolded as we know that `B` and `Y` are conjunctions and we will only deconstruct the `and/2` and `empty/0` function symbols but stop unfolding (possibly recursive) predicate calls.
- **global termination**: at the point when we memo `solve(A)` the variable `A` will be bound to a predicate call. As we have marked the argument to `solve/1` as `nonvar` generalization will just keep the top-level predicate symbol. As there are only finitely many predicate symbols, global termination is ensured.

Specialising for `solve(dapp(X,Y,Z,R))` now gives a Jones-optimal output.

```
%%% solve(dapp(A,B,C,D)) :- solve__0(A,B,C,D).
%%% solve(app(A,B,C)) :- solve__1(A,B,C).
solve__0(B,C,D,E) :-  solve__1(C,D,F), solve__1(B,F,E).
solve__1([],B,B).
solve__1([B|C],D,[B|E]) :-  solve__1(C,D,E).
```

LOGEN will in general produce a specialised program which is slightly better than the original program in the sense that it will generate code only for those predicates that are reachable in the predicate dependency graph from the initial call. E.g., for `solve(app(X,Y,R))` only two clauses for `app/3` will be produced, not a clause for `dapp/4`.

It is relatively easy to see that Jones optimality will be achieved for any properly encoded object program and any call to the object program. Indeed, any call of the form `solve(p(`$t_1, \ldots, t_n$`)` will be generalised into `solve(p(_,...,_)` keeping information about the predicate being called; unfolding this will only match the clauses of $p$ as the call `clause(X,Y)` is marked **unfold** and all of the parsing structure (`and/2` and `empty/0`) will then be removed by further unfolding, leaving only predicate calls to be memoised. These are then generalised and specialised in the same manner.

### 5.3.2 Rewriting Vanilla

The more principled solution is to rewrite the vanilla interpreter, so that the conjunction encoding and the object level atoms are clearly separated. The attentive reader may have noticed that above we have actually enforced that conjunctions are encoded as lists, with `empty/0` playing the role of `nil/0` and and/2 playing the role of ./2. The following vanilla interpreter makes this explicit and thus properly enforces this encoding. It is also more efficient, as it no longer attempts to find definitions of `empty` and `and` within the `clause` facts.

```
solve([]).
solve([H|T]) :- solve_atom(H), solve(T).
solve_atom(H) :- clause(H,Bdy), solve(Bdy).

clause(dapp(X,Y,Z,R), [app(Y,Z,YZ), app(X,YZ,R)]).
clause(app([],R,R), []).
clause(app([H|X],Y,[H|Z]), [app(X,Y,Z)]).
```

We can now annotate all calls to `solve` as **unfold**, knowing that this will only deconstruct the conjunction represented as a list. However, the call to `solve_atom` cannot be unfolded, as with recursive object programs we may perform infinite unfolding. LOGEN now produces the following specialised program for the query `solve_atom(dapp(X,Y,Z,R))`, having marked the argument to `solve_atom` calls as `nonvar`.[3]

```
solve_atom__0(B,C,D,E) :- solve_atom__1(C,D,F), solve_atom__1(B,F,E).
```

[3]The predicate `solve` does not have to be given a filter declaration as it is only unfolded and never residualised.

```
solve_atom__1([],B,B).
solve_atom__1([B|C],D,[B|E]) :- solve_atom__1(C,D,E).
```

We have again achieved Jones-Optimality, which holds for any object program and any object-level query.

An almost equivalent solution would be to improve the original vanilla interpreter so that atoms are tagged by a special function symbol, e.g., as follows:

```
solve(empty).
solve(and(A,B)) :- solve(A), solve(B).
solve(atom(X)) :-  solve_atom(X).
solve_atom(H) :- clause(H,Bdy), solve(Bdy).
clause(dapp(X,Y,Z,R),and(atom(app(Y,Z,YZ)),atom(app(X,YZ,R)))).
clause(app([],L,L),empty).
clause(app([H|X],Y,[H|Z]),atom(app(X,Y,Z))).
```

We have again clearly separated the control structures from the predicate calls and we can basically get the same result as above (by marking all calls to solve as **unfold** and the call to solve_atom as **memo**).

### 5.3.3 Reflections

So, what are the essential ingredients that allowed us to achieve Jones optimality where others have failed?

- First, the offline approach allows us to precisely steer the specialisation process in a predictable manner: we know exactly how the interpreter will be specialised independently of the complexity of the object program. A problem with online techniques is that they may work well for some object programs, but then be "fooled" by other (more or less contrived) object programs; see [123, 72]. (On the other hand, online techniques can be capabable for removing several layers of self-interpretation in one go. An offline approach in general and our approach in particular will typically only be able to remove one layer at a time.)
- Second, it was also important to have refined enough annotations at our disposal. Without the **nonvar** annotation we would not have been able to specialise the original vanilla self-interpreter: we cannot mark the argument to `solve` as static and marking it as dynamic means that no specialisation will occur. Hence, considerable rewriting of the interpreter would have been required if we just had **static** and **dynamic** at our disposal.[4]

[4]We leave this as an exercise for the interested reader. See also Section 7.1 later in this part.

# 6 Jones-Optimality for a Debugger

Let us now try to extend the above interpreter, to do something more useful. The code below implements a tracing version of `solve` which takes two extra arguments: a counter for the current indentation level and a list of predicates to trace.

```
dsolve([],_,_).
dsolve([H|T],Level,ToTrace) :-
      (debug(H,ToTrace)
       -> (indent(Level),print('Call: '),print(H),nl,
           dsolve_atom(H,s(Level),ToTrace),
           indent(Level),print('Exit: '),print(H),nl)
        ;  dsolve_atom(H,Level,ToTrace)
       ),
      dsolve(T,Level,ToTrace).

debug(Call,ToTrace) :- Call=..[P|Args],
     length(Args,Arity), member(P/Arity,ToTrace).

:- filter indent(dynamic).
indent(0).
indent(s(X)) :- print('>'),indent(X).

:- filter dsolve_atom(nonvar,dynamic,static).
dsolve_atom(H,Level,TT) :-
   clause(H,Bdy), dsolve(Bdy,Level,TT).
```

Basically, the annotation of dsolve and dsolve_atom calls are exactly as before: calls to `dsolve` are unfolded, calls to `dsolve_atom` are not. As the new predicates are concerned, all calls to `indent` are marked **memo**, and all calls to `print` and `nl` are marked **rescall**. Everything else is marked **unfold** or **call**.

For `dsolve_atom(dapp([a,a,a],[b],[c],R),0,[])` we get the following almost optimal code:

```
dsolve_atom__0(B,C,D,E,F) :-
  dsolve_atom__1(C,D,G,F), dsolve_atom__1(B,G,E,F).
dsolve_atom__1([],B,B,C).
dsolve_atom__1([B|C],D,[B|E],F) :- dsolve_atom__1(C,D,E,F).
```

In fact, the extra last argument of both predicates can be easily removed by the FAR redundant argument filtering post-processing of [81] which produces a Jones-optimal result:

```
dsolve_atom__0(A,B,C,D) :-
  dsolve_atom__1(B,C,E),dsolve_atom__1(A,E,D).
dsolve_atom__1([],A,A).
dsolve_atom__1([A|B],C,[A|D]) :-  dsolve_atom__1(B,C,D).
```

Again, is is not too difficult to see that LOGEN together with the FAR post-processor [81] produces a Jones-optimal result for every object program $P$ and call $C$, provided that none of the predicates reachable from $C$ are traced.

For `dsolve_atom(dapp([a,a,a],[b],[c],R),0,[app/3])` we get the following very efficient tracing version of our object program, where the debugging statements have been weaved into the code. This specialised code now runs with minimal overhead, and there is no more runtime checking whether a call should be traced or not:

```
dsolve_atom__0(B,C,D,E,F) :-
  indent__1(F),print('Call: '),print(app(C,D,G)),nl,
  dsolve_atom__2(C,D,G,s(F)),
  indent__1(F),print('Exit: '),print(app(C,D,G)),nl,
  indent__1(F),print('Call: '),print(app(B,G,E)),nl,
  dsolve_atom__2(B,G,E,s(F)),
  indent__1(F),print('Exit: '),print(app(B,G,E)),nl.
indent__1(0).
indent__1(s(B)) :-  print('>'),indent__1(B).
dsolve_atom__2([],B,B,C).
dsolve_atom__2([B|C],D,[B|E],F) :-
  indent__1(F),print('Call: '),print(app(C,D,E)),nl,
  dsolve_atom__2(C,D,E,s(F)),
  indent__1(F),print('Exit: '),print(app(C,D,E)),nl.
```

Running the specialised program for `dsolve_atom__0([a,b,c],[],[d],R,0)`, corresponding to the call `dsolve_atom(dapp([a,b,c],[],[d],R),0,[app/3])` to the original program, prints the following trace:

```
  | ?- dsolve_atom__0([a,b,c],[],[d],R,0).
  Call: app([],[d],_837)
  Exit: app([],[d],[d])
```

```
Call: app([a,b,c],[d],_525)
>Call: app([b,c],[d],_1341)
>>Call: app([c],[d],_1601)
>>>Call: app([],[d],_1891)
>>>Exit: app([],[d],[d])
>>Exit: app([c],[d],[c,d])
>Exit: app([b,c],[d],[b,c,d])
Exit: app([a,b,c],[d],[a,b,c,d])
R = [a,b,c,d] ?
yes
```

#### 6.0.4 Some experimental results

We now present some experimental results for specialising the `solve` and `dsolve` interpeters. The results are summarised in Table 1. The results were obtained on a Powerbook G4 running at 1 Ghz with 1Gb RAM and using SICStus Prolog 3.10.1.

The `partition4` object program calls append to partition a list into 4 identical sublists, and has been run for a list of 1552 elements. The `fibonacci` object program computes the Fibonacci numbers in the naive way using peano arithmetic. This program was benchmarked for computing the 24th Fibonacci numbers. Exact queries can be found in the DPPD library [71]. The FAR filtering [81] has not been applied to the specialised programs. The time needed to generate and run the generating extensions was negligible (more results, with full times can be found later, for more involved interpreters where this time is more significant).

Table 1: Specialising `solve` and `dsolve` using LOGEN

| object program | `solve` | specialised | speedup | `dsolve` | specialised | speedup |
|---|---|---|---|---|---|---|
| partition4 | 350 ms | 200 ms | 1.75 | 1590 ms | 220 ms | 7.23 |
| fibonacci | 890 ms | 170 ms | 5.24 | 4670 ms | 180 ms | 25.94 |

#### 6.0.5 Adding more functionality

It should be clear how one can extend the above logic program interpreters. A good exercise is to add more logical connectives, such as disjunction and implication, to the debugging interpreter `dsolve` and then see whether one can obtain something similar to the Lloyd-Topor transformations [84] automatically by specialisation (with the added benefit that debugging can still be

performed at the source level).

We will now show how one can handle interpreters for other programming paradigms. In such a setting variables and their values may have to be stored in some environment structure rather than relying on the Prolog variable model. This will raise a new challenge, which we tackle next.

# 7 More Sophisticated Binding-Types

So far we have come by with just three binding types for arguments: static, dynamic, and nonvar. The latter denotes a simple kind of so-called *partially static* data [59]. For more realistic programs, however, it is often essential to be able to deal with more sophisticated partially static data. For example, interpreters often have an environment, and at specialisation time we may know the actual variables store in the environment but not their value. Take the following simple interpreter for arithmetic expressions using addition, constants and variables whose value is stored in an environment:

```
int(cst(C),_E,C).
int(var(V),E,R) :- lookup(V,E,R).
int(+(A,B),E,R) :- int(A,E,Ra), int(B,E,Rb), R is Ra+Rb.

lookup(V,[(V,Val)|_T],Val).
lookup(V,[(_Var,_)|T],Res) :- lookup(V,T,Res).
```

A typical query to the above program would be

```
| ?- int(+(var(a),var(b)),[(a,1),(b,3),(c,5)],Res).
Res = 4 ?
yes
```

Now, if at specialisation time we know the variables of the environment list but not their value, this would be represented by an atom to specialise:

```
int(+(var(a),var(b)),[(a,_),(b,_),(c,_)],R).
```

We cannot declare the environment as static and the best we can do, given the binding types we have seen so far, is to declare the environment as nonvar:

```
:- filter int(static,nonvar,dynamic).
```

Unfortunately, this means that LOGEN will replace `[(a,_),(b,_),(c,_)]` by `[_|_]`, hence leading to suboptimal specialisation. For example, we cannot unfold `lookup` because we now no longer know the length of the environment.

## 7.1 Binding-Time improvements and bifurcation

One way to overcome such limitations is often to rewrite the program to be specialised into a semantically equivalent program which specialises better, i.e., in which more arguments can be classified as static and/or more calls can be unfolded. This process is called *binding-time improvement*, see, e.g., Chapter 12 of [59].

One simple binding-time improvement for this particular problem is to define an auxilary predicate as follows:

```
aux(Expr,A,B,C,Res) :- int(Expr,[(a,A),(b,B),(c,C)],Res).
```

We can now fully unfold all calls to `int` and `lookup` and declare the arguments of `aux` as follows:

```
:- filter aux(static,dynamic,dynamic,dynamic,dynamic).
```

However, this solution is rather ad-hoc and only works because the above interpreter is non-recursive and hence no calls to `int` have to be memoised. Hence, this solution can only work in special circumstances.

A more principled solution, is to apply a binding-time improvement sometimes called *bi-furcation* [23, 96]. This consists of splitting the environment into two parts (the static and the dynamic part) and then rewriting the interpreter accordingly. Here, a solution is to split the environment into two lists: a static one containing the variable names and a dynamic list containing the actual values. We would then rewrite our interpreter as follows:

```
:- filter int(static,static,dynamic,dynamic).
int(cst(C),_E,_E2,C).
int(var(V),E,E2,R) :- lookup(V,E,E2,R).
int(+(A,B),E,E2,R) :- int(A,E,E2,Ra), int(B,E,E2,Rb), R is Ra+Rb.

:- filter lookup(static,static,dynamic,dynamic).
lookup(V,[V|_],[Val|_],Val).
lookup(V,[_|T],[_|ValT],Res) :- lookup(V,T,ValT,Res).
```

We can now fully unfold all calls to `int` and `lookup`. One could also decide not to unfold the calls to `int` or to `lookup(V,E,E2,R)` without much loss of specialisation, and the technique would also work for a recursive interpreter.

There are however several problems with this approach:

– it can be very cumbersome and errorprone to rewrite the program

– for every different annotation we may have to rewrite the program in a different way
– if the dynamic and static data are not as neatly separated as above, it can be non-trivial to find a proper separation
– the final result is not always "optimal". E.g., in the example above the information that the variable list and the value list must be of the same length is no longer explicit, resulting in a suboptimal residual program.
  For example, specialising for `lookup(b,[a,b,c],[1,X,Y],Res)` gives:

```
%%% lookup(b,[a,b,c],[1,X,Y],Res) :- lookup__0([1,X,Y],Res).
%%% lookup(b,[a,b,c],A,B) :- lookup__0(A,B).
lookup__0([B,C|D],C).
```

  This is less efficient than the result we will obtain later below, mainly because the value list has still to be deconstructed and examined at runtime (via the unification with `[B,C|D]`).

Luckily, LOGEN provides a better way of solving this problem by allowing the user to define their own binding-types. For the interpreter above we would like to be able to define a custom binding-type describing a list of pairs whose first element is static and the second dynamic. In the rest of this section we formalise and describe how this can be achieved.

## 7.2 Formal Definition of Binding-Types

In what follows, we introduce the notion of a *binding-type* to characterise partially instantiated specialisation-time values in a more precise way. Like a traditional type in logic programming [4], a binding-type is conceptually defined as a set of terms closed under substitution and represented by a term constructed from *type variables* and *type constructors* in the same way that a data term is constructed from ordinary variables and function symbols. However, to characterise specialisation-time values rather than run-time values, we assume three predefined, atomic types, i.e. *static*, *dynamic* and *nonvar* ($\in \mathcal{C}$).

Formally, a *type* is thus

– either a *type variable*,
– a term of the form `static`, `dynamic`, or `nonvar`,
– a term of the form `term(`$\sigma$`)` where $\sigma = f(\tau_1, \ldots, \tau_n)$ and $f$ is a function symbol of arity $n \geq 0$ and $\tau_i$ are types,
– or a term of the form `type(`$\tau$`)` where $\tau$ consists of a *type constructor* of arity $n \geq 0$ applied to $n$ types.

The use of the `term` and `type` tags allows the set of function symbols and type constructors to overlap and avoids cumbersome renamings. We will introduce some shorthand notations below. Formally, new types can now be defined as follows:

**Definition 7.1** A *type definition* for a type constructor $c$ of arity $n$ is of the form:

`:- type` $c(V_1, \ldots, V_n)$ `---> (`$\tau_1$ `;` $\ldots$ `;` $\tau_k$`).`

with $k \geq 1, n$ and where $V_1, \ldots, V_n$ are distinct type variables, and $\tau_i$ are types which only contain type variables in $\{V_1, \ldots, V_n\}$.
A *type system* $\Gamma$ is a set of type definitions, exactly one for every type constructor $c$ different from `static`, `dynamic`, and `nonvar`. We will refer to the type definition for $c$ in $\Gamma$ by $Def_\Gamma(c)$.

For convenience, LOGEN also accepts the following shorthand notations as types:

- a function symbol $f$ of arity $n \geq 0$ applied to $n$ types, provided that $n = 0 \Rightarrow f \notin \{static, dynamic, nonvar\}$ and $n = 1 \Rightarrow f \notin \{type, list, term\}$. This is then equivalent to the type `term(`$f$`(`$\tau_1, \ldots, \tau_n$`))`.
- or a term of the form `list(`$\tau$`)` where $\tau$ is a type. This is equivalent to `type(list(`$\tau$`))`, where the type constructor `list` is pre-defined as follows:
  `:- type list(T) ---> [ ] ; [T | list(T)].`
  We will refer to the type definition for $c$ in $\Gamma$ by $Def_\Gamma(c)$.

We define *type substitutions* to be finite sets of the form $\{V_1/\tau_1, \ldots, V_k/\tau_k\}$, where every $V_i$ is a type variable and $\tau_i$ a type. Type substitutions can be applied to types (and type definitions) to produce *instances* in exactly the same way as substitutions can be applied to terms. For example, $list(V)\{V/static\} = list(static)$. A type or type definition is called *ground* if it contains no type variables.

In general, a specialisation-time value (or data term) can be characterised by a number of binding-types. This relation is made explicit by a *type judgment*.

**Definition 7.2** We now define *type judgements* relating terms to types in the type system $\Gamma$.

- $t$ : `dynamic` holds for any term $t$
- $t$ : `static` holds for any ground term $t$
- $t$ : `nonvar` holds for any non-variable term $t$
- $t$ : `type`$(c(\tau'_1, \ldots, \tau'_k))$ if there exists a ground instance of the type definition $Def_\Gamma(c)$ which has the form `:- type` $c(\tau'_1, \ldots, \tau'_k)$`--> (` $\ldots$ `;`$\tau$`;` $\ldots$ `)` and where $t : \tau$
- $f(t_1, \ldots, t_n)$ : `term`$(f(\tau_1, \ldots, \tau_n))$ if $t_i : \tau_i$ for $1 \leq i \leq n$.

Note that our definitions guarantee that types are downwards-closed (i.e., $t : \tau \Rightarrow t\theta : \tau$).

A few examples are as follows: $[\,] : static$, $[\,] : struct([\,])$, $[\,] : list(static)$, $[\,] : list(dynamic)$, $s(0) : static$ hence $[s(0)] : list(static)$, $X : dynamic$ and $Y : dynamic$ hence $[X, Y] : list(dynamic)$.

## 7.3 Using binding-types

Basically, the three basic binding types are now used to control generalisation and filtering within the offline partial deduction algorithm of Section 3.2 as follows:

- an argument marked as *dynamic* is replaced by a fresh variable and there will be an argument for it in the residual predicate;
- an argument marked as *static* is not generalised, and there will be no argument for it in the residual predicate;
- an argument marked as *nonvar* the top-level function symbol willl be kept, but all of its arguments replaced by fresh variables. There will be one argument in the residual predicate per argument of the top-level function symbol.
- an argument marked as *term(*$f(\tau_1, \ldots, \tau_n)$*)* will basically be dealt with like the *nonvar* case, except that the top-level function symbol has to be $f$ and every sub-argument of $f$ will be recursively generalised and filtered according to the binding-types $\tau_i$.
- for an argument marked as *type(*$t(\tau_1, \ldots, \tau_n)$*)* the type definition of $t$ will be looked at and the argument will be treated according to the body of the definition. For disjunctions like $\tau_1$ `;` $\tau_2$ the algorithm will first attempt to apply $\tau_1$, and if that is not successful it will apply $\tau_2$.

For example, given the declaration `:- filter p(static,dynamic,nonvar).` the call `p(a,[b],f(c,d))` will be generalised into `p(a,_,f(_,_))` and the residual version of the call will be something like `p__1([b],c,d)`.

Given the declaration `:- filter p(static,dynamic,term(f(static,dynamic))).` the call will be generalised into `p(a,_,f(c,_))` and the residual version will be something like `p__2([b],d)`.

Finally, using `:- filter p(static,list(dynamic),static).` as filter declaration, this call will be generalised into `p(a,[_],f(c,d))` with the residual version being `p__3(b)`.

Let us now try to tackle the original arithmetic `int/3` interpreter using the more refined binding-types. First, we define a new type, describing a list of pairs whose first element is static and whose second element is given by a parameter of the type constructor (so as to show how parameters can be used):

```
:- type bind_list(X) ---> list((static,X)).
```

For the interpreter we can now simply provide the following filter declarations:

```
:- filter int(static,type(bind_list(dynamic)),dynamic).
:- filter lookup(static,type(bind_list(dynamic)),dynamic).
```

While these annotations and types were derived by hand, we believe that it is possible to derive them by adapting the polymorphic binding-time analysis for Mercury presented in a companion paper [122]. For more details see [122].

Let us now use LOGEN to specialise the original `int/3` interpreter for the query `lookup(b,[(a,1),(b,X),(c,Y)],Res)`. This gives the following specialised code:

```
%%% lookup(b,[(a,A),(b,B),(c,C)],D) :- lookup__0(A,B,C,D).
lookup__0(B,C,D,C).
```

This code is much more efficient, as linear time lookup of variable bindings has been replaced by basically constant time lookup in the argument list.

Let us now specialise the interpreter for a full-fledged query:
`int(+(cst(3),+(+(cst(2),cst(5)),+(var(y),+(var(x),var(y))))),`
`[(a,1),(b,2),(x,3),(y,4)],X)`. This produces the following satisfactory result, where the arithmetic expression has been fully compiled into Prolog code.

```
int__0(B,C,D,E,F) :-  G is (2 + 5), H is (D + E),
                      I is (E + H), J is (G + I), F is (3 + J).
```

One can see that the reduction `G is (2+5)` has not been performed by the specialiser. This shows an aspect where an online specialiser could have fared better, as it could have realised that, for this particular instruction, the right hand side of the `is/2` was actually known (even though it is in general dynamic). Still, it is possible to instruct LOGEN to try to perform calls using the so-called **semicall** annotation [78]. Another alternative is to binding-time improve the program by inserting an explicit if-statement, changing the 3rd clause of the interpreter as follows:

$$\texttt{int(+(A,B),E,E2,R) :-}\ \underbrace{\texttt{int(A,E,E2,Ra)}}_{unfold},\ \underbrace{\texttt{int(B,E,E2,Ra)}}_{unfold},$$
$$(\ \underbrace{\texttt{ground((Ra,Rb))}}_{call}\ \texttt{->}\ \underbrace{\texttt{R is Ra + Rb}}_{call}\ \texttt{;}\ \underbrace{\texttt{R is Ra + Rb}}_{rescall}\ ).$$

where the if-statement itself is marked static and performed at specialisation time. The resulting specialised interpreter is then:

```
int__0(B,C,D,E,F) :-  G is (D + E), H is (E + G),
                      I is (7 + H),  F is (3 + I).
```

### 7.4 Revisiting Vanilla again

Finally, let us present a third solution for specialisng the Vanilla self-interpreter from Section 5.3. Indeed, we can now use the following more precise binding types on the original interpreter, thus ensuring that relevant information will be kept by the generalisation:

```
:- type vexp ---> (empty ; and(type(vexp),type(vexp))
                         ; type(predcall)).
:- type predcall ---> (app(dynamic,dynamic,dynamic)
                     ;  dapp(dynamic,dynamic,dynamic,dynamic)).
:- filter solve(type(vexp)).
```

This will not give full Jones optimality, due to the bad way in which the original `solve` is written, but it will at least give much better specialisation than was possible using just **static**, **dynamic**, and **nonvar**.

## 8 Lambda Interpreter

Based on the insights of the previous section, we now tackle a more substantial example. We will present an interpreter for a small functional language. The interpreter still leaves much to be desired from a functional programming language perspective, but the main purpose is to show how to specialise a non-trivial interpreter for another programming paradigm. The interpreter will use an environment, very much like the one in the previous section, to store values for variables and function arguments. The full annotated source code is available with the LOGEN distribution at `http://www.ecs.soton.ac.uk/~mal/systems/logen.html`.

To keep things simple, we will not use a parser but simply use Prolog's operator declarations to encode the functional programs. The following shows how to encode the fibonacci function for our interpreter:

```
:- op(150,fx,$). /* to indicate variables */
:- op(150,fx,&). /* to indicate constants */
:- op(150,yfx,'===’). /* to define functions */
:- op(150,yfx,@). /* to do calls to defined functions */
:- op(250,yfx,'->’). /* for sequential composition */

fib ===  lambda(x,if($x = &0,  &1,
      if($x = &1,  &1,
        (fib @ ($x - &1) + fib @ ($x - &2))))).
```

The source code of the interpreter is as follows. As usual in functional programming, one distinguishes between constructors (encoded using `constr/2`) and functions (encoded using `lambda/2`). Functions can be defined statically using the `===` declarations which can then be extracted using the `fun/1` expression. One can use `@` as a shorthand to call such defined functions. One can introduce local variables using the `let/3` expression. The predicate `eval/3` computes the normal form of an expression. The rest of the code should be pretty much self-explanatory. To keep the code simpler, we have not handled renaming of the arguments of lambda expressions (it is not required for the examples we will deal with).

```
eval('&'(C),_Env,constr(C,[])).  /* 0-ary constructor */
eval(constr(C,Args),Env,constr(C,EArgs)) :- l_eval(Args,Env,EArgs).
eval('$'(VKey),Env,Val) :-  /* variable */ lookup(VKey,Env,Val).
eval('+'(X,Y),Env,constr(XY,[])) :- eval(X,Env,constr(VX,[])),
        eval(Y,Env,constr(VY,[])), XY is VX+VY.
eval('-'(X,Y),Env,constr(XY,[])) :- eval(X,Env,constr(VX,[])),
        eval(Y,Env,constr(VY,[])), XY is VX-VY.
eval('*'(X,Y),Env,constr(XY,[])) :- eval(X,Env,constr(VX,[])),
        eval(Y,Env,constr(VY,[])), XY is VX*VY.
eval(let(VKey,VExpr,InExpr),Env,Result) :- eval(VExpr,Env,VVal),
        store(Env,VKey,VVal,InEnv), eval(InExpr,InEnv,Result).
eval(if(Test,Then,Else),Env,Res) :- eval_if(Test,Then,Else,Env,Res).
eval(lambda(X,Expr),_Env,lambda(X,Expr)).
eval(apply(Arg,F),Env,Res) :- eval(F,Env,FVal),
        eval(Arg,Env,ArgVal), eval_apply(ArgVal,FVal,Env,Res).
eval(fun(F),_,FunDef) :-  '==='(F,FunDef).
eval('@'(F,Args),E,R) :- eval(apply(Args,fun(F)),E,R).
eval(print(X),Env,FVal) :- eval(X,Env,FVal),print(FVal),nl.
eval('->'(X,Y),Env,Res) :-  /* seq. composition */
        eval(X,Env,_), eval(Y,Env,Res).

eval_apply(ArgVal,FVal,Env,Res) :- rename(FVal,Env,lambda(X,Expr)),
        store(Env,X,ArgVal,NewEnv), eval(Expr,NewEnv,Res).

rename(Expr,_Env,RenExpr) :- RenExpr=Expr.  /* sufficient for now */

l_eval([],_E,[]).
l_eval([H|T],E,[EH|ET]) :- eval(H,E,EH), l_eval(T,E,ET).
```

```
eval_if(Test,Then,_Else,Env,Res) :- test(Test,Env), !, eval(Then,Env,Res).
eval_if(_Test,_Then,Else,Env,Res)) :- eval(Else,Env,Res).

test('='(X,Y),Env) :- eval(X,Env,VX),eval(Y,Env,VX).

store([],Key,Value,[Key/Value]).
store([Key/_Value2|T],Key,Value,[Key/Value|T]).
store([Key2/Value2|T],Key,Value,[Key2/Value2|BT]) :-
        Key\==Key2,store(T,Key,Value,BT).

lookup(Key,[Key/Value|_T],Value).
lookup(Key,[Key2/_Value2|T],Value) :-
        Key\==Key2,lookup(Key,T,Value).
```

## 8.1 Handling the cut

One may notice that the above program does use a cut in the code for `eval_if`. Previous version of LOGEN did not support the cut, but it turns out that specialising the cut is actually very easy to do: basically all one has to do is to simply mark the cuts using either the `call` or `rescall` annotations we have already encountered. It is of course up to the annotator to ensure that this is sound, i.e., one has to ensure that:

- if a cut is marked `call`, then whenever it is reached and executed at specialisation time the calls to the left of the cut will never fail at runtime.
- if a cut is marked as `rescall` within a predicate $p$, then no calls to $p$ are unfolded. One can relax this condition somewhat, e.g., one may to be able to unfold such a predicate $p$ if all computations are deterministic (like in our functional interpreter) but one has to be very careful when doing that.

These conditions are sufficient to handle the cut in a sound, but still useful manner.

## 8.2 Annotations

To be able to specialise this interpreter we need the power of LOGEN's binding types. The structure of the environment is much like in the previous section, but here we have more information about the structure of values that the interpreter manipulates and stores. Basically, values are encoded using `constr/2`, whose first argument is the symbol of the constructor being encoded and the second argument is a list containing the encoding of the arguments. A lambda expression is also a valid value.

```
:- type value_expression =
      (constr(dynamic,list(type(value_expression))) ;
       lambda(static,static)).
:- type env = list( static / type(value_expression)).
```

We can now annotate the calls of our program. Basically, all built-ins have to be marked **rescall** but all user calls can be marked as **unfold** except for the call:

`eval_apply(ArgVal,FVal,Env,Res)`.

We thus supply the following filter declaration:

```
:- type result = ( type(value_expression) ; dynamic).
:- filter eval_apply(type(result),type(result),type(env),dynamic).
```

Note that we use a union type for `result`, because often (but not always) we will have partial information about the result types. Union types are thus a way to allow LOGEN to make some online decisions: during specialisation it will check whether the first and second argument of `eval_apply` match the `value_expression` type and only if they do not will it treat the arguments as dynamic.

## 8.3 Experiments

When specialising this program for, e.g., calling the `fib` function we get something very similar to the (naive) fibonacci program one would have written in Prolog in the first place:

```
%% eval_apply(constr(A,[]),lambda(x,if($x= &0,&1,if($x= &1,&1,
%%     fib@($x- &1)+fib@($x- &2)))),[x/constr(B,[])],C) :-
%%     eval_apply__2(A,B,C).
eval_apply__2(0,B,constr(1,[])) :-  !.
eval_apply__2(1,B,constr(1,[])) :-  !.
eval_apply__2(B,C,constr(D,[])) :-
  E is (B - 1), eval_apply__2(E,B,constr(F,[])),
  G is (B - 2), eval_apply__2(G,B,constr(H,[])),  D is (F + H).
```

This specialised code runs about 14 times faster than the original, and even running all of LOGEN, the generating extension and then the specialised program is still 7 times faster than running the original program. Full details of this experiment can be found in Table 2.

Furthermore, speedups are likely to get much bigger for more complicated programs, with more functions and more arguments and variables. Indeed, in Table 2 we have also specialised the interpreter for the following slightly bigger functional program `loop_fib` which has extra loop variables, already resulting in a bigger speedup:

```
loop_fib === lambda(cur,let(cur1,$cur + &1, let(cur2, $cur1 + &1,
              let(cur3, $cur2 + &1,  if(($cur = &22),
                  (fib @ ($cur)),
                  (print(constr(fibonacci,[$cur,fib @ ($cur)]))
                    ->  (loop_fib @ ($cur1)))))))).
```

Note that LOGEN has only to be run once for the `eval` interpreter; the same generating extension can then be used for any functional programs. Similarly, the specialised code can then be used for any call to the functional program and the generating extension only has to be run once per functional program that is compiled.[5]

Table 2: Specialising `eval` using LOGEN

| function call | `eval` runtime | cogen time | genex time | specialised runtime | speedup | speedup (incl. gx) | speedup (incl. gx,cogen) |
|---|---|---|---|---|---|---|---|
| fib(24) | 1050 ms | 60 ms | 15ms | 75 ms | 14.0 | 11.7 | 7 |
| loop_fib(0) | 2030 ms | 60 ms | 20ms | 90 ms | 22.6 | 18.5 | 11.9 |

# 9 Discussion and Conclusion

Probably the most closely related work is [58] which treats untype first-order functional languages, and gives a list of recommendations on how to write interpreters that specialise well. Even though [58] does of course not address the specific issues that arise when specialising logic programming interpreters, many points raised in [58] are also valid in the logic programming setting. For example, [58] suggests to "Write your interpreter compositionally" which is exactly what we have done for our lambda interpreter in Section 8 and which makes it much easier to ensure termination of the specialisation process. [58] also warns of "data structures that contain static data, but can grow unboundedly under dynamic control" (such as a stack). The environment in the lambda interpreter contained static data but its length was fixed and so caused no problem; however if we were to add an activation stack to our interpreter in Section 8 we would have to resort to the recipes suggested in [58].

[5] In the speedup figures we suppose that the time needed for consulting is the same for the original and specialised program. In our experiments consulting the specialised program was actually slightly faster, but this may not always be the case.

We have already discuss related work in the logic programming community [105, 117, 114, 13, 21, 67, 123, 72]. In the functional community there has been a lot of recent interest in Jones optimality; see [57, 85, 116, 38]. For example, [38] shows theoretically the interest of having a Jones-optimal specialiser and the results should also be relevant for logic programming.

As far as future work is concerned, the most challenging topic is probably to provide a fully automatic binding-time analysis. As already mentioned, the binding-time analysis in [122] may prove to be a good starting point. Still, it is likely that at least some user intervention will be required in the foreseeable future to specialise more complicated interpreters.

Another avenue for further investigation is to move from interpreters to program transformers and analysers. A particular kind of program transformer is of course a partial evaluator, and one may wonder whether we can specialise, e.g., the code from Section 3. Actually, it turns out we can now do this and, surprisingly or not, the specialised specialisers we obtain in this way are quite similar to the one generated by LOGEN directly. This issue is investigated in [22], proving some first encouraging results.

In conclusion, we have shown how to use offline specialisation in general and LOGEN in particular to specialise logic programming interpreters. We have shown how to obtain Jones-optimality for simple self-interpreters, as well as for more involved interpreter such as a debugger. We have also shown how to specialise interpreters for other programming paradigms, using more sophisticated binding-types. We have also presented some experimental results, highlighting the speedups that can be obtained, and showing that the LOGEN system can be useful basis for generating compilers for high-level languages. Indeed, we soon hope to be able to apply LOGEN to derive a compiler from the interpreter in [74], and then compiling high-level B specifications into Prolog code for fast animation and verification.

# Part II

# Modular Specialisation

In this part of the deliverable, we describe how we have adapted the LOGEN partial evaluation system in particular in order to specialise modular programs (and specialising them in a modular fashion, i.e., allowing one to seperately specialise different modules).

## 10 Introduction

Making a specialisation tool suite modular poses interesting challenges. In traditional compilation, modules are divided into a public part (interface) and a private part (implementation). If user code changes in the implementation part, no other module needs recompilation. If the interface part of a module changes, only the modules that imported the modified interface need recompilation. On the other hand, in most approaches to specialisation, even with no change of any interface, implementation changes can have two non-local influence:

- they can influence the compilation of code *called* by the modified code. This is typical to specialisation: adding or changing calls will change the way the called code needs to be specialised.

- they can also influence the compilation of code that *calls* the modified code. This is only the case if cross-module unfolding is performed; it is similar to cross-module inlining of function calls in traditional compilers.

We will not study the second case in detail in the sequel because it is relatively well-known. It influences recompilation in the same direction as interface changes does, i.e. from callee to caller modules. It can theoretically be accounted for by considering the whole inlined predicate body as part of the interface for the module. In practice it can be implemented by explicit dependency tracking.

The first case is more interesting because it means that simply compiling each module separately in some correct order is not sufficient. In the sequel we describe how LOGEN has been extended to support this kind of modular compilation.

# 11 Making specialisation modular

We have adapted the LOGEN partial evaluation system to specialise modular programs incrementally. In analogy with modular compilation, we have designed a scheme that allows specialisation to be performed one module at a time, while tracking sufficient dependency information to know which modules need respecialisation after some of the source files have been modified.

## 11.1 Generating extension

LOGEN works by translating each source module (pl file) into a *generating extension* (gx file) containing the definition of specialising predicates: for each predicate for the source module, the gx file has a corresponding predicate whose role is to generate a specialised version of the original predicate. The predicate in the gx file has its name suffixed with "_u" and an extra argument "Code" that gets bound to the specialised Prolog code.

Consider, for example, the predicate `map(Goal, List1, List2)`. LOGEN will produce in the generating extension a predicate `map_u(Goal, List1, List2, Code)`. Its role is to produce residual code: the call, say, `map_u(q, [A,B,C], [d,e,f], Code)` will produce `Code = (q(A,d), q(B,e), q(C,f))`. This code is then used to define a specialised predicate `map__0(A,B,C)`. (Specialised predicates get arbitrarily mangled names.)

The specialised code typically contains calls to other specialised predicates, for example when specialising a predicate `p` that calls `map`. This triggers a chain of specialisations that can potentially sweep through the entire program, across modules, in an uncontrollable way.

However, if we disregard unfolding, then the specialised code should only contain a *call* to specialised version of the predicates it uses, not actual code coming from these other predicates. For example, the specialisation of

```
p(Goal, List) :- map(Goal, List, [hel,lo,world]).
```

for the call pattern `p(q, [A,B,C])` might be `Code = map__0(A,B,C)`, where `map__0` is the predicate of the above example.

Thus, to build the specialised version of a predicate it is sufficient to know the *name* of the specialised predicates that we must call. The definition of `p__0` containing a call to `map__0` can be generated before the definition of `map__0` itself.

So, to make specialisation modular, during the specialisation of a predicate we must limit ourselves to looking up names of other predicates, or inventing new names if needed. A different tool is used to control the process globally and specialise the called predicates.

## 11.2 Structure of the gx file

In the non-modular implementation of LOGEN, the predicate `foo_u` in the generating extension was responsible for looking up the specialisation pattern in a memo table, and generating and storing the corresponding code if it was not found in the memo.

These bookkeeping aspects have been dissociated from the specialisation process itself in the module-aware version of LOGEN. Each source code predicate `foo` triggers the creation of three predicates in the gx file:

- `foo_request` looks up the specialisation pattern in the memo table. If it is not found there, a new mangled name is invented using a counter (e.g. `foo__2`) and saved for future reference. No specialisation is actually performed.
- `foo_u` is the specialiser proper.
- `foo_spec` generates the code for a particular specialisation pattern by calling `foo_u` and writing the result at the proper location.

## 11.3 The memo and spec files

For bookkeeping, LOGEN creates for each pl source module two extra files: a memo file acting as look-up table matching specialisation patterns to mangled names, and a spec file containing the actual specialised predicates. The set of all spec files (and only them) constitutes the final, fully specialised program.

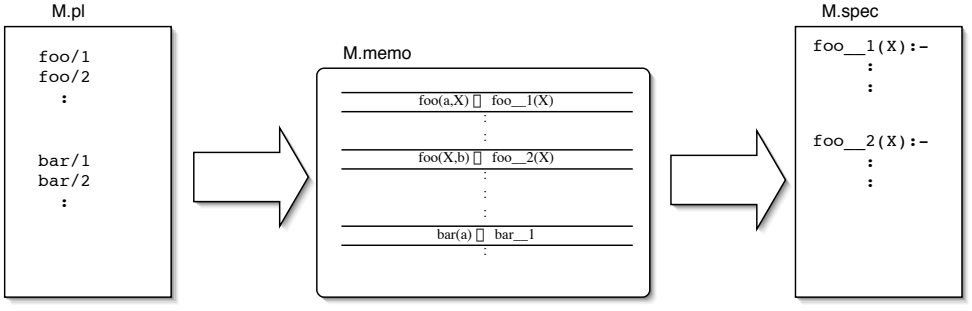



Figure 6: Specialisation via *memo* and *spec* files

For example, suppose we have a predicate `foo` in a module M that calls predicate `bar` in module N. To specialise it with a given pattern, we use `foo_spec` in file `M.gx`. This call requires the following information:

- the `M.gx` file, for `foo_u`, called by `foo_spec`;
- the `N.gx` file, for `bar_request`, called by `foo_u`;
- the `N.memo` file, where `bar_request` looks up and possibly adds a mangled name for the specific call pattern of `bar`.

The actual specialisation of `bar` is not performed, but the call to `bar_request` has recorded this pattern in `N.memo`. The basic idea is thus to specialise predicates one module at a time. The memo file records, besides the mangled names, which call patterns have been requested and which ones have been done.

## 11.4 Module driver

Given that specialising a predicate in module M only requires the gx and memo files for the modules directly used by M, it makes sense to specialise all predicates of M before proceeding to another module. An external loop called the "module driver" loads the `M.memo` file and enumerates all specialisation patterns that are marked as requested and not done. For each of them, the procedure described in 11.3 is applied.

This may create new requested entries in the memo table of both M and other modules that M directly depends on. The whole operation must thus be repeated until M reaches a fixed point with only "done" entries left in `M.memo`. Then the procedure must be repeated for all modules (which might sometimes create new entries in `M.memo` as well), until we reach a global fixed point with only "done" entries left in all the memo files.

The entry points for this process are specified by the user. Specialisation propagates recursively through the whole program, though the process is now modular: only one module and its direct dependencies need to be loaded in memory and processed at a time, including a small set of short memo files.

## 11.5 Example

This section expands the example of 11.3 with real code. It shows all the steps and the generated Prolog code. Assume the source files of figures 7 and 8, which together define a matcher that looks for a substring `Pat` inside of a string `T`.

Assume that the user (or some other part of the program) requests the specialisation pattern `match("xy", T)`. A line similar to the following one is entered into `m.memo`:

```
memo_table(m, match("xy", T), match__0(T), requested).
```

```
:- use_module(n).

:- residual match/2.
:- filter match(static,dynamic):match.
logen(match, match(Pat,T)) :-
        logen(memo, match1(Pat,T,Pat,T)).
```

Figure 7: The `m.pl` source file

```
:- residual match1/4.
:- filter match1(static,dynamic,static,dynamic):match1.
logen(match1, match1([],Ts,P,T)).
logen(match1, match1([A|Ps],[B|Ts],P,[X|T])) :-
        logen(rescall, A\==B),
        logen(memo, match1(P,T,P,T)).
logen(match1, match1([A|Ps],[A|Ts],P,T)) :-
        logen(unfold, match1(Ps,Ts,P,T)).
```

Figure 8: The `n.pl` source file

LOGEN starts by generating gx files for all the involved source modules. These files are a static translation of their corresponding source files; they do not depend on other modules. Figure 9 shows the generated source of `match_u`. Note the pattern of variables in the call to `match1_request`, which corresponds to the call found in the original `match`.

```
match_u(B,C,Code) :-
        match1_request(B,C,B,C,Code).
```

Figure 9: The `match_u` predicate in `m.gx`

We then specialise the module `m`: we load the files `m.gx`, `m.memo`, `n.gx` and `n.memo` (the latter is currently empty) and iterate through the entries of the `memo_table` for the module `m` that are marked as `requested`. This triggers a call to `match_spec`:

1. `match_spec` calls `match_u` to generate the residual code;
2. `match_u` calls `match1_request` defined in `n.gx`;
3. `match1_request` invents a mangled name for `match1`, say `match1__42`, and inserts the following line in `n.memo`:

```
memo_table(n, match1("xy",T,"xy",U), match1__42(T,U), requested).
```

The residual code generated by `match1_request` is a call to this newly created predicate, which `match_spec` writes into the `m.spec` file. It finally marks the originally requested specialisation pattern as “done”.

```
match__0(T) :-
        match1_42(T,T).
```

Figure 10: The generated file `m.spec`

Because `n.memo` contains a “requested” entry, specialisation must now proceed to module `n`. This time we only need to load `n.gx` and `n.memo`. The requested entry triggers a call to `match1_spec`. The corresponding residual code will now be completely generated by recursive calls to `match1_u`, which does not depend on external modules any more. Finally, `match1_spec` generates the code of figure 11 and updates the `n.memo` file to:

```
memo_table(n, match1("xy",T,"xy",U), match1__42(T,U), done).
```

which completes the specialisation.

```
match1__42([A|_], [_|B]) :-
        x \== A,
        match1__42(B, B).
match1__42([x,A|_], [_|B]) :-
        y \== A,
        match1__42(B, B).
match1__42([x,y|_], _).
```

Figure 11: The generated file `n.spec`

# 12 Tracking changes in the source

If the source code of a module M is modified, the following files are invalid:

- `M.gx`, which can easily be regenerated;
- `M.spec`, which must be thrown away;

- `M.memo`, which must *not* be discarded: instead, we just mark all the entries as "requested" again.

The purpose of keeping the memo file is to keep track of which predicates, and which specialisation patterns, the modules using M are asking for. All these patterns are still present in `M.memo`.

After having invalidated as above all the files corresponding to changes in the source files, then we can restart the specialisation process described in 11.4. This will only respecialise M, unless M now asks for new specialisation patterns for predicates from other modules. This case which is taken care of as above, without clearing any more gx, spec or memo file. For example, say we change the definition of `match` (see above) so that it now prints debugging information when it is called. Then the `m.gx` file will be regenerated, and `match_spec` will again call `match_u`. This puts the `write` in the residual code, and then calls `match1_request` as previously. This has the effect of generating a residual call to the same `match1__42`, because the pattern did not change. The files for the module `n` are not modified, and `n` will not have to be specialised again.

## 12.1 Dead patterns

The process described above is robust in the sense that the set of spec files after respecialisation is guaranteed to contain the optimal specialised code. However, it can contain dead code as well, because no entry is ever removed from the memo files.

To continue the example of the previous section, if we had modified `m.pl` to change the call pattern to `match1`, then a new `match1__43` would have been invented, and then implemented in `n.spec`. But `n.memo` and `n.spec` still mention `match1__42`, though it is no longer used.

In other words, with time, while the source undergoes modification, the set of specialisation patterns only grows. It is likely that some of these patterns were requested by source code long gone. The easier solution is from time to time to completely clean up all the spec and memo files and start specialisation from scratch again.

The current implementation marks each memo file entry with an additional flag ("user" or "internal") to distinguish between entries explicitely requested by the user and entries merely generated by the specialisation of other predicates. This allows the memo file to be cleaned up without removing the "user" requested patterns.

# 13 Future work

We presented an approach to modular specialisation, which still has to be validated by large-scale tests. Its implementation in LOGEN is not complete at the moment. The missing piece is the module driver (11.4).

## 13.1 Unfolding

We outline below a possible technique to allow unfolding, i.e. inlining code across modules during specialisation.

Each module can record (e.g. in the memo file) which other modules it contains code from. More precisely, a module N is listed in M.memo exactly if a predicate from module N was (directly or indirectly) unfolded into a specialised predicate written in M.spec.

This information is easy to collect at specialisation time, by adding to each `foo_u` predicate in the gx files an additional argument through which it can return, in addition to the specialised Prolog code, a list of modules of which this code contains specialised bits. Each predicate `bar_u` in module N only has to make sure that N is indeed in the list. Thus if `foo_u` directly calls `bar_u` and sticks the code produced by `bar_u` into a larger piece of code, the module dependency list will contain both M (by `foo_u`) and N (by `bar_u`). Then a reference to M and N are recorded into `M.memo` when the resulting code is written to `M.spec`.

When the user modifies the source of a module M, we can simply clear the spec files of all the modules whole memo file contains a reference to M, instead of just `M.spec`.

While causing potentially large-scale respecialisation, this approach is probably acceptable because unfolding across modules is expected to be limited.

## 13.2 Related work

Modular specialisation presents no particular theoretical challenge; the practical challenge is to manage the dependencies correctly, using appropriate internal data. While modularisation can be seen as a nice and useful programming language feature, it is common to use it to minimize, for large programs, the increasingly large amount of time needed to perform static analysis like specialisation or compilation.

There is no general recipe to make a given static analyser modular. The solution presented here is specific to the LOGEN system. Many other kinds of systems (compilers, specialisers, type inferencers...) are capable of modular analysis.

## 13.3 Conclusion

We expect to be able to validate the approach presented in the present chapter on large applications. Our solution is purely practical; it has no theoretical draw-backs like loss of precision. The currently implemented part limits unfolding to intra-module calls, but we hope to be able to lift even this restriction.

# Part III

# CLP Specialisation

This part of the delivarable demonstrates that the *cogen* approach is also applicable to the specialisation of constraint logic programs and leads to effective specialisers. We present the basic specialisation technique for CLP(R) and CLP(Q) programs and show experimental results using the LOGEN system.

## 14 Introduction

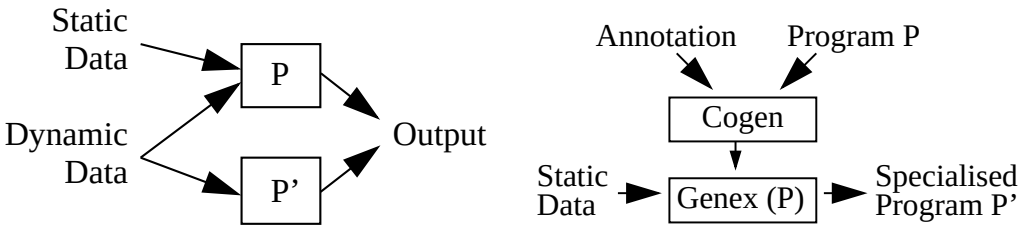
Program specialisation, also called partial evaluation (see, e.g., [59]), is an automatic technique for program optimization. Specialisation optimises programs by distinguishing between static and dynamic input data. Using the static data, parts of the original program can be evaluated at specialisation time, resulting in a hopefully more efficient residual program. The residual program is only dependent on the dynamic data (Fig. 12(a)) and can offer a substantial speed increase.

Despite some recent interest, there has been surprisingly little work on specialization of constraint logic programs. Indeed after some work in the early 90s [108, 107] there has been a long period of relative inactivity, especially compared to the success that constraint logic programming has encountered for practical applications. Only very recently, new research is emerging [27, 28, 100, 118] which is trying to tackle this difficult but practically relevant problem.

This paper presents an introduction to program specialisation of Constraint Logic Programs (CLP) and presents our newly developed technique and its implementation. We illustrate our technique and implementation on several examples and we also present experiments which evaluate the power and efficiency of our implementation. Our work presents the first offline specialiser for CLP, and it is also the first compiler generator for CLP. Our goal was to develop a system with fast and predictable specialisation times, and plan to integrate the tool into the Ciao Prolog system. To ensure wide applicability we also cater for non-declarative features.

### 14.1 Offline vs Online Specialisation

In an *offline* specialiser almost all the control decisions are taken before the actual specialisation phase in a preliminary analysis phase referred to as *binding-time analysis* (BTA). *Online* partial evaluators typically do not make use of such a preliminary phase phase but instead take their decisions on the fly, using the actual values of the static data. Note, however, that offline




(a) The specialised program P' is only dependent on the dynamic input

(b) Overview of the Cogen approach. Given a program and an annotation the Cogen produces a Genex, or specialised specialiser

Figure 12: Program Specialisation

partial evaluators maintain — during specialisation — a list of calls that have been previously specialised or are pending [59]. This is called *memoisation*.

Online partial evaluators are in principle more powerful, as they base their control decisions on more precise information. However, one of the big advantages of the offline approach is the efficiency of the specialisation process itself: indeed, once the annotations have actually been derived, the specialiser is relatively simple, and can be made to be very efficient, since all decisions concerning local control are made before and not during specialisation. Other advantages of the offline approach is the better predictability of the output, i.e., it is easier to predict beforehand (based on the outcome of the BTA) what will happen during the specialisation phase. The simplicity of the specialiser also means that it is much easier to achieve *self-application*, i.e., specialise the specialiser itself using partial evaluation. Self-application enables a partial evaluator to generate so-called "compilers" from interpreters using the second Futamura projection and a compiler generator ($cogen$) using the third Futamura projection (see, e.g., [59]). However, the actual creation of the $cogen$ according to the third Futamura projection is not of much interest to users since $cogen$ can be generated once and for all when a specialiser is given. This is known as the *cogen-approach* and has been successfully applied in many programming paradigms [8, 104, 50, 52, 9, 3, 78].

# 15 Logen

LOGEN [78] is an offline specialiser for logic programs which uses the $cogen$ approach. Given an annotated version of program P, a(P), specifying for example which inputs will be static and dynamic, LOGEN produces a specialised specialiser, or Genex (generating extension) for the

program P. Running the Genex with particular values for the static inputs produces the specialised program (Fig. 12(b)). As the Genex is not dependent on actual data values it can be reused to produce different specialised programs. Using this approach the efficiency of the specialisation process is greatly improved in situations where the same program is specialised multiple times.

The annotated version a(P) employed by LOGEN contains:

1. for every predicate, a description of which arguments will be *static* or *dynamic*. It is also possible to only annotate certain parts of arguments as static.

2. for every predicate call in the program, whether this call should be *unfolded* or *memoised*, i.e., whether it should be performed within the specialiser or whether the call should be performed at runtime. In the former case, the Genex generated by LOGEN will replace (during specialisation) this predicate call in the body by its definition, performing all the needed substitutions. In the latter case, the predicate call will be *generalised* by replacing all parts which are marked as *dynamic* by a fresh variable. It will then be checked whether the generalised call has been encountered before. If it has not been encountered before, the original program will be specialised for this generalised call.

## 15.1 Logen for CLP($R$) and CLP($Q$)

Constraint Logic Programming over the real domain, CLP($R$) , and the rational domain, CLP($Q$) , offer a powerful mathematical solver for the domains of real and rational numbers. The CLP($R$) and CLP($Q$) schemes used in this document and related tool are instances of the general Constraint Logic Programming Scheme introduced by Jaffar *et al.* [55].

Specialisation of CLP($R$) or CLP($Q$) programs using existing offline specialisation techniques causes problems as the program state is not limited to the goal stack but also populates a constraint store. This means that LOGEN [78] cannot properly handle CLP programs. Indeed, LOGEN either performs *all* the constraint processing at specialisation time, or *all* the constraint processing at runtime — it is not possible to *partially evaluate* constraints. This is obviously a serious limitation and with the increasing adoption of CLP languages by industry it is important that tools allow for efficient specialisation of CLP programs.

Based upon the current LOGEN system we have thus developed a new version of LOGEN that can handle full CLP($R$) or CLP($Q$) programs. It supports constraint specialisation across predicates by memoising constraints and retains the full power of the original LOGEN to specialise ordinary logic programming constructs. In the next section we show how this is achieved.

# 16 Specialisation of pure CLP($R$) and CLP($Q$) Programs

In this section we explain how the two main operations of LOGEN, unfolding and memoisation, are adapted in order to handle CLP.

## 16.1 Unfolding and Simplification

The classical unfold transformation replaces a predicate call with the predicate body, performing all the needed substitutions. In CLP($R$) the state of uninstantiated variables is held in the constraint store, after unfolding the residual constraints have to be extracted from the constraint store, projected and simplified and then added back into the residual program.

Let us examine the trivial CLP($R$) program in Fig. 13, which multiplies $X$ by an integer $Y$ to give $R$. Fig. 14 demonstrates how to unfold this program for the call `multiply(X,2,R)`. After each recursive call to multiply, a new constraint is added to the constraint store ($C_{1..3}$). After the unfolding has completed the final constraints in $C_3$ and variable assignments can be extracted. These are then projected onto the variables $X, R$ of the top-level query and simplified to produce the following residual program:

```
multiply(X,2.0,R) :-   {R = 2.0 * X).
```

Careful attention must be paid to the simplification of the residual constraints. During unfolding an entailment check ensures that redundant clauses are removed from the specialised program. [87] demonstrates the optimizations available through constraint reordering and removal when the removal does not effect control flow. If a constraint is likely to fail and hence cause backtracking then it should be added to the constraint store as early as possible to ensure less time is wasted in unneeded calculations.

```
multiply(_,Y,R) :-  {Y = 0.0, R = 0.0}.
multiply(X,Y,R) :-  {Y > 0 ,Y1 = Y -1, R = X + R1},
                    multiply(X,Y1,R1).
```

Figure 13: Trivial CLP($R$) Multiplication predicate

## 16.2 Memoisation

In the original LOGEN, when a call $c$ is memoised it is first generalised, by replacing any parts of the call that are marked as dynamic by a fresh variable. For example, if $c = p(2, 3)$ and if the

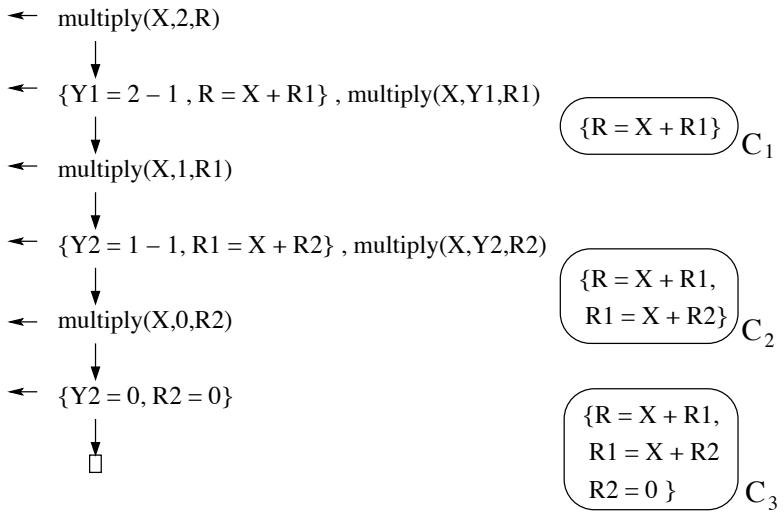



Figure 14: The multiply predicate is unfolded, producing the residual constraints $C_3$

second argument to $p$ is marked dynamic, we obtain a generalised call $c' = p(2, V)$. LOGEN (or rather the generating extension generated by LOGEN) then searches its memo table for a variant of $c'$. If $c'$ is a variant of any previously memoised call, $c$ will be replaced with the residual call from the memo table (this might e.g. be $p\_2(V)$). Otherwise $c'$ has to be specialised on its own, and residual code has to be generated for it.

In the CLP setting the original LOGEN memoisation must be extended to take into account the constraint store. As a CLP variable may be uninstantiated but still bound to constraints it can not always be clearly marked as static or dynamic. A new variable type, constraint, was introduced so that the constraints can be propagated throughout the specialised program. A variable can still be marked as dynamic to limit constraint propagation.

When a call, $c$, is memoised for the first time the current constraints, projected onto the arguments of $c$, must also be stored. During subsequent specialisation, constraints can be discarded if they are already entailed by the system. Therefore a memoised call can only be reused if the current constraints are at least as restrictive as the constraints stored for the memoised call.

## 16.3 Rounding Errors with CLP($R$)

Currently the specialisation phase uses the Rational domain, CLP(Q), to generate specialised code for the CLP($R$) engine. During the specialisation phase the residual constraint store becomes part of the specialised program. Fig. 15 demonstrates it is not always possible to retrieve exact numbers from the CLP($R$) engine and therefore truncation errors can be introduced into

the specialised program.

```
| ?- {21/20 * Y > X},{21/20*X > Y}.
{Y-1.05*X<-0.0},
{Y-0.9523809523809523*X>0.0} ?
yes
```

Figure 15: Demonstration of CLP(R) rounding problems. The output from the CLP engine is dependent on the ordering of the variables.

# 17 Examples and Experiments

In this section we illustrate our technique on a non-trivial example. Fig. 16 calculates the balance of a loan over $N$ periods. `Balances` is a list of length $N$. The interest rate is controlled by the loan scheme and decided by the amount of the initial loan. The map predicate is used to apply the loan scheme over the list of balances.

## 17.1 Unfolded Example

In Fig. 17 the loan predicate has been specialised to calculate the balances over two periods for a principal loan over 4000. As the length of the list is known all of the recursive calls can be executed at specialisation time. The map, scheme, and calcLoan calls have been unfolded and the resultant code has been inlined in the specialised code. The two redundant loan schemes have been removed from the final code. The specialised predicate (Fig. 17) runs 68% faster than the original predicate in Fig. 16.

## 17.2 Memo Example

In Fig. 18 the map predicate from the loan program has been specialised to use either scheme1 or scheme2. The length of the list has not been specified so the recursive call must be memoised. The calls in the body of map have been unfolded and the residual code inlined in the specialised code. The removal of the overhead from the univ and call operators combined with the simplification of the loan calculation to include the hard coded interest rate produces a 57% speed up over the original predicate.

```
loan(Principal, Balances, Repay) :-   {Principal>=7000},
            Term = [Principal|Balances], map(scheme1, Term, Repay).
loan(Principal, Balances, Repay) :-  {Principal >= 4000, Principal<7000},
            Term = [Principal|Balances], map(scheme2, Term, Repay).
loan(Principal, Balances, Repay) :-   {Principal >= 1000, Principal<4000},
            Term = [Principal|Balances], map(scheme3, Term, Repay).
loan(Principal, Balances, Repay) :-  {Principal >= 0, Principal<1000},
            Term = [Principal|Balances], map(scheme4, Term, Repay).

scheme1(Amount, NewAmount, Repayment) :-
  {Interest = 0.005}, calcLoan(Amount, NewAmount, Interest,Repayment).
scheme2(Amount, NewAmount, Repayment) :-
  {Interest = 0.01}, calcLoan(Amount, NewAmount, Interest,Repayment).
scheme3(Amount, NewAmount, Repayment) :-
  {Interest = 0.015}, calcLoan(Amount, NewAmount, Interest,Repayment).
scheme4(Amount, NewAmount, Repayment) :-
  {Interest = 0.02}, calcLoan(Amount, NewAmount, Interest,Repayment).

map(_,[_],_).
map(SCHEME,[H1,H2|Tail], Repayment) :- Call =.. [SCHEME,H1,H2, Repayment],
            call(Call), map(SCHEME,[H2|Tail], Repayment).

calcLoan(Amount,NewTotal, Interest, Repayment) :-
  {NewTotal = Amount + (Amount * Interest) - Repayment}.
```

Figure 16: Loan.pl, calculates the balance of a loan over $N$ periods for a given loan scheme and repayment.

## 17.3 Summary of experimental results

Table 3 summarises our experimental results. The timings were obtained by using SICStus Prolog 3.10 on a 2.4 GHz Pentium 4. The second column contains the time spent by cogen to produce the generating extension. The third column contains the time that the generating extension needed to specialise the original program for a particular specialisation query. The fourth column contains the time the specialised program took for a series of runtime queries and the fifth column contains the results from the original programs. The final column contains the speedup of the specialised program as compared to the original. Full details of the experiments (source code, queries,...) can be found at [71].

```
loan__1(Principal,C,D,E) :-
  { Principal >= (7000), Principal = (((200/201)*C) + ((200/201)*E)),
     D = (((201/200) * C) - E)  }.
loan__1(Principal,G,H,I) :-
 { Principal < (7000),
   Principal = (((100/101)*G) + ((100/101)*I)), H = (((11/10)*G) - I) }.
loan(Principal,[B,C],D) :-  { Principal > (4000) },  loan__1(A,B,C,D).
```

Figure 17: Specialised version of the loan predicate for loan(X,[P1,P2], R) where X $>$ 4000

```
map(scheme1,A,B) :- map__1(A,B).
map(scheme2,A,B) :- map__2(A,B).
map__1([B],C).
map__1([D,E|F],G) :- {E = ((201/200) * D ) - G },  map__1([E|F],G).
map__2([B],C).
map__2([D,E|F],G) :- { E = ((101/100) * D) - G }, map__2([E|F],G).
```

Figure 18: Specialised version of the loan example for calls map(scheme1, T, R) and map(scheme2,T,R). In this example the recursive call to map__1 is memoed as the length of the list is not known at specialisation time

# 18 Non-declarative Programs

It is well known that to properly handle Prolog programs with non-declarative features one has to pay special attention to the left-propagation of bindings and of failure [106, 102]. Indeed, for calls $c$ to predicates with side-effects (such as `nl/0`) "$c, fail$" is not equivalent to "$fail, c$". Other predicates are called "propagation sensitive" [106]. For calls $c$ to such predicates, even though $c, fail \equiv fail$ may hold, the equivalence $(c,\ X = t) \equiv (X = t,\ c)$ does not. One such predicate is `var/1`: we have, e.g., `(var(X),X=a)` $\not\equiv$ `(X=a,var(X))`. Predicates can both be propagation sensitive and have side-effects (such as `print/1`). The way this problem is overcome in the

| Program | Cogen Time | Genex Time | Runtime | Original | Speedup |
|---|---|---|---|---|---|
| *multiply* | 0 ms | 20 ms | 10 ms | 3780 ms | 37800 % |
| *loan_unfold* | 0 ms | 0 ms | 385 ms | 647 ms | 68 % |
| *loan_map* | 0 ms | 0 ms | 411 ms | 647 ms | 57 % |
| *ctl_clp* | 0 ms | 100 ms | 17946 ms | 24245 ms | 35 % |

Table 3: Experimental results

LOGEN system [78] is via special annotations which selectively prevent the left-propagation of bindings and failure. This allows the LOGEN system to handle almost full Prolog[6], while still being able to left-propagate bindings whenever this is guaranteed to be safe. In a CLP setting, the whole issue gets more complicated in that one also has to worry about the left-propagation of constraints. Take for instance the clause `p(X) :- var(X),{X=<2}` and suppose we transform it into `p_1(X) :- {X=<2},var(X)`. The problem is now that the query `{X>=2},p_1(X)` to the specialised program fails while the original query `{X>=2},p(X)` succeeds with a computed answer `X=2.0`. To overcome this problem we have extended the scheme from [78] to enable us to selectively prevent the left-propagation of constraints. Using our new system we are now in a position to handle full CLP programs with non-declarative features. Take for example the following simple CLP($R$) program:

```
p(X,Y) :- {X>Y}, print(Y), {X=2}.
```

Using our system we can specialise this program for, e.g., the query `p(3,Z)` yielding the following, correct specialised program:

```
p__0(Y) :- {3>Y}, print(Y), fail.
```

# 19 Future, Related Work and Conclusions

There has been some early work on specialization of CLP programs [108, 107] and optimization [86]. There has been some recent interest in online specialisation techniques for constraint logic programs [27, 28, 100, 118, 99]. To the best of our knowledge there is no work on offline specialisation of CLP programs, and to our knowledge none of the above techniques can handle non-declarative programs.

The binding-time analysis of [78], based on a termination analysis for logic programs needs to be adapted to take the constraints into account. At this stage it is not clear whether existing termination analysis techniques for constraint logic programs such as [92] can be used to that end.

There is still scope to improve the code generator of our system, e.g., by using more sophisticated reordering as advocated in [86] Other possibilities might be to convert CLP operations into standard Prolog arithmetic (e.g., using `is/2`) when this is safe. Extension to other domains, such as CLP(FD) are being investigated. Finally, we plan to integrate LOGEN into the Ciao Prolog system, making the use of our system more transparent.

[6] Predicates which inspect and modify the clause database of the program being specialised, such as `assert/1` and `retract/1` are not supported; although it is possible to treat a limited form of them.

The specialisation of CLP programs is an important research area for partial evaluation. We have implemented a working version of an offline CLP($R$) specialiser and first results look promising.

# Part IV
# Self-Application

In this part we show how to derive a self-applicable partial evaluator from our LOGEN compiler generator system. Apart from academic curiosity, this allows one to easily generate more or less optimised specialised specialisers, just by tuning the annotations. One can also easily generate debugging versions of the specialised specialisers. It can also be used to obtain a binding-time analysis, applying the CiaoPP system on purposely generated specialisers (generated for analysis purposes and not intended to be run).

## 20 Introduction and Summary

*Partial evaluation* has received considerable attention over the past decade both in functional (e.g. [59]), imperative (e.g. [3]) and logic programming (e.g. [33, 65, 101]). In the context of pure logic programs, partial evaluation is often referred to as *partial deduction*, the term partial evaluation being reserved for the treatment of impure logic programs. We will adhere to this convention in this paper.

Guided by the *Futamura projections* (see e.g. [59]) a lot of effort, especially in the functional partial evaluation community, has been put into making systems self-applicable. A partial evaluation or deduction system is called *self-applicable* if it is able to effectively[7] specialise itself. The practical interests of such a capability are manifold. The most well-known are related to the second and third *Futamura projections* [30].

#### History of self-application for logic programming

Not surprisingly, writing an effectively self-applicable specialiser is a non-trivial task — the more features one uses in writing the specialiser the more complex the specialisation process becomes, as the specialiser then has to handle these features as well. For a long time it was believed that in order to develop a self-applicable specialiser for logic programs one needed to write a clean, pure and simple specialiser. In practice, this meant using few (or even no) impure features in the implementation of the specialiser. For this the *ground representation* [49] was believed to be key, in which variables of the source program are represented by ground constants within the specialiser. Indeed, the ground representation allows one to freely manipulate the

[7] This implies some efficiency considerations, e.g. the system has to terminate within reasonable time constrains, using an appropriate amount of memory.

source program to be specialised in a declarative manner. The *non-ground representation*, where source-level variables are represented as variables in the program specialiser, suffers from some semantical problems [90] and requires some non-declarative features (such as `findall/3`) in order to perform the specialisation.

Some early attempts at self-application [29] used the non-ground representation, but the self-applying led to incorrect results as the specialiser did not properly handle the non-declarative constructs that were employed in its implementation[8]. Other specialisers like MIXTUS [106], PADDY [102] and ECCE [79] use the non-ground representation, but none of them are able to effectively specialise themselves.

The ground representation approach towards self-application was pursued in [11], [70], [94], and [13, 44, 43] leading to some self-applicable specialisers:

- SAGE [43], a self-applicable partial evaluator for Gödel. While the speedups obtained by self-application are respectable, the process takes a very long time (several hours) and the obtained specialised specialisers are still extremely slow. This is probably due to the explicit unification algorithm required by the ground representation. To effectively specialise this much more powerful specialisation techniques would be required to obtain reasonably efficient specialisers. Similar performance problems were encountered in the earlier work [11].
- LOGIMIX [94, 59], a self-applicable partial evaluator for a subset of Prolog, including *if-then-else*, side-effects and some built-ins. LOGIMIX uses a meta-interpreter (sometimes called *InstanceDemo*) for the ground representation in which the goals are "lifted" to the non-ground representation for resolution. This avoids the use of an explicit unification algorithm, at the expense of some power[9]. Unfortunately, LOGIMIX gives only modest speedups (when compared to results for functional programming languages, see [94]), but it was probably the first practical self-applicable specialiser for a logic programming language.

Given the problem in developing a truly practical self-applicable specialiser for logic programs, the attention shifted to the *cogen approach* [50]: instead of trying to write a partial evaluation system which is neither too inefficient nor too difficult to self-apply, one simply writes a compiler generator directly. Indeed, the actual creation of the cogen according to the third Futamura projection is in general not of much interest to users since the cogen can be generated once and for all when a specialiser is given. This approach was pursued in [63, 78] leading to the LOGEN system, which can produce specialised specialisers much more efficiently than any

[8] A problem mentioned in [11], see also [94, 70].

[9] This idea was first used by Gallagher in [32, 33] and then later in [75] to write a declarative meta-interpreter for integrity checking in databases.

of the self-applicable systems mentioned above. The resulting specialisers themselves are also much more efficient.

**A new attempt at self-application**

In a sense the cogen approach has closed the practical debate on self-application for logic programming languages: one can get most of the benefits of self-application without writing a self-applicable specialiser. Still, there is the question of academic curiosity: is it really impossible to derive the cogen written by hand in [63, 78] by self-application? Also, having a self-applicable specialiser is sometimes more flexible as we may generate different cogens for different purposes (such as one with debugging enabled). One may produce more or less optimised cogens by tweaking the specialisation process, and better control the tradeoff between specialisation time and quality of the optimised code. Maybe there are other situations where a self-applicable partial evaluation system is preferrable to a cogen: Glück's specialiser projections [39] and the semantic modifiers of Abramov and Glück [1] may be such a setting.

This paper aims to answer some of these questions. Indeed, after the development of LOGEN we realised that one could translate LOGEN into a classical partial evaluator without too much difficulty. Furthermore, using new annotation facilities developed for the second version of LOGEN [78], one can actually make this partial evaluator (henceforth called LIX) self-applicable. By self-applying LIX we obtain generating extensions via the second Futamura projection which are very similar to the ones produced by LOGEN and the cogen obtained via the third Futamura projection also has lot of similarities to the code of LOGEN. The performance of this self-applicable partial evaluator is (after self-application) on par with LOGEN, and is thus much faster than any of the previous self-applicable logic programming specialisers. In the paper we also show some potential practical applications of this self-applicable specialiser.

The code of the specialiser itself is also surprisingly simple, but uses a few non-declarative features and does not use the ground representation. So, contrary to earlier belief, declarativeness and the ground representation were not the best way to climb the mountain of self-application.

In summary, Futamura's insight was that a cogen could be derived by a self-applicable specialiser. The insight in [50] was that a cogen is just a simple extension of a binding-time analysis, while our insight is that an effective self-applicable specialiser can be derived by transforming a cogen.

# 21 The Partial Evaluator

LOGEN and LIX are both offline partial evaluators. An offline partial evaluator works on an

annotated version of the source program, these annotations are used to guide the specialisation process. There are two kinds of annotations:

- **filter declarations**, indicating whether arguments to predicates are **static** or **dynamic**. This influences the global control.
- **clause annotations**, indicating how every call in the body should be treated during unfolding. These influence the local control.

## 21.1 The Basic Annotations

A common annotation format is used for both the LIX and LOGEN systems. Each call in the program is annotated using `logen/2` and arguments are annotated using filter declarations. The head of a clause is annotated with an identifier. The format of the annotations is demonstrated in the following append example:

```
:- filter append(static,dynamic,dynamic).
logen(app, append([],L,L)).
logen(app, append([H|T], L, [H|T1])) :- logen(unfold, append(T,L,T1).
```

The first argument to append has been marked as **static**, it will be known at specialisation time, and the other arguments have been marked **dynamic**. The recursive call to append is annotated for unfolding, the first argument is known thus guaranteeing termination at specialisation time. Some of the basic annotations are:

- **unfold** for reducible predicates, they will be unravelled during specialisation.
- **memo** for non-reducible predicates, they will be added to the memoisation table and replaced with a generalised residual predicate.
- **call** the call will be made during specialisation.
- **rescall** the call will be kept and will appear in the final specialised code.

## 21.2 The Source Code

We now present the main body of the LIX partial evaluator[10]. An atom $A$ is specialised by calling `lix(`$A$`,Res)`. The `memo/2` and `memo_table/2` predicates return in their second argument a call to a new specialised predicate where static arguments have been removed and dynamic ones generalised. Generalisation and filtering are performed by `generalise_and_filter/3`. It returns in its second argument the generalised call (to be unfolded) and in its third argument the call to the specialised predicate. It uses the annotations defined by `filter/2` to perform its

[10]The LIX system can be downloaded from:
`http://www.ecs.soton.ac.uk/~sjc02r/lix/lix.html`.

task. The predicate `gensym/2` is used to create unique names for the specialised predicates. The predicate `unfold/2` computes the bodies of specialised predicates. A call annotated as `memo` is replaced by a call to the specialised version. If it does not already exist it is created by `memo/2`. A call annotated as `unfold` is further unfolded; a call annotated as **call** is completely evaluated; finally, a call annotated as `rescall` is added to the residual code without modification (for built-ins that cannot be evaluated or code that is defined elsewhere). All clauses defining the new predicate are collected using `findall/3` and pretty printed. To save space the definition of `pretty_print_clauses/1` is not given.

```
:- dynamic memo_table/2,flag/2.
lix(CallToSpecialise, ResidualCall) :-
        print(':- dynamic flag/2, memo_table/2.\n'),
        print(':- use_module(library(lists)).\n'),
        memo(CallToSpecialise, ResidualCall).
memo(Call, Residual) :-
        (   memo_table(Call, Residual) ->  true
        ;   generalise_and_filter(Call, GenCall, ResidualPred),
            assert(memo_table(GenCall,ResidualPred)),
            findall((ResidualPred:-Body), unfold(GenCall,Body), Clauses),
            format('/*~k=~k*/~n', [ResidualPred,GenCall]),
            pretty_print_clauses(Clauses),
            memo_table(Call, Residual)
        ).
unfold(Head, Residual) :- ann_clause(_, Head, Body),body(Body, Residual).
body(true, true).
body((A,B), (ResA,ResB)) :- body(A, ResA), body(B, ResB).
body(logen(call,Call), true) :- call(Call).
body(logen(rescall,Call), Call).
body(logen(memo,Call), Residual) :- memo(Call, Residual).
body(logen(unfold,Call), Residual) :- unfold(Call, Residual).

generalise_and_filter(Call, GenCall, ResidualPred) :-
        filter(Call, Filter), Call=..[Head|Args],
        gen_filter(Filter, Args, GenArgs, ResArgs), GenCall=..[Head|GenArgs],
        gensym(Head, ResHead), ResidualPred =..[ResHead|ResArgs].
gen_filter([], [], [], []).
gen_filter([static|A], [B|C], [B|D], E) :-
        gen_filter(A, C, D, E).
gen_filter([dynamic|A], [_|B], [C|D], [C|E]) :-
        gen_filter(A, B, D, E).

/* code for unique symbol generation, using dynamic flag/2 */
oldvalue(Sym, Value) :- flag(gensym(Sym), Value), !.
oldvalue(_, 0).
```

```
set_flag(Sym, Value) :-
        nonvar(Sym), retract(flag(Sym,_)), !, asserta(flag(Sym,Value)).
set_flag(Sym, Value) :- nonvar(Sym), asserta(flag(Sym,Value)).

gensym(Head, ResidualHead) :-
        var(ResidualHead), atom(Head),
        oldvalue(Head, OldVal), NewVal is OldVal+1,
        set_flag(gensym(Head), NewVal), name(A__, "__"),
        string_concat(Head, A__, Head__),
        string_concat(Head__, NewVal, ResidualHead).
append([], A, A).
append([A|B], C, [A|D]) :-  append(B, C, D).
string_concat(A, B, C) :- name(A, D), name(B, E),
        append(D, E, F), name(C, F).

/* Printing and Flatten Clauses removed to save space */
/* Clause Database: automatically created from annoated file */
ann_clause(1, app([],A,A), true).
ann_clause(2, app([A|B],C,[A|D]), logen(memo,app(B,C,D))).
filter(app(_,_,_), [dynamic,static,dynamic]).
```

### 21.3 Specialised Code

To specialise code we use the `lix/2` entry point. Calling `lix(app(A,[b],C),Res)` specialises the append predicate to append `[b]` to the end of a list:

```
app__1([], [b]).
app__1([A|B], [A|C]) :- app__1(B, C).
```

The generation of the above code took $0.318$ ms[11] This is a very simple example to demonstrate the partial evaluator. The specialisation of a non-trivial Vanilla debugging interpreter is given in appendix 25.3.

## 22 Towards Self-Application

We have presented the main body of the code for the LIX system. For a partial evaluator to be self-applicable it must be able to effectively handle all of the features it uses. The system we have presented so far uses a few non-declarative features and does not use the ground representation. In this section will shall introduce the required extension to make LIX self-applicable.

[11] Benchmarks performed using SICStus Prolog 3.10.1 for Linux on a Pentium 2.4GHz with 512MB RAM.

## 22.1 The nonvar Binding-Type

We now present a new feature derived from LOGEN which is useful when specialising interpreters. This annotation will be the key for effective self-application.

In addition to marking arguments to predicates as **static** or **dynamic**, it is also possible to use the binding-type **nonvar**. This means that this argument is not a free variable and will have at least a top-level function symbol, but it is not necessarily ground. During generalisation, the top level function symbol is kept but all its sub-arguments are replaced by fresh variables. For filtering, every sub-argument becomes a new argument of the residual predicate.

A small example will help to illustrate this annotation:

```
:- filter p(nonvar).
p(f(X)) :- p(g(a)).
p(g(X)) :- p(h(X)).
p(h(a)).
p(h(X)) :- p(f(X)).
```

If we mark no calls as unfoldable, we get the following specialised program for the call p(f(Z)):

```
%%% entry point:  p(f(Z)) :- p__0(Z)
p__0(B) :-   p__1(a).
p__1(B) :-   p__2(B).
p__2(a).
p__2(B) :-  p__0(B).
```

If we mark everything except the last call as unfoldable we obtain:

```
p__0(B).
p__0(B) :- p__0(a).
```

The `gen_filter/2` predicate in the LIX source code is extended to handle the **nonvar** annotation:

```
gen_filter([nonvar|A], [B|C], [D|E], F) :-
        B=..[G|H], length(H, I), length(J, I),
        D=..[G|J], gen_filter(A, C, E, K), append(J, K, F).
```

## 22.2 Treatment of findall

In LIX `findall` is used to collect the clauses when unfolding a call; hence we have to be able to treat this feature during specialisation.

Handling `findall` is actually not much different from handling negation in [78]. There is a static version (`findall`), in which the call is executed at specialisation time, and a dynamic version (`resfindall`), where it is executed at runtime. In both cases, the second argument must be annotated. For `resfindall`, much like `resnot` in [78], the annotated argument should be deterministic and should not fail (which can be ensured by wrapping the argument into a `hide_nf` annotation, see [78]). Also, if a `findall` is marked as static then the call should be sufficiently instantiated to fully determine the list of solutions. The following code is used in the subsequent examples:

```
:- filter all_p(static,dynamic).
all_p(X,Y) :- findall(X,p(X),Y).
:- filter p(static).
p(a). p(b).
```

If the `findall` is marked as residual and we `memo p(X)` inside it then the specialised program for `all_p(a,Y)` is:

```
all_p__0(A) :- findall(a,p__1,A).
p__1.
```

If we mark `p(X)` as `unfold` we get:

```
all_p__0(A) :- findall(a,true,A).
```

For self-application, only `resfindall` is actually required. The `body/2` predicate is extended as follows:

```
body(resfindall(Vars,G2,Sols), findall(Vars,VS2,Sols)) :-
     body(G2,VS2).
```

## 22.3 Treatment of if

In the LIX code an `if-then-else` is used in `memo/2`. In this case the `if` is dynamic, the body of the conditional will be computed, along with those of the branches and an `if` statement will be constructed in the residual code. LIX is also extended to handle a static `if` which is performed at specialisation time.

```
body(resif(A,B,C), (D->E;F)) :-
           body(A, D), body(B, E), body(C, F).
body(if(A,B,C), D) :-
           (body(A, _) -> body(B, D) ; body(C, D)).
```

## 22.4 Handling the cut

This is actually very easy to do, as with careful annotation the cut can be treated as a normal built-in call. The cut must be annotated using **call**, where it is performed at specialisation time, or **rescall**, where it is included in the residual code. It is up to the annotator to ensure that this is sound, i.e. LIX assumes that:

- if a cut marked **call** is reached during specialisation then the calls to the left of the cut will never fail at runtime.
- if a cut is marked as **rescall** within a predicate $p$, then no calls to $p$ are unfolded.

These conditions are sufficient to handle the cut in a sound, but still useful manner.

# 23 Self-Application

Using the features introduced in Section 22 and the basic annotations from Section 21.1, LIX can be successfully annotated for self-application. Self-application allows us to achieve the Futamura projections as depicted in Fig. 2.

## 23.1 Generating Extensions

In Section 21.3 we specialised `app/3` for the call `app(A,[b],C)`. If a partial evaluator is fully self-applicable then it can specialise itself for performing a particular specialisation, producing a *generating extension*. This process is the second Futamura projection. When specialising an interpreter the generating extension is a compiler.

A generating extension for the append predicate can be created by calling:

`lix(lix(app(A,B,C),R),R1)`

which creates a specialised specialiser for the append predicate.

```
/*Generated by Lix*/
:- dynamic flag/2, memo_table/2.
/* oldvalue__1(_5557,_5586) = oldvalue(_5557,_5586) */
oldvalue__1(A, B) :- flag(gensym(A), B), !.
oldvalue__1(_, 0).

/* set_flag__1(_7128,_7153) = set_flag(gensym(_7128),_7153) */
set_flag__1(A, B) :- retract(flag(gensym(A),_)), !,
                     asserta(flag(gensym(A),B)).
set_flag__1(A, B) :- asserta(flag(gensym(A),B)).

/* gensym__1(_4392) = gensym(app,_4392) */
```

```
gensym__1(A) :- var(A), oldvalue__1(app, B),
                C is B+1,set_flag__1(app, C),
                name(C, D), name(A, [97,112,112,95,95|D]).
/* Printing and Flatten Clauses removed to save space */

/* unfold__1(_6925,_6927,_6929,_6956) = unfold(app(_6925,_6927,_6929),_6956) */
unfold__1([], A, A, true).
unfold__1([A|B], C, [A|D], E) :- memo__1(B, C, D, E).

/* memo__1(_2453,_2455,_2457,_2484) = memo(app(_2453,_2455,_2457),_2484) */
memo__1(A, B, C, D) :-
        (   memo_table(app(A,B,C), D) -> true
        ;   gensym__1(E), F=..[E,G,H],
            assert(memo_table(app(G,B,H),F)),
            findall((F:-I), unfold__1(G,B,H,I), J),
            format('/*~k=~k*/~n', [F,app(G,B,H)]),
            pretty_print_clauses__1(J),
            memo_table(app(A,B,C), D)
        ).
/* lix__1(_1288,_1290,_1292,_1319) = lix(app(_1288,_1290,_1292),_1319) */
lix__1(A, B, C, D) :- print('/*Generated by Lix*/\n'),
                      memo__1(A, B, C, D).
```

This is almost entirely equivalent to the proposed specialised unfolders in [63, 76]. It is actually slightly better as it will do flow analysis and only generate unfolders for those predicates that are reachable from the query to be specialised. Note the `gensym/2` predicate is specialised to produce only symbols of the form `app__N`. Generation of the above took $3.3$ ms.

The generating extension for append can be used to specialise the append predicate for different sets of static data. Calling the generating extension with `lix__1(A,[b],C,R)` creates the same specialised version of the append predicate as in Section 21.3:

```
app__1([], [b]).
app__1([A|B], [A|C]) :- app__1(B, C).
```

However using the generating extension is faster, for this small example $0.212$ ms instead of $0.318$ ms. Using a larger benchmark, unfolding (as opposed to memoising) the append predicate for a $10,000$ item list produces more dramatic results. To generate the same code the generating extension takes $40$ ms compared to $990$ ms for LIX. The overhead of creating the generating extension for the larger benchmark is only $10$ ms. Generating extensions can be very efficient when a program is to be specialised multiple times with different static data.

## 23.2 Lix Compiler Generator

The third Futamura projection is realised by specialising the partial evaluator to perform the second Futamura projection. By this process we obtain a *compiler generator* (*cogen* for short), a program that transforms interpreters into compilers. By specialising LIX to create generating extensions we create LIX-COGEN, a self-applied compiler generator. This can be achieved with the query `lix(lix(lix(Call,R),R1),R2)`. An extract from the produced code is now given:

```
/*unfold__13(Annotation, Generated Code, Specialisation Time) */
unfold__13(true, true, true).
unfold__13((A,B), (C,D), (E,F)) :-
        unfold__13(A, C, E),
        unfold__13(B, D, F).
unfold__13(logen(call,A), true, call(A)).
unfold__13(logen(rescall,A), A, true).
...
```

This has basically re-generated the 3-level cogen described in [63, 76]. In the **rescall** annotation for example, the call ($A$) will become part of the residual program, and nothing ($true$) is performed at specialisation time.

The generated LIX-COGEN will transform an annotated program directly into a generating extension, like the one found in section 23.1. However LIX-COGEN is faster: to create the same generating extension from an input program of $1,000$ predicates LIX-COGEN takes only $3.9$ s compared to $100.9$ s for LIX.

# 24 Comparison

## 24.1 Logen

The LOGEN system is an offline partial evaluation system using the cogen approach. Instead of using self-application to achieve the third Futamura projection, the LOGEN compiler generator is hand written. LIX was derived from LOGEN by rewriting it into a classical partial evaluation system. Using the second Futamura projection and self-applying LIX produces almost identical generating extensions to those produced by LOGEN. Apart from the predicate names the specialised unfolders generated by the two systems are the same:

| LOGEN Generating Extension | LIX-COGEN Generating Extension |
|---|---|
| …<br>`app__u([],A,A,true).`<br>`app__u([A\|B],C,[A\|D],E) :-`<br>`  app__m(B,C,D,E).`<br>… | …<br>`unfold__1([], A, A, true).`<br>`unfold__1([A\|B], C, [A\|D], E) :-`<br>`  memo__1(B, C, D, E).`<br>… |

While LOGEN is a hand written compiler generator, LIX must be self-applied to produce the same result as in Section 23.2. If we compare the LOGEN source code to the output in Section 23.2 we find very similar clauses in the form of `body/3` (note however, that the order of the last two arguments is reversed).

```
body(true,true,true).
body((G,GS),(G1,GS1),(V,VS)) :-
  body(G,G1,V),
  body(GS,GS1,VS).
body(logen(call,Call),Call,true).
body(logen(rescall,Call),true,Call).
```

LOGEN

```
unfold__13(true, true, true).
unfold__13((A,B), (C,D), (E,F)) :-
      unfold__13(A, C, E),
      unfold__13(B, D, F).
unfold__13(logen(call,A), true, call(A)).
unfold__13(logen(rescall,A), A, true).
```

LIX-COGEN

Unlike LIX, LOGEN does not perform flow analysis. It produces unfolders for all predicates in the program, regardless of whether or not they are reachable.

## 24.2 Logimix and Sage

Comparisons of the initial *cogen* with other systems such as LOGIMIX, PADDY, and SP can be found in [63]. The time taken to produce the generating extensions was 50 times faster using LOGEN (0.02 s instead of 1.10 s or 0.02 s instead of 0.98 s) and the specialisation times were about 2 times faster. It is likely that a similar relationship holds between LIX and LOGIMIX. Self-applying SAGE is not possible for normal users, so we had to take the timings from [43]: generating the compiler generator takes about $100$ hours (including garbage collection), creating a generating extension for the examples in [43] took at least $7.9$ hours ($11.8$ hours with garbage collection). The speedups from using the generating extension instead of the partial evaluator range from $2.7$ to $3.6$ but the execution times for the system (including pre- and post-processing) still range from $113$ s to $447$ s.

# 25 New Applications

Apart from the academic satisfaction of building a self-applicable specialiser, we think that there will be practical applications as well. We elaborate on a few in this section.

## 25.1 Several Versions of the Cogen

In the development of new annotation and specialisation techniques it is often useful to have a debugging specialisation environment without incurring any additonal overhead when it is not required. Using LIX we can produce a debugging or non-debugging specialiser from the

same base code, the overhead of debugging being specialised away when it is not required. By augmenting LIX with extra options we can produce several versions of the cogen depending on the requirements:

- a debugging cogen, useful if the specialisation does not work as expected
- a profiling cogen
- a simple cogen, whose generating extensions produce no code but which can be fed into termination analysers or abstract interpreters to obtain information to check the annotations.

We could also play with the annotations of LIX to produce more or less aggressive specialisers, depending on the desired tradeoff between specialisation time, size of the specialised code and the generating extensions, and quality of the specialised code. This would be more flexible and maintainable than re-writing LOGEN to accomodate various tradeoffs.

## 25.2 Extensions for Deforestation/Tupling

LIX is more flexible than LOGEN: we do not have to know beforehand which predicates are susceptible to being unfolded or memoised. Hence, LIX can handle a potentially unbounded number of predicates. Using this allows LIX to perform a simple form of conjunctive partial deduction [24].

For example, the following is the well known double append example where conjunctive partial deduction can remove the unnecessary intermediate datastructure XY (this is *deforestation*):

```
doubleapp(X,Y,Z,XYZ) :- append(X,Y,XY), append(XY,Z,XYZ).
append([],L,L).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

When annotating this example for LIX we can now simply annotate a conjunction as memo (which is not allowed in LOGEN):

```
ann_clause(1,doubleapp(A,B,C,D), (memo((append(A,B,E),append(E,C,D))))).
...
```

Running LIX on this will produce a result where the intermediate datastructure has been removed (after post-processing, as in [24]):

```
/* atom specialised: doubleapp(A,B,C,D), benchmark info: 0 ms */
doubleapp(A,B,C,D) :- doubleapp__0(A,B,C,D).
append__2([],B,B).
append__2([C|D],E,[C|F]) :- append__2(D,E,F).
```

```
conj__1([],[],B,B).
conj__1([],[C|D],E,[C|F]) :- append__2(D,E,F).
conj__1([G|H],I,J,[G|K]) :- conj__1(H,I,J,K).
doubleapp__0(B,C,D,E) :- conj__1(B,C,D,E).
```

For this example to work in LOGEN we would need to declare every possible conjunction skeleton beforehand, as a specialised unfolder predicate has to be generated for every such conjunction. LIX is more flexible in that respect, as it can unfold a conjunction even if it has not been declared before.

We have also managed to deal with the rotate-prune example from [24], but more research will be needed into the extent that the extra flexibility of LIX can be used to do deforestation or tupling in practice. It should be possible, for example, to find out whether there is a bounded number of conjunction skeletons simply by self-application.

## 25.3 A Non-Trivial Interpreter Example

We now demonstrate that LIX can handle more complicated examples by introducing a Vanilla debugging interpreter using structured conjunctions, where every conjunction is either $empty$ or is an $and$ and the left part is an atom and the right handside a conjunction. The code below implements a tracing version of `solve` which takes two extra arguments: a counter for the indentation level and a list of predicates to trace.

```
/* Clause DB */
dclause(app([],L,L), empty).
dclause(app([H|X], Y, [H|Z]), and(app(X,Y,Z),empty)).
dclause(rev(A,B), and(rev(A,[],B),empty)).
dclause(rev([],A,A), empty).
dclause(rev([A|B],C,D), and(rev(B,[A|C],D),empty)).
dclause(rev_app(A,B,C), and(rev(A,D),and(app(D,B,C),empty))).

/* Vanilla Debugging Interpreter */
dsolve(empty,_,_).
dsolve(and(A,B), Level,ToTrace) :-
  (debug(A,ToTrace) ->
      ( indent(Level), print('Call: '), print(A), nl,
        dsolve(A,s(Level), ToTrace),
        indent(Level), print('Exit: '), print(A), nl)
  ;
      dsolve(A,Level, ToTrace)),
  dsolve(B,Level,ToTrace).
```

```
dsolve(X,Level, ToTrace) :-
        dclause(X,Y),
        dsolve(Y,Level, ToTrace).
indent(0).
indent(s(A)) :- print('>'), indent(A).
debug(Call,ToTrace) :-
        Call =.. [P|Args],
        length(Args, Arity), member(P/Arity, ToTrace).
```

For `dsolve(rev_app(A,B,C), L, [app/3])` we get the following efficient tracing version of our object program, where the debugging statements have been weaved into the code. This specialised code now runs with minimal overhead, and there is no more runtime checking for whether a call should be traced or not.

```
/* indent/1 */
indent__1(0).
indent__1(s(A)) :- print(>), indent__1(A).
/* rev/3 */
dsolve__3([], A, A, _).
dsolve__3([A|B], C, D, E) :- dsolve__3(B, [A|C], D, E).
/* rev/2 */
dsolve__2(A, B, C) :- dsolve__3(A, [], B, C).
/* app/3 */
dsolve__4([], A, A, _).
dsolve__4([A|B], C, [A|D], E) :-
        indent__1(E),print('Call: '),print(app(B,C,D)),nl,
        dsolve__4(B, C, D, s(E)),
        indent__1(E),print('Exit: '),print(app(B,C,D)),nl.
/* rev_app/3 */
dsolve__1(A, B, C, D) :-
        dsolve__2(A, E, D),
        indent__1(D),print('Call: '),print(app(E,B,C)),nl,
        dsolve__4(E, B, C, s(D)),
        indent__1(D),print('Exit: '),print(app(E,B,C)),nl.
```

Running the specialised code for `dsolve__1([a,b,c],[d,e,f],C,0)`, the same as running `dsolve(rev_app([a,b,c],[d,e,f],C),0,[app/3])` in the original, produces the following trace:

```
| ?- dsolve__1([a,b,c],[d,e,f],C,0).
Call: app([c,b,a],[d,e,f],_555)
```

```
>Call: app([b,a],[d,e,f],_1056)
>>Call: app([a],[d,e,f],_1228)
>>>Call: app([],[d,e,f],_1436)
>>>Exit: app([],[d,e,f],[d,e,f])
>>Exit: app([a],[d,e,f],[a,d,e,f])
>Exit: app([b,a],[d,e,f],[b,a,d,e,f])
Exit: app([c,b,a],[d,e,f],[c,b,a,d,e,f])
C = [c,b,a,d,e,f] ?
yes
```

# 26 Conclusions and Future Work

We have presented an implemented, effective and surprisingly simple, self-applicable partial evaluation system for Prolog and have demonstrated that the ground representation is not required for a partial evaluation system to be self-applicable. The LIX system can be used for the specialisation of non-trivial interpreters, and we hope to extend the system to use more sophisticated binding types developed for LOGEN.

While LIX and LOGEN essentially perform the same task, there are some situations where a self-applicable partial evaluation system is preferrable. LIX can potentially produce better generating extensions, using specialised versions of `gensym` and performing some of the generalisation and filtering beforehand. We have shown the potential for the use of LIX in deforestation, and in producing multiple cogens from the same code. The overhead of a debugging cogen can be specialised away when it is not required or a more aggressive specialiser can be generated by tweaking the annotations.

# Part V

# Framework for A Fully Automatic Binding Time Analysis

In this part of the deliverable, we link the previously formalised binding-types with binary clause semantics. Offline partial evaluation techniques rely on an annotated version of the source program to control the specialisation process. These annotations are required to ensure termination of the partial evaluation. We present an outline of the algorithm for generating these annotations automatically. The process is made up from indenpendent components which if missing could be replaced by a manual procedure or oracle, making the procedure semi-automatic. Finally, a *draft* of a worked example is being presented. This represents work in progress and further work needs to be carried out.

## 27 Offline Partial Evaluation

Most off-line approaches perform what is called a *binding-time analysis* prior to the specialization phase. In essence, a *binding-time analysis* (BTA for short) does the following: given a program and an approximation of the input available for specialization, it approximates all values within the program and generates annotations that steer (or control) the specialization process. The partial evaluator (or the compiler generator generating the specialised partial evaluator) then uses the generated annotated program to guide the specialization process. This process is illustrated in Figure 19.

Within the annotated program there are two types of annotations.

- *Filter declarations*: these give each argument of a predicate a *binding-type*. A binding-type indicates something about the structure of an argument, For example, it could indicate which parts of the argument are *dynamic* (possibly unbound at specialization time) and which parts are known (*static*) at specialization time. (Other more precise binding types are possible). These annotations influence the *global control*, in that dynamic parts are generalised away (i.e., replaced by fresh variables) and known parts are kept unchanged.

- *Clause annotations*: these indicate how every call in the body should be treated during specialization, Essentially, the annotations determine whether a call is unfolded at specialization time or at run time. These influence the *local control* [91].

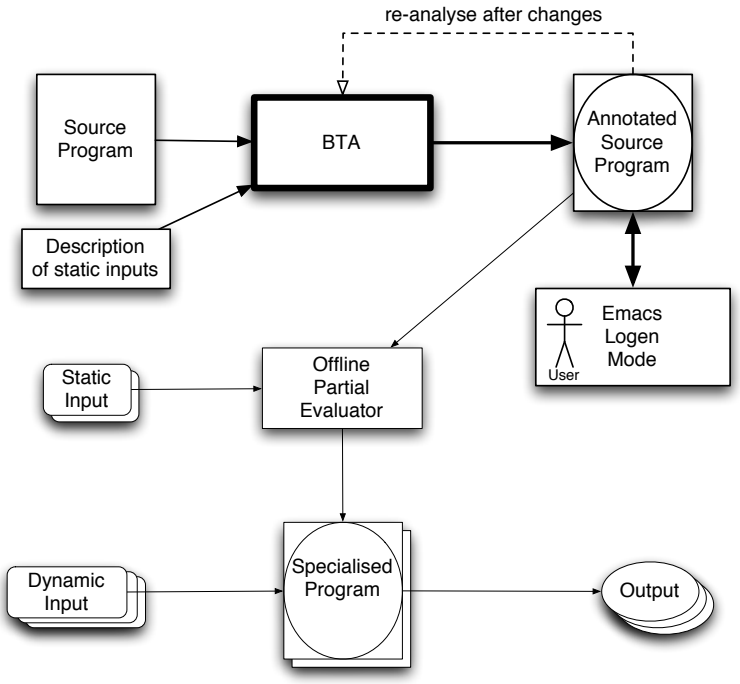



Figure 19: The role of the BTA

In this paper we outline a procedure for generating these annotations automatically. There are several independent components of the procedure. Hence, if one of these components is missing it could be replaced by a manual procedure or oracle, making the whole process semi-automatic.

The input to the procedure is (i) a program to be specialized, (ii) a set of types, and (iii) a goal whose arguments are typed with respect to the given types. Note that the program need not be a typed program in the usual sense. The types simply define sets of terms which form the basis for the binding type annotations, and the types do not have to be associated with particular arguments of predicates.

The independent components that are part of the overall automatic BTA procedure are (i) a type determinization algorithm, (ii) an abstract interpreter over a domain of determinized (disjoint) types, and (iii) a termination analyser, incorporating for example an abstract interpreter over a domain of convex hulls.

# 28 Binding Types

The basis of the BTA is a classification of arguments using abstract values. Abstract values can be chosen in different ways. The BTA procedure described here is independent of the particular choice of abstraction; the only essential aspect of the abstraction is that it must be possible to determine whether an argument is possibly unbound (that is, it is could be a variable). In this case, the argument is called *dynamic*.

The simplest approach is to classify arguments within the program to be specialised as either *static* or *dynamic*. The value of a static argument will be *definitely ground*. A dynamic argument can be either ground or not ground, that is, it can be any term. The use of static-dynamic binding types was introduced for functional programs, and has been used in logic program BTAs [94]. While often sufficient for functional programs, it sometimes proves to be too weak for logic programs: in logic programming partially instantiated data structures appear naturally even at runtime. A simple classification of arguments into "fully known" or "totally unknown" is therefore unsatisfactory and would prevent specializing a lot of "natural" logic programs such as the vanilla metainterpreter [49, 89] or most of the benchmarks from the DPPD library [71].

Other more expressive binding types can be based on lists or other data-types. Using typical notation for defining types, one can define lists (or trees etc.) whose elements are ground, dynamic, numbers, nested structures and so on. These types must be representable as a set of regular types rules. The sets of ground and non-ground terms can also be defined as regular types. Examples are given in [34].

## 28.1 Derivation of Filter Declarations

The given set of types is first transformed into a set of disjoint regular types. This process is called *determinization* and is described in detail in another report [34]. The disjoint types define an abstract domain, which is input, along with the typed goal, to an abstract interpreter over that domain. The details of the abstract interpretation algorithm are also given in [34].

For example, the simple binding types *static* and *dynamic* are determinized to the disjoint types *static* and *non-static*. Note that *dynamic* is the union of *static* and *non-static*.

The output of the abstract interpreter is an assignment of a set of disjoint types to each argument. Let $p/n$ be a predicate in the program. Then the analyser assigns a binding type $p(S_1, \ldots, S_n)$ to $p$, where $S_j$ ($1 \leq j \leq n$) is a set of disjoint types $\{t_{j_1}, \ldots, t_{j_k}\}$. The meaning of the assignment is that, whenever $p/n$ is called during specialization, its $j^{th}$ argument is contained in of the types $\{t_{j_1}, \ldots, t_{j_k}\}$ and is not contained in any type other than $\{t_{j_1}, \ldots, t_{j_k}\}$.

The assignment of types to arguments for a predicate $p$ is called a *filter* for $p$. The filter

is used in the generalization operation dueing partial evalaution. We assume that there is a generalization operation, which takes as input an atom for predicate $p$ and a filter for $p$, and returns another, more general atom for $p$. The LOGEN generalization procedure was modified to handle the disjoint determinized types described above.

# 29 Clause Annotations

Each clause is annotated indicating how every call in the body should be treated during specialization. This is needed to ensure local termination. Some of the basic annotations are:

- *unfold*: for reducible predicates: they will be unfolded during specialization.
- *memo*: for non-reducible predicates: they will be added to the memoisation table and replaced with a generalized residual predicate (using the filter for that predicate).
- *call*: for reducible predicates; they will be completely evaluated during specialization.
- *rescall*: for builtins and imported predicates; the call will be kept and will appear in the final specialised code.

## 29.1 Example Annotation

The simple annotations are demonstrated using the LOGEN annotation syntax:

```
:- filter append(static, dynamic, dynamic).

append([],L,L).
append([H|T],L,[H|T1]) :- logen(unfold, append(T,L,T1).
```

The first argument to append has been marked as `static`; it will be known at specialization time. The other arguments have been marked `dynamic`, so they may or may not be known at specialization time.

The filter declaration above could be derived automatically from the append program, the input types `static` and `dynamic`, and the typed goal `append(static, dynamic, dynamic)`. After determinizing the input types, the filter would be represented in the following equivalent form.

```
:-filter append([static],[static,nonstatic],[static,nonstatic]).
```

In other words, the second and third argument can be either static or non-static, which is the same as dynamic. The fact that a type such as `[static, nonstatic]` is dynamic can easily be detected [34].

Note that the analysis guarantees correct filters. In a manually written filter, it is not guaranteed that calls to $append$ would actually satisfy the given binding types (e.g. that $append$ would always be called with a ground first argument).

Examining the clause annotation in the example, the recursive call to append is annotated for unfolding. The first argument is static and this is sufficient to guarantee termination at specialization time (this will be formally checked by the termination analysis component of our algorithm). If we could not guarantee the termination of the recursive call then it would have to be marked as *memo*.

## 29.2 Derivation of Filter Declarations in the Presence of Clause Annotations

In the derivation of filter declarations mentioned above it was stated that whenever $p/n$ is called during specialization, its $j^{th}$ argument has a value given by one of the types $\{t_{j_1}, \ldots, t_{j_k}\}$. It was not explained what is meant by "whenever $p/n$ is called during specialization". In fact, this depends on the clause annotations, since the decision of whether to memo or call an atom affects the propagation of binding types.

Standard abstract interpretation, which assumes a complete computation, is modified to allow for memo-ing of calls. Memo-ed called are simply ignored when propagating binding types. The same holds for *rescall* annotations. Note that we still derive filters for memo-ed calls: it is only for propagation of binding types that they are ignored.

# 30 Termination Checking Based on Binary Clause Semantics

There are two separate termination requirements during partial evaluation. *Local termination* concerns the avoidance of infinite derivations in which atoms annotated as *unfold* or *call* are selected. *Global termination* concerns the set of calls that are unfolded in the course of the partial evaluation, which should be finite. Finiteness of the set is ensured by a generalization operation driven by the filter declarations.

Both of these termination questions are approached via the *binary clause semantics* [17], a representation of a program's computations that makes possible reasoning about loops, and hence termination analysis. Informally, the binary clause semantics of a program $P$ is the set of all pairs

of atoms (called binary clauses) $p(\bar{X})\theta \leftarrow q(\bar{t})$ such that $p$ is a predicate, $p(\bar{X})$ is a most general atom for $p$, and there is a finite derivation (with leftmost selection rule) $\leftarrow p(\bar{X}), \ldots, \leftarrow (q(\bar{t}), Q)$ with computed answer substitution $\theta$. In other words a call to $p(\bar{X})$ is followed some time later by a call to $q(\bar{t})$, computing a substitution $\theta$. We can thus use this information to reason about the set of calls that follow any initial call (by taking appropriate instances of the binary clauses). In particular, loops are represented by binary clauses with the same predicate occurring in head and body.

The binary semantics is in general infinite, but we can make safe approximations of the set of binary clauses using standard abstract interpretation. The relevant abstraction is based on the size of terms (i.e. we abstract an argument by its size with respect to some measure). We use a domain of convex hulls to abstract the set of binary clauses with respect to a given measure.

Using such an abstraction, we aim to obtain a finite set of binary clauses of form $p(\bar{X}) \leftarrow \pi(\bar{X}, \bar{Y}), q(\bar{Y})$, where $\pi(\bar{X}, \bar{Y})$ is a linear relation between arguments $\bar{X}$ and $\bar{Y}$ which represent the sizes of the respective concrete arguments. Termination proofs (for calls to a predicate $p$) require us to prove that for every abstract binary clause $p(\bar{X}) \leftarrow \pi(\bar{X}, \bar{Y}), p(\bar{Y})$, $\pi(\bar{X}, \bar{Y})$ entails a strict reduction in size for some bounded (rigid) component of $\bar{X}$. Boundedness (rigidity) of the arguments is established by another abstract interpretation very similar to binding type propagation. The details are not discussed here, except to say that a size measure can be derived for each binding type. A term is rigid with respect to a measure if all its instances have the same size.

As we did for for binding type propagation, we need to modify binary clauses to take account of the annotations, since selection of atoms annotated *memo* or *rescall* are not allowed.

Given an annotated program $P$, the binary program is generated by a transformation developed for backwards analysis [35], and modified for annotated programs. The transformation is carried out in LOGEN itself, by partial evaluation of the interpreter shown in Section 32.

## 30.1 Checking For Local Termination

Given the binary clause program, we compute an abstraction using convex hulls (currently we use a convex hull analyser kindly supplied by Samir Genaim and Mike Codish [36]). The specialized binary clause program represents a binary clause $p(\bar{t}) \leftarrow q(\bar{s})$ as $bin\_solve(p(\bar{t}), q(\bar{s}))$. We examine the binary clauses and look for unfold loops, which are abstract binary clauses of the form:

$$bin\_solve(p(s_1, \ldots, s_n), unfold(p(t_1, \ldots, t_n))) \leftarrow C(s_1, \ldots, s_n, t_1, \ldots, t_n)$$

where $C(s_1, \ldots, s_n, t_1, \ldots, t_n)$ is some linear constraint. We then check whether for some argument $j$ there is a decrease from $s_j$ to $t_j$. If so, we check whether the $j^{th}$ argument is rigid with respect to the current filter for $p$. If so, termination is ensured, and otherwise not.

## 30.2 Checking for Global Termination

An annotated program output from the algorithm described above is then examined for global termination properties. Again, an analysis based on binary clauses is used, but this time, we seek abstract binary clauses of the following form:

$$bin\_solve(p(s_1, \ldots, s_n), memo(p(t_1, \ldots, t_n))) \leftarrow C(s_1, \ldots, s_n, t_1, \ldots, t_n)$$

Such clauses represent recursive introduction of memo-ed calls. The binary clause transformation incorporates the generalization operator, as can be seen in the interpreter shown in Section 32, since the actual memo-ed call is produced by the generalization of $p(t_1, \ldots, t_n)$ into $memo(p(t_1, \ldots, t_n))$, with respect to the filter for $p$.

In global termination checking, the conditions on the constraint $C(s_1, \ldots, s_n, t_1, \ldots, t_n)$ are more relaxed compared to local termination conditions. We are willing to generate an infinite number of memo-ed calls, as long as the set of those calls is finite (up to variable renaming). Indeed, if we encounter a memo-ed call that has already been treated before, no new processing will arise. This situation is similar to termination of tabled logic programs and hence the techniques for ensuring *quasi-termination* of [25, 80] form a good starting point. There are in fact two conditions that will ensure global termination:

- Either we can use the same condition as for local termination and require a strict *decrease* in size for some rigid argument.
- or we are satisfied if the arguments are not *increasing* in size, provided that the norms are so-called *finitely partitioning* [25, 80], meaning that only finitely many terms (up to variable renaming) have the same size. For instance, the term size norm is finitely partitioning but the list length is not (e.g., `[0], [s(0)], [s(s(0))],...` all have the same size 1).

Supposing we have finitely partitioning norms, our BTA will, for each binary clause as shown above, seek arguments which increase in size. If such arguments exist, then global termination is not ensured, and we must perform some further generalisation during partial evaluation in order to ensure termination. This can be achieved by changing the filter for the increasing argument to `dynamic`.

# 31 Outline of Algorithm

We now outline the main operations and steps in the algorithm. The core of the algorithm is a loop which propagates the binding types with respect to the current clause annotations, generated the binary program and checks the current filters for termination conditions, and modifies the clause annotations accordingly. Initially, all body atoms are annotated as *unfold* or *call*, except for imported predicates which are annotated *rescall*. Annotations can be changed to *memo* or *rescall*, until termination is established.

The algorithm below does not check for global termination. This is discussed in the next section.

```
Initialization:
   Get program types and determinize them;
   Initialize all clause annotations to
        - unfold for defined user predicates,
        - call for builtins
        - rescall for imported predicates;
   Compute filters w.r.t. current clause annotations
      and initial filter-annotation for the query;

Repeat:
   Finished := True;
   Bin := binary program for local unfolding, w.r.t. current
       clause annotations;
   ConvexBin := convex hull abstraction of Bin;
   For each clause annotation unfold(A)
      Let Filter(A) := the current filter for A;
      If
          Filter(A) does not terminate w.r.t. ConvexBin
      Then
         change unfold(A) to memo(A);  Finished := False;
      EndIf
   EndFor
    For each clause annotation call(A)
      Let Filter(A) := the current filter for A;
      If
          Filter(A) is insufficiently instantiated to call A
      Then
```

```
         change call(A) to rescall(A);  Finished := False;
      EndIf
   EndFor
   Compute new filters w.r.t. current clause annotations;
   If
      global non-termination is indicated in ConvexBin
   Then
      change the increasing arguments in the filters
         for memo-ed predicates to dynamic

Until Finished;

Return current filters and current clause annotations;
```

Termination of the main loop is ensured since there is initially only a finite number of *unfold* or *call* clause annotations, and each iteration of the loop eliminates one or more *unfold* or *call* annotations.

# 32 Generation of the Binary Clause Semantics

To determine the binary clause semantics needed for the termination analysis, we can actually leverage our ASAP tools and obtain it by specializing an interpreter (called `Bin_solve`) using LOGEN. `Bin_solve` is a binary clause interpreter based on the simple vanilla interpreter solve. Specializing bin_solve with respect to the filter goals and annotated programs we wish to analyse produces a program whose semantics is the binary clause semantics for either local unfolding or memoed calls.

```
:- filter solve(type(list(nonvar))):solve.
bin_solve([unfold(H)|_T],unfold(H)).
bin_solve([memo(H)|_T],memo(H1)) :- filter(H.,F), generalise(H,F,H1).
bin_solve([unfold(H)|_T],RecCall) :-
        bin_solve_atom(H,RecCall).
bin_solve([unfold(H)|T],RecCall) :-
        solve_atom(H),
        bin_solve(T,RecCall).
bin_solve([memo(_)|T],RecCall) :-
        bin_solve(T,RecCall).
```

```
:- filter bin_solve_atom(nonvar_nf,dynamic).
bin_solve_atom(H,Rec) :-
            rule(H,Bdy),
                bin_solve(Bdy,Rec).

:- filter test(dynamic,dynamic).
test(H,Rec) :-
                filtered(H),
                bin_solve([unfold(H)],Rec).

filtered(ann_dapp(_,_,_,_)).
filtered(ann_app(_,_,_)).
:- module('~/cvs_root/cogen2/logen_examples/bta_test/bin_solve.pl.memo',
          [test__0/2,bin_solve_atom__1/2,solve_atom__2/4,
                     bin_solve_atom__3/2,solve_atom__4/3]).
gensym(num__num,5).
memo_table(0,test(A,B),test__0(A,B),done(user),[]).
memo_table(1,bin_solve_atom(ann_dapp(A,B,C,D),E),
             bin_solve_atom__1(ann_dapp(A,B,C,D),E),
             done(internal),[]).
memo_table(2,solve_atom(ann_dapp(A,B,C,D)),
             solve_atom__2(A,B,C,D),
             done(internal),[]).
memo_table(3,bin_solve_atom(ann_app(A,B,C),D),
             bin_solve_atom__3(ann_app(A,B,C),D),
             done(internal),[]).
memo_table(4,solve_atom(ann_app(A,B,C)),
             solve_atom__4(A,B,C),
             done(internal),[]).

test__0(ann_dapp(B,C,D,E),ann_dapp(B,C,D,E)).
test__0(ann_dapp(B,C,D,E),F) :-
  bin_solve_atom__1(ann_dapp(B,C,D,E),F).
test__0(ann_app(B,C,D),ann_app(B,C,D)).
test__0(ann_app(B,C,D),E) :-
  bin_solve_atom__3(ann_app(B,C,D),E).
bin_solve_atom__1(ann_dapp(B,C,D,E),ann_app(C,D,F)).
```

```
bin_solve_atom__1(ann_dapp(B,C,D,E),F) :-
  bin_solve_atom__3(ann_app(C,D,G),F).
bin_solve_atom__1(ann_dapp(B,C,D,E),memo(app(B,F,E))) :-
  solve_atom__4(C,D,F).
solve_atom__2(B,C,D,E) :-
  solve_atom__4(C,D,F).
bin_solve_atom__3(ann_app([B|C],D,[B|E]),memo(ann_app(C,D,E))).
solve_atom__4([],B,B).
solve_atom__4([B|C],D,[B|E]).
```

# 33 Worked Example

We demonstrate the algorithm using a worked example of a pattern matching program.

The `match` predicate identifies a pattern `Pat` in a string `T`.

```
match(Pat,T) :-
        match1(Pat,T,Pat,T).
match1([],Ts,P,T).
match1([A|Ps],[B|Ts],P,[X|T]) :-
        A\==B,
        match1(P,T,P,T).
match1([A|Ps],[A|Ts],P,T) :-
        match1(Ps,Ts,P,T).
```

Initialise the annotations to all unfold and call.

```
:- filter match(list, dynamic).
match(Pat,T) :-
        unfold(match1(Pat,T,Pat,T)).
match1([],Ts,P,T).
match1([A|Ps],[B|Ts],P,[X|T]) :-
        call(A\==B),
        unfold(match1(P,T,P,T)).
match1([A|Ps],[A|Ts],P,T) :-
        unfold(match1(Ps,Ts,P,T)).
```

Specializing through bin_solve to obtain the abstract clause semantics:

```
bin_solve_atom__0(match(B,C),unfold(match1(B,C,B,C))).
bin_solve_atom__1(match1([B|C],[D|E],F,[G|H]),unfold(match1(F,H,F,H))).
bin_solve_atom__1(match1([B|C],[B|D],E,F),unfold(match1(C,D,E,F))).
```

Abstracting the programming with respect to list length norm

```
:- filter match(list,dynamic).
:- filter match1(list,dynamic,list,dynamic).
bin_solve_atom__0(match(B,_),unfold(match1(B,_,B,_))).
bin_solve_atom__1(match1(1+C,_,F,_),unfold(match1(F,_,F,_))).
bin_solve_atom__1(match1(1+C,_,E,_),unfold(match1(C,_,E,_))).
```

For each binary clause of the form: `bin_solve_atom(p(--),unfold(p(--))`:

```
bin_solve_atom__1(match1(1+C,_,F,_),unfold(match1(F,_,F,_))).
```

Argument 1 does not show decrease 1+C → F
Argument 3 does not show decrease F → F
Offending call must be marked as memo.

```
:- filter match(list, dynamic).
match(Pat,T) :-
        unfold(match1(Pat,T,Pat,T)).
match1([],Ts,P,T).
match1([A|Ps],[B|Ts],P,[X|T]) :-
        call(A\==B),
        memo(match1(P,T,P,T)).
match1([A|Ps],[A|Ts],P,T) :-
        unfold(match1(Ps,Ts,P,T)).

bin_solve_atom__0(match(B,C),unfold(match1(B,C,B,C))).
bin_solve_atom__1(match1([B|C],[B|D],E,F),unfold(match1(C,D,E,F))).
```

Abstract program again . . .

```
bin_solve_atom__0(match(B,C),unfold(match1(B,C,B,C))).
bin_solve_atom__1(match1(1 +C,_,E,_),unfold(match1(C,_,E,_))).
```

For each binary clause of the form: `bin_solve_atom(p(--),unfold(p(--))`:

```
bin_solve_atom__1(match1(1 +C,_,E,_),unfold(match1(C,_,E,_))).
```

Argument 1 does show decrease 1+C → C, and is rigid with respect to the size measure.
Hence, local termination is ensured.

# Part VI

# A Higher-Order Binding-Time Analysis for Mercury

In this work, we develop a binding-time analysis for the logic programming language Mercury. We introduce a precise domain of binding-times, based on the type information available in Mercury programs, that allows to represent partially static data structures for specialisation. The analysis is polyvariant, and deals with the module structure and higher-order capabilities of Mercury programs.

## 34 Introduction

Program specialisation is a technique that transforms a program into another program, by precomputing some of its operations. Assume we have a program $P$ of which the input can be divided in two parts, say $s$ and $d$. If one of the input parts, say $s$, is known at some point in the computation, we can *specialise* $P$ with respect to the available input $s$. This specialisation process comprises performing those computations of $P$ that depend only on $s$, and recording their *results* in a new program, together with the *code* for those computations that could not be performed (because they rely on the input part $d$ – unknown at this point in the computation). The result of the specialisation is a new program, $P_s$ that computes, when provided with the remaining input part $d$, the *same* result as $P$ does when provided with the complete input $s + d$. Comprising a mixture of program evaluation and code generation, the program specialisation process is also often referred to by the names *partial evaluation*, *mixed computation* or *staged computation*.

Staging the computations of a program can be useful (usually in terms of efficiency) when different parts of a program's input become known at different times during the computation. The best benefit can be obtained when a single program must be run a number of times while a part of its input remains constant over the different runs. In this case, the program can first be specialised with respect to the constant part of the input, while afterwards the resulting program can be run a number of times, once for each of the remaining (different) input parts. In such a staged approach, the computations that depend only on the constant input part are performed only once – during specialisation. In the non-staged approach, *all* computations – including those depending on the constant part – are performed over and over again in each run of the program.

When using program specialisation to stage the computations of a program, the basic problem is deciding what computations can be safely performed during the specialisation process. The driving force behind this decision is twofold. Firstly, the specialisation process itself must terminate; that is, the specialiser may not to get into a loop when evaluating a sequence of computations from the program that is to be specialised. Secondly, the obtained degree of specialisation should be "as good as possible", meaning that a fair amount of computations that *can* be performed during specialisation *are* effectively performed during specialisation.

The key factor determining whether a computation can be performed during specialisation is the fact whether enough input values are available to compute a result. If that is the case, the specialiser can perform the computation; if not, it should generate code to perform this computation at a later stage. Binding-time analysis is a static analysis that, given the program and a description about the available partial input with respect to which the program will be specialised, computes for every statement in the program what input values will be known when that statement is reached during specialisation. In addition, the analysis computes — according to some control strategy — whether or not the statement should be evaluated during specialisation.

Once the program $P$ and its available partial input $s$ has been analysed by binding-time analysis, specialisation of $P$ with respect to $s$ boils down to evaluating those statements in $P$ that are annotated as such by the binding-time analysis. This specialisation technique is called *off-line*, the reason being that most of the control decisions have been taken by the binding-time analysis. This in contrast with the so-called *on-line* specialisation technique in which the program to be specialised is not analysed by any binding-time analysis, but is directly evaluated with respect to its partial input under the supervision of a control system that decides – for every statement under consideration – on the fly whether or not it can safely be evaluated. Both approaches towards specialisation have their advantages and disadvantages. In this work, we concentrate on *off-line* specialisation and construct a binding-time analysis for the logic programming language Mercury.

## 34.1 Binding-time Analysis and Logic Programming

Using binding-time analysis to control the behaviour of the specialisation has been thoroughly investigated in a number of programming paradigms. Breaking work on off-line program specialisation of imperative languages include C-mix by Andersen [2] and more recently Tempo [19, 53] by Consel and his group. Also in the context of functional language specialisation, most work focusses on binding-time analysis and off-line specialisation, originally motivated to achieve better self-application [31, 56]. Whereas initial analysis dealt with first-order languages [56], more recently developed analyses deal with higher-order aspects [42, 10], polymorphism

[93, 48] and partially static data structures [69].

In the field of logic programming, however, only little attention has been paid to off-line program specialisation. Known exceptions are LOGIMIX [95] and LOGEN [64] that develop different approaches to off-line program specialisation for Prolog. Both cited works, however, lack an automatic binding-time analysis and rely on the user to provide the specialiser with suitable annotations of the program. To the best of our knowledge, the only attempt to construct an automatic binding-time analysis for logic programming is [14] and our own work about which we report in [77]. The approach of [14] is particular, in the sense that it obtains the required annotations not by analysing the subject program directly but rather by analysing the behaviour of an *on-line* program specialiser on the subject program. Although conceptually interesting, the latter approach is overly conservative and restricts the number of computations that can be performed during specialisation. Indeed, [14] decides whether to unfold a call or not based on the original program, not taking current annotations into account. This means that call can either be completely unfolded or not at all. The binding-time analysis first described in [120] and employed in [77] is also particular in the sense that it obtains its annotations by repeatedly appyling an automatic termination analysis. If the termination analysis identifies a call as possibly non-terminating, that call is marked such that it will not be reduced by the specialiser. Then the termination analysis is rerun to prove termination of the program under the assumption that each call that is marked as non-reducible is not evaluated. The process is repeated until termination of the (annotated) program can be proven.

Both the approach of [14] and [77] have been designed towards dealing with untyped and unmoded logic programming languages. The fact that most logic programming languages are untyped makes it hard – if not impossible – to represent the availability of *partial* input in a sufficiently precise way during the analysis. More importantly, the lack of control flow information in the program makes it nearly impossible to approximate the data flow in a sufficiently precise way and renders the derivation of a binding-time analysis by ”classic” abstract interpretation techniques not straightforward, hence the approaches of [14] and [77]. In this work, we construct a completely automatic binding-time analysis for the recently introduced logic programming language Mercury. Being a strongly typed and moded language, it lifts the obstacles encountered with more traditional logic programming languages and allows to construct a “traditional” binding-time analysis along the lines of [42, 60] based on data flow analysis. Yet, its more involved data- and control flow features – inherent to a logic programming language – render the derivation of an automatic binding-time analysis a daunting and not straightforward task.

## 34.2 Mercury

The design of Mercury was started in October 1993 by researchers at the University of Melbourne. While logic programming languages had been around for quite some time, no one seemed to fully realise the theoretical advantages such a language would have over more traditional, imperative languages. These advantages are widely known, and are summarised for example in [111]: a higher level of expressivity (enabling the programmer to concentrate on *what* has to be done rather than on *how* to do it), the availability of a useful formal semantics (required for the – relatively – straightforward design of analysis and transformation tools), a semantics that is independent of any order of evaluation (useful for parallelising the code), and a potential for declarative debugging [82]. While a language like Prolog does offer some of these advantages, others are destroyed by the impure features of the language.

The main objective of the Mercury designers was to create a logic programming language that would be *pure* and useful for the implementation of a large number of *real-world* applications. To achieve this goal, the main design objectives of Mercury can be summarised as follows [111]: *Support for the creation of reliable programs*. This involves a language that allows to detect some classes of bugs at compile-time. *Support for programming in teams*. Large software systems are usually build by a number of programmers. The language must provide good support for creating a single application from multiple parts that are build (sometimes in isolation) by different programmers. These two objectives form a major departure from Prolog which, at the time, had basically no support for programming in the large, and which does not allow a lot of type-, mode- and determinism errors to be caught at compile-time. Another important objective was *support for the creation of efficient programs*. The efficiency of the language implementation had to be at least comparative with (but preferably better than) comparable languages.

To meet these design objectives, Mercury was fitted with a strong system of type-, mode- and determinism declarations. Apart from providing excellent comments on how the data used in a predicate should look and how the code is supposed to be used, these declarations enable the compiler to perform a number of analyses and to spot a substantial number of bugs at compile time, rather than producing a program that shows some unexpected behaviour at run-time as is often the case with Prolog. Also, the availability of declarations allows to fix the evaluation order of the body atoms in a predicate and provides as such the basis for an efficient execution mechanism of the language [20, 110, 112]. Mercury is equipped with a modern module system that enables to hide some data definitions and to encapsulate both data and code, and provides as such support for programming-in-the-large activities.

# 35 A Domain of Binding-times

Binding-time analysis can be seen as an application of abstract interpretation over a domain of *binding-times*. A binding-time abstracts a value by specifying at what time during a 2-stage computation[12] the value becomes known. In their most basic form, the binding-time of a value is either *static* or *dynamic*, denoting a value that is known early, during specialisation, or late, during evaluation of the residual program, respectively.

It is recognised [60] that for a logic programming language, approximating values by either *static* or *dynamic* is too coarse grained in general. Indeed, most logic programs use a lot of *structured* data, where data values are represented by structured terms. Consequently, the input to the specialiser usually consists of a partially instantiated term: a term that is less instantiated than it would be at run-time. Approximating a partially instantiated term by *dynamic* usually results in too much information loss, possibly resulting in missed specialisation opportunities. Therefore, we use the structural information from the type system of Mercury to represent more detailed binding-times, capable of distinguishing between the computation stages in which *parts* of a value (according to that value's type) become known.

Mercury's type system is based on a polymorphic many-sorted logic, and corresponds to the Mycroft-O'Keefe type system [97]. Basically, the types are discriminated union types and support parametric polymorphism: a type definition can be parametrised with some type variables, as the following example in Mercury syntax shows.

**Example 1** `:- type list(T) ---> [] ; [T | list(T)].`

The above defines a polymorphic type `list(T)`: it defines values of this type to be terms that are either `[]` (the empty list) or of the form `[A|B]` where `A` is a value of type `T` and `B` is a value of type `list(T)`.

Formally, if we denote with $\Sigma_{\mathcal{T}}$ the set of type constructors and with $V_{\mathcal{T}}$ the set of type variables of a language $\mathcal{L}$, the set of *types* associated to $\mathcal{L}$ is represented by $\mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$; that is the set of terms that can be constructed from $\Sigma_{\mathcal{T}}$ and $V_{\mathcal{T}}$. A type containing variables is said to be *polymorphic*, otherwise it is a *monomorphic* type. A *type substitution* is a substitution from type variables to types. The application of a type substitution to a polymorphic type results in a new type, which is an *instance* of the original type.

As usual, the set of program values is denoted by $\mathcal{T}(\mathcal{V}, \Sigma)$; that is the set of terms that can be constructed from a set $\Sigma$ of function symbols and a set $\mathcal{V}$ of program variables.

[12] Generalisations exist in which computations are staged over more than 2 stages (see e.g. [41]). In this work, we focus on a traditional 2-stage process, dividing the computations in a program over *specialisation-time* versus *run-time*.

The relation between a type and the values (terms) that constitute the type is made explicit by a *type definition* that consists of a number of *type rules*, one for every type constructor. Example 1 shows the type rule associated to the `list/1` type constructor. Formally, a type rule is defined as follows:

**Definition 35.1** The *type rule* associated to a type constructor $h/n \in \Sigma_{\mathcal{T}}$ is a definition of the form

$$h(\overline{T}) \rightarrow f_1(\overline{\tau}_1) \; ; \; \ldots \; ; \; f_k(\overline{\tau}_k).$$

where $\overline{T}$ is a sequence of $n$ type variables from $V_{\mathcal{T}}$ and for $1 \leq i \leq k$, $f_i/m \in \Sigma$ with $\overline{\tau}_i$ a sequence of $m$ types from $\mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$ and all of the type variables occurring in the right hand side occur in the left hand side as well. The function symbols $\{f_1, \ldots, f_k\}$ are said to be associated with the type constructor $h$. A finite set of type rules is called a type definition.

Given a type substitution, we define the notion of an instance of a type rule in a straightforward way. In theory, every type (constructor) can be defined by a type rule as above. In practice, however, it is useful to have some types builtin in the system. For Mercury, the types `int`, `float`, `char`, `string` are builtin types whose denotation is predefined and is the set of integers, floating point numbers, characters and strings respectively.

Mercury is a statically typed language, in which the (possibly polymorphic) type of every term occurring in the program text is known at compile-time. In what follows, we use the type definition to construct, for every type occurring in the program, a finite description of the *structure* that values belonging to the denotation of a particular type can take. The relevance of such a description is in the fact that it can be used to abstract the values belonging to the denotation of the type according to their structure, allowing the construction of a precise abstract domain for program analysis, in particular binding-time analysis.

To extract a structural description of a type from a type definition, we introduce the notion of a type-path being a sequence of functor/argument position pairs that is meant to denote a path through the type definition from a type to an occurrence of one of its subtypes. In fact, a type itself can be represented as a (possibly infinite) set of such paths, one for every path from the type that is being defined to some subtype occurring at a particular position within some term belonging to the denotation of that type. More formally, we denote the set of all such sequences over $\Sigma \times \mathbb{N}$ by *TPath*. The empty sequence is denoted by $\langle\rangle$, and given $\delta, \epsilon \in$ *TPath*, we denote with $\delta \bullet \epsilon$ the sequence obtained by concatenating $\epsilon$ to $\delta$. A *type tree* for a particular type can then be defined as follows:

**Definition 35.2** Given a type $\tau \in \mathcal{T}(\Sigma_{\mathcal{T}}, V_{\mathcal{T}})$, the *type tree* of $\tau$, denoted by $\mathcal{L}_\tau$, is a set of sequences from *TPath* and is recursively defined as:

- $\langle\rangle \in \mathcal{L}_\tau$
- if $t = h(\overline{T})\theta$ with $h(\overline{T}) \rightarrow f_1(\overline{\tau}_1); \ldots; f_k(\overline{\tau}_k)$ a type rule and $\theta$ a type substitution, then for all $i \in \{1 \ldots k\}$, and if $f_i/m \in \Sigma$, for each $j \in \{1 \ldots m\}$, $\langle(f_i, j)\rangle \bullet \delta \in \mathcal{L}_\tau$ where $\delta \in \mathcal{L}_{(\tau_{i_j})\theta}$ and $\tau_{i_j}$ denotes the $j$-th type in $\overline{\tau}_i$.

**Example 2** *Reconsider the type $list(T)$ from Example 1. The type tree of $list(T)$ is the infinte set of type paths*

$$\mathcal{L}_{list(T)} = \left\{ \begin{array}{l} \langle\rangle \\ \langle([],1)\rangle \\ \langle([],2)\rangle \\ \langle([],2),([],1)\rangle \\ \langle([],2),([],2)\rangle \\ \langle([],2),([],2),([],1)\rangle \\ \langle([],2),([],2),([],2)\rangle \\ \langle([],2),([],2),([],2),([],1)\rangle \\ \ldots \end{array} \right\}$$

The general idea now is to define, for any type $\tau$, a finite approximation of $\mathcal{L}_\tau$ that provides a good characterisation of the structure of terms of type $\tau$. First we introduce the following notation that formally defines the type that is identified by a type-path within another type.

**Definition 35.3** Let $\tau$ be a type of the form $\tau = h(\overline{T})\theta$, $h \in \Sigma_\mathcal{T}$ defined by $h(\overline{T}) \rightarrow f_1(\tau_{1_1}, \ldots, \tau_{1_{k_1}}); \ldots; f_n(\tau_{n_1}, \ldots$ and $\delta \in$ *TPath* of the form $\delta = \langle(f, i)\rangle \bullet \epsilon$. Then we have that $\tau^\delta = \tau^\epsilon_{j_i}$ if $f = f_j$ for some $j$. Moreover, $\tau^{\langle\rangle} = \tau$ for any type $\tau$.

Note that for $\tau \in \mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$ and $\delta \in$ *TPath*, $\tau^\delta$ is only defined in case $\delta \in \mathcal{L}_\tau$. Also note that a type path $\delta \in \mathcal{L}_\tau$ can also be used to identify a particular subterm in a term $t : \tau$, if it exists. Indeed, if $\delta \in$ *TPath* is of the form $\delta = \langle(f, i)\rangle \bullet \epsilon$ and $t = f(t_1, \ldots, t_n)$ we define $t^\delta = t^\epsilon_i$.

**Example 3** *If $\tau = list(T)$ we have for example that*

$$\tau^{\langle\rangle} = list(T), \tau^{\langle([],1)\rangle} = T \textit{ and } \tau^{\langle([],2)\rangle} = \tau^{\langle([],2),([],2)\rangle} = list(T).$$

*Similarily for a term $t = [1, 2]$ we have for example that*

$$t^{\langle\rangle} = [1, 2], t^{\langle([],1)\rangle} = 1 \textit{ and } t^{\langle([],2)([],1)\rangle} = 2.$$

Given the definition of a type tree, we introduce the following equivalence relation on the paths in a type tree $\mathcal{L}_\tau$. We define $\equiv$ (in $\mathcal{L}_\tau$) as the least transitive relation such that for any $\delta, \alpha \in \mathcal{L}_\tau$: if $\delta = \alpha \bullet \epsilon$ and $\tau^\delta = \tau^\alpha$ then $\alpha \equiv \delta$. Informally, two type paths in a type tree are equivalent if either one of the paths is an extension of the other while both identify the same type, or the paths share a common initial subpath that identifies the same type as both paths in $\mathcal{L}_\tau$. In what follows, we restrict our attention to (possibly polymorphic) types that are not defined in terms of a strict instance of itself. That is, we assume for any type $\tau$ and $\delta \in \mathcal{L}_\tau$ that $\tau \not< \tau^\delta$ (where $<$ denotes the strict instance relation). This is a natural condition and is related to the polymorphism discipline of definitional genericity [68]. For any such type $\tau$, the equivalence relation $\equiv$ partitions the (possibly infinite set) $\mathcal{L}_\tau$ into a finite number of equivalence classes. For any $\delta \in \mathcal{L}_\tau$, the equivalence class of $\delta$ is defined as

$$[\delta] = \{\gamma \in \mathcal{L}_\tau \mid \delta \equiv \gamma\}.$$

The least element of an equivalence class $[\delta]$ exists and is defined as follows.

$$\overline{[\delta]} = \alpha \in [\delta] \text{ such that } \forall \beta \in [\delta] : \beta = \alpha \bullet \epsilon \text{ for some } \epsilon \in \textit{TPath}$$

Next, we define, for a type $\tau$, its *type graph* as the finite set of minimal elements of the equivalence classes of $\mathcal{L}_\tau$:

**Definition 35.4** For a type $\tau \in \mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$, we denote $\tau$'s type graph by $\mathcal{L}_\tau^{\equiv}$ which is defined as

$$\mathcal{L}_\tau^{\equiv} = \{\overline{[\delta]} \mid \delta \in \mathcal{L}_\tau\}.$$

A type graph $\mathcal{L}_\tau^{\equiv}$ provides a finite approximation of the structure of terms of type $\tau$: every path in $\mathcal{L}_\tau^{\equiv}$ abstracts a number of subterms of the term according to their type ánd position in the term. For the $list(T)$ type from above, $\mathcal{L}_{list(T)}^{\equiv} = \{\langle\rangle, \langle([], 1)\rangle\}$. The path $\langle\rangle$ represents all subterms of type $list(T)$ in a term of type $list(T)$, whereas $\langle([], 1)\rangle$ represents all subterms of type $T$ occurring in the first argument position of a functor $[|]$. In other words, $\langle\rangle$ can be seen as identifying the skeleton of the list, whereas $\langle([], 1)\rangle$ as identifying the elements of the list. Note that due to the particular definition of $\equiv$, two subterms of a same type are not necessarily abstracted by the same node in $\mathcal{L}_\tau^{\equiv}$. This is the case when $\mathcal{L}_\tau$ contains two type paths identifying the same type without them being equivalent, as in the next example.

**Example 4** *Consider the type $pair(T)$ defined as*

$$pair(T) \longrightarrow (T - T).$$

*A term of the type $pair(T)$ is a term $(A - B)$ where $A$ and $B$ are terms of type $T$. For $\tau = pair(T)$,*

$$typetree_\tau = \mathcal{L}_\tau^{\equiv} = \left\{ \begin{array}{c} \langle\rangle \\ \langle(-), 1\rangle \\ \langle(-), 2\rangle \end{array} \right\}$$

*Although $\langle(-), 1\rangle$ and $\langle(-), 2\rangle$ identify subterms of the same type $T$, they are not equivalent according to the definition of equivalence.*

The ability to distinguish between two occurrences of the same type in $\mathcal{L}_\tau^{\equiv}$ allows a more precise characterisation of terms of type $\tau$. This is illustrated with Example 4, in which it allows to distinguish between both elements of a pair. Type based analyses [121, 15, 66] have a coarser granularity. All paths leading to nodes of the type tree of the same type are placed in the same equivalence class.

Now, one can obtain an abstract characterisation of terms of type $\tau$, based on the structure of the term (or at least the type it belongs to), by associating an abstract value to each of the paths in $\mathcal{L}_\tau^{\equiv}$. For binding-time analysis, we are interested in the time a (part of a) value becomes known in the computation process. We use the abstract values $\mathcal{B} = \{\textit{static}, \textit{dynamic}\}$. $static$ denotes that the binding certainly occurs at specialisation time; $dynamic$ that it is not known when (and in case of logic programs "if") the binding occurs. A binding-time associates a value from $\mathcal{B}$ to each of the paths in a type graph.

**Definition 35.5** A *binding-time* for a type $t \in \mathcal{T}(\Sigma_\mathcal{T}, V_\mathcal{T})$ is a function

$$\beta : \mathcal{L}_t^{\equiv} \mapsto \mathcal{B}$$

such that $\forall \delta \in dom(\beta)$ holds that $\beta(\delta) =$ *dynamic* implies that $\beta(\delta') =$ *dynamic* for all $\delta' \in dom(\beta)$ with $\delta' = \delta \bullet \epsilon$ for some $\epsilon \in$ *TPath*. The set of all binding-times (independent of the type) is denoted by $\mathcal{BT}$.

The relation between terms and the binding-times that approximate them is given by the following abstraction function.

**Definition 35.6** The *binding-time abstraction* is a function $\alpha : \mathcal{T}(\Sigma, \mathcal{V}) \mapsto \mathcal{BT}$ and is defined as follows:

$$\alpha(t : \tau) = \left\{ (\delta, v) \;\middle|\; \begin{array}{l} \delta \in \mathcal{L}_\tau^{\equiv} \text{ and } v = \textit{dynamic} \text{ if } \exists\theta \text{ and a subterm } t^{\delta'} \text{ in } t\theta \\ \text{such that } t^{\delta'} \text{ is a variable and } \delta \equiv \delta' \\ v = \textit{static} \text{ otherwise} \end{array} \right\}$$

If a term $t : \tau$ contains a subterm $t^{\delta'}$ that is a variable, then the binding-time abstraction associates the value *dynamic* to the path in $\mathcal{L}_\tau^{\equiv}$ that identifies this subterm and to all its extensions in $\mathcal{L}_\tau^{\equiv}$.

**Example 5** *Given the following terms of type $list(T)$ as defined in Example 1, their binding-time abstraction is:*

$$\begin{array}{lcl} \alpha([]) & = & \{(\langle\rangle, \textit{static}), (\langle([],1)\rangle, \textit{static})\} \\ \alpha([X_1, X_2] & = & \{(\langle\rangle, \textit{static}), (\langle([],1)\rangle, \textit{dynamic})\} \\ \alpha(X) & = & \{(\langle\rangle, \textit{dynamic}), (\langle([],1)\rangle, \textit{dynamic})\} \\ \alpha([X|Y]) & = & \{(\langle\rangle, \textit{dynamic}), (\langle([],1)\rangle, \textit{dynamic})\} \end{array}$$

*Since the term [] does not contain any variable, it is abstracted by a binding-time specifying that the list's skeleton as well as its elements are static. A term $[X_1, X_2]$ is approximated by a binding-time specifying that the list's skeleton is static, but its elements are dynamic. A variable is abstracted by a binding-time specifying that the list's skeleton as well as its elements are dynamic. Also a term $[X|Y]$ is approximated by a binding-time stating that its list skeleton as well as its elements are dynamic due to the presence of the variable subterm $Y : list(T)$.*

The following example shows why, if the value *dynamic* is associated to a path $\delta$ in a binding-time for a type $\tau$, *dynamic* is also associated to all extensions of $\delta$ in $\mathcal{L}_\tau^{\equiv}$.

**Example 6** *Consider a type definition for a tree of integers:*

```
inttree ---> nil ; t(int, inttree, inttree).
```

*The type graph of $\tau =$ inttree, $\mathcal{L}_\tau^{\equiv}$ contains only two paths: $\langle\rangle$ denoting the tree's skeleton, and $\langle t, 1\rangle$ denoting the integer elements in the tree. We have*

$$\alpha(t(0, X, t(1, \textit{nil}, \textit{nil}))) = \{(\langle\rangle, \textit{dynamic}), (\langle t, 1\rangle, \textit{dynamic})\}.$$

*Although all subterms of type int in the term $t(0, X, t(1, \textit{nil}, \textit{nil}))$ are non-variable terms, we cannot abstract them to static. Indeed, the variable $X$ in the term, being of type inttree, possibly represents some unknown integer elements.*

To make our approximations suitable for a binding-time analysis, we define a partial order relation on $\mathcal{BT}$:

**Definition 35.7** Let $\beta, \beta' \in \mathcal{BT}$ such that $dom(\beta) \subseteq dom(\beta')$ or $dom(\beta') \subseteq dom(\beta)$. We say that $\beta$ *covers* $\beta'$, denoted by $\beta \succeq \beta'$ if and only if $\forall \delta \in dom(\beta) \cap dom(\beta')$ holds that $\beta'(\delta) =$ *dynamic* implies $\beta(\delta) =$ *dynamic*.

If a binding-time $\beta$ covers another binding-time $\beta'$, then $\beta$ is "at least as dynamic" as $\beta'$. Note that the relationship between $dom(\beta)$ and $dom(\beta')$ implies that the *covers* relation is only defined between two binding-times that are derived from types $\tau$ and $\tau'$ such that either $\tau$ is an instance of $\tau'$ or $\tau'$ is an instance of $\tau$.

**Example 7** *Recall the binding-times obtained by abstracting the terms in Example 5. We have that*

$$\alpha(X) \succeq \alpha([X_1, X_2]) \succeq \alpha([])$$

In what follows, we extend the notion of the $\succeq$ relation to include the elements $\{\top, \bot\}$ such that $\top \succeq \beta$ and $\beta \succeq \bot$ for all $\beta \in \mathcal{BT}$. If we denote with $\mathcal{BT}^+$ the set $\mathcal{BT}^+ = \mathcal{BT} \cup \{\top, \bot\}$, $(\mathcal{BT}^+, \succeq)$ forms a complete lattice. Wherever appropriate, we use $\bot$ and $\top$ to denote, for a particular type, a binding-time in which all paths are mapped to *static*, respectively a binding-time in which all paths are mapped to *dynamic*. Occasionally we will also call such binding-times completely static and completely dynamic, respectively.

We conclude this section by introducing some more notation. First, if $\beta$ denotes a binding-time for a type $\tau$ and $\delta \in dom(\beta)$, then $\beta^\delta$ denotes the binding-time for a type $\tau^\delta$ that is obtained as follows:

$$\beta^\delta = \left\{ \; (\gamma, \beta([\overline{\delta \bullet \gamma}])) \;\middle|\; \gamma \in \mathcal{L}^{\equiv}_{\tau^\delta} \; \right\}.$$

In other words, if $\beta = \alpha(t)$ then $\beta^\delta = \alpha(t^\delta)$. Finally, let $\tau, \tau_1, \ldots, \tau_n$ be types and $f \in \Sigma$ such that $f(t_1 : \tau_1, \ldots, t_n : \tau_n)$ is a term in the denotation of $\tau$. If $\beta_1, \ldots, \beta_n$ are binding-times for the types $\tau_1, \ldots, \tau_n$, we denote with $f(\beta_1, \ldots, \beta_n)$ the *least dynamic* binding-time for type $\tau$ such that $\beta^{[\overline{(f,i)}]} \succeq \beta_i$ for all $i$.

# 36 A Modular Binding-time Analysis for Mercury

In what follows, we develop a polyvariant binding-time analysis. The final output of the analysis is an annotated program in which each of the original procedures may occur in several annotated versions, depending on the binding-times of the (input) arguments with respect to which the procedure was called. Each such version contains the binding-times of the local variables and output arguments as well as instructions stating for each subgoal of the procedure's body whether or not it should be evaluated during specialisation. Correctness of the analysis ensures that if a particular call $p(t_1, \ldots, t_n)$ occurs during specialisation, the analysis has created a version of the called procedure that is annotated with respect to the particular call's binding-time abstraction $p(\alpha(t_1), \ldots, \alpha(t_n))$. Before we define the actual analysis, we introduce Mercury's module system and define some necessary machinery to base the analysis upon.

## 36.1 Mercury's module system

A Mercury program is defined as a set of Mercury modules. The basic module system of Mercury is simple. A module consists of an *interface* part and an *implementation* part. The interface part contains those type definitions and procedure declarations that the module provides (or *exports*) towards other modules. In other words, the types and procedures declared in the interface part of a module are visible and can be used (or *imported*) by other modules. Apart from the implementation of the procedures that are declared in the module's interface, its implementation part possibly contains additional type definitions and the declaration and implementation of additional procedures. These types and procedures are only visible in the implementation part of this module, and can not be used by other modules.

Note that the way in which the modules import each other impose a hierarchy on the modules that constitute a program. Following the terminology of [103], we use the notation *imports*$(M, M')$ to indicate that the module $M$ imports the interface of $M'$ and *imported*$(M)$ to denote the set of modules that are imported by $M$, that is: *imported*$(M) = \{M' \mid$ *imports*$(M, M')\}$. Figure 20 shows an example of a module hierarchy in Mercury in which we graphically represent a module by a box, and denote *imports*$(M, M')$ by an arrow from $M$ towards $M'$. In the example, we have

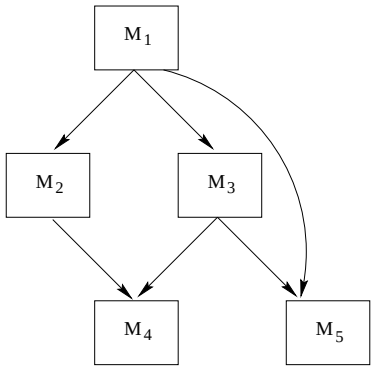



Figure 20: A sample module hierarchy.

that *imported*$(M_1) = \{M_2, M_3, M_5\}$. Note that in Mercury, the *imports* relation is not transitive; when a module $M$ imports the interface of a module $M'$, it becomes dependent on the interfaces imported by $M'$ (and those imported therein) but it does not import these itself. While in Mercury modules may depend on each other in a circular way, we restrict our attention to programs in which no circular dependencies exist between the modules. We discuss circular dependencies in Section 39. The module system described above is to some extent a simplification of Mercury's real module system, in which modules can be constructed from submodules. While submodules do provide extra means to the programmer to control encapsulation and visibility of declarations,

they do not pose additional conceptual difficulties and we do not consider them in the remainder of this work.

In this work, we aim at developing a binding-time analysis that is as modular as possible. Ultimately, a modular analysis deals with each module of a program in isolation. We will discuss throughout the text to what extent our binding-time analysis is modular in this respect.

## 36.2 Mercury programs for analysis

Mercury is an expressive language, in which programs can be composed of predicates and functions, one can use DCG notation, etc. However, if we consider only programs that are type correct and well-moded – which is natural, since the compiler should reject programs that are not [112] – such a program can be translated into *superhomogeneous form* [112]. Translation to superhomogeneous form involves a number of analysis and transformation steps. These include translating an $n$-ary function definition into an $n+1$ ary predicate definition [113], making the implicit arguments in DCG-predicate definitions and calls explicit, and copying and renaming predicate definitions and calls such that every remaining predicate definition has a single mode declaration associated with it [112] that specifies for each argument whether it is an input or output argument. As such, every predicate definition is transformed in a set of so-called *procedure* definitions, with one procedure for every mode in which the original predicate is used.

For our analysis purposes, we assume that a Mercury program is given in superhomogeneous form. This does not involve any loss of generality, as the transformation from a plain Mercury program into superhomogeneous form is completely defined and automated [112]. Formally, the syntax of Mercury programs in superhomogeneous form can be defined as follows. We use the symbol $\Pi$ to refer to the set of *procedure* symbols underlying the language associated to the program. As such, we consider two procedures that are derived from the same predicate as having different procedure symbols.

**Definition 36.1**

$$
\begin{array}{lcl}
Proc & ::= & p(\overline{X}) : -G. \\
Goal & := & Atom \mid not(G) \mid (G_1\ ,\ G_2) \mid (G_1\ ;\ G_2) \mid \textit{if}\ G_1\ \textit{then}\ G_2\ \textit{else}\ G_3 \\
Atom & ::= & X := Y \mid X == Y \mid X \Rightarrow f(\overline{Y}) \mid X \Leftarrow f(\overline{Y}) \mid p(\overline{X})
\end{array}
$$

where $p/n \in \Pi$ and $\overline{X}$ is a sequence of $n$ distinct variables of $\mathcal{V}$, $f/m \in \Sigma$, $\overline{Y}$ a sequence of $m$ distinct variables of $\mathcal{V}$, and $G, G_1, G_2, G_3 \in Goal$.

The definition of a procedure $p$ in superhomogeneous form consists of a single clause. The sequence of arguments in the head of the clause, denoted by $\mathit{Args}(p)$, are distinct variables,

explicit unifications are created for these variables in the body goal – denoted by $\mathcal{B}ody(p)$ – and complex unifications are broken down in several simpler ones. The arguments of a procedure $p$ are divided in a set of input arguments, denoted by $\texttt{in}(p)$ and a set of output arguments denoted by $\texttt{out}(p)$. A goal is either an atom or a number of goals connected by *conjunction*, *disjunction*, *if then else* or *not*. An atom is either a unification or a procedure call. Note that, as an effect of mode analysis [112], unifications are categorised as follows:

- An *assignment* of the form $X := Y$. For such a unification, $Y$ is input, whereas $X$ is output.
- A *test* of the form $X == Y$. Both $X$ and $Y$ are input to the unification and of atomic type.
- A *deconstruction* of the form $X \Rightarrow f(\overline{Y})$. In this case, $X$ is input of the unification whereas $\overline{Y}$ is a sequence of output variables.
- A *construction* of the form $X \Leftarrow f(\overline{Y})$. In this case $X$ is output of the unification whereas $\overline{Y}$ is a sequence of input variables.

During the translation into superhomogeneous form, unifications between values of a complex data type may be transformed into a call to a newly generated procedure that (possibly recursively) performs the unification. For any goal $G$, we denote with $\texttt{in}(G)$ and $\texttt{out}(G)$ the set of its input, respectively output variables[13]

**Example 8** *Consider the classical definition of the* `append/3` *predicate, both in normal syntax and in superhomogeneous form for the mode* `append(in,in,out)` *as depicted in Fig. 21.*

| `append/3` | `append/3` in superhomogeneous form |
|---|---|
| `append([],Y,Y).` | `append(X,Y,Z):-` |
| `append([E\|Es],Y,[E\|R]):-` | `  (X⇒[], Z:=Y ;` |
| `  append(Xs,Y,R).` | `  X⇒[E\|Es], append(Es, Y, R), Z⇐[E\|R]).` |

Figure 21: The `append/3` predicate and `append(in,in,out)` in superhomogeneous form.

According to Definition 36.1, conjunctions and disjunctions are considered binary constructs. This differs from their representation inside the Melbourne compiler [109], where conjunctions

[13]Although Mercury has some support for more involved modes – other than input versus output – that are necessary to support *partially instantiated data structures* at run-time, release 0.9 of the Mercury implementation [109] does not fully support these.

and disjunctions are represented in flattened form. Our syntactic definition however facilitates the conceptual handling of these constructs during analysis.

For analysis purposes, we assume that every subgoal of a procedure body is identified by a unique program point, the set of all such program points is denoted by $\mathcal{P}p$. If we are dealing with a particular procedure, we denote with $\eta_0$ the program point associated with the procedure's head atom, and with $\eta_b$ the program point associated to its body goal. The set of program points identifying the subgoals of a goal $G$ is denoted by $\mathcal{P}ps(G)$, this set includes the program point identifying $G$ itself. If the particular program point identifying a goal $G$ in a procedure's body is important, we subscribe the goal with its program point, as in $G_\eta$ or explicitly state that $\mathcal{P}p(G) = \eta$. An important use of program points is to identify those atoms in the body of a procedure in which a particular variable becomes initialised or, said otherwise, those atoms of which the variable is an output variable. This information is computed by mode analysis, and we assume the availability of a function

$$\texttt{init} : \mathcal{V} \mapsto \wp(\mathcal{P}p)$$

with the intended meaning that, for a variable $V$ used in some procedure, if $\texttt{init}(V) = \{\eta_1, \ldots, \eta_n\}$, the variable $V$ is an output variable of the atoms identified by $\eta_1, \ldots, \eta_n$. Note that the function `init` is implicitly associated with a particular procedure, which we do not mention explicitly. When we use the function `init`, it will be clear from the context to what particular procedure it is associated.

**Example 9** *Let us recall the definition of* `append/3` *in superhomogeneous form for the mode* `append(in,in,out)`*, with the atoms and structured goals occurring in the procedure's definition explicitly identified by subscribing them with their respective program point as in Figure 22. We denote the program points associated to a structured goal by subscripting the goal*

```
append(X,Y,Z)0:-
  ((X⇒[]1, Z:=Y2)c1 ;
   (X⇒[E|Es]3, (append(Es, Y, R)4, Z⇐[E|R]5)c2)c3)d1.
```

Figure 22: `append/3` with explicit program points.

*with the characters 'c' for conjunction and 'd' for disjunction, accompanied by a natural number. From mode analysis, it follows that*

$$\begin{array}{lll} \texttt{init}(X) = \{0\} & \texttt{init}(E) = \{3\} & \texttt{init}(R) = \{4\} \\ \texttt{init}(Y) = \{0\} & \texttt{init}(Es) = \{3\} & \texttt{init}(Z) = \{2, 5\} \end{array}$$

*Or, put otherwise, $X$ and $Y$ (being input arguments) are initialised in the procedure's head, $E$ and $Es$ are initialised in the deconstruction identified by program point $3$, $R$ is initialised in the recursive call whereas $Z$ is initialised either by the assignment $Z := Y$ (program point $2$) or by the construction $Z \Leftarrow [E|R]$ (program point $5$).*

## 36.3 A modular analysis

In order to make the binding-time analysis as much modular as possible, we devise an analysis that works in two phases. In a first phase, we represent binding-times and the relations that exist between them according the data flow in the program in a symbolic way. Doing so enables to perform a large part of the data-flow analysis on this symbolic representation and hence independent of a particular call pattern. It is only in the second phase that call patterns in the form of the binding-times of a procedure's input arguments are combined with the symbolic information derived from the first phase, computing the actual binding-times of the remaining variables and the annotations. The first phase of the analysis hence is *call independent* whereas the second phase is *call dependent*. Obviously, the call independent phase of the analysis does not need to be repeated in case a procedure is called with a different binding-time characterisation of its arguments and consequently, the result of a module's call independent analysis can be used regardless the context the module is used in, and must not be repeated when the module is used in different programs. Since the domain of binding-times is condensing [54], the call-independent analysis preserves the precision that would be obtained by a call-dependent analysis.

To symbolically represent the binding-time of a variable at a particular program point, we introduce the concept of a *binding-time variable*, the set of which is denoted by $\mathcal{V}_{\mathcal{BT}}$. We will denote elements of this set as variables subscribed by a program point. If $V$ is a variable occurring in a goal $G$, and $\eta$ is a program point identifying an atom in $G$, then the binding-time variable $V_\eta \in \mathcal{V}_{\mathcal{BT}}$ symbolically represents the binding-time of $V$ at program point $\eta$. Given a type path $\delta \in \mathit{TPath}$, we use the notation $V_\eta^\delta$ to denote the subvalue identified by $\delta$ in the binding-time of $V$ at program point $\eta$.

**Example 10** *Given the definition of* `append/3` *from Example 9, the binding-time variables $X_0$, $Z_2$, $Z_5$ and $Z_0$ denote, respectively the binding-time of $X$ at the program point $0$ and the binding-times of $Z$ at the program points $2, 5$ and $0$.*

Apart from the binding-time variables that correspond with program variables, we introduce a number of extra binding-time variables that we use to symbolically represent some control information that will be collected (and needed) during the binding-time analysis. For each program

point $\eta$, we introduce two such variables, $\mathcal{R}_\eta$ and $\mathcal{C}_\eta$, that range over the set of binding-times $\{\bot, \top\}$. Their intended meaning is as follows:

- if $\mathcal{R}_\eta = \bot$, the goal identified by $\eta$ reduces either to *true* or *fail* during specialisation, or to some residual code which is guaranteed not to fail at run-time. If, on the other hand, $\mathcal{R}_\eta = \top$, the goal identified by $\eta$ possibly reduces to residual code that can fail at run-time.

- if $\mathcal{C}_\eta = \top$, the goal identified by $\eta$ is under dynamic control in the procedure's body, which is not the case if $\mathcal{C}_\eta = \bot$. We say that an atom is under dynamic control if the fact whether it will be evaluated depends on the success or failure of another goal, say $G_{\eta'}$ while success or failure of that goal is undecided at specialisation-time (that is $\mathcal{R}_{\eta'} = \top$).

Note that these binding-time variables – which we will refer to as *control variables* – are boolean in the sense that they will only assume a value that is either $\bot$ or $\top$. During the binding-time analysis, these control variables collect the necessary information to implement the control strategy of the specialiser. Our analysis models a rather conservative specialisation strategy, in the sense that during specialisation, no atoms are reduced that are under dynamic control. The general idea of this control strategy is as follows: if during specialisation only atoms are reduced that are not under dynamic control, only atoms are reduced that would also be evaluated by an equivalent single stage computation (where the static input part is extended with some "dynamic" input). Indeed, their being evaluated depends only on goals that are – during specialisation– sufficiently reduced in order to decide success or failure. Hence, no atoms are "speculatively" reduced, guaranteeing termination of the reduction process (constituting local termination) under the assumption that the equivalent single stage computation terminates.

**Example 11** *Consider the following code fragment*

```
if X ⇒ [] then p(X) else q(X)
```

*Both atoms $p(X)$ and $q(X)$ are under dynamic control if $X$'s binding-time does not allow the specialiser to decide whether or not the test $X \Rightarrow []$ will succeed during specialisation. Indeed, the specialiser has no means of knowing which of the branches will be taken during the second stage of the computation.*

In general, the binding-time of a program variable can depend on the binding-times of other program variables (according to the data flow) and on the value of the appropriate control variables (according to the control strategy). The values of the control variables that are associated to a goal in turn depend on the binding-times of that goal's input variables. Symbolically, we

can represent these dependencies by a number of constraints between the involved binding-time variables. In general:

**Definition 36.2** A *binding-time constraint* is a constraint of the following form:

$$V_\eta^\delta \succeq X_{\eta'}^\gamma \qquad V_\eta^\delta \succeq \top$$

$$V_\eta^\delta \succeq^* X_{\eta'}^\gamma \qquad V_\eta^\delta \succeq^* \top$$

where $V_\eta, X_{\eta'} \in \mathcal{V}_{\mathcal{BT}}$ and $\delta, \gamma \in$ ***TPath***. The set of all binding-time constraints is denoted by $\mathcal{BTC}$.

A constraint of the form $V_\eta^\delta \succeq X_{\eta'}^\gamma$ denotes that the binding-time represented by $V_\eta^\delta$ must be at least as dynamic as (or *cover*) the binding-time represented by $X_{\eta'}^\gamma$. Note that such a constraint requires the types of $V$ and $X$, denoted by $\tau_V$ and $\tau_X$ to be such that $\tau_V^\delta$ and $\tau_X^\gamma$ are instances of one another, in order for their binding-times to be comparable. The intended meaning of a constraint of the form $V_\eta^\delta \succeq^* X_{\eta'}^\gamma$ is that the binding-time represented by $V_\eta^\delta$ is at least as dynamic as the binding-time value associated to the path identified by $\gamma$ in the binding-time represented by $X_{\eta'}^\gamma$. Note that such a constraint does not require $\tau_V^\delta$ and $\tau_X^\gamma$ to be of comparable types; it simply expresses that if the node identified by $\gamma$ in the binding-time represented by $X_{\eta'}$ is *dynamic*, so must be the node identified by $\delta$ in $V_\eta$ and by definition of a binding-time, so must be all its descendant nodes. Remark that we also allow constraints of which the right-hand side is the constant $\top$. Although we occasionally also consider constraints of which the right-hand side is the constant $\bot$, we do not explicitly mention these in the definition, as these constraints are superfluous: for any $X_\eta \in \mathcal{V}_{\mathcal{BT}}$ and $\delta \in$ ***TPath***, it holds by definition that $X_\eta^\delta \succeq \bot$.

**Example 12** *Reconsider the definition of* append/3 *in Fig. 22. Some examples of binding-time constraints between binding-time variables from* append/3 *and their intended meaning are:*

| | |
|---|---|
| $Z_2 \succeq Y_0$ | *the binding-time associated to* $Z$ *at program point* $2$ *is at least as dynamic as the binding-time associated to* $Y$ *at program point* $0$ |
| $E_3 \succeq X_0^{\langle[],1\rangle}$ | *the binding-time associated to* $E$ *at program point* $3$ *is at least as dynamic as the subvalue denoted by* $\langle[],1\rangle$ *of the binding-time associated to* $X$ *at program point* $0$ |
| $Z_5^{\langle[],1\rangle} \succeq E_3$ | *the subvalue denoted by* $\langle[],1\rangle$ *in the binding-time of* $Z$ *at program point* $5$ *is at least as dynamic as the binding-time associated to* $E$ *at program point* $3$ |
| $\mathcal{R}_3 \succeq^* X_0$ | *the atom at program point* $3$ *reduces to true, fail or code that is guaranteed to succeed if* $X_0$ *represents a binding-time in which the root node* $\langle\rangle$ *is bound to static* |
| $\mathcal{C}_4 \succeq \mathcal{R}_3$ | *the atom at program point* $4$ *is under dynamic control if the atom at program point* $3$ *possibly reduces to code that might fail* |

A set of binding-time constraints is called a binding-time constraint system (or simply a constraint system). Given a constraint system $\mathcal{C}$, we define vars($\mathcal{C}$) as the set of all binding-time variables $X_\eta$ that occur in some constraint $C \in \mathcal{C}$. The link between a binding-time constraint system and the actual binding-times it represents is formalised as a (minimal) solution to the constraint system.

**Definition 36.3** A *solution* to a binding-time constraint system $\mathcal{C}$ is a substitution $\sigma : \mathcal{V}_{\mathcal{BT}} \mapsto \mathcal{BT}$ mapping binding-time variables to binding-times with $dom(\sigma) =$ vars($\mathcal{C}$) such that

- for every constraint $V_\eta^\delta \succeq \top \in \mathcal{C}$ and $V_\eta^\delta \succeq^* \top \in \mathcal{C}$ it holds that $\sigma(V_\eta)^\delta \succeq \top$
- for every constraint $V_\eta^\delta \succeq X_{\eta'}^\gamma \in \mathcal{C}$ it holds that $\sigma(V_\eta)^\delta \succeq \sigma(X_{\eta'})^\gamma$
- for every constraint $V_\eta^\delta \succeq^* X_{\eta'}^\gamma \in \mathcal{C}$ it holds that $\sigma(X_{\eta'})(\gamma) = \textit{dynamic} \Rightarrow \sigma(V_\eta)^\delta \succeq \top$

Given two solutions $\sigma$ and $\sigma'$ to $\mathcal{C}$, we define that $\sigma \sqsupseteq \sigma'$ if for all $V_\eta \in dom(\sigma')$ it holds that $V_\eta \in dom(\sigma)$ and $\sigma(V_\eta) \succeq \sigma'(V_\eta)$. A solution $\sigma$ is a *least solution* for $\mathcal{C}$ if for every solution $\sigma'$ for $\mathcal{C}$ it holds that $\sigma' \sqsupseteq \sigma$.

Remember, a solution must also satisfy the condition of Definition 35.5, i.e. if $\sigma(X_{\eta'})^{\gamma} = \textit{dynamic}$ then also $\sigma(X_{\eta'})^{\gamma \bullet^{\epsilon}} = \textit{dynamic}$ for any extension $\epsilon$. We will sometimes use a constraint of the form $V_{\eta}^{\delta} \succeq X_{\eta'}^{\gamma'} \sqcup Y_{\eta''}^{\gamma''}$ (analogously for $\succeq^*$) as shorthand notation for the set of constraints $\{V_{\eta}^{\delta} \succeq X_{\eta'}^{\gamma'}, V_{\eta}^{\delta} \succeq Y_{\eta''}^{\gamma''}\}$. Indeed, from Definition 36.3 it can be seen that in any solution $\sigma$ satisfying the latter two constraints, it holds that $\sigma(V_{\eta})^{\delta} \succeq \sigma(X_{\eta'}^{\gamma'}) \sqcup \sigma(Y_{\eta''}^{\gamma''})$, where $\sqcup$ denotes the least upper bound on $(\mathcal{BT}^{+}, \succeq)$.

**Example 13** *Consider the following binding-time constraint system and its least solution. For sake of simplicity, we assume that all binding-time variables are boolean and range over the set* $\{\textit{dynamic}, \textit{static}\}$.

| *Binding-time constraint system* | *Least solution* |
|---|---|
| $X_{\eta_1} \succeq \top$ | |
| $R_{\eta_3} \succeq X_{\eta_2}$ | $\left\{ \begin{array}{ll} (X_{\eta_1}, \textit{dynamic}) & (X_{\eta_2}, \textit{static}) \\ (R_{\eta_3}, \textit{static}) & (Y_{\eta_4}, \textit{dynamic}) \end{array} \right\}$ |
| $Y_{\eta_4} \succeq X_{\eta_1}$ | |
| $Y_{\eta_4} \succeq R_{\eta_3}$ | |

In what follows, we formulate our analysis as a call-independent abstract semantics. We define the abstract "meaning" of a goal, be it an atom or a structured goal, as a set of binding-time constraints (description domain $\wp(\mathcal{BTC})$) that reflect the data flow between the input- and output arguments of the goal. An essential operator for the symbolic data flow analysis is a projection operator that basically rewrites a set of constraints such that every constraint expresses (or constrains) the binding-time of a local variable within a procedure in function of the binding-time(s) of that procedure's input arguments. Such a constraint is said to be in normal form:

**Definition 36.4** A binding-time constraint is in *normal form* with respect to a procedure $p \in Proc$ if it is either of the form

- $V_{\eta}^{\delta} \succeq \top$
- $V_{\eta}^{\delta} \succeq X_{\eta_0}^{\gamma}$ with $X \in \texttt{in}(p)$ and $\eta_0$ the program point associated to $p$'s head atom.

and analogously for constraints of this form using $\succeq^*$.

**Example 14** *Reconsider the binding-time constraints from Example 12. The constraints*

$$Z_2 \succeq Y_0 \qquad E_3 \succeq X_0^{\langle [], 1 \rangle} \qquad \mathcal{R}_3 \succeq^* X_0$$

*are in normal form with respect to* `append/3`*, whereas the constraints*

$$Z_5^{\langle [], 1 \rangle} \succeq E_3 \qquad \mathcal{C}_4 \succeq \mathcal{R}_3$$

*are not.*

Projection of a constraint involves unfolding the (subvalue of the) binding-time variable in its right-hand side with respect to a single constraint on (a subvalue of) this variable. If we consider two subvalues of a binding-time variable, say $X_\eta^\delta$ and $X_\eta^\gamma$, one of them is a subvalue of the other if either $\delta$ is an extension of $\gamma$ or vice versa. This is captured by the following definition:

**Definition 36.5** We define ext : *TPath* $\times$ *TPath* $\mapsto$ *TPath* $\times$ *TPath* as follows:

$$\text{ext}(\gamma,\delta) = \begin{cases} (\langle\rangle, \epsilon) & \text{if } \gamma = \delta \bullet \epsilon \\ (\epsilon, \langle\rangle) & \text{if } \gamma \bullet \epsilon = \delta \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that if $\text{ext}(\gamma,\delta) = (\epsilon,\epsilon')$ then $\gamma \bullet \epsilon = \delta \bullet \epsilon'$. Unfolding a constraint $X_\eta^\gamma \succeq Y_{\eta'}^\delta$ with respect to another constraint result in a new constraint on (a subvalue of) $X_\eta^\gamma$, with as right hand side the appropriate subvalue of the right hand side of the constraint that was used for unfolding. To denote a subvalue of a constraint's right hand side $\phi$ (which is either a binding-time variable or a one of the constants $\top$ or $\bot$), we use the notation $\phi^{\overline{\bullet\epsilon}}$. If $\phi$ denotes a variable $X_\eta^\gamma$, then $\phi^{\overline{\bullet\epsilon}}$ equals $X_\eta^{\overline{[\gamma\bullet\epsilon]}}$. Otherwise, if $\phi$ denotes one of the constants $\bot$ or $\top$, $\phi^{\overline{\bullet\epsilon}}$ simply equals $\phi$. Note the use of the least element of the equivalence class, $\overline{[\gamma \bullet \eta]}$, to denote an element of the appropriate type graph $\mathcal{L}_\tau^{\equiv}$ (rather than the type tree $\mathcal{L}_\tau$). The projection operation is defined in Definition 36.6 and basically consists of a fixed point iteration over an unfolding operator followed by a selection operation that retrieves the constraints of interest from the fixed point. Recall that $\eta_0$ identifies the head atom of the procedure of interest.

**Definition 36.6** The *projection* of a set $S \subseteq \wp(\mathcal{BTC})$ on a set of binding-time variables $V \subseteq \mathcal{V}_{\mathcal{BT}}$ is denoted by $\text{proj}_V S$ and defined as

$$\text{proj}_V(S) = \{X \succeq^{(*)} \phi \in \text{lfp}(\text{unf}_S) \mid X \in V\}$$

where $\text{unf}_S$ is defined in Figure 23.

The symbolic analysis is defined in Definition 36.7. The result of analysing a program is a mapping (from the semantic domain $Den$) that maps a procedure symbol $p$ to a set of binding-time constraints on the variables that occur in the definition of the procedure $p$. The constraints are in normal form. Polyvariance is immediate, since all constraints are expressed in terms of the procedure's input arguments, which are represented symbolically and hence can be instantiated by any call pattern. The analysis is defined by a number of semantic functions defining the abstract semantics of a program $\mathbf{P} : Prog \mapsto Den$ in terms of the semantics of the individual procedures, goals and atoms.

$$\text{unf}_S : \wp(\mathcal{BTC}) \mapsto \wp(\mathcal{BTC})$$

$$\text{unf}_S(I) = \{X_\eta \succeq^{(*)} Y_{\eta_0}, X_\eta \succeq^{(*)} \top \in S\} \cup S_1 \cup S_2 \cup S_3$$

where

$$\begin{aligned}
S_1 &= \{X_\eta^{\overline{[\gamma \bullet \epsilon]}} \succeq \phi^{\overline{\bullet \epsilon'}} \mid X_\eta^\gamma \succeq Y_{\eta'}^\delta \in S, Y_{\eta'}^{\delta'} \succeq \phi \in I, \text{ and ext}(\delta, \delta') = (\epsilon, \epsilon')\} \\
S_2 &= \{X_\eta^\gamma \succeq^* \phi \mid X_\eta^\gamma \succeq Y_{\eta'} \in S \text{ and } Y_{\eta'} \succeq^* \phi \in I\} \\
S_3 &= \{X_\eta \succeq^* \phi^{\overline{\bullet \epsilon}} \mid X_\eta \succeq^* Y_{\eta'}^\delta \in S, Y_{\eta'}^{\delta'} \succeq \phi \in I \text{ and ext}(\delta, \delta') = (\langle\rangle, \epsilon)\}
\end{aligned}$$

Figure 23: The projection $\text{proj}_V$

**Definition 36.7** The *call independent abstract semantics* for description domain $\wp(\mathcal{BTC})$ has semantic domain

$$Den : \Pi \mapsto \wp(\mathcal{BTC})$$

and semantic functions

$$\mathbf{P} : Prog \mapsto Den$$

$$\mathbf{C} : Proc \mapsto Den \mapsto Den$$

$$\mathbf{G} : Goal \mapsto Den \mapsto \wp(\mathcal{BTC})$$

$$\mathbf{A} : Atom \mapsto Den \mapsto \wp(\mathcal{BTC})$$

and is defined in Figures 24 and 25.

The result of analysing a program is a denotation, $\mathbf{P}[\![P]\!]P$, in the domain $Den$, which is a mapping from a predicate symbol to a set of binding-time constraints. This mapping is defined as the least fixed point of applying the analysis function $\mathbf{C}$ to each individual procedure. The analysis function $\mathbf{C}$ constructs a partial denotation for a particular procedure, given a (possibly incomplete) denotation that represents the result of analysis of the whole program so far. The analysis functions $\mathbf{G}$ and $\mathbf{A}$ map respectively a structured goal and an atomic goal to a set of binding-time constraints, given a denotation – again representing the result of analysing the whole program so far. In general, the result of analysing a complex goal is the union of the constraints obtained by analysing each subgoal in isolation, together with a number of additional constraints on the control variables associated with the goal and its subgoals. These constraints are simple, as they merely reflect the propagation of the control variable's value, either from the

$$\mathbf{P}[\![P]\!] = \text{lfp}(\bigcup_{p \in Proc(P)} \mathbf{C}[\![p]\!])$$

$$\mathbf{C}[\![p(\overline{X}) \leftarrow G_\eta]\!]d = \{(p, \mathbf{G}[\![G_\eta]\!]d)\}$$

$$\begin{array}{lcl}
\mathbf{G}[\![(G'_{\eta'}, G''_{\eta''})_\eta]\!]d & = & \mathbf{G}[\![G'_{\eta'}]\!]d \sqcup \mathbf{G}[\![G''_{\eta''}]\!]d \sqcup CC_{\text{conj}}(\eta, \eta', \eta'') \\
\mathbf{G}[\![\mathit{not}_\eta(G_{\eta'})]\!]d & = & \mathbf{G}[\![G_{\eta'}]\!]d \sqcup CC_{\text{not}}(\eta, \eta') \\
\mathbf{G}[\![\mathit{if}_\eta\ G'_{\eta'}\ \mathit{then}\ G''_{\eta''}\ \mathit{else}\ G'''_{\eta'''}]\!] & = & \mathbf{G}[\![G'_{\eta'}]\!]d \sqcup \mathbf{G}[\![G''_{\eta''}]\!]d \sqcup \mathbf{G}[\![G'''_{\eta'''}]\!]d \sqcup CC_{\text{if}}(\eta, \eta', \eta'', \eta''') \\
\mathbf{G}[\![(G'_{\eta'}; G''_{\eta''})_\eta]\!]d & = & \mathbf{G}[\![G'_{\eta'}]\!]d \sqcup \mathbf{G}[\![G''_{\eta''}]\!]d \sqcup CC_{\text{disj}}(\eta, \eta', \eta'') \\
\mathbf{G}[\![A_\eta]\!]d & = & \mathbf{A}[\![A_\eta]\!]d \sqcup \{X_\eta \succeq X_{\eta'} \mid X \in \texttt{in}(A), \eta' \in \texttt{reach}(X, \eta)\}
\end{array}$$

$$\begin{array}{lcl}
\mathbf{A}[\![X =\!=_\eta Y]\!]d & = & \{\mathcal{R}_\eta \succeq^* X_\eta \sqcup Y_\eta\} \\
\mathbf{A}[\![X :=_\eta Y]\!]d & = & \{X_\eta \succeq Y_\eta \sqcup \mathcal{C}_\eta,\ \mathcal{R}_\eta \succeq \bot\} \\
\mathbf{A}[\![X \Rightarrow_\eta f(\overline{Y})]\!]d & = & \bigsqcup_{Y_i \in \overline{Y}} \{Y_{i,\eta} \succeq X_\eta^{\overline{[\langle f,i \rangle]}} \sqcup \mathcal{C}_\eta\} \cup \{\mathcal{R}_\eta \succeq^* X_\eta\} \\
\mathbf{A}[\![X \Leftarrow_\eta f(\overline{Y})]\!]d & = & \bigsqcup_{Y_i \in \overline{Y}} \{X_\eta^{\overline{[\langle f,i \rangle]}} \succeq Y_{i,\eta} \sqcup \mathcal{C}_\eta\} \cup \{\mathcal{R}_\eta \succeq \bot\} \\
\mathbf{A}[\![p(X_1, \ldots, X_n)_\eta]\!]d & = & \rho(\text{proj}_{\mathit{Args}(p), \mathcal{R}_{\eta_b}}\, d\, p) \sqcup \{X_{i,\eta} \succeq \mathcal{C}_\eta \mid X_i \in \texttt{out}(p)\}
\end{array}$$

where $\mathit{Args}(p)$ denotes the sequence of formal arguments in the definition of $p/n$, $\eta_b$ is associated to the body goal in the definition of $p/n$ and $\rho$ is a renaming mapping the sequence of formal arguments $\mathit{Args}(p)$ to the sequence of actual arguments $\langle X_1, \ldots, X_n \rangle$ and $\mathcal{R}_{\eta_b}$ to $\mathcal{R}_\eta$.

Figure 24: The call independent abstract semantics

$$CC_{\text{conj}}(\eta, \eta', \eta'') = \left\{ \begin{array}{ccc} \mathcal{C}_{\eta'} \succeq \mathcal{C}_{\eta} & \mathcal{C}_{\eta''} \succeq \mathcal{C}_{\eta} & \mathcal{C}_{\eta''} \succeq \mathcal{R}_{\eta'} \\ & & \\ \mathcal{R}_{\eta} \succeq \mathcal{R}_{\eta'} & \mathcal{R}_{\eta} \succeq \mathcal{R}_{\eta''} & \end{array} \right\}$$

$$CC_{\text{disj}}(\eta, \eta', \eta'') = \left\{ \begin{array}{cc} \mathcal{C}_{\eta'} \succeq \mathcal{C}_{\eta} & \mathcal{C}_{\eta''} \succeq \mathcal{C}_{\eta} \\ & \\ \mathcal{R}_{\eta} \succeq \mathcal{R}_{\eta'} & \mathcal{R}_{\eta} \succeq \mathcal{R}_{\eta''} \end{array} \right\}$$

$$CC_{\text{not}}(\eta, \eta') = \left\{ \begin{array}{cc} \mathcal{C}_{\eta'} \succeq \mathcal{C}_{\eta} & \mathcal{R}_{\eta} \succeq \mathcal{R}_{\eta'} \end{array} \right\}$$

$$CC_{\text{if}}(\eta, \eta', \eta'', \eta''') = \left\{ \begin{array}{ccc} \mathcal{C}_{\eta'} \succeq \mathcal{C}_{\eta} & \mathcal{C}_{\eta''} \succeq \mathcal{C}_{\eta} & \mathcal{C}_{\eta'''} \succeq \mathcal{C}_{\eta} \\ & & \\ \mathcal{C}_{\eta''} \succeq \mathcal{R}_{\eta'} & \mathcal{C}_{\eta'''} \succeq \mathcal{R}_{\eta'} & \\ & & \\ \mathcal{R}_{\eta} \succeq \mathcal{R}_{\eta'} & \mathcal{R}_{\eta} \succeq \mathcal{R}_{\eta''} & \mathcal{R}_{\eta} \succeq \mathcal{R}_{\eta'''} \end{array} \right\}$$

Figure 25: The call independent abstract semantics (ctd.)

goal to its subgoals (in case of the control variable $\mathcal{C}$) or from the goal's subgoals to the goal itself (in case of $\mathcal{R}$). The binding-time variables denoting dynamic control denote that a goal is under dynamic control *with respect to the procedure's body*. The negated goal in a negation is under dynamic control only if the negation itself is. Observe that if $A$ reduces to true or is guaranteed to succeed, then not$(A)$ fails. And if $A$ fails then not$(A)$ succeeds. So we can say that the negation reduces to true, fail, or residual code which is guaranteed to succeed if the negated goal does. The propagation in the other constructs is similar: the subgoals of an if-then-else are under dynamic control if the if-then-else is under dynamic control. Moreover, both the then and else goals are under dynamic control if the test goal possibly reduces to residual code which could fail at run time. If each of the if-then-else's subgoals reduces to true, fail or code that is guaranteed to succeed, so does the if-then-else. The subgoals of a conjunction are under dynamic control if the conjunction itself is. Moreover, the second conjunct is under dynamic control if the first conjunct possibly reduces to residual code that could fail. If both conjuncts reduce to true, fail or code that is guaranteed to succeed, so does the conjunction. To conclude, if a disjunction is under dynamic control, so are both disjuncts. If both disjuncts reduce to true, fail or code that is guaranteed to succeed, so does the disjunction.

**Example 15** *Reconsider the definition of* `append/3` *in Figure 22. The body goal contains the following structured subgoals: a conjunction identified by program point* $c_1$ *with the atomic conjuncts identified by program points* $1$ *and* $2$*, a second conjunction identified by* $c_2$ *with the atomic conjuncts identified by program points* $4$ *and* $5$*, a third conjunction identified by* $c_3$ *with the conjuncts identified by program points* $3$ *and* $c_2$ *and a disjunction identified by program point* $d_1$ *with the disjuncts identified by* $c_1$ *and* $c_3$*. The binding-time constraints that are associated to each of these structured goals are as follows:*

| | | |
|---|---|---|
| $(c_1)$ | $\mathcal{C}_1 \succeq \mathcal{C}_{c_1}$ | $\mathcal{R}_{c_1} \succeq \mathcal{R}_1$ |
| | $\mathcal{C}_2 \succeq \mathcal{C}_{c_1}$ | $\mathcal{R}_{c_1} \succeq \mathcal{R}_2$ |
| | $\mathcal{C}_2 \succeq \mathcal{R}_1$ | |
| $(c_2)$ | $\mathcal{C}_4 \succeq \mathcal{C}_{c_2}$ | $\mathcal{R}_{c_2} \succeq \mathcal{R}_4$ |
| | $\mathcal{C}_5 \succeq \mathcal{C}_{c_2}$ | $\mathcal{R}_{c_2} \succeq \mathcal{R}_5$ |
| | $\mathcal{C}_5 \succeq \mathcal{R}_4$ | |
| $(c_3)$ | $\mathcal{C}_3 \succeq \mathcal{C}_{c_3}$ | $\mathcal{R}_{c_3} \succeq \mathcal{R}_3$ |
| | $\mathcal{C}_{c_2} \succeq \mathcal{C}_{c_3}$ | $\mathcal{R}_{c_3} \succeq \mathcal{R}_{c_2}$ |
| | $\mathcal{C}_{c_2} \succeq \mathcal{R}_3$ | |
| $(d_1)$ | $\mathcal{C}_{c_1} \succeq \mathcal{C}_{d_1}$ | $\mathcal{R}_{d_1} \succeq \mathcal{R}_{c_1}$ |
| | $\mathcal{C}_{c_3} \succeq \mathcal{C}_{d_1}$ | $\mathcal{R}_{d_1} \succeq \mathcal{R}_{c_3}$ |

The binding-time constraints that are associated to an atomic goal are somewhat more involved. Apart from binding-time constraints on the atom's output variables, analysing an atom $A_\eta$ also possibly results in a binding-time constraint on the control variable $\mathcal{R}_\eta$, indicating under what conditions the atom can be reduced to true, fail, or code that is guaranteed to succeed. Moreover, when creating the binding-time constraints on the atom's output variables, the control variable $\mathcal{C}_\eta$ must be taken into account, in order to guarantee that the particular binding-time is made $\top$ in case the atom is under dynamic control.

Note that in the definition of **A** the binding-time variables that refer to the *input* variables of an atom at program point $\eta$ are indexed by the program point $\eta$. Consequently, a number of additional constraints must be created for each atom, relating the binding-time of such an input argument at program point $\eta$ with its binding-time at the program point(s) where the binding-time was created, being output of some other atom.

A test does not have any output variables, so it only creates constraints on control variables. The atom reduces to true, fail or code that is guaranteed to succeed when both input variables are bound to an outermost functor. An assignment $X := Y$ introduces the constraints specifying that the binding-time of $X$ at program point $\eta$ must be at least as dynamic as the binding-time of $Y$ at program point $\eta$. Recall that the latter's value is constrained to be at least as dynamic as the least upper bound of the binding-times of $Y$ at the reachable program points where $Y$ is assigned a value. Moreover, if the assignment is under dynamic control, $X_\eta$ must be assigned the value $\top$. This is guaranteed by adding $\sqcup\, \mathcal{C}_\eta$ to the right-hand side of the constraint on $X_\eta$. Even if an assignment is not reduced, it can never fail at run time. Hence the (superfluous) constraint $\mathcal{R}_\eta \succeq \bot$. A deconstruction introduces some binding-time constraints indicating that the binding-time of the newly introduced variables must be at least as dynamic as the corresponding subvalue in the binding-time of the variable that is deconstructed. Also in this case, the least upper bound with $\mathcal{C}_\eta$ guarantees that, if the deconstruction is under dynamic control, the newly introduced binding-time variables will be forced to have the value $\top$. If the deconstructed variable is bound to at least an outermost functor, the deconstruction reduces to true or fail at specialisation time. Otherwise, a residualised deconstruction can either succeed or fail at run time which is reflected by the fact that in that case $\mathcal{R}_\eta$ will have the value $\top$. When handling a construction on the other hand, the binding-time of the constructed variable is constrained by the binding-times of the variables used in the construction. Again, if the construction is under dynamic control, the constructed binding-time is guaranteed to be $\top$ by the use of the least upper bound with $\mathcal{C}_\eta$. Even when residualised, a construction can never fail, so again the (superfluous) constraint $\mathcal{R}_\eta \succeq \bot$ is introduced.

**Example 16** *Reconsider the definition of* `append/3` *in Figure 22. The constraints that are associated to the unifications in* `append/3`*'s body goal are as follows. The numbers in the left hand side column denote the particular unification's program point.*

| | | |
|---|---|---|
| $(1)$ | $\mathcal{R}_1 \succeq^* X_0$ | |
| $(2)$ | $\mathcal{R}_2 \succeq \bot$ | $Z_2 \succeq Y_0$ |
| $(3)$ | $\mathcal{R}_3 \succeq^* X_0$ | $E_3 \succeq X_0^{\langle[[],1\rangle}$<br>$Es_3 \succeq X_0^{\langle\rangle}$ |
| $(5)$ | $\mathcal{R}_5 \succeq \bot$ | $Z_5^{\langle[[],1\rangle} \succeq E_3$<br>$Z_5^{\langle\rangle} \succeq R_4$ |

Finally, handling a procedure $p(X_1, \ldots, X_n)$ call involves retrieving the constraints for the called procedure $p$ from the denotation and projecting these onto the set of variables $\mathcal{A}rgs(p) \cup \{\mathcal{R}_{\eta_b}\}$. This projection operation makes sure that the constraints on these variables are in normal form, i.e. that they are expressed in terms of $\mathtt{in}(p)$. The resulting set of constraints is then renamed to the context of the call: the formal arguments of $p$, $\mathcal{A}rgs(p)$ are renamed to their corresponding actual argument in $\langle X_1, \ldots, X_n\rangle$ and the constraints on $\mathcal{R}_{\eta_b}$ are renamed to constraint on $\mathcal{R}_\eta$, expressing that the call reduces to true, fail or code that is guaranteed to succeed if the body of the called procedure reduces to true, fail or code that is guaranteed to succeed.

**Example 17** *Let $P$ denote the program consisting only of the definition of* `append/3` *depicted in Figure 22 and let $(1)$ and $(2)$ denote, respectively, the sets of constraints depicted in Examples 15 and 16. The fixed point computation for $\mathbf{P}[\![P]\!]$ starts with an empty denotation and hence, in the first round of the computation, the recursive call does not introduce any constraints; the result of $\mathbf{C}[\![$*`append/3`*$]\!]\{\}$ is a denotation that maps* `append/3` *to the constraint set $(1) \cup (2)$. It is only in the second round, when the constraints are projected and renamed, that the recursive call adds the constraints*

$$R_4 \succeq Y_0 \qquad R_4^{\langle([[],1)\rangle} \succeq Es_3^{\langle([[],1)\rangle} \qquad \mathcal{R}_4 \succeq^* Es_0$$

*One can verify that in a next round no new constraints are introduced by the recursive call, and hence $\mathbf{P}[\![P]\!]$ results in a denotation that associates* `append/3` *to the union of the constraints derived above with the sets $(1)$ and $(2)$.*

## 36.4 From constraints to annotations

Once we have computed $\mathbf{P}[\![P]\!]$, it suffices to have a set of binding-times for the input variables of a procedure $p$ in order to compute the binding-times of the remaining variables in the definition

of $p$, as well as the annotations that are associated with a particular atom in the definition of $p$. Let us first introduce the semantic domain $Call$, that we use to represent a call in the domain of binding-times:

$$Call = \{p(\beta_1, \ldots, \beta_n) \mid p/n \in \Pi \text{ and } \forall i : \beta_i \in \mathcal{BT}^+\}$$

To ease notation, we assume that such a call contains a binding-time for each argument (input as well as output). However, since these calls are used to represent the binding-times of the *input* arguments of the call only, we asume the binding-times of the output arguments to be $\bot$. We will denote elements of $Call$ by a single greek letter $\pi$ if the particular procedure/argument combination is irrelevant. We can now define the annotation of a procedure with respect to a particular call as follows:

**Definition 36.8** Given a denotation $d \in Den$ for a program $P$ and a call $p(\beta_1, \ldots, \beta_n) \in Call$, the *procedure annotation (of a procedure $p \in Proc(P)$) induced by a call $p(\beta_1, \ldots, \beta_n)$* is defined as the least solution $\sigma$ of $(d\, p)$ in which $\sigma(X_i) = \beta_i$ for every $X_i \in \mathtt{in}(p)$.

Being a solution of the set of binding-time constraints associated to a procedure $p$, a procedure annotation not only provides binding-times for all program variables in $p$, but also maps every binding-time variable of the form $\mathcal{C}_\eta$ to either $\bot$ or $\top$, denoting respectively that the goal at program point $\eta$ in the procedure's body should be evaluated during specialisation, or be residualised. Being a *least* solution, a procedure annotation contains the least dynamic binding-times while still satisfying the congruence relation. As such, a procedure annotation of a procedure $p$ with respect to a call $\pi$ represents control information for a specialiser as to how to treat each subgoal of the body of $p$, when a call to $p$ is approximated by $\pi$.

A polyvariant analysis for a program $P$ and an initial call $p(\beta_1, \ldots, \beta_n)$ can then be performed by first computing the procedure annotation $\sigma$ of $p$ induced by $p(\beta_1, \ldots, \beta_n)$ and consecutively computing, for every call $q(X_1, \ldots, X_m)$ that occurs at some program point $\eta$ in the definition of $p$, the procedure annotation of $q$ induced by $q(\sigma(X_{1_\eta}), \ldots, \sigma(X_{m_\eta}))$. This process is repeated recursively until no more abstract calls are encountered for which no procedure annotation has been constructed yet. In other words, a polyvariant annotation process for a program $P$ with initial call $\pi$ boils down to computing the abstract callset of $(P, \pi)$: The set of abstractions of all calls that can possibly be encountered during evaluation of $P$ with respect to a call that is abstracted by $\pi$. Formally, we define also this annotation process by a number of semantic functions that define the meaning of a program $P$ with respect to an initial call $\pi$ as a set of calls in the domain of binding-times.

$$\mathbf{P_c}[\![P]\!]\pi = \text{lfp}(\bigcup_{p \in Proc(P)} \mathbf{C_c}[\![p]\!]\pi)$$

$$\mathbf{C_c}[\![p(X_1,\ldots,X_n) \leftarrow B]\!]\pi S = \bigcup_{p(\beta_1,\ldots,\beta_b) \in S \cup \{\pi\}} \mathbf{G_c}[\![B]\!]p(\beta_1,\ldots,\beta_n)$$

$$\begin{array}{lcl}
\mathbf{G_c}[\![\textit{not}(G)]\!]\pi & = & \mathbf{G_c}[\![G]\!]\pi \\
\mathbf{G_c}[\![G_1, G_2]\!]\pi & = & \mathbf{G_c}[\![G_1]\!]\pi \cup \mathbf{G_c}[\![G_2]\!]\pi \\
\mathbf{G_c}[\![G_1; G_2]\!]\pi & = & \mathbf{G_c}[\![G_1]\!]\pi \cup \mathbf{G_c}[\![G_2]\!]\pi \\
\mathbf{G_c}[\![\textit{if } G_1 \textit{ them } G_2 \textit{ else } G_3]\!]\pi & = & \mathbf{G_c}[\![G_1]\!]\pi \cup \mathbf{G_c}[\![G_2]\!]\pi \cup \mathbf{G_c}[\![G_3]\!]\pi \\
\mathbf{G_c}[\![q(Y_1,\ldots,Y_n)]\!]\pi & = & \{q(\sigma_\pi(Y_1),\ldots,\sigma_\pi(Y_n))\}
\end{array}$$

and $\mathbf{G_c}[\![A]\!]\pi = \emptyset$ for any other atomic goal $A$ and where $\sigma_\pi$ denotes the procedure annotation induced by $\pi \in Call$.

Figure 26: The annotation semantics

**Definition 36.9** The *first-order* annotation semantics has semantic domain $Den_c : \wp(Call)$ and semantic functions

$$\mathbf{P_c} : Prog \mapsto Call \mapsto Den_c$$

$$\mathbf{C_c} : Proc \mapsto Call \mapsto Den_c \mapsto Den_c$$

$$\mathbf{G_c} : Goal \mapsto Call \mapsto Den_c$$

defined in Figure 26.

The definition of the semantic functions $\mathbf{P_c}$, $\mathbf{C_c}$ and $\mathbf{G_c}$ is straightforward. The semantic domain $Den_c = \wp(Call)$ represents the set of all abstract callsets. The semantics of a program $P$ with respect to an initial call $\pi$ is defined as the least fixed point of repeatedly computing the semantics of each procedure (by $\mathbf{C_c}$) in $P$ within the context of this initial call and a (possibly incomplete) denotation containing the result of analysis so far. The analysis function $\mathbf{C_c}$ constructs a partial denotation for a particular procedure as the union of the denotations obtained by analysing the procedure's body goal with respect to every call to the procedure encountered so far. The semantics of an individual goal $G$ in the body of a procedure $p$ is defined with respect to a call $\pi$ to $p$. The definition of $\mathbf{G_c}$ is straightforward, as it only collects the abstract calls encountered in the annotation of $p$ induced by $\pi$. Note that the analysis is guaranteed to create a finite number of procedure annotations since every procedure has a finite number of arguments, every such argument can only be approximated by a finite number of binding-times, and hence only a finite number of call patterns can be constructed for a particular procedure.

## 36.5 On the modularity of the approach

In summary, the binding-time analysis we have developed so far is to be performed in two phases. The first phase of the process performs the data flow analysis in a symbolic way. A procedure is analysed independent of a particular call pattern, and the analysis handles procedure calls by projecting and renaming the constraints that are associated to the called procedure. For a program that is divided into several modules, this means that the constraint generating phase of the analysis can be performed one module at a time, bottom-up in the module hierarchy if we consider hierarchies without circularities. Reconsider the module hierarchy from Fig. 20. The result of bottom-up analysis of this hierarchy is depicted in Fig. 27. First, the modules at the bottom level, $M_4$ and $M_5$ are analysed. Since these modules do not import any other modules, they can be treated as regular programs, and we can simply compute $\mathbf{P}[\![M_4]\!]$ and $\mathbf{P}[\![M_5]\!]$. The rounded boxes in the figure denote the result of computing $\mathbf{P}[\![M]\!]$ for a particular module $M$. The shaded part of the box represent this denotation, restricted to the procedures from the module's interface. Subsequently, the modules $M_2$ and $M_3$ can be analysed, since their analysis only requires

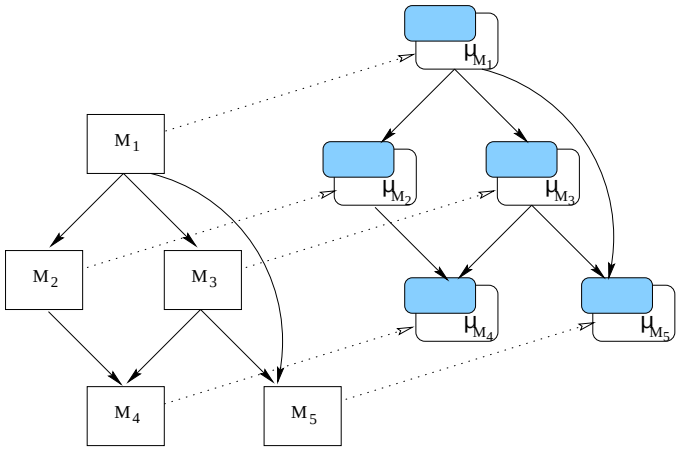



Figure 27: Bottom-up analysis of the module hierarchy.

the constraints from the interface procedures of $M_4$, respectively $M_4$ and $M_5$. Computation of $\mathbf{P}[\![M_2]\!]$ and $\mathbf{P}[\![M_3]\!]$ can proceed as before, with the exception that the fixed point computation should not be started from the empty denotation, but rather from $\mathbf{P}[\![M_4]\!]$ and $\mathbf{P}[\![M_4]\!] \cup \mathbf{P}[\![M_5]\!]$ respectively. Finally, since now the result is available of analysing $M_2$, $M_3$ and $M_5$, the module $M_1$ can be analysed. Note that in this process, each module is analysed only once. If a module, like $M_5$ in the example, is imported in more than one module, analysing the latter modules only requires the *result* of analysing the former.

The second phase of the analysis, computing the procedure annotations, is naturally a call-dependent process. Consequently, annotating a multi-module program for an initial call to a

procedure $p$ in the top-level module requires the constraints for all the procedures (spread out over all modules) that are in the call graph for $p$. One could argue that this corresponds to analysing a multi-module program as if it was a single-module monolithic program. However, it should be noted that computing a procedure annotation induced by a particular call is a rather cheap process. Since the involved constraints are in normal form, it merely consists of performing a substitution on the right-hand side of the constraints and computing their least upper bounds. The hard part of the analysis – tracing the data flow between the input- and output arguments of a procedure – which possibly involves procedure calls over module boundaries, is done at the symbolic level, in a modular fashion.

# 37 Higher-order Binding-time Analysis

Mercury is a higher-order language in which *closures* can be created, passed as arguments of predicate calls, and in turn be called themselves. To describe the higher-order features of the language, it suffices to extend the definition of superhomogeneous form (see Definition 36.1) with two new kinds of atoms:

- A *higher-order unification* which is of the form $X \Leftarrow p(V_1, \ldots, V_k)$ where $X, V_1, \ldots, V_k \in \mathcal{V}$ and $p/n \in \Pi$ with $k \leq n$.
- A *higher-order call* which is of the form $X(V_{k+1}, \ldots, V_n)$ where $X$,$V_{k+1}$,…,$V_n \in \mathcal{V}$ with $0 \leq k \leq n$.

A higher-order unification $X \Leftarrow p(V_1, \ldots, V_k)$ constructs a closure from an $n$-arity procedure $p$ by *currying* the first $k$ arguments (with $k \leq n$). The result of the construction is assigned to the variable $X$ and denotes a procedure of arity $n-k$. Such a closure can be called by a higher-order call of the form $X(V_{k+1}, \ldots, V_n)$ where $V_{k+1}, \ldots, V_n$ are the $n-k$ remaining arguments. The effect of evaluating the conjunction $X \Leftarrow p(V_1, \ldots, V_k), X(V_{k+1}, \ldots, V_n)$ equals the effect of evaluating $p(V_1, \ldots, V_n)$.[14]

In order to represent higher-order *types* it suffices to add a special type constructor, $pred$, to $\Sigma_{\mathcal{T}}$. This constructor is special in the sense that it can be used with any arity and it has no type rule associated with it. Consequently, a higher-order type corresponds with a leaf node in

[14]When writing Mercury code, the programmer can also use lambda expressions to construct closures. These can, however, be converted into a regular procedure definition which is then again used to construct the closure as above. The Melbourne Mercury compiler does this conversion as part of the translation into superhomogeneous form. Note that closures cannot be constructed from other closures: once a closure is created, one can only call it or pass it as an argument to another procedure.

a type tree. In what follows we represent higher-order types as $pred(t_1, \ldots, t_k)$ with $t_1, \ldots, t_k$ first-order types. We furthermore assume that higher-order types are not used in the definition of other types; that is, values of higher-order type are only constructed, called, or passed around as arguments of a procedure call.[15]

The basic problem when analysing a procedure involving higher-order calls, is that the control flow in the procedure is determined by the values of the higher-order variables. To retrieve a set of suitable binding-time constraints between the in- and output arguments of a higher-order call $X(Y_{k+1}, \ldots, Y_n)$, it is necessary to know to some extent to what closures $X$ can be bound to during specialisation. Consequently, to achieve an acceptable level of precision, the symbolic data flow analysis needs to be enhanced by some form of *closure analysis* [60, 98] which basically computes for every higher-order call an approximation of the closures that may be bound to the higher-order variable involved. In what follows, we will first define a suitable representation for such closure information, and reformulate the first phase of our binding-time analysis so that it integrates the derivation of closure information with the derivation of binding-time constraint systems. Doing so basically transforms the process of building constraint systems into a call dependent process, since closures can be passed around by procedure calls and hence the analysis needs to take the closure information from a particular call pattern into account. We conclude this section with a discussion on the modularity of the higher-order approach.

## 37.1 Representing closures

In order to use closures during binding-time analysis, where concrete values of the closure's curried arguments are approximated by binding-times, we introduce the notion of a *binding-time closure* as follows.

**Definition 37.1** A *binding-time closure* is a term of the form $p(\beta_1, \ldots, \beta_k)$ where $p/n \in \Pi$, $k \leq n$ and $\beta_1, \ldots, \beta_k \in \mathcal{BT}^+$. The set of all such binding-time closures is denoted by $\mathcal{C}los$.

If $p/n \in \Pi$, $p(\beta_1, \ldots, \beta_k)$ approximates a set of procedures of arity $n-k$, each being an instance of $p$ in which the first $k$ arguments are fixed and whose values are approximated by the binding-times $\beta_1, \ldots, \beta_k$.

**Example 18** *Given the traditional* `append/3` *procedure and* $\beta_l$ *being a binding-time approximating terms of type* `list(T)` *that are instantiated at least up to a list skeleton,* $append$*,* $append(\beta_l)$ *and* $append(\bot, \beta_l)$ *are examples of binding-time closures of arity* $3$*,* $2$ *and* $1$ *respectively.*

[15] In fact, this is also a limitation of release 0.9 of the Mercury implementation [109].

In order to obtain a precise binding-time analysis, we approximate the value of a higher-order variable with a *set* of binding-time closures. A singleton set $\{c\}$ describes that the higher-order variable under consideration is, during specialisation, definitely bound to a closure that is approximated by $c$. In general, a set $\{c_1, \ldots, c_n\}$ describes that the higher-order variable under consideration is bound during specialisation to a closure that is approximated either by $c_1$, $c_2$,..., or $c_n$. To make this representation explicit, we alter the definition of the domain $\mathcal{B}$. Instead of containing only the values *static* and *dynamic*, we now include a value *static*$(S)$ with $S$ being a set of binding-time closures. Note that, if we define *dynamic* $>$ *static* as before and *static*$(S_1)$ $>$ *static*$(S_2)$ if and only if $S_1 \supseteq S_2$, $\mathcal{B}$ is still partially ordered. Since the binding-times now include higher-order binding-times, we alter the definition of the partial order relation on $\mathcal{BT}$:

**Definition 37.2** Let $\beta, \beta' \in \mathcal{BT}$ such that $dom(\beta) \subseteq dom(\beta')$ or $dom(\beta') \subseteq dom(\beta)$. We say that $\beta$ *covers* $\beta'$, denoted by $\beta \succeq \beta'$ if and only if $\forall \delta \in dom(\beta) \cap dom(\beta')$ holds that

- $\beta'(\delta) =$ *dynamic* implies $\beta(\delta) =$ *dynamic*, and
- $\beta'(\delta) =$ *static*$(S')$ implies $\beta(\delta) =$ *static*$(S)$ and $S \supseteq S'$.

Note that, with this new definition, the covers relation remains only defined between two binding-times that are derived from types that are instances of each other. In case of higher-order binding-times this means that both sets of binding-time closures contain closures of identical arity and argument types. Like before, we denote with $\mathcal{BT}^+$ the set $\mathcal{BT} \cup \{\top, \bot\}$, and $(\mathcal{BT}^+, \succeq)$ forms a complete lattice.

## 37.2 Higher-order binding-time analysis

We now reformulate the analysis from Section 36 such that it takes the higher-order constructs of Mercury into account. As a first observation, note that the binding-time constraints that are associated to first-order unifications and structured goals (see Figures 24 and 25) remain unchanged in the context of a higher-order analysis. To deal with higher-order constructions, we add an extra form of binding-time constraint to $\mathcal{BTC}$; namely a constraint of the form $X_\eta \succeq p(X_1, \ldots, X_k)$. The intended meaning is that the (higher-order) binding-time associated to $X$ at program point $\eta$ should at least contain a closure constructed from $p$ and the binding-times of its arguments at program point $\eta$. Formally, we extend the definition of a solution (Definition 36.3) such that for every constraint of the form $X_\eta \succeq p(X_{1_\eta}, \ldots, X_{k_\eta})$ it holds that

$$\sigma(X_\eta) \succeq static(\{p(\beta_1, \ldots, \beta_k)\}) \text{ where } \beta_i \succeq \sigma(X_{i_\eta}) \text{ for } 1 \leq i \leq k.$$

The main difference with the symbolic data flow analysis of Section 36 in a higher-order setting is that a set of constraints can no longer be associated to a procedure symbol (as in the semantic domain $Den$). Instead, in the higher-order analysis, we associate a set of binding-time constraints with a particular abstract call. Therefore, we define the analysis as an abstract semantics as before, but over the new semantic domain

$$Den_{cc} : Call \mapsto \wp(\mathcal{BTC}).$$

The notion of a procedure annotation of a procedure $p$ induced by a call $p(\beta_1, \ldots, \beta_n)$ is straightforwardly adapted for use with a denotation in $Den_{cc}$ rather than in $Den$. Moreover, given two such mappings $f, g \in Den_{cc}$, we define $f \cup g$ as a mapping in $Den_{cc}$ with $dom(f \cup g) = dom(f) \cup dom(g)$ and

$$\forall x \in dom(f \cup g) : (f \cup g)(x) = \begin{cases} f(x) \cup g(x) & \text{if } x \in dom(f) \cap dom(g) \\ f(x) & \text{if } x \in dom(f) \text{ and } x \notin dom(g) \\ g(x) & \text{if } x \in dom(g) \text{ and } x \notin dom(f) \end{cases}$$

The resulting analysis is a call-dependent analysis that is basically a combination of the call-independent and call-dependent analyses of Section 36.

**Definition 37.3**

The *higher-order semantics* has semantic domain

$$Den_{cc} : Call \mapsto \wp(\mathcal{BTC})$$

and semantic functions

$$\mathbf{P_{cc}} : Prog \mapsto Call \mapsto Den_{cc}$$

$$\mathbf{C_{cc}} : Proc \mapsto Call \mapsto Den_{cc} \mapsto Den_{cc}$$

$$\mathbf{G_{cc}} : Goal \mapsto Call \mapsto Den_{cc} \mapsto Den_{cc}$$

$$\mathbf{A_{cc}} : Atom \mapsto Call \mapsto Den_{cc} \mapsto Den_{cc}$$

defined in Figure 28.

Again, the meaning of a program is defined as a fixed point computation over the meaning of the individual procedures in the program given a binding-time abstraction of the call with respect to which the program must be specialised. Each procedure is analysed (by $\mathbf{C_{cc}}$) within the context of this initial call and a denotation (in $Den_{cc}$) representing the (possibly incomplete) results of analysis so far. The definition of $\mathbf{G_{cc}}$, defining the abstract meaning of a goal, is

$$\mathbf{P_{cc}}[\![P]\!]\pi = \text{lfp}(\bigcup_{p \in Proc(P)} \mathbf{C_{cc}}[\![p]\!]\pi)$$

$$\mathbf{C_{cc}}[\![p(X_1,\ldots,X_n) \leftarrow B]\!]\pi d = \bigcup_{p(\beta_1,\ldots,\beta_n) \in dom(d) \cup \{\pi\}} \mathbf{G_{cc}}[\![B]\!]p(\beta_1,\ldots,\beta_n)d$$

$$\begin{array}{lcl}
\mathbf{G_{cc}}[\![(G'_{\eta'}, G''_{\eta''})_\eta]\!]\pi d & = & \mathbf{G_{cc}}[\![G'_{\eta'}]\!]\pi d \cup \mathbf{G_{cc}}[\![G''_{\eta''}]\!]\pi d \cup \{(\pi, CC_{\text{conj}}(\eta,\eta',\eta''))\} \\
\mathbf{G_{cc}}[\![\mathit{not}_\eta(G_{\eta'})]\!]\pi d & = & \mathbf{G_{cc}}[\![G_{\eta'}]\!]\pi d \cup \{(\pi, CC_{\text{not}}(\eta,\eta'))\} \\
\mathbf{G_{cc}}[\![\mathit{if}_\eta\ G'_{\eta'}\ \mathit{then}\ G''_{\eta''}\ \mathit{else}\ G'''_{\eta'''}]\!]\pi d & = & \mathbf{G_{cc}}[\![G'_{\eta'}]\!]\pi d \cup \mathbf{G_{cc}}[\![G''_{\eta''}]\!]\pi d \cup \mathbf{G_{cc}}[\![G'''_{\eta'''}]\!]\pi d \\
 & & \cup\{(\pi, CC_{\text{if}}(\eta,\eta',\eta'',\eta'''))\} \\
\mathbf{G_{cc}}[\![(G'_{\eta'}; G''_{\eta''})_\eta]\!]\pi d & = & \mathbf{G_{cc}}[\![G'_{\eta'}]\!]\pi d \cup \mathbf{G_{cc}}[\![G''_{\eta''}]\!]\pi d \cup \{(\pi, CC_{\text{disj}}(\eta,\eta',\eta''))\} \\
\mathbf{G_{cc}}[\![A_\eta]\!]\pi d & = & \mathbf{A_{cc}}[\![A_\eta]\!]\pi d \cup \{(\pi, S)\} \\
 & & \text{where } S = \{X_\eta \succeq X_{\eta'} \mid X \in \texttt{in}(A), \eta' \in \texttt{reach}(X,\eta)\}
\end{array}$$

$$\begin{array}{lcl}
\mathbf{A_{cc}}[\![U]\!]\pi d & = & \{(\pi, \mathbf{A}[\![U]\!]d)\} \text{ for a first-order unification } U \\
\mathbf{A_{cc}}[\![X \Leftarrow p(X_1,\ldots,X_k)_\eta]\!]\pi d & = & \{(\pi, \{X_\eta \succeq p(X_1,\ldots,X_n) \sqcup \mathcal{C}_\eta, \mathcal{R}_\eta \succeq \bot\})\} \\
\mathbf{A_{cc}}[\![q(Y_1,\ldots,Y_n)_\eta]\!]\pi d & = & S_1 \cup S_2 \text{ where} \\
 & & \quad S_1 = \{(q(\beta_1,\ldots,\beta_n), \{\})\} \\
 & & \quad S_2 = \{(\pi, \rho(\text{proj}_{\mathit{Args}(q),\mathcal{R}_{\eta_b}}(d\, q(\beta_1,\ldots,\beta_n))))\} \\
 & & \quad \text{with } \beta_i = \sigma_\pi(Y_{i_\eta}) \\
\mathbf{A_{cc}}[\![X(Y_{k+1},\ldots,Y_n)_\eta]\!]\pi d & = & S_1 \cup S_2 \text{ where} \\
 & & \quad S_1 = \{(q(\beta_1,\ldots,\beta_n),\ \{\}) \mid q(\beta_1,\ldots,\beta_k) \in S\} \\
 & & \quad \text{where } \sigma_\pi(X_\eta) = \mathit{static}(S) \\
 & & \quad \text{and } \beta_i = \sigma_\pi(Y_i) \text{ for } k+1 \le i \le n \\
 & & \quad S_2 = \{(\pi, \bigcup_{\pi' \in dom(S_1)} \rho(\text{proj}_V(d\pi')))\} \\
 & & \quad \text{where } V = \mathit{Args}(p) \cup \{\mathcal{R}_{\eta_b}\}
\end{array}$$

Figure 28: The higher-order semantics

basically identical to the definition of $\mathbf{G}$ from Section 36, apart from the facts that (1) it threads a denotation as well as the abstract call to the procedure that is currently being analysed and (2) it associates this abstract call to the constraints for a particular goal. The same observations hold for the definition of $\mathbf{A_{cc}}$. The constraints derived for a first-order unification are identical to those derived by $\mathbf{A}$. A higher-order construction results in a constraint stating that the binding-time of the higher-order variable must contain at least the abstract closure created at this program point with the usual condition that the construction must not be under dynamic control. Being a construction, reduction can never result in code that might fail during exeuction, hence the (superfluous) constraint on $\mathcal{R}_\eta$.

Handling procedure calls is somewhat more involved than in the first-order case. Retrieving the constraints associated to a first-order call from the denotation now requires to compute the binding-times of the arguments in of the call. As before, $\sigma_\pi$ represents the procedure annotation induced by the call $\pi$. The binding-time variables in the resulting (projected) constraints are again renamed to the actual arguments of the call $X_1, \ldots, X_n$ and the control variable $\mathcal{R}_{\eta_b}$ is renamed to $\mathcal{R}_\eta$, as before. As for the other goals, the resulting constraints are associated to the abstract call $\pi$ for which the surrounding procedure is being analysed. The resulting mapping, in Figure 28 denoted by $S_2$, is updated with the mapping $\{(q(\beta_1, \ldots, \beta_n), \{\})\}$ in order to make sure that the call $q(\beta_1, \ldots, \beta_n)$ is in the domain of the newly constructed denotation, and hence will be analysed during a next round of the analysis. Note that the use of $\cup$ guarantees that if the call was already in the domain of the donation, the set of constraints associated to it remains unchanged. A higher-order call is basically handled as a set of first-order calls. First, the binding-time of the higher-order variable is retrieved from the procedure annotation $\sigma_\pi$ for the currently analysed procedure/call combination. If this binding-time equals *static*$(S)$, each closure $q(\beta_1, \ldots, \beta_k) \in S$ is transformed to a first-order call by adding $\sigma_\pi(X_{k+1}), \ldots, \sigma_\pi(X_n)$ to its arguments. From then on, the call is handled as a first-order call. The constraints associated to this call are retrieved from the denotation and added to the denotation under construction, and the call itself is added to the domain of the denotation under construction.

## 37.3 On the modularity of the approach

In a higher-order setting, the constraint generation phase of our binding-time analysis is a call dependent process. Indeed, the data flow dependencies in a procedure are determined by the closures contained in the procedure's call pattern. This suggests that the advantage of modularity, associated to the constraint based technique in a first-order setting, might no longer hold in a higher-order setting. However, to some extent the analysis can still be performed in a bottom-up,

modular way. Consider a module $M$ that exports the predicates $p_1, \ldots, p_n$. We can than compute

$$\bigcup_{p \in \{p_1,\ldots,p_n\}} \mathbf{P_{cc}}[\![P]\!]p(\top, \ldots, \top).$$

At first sight, it might seem strange to perform a call-dependent analysis with respect to an inital call in which all arguments are approximated by $\top$. However, recall that only the higher-order parts of the call patterns influence the resulting constraint systems. Hence, for those procedures that have no higher-order arguments, the constraint system derived by the call dependent analysis for a call $p(\top, \ldots, \top)$ equals the one derived by the call independent analysis of Section 36, and it can readily be used by other modules importing these procedures. Note that the call dependent nature of the process ensures that closure information that is constructed in a module $M$, is propagated inside $M$ itself. It is only if closure information is "lost" over a module boundary that the resulting analysis is less precise than a full call dependent analysis over the complete multi-module program. This is the case when, in some module, closure information is available in some arguments of a call to an imported procedure $p$ whereas, being imported, the constraints that are used for $p$ are those obtained by analysing $p(\top, \ldots, \top)$.

# 38 Example

In this section, we present an example, and use it to discuss to what extent the proposed analysis is also applicable in the context of Prolog.

## 38.1 A simple interpreter

Consider the simple interpreter for arithmetic expressions depicted in Figure 29. The program consists of a number of type definitions and two predicates. The type `env` defines an environment as a list of elements, each element being a pair (type `elem`) consisting of an identifier (type `ident`) and an integer (type `int`). We assume that the types `ident` and `int` are atomic and builtin. The type (`exp`) defines an expression as either a constant integer, a variable denoted by an identifier, or the sum of two expressions.

The predicate `lookup/3` takes an identifier and an environment as input, searches the value associated to the identifier in the environment en returns this value or fails. Note that the predicate is defined as being non-deterministic in order to mimick a purely declarative implementation in Prolog. The interpreter itself is represented by the predicate `int` which takes an expression and an environment as input and returns the value of the expression or fails. Both predicates are given in superhomogeneous form.

```
:- type env --> nil ; cons(elem, env).
:- type elem --> pair(ident,int).

:- type exp --> cst(int) ; var(ident) ; +(exp,exp).

:- pred lookup(ident, env, int).
:- mode lookup(in,in,out) is multi.

lookup(V,E,Val):- E⇒1 cons(A,As), A⇒2 pair(I,VI), (
   V==3 I, Val:=4 VI
   ;
   lookup(V,As,T)5, Val:=6 T).

:- pred int(exp,env,int).
:- mode int(in,in,out) is multi.

int(E,Env,R):-(
   E⇒1 cst(C), R:=2 C
   ;
   E⇒3 var(V), lookup(V,Env,Val)4, R:=5 Val
   ;
   E⇒6 +(A,B), int(A,Env,R1)7, int(B,Env,R2)8, plus(R1,R2,R)9).
```

Figure 29: A simple interpreter

After call-independent analysis, the binding-time constraints associated with the `lookup/3` predicate are as follows. All constraints are in normalised form. Where relevant, a binding-time variable is indexed by a subscript indicating the program point at which the constraint holds. Recall that the $\succeq$-constraints express the regular data flow, whereas the $\succeq^*$-constraints reflect the specialisation-strategy: a constraint $X \succeq^* Y^\delta$ denotes that the binding-time of $X$ cannot be static if the node $\delta$ in the binding-time of $Y$ is marked *dynamic*. Such a constraint is due to the presence, earlier in the predicate, of a deconstruction (or test) on $Y^\delta$ that may be residualised and subsequently fail at run-time.

$$
\begin{array}{l}
A \succeq E^{\langle(cons,1)\rangle} \\
As \succeq E^{\langle(cons,2)\rangle} \\
I \succeq E^{\langle(cons,1),(pair,1)\rangle} \\
I \succeq^* E \\
VI \succeq E^{\langle(cons,1),(pair,2)\rangle} \\
VI \succeq^* E \\
Val_4 \succeq E^{\langle(cons,1),(pair,2)\rangle} \\
Val_4 \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \sqcup E^{\langle(cons,1),(pair,1)\rangle} \sqcup V \\
T \succeq E^{\langle(cons,1),(pair,2)\rangle} \\
T \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \sqcup E^{\langle(cons,1),(pair,1)\rangle} \sqcup V \\
Val_6 \succeq E^{\langle(cons,1),(pair,2)\rangle} \\
Val_6 \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \sqcup E^{\langle(cons,1),(pair,1)\rangle} \sqcup V
\end{array}
$$

The interpretation of these constraints is as follows. The data-flow (or $\succeq$) constraints are obtained in a straightforward way, by projecting the constraints obtained from the unifications. The strategy (or $\succeq^*$) constraints are somewhat more involved. The constraints $I \succeq^* E$ and $VI \succeq^* E$ denote that $I$ and $VI$ must be $\top$ in case $E$ is not bound to an outermost functor. Indeed, if $E$ is not bound to an outermost functor, the deconstruction at program point 1 cannot be reduced at specialisation-time and the atom at program point 2 (in which $I$ and $VI$ are assigned their value) is under dynamic control and hence not to be reduced. Subsequently, the construction at program point 4 is under dynamic control if one of the preceeding atoms cannot be reduced or results in code that may fail at runtime, which is the case if either the environment $E$, the elements of the environment ($E^{\langle(cons,1)\rangle}$), the identifiers within each such element ($E^{\langle(cons,1),(pair,1)\rangle}$ or the variable $V$ is not bound to an outermost function. Similar considerations explain the $\succeq^*$ constraints on $T$ and $Val$ at program point 6 in the other branch of the disjunction. The constraints on $T$ are equal to the least upper bound of those (in the least fixed point) on $Val_4$ and $Val_6$. Recall that the constraints on $T$, which originate from the recursive call, are obtained from $T \succeq Val_4 \sqcup Val_6$.

The binding-time constraints derived for the `int/3` predicate are as follows.

$$
\begin{aligned}
&C \succeq E^{\langle(cst,1)\rangle}\\
&R_2 \succeq E^{\langle(cst,1)\rangle}\\
&R_2 \succeq^* E\\
&V \succeq E^{\langle(var,1)\rangle}\\
&Val \succeq Env^{\langle(cons,1),(pair,2)\rangle}\\
&Val \succeq^* Env \sqcup Env^{\langle(cons,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,1)\rangle} \sqcup E^{\langle(var,1)\rangle}\\
&R_5 \succeq Env^{\langle(cons,1),(pair,2)\rangle}\\
&R_5 \succeq^* E \sqcup Env \sqcup Env^{\langle(cons,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,1)\rangle} \sqcup E^{\langle(var,1)\rangle}\\
&A \succeq E^{\langle(+,1)\rangle}\\
&B \succeq E^{\langle(+,2)\rangle}\\
&R1 \succeq E^{\langle(+,1),(cst,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,2)\rangle}\\
&R1 \succeq^* E \sqcup E^{\langle(+,1)\rangle} \sqcup E^{\langle(+,2)\rangle} \sqcup E^{\langle(+,1),(var,1)\rangle} \sqcup E^{\langle(+,2),(var,1)\rangle} \sqcup\\
&\qquad Env \sqcup Env^{\langle(cons,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,1)\rangle}\\
&R2 \succeq E^{\langle(+,2),(cst,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,2)\rangle}\\
&R2 \succeq^* E \sqcup E^{\langle(+,1)\rangle} \sqcup E^{\langle(+,2)\rangle} \sqcup E^{\langle(+,1),(var,1)\rangle} \sqcup E^{\langle(+,2),(var,1)\rangle} \sqcup\\
&\qquad Env \sqcup Env^{\langle(cons,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,1)\rangle}\\
&R_9 \succeq E^{\langle(+,1),(cst,1)\rangle} \sqcup E^{\langle(+,2),(cst,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,2)\rangle}\\
&R_9 \succeq^* E \sqcup E^{\langle(+,1)\rangle} \sqcup E^{\langle(+,2)\rangle} \sqcup E^{\langle(+,1),(var,1)\rangle} \sqcup E^{\langle(+,2),(var,1)\rangle} \sqcup\\
&\qquad Env \sqcup Env^{\langle(cons,1)\rangle} \sqcup Env^{\langle(cons,1),(pair,1)\rangle}
\end{aligned}
$$

These constraints are obtained in a similar way as those for the `lookup` predicate.

Assume we want to specialise this program for the query

```
int(+(cst(2),+(var(x),cst(3))), [pair(y,Yval),(x,Xval)],Res)   (1)
```

i.e., the expression to compute is fully instantiated and the domain of the environment mapping is fully defined but the concrete values associated to the identifiers are as yet unknown. These degrees of instantiation are expressed by the binding-times $\beta_{exp}$ defined for the type `exp` and $\beta_{env}$ defined for the type `env`.

$$
\beta_{exp} = \left\{ \begin{array}{lll} (\langle\rangle, \mathit{static}), & (\langle(cst,1)\rangle, \mathit{static}), & (\langle(var,1)\rangle, \mathit{static})\\ (\langle(+,1)\rangle, \mathit{static}), & (\langle(+,2)\rangle, \mathit{static}) & \end{array} \right\}
$$

$$
\beta_{env} = \left\{ \begin{array}{l} (\langle\rangle, \mathit{static})\\ (\langle(cons,1),(pair,1)\rangle, \mathit{static})\\ (\langle(cons,1),(pair,2)\rangle, \mathit{dynamic}) \end{array} \right\}
$$

Note that the abstract call `int(`$\beta_{exp}$`,`$\beta_{env}$`,_)` will give rise to an abstract call `lookup(`*static*`,`$\beta_{env}$`,_)`. In the least solution of the constraints for `lookup` with respect to this call, we obtain that the output argument $Val = Val_4 \sqcup Val_6 =$ *dynamic*. However, the input to each test or deconstruction in `lookup` is at least bound to an outermost functor and hence is a candidate for reduction. In addition, if we look at the strategy constraints

$$\begin{aligned}
&\mathcal{C}_2 \succeq^* E\\
&\mathcal{C}_3 \succeq^* E \sqcup E^{\langle(cons,1)\rangle}\\
&\mathcal{C}_4 \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \sqcup V \sqcup E^{\langle(cons,1),(pair,1)\rangle}\\
&\mathcal{C}_5 \succeq^* E \sqcup E^{\langle(cons,1)\rangle}\\
&\mathcal{C}_6 \succeq^* E \sqcup E^{\langle(cons,1)\rangle} \sqcup V \sqcup E^{\langle(cons,1),(pair,1)\rangle}
\end{aligned}$$

we derive that none of the atoms is under dynamic control and consequently, each atom can be annotated as reducible.

Consequently, for the `int` predicate we obtain $R =$ *dynamic* but similarily to the case of the `lookup` predicate, none of the atoms is under dynamic control and the input to each unification is bound to at least an outermost constructor. Hence all unifications can be reduced. Only the predicate `plus`, which we assume builtin, has both input arguments *dynamic* and need to be residualised. The result of specialisation using the obtained annotations is the residual program
`int(Xval,Yval,Res):- plus(Xval,3,T), plus(2,T,Res).`

## 38.2 The Prolog case

The basic characteristic of Mercury that make this work feasible is the presence of type- and mode information. Hence, one may ask to what extent the technique can be carried over to the analysis of (pure) Prolog programs. Let us assume that the same type information as above is available. Given that the normal use of the `int/3` predicate is with mode `(i,i,o)`, a mode analysis is able to show that `lookup/3` is also called with mode `(i,i,o)` and that both predicates return a ground answer. Taking care that variables in output positions of predicates are first occurrences (hence free variables) one can obtain a normalisation that is almost a replica of the Mercury code.

```
lookup(V,E,Val):- E=cons(A,As), A=pair(I,VI), V=I, Val=VI.
lookup(V,E,Val):- E=cons(A,As), A=pair(I,VI), lookup(V,As,T), Val=T.

int(E,Env,R):- E = cts(C), R=C.
int(E,Env,R):- E = var(V), lookup(E,Env,Val), R=Val .
int(E,Env,R):- E = +(A,B), int(A,Env,R1), int(B,Env,R2), is+(R1,R2,R).
```

Using the mode information about the variables participating in unifications, one could classify them into tests, assignments, constructions and deconstructions as in the Mercury code. There is one difference. In the case of Mercury, assignments and constructions are guaranteed to succeed. In the case of our mode analysis, a variable not having mode input can still be partially intantiated, hence the unfication could fail at run-time. This will not happen in the example at hand. Indeed a simple local analysis shows that the variables being assigned are effectively free. E.g. in `Val=VI`, $Val$ is the first occurrence of the output variable. Whether a unification $\eta$ can fail has to be properly encoded in the special binding-time analysis variable $\mathcal{R}_\eta$. Apart from this, given the type information and the specification of the query to be specialised, the binding time analysis as done for Mercury can be performed, leading to the same annotations and hence, a specialiser as Logen[77] could derive the same specialised code.

Finally, it is feasible to handle more complex modes than simply input and output. In [15], a more refined mode analysis, called rigidity analysis is developed. Given a term $t$ of type $\tau$, it considers all subtypes $\tau'$ of $\tau$. The term is $\tau'$-rigid if it cannot have a well-typed instance that has a variable as a subterm of type $\tau'$. Such a type based rigidity analysis can provide more detailed mode information that has the potential to contribute to a better binding-time analysis. For example, such an analysis could show that a term of type `elem` (cnfr. the simple interpreter) that is not ground, is `ident`-rigid.

To conclude the discussion of this example, we note that — within the context of Prolog – the results obtained by the binding-time analysis could be directly fed to the LOGEN off-line partial deduction system [64, 77]. This system uses the notion of a *binding-type* to characterise specialisation-time values. Basic binding-types are *static* — characterising a value as ground — and *dynamic* – characterising a value as possibly non-ground – but more involved binding-types can be declared by the user using binding-type rules, much in the same way as types are declared by type rules.

In the interpreter example, the binding-times $\beta_{exp}$ and $\beta_{env}$ could be translated to the following binding-type definitions:

```
:- type exp ---> cst(static) ; var(static) ; +(static,static).

:- type elem --> pair(static,dynamic).

:- type env ---> nil ; cons(elem,env).
```

Input to the LOGEN system would then consist of the program in which every call is annotated as reducible (by means of the `unfold` annotation [64, 77]) together with the binding-type classification of the query `int(exp,env,dynamic)`.

# 39 Discussion

Constraint based (binding-time) analysis has been considered before. In [46], Henglein develops such a constraint-based (higher-order) binding-time analysis for $\lambda$-calculus by viewing the problem as a type inference problem for annotated $\lambda$-terms in a two-level $\lambda$-calculus. A set of constraints capturing local binding-time requirements is created and transformed into a normal form. A solver is used to find a consistent minimal binding-time classification. The analysis is redeveloped, concentrating on the aspect of polyvariance, for a PCF-like language in [48]. Henglein's analysis is scaled up by Bondorf and Jørgensen in [12], where they construct three (monovariant) analyses to be used in the partial evaluator Similix [10]. An important conceptual advantage, mentioned among others in [12], of doing binding-time analysis by constraint normalisation is the fact that the constraint based approach is viewed as a more *elegant* description of the analysis, compared with a direct abstract interpretation approach in which the source code is abstractly interpreted over the domain of binding-times. Indeed, in the constraint-based approach, problem and solution are separated: the constraint system expresses the binding-time *requirements* on the involved variables, whereas actual binding-times are contained in a *solution* to the constraint system. A practical consequence of this separation is that the data flow analysis, being performed at the symbolic level, needs to be performed only once for each predicate (in a first-order setting) rather than performing a separate analysis for every (abstract) call to the predicate. This result extends – at least to some extent – to a higher-order setting in the sense that the data flow analysis needs to be performed only once for each combination of a predicate with the closure information from its arguments.

In this work, we have shown that a constraint-based approach is also feasible for the logic programming language Mercury. The available type information allows to construct a precise domain of binding-times, whereas the available mode information allows to express the data flow constraints in a sufficiently precise way. Apart from being modular, the resulting analysis is polyvariant, and able to deal with partially instantiated data structures. Strongly related to our domain of binding-times is the domain proposed and used by Launchbury and Mogensen. Launchbury [69] defines a system of types and derives a finite domain of *projections* over each type. Such a projection maps a value to a part of the value that is definitely static, as such "blanking" out the dynamic part. In recent work [6, 7], a binding-time analysis is presented for the lambda calculus that allows an expression to be both static and dynamic at the same time; the general idea is to be able to access statically the (static) components of a residualised data structure. The exact relation and/or integration with a fine-grained domain of binding-times as employed by our technique is an interesting topic for further research.

Upgrading binding-time analysis to deal with Mercury's higher-order constructs requires clo-

sure information. In the literature, also closure analysis has been formulated by means of abstract interpretation [10, 18] as well as by constraint solving [45, 98, 47]. Bondorf and Jørgensen [12] develop a constraint-based flow analysis that traces higher-order flow as well as flow of constructed (first-order) values. In this work, we have combined closure analysis with binding-time analysis and used constraints to express the first-order as well as the higher-order data flow. We have enhanced the domain of binding-times to include a set of closures that represents the binding-time of a higher-order value, and formulated the constraint-generation phase as a call dependent process in which however only the higher-order parts of the call pattern determine the result of analysis. During constraint generation, the constraints involving higher-order values are evaluated, and the resulting closure information is used to decide what constraints to incorporate, possibly propagating closure information down into the called procedures.

We have discussed in detail how the analysis can be applied to multi-module programs according to a one module at a time scenario in Sections 36.5 and 37.3. If we do not wish to propagate closure information over module boundaries, the constraint generation phase can be performed one module at a time, bottom-up in the module hierarchy. Remaining issues are precisely such inter-module closure propagation and the handling of circularities in the module hierarchy. Recent work [16] presents a framework for the (call-dependent) analysis of multi-module programs that solves both problems. The key invariant in the approach of [16] is that at each stage of the process, the analysis results are correct, but reanalysis may – when more information is available – produce more accurate results. The analysis performs some extra bookkeeping such that, when a module is analysed, it records both the call patterns occurring in the calls to the imported procedures, and the analysis results of the module's exported procedures. When the recorded information contains new calls (or calls with a more accurate call pattern) to the imported modules, the analysis may decide to reanalyse the relevant imported modules with respect to the more accurate call patterns. Likewise, the recording of more accurate analysis results for a module's exported procedures can trigger the reanalysis of those modules that would possibly profit from these more accurate results. Note that our binding-time analysis neatly fits such an approach: initially, a module's exported procedures are analysed with respect to $\top$ (no closure information is available). The resulting binding-time constraint systems are correct, but could possibly be rendered more precise, when the procedures are (re)analysed with respect to a more accurate call pattern (one that *does* contain some closure information). To the best of our knowledge, the binding-time analysis of modular programs has been considered only occasionally before. Henglein and Mossin [48] note that a symbolic representation of binding-times allows a modular approach. Based on such a symbolic analysis, [26] present a method to specialise a multi-module program – written in a simple yet higher-order functional language – by constructing, for each of the modules, a generating extension, while using only

the result of a call-independent binding-time analysis. The analysis assumes that annotations indicating whether a function must be unfolded are given by hand and is restricted to module hierarchies without circular dependencies.

To summarise, we can state that few binding-time analyses have been developed that are polyvariant, deal with partially instantiated data, modules *and* higher-order constructs for a realistic language. Our binding-time analysis achieves this for the Mercury language by combining a number of known techniques: partially instantiated structures are dealt with by incorporating a structured and precise domain of binding-times, polyvariance and modularity are achieved by computing the binding-times symbolically and higher-order information is incorporated by propagating closure information during the symbolic phase of the analysis. Two important limitations of our technique are in the modularity of the approach, in particular the lack of propagation of closure information over module boundaries and the handling of circularities in the module dependency graph. Fortunately, both issues can be addressed by imposing a system like [16] on top of our technique.

# References


[1] S. M. Abramov and R. Glück. Semantics modifiers: an approach to non-standard semantics of programming languages. In M. Sato and Y. Toyama, editors, *Third Fuji International Symposium on Functional and Logic Programming*, page to appear. World Scientific, 1998.

[2] L. Andersen. Binding-time analysis and the taming of C pointers. In *PEPM93*, pages 47–58. ACM, 1993.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[4] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[5] K. R. Apt and F. Turini. *Meta-logics and Logic Programming*. MIT Press, 1995.

[6] K. Asai. Binding-time analysis for both static and dynamic expressions. In *Static Analysis Symposium*, pages 117–133, 1999.

[7] K. Asai. Binding-time analysis for both static and dynamic expressions. *New Generation Computing*, 20(1):27–52, 2001.

[8] L. Beckman, A. Haraldson, Ö. Oskarsson, and E. Sandewall. A partial evaluator and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.

[9] L. Birkedal and M. Welinder. Hand-writing program generator generators. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings, Proceedings of PLILP'91*, LNCS 844, pages 198–214, Madrid, Spain, 1994. Springer-Verlag.

[10] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[11] A. Bondorf, F. Frauendorf, and M. Richter. An experiment in automatic self-applicable partial evaluation of Prolog. Technical Report 335, Lehrstuhl Informatik V, University of Dortmund, 1990.

[12] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.

[13] A. F. Bowers and C. A. Gurr. Towards fast and declarative meta-programming. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 137–166. MIT Press, 1995.

[14] M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Programming Languages and Systems, Proc. of ESOP'98, part of ETAPS'98*, pages 27–41, Lisbon, Portugal, 1998. Springer-Verlag. LNCS 1381.

[15] M. Bruynooghe, W. Vanhoof, and M. Codish. Pos(T) : Analyzing dependencies in typed logic programs. In *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Revised Papers*, volume 2244 of *LNCS*, pages 406–420. Springer-Verlag, 2001. URL = http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=37386.

[16] F. Bueno, M. de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A model for inter-module analysis and optimizing compilation. In K. Lau, editor, *Preproceedings of LOPSTR 2000*, pages 64–71, 2000.

[17] M. Codish and C. Taboch. A semantic basic for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[18] C. Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 264–272. ACM, 1990.

[19] C. Consel et al. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 54–72. Springer-Verlag, 1996.

[20] T. Conway, F. Henderson, and Z. Somogyi. Code generation for Mercury. In J. Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 242–256, Cambridge, 1995. MIT Press.

[21] Y. Cosmadopoulos, M. Sergot, and R. W. Southwick. Data-driven transformation of meta-interpreters: A sketch. In H. Boley and M. M. Richter, editors, *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, volume 567 of *LNAI*, pages 301–308, Kaiserslautern, FRG, July 1991. Springer Verlag.

[22] S. Craig and M. Leuschel. Lix: An effective self-applicable partial evaluator for prolog. Submitted, Nov 2003.

[23] A. De Niel, E. Bevers, and K. De Vlaminck. Partial evaluation of polymorphically typed functional languages: The representation problem. In M. Billaud and et al., editors, *Analyse Statique en Programmation Équationelle, Fonctionelle, et Logique (Bigre, vol. 74)*, pages 90–97, October 1991.

[24] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.

[25] S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K. Sagonas. Termination analysis for tabled logic programming. In N. Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, pages 111–127, Leuven, Belgium, July 1998.

[26] D. Dussart, R. Heldal, and J. Hughes. Module-sensitive program specialisation. In *SIGPLAN '97 Conference on Programming Language Design and Implementation, June 1997, Las Vegas*, pages 206–214. ACM, 1997.

[27] F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In *Logic Based Program Synthesis and Transformation. Proceedings of Lopstr'2000*, LNCS 1207, pages 125–146, 2000.

[28] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying ctl properties of infinite-state systems by specializing constraint logic programs. In *Proceedings of VCL'2001*, Florence, Italy, September 2001.

[29] H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2 & 3):91–118, 1988.

[30] Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[31] Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[32] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[33] J. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.

[34] J. Gallagher and K. Henriksen. Abstract domains based on regular types. Technical Report ASAP Deliverable D4.5, Dept. of Computer Science, Roskilde University, Roskilde, Denmark, January 2004.

[35] J. P. Gallagher. A Program Transformation for Backwards Analysis of Logic Programs. In M. Bruynooghe, editor, *Pre-proceedings of the International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR 2003)*, volume CW 365 of *Katholieke Universiteit Leuven, Dept. of Computer Science, Technical Report*, pages 113–122, 2003.

[36] S. Genaim and M. Codish. Inferring termination conditions of logic programs by backwards analysis. In *International Conference on Logic for Programming, Artificial intelligence and reasoning*, volume 2250 of *Springer Lecture Notes in Artificial Intelligence*, pages 681–690, 2001.

[37] A. J. Glenstrup and N. D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, LNCS 1181, pages 273–284. Springer-Verlag, 1996.

[38] R. Glück. Jones optimality, binding-time improvements, and the strength of program specializers. In *Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, pages 9–19. ACM Press, 2002.

[39] R. Glück. On the generation of specialisers. *Journal of Functional Programming*, 4(4):499–514, 1994.

[40] R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. Swierstra and M. Hermenegildo, editors, *Programming Languages, Implementations, Logics and Programs (PLILP'95)*, LNCS 982, pages 259–278, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[41] R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10:113–158, 1997.

[42] C. Gomard and N. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[43] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[44] C. A. Gurr. Specialising the ground representation in the logic programming language Gödel. In Y. Deville, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'93*, Workshops in Computing, pages 124–140, Louvain-La-Neuve, Belgium, 1994. Springer-Verlag.

[45] N. Heintze. Set-based analysis of ML programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.

[46] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Springer-Verlag, 1991.

[47] F. Henglein. Simple closure analysis. Technical Report D-193, DIKU Semantics Report, 1992.

[48] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 287–301. Springer-Verlag, 1994.

[49] P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.

[50] C. K. Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.

[51] C. K. Holst. Finiteness analysis. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, LNCS 523, pages 473–495. Springer-Verlag, August 1991.

[52] C. K. Holst and J. Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

[53] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *PEPM97*, pages 63–73. ACM, 1997.

[54] D. Jacobs and A. Langen. Static analysis of logic programs for independent AND-parallelism. *Journal of Logic Programming*, 13(2 &3):291–314, May/July 1992.

[55] J. Jaffar, S. Michaylov, and R. H. C. Yap. A methodology for managing hard constraints in CLP systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 306–316, Toronto, Ontario, Canada, June 1991.

[56] N. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Springer-Verlag, 1985.

[57] N. D. Jones. Partial evaluation, self-application and types. In M. S. Paterson, editor, *Automata, Languages and Programming*, LNCS 443, pages 639–659. Springer-Verlag, 1990.

[58] N. D. Jones. What not to do when writing an interpreter for specialisation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 216–237, Schloß Dagstuhl, 1996. Springer-Verlag.

[59] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[60] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[61] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, LNCS 202, pages 124–140, Dijon, France, 1985. Springer-Verlag.

[62] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[63] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag.

[64] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings Dagstuhl Seminar on Partial Evaluation*, pages 238–262, Schloss Dagstuhl, Germany, 1996. Springer-Verlag, LNCS 1110.

[65] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 49–69. Springer-Verlag, 1992.

[66] V. Lagoon, F. Mesnard, and P. Stuckey. Termination analysis zith types is more accurate. In *19th International Conference on Logic Programming, ICLP*, LNCS. Springer-Verlag, 2003. To appear.

[67] A. Lakhotia and L. Sterling. How to control unfolding when specializing interpreters. *New Generation Computing*, 8:61–70, 1990.

[68] T. K. Lakshman and U. S. Reddy. Typed prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In K. Saraswat, Vijay; Ueda, editor, *Proceedings of the 1991 International Symposium on Logic Programming (ISLP'91)*, pages 202–220, San Diego, CA, 1991. MIT Press.

[69] J. Launchbury. Dependent sums express separation of binding times. In K. Davis and J. Hughes, editors, *Functional Programming, Glasgow, Scotland, 1989*, pages 238–253. Springer-Verlag, 1990.

[70] M. Leuschel. Partial evaluation of the "real thing". In L. Fribourg and F. Turini, editors, Logic Program Synthesis and Transformation — Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, LNCS 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.

[71] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.

[72] M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In T. Æ. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation - Essays dedicated to Neil Jones*, LNCS 2?56, pages 379–403. Springer-Verlag, 2002.

[73] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[74] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[75] M. Leuschel and D. De Schreye. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 253–263, La Jolla, California, June 1995. ACM Press.

[76] M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using a hand-written compiler generator. Technical Report DSSE-TR-99-6, Department of Electronics and Computer Science, University of Southampton, September 1999.

[77] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2002. URL = http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=38686.

[78] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.

[79] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[80] M. Leuschel, B. Martens, and K. Sagonas. Preserving termination of tabled logic programs while unfolding. In N. Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, pages 189–205, Leuven, Belgium, July 1998.

[81] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 83–103, Stockholm, Sweden, August 1996. Springer-Verlag.

[82] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5:133–154, 1987.

[83] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[84] J. W. Lloyd and R. W. Topor. Making PROLOG more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[85] H. Makholm. On Jones-optimal specialization for strongly typed languages. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, LNCS 1924, pages 129–148. Springer-Verlag, 2000.

[86] K. Marriott and P. Stuckey. The 3 r's of optimizing constraint logic programs: Refinement, removal, and reordering. In *Proceedings of POPL'93*, pages 334–344. ACM Press, 1993.

[87] K. G. Marriott and P. J. Stuckey. The 3 r's of optimizing constraint logic programs: refinement, removal and reordering. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 334–344. ACM Press, 1993.

[88] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K.U. Leuven, February 1994.

[89] B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995.

[90] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *The Journal of Logic Programming*, 22(1):47–99, 1995.

[91] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.

[92] F. Mesnard and S. Ruggieri. On proving left termination of constraint logic programs. *ACM Transactions on Computational Logic*, 2003. to appear.

[93] T. Mogensen. Binding Time Analysis for Polymorphically Typed Higher Order Languages. In J. Diaz and F. Orejas, editors, *TAPSOFT'89, Barcelona, Spain*, volume 352 of *LNCS*, pages 298–312. Springer-Verlag, 1989.

[94] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.

[95] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Proceedings LOPSTR'92*, pages 214–227. Springer-Verlag, Workshops in Computing Series, 1993.

[96] T. Æ. Mogensen. Separating binding times in language specifications. In *Proceedings of FPCA'89*, pages 12–25. ACM press, 1989.

[97] A. Mycroft and R. A. O'Keefe. A polymorphic type system for PROLOG. *Artificial Intelligence*, 23(3):295–307, 1984.

[98] J. Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, 1995.

[99] J. C. Peralta. *Analysis and Specialisation of Imperative Programs: An approach using CLP*. PhD thesis, Department of Computer Science, University of Bristol, 2000.

[100] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of clp programs. In M. Leuschel, editor, *Logic-based Program Synthesis and Transformation (LOPSTR'2002)*, LNCS 2664, pages 90–108, Madrid, Spain, September 2002. Springer-Verlag.

[101] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.

[102] S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[103] G. Puebla and M. Hermenegildo. Some issues in analysis and specialization of modular Ciao-Prolog programs. In M. Leuschel, editor, *Proceedings of the Workshop on Optimization and Implementation of Declarative Languages*, Las Cruces, 1999. In Electronic Notes in Theoretical Computer Science, Volume 30 Issue No.2, Elsevier Science.

[104] S. A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.

[105] S. Safra and E. Shapiro. Meta interpreters for real. In H.-J. Kugler, editor, *Proceedings of IFIP'86*, pages 271–278, 1986.

[106] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[107] D. A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In N. D. Jones and P. Hudak, editors, *ACM Symposium on Partial Evaluation*

*and Semantics-Based Program Manipulation*, pages 62–71. ACM Press Sigplan Notices 26(9), 1991.

[108] D. A. Smith and T. Hickey. Partial evaluation of a CLP language. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 119–138. MIT Press, 1990.

[109] Z. Somogyi et al. The Melbourne Mercury compiler, release 0.9.

[110] Z. Somogyi, F. Henderson, and T. Conway. The implementation of Mercury, an efficient purely declarative logic programming language. In *Proceedings of the ILPS'94 Postconference Workshop on Implementation Techniques for Logic Programming Languages*, 1994.

[111] Z. Somogyi, F. Henderson, and T. Conway. Logic programming for the real world. In *Proceedings of the ILPS'95 Postconference Workshop on Visions for the Future of Logic Programming*, 1995.

[112] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[113] Z. Somogyi, F. Henderson, T. Conway, A. Bromage, T. Dowd, D. Jeffery, P. Ross, P. Schachte, and S. Taylor. Status of the Mercury system. In *Proceedings of the JICSLP'96 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, 1996.

[114] L. Sterling and R. D. Beer. Metainterpreters for expert system construction. *The Journal of Logic Programming*, 6(1 & 2):163–178, 1989.

[115] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[116] W. Taha, H. Makholm, and J. Hughes. Tag elimination and Jones-optimality. In O. Danvy and A. Filinski, editors, *Programs as Data Objects, Second Symposium, PADO 2001*, LNCS 2053, pages 257–275, Aarhus, Denmark, May 2001. Springer-Verlag.

[117] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, 1986.

[118] Y. Tao, W. Grosky, and C. Liu. An Automatic Partial Deduction System for Constraint Logic Programs. In *9th International Conference on Tools with Artificial Intelligence (ICTAI '97)*, pages 149–157, Newport Beach, CA, USA, Nov. 1997. IEEE Computer Society.

[119] P. Thiemann. Cogen in six lines. In *International Conference on Functional Programming*, pages 180–189. ACM Press, 1996.

[120] W. Vanhoof and M. Bruynooghe. Binding-time annotations without binding-time analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, Proceedings*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 707–722. Springer-Verlag, 2001. URL = http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=36223.

[121] W. Vanhoof and M. Bruynooghe. When size does matter - Termination analysis for typed logic programs. In *Logic-based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001, Selected Papers*, volume 2372 of *Lecture Notes in Computer Science*, pages 129–147. Springer-Verlag, 2002. URL = http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=39276.

[122] W. Vanhoof, M. Bruynooghe, and M. Leuschel. Binding-time analysis for Mercury. submitted, 2003.

[123] W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'97*, LNCS 1463, pages 322–342, Leuven, Belgium, July 1997.