Specializing Interpreters using Offline Partial Deduction

Michael Leuschel, Stephen Craig, Maurice Bruynooghe and Wim Vanhoof
mal@ecs.soton.ac.uk, maurice@cs.kuleuven.ac.be, wva@info.fundp.ac.be

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2003-5
November 2003

http://www.dsse.ecs.soton.ac.uk/techreports

School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, United Kingdom

Specializing 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@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 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 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* [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 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.

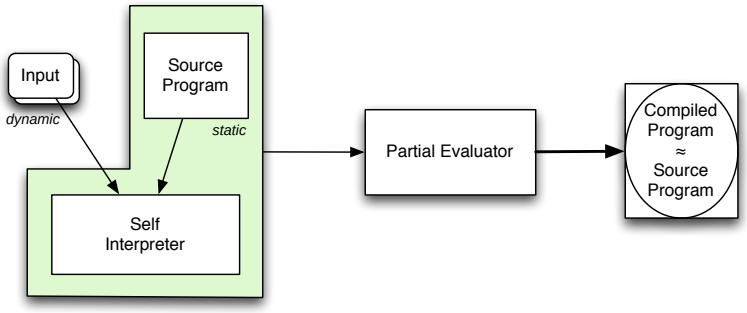

Fig. 1. Jones Optimality

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 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., [41, 46, 43, 5, 6, 26, 49, 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 [49], 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 off-line 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 on-line 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 described in Section 2.3. 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 in Section 8. Finally, we conclude in Section 9.

2 Offline Partial Evaluation and the Cogen Approach

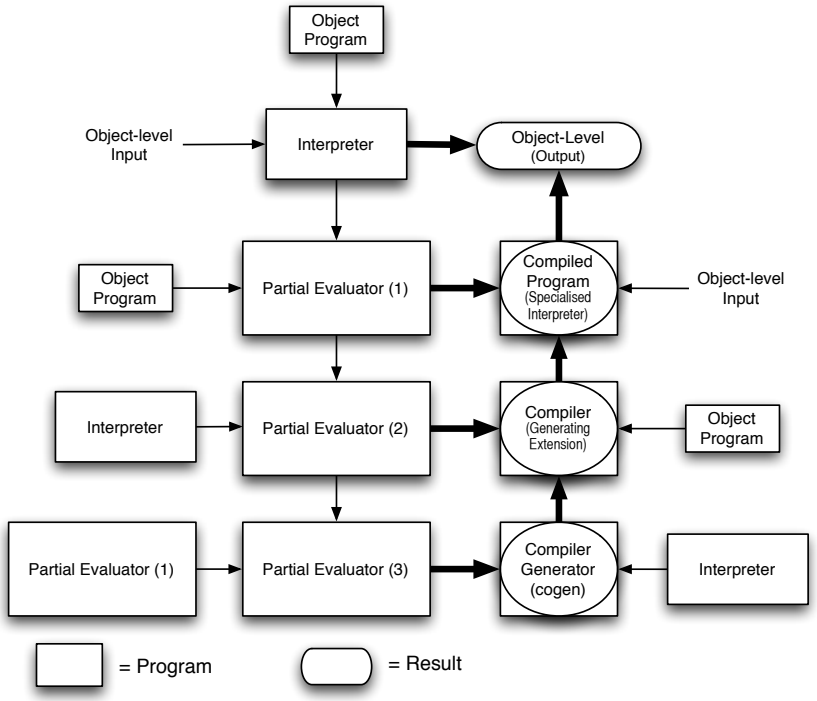

Fig. 2. Illustrating the three Futamura projections

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

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

manifold. The most well-known lie with the so called second and third *Futamura projections* [9]. 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 [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 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 [22, 23, 21] for further details. In the context of logic programming languages the off-line approach was used in [39] and to some extent also in [14].

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* [16, 18, 4, 1, 12, 47] (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 [24, 31], which we describe in more detail in the next section.

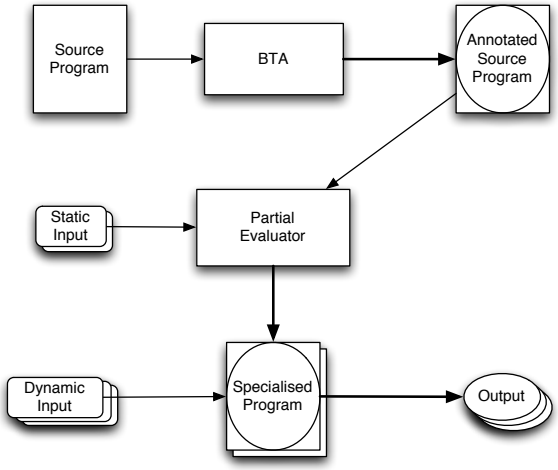

Fig. 3. Offline Partial Evaluation

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).

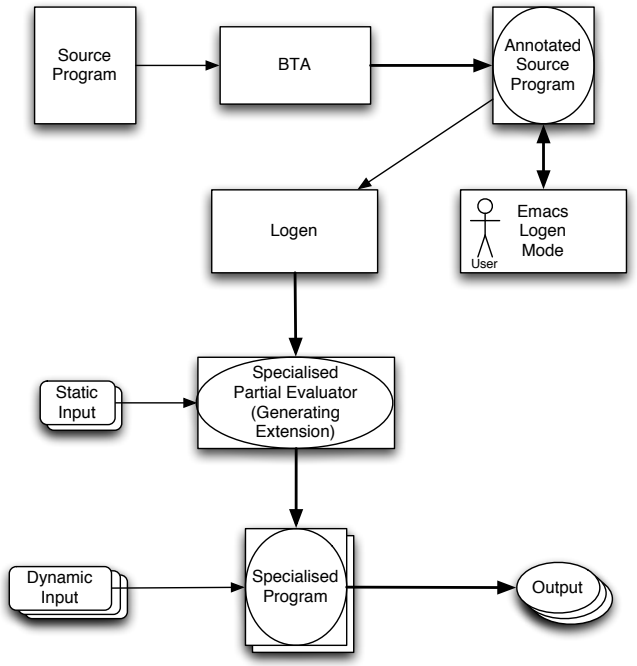

Fig. 4. Illustrating the LOGEN system and the *cogen* approach

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 paper, we suppose familiarity with basic notions in logic programming. We follow the notational conventions of [34]. 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 [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 5. The underlined atoms are called selected atoms. Here there is only one branch, and its computed answer is `R = [a,b,c]`.

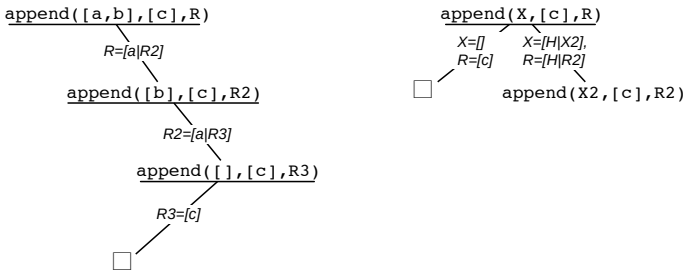

Fig. 5. Complete and Incomplete SLD-trees for the `append` program

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 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 θ the composition of the* mgus *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 [10, 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 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 3.1 (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 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 [44], 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`.

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).
```

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

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

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 for the call `int(and(or(or(var(0),and(not(var(1)),var(2))),false),true), [A,B,C],D)`, which is more efficient as the environment list has vanished and no longer needs to be inspected:

```
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).
```

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., [15, 3]), 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" [19, 36].

Online partial evaluators such as ECCE [32] or MIXTUS [42] 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 [49] and [28] for more details. [49] 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 whiich 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).
solve(and(A,B)) :- solve(A), solve(B).
                   memo      unfold
solve(X) :- clause(X,Y), solve(Y).
            unfold       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.

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))).
```

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

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 [49, 28]. (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]

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).
```

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

```
:- 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 [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 numbers. 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).

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).

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

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 [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 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 [21].

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 *bifurcation* [8, 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).
```

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 [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, 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(`σ`)` where $\sigma = f(\tau_1, \ldots, \tau_n)$ and f is a function symbol of arity $n \geq 0$ and τ_i are types,
- or a term of the form `type(`τ`)` where τ 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 2. *A* type definition *for a type constructor c of arity n is of the form:*

`:- type` $c(V_1, \ldots, V_n)$ `---> (`τ_1 `;` $\ldots$ `;` τ_k`).`

with $k \geq 1, n$ and where $V_1, \ldots, V_n$ are distinct type variables, and τ_i are types which only contain type variables in $\{V_1, \ldots, V_n\}$.
A type system *Γ 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 Γ 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(`τ`)` where τ is a type. This is equivalent to `type(list(`τ`))`, 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 Γ 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 τ_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 3. *We now define* type judgements *relating terms to types in the type system Γ.*
– 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` $c(\tau'_1, \ldots, \tau'_k)$`--> (...;`$\tau$`;...)` *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 τ_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 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))` 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 [48]. For more details see [48].

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 [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:

```
int(+(A,B),E,E2,R) :- int(A,E,E2,Ra), int(B,E,E2,Ra),
                      ( ground((Ra,Rb)) -> R is Ra + Rb ; R is Ra + Rb ).
```

(unfold, unfold, call, call, 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).
```

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.

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.

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 [20] which treats untype first-order functional languages, and gives a list of recommendations on how to write interpreters 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 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. [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 discuss related work in the logic programming community [41, 46, 43, 5, 6, 26, 49, 28]. In the functional community there has been a lot of recent interest in Jones optimality; see [19, 36, 45, 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 [48] 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,

[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.

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 [7], 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 [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. Finally, we thank the ASAP project participants for their stimulating feedback and their help in adapting LOGEN for Ciao-Prolog.

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. 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.
7. S. Craig and M. Leuschel. Lix: An effective self-applicable partial evaluator for prolog. Submitted, Nov 2003.

8. 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.
9. Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
10. 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.
11. 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.
12. 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.
13. R. Glck. 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. 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.
15. 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.
16. C. K. Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.
17. 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.
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. Safra and E. Shapiro. Meta interpreters for real. In H.-J. Kugler, editor, *Proceedings of IFIP'86*, pages 271–278, 1986.
42. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

43. L. Sterling and R. D. Beer. Metainterpreters for expert system construction. *The Journal of Logic Programming*, 6(1 & 2):163–178, 1989.
44. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.
45. 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.
46. 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.
47. P. Thiemann. Cogen in six lines. In *International Conference on Functional Programming*, pages 180–189. ACM Press, 1996.
48. W. Vanhoof, M. Bruynooghe, and M. Leuschel. Binding-time analysis for Mercury. submitted, 2003.
49. 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.