Specialising Interpreters using Offline Partial Deduction

Michael Leuschel[1], Stephen J. Craig[1],
Maurice Bruynooghe[2], and Wim Vanhoof[3]

[1] Department of Electronics and Computer Science
University of Southampton
`mal@ecs.soton.ac.uk`
[2] Department of Computer Science
Katholieke Universiteit Leuven
`Maurice.Bruynooghe@cs.kuleuven.ac.be`
[3] University of Namur
`wva@info.fundp.ac.be`

Abstract. 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 [21] 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 is 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 the interpreter, which is sometimes akin to a compiled version of the object program [10].

The ultimate goal in that setting is to achieve so-called *Jones optimality* [19, 21, 36], i.e., fully getting rid of a layer of interpretation (called the "optimality criterion" in [21]). More precisely, if we have a self-interpreter `sint` for a programming language L, i.e., an interpreter for L written in that same language L, and then specialise `sint` for a particular object program `p` we would like to obtain a specialised interpreter `p'` which is as least as efficient as `p` (see Figure 1).

The reason one uses a self-interpreter, rather than an interpreter in general, is so as to be able to directly compare the running times of `p` and `p'` (as they are written in the same programming language L).

More formally, if D is the input domain of `p` and $t_p(\mathtt{i})$ is the running time of the program p on the input `i`, we want that $\forall d \in D : t_{\mathtt{p'}}(\mathtt{d}) \leq t_{\mathtt{p}}(\mathtt{d})$.

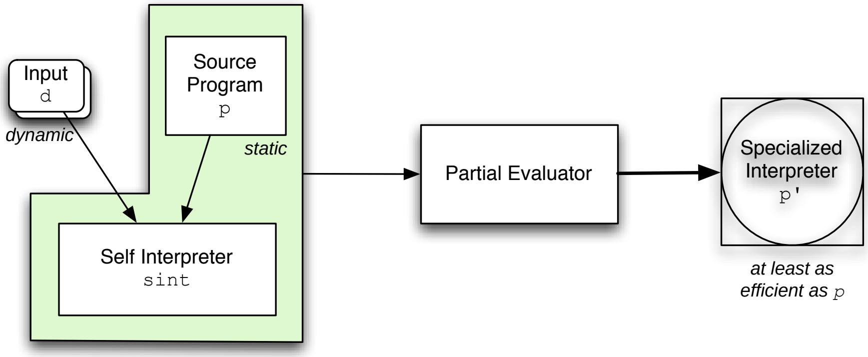

Fig. 1. Jones Optimality

In this paper 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 we will progress 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 `p` with none of its predicates being spyed on we will always get a specialised debugger equivalent to `p`. 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., [42, 47, 44, 5, 7, 26, 50, 28] 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 [50], 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 in this paper 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 offline approach is that it allows to steer the specialisation process far better than online techniques.

This steering is of particular importance in the current setting, since all of the previous research using automatic online techniques has shown that specialising interpreters (in general and especially Jones optimality) is hard to achieve.

The paper is structured as follows. In Section 2 we present the basics of offline partial evaluation and of the so-called cogen approach to specialisation employed by LOGEN. The LOGEN system itself is introduced in Section 2.3. In Section 3 we focus on offline techniques in logic programming as employed by LOGEN. We then show how a simple, non-recursive interpreter can be specialised in Section 4 before moving to a self-interpreter in Section 5, for which we achieve Jones-optimality. In Section 6 this self-interpreter is extended into a debugger, for which Jones-optimality is also achieved. Section 7 then presents more sophisticated features of LOGEN, required to tackle interpreters for other programming paradigms. Their use is illustrated in Section 8. Finally, we conclude in Section 9.

2 Offline Partial Evaluation and the Cogen Approach

2.1 Offline Specialisation

Inspired by the seminal work of Futamura [10], the functional partial evaluation community has put a lot of effort in developing self-applicable partial evaluators. The first successful self-application was reported in [22], and later refined in [23] (see also [21]). The main idea which made this self-application possible was to separate the specialisation process into two phases, as depicted in Figure 2:

- 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 *offline* 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. We refer to [22, 23, 21] for further details. In the context of logic programming languages the offline approach was used to achieve self-application in [39, 15] and more recently in [8].

2.2 The Cogen approach

Given a self-applicable partial evaluator, one can construct a so-called *compiler generator* (a *cogen* for short) using Futamura's third projection (see e.g. [21]). A *cogen* is a program that given a binding-time annotated program produces a specialiser for that program. If the annotated program is an interpreter, this specialiser can be viewed as a compiler, hence the name "compiler generator."

Obtaining an efficient cogen by self-application is a quite difficult task. This has led several researchers to pursue the so-called *cogen approach* to program specialisation [17, 18, 4, 1, 14, 48]. The idea behind this approach is to write the *cogen* directly by hand, rather than trying to obtain it by self-application. This

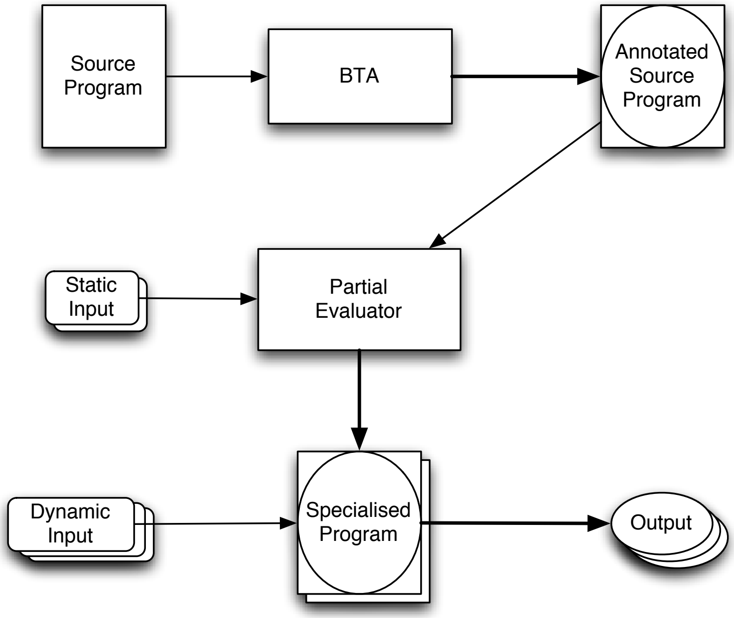

Fig. 2. Offline Partial Evaluation

turns out to be less difficult than one could imagine. Also, from a user's point of view, it is not important how a *cogen* was generated; what is important is that a *cogen* exists and that it is efficient and produces efficient, non-trivial specialised specialisers.

2.3 Overview of LOGEN

The application of the cogen approach in a logic programming setting has lead to the LOGEN system [24, 31], which we describe in more detail in the next section.

Figure 3 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.[1] This also allows an expert to inspect and manually refine the annotations to get better specialisation.

---

[1] We have developed a special LOGEN Emacs mode as well as a Tcl/Tk editor for this task. The figure 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.

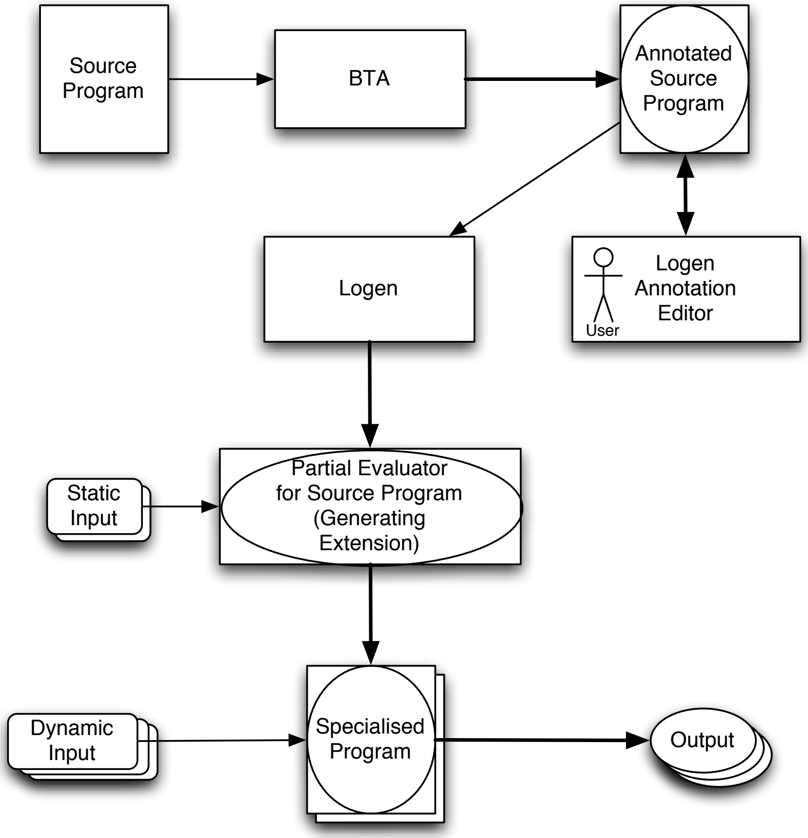

Fig. 3. Illustrating the LOGEN system and the *cogen* approach

- Second, LOGEN is run on the annotated source program and produces a specialiser for the source program, 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 describe the process of offline partial evaluation of logic programs and give a better understanding of how LOGEN specialises its source programs.

Throughout this paper, we suppose familiarity with basic notions in logic programming. We follow the notational conventions of [34]. In particular, in programs, we denote variables by strings starting with an upper-case symbol, while the notations for constants, functions and predicates begin with a lower-case character.

3.1 Partial Deduction

The term "partial deduction" has been introduced in [25] to replace the term partial evaluation in the context of pure logic programs (no side effects, no cuts). Though in some parts of the paper 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 4. The underlined atoms are called selected atoms. Here there is only one branch, and its computed answer is `R = [a,b,c]`.

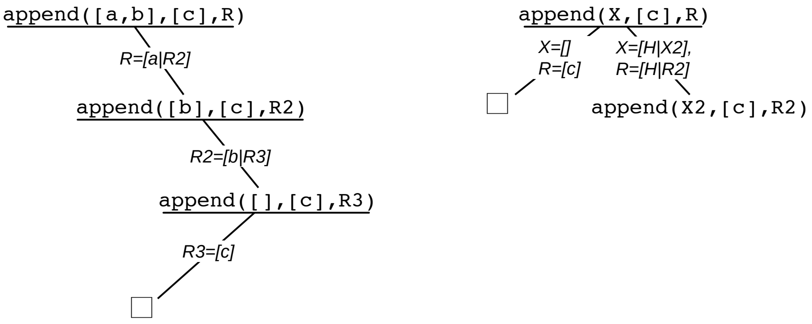

Fig. 4. Complete and Incomplete SLD-trees for the `append` program

Partial deduction builds upon this approach with two major differences:

- At some step in building the SLD-tree, it is possible *not* to select an atom, hence leaving a leaf with a non-empty goal. The motivation is that lack of the full input may cause the SLD-tree to have extra branches, in particular infinite ones. For example, in Figure 4 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. Incomplete branches do not produce computed answers, they produce conditional answers which can be expressed as program clauses by taking the resultants of the branches as defined below.
- Because of the atoms left in the leaves (in the bodies of the resultants), we may have to build a series of SLD-trees to ensure that every such atom is covered by some root of some tree. The fact that every leaf is an instance of a root is called *closedness* (sometimes also *coveredness*). In the example of Figure 4 the leaf atom `append(X2,[c],R2)` is already an instance of its root atom, hence closedness is already ensured and there is no need to build more trees.

Definition 1. *Let P be a program, $G = \leftarrow Q$ a goal, D a finite SLD-derivation of $P \cup \{G\}$ ending in $\leftarrow B$, and θ 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 4 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., are an instance of some atom in A. As we have seen, constructing a specialised program requires us to construct an SLD-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 [11, 38]):

- 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 [29]. In offline partial deduction the local control is hardwired, in the form of annotations added to the source program (either by the BTA, the user, or both). 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 specialiser works on an annotated version of the source program. In our approach, we use two kinds of annotations:

- *Filter declarations*, which declare which arguments to which predicates are static and which ones dynamic. This influences the global control only.
- *Clause annotations*, which indicate for every call in the body how that call should be treated during unfolding. This thus influences the local control only. For now, we assume that a call is either annotated by **memo** — indicating that it should not be unfolded – or by **unfold** — indicating that it should be unfolded. We introduce more annotations later on.

There is of course an interplay between these two kinds of annotations, and we return to this below.

First, let us consider as example an annotated version of the `append` program from above in which the filter declarations annotate the second argument as static while the others are dynamic and the clause annotations annotate the recursive call as **memo** to prevent its unfolding. 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 4 and produce the resultants above.

The following is a general algorithm for offline partial deduction given filter declarations and clause annotations.

Algorithm 3.1 (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 $A$ using the clause annotations in the annotated source program
   if a selected atom $S$ is annotated as memo then
      generalise $S$ into $S'$ by replacing all arguments declared as dynamic
                 by the filter declarations with a fresh variable
      if no variant of $S'$ is in $M$ then add it to $M$ end
   end
   pretty print the specialised clauses of $A$
until all atoms in $M$ are marked
```

In practice, renaming transformations [12] are also involved: Every atom in M is assigned a new predicate name, whose arity is the number of arguments declared 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 4 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 more precise picture, we present a Prolog version of the above algorithm. The code is runnable (using an implementation of `gensym`, see [45], to generate new predicate names). We assume that the filter declarations and clause annotations of the source program are represented by the definition of a `filter/2` and `rule/2` predicate respectively. We discuss a more user-friendly representation of these annotations in LOGEN later in the chapter.

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 the difference between `ResX`, `GenX` and `FX`. Consider for example the filter declaration for `app` given below with `X = app(S,[],S)` as call. The generalised call to be unfolded, `GenX` becomes `app(Y,[],Z)`; `FX`, the head of the specialised version becomes for example `app__0(Y,Z)` in which case the original call is to be replaced by `ResX = app__0(S,S)`.

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).
```

Let us now examine the behaviour of this specialiser for our earlier append example. First, we have to produce an annotated version of the append program:

```
/* the annotated source program: */
/* filter indicates how to generalise and filter */
filter(app(_,_,_),[dynamic,static,dynamic]).

/* rule annotates the clauses and indicates how to unfold */
rule(app([],L,L),call(true)).
rule(app([H|X],Y,[H|Z]),memo(app(X,Y,Z))).
```

Calling the specialiser with `memo(app(X,[c],Y))` produces the following specialised program as output:

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

The full treatment in LOGEN is a lot more complicated as LOGEN supports a more user friendly syntax as well as various features to be introduced in the next sections.

3.3 Local and global termination

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

- **Local nontermination:** The unfolding predicate `unfold/2` may fail to terminate or provide infinitely many answers.
- **Global nontermination:** 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]

---

[2] One often tries to ensure that a static argument is of so-called *bounded static variation* [21], so that global termination is guaranteed.

To overcome the first problem, we may have to annotate certain calls as **memo** rather than **unfold**. In the worst case, every call is annotated 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 declare 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 happens, we have to mark the offending call as **rescall** rather than **call**.

4 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 propositional variables. The $int(Prog, Env, Result)$ predicate takes two input arguments, the propositional formula and the environment containing a truth function for the propositional variables and produces the result. The environment is a list of truth 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).
```

As was indicated in Figure 3, the source program that serves as input for LOGEN needs annotations. The **filter** declaration declares how the arguments of the residual predicates have to be treated. The annotation **static** announces that the value of argument will be known at specialisation time; the annotation **dynamic** that the value of the argument will not necessarily be 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 that used in the previous section. For example, for the propositional interpreter 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**) while the environment will only be known at runtime (**dynamic**).

Next we must annotate the clauses in the original program to control the specialisation. This has to be done either manually by the user (possibly with the help of some annotation aware editor) or by an automatic binding-time analysis. The following constructs can be used to annotate the calls in the clause bodies of the 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** for built-ins or user defined predicates that should be fully evaluated without further intervention of the specialiser.
- **rescall** for calls to be kept as such in the 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 highlight 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 in the environment, let us first be cautious and mark the call to `lookup` as a **rescall**:
`int(var(X),Env, Z) :-` $\underbrace{\texttt{lookup(X,Env,Z)}}_{rescall}$.

Let us specialise the interpreter for the logical formula:
$((var(0) \lor (var(1) \land \neg var(2))) \lor false) \land true$. 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**, several instantiated occurrences appear in each resultant.

```
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-ins as **rescall** and the recursive call as **memo**, we obtain a specialised program containing `lookup__1/3`, a specialised version of `lookup/3`; however, the specialised version is but a renaming of the original as all its arguments where declared as dynamic:

```
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).
```

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:

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

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 a teaser, after declaring

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

one can specialise the interpreter for the call:

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

obtaining the following more efficient 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).
```

Indeed, the environment list has vanished and need not to be manipulated.

5 Specialising the Vanilla Self-Interpreter

5.1 Background

A classical benchmark for partial deduction has been the so-called *vanilla meta-interpreter* (see, e.g., [16, 3]). This interpreter is a self-interpreter because it can handle the language in which it is written. The following is the vanilla meta-interpreter, along with an encoding of the double-append object program:

```
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 deduction. Indeed, we would like to specialise the interpreter so as to obtain a residual program at least as efficient as 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 which is at least as efficient as:

```
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" [19, 36].

Online partial evaluators such as ECCE [32] or MIXTUS [43] 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 [37] (discussing the parsing problem) and the more recent [50] and [28] for more details. [50] 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 annotation **nonvar**. This means that the argument is not necessarily ground but has at least a top-level function symbol at specialisation time. When generalising the call, LOGEN keeps the top-level function symbol while replacing all its sub-arguments by fresh variables. Finally, these subarguments become arguments in the specialised version constructed by LOGEN.

A small example will help to illustrate this annotation:

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

Marking every call as **memo** (hence no unfolding), we obtain the following specialised program for the call p(f(Z,Z)). The first comment line indicates the renamings that LOGEN has performed.

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

If we mark the last call as **memo** and all others as **unfold**, we obtain:

```
%%% p(f(A,B)) :- p__0(A,B).
p__0(A,A).
p__0(A,A) :- p__0(a,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)`. Hence we 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 annotations which we introduce later in Section 7. This will give some improvements, but not full Jones optimality, due to the bad way in which solve is written.

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).
```

$$\texttt{solve(and(A,B)) :- }\underbrace{\texttt{solve(A)}}_{memo}\texttt{, }\underbrace{\texttt{solve(B)}}_{unfold}\texttt{.}$$

$$\texttt{solve(X) :- }\underbrace{\texttt{clause(X,Y)}}_{unfold}\texttt{, }\underbrace{\texttt{solve(Y)}}_{unfold}\texttt{.}$$

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 defined by facts. The calls `solve(B)` and `solve(Y)` can be unfolded as we know that `B` and `Y` are conjunctions. LOGEN will deconstruct the `and/2` and `empty/0` function symbols. However, as `solve(A)` is marked **memo**, the possibly recursive predicates of the object program are not unfolded.
- **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(`$t_1,\dots,t_n$`))` will be generalised into `solve(p(`$\_,\dots,\_$`))` 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.

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).
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**).

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 [50, 28]. (On the other hand, online techniques are capabable of

---

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

removing several layers of self-interpretation in one go. An offline approach will typically only be able to remove one layer at a time.)

- Second, it was also important to have sufficiently refined 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]
- Third, it is important that the meta-interpreter is written in such a way that the specialiser can distinguish between conjunctions and object level calls and can treat them differently.

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 marked as **unfold** while calls to `dsolve_atom` are marked as **memo**. The if-then-else is marked **call**, i.e., it will be executed at specialisation time. As far as the new predicates are concerned, all calls to `indent`

---

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

are marked **memo**, and all calls to `print` and `nl` are marked **rescall**. All other user defined predicate are marked as **unfold** and built-ins as **call**. Note that the above interpreter uses non-declarative predicates, and hence one has to be careful about "left-propagation" of bindings [43]. In our case, one has to be careful not to left-propagate bindings onto the first `print(H)` call, as this could change the observable behaviour of the debugger. LOGEN provides special annotations (such as `hide_nf`, see [31]) to prevent these problems. However, in our case we do not need those annotations as the call `dsolve_atom(H,s(Level),ToTrace)` is marked **memo** and hence will not generate any bindings that could affect `print(H)`.

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 [33] 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 [33] 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
```

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 number. Exact queries can be found in the DPPD library [27]. The FAR filtering [33] 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 in the paper 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

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 [35] 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 Annotations

So far we have come by with just three annotations for arguments in filter declarations: static, dynamic, and nonvar. The latter denotes a simple kind of so-called *partially static* data [21]. 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 annotate `lookup` with **unfold** because the environment is an open ended list at specialisation time.

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 [21].

One simple binding-time improvement for this particular problem is to define an auxiliary entry point as follows:

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

Now, we can annotate the calls to `int` and `lookup` with **unfold** and the calls to `is` with **rescall** and use the following filter declaration:

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

However, this solution only works because we can completely unfold the predicates `int` and `lookup`. Hence, this solution is rather ad-hoc and works only in special circumstances. For example, if the object language supports recursive procedures, this will not work.

A more principled solution, is to apply a binding-time improvement sometimes called *bifurcation* [9, 40]. 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).
```

One can annotate now all calls to `int` and `lookup` with **unfold**. It is even possible to annotate calls to `int` or to `lookup(V,E,E2,R)` as **memo** without loosing much specialisation as one part of the split environment is static and still available when specialising `lookup`.

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]`).

LOGEN provides a better way of solving this problem by allowing its users to define their own annotations using what we will call binding-types. For the interpreter above we would like to be able to define a custom annotation 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 present a polished version of the notion of a *binding-type* as introduced in [31] in order to characterise partially instantiated specialisation-time values in a more precise way. Like a traditional type in logic programming [2], 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, the underlying type system is different from the one of Mercury used in [49] for developing binding-types where the right hand side of a rule consists of a number of alternatives of the form $f(\tau_1, \ldots, \tau_k)$ with f a function symbol and the τ_i types. The LOGEN user has to cope with untyped Prolog programs and his interest is not in well-typing them but in concisely expressing the relevant binding-types. Hence LOGEN allows for union types and for function symbols anywhere in the names of types and in the right hand side of type rules. To distinguish between function symbols and type constructors, a wrapper `type/1` is used for the latter. The wrapper is ommitted for the predefined binding-types *static/0*, *dynamic/0*, *nonvar/0*, and *list/1*. Formally, a type is inductively defined as follows:

Definition 2. *The set of types is the least set defined by the following rules:*

- *A type variable is a type.*
- `static`*,* `dynamic`*, and* `nonvar` *are types.*
- *If* `t` *is a type then* `list(t)` *is a type.*
- *If* `c/n` *is a type constructor different from* `static`*,* `dynamic`*,* `nonvar` *and* `list/1` *and* $\tau_1, \ldots, \tau_n$ *are types then* `type(c(`$\tau_1, \ldots, \tau_n$`))` *is a type.*
- *If* `f/n` *is a function symbol and* $\tau_1, \ldots, \tau_n$ *are types then* `f(`$\tau_1, \ldots, \tau_n$`)` *is a type.*

As user programs may use the predefined binding-types as function symbols, the need could arise to refer to these function symbols in a binding type. Therefore, LOGEN also provides a wrapper `term/1`. For example, `term(static)` is the type denoting the singleton set with the function symbol *static* and not the binding-type `static`. To keep the exposition simple, we have not included the term wrapper in the above definition of types and we will ommit it entirely in what follows.

The set of terms denoted by a type of the form $\texttt{f(}\tau_1, \ldots, \tau_n\texttt{)}$ are all the terms of the form $f(t_1, \ldots, tn)$ with for all i: $t_i \in \tau_i$. For types of the form $\texttt{type(c(}\tau_1, \ldots, \tau_n\texttt{))}$, the denotation has to be defined by a type rule.

Definition 3. *A* type rule *for a type constructor c of arity n is of the form:*

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

with $k \geq 1$, $n \geq 0$ and where $V_1, \ldots, V_n$ are distinct type variables, and $\tau_1, \ldots, \tau_k$ are distinct types. Any type variable occurring in the right hand side must occur also in the left hand side. A set of type rules is a type definition*.*

With $n = 0$, a type rule defines a monomorphic or ground type, with $n > 0$, the type is polymorhic and the type rule defines the denotation for every type instance of the polymorphic type. For example the type rule corresponding with the predefined type `list(V)` is:

```
:- type list(V) ---> [ ] ; [V | list(V)].
```

Every type $\texttt{type(c(}\tau_1, \ldots, \tau_n\texttt{))}$ used in the annotations of LOGEN's input must be defined, i.e., there must be a type rule with left hand side $\texttt{c(}V_1, \ldots, V_n\texttt{)}$ and, for all types $\texttt{type(}\tau\texttt{)}$ occurring in the right hand side of the type rule, the type $\texttt{type(}\tau\{V_1/\tau_1, \ldots, V_n/\tau_n\}\texttt{)}$ must be defined.

Now we can formally define the denotations of types:

Definition 4. $[[\tau]]$*, the set of terms denotated by a type τ is defined as follows:*

- $[[dynamic]] = \{t \mid t \text{ is a term}\}$.
- $[[static]] = \{t \mid t \text{ is a ground term}\}$.
- $[[nonvar]] = \{t \mid t \text{ is a non-variable term}\}$.
- $[[type(c(\tau_1, \ldots, \tau_n))]] = \{t \mid t \in [[\tau]]$ *and there is a type rule of the form* $\texttt{:- type } c(V_1, \ldots, V_n) \texttt{ ---> } (\ldots; \tau; \ldots)$ *and* $t \in [[\tau\{V_1/\tau_1, \ldots, V_n/\tau_n\}]]$.
- $[[f(\tau_1, \ldots, \tau_n)]] = \{f(t_1, \ldots, t_n) \mid t_i \in [[\tau_i]] \text{ for all } i\}$.
- $[[list(\tau)]] = \{[\,]\} \cup \{[t_1 \mid t_2] \mid t_1 \in [[\tau]] \text{ and } t_2 \in [[list(\tau)]]\}$

Note that our definitions guarantee that types are downwards-closed (i.e., $t \in [[\tau]]$ implies $t\theta \in [[\tau]]$).

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

7.3 Using binding-types

The three basic binding types that are now used to control generalisation and filtering (the predicate `generalise_and_filter`) within the offline partial deduction algorithm of Section 3.2 are as follows:

- An argument marked as **dynamic** is replaced by a fresh variable and there will be a corresponding argument in the residual predicate.
- An argument marked as **static** is not generalised, and there will be no corresponding argument in the residual predicate.
- The top-level function symbol of an argument marked as **nonvar** will be kept, while all of its arguments are replaced by fresh variables. There will be one argument in the residual predicate for each argument of the top-level function symbol.
- An argument marked as `f`$(\tau_1, \ldots, \tau_n)$ is basically 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 τ_i.
- For an argument marked as `type(c(`$\tau_1, \ldots, \tau_n$`))` the type rule of c will be looked at and the argument will be treated according to the body of the rule. For disjunctions like τ_1 `;` τ_2 the algorithm will first attempt to apply τ_1, and if that is not successful it will apply τ_2.

For example, given the declaration `:- filter p(static,dynamic,nonvar).` the call `p(a,[b],f(c,d))` is generalised into `p(a,_,f(_,_))` and the residual version of the call is of the form `p__1([b],c,d)`. Given the declaration " `:- filter p(static,dynamic,f(static,dynamic)).`" the call is generalised into `p(a,_,f(c,_))` and the residual version is of the form `p__2([b],d)`. Finally, using "`:- filter p(static,list(dynamic), static).`" as filter declaration, the same call is generalised into `p(a,[_],f(c,d))` with the residual version being of the form `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).
```

Given these filter declarations, we can now annotate the clause bodies as follows:

`int(cst(C),_E,C).`

`int(var(V),E,R) :-` $\underbrace{\mathtt{lookup(V,E,R)}}_{unfold}$.

`int(+(A,B),E,R) :-` $\underbrace{\mathtt{int(A,E,Ra)}}_{unfold}$, $\underbrace{\mathtt{int(B,E,Rb)}}_{unfold}$, $\underbrace{\mathtt{RisRa} + \mathtt{Rb}}_{rescall}$.

```
lookup(V,[(V,Val)|_T],Val).
```

$$\texttt{lookup(V,[(\_Var,\_)|T],Res) :- }\underbrace{\texttt{lookup(V,T,Res)}}_{unfold}.$$

While these annotations and types were derived by hand, we believe that it is possible to derive them automatically. One approach is to adapt the polymorphic binding-time analysis for Mercury presented in a companion chapter [49] of this book. For more details see [49]. A fully automatic monomorphic binding-time analysis, refining earlier work in [6, 31] is currently being implemented within the EU-funded project ASAP (see `http://clip.dia.fi.upm.es/Projects/ASAP/`).

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 results in 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 [31]. Another alternative is to binding-time improve the program by inserting an explicit if-statement, changing the 3rd clause of the interpreter as follows:

$$\texttt{int(+(A,B),E,E2,R) :- }\underbrace{\texttt{int(A,E,E2,Ra)}}_{unfold},\ \underbrace{\texttt{int(B,E,E2,Ra)}}_{unfold},$$

$$(\ \underbrace{\texttt{ground((Ra,Rb))}}_{call}\ \texttt{->}\ \underbrace{\texttt{R is Ra + Rb}}_{call}\ \ ;\ \ \underbrace{\texttt{R is Ra + Rb}}_{rescall}\ ).$$

where the if-statement itself is marked **call** and executed 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 specialising 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)).
```

Given these filter declarations, we can mark the calls `solve(A)`, `solve(B)` and `clause(X,Y` as **unfold**, and mark the call `solve(Y)` as **memo**. 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 shown below. 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).
```

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 up to the binding time analysis 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. Details about handling the cut in an online specialiser can be found in [41, 43].

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 it will treat the arguments as dynamic (the second alternative in the type `result`) when they do not.

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 when including the specialisation time, i.e., the time to run LOGEN and the generating extension, 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, the experiments described below indicate that speedups are getting bigger for more complicated object programs with more functions and more arguments and variables. One reason being that more complicated object programs will have more variables, and hence looking up variable values in the list environment will get more and more expensive, whereas lookup in the specialised program will be basically a constant time operation (relevant variables are arguments of the specialised predicates). Indeed, the results of specialising the interpreter for the following slightly bigger functional program that has extra loop variables results in bigger speedups.

```
loop_fib === lambda(cur,let(cur1,$cur + &1, let(cur2, $cur1 + &1,
              let(cur3, $cur2 + &1,  if(($cur = &21),
                  (fib @ ($cur)),
                  (print(constr(fibonacci,[$cur,fib @ ($cur)]))
                    ->  (loop_fib @ ($cur1)))))))).
```

In the same table one can see figures for `loop_fib2`, `loop_fib3`, `loop_fib4`, `loop_fib5`, each with 3 more variables in the environment than its predecessor, but apart from that behaving identically to `loop_fib`. As can be seen, the specialised programs basically all run in the same time (60–70 ms), whereas the original interpreter runs considerably slower with more variables, increasing the speedup to 45 for `loop_fib5`.

Note that LOGEN has only to be run once for the `eval` interpreter; the same generating extension can then be used for specialising the interpreter with respect to any functional program. Similarly, the specialised code can then be used for any call to the given functional program.[5]

9 Discussion and Conclusion

Probably the most closely related work is [20] which treats untyped first-order functional languages, and gives a list of recommendations on how to write in-

---

[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.

Table 2. Specialising `eval` using LOGEN

function call	`eval` runtime	logen time	genex time	specialised runtime	speedup	speedup (incl. gx)	speedup (incl. gx,logen)
fib(24)	1050 ms	60 ms	15ms	75 ms	14.0	11.7	7
loop_fib(0)	1430 ms	60 ms	30ms	60 ms	23.8	15.9	9.5
loop_fib2(0)	1940 ms	60 ms	40ms	60 ms	32.3	19.4	12.1
loop_fib3(0)	2460 ms	60 ms	50ms	60 ms	41.0	22.4	14.5
loop_fib4(0)	2540 ms	60 ms	50ms	70 ms	36.3	21.2	14.1
loop_fib5(0)	3150 ms	60 ms	60ms	70 ms	45.0	24.2	16.6

terpreters that specialise well. Even though [20] does of course not address the specific issues that arise when specialising logic programming interpreters, many points raised in [20] are also valid in the logic programming setting. For example, [20] suggests that you should "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. [20] 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 [20].

We have already discussed related work in the logic programming community [42, 47, 44, 5, 7, 26, 50, 28]. In the functional community there has been a lot of recent interest in Jones optimality; see [19, 36, 46, 13]. For example, [13] 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 [49] 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 [8], 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 a 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 [30], and then compiling high-level B specifications into Prolog code for fast animation and verification.

Acknowledgements

We would like to thank Mauricio Varea for helping us out with some of the figures and for valuable discussions. We are also very grateful to Armin Rigo for developing the initial Emacs mode. We also thank the ASAP project participants for their stimulating feedback and their help in adapting LOGEN for Ciao-Prolog. Finally, we thank the reviewers for their thoughtful comments.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
2. 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.
3. K. R. Apt and F. Turini. *Meta-logics and Logic Programming*. MIT Press, 1995.
4. 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.
5. 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.
6. M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Proceedings of the European Symposium on Programming (ESOP'98)*, LNCS 1381, pages 27–41. Springer-Verlag, April 1998.
7. 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.
8. S. Craig and M. Leuschel. Lix: An effective self-applicable partial evaluator for Prolog. In *Proceedings of FLOPS'04*, April 2004. To appear.
9. 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.
10. Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
11. J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

12. J. Gallagher and M. Bruynooghe. Some low-level transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 229–244, Leuven, Belgium, 1990.
13. 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.
14. 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.
15. 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.
16. 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.
17. C. K. Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.
18. 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.
19. 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.
20. 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.
21. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
22. 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.
23. 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.
24. 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.
25. J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 49–69. Springer-Verlag, 1992.
26. A. Lakhotia and L. Sterling. How to control unfolding when specializing interpreters. *New Generation Computing*, 8:61–70, 1990.
27. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
28. 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.

29. 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.
30. 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.
31. 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.
32. 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.
33. 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.
34. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
35. J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.
36. 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.
37. B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming.* PhD thesis, K.U. Leuven, February 1994.
38. 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.
39. 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.
40. T. Æ. Mogensen. Separating binding times in language specifications. In *Proceedings of FPCA'89*, pages 12–25. ACM press, 1989.
41. S. Prestwich. An unfold rule for full Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, Workshops in Computing, pages 199–213, University of Manchester, 1992. Springer-Verlag.
42. S. Safra and E. Shapiro. Meta interpreters for real. In H.-J. Kugler, editor, *Proceedings of IFIP'86*, pages 271–278, 1986.
43. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
44. L. Sterling and R. D. Beer. Metainterpreters for expert system construction. *The Journal of Logic Programming*, 6(1 & 2):163–178, 1989.
45. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.
46. 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.

47. 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.
48. P. Thiemann. Cogen in six lines. In *International Conference on Functional Programming*, pages 180–189. ACM Press, 1996.
49. W. Vanhoof, M. Bruynooghe, and M. Leuschel. Binding-time analysis for Mercury. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, LNCS this Volume. Springer-Verlag, 2004.
50. 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.