UNIVERSITY OF SOUTHAMPTON

Practicable Prolog Specialisation

by

Stephen-John Craig

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

June 2005

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Stephen-John Craig

In software development an emphasis is placed on creating reusable general programs which solve a wide class of problems, however it is a struggle to balance generality with efficiency. Highly parametrised modular code is reusable but suffers a penalty in terms of efficiency, in contrast carefully optimising the code by hand produces faster programs which are less general and have fewer opportunities for reuse. Partial evaluation is an automatic technique for program optimisation that optimises programs by exploiting known data.

While partial evaluation is improving, the uptake by mainstream users is disappointing. The aim of this thesis is to make partial evaluation accessible to a wider audience. A basic partial evaluation algorithm is given and then extended to handle the features encountered in real life Prolog implementations including constraint logic programming, coroutining and non-declarative constructs. Offline partial evaluation methods rely on an annotated version of the source program to control the specialisation process. A graphical development environment for specialising logic programs is presented allowing users to create, visualise and modify their annotated source programs.

An algorithm for automatically generating annotations is given using state of the art termination analysis, combined with type-based abstract interpretation for propagating the binding types. The algorithm has been fully implemented and we report on performance of the process on a series of benchmarks. In addition to an algorithm for generating a safe set of annotations we also investigate the generation of optimal annotations. A self-tuning system, which derives its own specialisation control for the particular Prolog compiler and architecture by trial and error is developed. The system balances the desire for faster code against code explosion and specialisation time.

Additionally it is demonstrated that the developed partial evaluator is self-applicable. The attempts to self-apply partial evaluators for logic programs have, of yet, not been all that successful. Compared to earlier attempts, the system is effective and surprisingly simple. The power and efficiency of the implementation is evaluated using the specialisation of a series of non-trivial interpreters.

Contents

List of Figures

List of Tables

Listings

Acknowledgements

There are many people who have contributed to this thesis in different ways. Without their help and support this thesis would not have been possible.

Firstly, I would like to thank Michael Leuschel for his guidance during the course of my PhD. He has offered advice throughout, both as a supervisor and a friend. Upon completion of this thesis I will set my targets on convincingly beating him at squash.

I have had the good fortune of being involved in the ASAP project. The meetings proved to be a valuable source of new research ideas. I would like to thank all of the partners in the project, in particular John Gallagher and Kim Henriksen from Roskilde, Andrew Moss and Henk Muller from Bristol and Germán Puebla and Manual Hermenegildo from Madrid.

Thanks to my friends for supporting me. Jonathan for going to the pub every lunchtime and the technical discussions over the pool table. John and Dan for proof reading my thesis, any errors that remain are my own. My colleges at DSSE for an interesting research environment, in particular Mauricio for having spare coffee whenever I ran out.

On a personal note I am indebted to my girlfriend, Emma, she has been a constant source of understanding and support. Finally, I would like to thank my family for their encouragement over the years.

Chapter 1

Introduction

The software engineer faces an endless struggle trying to balance efficiency and generality. Highly parameterised programs, using good modularisation offer benefits in the form of code reuse and maintainability at the price of efficiency. In contrast, optimising the code by hand for a specific case can produce fast efficient code but makes it difficult to develop and maintain.

1.1 Partial Evaluation

Program specialisation aims to improve the overall performance of programs through source to source transformations. This work focuses on a particular approach, known as *partial evaluation* (Jones et al., 1993), in which a program is transformed using partial information about the input of the program. To explain the concept of partial evaluation we go back to a simple model of program execution, shown in Figure 1.1. Program P takes two inputs, S and D, and when executed produces the output *Out*.

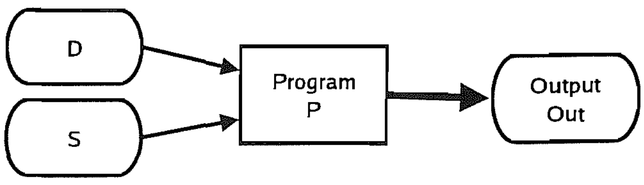


FIGURE 1.1: Simple program evaluation

Partial evaluation attempts to classify the input of a program into two main categories: data that will be known before execution, and data which will only be known at run-time. In Figure 1.2, a program P is specialised by fixing part of the input and then precomputing those parts of P that depend only on the known parts of the input. The obtained transformed program P' is less general than the original but can be much more efficient. The program P' is called the *residual* program. The part of the input that is fixed, in this case S, is referred to as the *static* input, while the remainder, labelled D, is the *dynamic* input. The residual program computes the same function as the original program, but naturally only for inputs with the same static data.

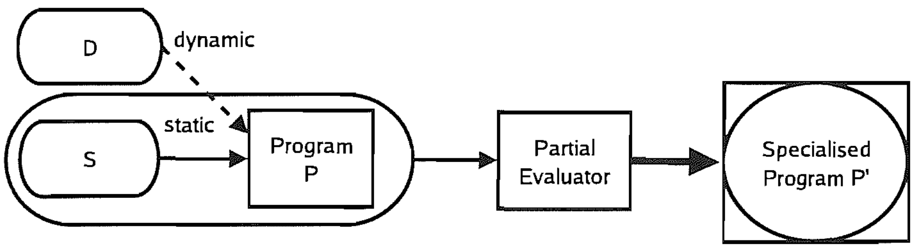


FIGURE 1.2: Partial Evaluation

The theoretical basis for program specialisation was first formulated and proven as Kleene's *s-m-n* theorem (Figure 1.1) over 50 years ago (Kleene, 1952). However, Kleene's constructions were interested in the theoretical issues of computability not efficiency, and produced specialised versions that were more complex than the originals. In contrast, partial evaluation aims to derive more efficient programs by exploiting the known static input.

Forall $f(x, y)$ there exists a primitive recursive function σ such that

$$f(x, y) = \sigma(f, x)(y) \tag{1.1}$$

Partial evaluation has received considerable attention over the past decade both in functional (e.g. Jones et al. (1993)), imperative (e.g. Andersen (1994); Peralta and Gallagher (1997)) and logic programming (e.g. Gallagher (1993); Komorowski (1992); Leuschel et al. (2004b); Pettorossi and Proietti (1994)).

The classic example from the literature (see e.g. Jones et al. (1993)) involves the *power* function shown in Figure 1.3. The recursive function *power* raises x to the y, if the power is even then the answer is the square of $x^{\frac{y}{2}}$, otherwise the answer is $x \times x^{y-1}$.

```
power(x,y) : if y=0 then 1
             else if even(y) then square(power(x,y/2))
             else x * power(x,y-1)
```

FIGURE 1.3: Function to compute x^y

If y is known before execution, i.e. it is *static*, we can partially evaluate *power* for a fixed y and a *dynamic* x. For $Y = 5$, this produces the specialised *power5* function shown in Figure 1.4. The recursive calls to *power* have been unrolled, and all operations only dependent on the *static* input variable y have been precomputed. The residual program is dependent only on the remaining *dynamic* argument x.

```
power5(x) : x * square(square(x))
```

FIGURE 1.4: Specialised power function from Figure 1.3 to compute x^5

1.2 Controlling Partial Evaluation

Partial evaluation is a complex process, in standard evaluation decisions are based on the programs input, however in partial evaluation some of the input may be missing. These control issues determine the quality of the produced code and ensure the specialisation process terminates. The control problem has been approached by two different methods: *online* and *offline*. See Leuschel and Bruynooghe (2002) and Glück and Sørensen (1996) for a thorough discussion of control issues relating to the specialisation of logic and functional languages.

In partial evaluation a decision must be made about each call in the program. The call can either be made at specialisation time under the control of the partial evaluator, or become part of the final specialised program. In the *power* example from Figure 1.4 the recursive calls to *power* have been unrolled during specialisation, this process is referred to as *unfolding*.

1.2.1 Online Partial Evaluation

In online partial evaluation all of the control decisions are made during the specialisation phase. Online specialisers usually monitor the growth of the evaluation history during specialisation, and continue computing as long as there is some evidence that interesting computations are performed and that it appears to terminate. Online techniques are potentially more precise as they have all of the static input available for making decisions but their behaviour can be more difficult to predict.

Take for example the well known Prolog append/3 example, Listing 1.1, specialised for the goal `append([a,b,A], B,C)`.

```
append([],A,A).
append([A|As], B, [A|Cs]) :- append(As, B, Cs).
```

LISTING 1.1: append/3 example in Prolog

An online partial evaluator decides whether to *unfold* a call based on the current available static data and the unfolding history. Often the size of the arguments are monitored, if there is a decrease from one iteration to the next then there is some evidence that the process might terminate. The size of an argument is be represented by a *norm*, a mapping function from a term to a natural number. For example, a list length norm would map a list to a natural number representing the length of the list.

This is illustrated in Figure 1.5. At Step 1 there is a decrease in the size of the first argument `[a,b|A]` (norm = 2 + length(A)) → `[b|A]` (norm = 1 + length(A)), so the call is safe to unfold and the first clause has not been used so some useful work has been done. At Step 2 there is again a decrease from `[b|A]` (norm = 1 + length(A)) → `A` (norm = length(A)) so we continue unfolding. However at Step 3 we can not demonstrate a decrease in size from `A` → `A'`, unfolding this call may be unsafe. The important point is that the available **static** data at each point was used in the decision process. In contrast, traditional offline partial evaluation techniques make their unfolding decisions based on an approximation of the static data and do not use the unfolding history. The only safe approximation for the goal `append([a,b|A], B,C)` would be `append(dynamic, dynamic, dynamic)` and no unfolding would take place. The termination criteria for this example is based on the "size" of the arguments, this relies on well-founded orders so that it must not be possible to infinitely decrease in size (e.g. Bruynooghe et al. (1992); Dershowitz and Jouannaud (1990)). Homeomorphic embedding (e.g. Leuschel et al. (1998); Sørensen and Glück (1995)) can also be used and checks if an ancestor is embedded in the value, i.e. can you strike out some part of the value to create something we have seen before. Deciding whether to unfold a call influences *local* control, i.e.

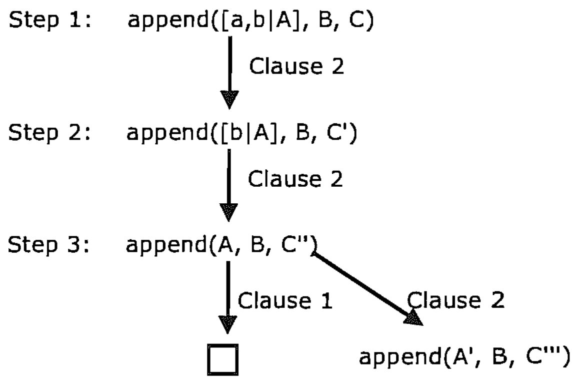


FIGURE 1.5: Unfolding `append([a,b|A], B,C)`

whether the current computation branch will ever terminate. *Global* control concerns the possible creation of an infinite number of different specialised predicates, `p(1)`, `p(2)`, `p(3)`, ..., etc. Step 3 in Figure 1.5 was not unfolded and requires the creation of an additional specialised predicate for `append(A,B,C)`. In this case we already have a specialised goal for `append([a,b|A], B,C)`, and by using well-founded orders (Martens and Gallagher, 1995) again we can determine if it is safe to specialise this goal. If we are in danger of attempting to specialise an infinite number of goals we must generalise the goal and respecialise. For example, `p(1)`, `p(2)`, `p(3)`, becomes `p(X)`.

Another approach to the global termination problem involves characteristic trees (Gallagher and Bruynooghe, 1991; Leuschel et al., 1998). Figure 1.6 shows the characteristic trees for specialising `append([a,b|A], B, C)` and `append(A,B,C)`. The characteristic tree represents the different predicates and the rule chosen to progress at each point, if two different specialisation goals produce the same characteristic tree then it may be better to generalise the two goals and produce only one specialised version. For example the specialisation goals `append(A, B, C)`, `append(A, [a], C` and `append(A, [b], C)` all share the same characteristic tree, it is sufficient to produce a single version for `append(A, B, C)`.

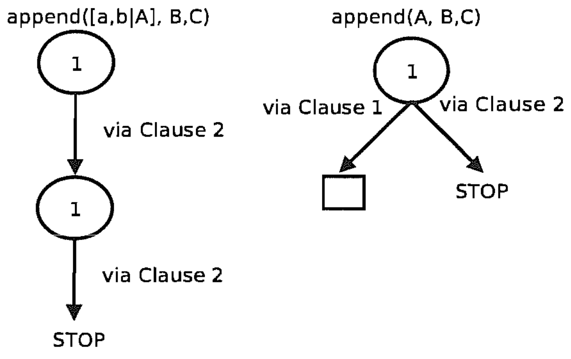


FIGURE 1.6: Characteristic Tree for unfolding `append([a,b|A], B,C)` and `append(A,B,C)`

1.2.2 Offline Partial Evaluation

Offline specialisation separates the specialisation into two phases, as depicted in Figure 1.7:

1. A *binding-time analysis* (BTA) is performed which, given a source program and an approximation of the input available for specialisation, approximates all values within the program and generates annotations that steer the specialisation process.

2. A (simplified) *specialisation phase*, which is guided by the annotations generated by the *BTA*. The annotations decide whether a call should be unfolded or residualised.

As most of the control decisions in this approach are taken beforehand it is referred to as offline. The specialisation phase of the offline approach is in general much more efficient since control decisions are made prior to and not during the specialisation phase. This is especially important in the scenario where the same program is to be respecialised several times. The binding-time analysis only needs to be performed once for the program to be specialised for different sets of static data.

The binding-time analysis is sometimes performed manually based on an approximation of the input arguments. The set of arguments that will be known at specialisation time is given, rather than their actual static values. This means the specialiser should make the same control decisions regardless of the actual static values, making offline specialisation more predictable. However, without the actual static data it cannot take full advantage of situations where extra information is known.

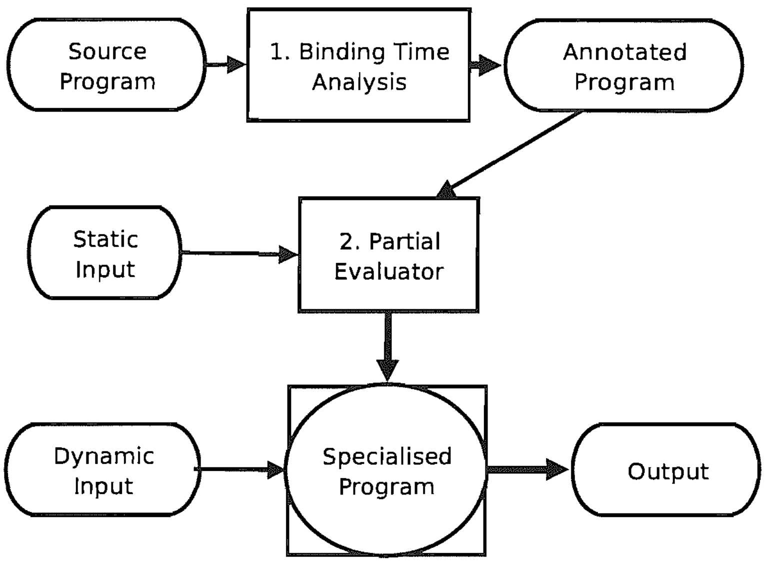


FIGURE 1.7: Overview of offline partial evaluation

1.3 Partial Evaluation of Interpreters

Partial evaluation produces useful results when applied to interpreters. 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 (Futamura, 1971).

The ultimate goal in that setting is to achieve *Jones optimality* (Jones et al., 1990, 1993; Makholm, 2000), i.e., fully removing a layer of interpretation (called the "optimality criterion" in Jones et al. (1993)). 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 1.8). 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).

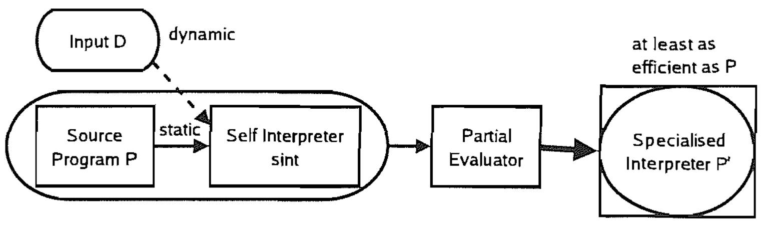


FIGURE 1.8: Jones Optimality

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 $\forall d \in D : t_{P'}(\mathrm{d}) \leq t_P(\mathrm{d})$.

1.3.1 Jones Optimality for Vanilla

We demonstrate the specialisation of interpreters and Jones optimality using the vanilla interpreter in Listing 1.2. The vanilla interpreter is a self-interpreter for Prolog, a Prolog interpreter written in Prolog. Calling solve_atom/1 looks up the program in the clause database (my_clause/2), and recursively calls solve/1 on the definition.

```
solve([]).
solve([A|T]) :- solve_atom(A), solve(T).

solve_atom(A) :- my_clause(A,B), solve(B).

my_clause(app([],L,L),[]).
my_clause(app([H|X],Y,[H|Z]),[app(X,Y,Z)]).
```

LISTING 1.2: The vanilla self-interpreter for Prolog with definition of app/3

To achieve the optimality criterion from (Jones et al., 1993) the specialiser must be able to fully remove the overhead of interpretation. Listing 1.3 is a specialised version of the interpreter for the goal solve_atom(app(A,B,C)). The definition is identical (after renaming) to the original definition of app/3. The overhead of interpretation has been removed and the optimality criterion has been met for this interpreter.

```
solve_atom__0([], A, A).
solve_atom__0([A|B], C, [A|D]) :-
        solve_atom__0(B, C, D).
```

LISTING 1.3: The vanilla interpreter specialised for solve_atom(app(A,B,C))

We will return to the vanilla interpreter in Chapter 9, where we will present some applications for specialising the vanilla interpreter.

1.4 Self-application

Guided by the *Futamura projections* (see e.g. Jones et al. (1993)) a lot of effort, especially in the functional partial evaluation community, has been put into making systems self-applicable. A partial evaluation or deduction system is called *self-applicable* if it is able to effectively[1] specialise itself. The most well-known practical interests of such a capability are related to the second and third *Futamura projections* (Futamura, 1971). 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, as already mentioned in Section 1.3 and Figure 1.8.

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.

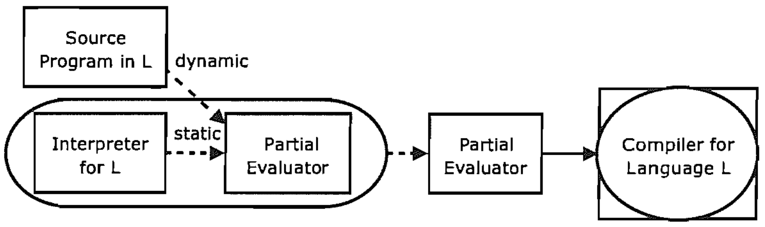


FIGURE 1.9: 2nd Futamura Projection: *Specialising the Partial Evaluator and an Interpreter to produce a compiler*

The third Futamura projection (Figure 1.10) now consists of specialising the partial evaluator to perform the second Futamura projection. By this process we obtain a *compiler generator* (*cogen*).

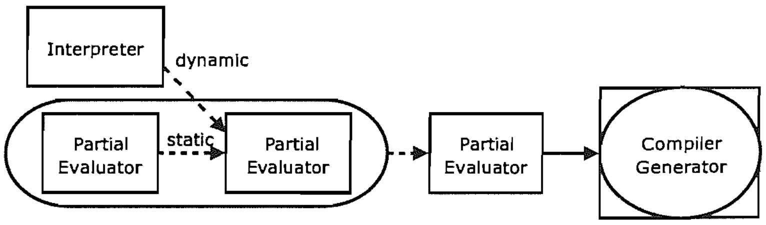


FIGURE 1.10: 3rd Futamura Projection: *Specialising the Partial Evaluator for performing the 2nd Futamura projection, producing a compiler generator*

The first successful self-application made use of offline techniques, and was reported in Jones et al. (1985), and later refined in Jones et al. (1989) (see also Jones et al.

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

(1993)). Offline techniques are beneficial for self-application as only the second simplified specialisation phase needs to be self-applied.

1.5 Prolog

Prolog (from *Programation et Logique*) is an implementation of a formal logic system (first-order Horn clauses). Throughout this thesis we assume a familiarity with Prolog, see Sterling and Shapiro (1994) for an introduction. We follow the notational conventions of Lloyd (1987). In particular, 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.

The declarative nature of Prolog allows programs to be run with incomplete input, a key concept of partial evaluation. So on the surface it might seem partial evaluation for Prolog is trivial, however in practice this is not the case. See Chapter 9 of Jones et al. (1993) or Leuschel et al. (2004b) for an overview. The examples and code presented in this thesis have been developed using SICStus[2] Prolog version 3.11.1.

For a Prolog partial evaluation system to be usable by wider audience it must support the features of a modern Prolog implementation. Some important developments include Constraint Logic Programming and coroutines.

1.5.1 Constraint Logic Programming

Constraint Logic Programming (CLP) extends traditional logic programming to include reasoning about relationships or 'constraints' in a particular domain. CLP(Q) offers a powerful constraint solver for the domain of rational numbers. See Marriott and Stuckey (1998) for an introduction to Constraint Logic Programming.

CLP allows the programmer to express the problem in a very high level language, specifying relationships between objects, while the underlying engine uses powerful incremental constraint solvers. For example, take the well known relationship from physics:

```
Force = Mass * Acceleration
```

This specifies a relationship between the three values Force, Mass and Acceleration. In traditional languages, you can not program this relationship directly but instead program how to derive each of the values from the others. To model this relationship the programmer would need to code three equations and then correctly choose the equation based on the input arguments.

[2]http://www.sics.se/isl/sicstuswww/site/index.html

```
Force        is Mass  * Acceleration
Mass         is Force / Acceleration
Acceleration is Force / Mass
```

Choosing an equation without knowing the right hand side would raise an exception in a Prolog system. CLP(Q) allows the programmer to represent the relationship directly:

```
{Force = Mass * Acceleration}
```

As the values for either Force, Mass or Acceleration become known the equation is updated and solved automatically (Listing 1.4).

```
| ?- {Force = Mass * Acceleration}, {Force = 10, Acceleration=2}.
Mass = 5.0,
Force = 10.0,
Acceleration = 2.0 ?
yes

| ?- {Force = Mass * Acceleration}, {Force = 10, Mass=5}.
Mass = 5.0,
Force = 10.0,
Acceleration = 2.0 ?
yes

| ?- {Force = Mass * Acceleration}, {Force = 10}.
Force = 10.0,
clpr:{10.0-Acceleration*Mass=0.0} ?
yes
```

LISTING 1.4: Example CLP session, using the values given the CLP solver updates and attempts to solve the equation

1.5.2 Coroutines

The computation rule in traditional Prolog systems is simple: "pick the leftmost goal of the current query". However, SICStus Prolog and other modern implementations support a more complex computation rule "pick the leftmost *unblocked* goal". A goal is *blocked* if the block condition is not satisfied, for example the arguments may not be sufficiently instantiated. SICStus Prolog defines the when/2 predicate as:

```
when(+Condition,:Goal)
    Blocks Goal until the Condition is true,
    where Condition is a goal with the restricted syntax:
    nonvar(X)
    ground(X)
    ?=(X,Y)
    Condition,Condition
    Condition;Condition
```

For example, the `is/2` predicate must be called with the second argument fully ground. If it is called with a non-ground argument an exception is thrown:

```
| ?- X is Y, Y = 2*5.
! Instantiation error in argument 2 of is/2
! goal:  _76 is _77
```

Coroutines can be used to delay the execution of the `is/2` until the second argument is sufficiently instantiated.

```
safe_is(X,Y) :- when(ground(Y), X is Y).
```

When the predicate `safe_is/2` is called, the `is/2` will be delayed until the call becomes safe.

```
| ?- safe_is(X,Y), Y = 2*5.
X = 10,
Y = 2*5 ?
yes
```

1.6 Contributions

The main aim of the work in this thesis is to make partial evaluation for Logic Programming accessible to a wider audience. To appeal to a wider audience it is important that:

- The system is as automatic as possible, but still gives expert users the power to control the specialisation.
- The specialiser handles real life programs, including the features of modern Prolog implementations.

We extend the techniques to handle features encountered in modern Prolog implementations including constraint logic programming, coroutines and some other logic features. We present a integrated development environment for specialising logic programs. This environment allows new users to visualise the annotation on their source programs without modifying their original code, and specialisation is achievable at the click of a button. To demonstrate the expressiveness of the system we present a series of increasingly complex interpreter specialisation examples, explaining the annotations and produced code.

We demonstrate a fully implemented algorithm for automatically deriving the offline annotations using state of the art termination analysis techniques, combined with type-based abstract interpretation for propagating binding types. The binding-time analysis is extended to a self-tuning, resource-aware offline specialisation algorithm. The main insight was that the annotations from offline specialisation can be used as the base for a genetic algorithm.

We develop the LIX partial evaluator for a considerable subset of full Prolog. We show it achieves non-trivial specialisation and it can be effectively self-applied. We demonstrate that, contrary to earlier beliefs, declarativeness and the use of the ground representation is not the best way to achieve self-application. Our insight is that an effective self-applicable specialiser can be derived by transforming a cogen.

The work in this thesis has contributed to a number of scientific publications, which are detailed below:

- Stephen-John Craig and Michael Leuschel, "A compiler generator for constraint logic programs", in M Broy and A Zamulin, editors, *Perspectives of System Informatics*, volume 2890 of *LNCS*, pages 148–161. Springer, 2003 (Craig and Leuschel, 2003).
- Stephen-John Craig and Michael Leuschel, "Lix: an effective self-applicable partial evaluator for Prolog", in Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings*, pages 85–99, 2004 (Craig and Leuschel, 2004).
- Stephen-John Craig, Michael Leuschel, John Gallagher, and Kim Henriksen, "Fully automatic Binding Time Analysis for Prolog", in Sandro Etalle, editor, *Logic Based Program Synthesis and Transformation, 14th International Workshop*, pages 61–70, 2004 (Craig et al., 2004).
- Michael Leuschel, Stephen-John Craig, Maurice Bruynooghe, and Wim Vanhoof, "Specializing interpreters using offline partial deduction", in Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, LNCS 3049. Springer-Verlag, 2004a (Leuschel et al., 2004a).
- Stephen-John Craig and Michael Leuschel, "Self-Tuning Resource Aware Specialisation for Prolog", *to appear in PPDP*, 2005.

1.7 Thesis Organisation

The remainder of this work is organised as follows. Chapter 2 introduces offline partial deduction for logic programs. Throughout the chapter an algorithm for partial deduction is derived and developed into the foundations of the partial evaluator, LIX, which is extended over the course of the thesis. We introduce the basic annotations and binding types.

Chapter 3 extends the LIX system introduced in Chapter 2 for self-application. Compared to earlier attempts at self-application, the LIX system is usable in terms of efficiency and can handle natural logic programming examples with partially static data structures, built-ins, side-effects, and some higher order and meta-level features such as `call` and `findall`. The work in this chapter is an extended version of Craig and Leuschel (2004).

The PYLOGEN system is introduced in Chapter 4. The PYLOGEN system provides a graphical interface into the LIX and LOGEN partial evaluators. The chapter gives a high level overview of the implementation and a description of its main features. The annotations used in the rest of the thesis are summarised.

Offline partial evaluators make use of a binding-time analysis phase as discussed in Section 1.2.2. Chapter 5 presents an algorithm for a fully automatic binding-time analysis using state of the art termination analysis techniques, combined with a new type based abstract interpretation for propagating binding types. The algorithm has been implemented as part of the PYLOGEN system and we present experimental results. The work in this chapter represents a collaboration and has been published as Craig et al. (2004).

In Chapter 6 we present the outline for a *self tuning* partial evaluation system, which derives its own specialisation control for the particular Prolog compiler and architecture by trial and error. We present the algorithm which is implemented in the PYLOGEN system and experimental results.

Chapter 7 extends the specialisation techniques to include coroutining. We discuss the problems specialising programs with coroutines and present specialised examples using the techniques. The chapter also introduces a new annotation based on the idea of delayed and guarded execution for partial evaluation.

Constraint Logic Programming (CLP) is an important paradigm in logic programming. CLP allows the programmer to model the system as a series of constraints over a domain which can then be reasoned about to produce an answer (or set of answers), in particular we look at the domain of rational numbers. Chapter 8 demonstrates that the partial evaluation system can be extended to handle the specialisation of CLP languages. The work in this chapter has been previously published as Craig and Leuschel (2003).

Chapter 9 presents experimental results using the partial evaluator. The chapter specialises the vanilla interpreter for a number of different purposes. We show that the partial evaluator is powerful enough to specialise complex interpreters and that it can achieve Jones Optimality. Notably we present an interpreter that when specialised performs the general program transformation given in Lloyd and Topor (1984).

Finally, Chapter 10 summarises the work and outlines the presented contributions. Future avenues of research are discussed.

Chapter 2

The Partial Evaluator

We now describe the process of offline partial evaluation of logic programs and develop the foundations of the LIX partial evaluation system. This should give a good understanding of the basic annotations and the algorithm behind the implementation. Over the remaining chapters the system will be extended into a fully fledged offline partial evaluator capable of self-application and the non-trivial specialisation of complex interpreters.

In the context of pure logic programs partial evaluation is referred to as *partial deduction*, the term partial evaluation being reserved for the treatment of impure logic programs (side effects, cuts). Later the system will be extended to include impure logic features, but for now we adhere to this terminology because "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.

2.1 Logic Programming

To begin with we review the basic components that make up a logic program. The basic definitions are based on Leuschel (1999), which in turn is inspired by Apt (1990); Lloyd (1987).

Definition 2.1 (alphabet). An alphabet consists of function symbols, predicate symbols, variables, connectives and punctuation symbols. Function and predicate symbols have an associated arity, indicating the number of arguments they take. Constants are functions symbols with an arity of 0, while propositions are predicate symbols with an arity of 0.

Definition 2.2 (terms). The set of terms (over some given alphabet) is inductively defined as follows:

- a variable is a term,
- a constant is a term and
- a function symbol f of arity $n > 0$ applied to a sequence $t_1, \ldots, t_n$ of n terms, denoted by $f(t_1, .., t_n)$ is also a term.

Definition 2.3 (atoms). The set of atoms (over some given alphabet) is defined in the following way:

- a proposition is an atom and
- a predicate symbol p of arity $n > 0$ applied to a sequence $t_1, \ldots, t_n$ of n terms, denoted by $p(t_1, \ldots, t_n)$, is an atom.

Definition 2.4 (literal). If A is an atom then the formulas A and $\neg A$ are called literals. A is called a positive literal and $\neg A$ a negative literal.

Definition 2.5 (clause). A clause is a formula of the form $\forall(H_1 \vee \ldots \vee H_m \leftarrow B_1 \wedge \ldots \wedge B_n)$, where $m \geq 0, n \geq, 0$ and $H_1, \ldots, H_m, B_1, \ldots, B_n$ are all literals. $H_1 \vee \ldots \vee H_m$ is called the head of the clause and $B_1 \wedge \ldots \wedge B_n$ is called the body. A (normal) program clause is a clause where $m = 1$ and H_1 is an atom. A definite program clause is a normal program clause in which $B_1, \ldots, B_n$ are atoms. A fact is a program clause with $n = 0$. A query or goal is a clause with $m = 0$ and $n > 0$. A definite goal is a goal in which $B_1, \ldots, B_n$ are atoms. The empty clause is a clause with $n = m = 0$, this corresponds to a contradiction. The symbol $\Box$ is also used to represent the empty clause.

Definition 2.6 (program). A (normal) program is a set of normal program clauses. A definite program is a set of definite program clauses.

We adhere to the usual logic programming notation:

- The universal quantifier encapsulating the clause is omitted,
- the comma is used instead of the conjunction in the body,
- variables are represented by uppercase letters and
- constants, function symbols and predicate symbols are represented by lowercase letters.

For example, the clause $\forall X(p(s(X)) \leftarrow (q(X) \wedge r(X)))$ is represented as $p(s(X)) \leftarrow q(X), r(X)$.

The definitions of *substitution* and *mgu* are required for the rest of this introduction.

Definition 2.7 (substitution). A substitution θ is a finite set of the form $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ where $X_1, \ldots, X_n$ are distinct variables and $t_1, \ldots, t_n$ are terms such that $X_i \neq t_i$. Each element X_i/t_i of θ is called a binding.

Definition 2.8 (mgu). Let S be a finite set of expressions. A substitution θ is a called a unifier of S iff the set $S\theta$ is a singleton. θ is called relevant iff its variables $vars(\theta)$ all occur in S. θ is called a most general unifier or mgu iff for each unifier σ of S there exists a substitution γ such that $\sigma = \theta\gamma$.

The Prolog execution model is based on *SLD-resolution* (Selection rule-driven Linear resolution for Definite clauses) see e.g. Lloyd (1987). We now define the components of SLD-resolution.

Definition 2.9 (SLD-derivation step). Let $G = \leftarrow L_1, \ldots, L_m, \ldots, L_k$ be a goal and $C = A \leftarrow B_1, \ldots, B_n$ a program clause such that $k \geq 1$ and $n \geq 0$. Then G' is derived from G and C using θ (and L_m) iff the following conditions hold:

1. L_m is an atom, called the selected atom (at position m), in G.
2. θ is a relevant and idempotent mgu of L_m and A.
3. G' is the goal $\leftarrow (L_1, \ldots, L_{m-1}, B_1, \ldots, B_n, L_{m+1}, \ldots, L_k)\theta$.

Definition 2.10 (SLD-derivation). Let P be a definite program and G a definite goal. An SLD-derivation of $P \cup \{G\}$ is a tuple $(\mathcal{G}, \mathcal{L}, \mathcal{C}, \mathcal{S})$ consisting of a sequence of goals $\mathcal{G} = \langle G_0, G_1, \ldots \rangle$, a sequence $\mathcal{L} = \langle L_0, L_1, \ldots \rangle$ of selected literals, a sequence $\mathcal{C} = \langle C_1, C_2, \ldots \rangle$ of variants of program clauses of P and a sequence $\mathcal{S} = \langle \theta_1, \theta_2, \ldots \rangle$ of mgu's such that:

- for $i > 0$, $vars(C_i) \cap vars(G_0) = \varnothing$;
- for $i > j$, $vars(C_i) \cap vars(C_j) = \varnothing$;
- for $i \geq 0$, L_i is a positive literal in G_i and G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1} and L_i;
- the sequences $\mathcal{G}, \mathcal{C}, \mathcal{S}$ are maximal given $\mathcal{L}$.

Definition 2.11 (SLD-refutation). An SLD-refutation of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty clause $\square$ as the last goal of the derivation.

Definition 2.12 (SLD-tree). An SLD-tree for $P \cup \{G\}$ is a labelled tree satisfying the following:

1. Each node of the tree is labelled with a definite goal along with an indication of the selected atom
2. The root node is labelled with G.
3. Let $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ be the label of a node in the tree and suppose that A_m is the selected atom. Then for each clause $A \leftarrow B_1, \ldots, B_q$ in P such that A_m and A are unifiable the node has one child labelled with
$\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$,
where θ is an idempotent and relevant mgu of A_m and A.
4. Nodes labelled with the empty goal $\square$ have no children.

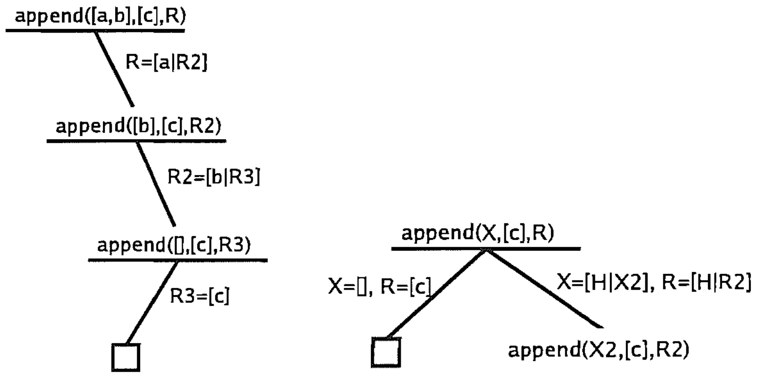
Definition 2.13 (computed answer). Let P be a definite program, G a definite goal and D an SLD-refutation for $P \cup \{G\}$ with the sequence $\langle \theta_1, \ldots, \theta_n \rangle$ of mgu's. The substitution $(\theta_1 \ldots \theta_n)|_{vars(G)}$ is then called the computed answer for $P \cup \{G\}$ (via D).

Formally, executing a logic program P for an atom A consists of building an SLD-tree (Definition 2.12) for $P \cup \{\leftarrow A\}$ and then extracting the computed answer substitutions (Definition 2.13) from every non-failing branch of that tree.

The append program is shown in Listing 2.1. The SLD-tree for `append([a,b],[c],R)` is presented in Figure 2.1(a). The selected atoms are underlined. In this example there is only one branch and its computed answer is `R = [a,b,c]`.

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

Listing 2.1: Append program



(a) SLD-tree for `append([a,b],[c],R)` (b) Incomplete SLD-tree for `append(X,[c],R)`

Figure 2.1: Complete and incomplete SLD-trees

2.2 Partial Deduction

Partial deduction builds upon this approach with two major differences:

1. At some step in building the SLD-tree, it is possible to *not* 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. The partial evaluator should not only avoid constructing infinite branches, but also branches which would cause inefficiencies in the specialised program. Incomplete branches do not produce computed answers, they produce conditional answers which can be expressed as program clauses by taking the *resultants* (Definition 2.14) of the branches.

Figure 2.1(b) is an incomplete SLD-tree for `append(X,[c],R)`, whose full SLD-tree would be infinite. The resultants of the derivations in Figure 2.1(b) are shown in Listing 2.2.

2. As atoms can be left in the leaves, 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* (Definition 2.15). In Figure 2.1(b) the leaf atom `append(X2,[c],R2)` is already an instance of its root atom `append(X,[c],R)`, hence closedness is already ensured and there is no need to build additional trees. If `append(X,[b],R)` were a leaf atom a new tree would have to be built as it is not an instance of any root atom.

Definition 2.14 (resultant). 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.

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

LISTING 2.2: Resultants of the derivations from Figure 2.1(b)

Definition 2.15 (closedness). For a given set of specialised atoms A the closedness condition requires that every atom in the body of the resultant is an instance of an atom in A. The closedness condition ensures that A forms a complete description of all possible runtime calls of the specialised program.

Partial deduction starts from an initial set of atoms A provided by the user, chosen in such a way that all runtime queries of interest are closed, i.e. each possible goal of interest is an instance of some atom in A. 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 (Gallagher, 1993; Martens and Gallagher, 1995):

- The *local control* controls the construction of the finite SLD-tree for each atom in A and thus determines *what* residual clauses are produced for the atoms in A.
- The *global control* controls the content of A. It decides *which* atoms are ultimately partially deduced. Care must be taken 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 Leuschel and Bruynooghe (2002). In offline partial deduction the local control is hardwired, in the form of annotations added to the source program during the binding-time analysis phase.

At a given node when building the SLD-tree the specialiser can choose to either:

unfold — continue building the SLD-tree for the selected atom, or

memo — choose *not* to select an atom, producing a leaf with a non-empty goal. The generalised atom is added to the set of atoms to specialise (if it is not an instance of an atom already in the set).

The global control is also partially hard-wired, by specifying which arguments to which predicate are **dynamic** and which ones are **static**. Generalisation of **dynamic** variables helps to ensure coveredness. For example, the selection of goals $p(1), \ldots, p(n)$ are all covered by the single atom $f(X)$.

2.3 An Offline Partial Deduction Algorithm

2.3.1 The Basic Annotations

As outlined earlier, an offline specialiser works on an annotated version of the source program. The annotation file contains two types of annotations:

- *Filter declarations* declare binding types for the arguments of the predicates. They specify which arguments are **static** and which are **dynamic**. This influences the global control only.
- *Clause annotations* indicate how every call in the body should be treated during unfolding. This influences the local control only. For now we assume that a call is either annotated by **memo** — indicating that it should not be unfolded, but instead generalised and specialised independently; or by **unfold** — indicating that it should be unfolded. More annotations will be introduced over the course of this thesis.

For compatibility, LIX reuses the annotations format from the LOGEN (Leuschel et al., 2004b) system. Each call in the program is annotated using `logen/2` and the binding types of arguments are given using `filter/1` declarations. The head of a clause is annotated with an identifier.

First, let us consider an annotated version of the append program (Listing 2.1). The filter declarations annotate the second argument as **static** while the remaining arguments are left **dynamic**, and the clause annotations annotate the recursive call as **memo** preventing its unfolding. These annotations are shown in Listing 2.3. The heads of both clauses are annotated with the app identifier.

```
/* the annotated source program: */
/* filter indicates how to generalise and filter */
:- filter append(dynamic ,static ,dynamic).

/* Clause annotations are converted into rule clauses on loading */
logen(app, append([],L,L)).
logen(app, append([H|T], L, [H|T1])) :- logen(memo, append(T,L,T1)).
```

LISTING 2.3: Annotated version of append from Listing 2.1

Given these annotations and a specialisation query `append(X,[c],Z)`, offline partial deduction would unfold exactly as depicted in Figure 2.1(b) and produce the resultants shown in Listing 2.2.

2.3.2 The Algorithm

Algorithm 1 is a general algorithm for offline partial deduction given filter declarations and clause annotations.

In practice, renaming transformations (Gallagher and Bruynooghe, 1990) are also involved: every atom in M is assigned to a predicate with a new name and whose arity is the number of arguments declared as dynamic (**static** arguments do not need to be passed around as they have already been built into the specialised code). The resultants of the derivations in Figure 2.1(b) would be transformed into the code in Listing 2.4. The second argument from the original append program is `static` and has been removed.

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

LISTING 2.4: Resultants of derivations from Figure 2.1(b) after renaming

To give a more precise picture, we present a Prolog version (Listing 2.6) of Algorithm 1. It should be noted that the algorithm performs a breadth first traversal of the atoms to specialise but for simplicity of implementation the Prolog code is depth first, this does not change the behaviour of the output program (though predicates may be printed in a different order). The code is runnable (using an implementation `pretty_printing_clauses/2` which prints and formats a list of clauses). We assume that the filter declarations and clause annotations of the source program are represented by `filter/2` and `rule/2` respectively. The annotations from Listing 2.3 are represented by the clauses in Listing 2.5.

2.3.3 Generalise and Filter

Generalisation and filtering transforms the arguments of a call based on the filter declaration. Generalisation is required for global termination, data marked as **dynamic**

Algorithm 1 Offline Partial Deduction

Input:A Program P and an atom A
Global:$MemoTable = \varnothing$

```
1:  generalise $A$ to give $A^G$
2:  filter $A^G$ to give $A^F$
3:  add $\langle A^G, A^F \rangle$ to $MemoTable$
4:  repeat
5:    select an unmarked pair $\langle A^G, A^F \rangle$ in $MemoTable$ and mark it
6:    STEP($A^F$, $\langle\rangle$, $\langle\langle$unfold , $A^G\rangle\rangle$)
7:  until all pairs in $MemoTable$ are marked
8:
9:  function STEP($Q, B, C$)
10:   {$Q$ is current goal}
11:   {$B$ is current residual code}
12:   {$C$ is remaining annotated atoms}
13:   if $C$ is $\varepsilon$ then
14:     pretty print the clause $Q$:-$B$
15:   else
16:     let $B = \langle A_1, ..., A_i \rangle$
17:     let $C = \langle\langle Ann_1, AA_1\rangle, ..., \langle Ann_j, AA_j\rangle\rangle$
18:     if $Ann_1$ is memo then
19:       generalise $AA_1$ to give $A^G$
20:       if $\exists\langle A^{G'}, A^{F'}\rangle \in MemoTable$ s.t. $AA_1$ is a variant of $A^{G'}$ then
21:         {$A^G$ has been previously added, compute Call to residual predicate}
22:         $\theta = mgu(AA_1, A^{G'})$
23:         $A^F = A^{F'}\theta$
24:       else
25:         {Compute residual predicate head and add call to pending list}
26:         filter $A^G$ to give $A^F$
27:         add $\langle A^G, A^F\rangle$ to $MemoTable$
28:       end if
29:       STEP($Q, \langle A_1, ..., A_i, A^F\rangle, \langle\langle Ann_2, AA_2\rangle..\langle Ann_j, AA_j\rangle\rangle$)
30:     else if $Ann$ is unfold then
31:       for all $Head$:-$Body$ in program $P$ do
32:         if $AA_1$ unifies with $Head$ giving $mgu\ \theta$ then
33:           $\theta = mgu(Head, AA_1)$
34:           let $BA' =$ concat($Body, \langle Ann_2, AA_2\rangle, \ldots, \langle Ann_n, AA_n\rangle$)
35:           STEP($Q\theta, B\theta, BA'\theta$)
36:         end if
37:       end for
38:     end if
39:   end if
40: end function
```

```
rule(append([],A,A), true).
rule(append([A|B],C,[A|D]), logen(memo,append(B,C,D))).
filter(append(_,_,_), [dynamic,static,dynamic]).
```

LISTING 2.5: Annotated append program from Listing 2.3 using `rule/2` and `filter/2`.

should be replaced with fresh variables. For instance a call `p(1)` with first argument marked **dynamic** will be transformed into `p(X)`, which can then be reused by a call to `p(2)`. This is normally done to avoid producing residual code for a possibly infinite set of goals e.g. `p(1)`, `p(2)`, `p(3)`, etc.

Filtering creates the residual predicate heads that will appear in the specialised code. Arguments marked **static** are discarded, and **gensym** is called to create a unique name for the predicate. For example a call `append(S,[],S)` with filter declaration `[dynamic, static, dynamic]` would be transformed into `append__0(X,Y)`. The second argument has been removed by filtering and the first and third arguments have been generalised and replaced by fresh variables.

This generalisation and filtering is performed by `generalise_and_filter/3` (lines 34–45). The second argument returns the generalised original call (no filtering) and the third argument is the generalised and filtered call.

2.3.4 Driving the Specialisation

An atom A is specialised by calling `memo(A,Res)`. The `memo/2` predicate (lines 5–16) returns in its second argument the call, after generalisation and filtering, to the new specialised predicate. The global side effect, `assert(memo_table(GenCall,FCall))` (line 11), is used to maintain the list of previously specialised calls. Finally, the last call to `memo_table(Call,ResCall)` (line 14) binds `ResCall` to the residual version of the call `Call`.

Note the difference between `ResCall`, `GenCall` and `FCall`. Consider for example the filter declaration for append from Listing 2.3 with `Call = append(S,[],S)`. The generalised call to be unfolded, `GenCall`, becomes `append(Y,[],Z)`; `FCall`, the filtered head of the specialised version, becomes `append__0(Y,Z)`; and the original call is to be replaced by `ResCall` = `append__0(S,S)`.

The predicate `unfold/2` (line 18) 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 **unfold** is further unfolded. All clauses defining the new predicate are collected using `findall/3` and pretty printed.

2.3.5 Built-ins

To be able to deal with built-ins, we also add two more annotations. A call annotated as **call** is completely evaluated and 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.

```
 :- dynamic
         memo_table/2,
         flag/2.

memo(Call,ResCall) :-
     (memo_table(Call,ResCall) ->
                true /* nothing to be done: already specialised */
      ;
                (
                  generalise_and_filter(Call,GenCall,FCall),
                  assert(memo_table(GenCall,FCall)),
                  findall((FCall:-B),unfold(GenCall,B),XClauses),
                  pretty_print_clauses(XClauses),
                  memo_table(Call,ResCall)
                )
      ).

unfold(X,Code) :-
         rule(X,B),
         body(B,Code).

body((A,B),(CA,CB)) :-
         body(A,CA),
         body(B,CB).
body(true,true).
body(logen(memo,C),ResC) :-
         memo(C,ResC).
body(logen(unfold,C),ResCode) :-
         unfold(C,ResCode).
body(logen(call,C),true) :-
         call(C).
body(logen(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).
```

```

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

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

gensym(Head, ResidualHead) :-
        var(ResidualHead),
        atom(Head),
        oldvalue(Head, OldVal),
        NewVal is OldVal+1,
        set_flag(gensym(Head), NewVal),
        name(A__, "__"),
        string_concat(Head, A__, Head__),
        string_concat(Head__, NewVal, ResidualHead).

append([], A, A).
append([A|B], C, [A|D]) :-
        append(B, C, D).

string_concat(A, B, C) :-
        name(A, D),
        name(B, E),
        append(D, E, F),
        name(C, F).

/* clause database: Automatically created from annotated file */
rule(append([],A,A), true).
rule(append([A|B],C,[A|D]), logen(memo,append(B,C,D))).
filter(append(_,_,_), [dynamic,static,dynamic]).
```

LISTING 2.6: Prolog implementation of Algorithm 1

2.3.6 Example

Let us now examine the behaviour of running the specialiser on the annotated append example (Listing 2.3). Calling the specialiser with `memo(append(X,[c],Y),R)` produces the specialised program in Listing 2.7.

```
append__1([],[c]) :- true.
append__1([A|B],[A|C]) :- append__1(B,C).
```

LISTING 2.7: Output from specialiser, running on the append example for the goal `memo(append(X,[c],Y),R)`

We now step through the execution of `memo(append(X,[c],Y),R)` on Listing 2.6.

1. `memo(append(X,[c],Y), R)` is called
2. `memo_table/2` is empty so the `if` test fails
3. Generalise and Filter `append(X,[c],Y)`:
   - `Call = append(X,[c],Y)`
   - `GenCall = append(A,[c],B)`
   - `FCall = append__1(A,B)`
4. Store `memo_table(append(A,[c],B), append__1(A,B))`
5. Findall solutions for `unfold(append(A,[c],B),Body)`
   (a) Matches `rule(append([], [c],B), true)`
      - `body(true,true)` matches

   (b) Matches `rule(append([C|D],[c],[C|E]),logen(memo,append(D,[c],E)))`
      - `body(logen(memo,append(D,[c],E)),Res)` matches
      - call `memo(append(D,[c],E),R1)`, this matches entry in memo table
      - body is replaced with `append__1(D,E)`
6. Pretty print clauses:
   - `append__1([],[c]) :- true.`
   - `append__1([A|B],[A|C]) :- append__1(B,C).`
7. Unify `R` with entry point `append__1(A,B)`

The generation of the code in Listing 2.7 took 0.318 ms[1]. This is a very simple example to demonstrate the partial evaluator, the specialisation of a non-trivial Functional language interpreter can be found in Section 3.6 and other examples can be found in Chapter 9 or on the LIX home page[2].

The specialisation process can be made much more efficient through self-application, this is discussed in Chapter 3.

2.4 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 non-termination:

[1]Benchmarks performed using SICStus Prolog 3.11.1 for Linux on a Pentium 2.4Ghz with 512MB RAM. Timings are averaged over 100,000 iterations.

[2]`http://www.ecs.soton.ac.uk/~sjc02r/lix/lix.html`

- **Local non-termination:** The unfolding predicate `unfold/2` may fail to terminate or provide infinitely many answers.
- **Global non-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.[3]

To overcome the first problem of local non-termination, 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.

To overcome global termination problems, we modify 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**. The call will no longer be executed during specialisation and will become part of the residual code.

2.5 Summary

We have presented the basic algorithm for an offline partial evaluator for logic programs. The algorithm is implemented as the foundations of the LIX partial evaluation system. We introduced a set of basic annotations to ensure the process terminates and produces results for simple examples. The binding-time analysis phase annotates the source program with a safe set of annotations to ensure termination, this can be done by hand or automatically. The automatic generation of safe annotations is discussed in depth in Chapter 5. An optimal set of annotations should not only guarantee the specialisation process terminates but that it also produces good quality residual code. The fine tuning of the annotations for performance and code size is discussed in Chapter 6.

The implementation is extended in Chapter 3 to become a fully fledged partial evaluator and demonstrates that effective self-application can be achieved.

[3]One often tries to ensure that a static argument is of *bounded static variation* (Jones et al., 1993), so that global termination is guaranteed.

Chapter 3

Self-application

The work in this chapter has been previously published as Craig and Leuschel (2004) for the Functional and Logic Programming, 7th International Symposium.

This chapter develops LIX into a self-applicable partial evaluator for a considerable subset of full Prolog. The partial evaluator is shown to achieve non-trivial specialisation and be effectively self-applied. The attempts to self-apply partial evaluators for logic programs have, of yet, not been all that successful. Compared to earlier attempts, our LIX system is usable in terms of efficiency and can handle natural logic programming examples with partially static data structures, built-ins, side-effects, and some higher-order and meta-level features such as `call/1` and `findall/3`. The LIX system is derived from the development of the LOGEN compiler generator system. It achieves a similar kind of efficiency and specialisation, but can be used for other applications. Notably, first attempts at using the system for deforestation and tupling in an offline fashion are shown. The chapter will demonstrate that, contrary to earlier beliefs, declarativeness and the use of the ground representation is not necessarily the best way to achieve self-applicable partial evaluators.

3.1 Introduction

Partial evaluators perform a source to source program transformation, optimising a program based on known **static** data. Partial evaluation of interpreters can produce interesting results. The **static** input is typically the object program and the actual runtime query is left **dynamic**. This specialisers the interpreter for a particular program (Figure 3.1), the static overhead of interpretation can be removed producing something akin to a compiled version of the object program (Futamura, 1971).

A partial deduction system is called *self-applicable* if it is able to successfully specialise itself and produce a worthwhile result (i.e. non-trivial specialisation).

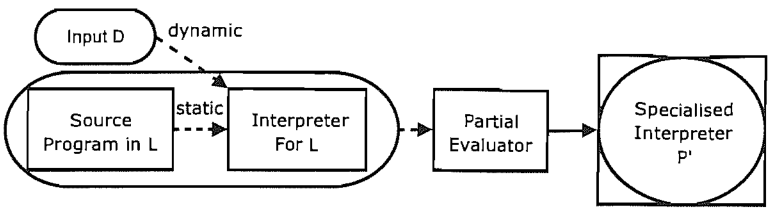


FIGURE 3.1: 1st Futamura Projection: *Specialising an interpreter to produce a compiled program*

Using the process shown in Figure 3.1 the user may want to specialise multiple source programs, each producing a different specialised object program. Each time the partial evaluator runs on the same interpreter, but each time the source program changes. The interpreter is **static** while the source program is **dynamic**. If the partial evaluator is self-applicable it is possible to partially evaluate the partial evaluator for performing the 1st Futamura projection producing a specialised partial evaluator for the interpreter. This specialised partial evaluator can transform any source program in L into a specialised object program, hence it is referred to as a compiler for language L. This is the basis of the 2nd Futamura projection (Figure 3.2).

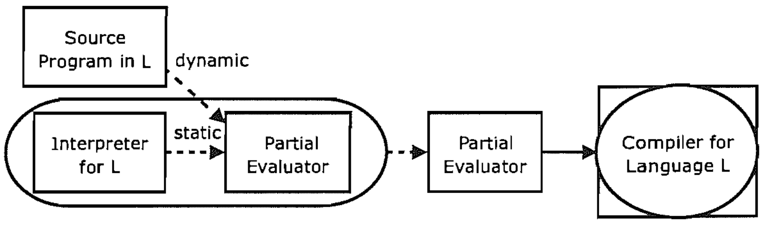


FIGURE 3.2: 2nd Futamura Projection: *Specialising the partial evaluator and an interpreter to produce a compiler*

Using the 2nd Futamura projection the user may want to generate compilers for different interpreters. Each time specialising the same partial evaluator but for a different interpreter. The partial evaluator itself is **static** but the interpreter is **dynamic**. The partial evaluator can now be specialised for performing the 2nd Futamura projection producing a specialised partial evaluator for specialising the partial evaluator. Given an interpreter it produces a specialised interpreter which in turn can produce specialised object code, hence it is called a *compiler generator* (*cogen* for short). Generating compilers from interpreters is the basis of the 3rd Futamura projection (Figure 3.3).

The Futamura projections also apply to other programs which are not interpreters. In this case the compiler is referred to as a *generating extension*. Specialising the partial evaluator makes the specialisation process itself much more efficient.

Offline techniques split the specialisation process into two parts: first a binding-time analysis phase, followed by a simplified specialisation phase which is guided by the

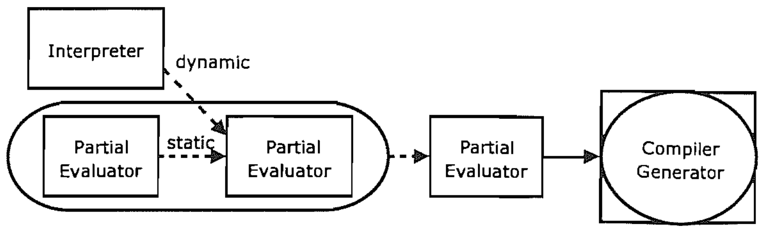


FIGURE 3.3: 3rd Futamura Projection: *Specialising the partial evaluator for performing the 2nd Futamura projection, producing a compiler generator*

results of the binding-time analysis. This separation is useful for self-application as only the second simplified phase has to be self-applied (Jones et al., 1993, 1985, 1989). In the context of logic programming languages the offline approach was used to achieve self-application in Gurr (1994a); Mogensen and Bondorf (1992).

3.1.1 History of Self-application for Logic Programming

Not surprisingly, writing an effective self-applicable specialiser is a non-trivial task — the more features one uses in writing the specialiser the more complex the specialisation process becomes, as the specialiser then has to handle these features as well. For a long time it was believed that in order to develop a self-applicable specialiser for logic programs one needed to write a clean, pure and simple specialiser. In practice, this meant using few (or even no) impure features in the implementation of the specialiser. For this the *ground representation* (Hill and Gallagher, 1998) was believed to be key, in which variables of the source program are represented by constants within the specialiser. Indeed, the ground representation allows one to freely manipulate the source program to be specialised in a declarative manner. The *non-ground representation*, where source-level variables are represented as variables in the program specialiser, can suffer from semantical problems (Martens and De Schreye, 1995b) and requires some non-declarative features (such as `findall/3`) in order to perform the specialisation.

Some early attempts at self-application (Fujita and Furukawa, 1988) used the non-ground representation, but the self-application led to incorrect results as the specialiser did not properly handle the non-declarative constructs that were employed in its implementation.[1] Other specialisers like MIXTUS (Sahlin, 1993), PADDY (Prestwich, 1992) and ECCE (Leuschel et al., 1998) use the non-ground representation, but none of them are able to effectively specialise themselves (or there is no or little speedup).

The ground representation approach towards self-application was pursued in Bondorf et al. (1990), Leuschel (1994), Mogensen and Bondorf (1992), and Bowers and Gurr (1995); Gurr (1994a,b) leading to some self-applicable specialisers:

[1]A problem mentioned in Bondorf et al. (1990), see also Leuschel (1994); Mogensen and Bondorf (1992).

- SAGE (Gurr, 1994a), a self-applicable partial evaluator for Gödel. While the speedups obtained by self-application are respectable, the process takes a very long time (several hours) and the obtained specialised specialisers are still extremely slow. This is probably due to the explicit unification algorithm required by the ground representation. To effectively specialise this explicit algorithm a much more powerful specialisation techniques would be required to obtain reasonably efficient specialisers (Leuschel and De Schreye, 1996). Similar performance problems were encountered in the earlier work (Bondorf et al., 1990).
- LOGIMIX (Jones et al., 1993; Mogensen and Bondorf, 1992), a self-applicable partial evaluator for a subset of Prolog, including *if-then-else*, side-effects and some built-ins. LOGIMIX uses a meta-interpreter (sometimes called *InstanceDemo* Hill and Gallagher (1998)) for the ground representation in which the goals are "lifted" to the non-ground representation for resolution. This avoids the use of an explicit unification algorithm, at the expense of some power.[2] Unfortunately, LOGIMIX gives only modest speedups (when compared to results for functional programming languages, see Mogensen and Bondorf (1992)), but it was probably the first practical self-applicable specialiser for a logic programming language.

Given the problem in developing a truly practical self-applicable specialiser for logic programs, the attention shifted to the *cogen approach* (Holst, 1989): instead of trying to write a partial evaluation system which is neither too inefficient nor too difficult to self-apply, one simply writes a compiler generator directly. Indeed, the actual creation of the cogen according to the third Futamura projection is in general not of much interest to users since the cogen can be generated once and for all when a specialiser is given. This approach was pursued in Jørgensen and Leuschel (1996); Leuschel et al. (2004b) leading to the LOGEN system, which can produce specialised specialisers much more efficiently than any of the self-applicable systems mentioned above. The resulting specialisers themselves are also much more efficient.

3.1.2 A New Attempt at Self-application

In a sense the cogen approach has closed the practical debate on self-application for logic programming languages: one can get most of the benefits of self-application without writing a self-applicable specialiser. Still, there is the question of academic curiosity: is it really impossible to derive the cogen written by hand in Jørgensen and Leuschel (1996); Leuschel et al. (2004b) by self-application? Also, having a self-applicable specialiser is sometimes more flexible as it can generate different cogens for different purposes (such as one with debugging enabled). It can produce more or less optimised cogens by tweaking the specialisation process, and better control the tradeoff between specialisation time and

[2] This idea was first used by Gallagher in Gallagher (1993, 1991) and then later in Leuschel and De Schreye (1995) to write a declarative meta-interpreter for integrity checking in databases.

quality of the optimised code. Maybe there are other situations where a self-applicable partial evaluation system is preferable to a cogen: Glück's specialiser projections (Glück, 1994) and the semantic modifiers of Abramov and Glück (2001) may be such a setting.

This chapter aims to answer some of these questions. Indeed, after the development of LOGEN it was realised that one could translate LOGEN into the classical style partial evaluator presented in Chapter 2. Furthermore, using new annotation facilities developed for the second version of LOGEN (Leuschel et al., 2004b), one can actually make LIX self-applicable. Self-applying LIX produces generating extensions via the second Futamura projection which are very similar to the ones produced by LOGEN, and the cogen obtained via the third Futamura projection also has a lot of similarities to the hand written code of LOGEN. The performance of this self-applicable partial evaluator is (after self-application) on par with LOGEN, and is thus much faster than any of the previous self-applicable logic programming specialisers. This chapter will also show some potential practical applications of this self-applicable specialiser.

The code of LIX itself, Listing 2.6, is surprising simple, but uses a few non-declarative features and does not use the ground representation. So, contrary to earlier belief, declarativeness and the ground representation were not the best way to climb the mountain of self-application. Indeed, the use of the non-ground representation makes our partial evaluator much more efficient and avoids all the complications related to specialising an explicit unification algorithm. The only drawback is that to safely deal with the non-ground representation, our partial evaluator needs to use some non-declarative features such as `findall/3`, and hence also has to be able to specialise them. Fortunately, this turned out to be less of a problem than anticipated.

In summary, Futamura's insight was that a cogen could be derived by a self-applicable specialiser. The insight in Holst (1989) was that a cogen is just a simple extension of a binding-time analysis, while our insight is that an effective self-applicable specialiser can be derived by transforming a cogen.

3.2 Deriving LIX from LOGEN

The LIX partial evaluator in Chapter 2 was created by transforming the LOGEN compiler generator. The basic insight was that it is possible to create a classical partial evaluator that when specialised would produce similar generating extensions. Figure 3.4 compares a small extract of code from both LOGEN and LIX, dealing with the **call** and **rescall** annotations.

The `body/3` predicate is explained in detail in Leuschel et al. (2004b). Briefly, the first argument is an annotated call, the second argument is the code that will appear in the generating extension and the third argument denotes the specialised code. The middle

```
body(logen(call,Call),Call,true).
body(logen(rescall,Call),true,Call).
```

LOGEN

```
body(logen(call,Call), true) :- call(Call).
body(logen(rescall,Call), Call) :- true.
```

LIX

FIGURE 3.4: Extract of body predicate from LOGEN and LIX

argument from body/3 in LOGEN has been transformed into a call in the LIX version. This call is annotated as **rescall** for self-application, and will hence appear in the generating extension produced by self-application. A more detailed comparison of the generating extensions and the produced cogen of LIX and LOGEN can be found in Section 3.5.1.

For example, the definition of p/2 in Listing 3.1 contains two annotated calls. The first call to is/2 is marked **call** and the call to print/1 is marked **rescall**.

```
p(X,Y) :-
      X1 is X +1,          %% annotated "call"
      print(f(X1,Y)).      %% annotated "rescall"
```

LISTING 3.1: p/2 contains two annotated calls: is/2 marked **call** and print/1 marked **rescall**

The produced generating extension will contain a specialised predicate for handling all calls to p/2 (Listing 3.2). The p_u/3 predicate contains an additional argument, the code that will become part of the residual code (in this example the print/1). Calls marked as **call** will be performed at specialisation time. This behaviour comes from the body/3 (LOGEN) and body/2 (LIX) predicates. In LIX the second argument of body/2 will become part of the residual code (it will appear in the extra argument in the specialised unfolder) and the body calls from body/2 will transformed and become the body of the specialised unfolder.

```
p_u(A, B, print(f(C,B))) :-
        C is A+1.
```

LISTING 3.2: Specialised unfolder for Listing 3.1

3.3 Towards Self-application

The main body of the code for the LIX system has already been given. For a partial evaluator to be self-applicable it must be able to effectively handle all of the features it uses. The system presented so far uses a few non-declarative features and does not use the ground representation. This section will introduce the required extensions to make LIX self-applicable.

Once the required extensions are added the LIX source code can be correctly annotated and LIX can be self-applied. Importantly it must handle findall/3, if-then-else and the cut. An additional binding type is also needed for self-application.

3.3.1 The nonvar Binding Type

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

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

A small example will help to illustrate this annotation:

```
:- filter p(nonvar).

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

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

```
%%% entry point:  p(f(Z)) :- p__0(Z)

p__0(B) :-   p__1(a).
p__1(B) :-   p__2(B).
p__2(a).
p__2(B) :-  p__0(B).
```

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

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

The `gen_filter/2` predicate in the LIX source code, Listing 2.6, is extended to handle the **nonvar** annotation (Listing 3.3). The incoming argument is deconstructed into its functor and sub-arguments, and then the sub-arguments are replaced by fresh variables making making a more general version. For the call in the final residual code the functor is discarded but the variables are kept.

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

LISTING 3.3: Extending LIX for the **nonvar** annotation

3.3.2 Treatment of findall

In LIX findall/3 is used to collect the clauses when unfolding a call; hence we have to be able to treat this feature during specialisation.

```
...
assert(memo_table(GenCall,FCall)),
findall((FCall:-B),unfold(GenCall,B),XClauses),
pretty_print_clauses(XClauses),nl,
...
```

Handling findall/3 is actually not much different from handling negation in Leuschel et al. (2004b). There is a static version (**findall**), in which the call is executed at specialisation time, and a dynamic version (**resfindall**), where it is executed at runtime. In both cases, the second argument must be annotated. For **resfindall**, much like **resnot** in Leuschel et al. (2004b), the annotated argument should be deterministic and should not fail (which can be ensured by wrapping the argument into a **hide_nf** annotation, see Leuschel et al. (2004b)). Also, if a findall/3 is marked as static then the call should be sufficiently instantiated to fully determine the list of solutions. The following code is used in the subsequent examples:

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

If the findall/3 is marked as residual and we **memo** p(X) inside it then the specialised program for all_p(a,Y) is:

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

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

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

For self-application, only **resfindall** is actually required. The body/2 predicate is extended to include Listing 3.4.

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

LISTING 3.4: Extending LIX for the **findall** annotation

3.3.3 Treatment of if

In the LIX code an if-then-else is used in memo/2.

```
 ...
(memo_table(Call,ResCall) ->
                ... % already specialised
      ;
                ... % needs to be specialised
)
 ...
```

In this case the if is dynamic, the residual body of the conditional along with its branches will be created and an if statement will be constructed in the residual code. LIX is also extended to handle a static if which is performed at specialisation time (Listing 3.5). During specialisation care must be taken to avoid back propagation from the branches of the if statement. The **hide_nf** annotation can be used to wrap the calls and prevent propagation of bindings.

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

LISTING 3.5: Extending LIX to handle the if annotations

3.3.4 Handling the cut

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

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

These conditions are sufficient to handle the cut in a sound, but still useful manner. In LIX the **cut** is used when creating unique symbol names. In LIX the cut is dynamic and is therefor annotated by **rescall**, this requires that all calls to oldvalue/2 are marked memo.

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

3.3.5 Treatment of assert

LIX uses the dynamic predicate memo_table/2 to store memo table entries. As all assertions and queries to the memo table are performed at run time and not during specialisation these calls can simply be marked as **rescall**. No further special treatment of assertions is required to specialise LIX.

3.4 Self-application

Using the features introduced in Section 3.3 and the basic annotations from Section 2.3.1, LIX can be successfully annotated for self-application. Self-application allows us to achieve the Futamura projections mentioned earlier in the Chapter.

An extract from the annotated version of the LIX source code can be seen in Listing 3.6. The full version of the annotated source code can be found in Appendix B. The memo/2 predicate is annotated using the new **resfindall** and **resif** annotations.

```
...
logen(memo, memo(A,B)) :-
        resif(logen(rescall,memo_table(A,B)),
              logen(rescall,true),
              ( logen(unfold,generalise_and_filter(A,C,D)),
                logen(rescall,assert(memo_table(C,D))),
                resfindall((D:-E),logen(memo,unfold(C,E)),F),
                logen(rescall,format('/*~k=~k*/~n',[D,C])),
                logen(memo,pretty_print_clauses(F)),
                logen(rescall,memo_table(A,B))
              )
             ).
logen(unfold, unfold(A,B)) :-
        logen(unfold, ann_clause(_,A,C)),
        logen(unfold, body(C,B)).
logen(body, body((A,B),(C,D))) :-
        logen(unfold, body(A,C)),
        logen(unfold, body(B,D)).
logen(body, body(logen(call,A),true)) :-
        logen(rescall, call(A)).
...
```

LISTING 3.6: An extract from the annotated LIX source code

3.4.1 Generating Extensions

In Section 2.3.6 we specialised app/3 for the call app(A,[b],C). If a partial evaluator is fully self-applicable then it can specialise itself for performing a particular specialisation, producing a *generating extension.* This process is the second Futamura projection. When specialising an interpreter the generating extension is a compiler.

A generating extension (Listing 3.7) for the append predicate can be created by calling lix(lix(app(A,B,C),R),R1), creating a specialised specialiser for append. This specialises LIX for specialising append (Figure 3.5).

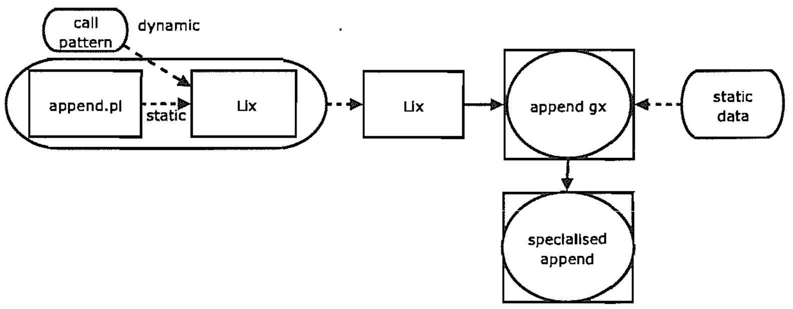


FIGURE 3.5: Specialising LIX for specialising the append program produces a generating extension for append. This is a specialised specialiser for append.

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

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

/* gensym__1(_4392) = gensym(app,_4392) */
gensym__1(A) :- var(A), oldvalue__1(app, B),
                C is B+1,set_flag__1(app, C),
                name(C, D), name(A, [97,112,112,95,95|D]).
/* Printing and Flatten Clauses removed to save space */

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

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

LISTING 3.7: Specialised specialiser or generating extension for the append predicate

This is almost entirely equivalent to the proposed specialised unfolders in Jørgensen and Leuschel (1996); Leuschel et al. (2004b). It is actually slightly better as it will do flow analysis and only generate unfolders for those predicates that are reachable from the query to be specialised. Note the gensym/2 predicate is specialised to produce only symbols of the form app__N. Generation of the above took 3.3 ms.

The generating extension for append can be used to specialise the append predicate for different sets of static data. Calling the generating extension with lix__1(A,[b],C,R) creates the same specialised version of the append predicate as in Section 2.3.6:

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

LISTING 3.8: Append specialised using the generating extension (Listing 3.7)

The unfold__1/4 predicate closely resembles the original app/3 predicate. During specialisation of LIX and the append program, the parsing and handling of the annotations has been hard coded. The extra argument in unfold__1/4 collects the residual code as it specialises append. The recursive call to app/3 has been transformed into a call to memo__1/4 which will add the call to the memo table and specialise only if it has not been done before.

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

LISTING 3.9: Extract from the append generating extension showing a specialised unfolder

A call to unfold__1/4 can be seen in Listing 3.10. The first answer represents the base case of append, there is no residual code (true) and bindings are made for A and C. The second answer is more complicated, the call to memo__1/4 is made which specialises and prints the resulting predicate. The residual code is a call to the predicate created by memo__1/4.

```
| ?- unfold__1(A,[b], C, Res).
A = [],
C = [b],
Res = true ? ;

/*app__1(_812,_810)=app(_812,'.'(b,[]),_810)*/
app__1([], [b]).
app__1([A|B], [A|C]) :-
        app__1(B, C).

A = [_A|_B],
C = [_A|_C],
Res = app__1(_B,_C) ? ;

no
```

LISTING 3.10: Calling the specialised unfolder for append

Using the generating extension is faster, for this small example 0.212 ms instead of 0.318 ms. Using a larger benchmark, unfolding (as opposed to memoising) the append predicate for a 10,000 item list produces more dramatic results. To generate the same code the generating extension takes 40 ms compared to 990 ms for LIX. The overhead of creating the generating extension for the larger benchmark is only 10 ms. Generating extensions can be very efficient when a program is to be specialised multiple times with different static data. This speed up is mainly due to the creation of the specialised unfolders; the code to be executed at specialisation time has only a minimal overhead. All the code to decipher the annotations has been removed and replaced with direct calls to either `memo` or `unfold` predicates.

3.4.2 Lix Compiler Generator

The third Futamura projection is realised by specialising the partial evaluator to perform the second Futamura projection. By this process we obtain a *compiler generator* (*cogen* for short): a program that transforms interpreters into compilers. By specialising LIX to create generating extensions we create LIX-COGEN, a self-applied compiler generator (Figure 3.6).

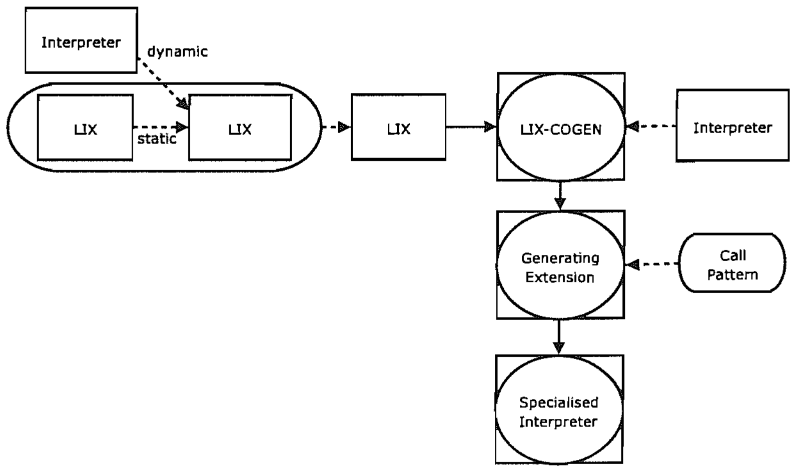


FIGURE 3.6: Specialising LIX for specialising creating generating extensions creates the LIX-COGEN . Given a program it will generate a specialised specialiser.

This can be achieved with the query `lix(lix(lix(Call,R),R1),R2)`. An extract from the produced code is now given:

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

LISTING 3.11: Extract from LIX-COGEN, created by self-applying LIX via the 3^rd Futamura projection

This has basically re-generated the 3-level cogen described in Jørgensen and Leuschel (1996); Leuschel et al. (2004b). In the **rescall** annotation, for example, the call (A) will become part of the residual program, and nothing (true) is performed at specialisation time.

This code extract demonstrates the importance of the **nonvar** annotation. The annotated version of the original unfold/2 is now shown.

```
:- filter unfold(nonvar,dynamic).
logen(unfold, unfold(X,Code)) :-
        logen(unfold, rule(X,B)),
        logen(unfold, body(B,Code)).
```

Without the **nonvar** annotation the first argument would be annotated **dynamic**, as the arguments to the call being unfolded may not be known at specialisation time. This would produce a single generic unfolder predicate much like the original LIX. The **nonvar** annotation is needed to generate the specialised unfolders.

The generated LIX-COGEN will transform an annotated program directly into a generating extension, like the one found in Section 3.4.1. However LIX-COGEN is faster: to create the same generating extension from an input program of 1,000 predicates LIX-COGEN takes only 3.9 s compared to 100.9 s for LIX.

Generation of the LIX-COGEN took only 72 ms. Once LIX-COGEN has been generated it can be reused for specialising different programs. It only has to be regenerated if LIX itself changes.

3.5 Comparison

3.5.1 Logen

The LOGEN system is an offline partial evaluation system using the cogen approach. Instead of using self-application to achieve the third Futamura projection, the LOGEN compiler generator is hand written. LIX was derived from LOGEN by rewriting it into a classical partial evaluation system. Using the second Futamura projection and self-applying

LIX produces almost identical generating extensions to those produced by LOGEN (and both systems can in principle treat full Prolog). Apart from the predicate names the specialised unfolders generated by the two systems are the same:

```
app__u([],A,A,true).
app__u([A|B],C,[A|D],E) :-
   app__m(B,C,D,E).
...
```

LOGEN Generating Extension

```
unfold__1([], A, A, true).
unfold__1([A|B], C, [A|D], E) :-
  memo__1(B, C, D, E).
...
```

LIX-COGEN Generating Extension

FIGURE 3.7: Comparison between generating extensions created by LOGEN and LIX

While LOGEN is a hand written compiler generator, LIX must be self-applied to produce the same result as in Section 3.4.2. If we compare the LOGEN source code to the LIX-COGEN in Section 3.4.2 we find very similar clauses in the form of body/3 (note however, that the order of the last two arguments is reversed).

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

LOGEN

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

LIX-COGEN

FIGURE 3.8: Comparison of LOGEN and the self-applied LIX-COGEN

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

3.5.2 Logimix and Sage

Comparisons of the initial *cogen* with other systems such as LOGIMIX, PADDY, and SP can be found in Jørgensen and Leuschel (1996). In essence, LOGEN was was 50 times faster than LOGIMIX at producing the generating extensions (0.02 s instead of 1.10 s or 0.02 s instead of 0.98 s), and the specialisation times were about 2 times faster. It is likely that a similar relationship holds between LIX and LOGIMIX given that LIX and LOGEN have similar performance. An important difference between LOGIMIX and LIX is the way the program to specialise is handled. LOGIMIX passes the ground program around as an argument while LIX makes use of the normal Prolog database to store the program. This allows LIX to take full advantage of the underlying Prolog engine. Unfortunately, LOGIMIX no longer runs on current versions of SICStus Prolog and we were thus unable to compare LIX and LOGIMIX directly. Similarly, Gödel no longer runs on current versions of SICStus Prolog, and hence we could not produce any timings for SAGE. However, timings from Gurr (1994a) indicate that the use of the ground representation means that SAGE is far too slow to be practical. Indeed, generating the compiler generator took about 100 hours and creating a generating extension for the examples in Gurr

(1994a) took at least 7.9 hours. The speedups from using the generating extension instead of the partial evaluator range from 2.7 to 3.6 but the execution times for the generating extensions still ranged from 113 s to 447 s.

3.5.3 Multi-level Languages

Our annotation scheme can be viewed as a two-level language. Contrary to MetaML (Taha and Sheard, 2000) our annotations are not part of the programming language itself (as we treat classical Prolog). It would be interesting to investigate to what extent one could extend our scheme for multiple levels of specialisation (Glück and Jørgensen, 1997).

3.6 A Non-trivial Interpreter Example

We now demonstrate that LIX can handle more complicated examples by introducing an interpreter for a simple functional language. The annotated source code is shown in Listing 3.12. The interpreter supports arithmetic operations, conditional tests, function definition and application, constants and variables. Variables are held in a partially static environment. The source code has been annotated using the annotations presented in this chapter. The annotations have been removed from the head of the clauses to increase readability.

```
eval(cst(A), _, constr(A,[])).
eval(constr(A,B), C, constr(A,D)) :-
      logen(unfold, l_eval(B,C,D)).
eval(var(A), B, C) :-
      logen(unfold, lookup(A,B,C)).
eval(plus(A,B), C, constr(D,[])) :-
      logen(unfold, eval(A,C,constr(E,[]))), logen(unfold, eval(B,C,constr(F,[]))),
      logen(rescall , D is E+F).
eval(minus(A,B), C, constr(D,[])) :-
      logen(unfold, eval(A,C,constr(E,[]))), logen(unfold, eval(B,C,constr(F,[]))),
      logen(rescall , D is E-F).
eval(times(A,B), C, constr(D,[])) :-
      logen(unfold, eval(A,C,constr(E,[]))), logen(unfold, eval(B,C,constr(F,[]))),
      logen(rescall , D is E*F).
eval(eq(A,B), C, constr(D,[])) :-
      logen(unfold, eval(A,C,E)), logen(unfold, eval(B,C,F)),
      resif(logen(rescall ,E=F), logen(rescall ,D=true), logen(rescall ,D=false)).
eval(let(A,B,C), D, E) :-
      logen(unfold, eval(B,D,F)), logen(unfold, store(D,A,F,G)),
      logen(unfold, eval(C,G,E)).
eval(if(A,B,C), D, E) :-
      logen(unfold, eval_if(A,B,C,D,E)).
eval(lambda(A,B), _, lambda(A,B)).
eval(apply(A,B), C, D) :-
      logen(unfold, eval(B,C,lambda(F,G))),
      logen(unfold, eval(A,C,H)), logen(unfold, store(C,F,H,I)),
      logen(memo, eval(G,I,D)).
```

```
eval(fun(A), _, B) :-
     logen(unfold, function(A,B)).
eval(print(A), _, constr(true,[])) :-
     logen(rescall, print(A)), logen(rescall, nl).
eval_if(A, B, _, C, D) :-
     logen(unfold, test(A,C)),logen(rescall, !), logen(unfold, eval(B,C,D)).
eval_if(_, _, A, B, C) :-
     logen(unfold, eval(A,B,C)).
test(eq(A,B), C) :-
     logen(unfold, eval(A,C,D)), logen(unfold, eval(B,C,D)).
l_eval([], _, []).
l_eval([A|B], C, [D|E]) :-
     logen(unfold, eval(A,C,D)), logen(unfold, l_eval(B,C,E)).

store([], A, B, [A/B]).
store([A/_|B], A, C, [A/C|B]).
store([A/B|C], D, E, [A/B|F]) :-
     logen(call, D\==A), logen(unfold, store(C,D,E,F)).
lookup(A, [A/B|_], B).
lookup(A, [B/_|C], D) :-
     logen(rescall, A\==B), logen(unfold, lookup(A,C,D)).
```

LISTING 3.12: An annotated interpreter for a simple function language

Listing 3.13 is the definition of Fibonacci in the functional language. It defines Fibonacci in a recursive fashion, if x is 0 or 1 the answer is returned directly otherwise it recursively calls itself.

```
function(fib, lambda(x,
                     if(eq(var(x),cst(0)),
                        cst(1),
                        if(eq(var(x),cst(1)),
                           cst(1),
                           plus(apply(minus(var(x),cst(1)),fun(fib)),
                                apply(minus(var(x),cst(2)),fun(fib)))
                           )
                        )
                    )).
```

LISTING 3.13: Fibonacci definition in the functional interpreter language

Specialising the interpreter (Listing 3.12) for running the Fibonacci definition (Listing 3.13) produces the specialised code in Listing 3.14. The overhead of interpretation has been removed, and the only remaining overhead is the extra structure `constr/2`. This can be removed with simple post-processing. The specialised code performs the same naïve Fibonacci calculation but has been converted to Prolog, the source language of the interpreter.

```
eval__1(constr(0,[]), constr(1,[])) :- !.
eval__1(constr(1,[]), constr(1,[])) :- !.
eval__1(constr(A,[]), constr(B,[])) :-
        C is A-1,
        eval__1(constr(C,[]), constr(D,[])),
        E is A-2,
        eval__1(constr(E,[]), constr(F,[])),
B is D+F.
```

LISTING 3.14: The functional interpreter specialised for the Fibonacci program

We now present the timing information for specialising the functional interpreter. All these timings are taken from averaging the execution time over 10,000 iterations. Table 3.1 shows the time taken to specialise the interpreter using LIX and using a generating extension (a specialised specialiser for the functional interpreter). Specialising using the generating extension is nearly four times faster than using LIX.

Benchmark	Time Taken
using LIX	3.74ms
using generating extension	0.94ms

TABLE 3.1: Time taken to specialise the functional interpreter

Table 3.2 shows timings for producing the generating extension. The generating extension can be created by LIX or by using the LIX-COGEN. Using LIX-COGEN is twice as fast as using LIX . The generating extension can be reused to specialise different source programs for the functional interpreter, it only has to be regenerated if the interpreter itself changes.

Benchmark	Time Taken
using LIX	21.93ms
using LIX-COGEN	10.98ms

TABLE 3.2: Time taken to create generating extension for the functional interpreter

Table 3.3 lists the time taken to generate LIX-COGEN. This however only has to be generated once as it is only dependent on LIX and not on any source programs. It only has to be regenerated if LIX changes.

Benchmark	Time Taken
using LIX	72.15ms

TABLE 3.3: Time taken to create LIX-COGEN

This section has demonstrated the expressive power of LIX by specialising a non-trivial interpreter for a functional language. Chapter 9 presents a series of increasingly complicated interpreters for specialisation.

3.7 New Applications

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

3.7.1 Several Versions of the Cogen

In the development of new annotation and specialisation techniques it is often useful to have a debugging specialisation environment without incurring any additional overhead when it is not required. Using LIX we can produce a debugging or non-debugging specialiser from the same base code, the overhead of debugging being specialised away when it is not required. By augmenting LIX with extra options we can produce several versions of the cogen depending on the requirements:

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

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

3.7.2 Extensions for Deforestation/Tupling

LIX is more flexible than LOGEN: we do not have to know beforehand which predicates are susceptible to being unfolded or memoised. Hence, LIX can handle a potentially unbounded number of predicates. Using this allows LIX to perform a simple form of conjunctive partial deduction (De Schreye et al., 1999).

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

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

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

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

Running LIX on this will produce a result where the intermediate data structure has been removed (after post-processing, as in De Schreye et al. (1999)):

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

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

We have also managed to deal with the rotate-prune example from De Schreye et al. (1999), but more research will be needed into the extent that the extra flexibility of LIX can be used to do deforestation or tupling in practice.

3.8 Summary

This chapter presented an implemented, effective and surprisingly simple self-applicable partial evaluation system for Prolog, and demonstrated that the ground representation is not required for a partial evaluation system to be self-applicable. Chapter 9 shows the LIX system can be used for the specialisation of non-trivial interpreters.

While LIX and LOGEN essentially perform the same task, there are some situations where a self-applicable partial evaluation system is preferable. LIX can potentially produce more efficient generating extensions, using specialised versions of gensym and performing some of the generalisation and filtering beforehand. There is potential for using LIX in deforestation and in producing multiple cogens from the same code. Tweaking the annotation of LIX allows the cogen generation to be controlled. The overhead of a debugging cogen can be removed or a more aggressive specialiser can be generated.

The annotations used to self-apply LIX were added by hand. Chapter 5 introduces an automatic binding-time analysis algorithm for generating annotations. It is unclear whether an automatic algorithm could generate the annotations used here due to their subtleness. However, it should be possible to use the automatic process and then modify the annotations for the desired results.

Chapter 4

PyLogen

One of the stated aims of this thesis is to make partial evaluation accessible to a wider audience. For a partial evaluation system to be usable by a wide audience it must be powerful enough to specialise real programs. Chapter 7 and Chapter 8 introduce extensions to the developed partial evaluator for constraint programming and coroutines, thus allowing a larger subset of real life programs to be handled. Chapter 9 demonstrates the power of the specialiser on a series of increasingly complex interpreters.

While the expressive power of the system is important, the system must also be easy to use. Annotating files for offline partial evaluation can be a complicated process. Chapter 5 introduces an implemented algorithm for deriving the annotations, while Chapter 6 optimises these annotations using a self tuning algorithm.

The final requirement is that the system as a whole is presented to the user in a easy to use fashion; to address this we have developed the PYLOGEN graphical interface. The system provides an interface to both the LOGEN and LIX systems. The LIX-COGEN is generated from LIX by self-application (Section 3.4.2), and once it has been generated it can be used in the same way as the hand written LOGEN.

This chapter presents the main features of the PYLOGEN interface along with a summary of the annotations and binding types used throughout the thesis. A more detailed tutorial of the system can be found in Appendix A.

4.1 System Overview

The underlying partial evaluation system is written in SICStus Prolog (Version 3.11.1) and is also being ported to Ciao Prolog. The Prolog source code is compiled to executable form for the Linux, Windows and OS X platforms. The graphical interface is written in a combination of Python[1] and Tk[2] as they both offer consistent cross platform support.

All communication between the interface and the underlying engine is performed using TCP sockets (Figure 4.1). This ensures a clean separation between the two parts and allows the engine to be potentially run on a remote server. The only restriction is that the server and client must have access to a shared file server.

Using a cogen to specialise programs can be a complicated process. After annotating the source file a specialised specialiser is created based on these annotations. This specialised specialiser (or generating extension) is then executed with the actual specialisation query, producing the final specialised code. The same generating extension can be reused for different sets of static data. It only has to be rebuilt if the annotations are modified. The PYLOGEN system makes this process transparent: the user simply specialises a file with a specialisation query. The system decides whether or not to rebuild the generating extension based on the timestamps of the annotated file. It also provides benchmarking tools and a Prolog shell to test the specialised programs.

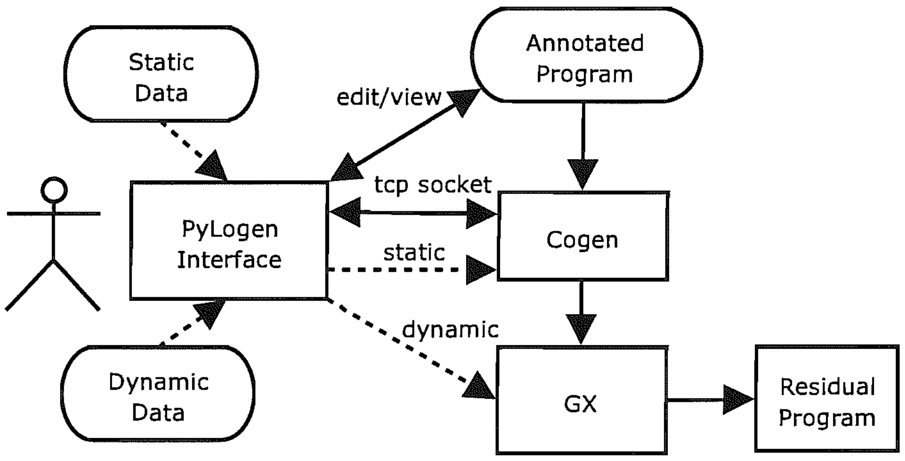


FIGURE 4.1: PYLOGEN communicates with the underlying partial evaluation engine using TCP sockets. The user can edit and view annotated files from the interface.

4.2 Annotated Files

Offline partial evaluators make use of an annotated source program to guide the specialisation process. The LIX and LOGEN partial evaluators share a common format for the annotated file (Listing 4.1).

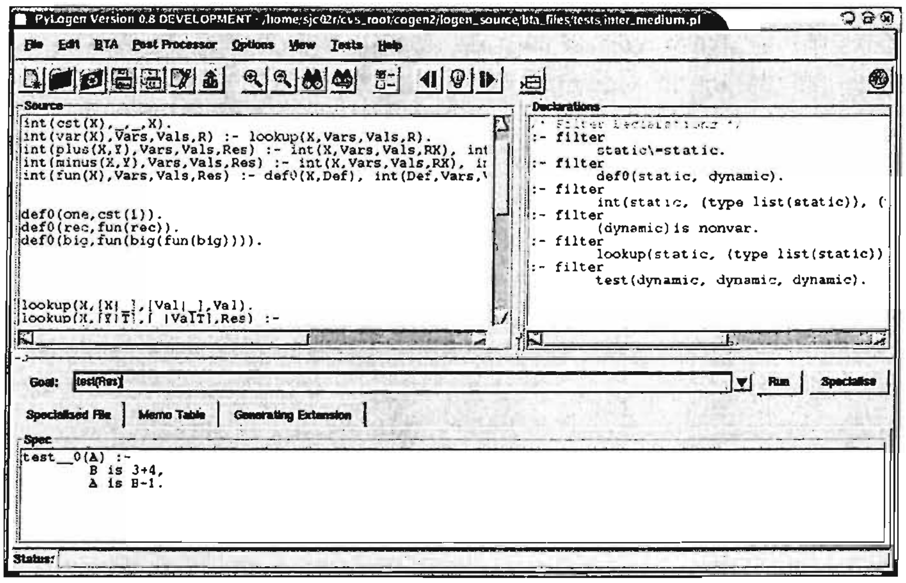
```
logen(solve, solve([])).
logen(solve, solve([A|B])) :-
        logen(memo, solve_atom(A)),
```

[1]Python 2.3, http://www.python.org/
[2]Python Tkinter, http://www.python.org/topics/tkinter/

```
        logen(unfold, solve(B)).
logen(solve_atom, solve_atom(A)) :-
        logen(unfold, my_clause(A,B)),
        logen(unfold, solve(B)).
logen(my_clause, my_clause(app([],A,A),[])).
logen(my_clause, my_clause(app([A|B],C,[A|D]),[app(B,C,D)])).
logen(my_clause, my_clause(p,[p])).
:- filter
        solve_atom(nonvar).
```

LISTING 4.1: Example of the Annotated version of the Vanilla interpreter

In the original version of LOGEN, as in most offline partial evaluators, these annotations are added by manually editing an annotation file. This is a tedious process and makes modifying the underlying source code difficult as it is marked up with annotations. The PYLOGEN graphical interface represents annotated files as a colour highlighted version of the original source code (Figure 4.2). Annotations are added to the program either using the automatic binding-time analysis (Chapter 5) or through the graphical tool. Clicking on a call in the program displays a menu of annotations. The interface allows the user to rapidly annotate files for specialisation and provides a simple way to see the results.



```
int(cst(X),_,_,X).
int(var(X),Vars,Vals,R) :- lookup(X,Vars,Vals,R).
int(plus(X,Y),Vars,Vals,Res) :- int(X,Vars,Vals,RX), int
int(minus(X,Y),Vars,Vals,Res) :- int(X,Vars,Vals,RX), i
int(fun(X),Vars,Vals,Res) :- def0(X,Def), int(Def,Vars,


def0(one,cst(1)).
def0(rec,fun(rec)).
def0(big,fun(big(fun(big)))).



lookup(X,[X|_],[Val|_],Val).
lookup(X,[Y|T],[_|ValT],Res) :-
```

```
/* Filter Declarations */
:- filter
        static\=static.
:- filter
        def0(static, dynamic).
:- filter
        int(static, (type list(static)), (
:- filter
        (dynamic)is nonvar.
:- filter
        lookup(static, (type list(static))
:- filter
        test(dynamic, dynamic, dynamic).
```

Goal: test(Res)

```
test__0(A) :-
        B is 3+4,
        A is B-1.
```

FIGURE 4.2: Screenshot of a PYLOGEN session. The top left window contains the annotated source code, the right window contains binding types and the lower pane displays the residual program.

The screen is split into three panes. The top left pane displays the source code and coloured annotations. This pane supports two modes: the first to edit the underlying source code and the second mode is used to change the annotations. The top right pane displays the filter declarations, which give the arguments of predicates a binding type. The lower pane is used for output; displaying the specialised file along with the associated generating extension and memo table.

4.2.1 Clause Annotations

In the annotated program every call is marked with a clause annotation. The clause annotations control how the call is handled during specialisation. In Listing 4.1 the **memo** and **unfold** annotations are used. The call to `solve_atom/1` is annotated **memo** and the other calls are marked **unfold**.

- **unfold** — The call is unfolded under the control of the partial evaluator. The call is replaced with the predicate body, performing all the needed substitutions.
- **memo** — The call is not unfolded, instead the call is generalised using the filter declaration and specialised independently.
- **call** — Static call. The call is made without the control of the partial evaluator. Usually used for calling built-ins, that will be sufficiently instantiated at specialisation time.
- **rescall** — Dynamic call. The unmodified call will become part of the residual program.
- **online** — The call will be unfolded/called if the safety criteria is met. See Chapter 7
- **if** — Static `if`. The condition will be evaluated and the branch will be chosen at specialisation time.
- **resif** — Dynamic `if`. The condition and both branches will be evaluated under the control of the partial evaluator but the **if** statement will remain in the residual program.
- **hide_nf** — Hide substitution and no failure. This annotation wraps calls and prevents propagation of bindings and failures (Leuschel et al., 2004b).
- **when** — Static `when/2`. The `when/2` is performed at specialisation time and must be triggered during execution of the current branch. See Chapter 7.
- **reswhen** — Dynamic `when/2`. The `when/2` is recreated in the residual program. The `Goal` should also be annotated. See Chapter 7
- **semiwhen** — Semi `when/2`. If the `Condition` is triggered during specialisation then the `Goal` will be specialised otherwise the `when/2` is recreated in the residual program. See Chapter 7
- **findall** — Static `findall/3`. The `findall/3` will be evaluated during specialisation. The second argument is higher order and must also be annotated.

- **resfindall** — Dynamic `findall/3`. The call will be specialised independently and the `findall/3` will appear in the residual program calling the specialised version of the call. The second argument is higher order and must also be annotated.

4.2.2 Binding Types

The binding types give information about the structure of arguments in the source program. The binding types are propagated through the program during the binding time analysis. The binding types are used for generalisation, i.e. **dynamic** arguments are thrown away, and filtering, i.e. **static** arguments are removed in the residual program. More information on binding types can be found in Chapter 5.

The basic binding types are:

- **static** — The argument is ground and will be known at specialisation time. The argument will be filtered and will not appear in the residual program.
- **dynamic** — The argument is not used at specialisation time, instead it will be replaced with a fresh variable. The argument will appear in the residual program.
- **semi** — The argument will not be less instantiated at run time. Ground parts of the argument will be filtered by the specialiser and will not appear in the residual program.
- **nonvar** — At least the top level functor will be known at specialisation time. The arguments will be generalised and replaced with fresh variables. The functor will be filtered in the residual program but the arguments will remain. See Section 3.3.1 in Chapter 3.

These basic binding types can be combined to produce more complex types. The behaviour of more complex types is based on their basic binding type components.

We use the ASCII notations of Mercury (Somogyi et al., 1996). For example, to define a list of **dynamic** variables:

```
:- type dynamic_list ---> [] ; [dynamic|dynamic_list].
```

During generalisation the contents of the list will be discarded, as they are marked **dynamic**. The structure of the list is classed as **static** and will be kept. The call `p([A,b,c])` is generalised to `p([X,Y,Z])` and filtered in the specialised programs to `p_1(X,Y,Z)`.

Binding types are displayed in the right panel of Figure 4.2 as `filter/1` declarations. The binding-time analysis algorithm introduced in Chapter 5 can also be used to propagate the filter declarations through the program based on the current annotations.

4.3 Summary

This chapter introduced the PYLOGEN partial evaluation system. PYLOGEN provides a graphical interface into the algorithms developed in this thesis along with the LOGEN and LIX partial evaluators. It allows programs to be loaded into the interface and annotated either by hand using point and click or using the automatic techniques. Results from specialisation are shown and can be benchmarked and tested using the built-in tools. The importance of the interface is to make partial evaluation for logic programming easier, providing a better environment to develop and specialise Prolog programs. The interface was used in the development and testing of the interpreter examples given in Chapter 9.

Chapter 5

An Automatic Binding-Time Analysis for Prolog

The work in this chapter has been published in the Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR) 2004 as *Fully Automatic Binding Time Analysis for Prolog* by SJ. Craig, M. Leuschel, J. Gallagher and K. Henriksen.

Offline partial evaluation techniques rely on an annotated version of the source program to control the specialisation process. These annotations guide the specialisation and ensure the termination of the partial evaluation. This chapter presents an algorithm for generating these annotations automatically. The algorithm uses state-of-the-art termination analysis techniques, combined with a new type-based abstract interpretation (Gallagher and Henriksen, 2004) for propagating the binding types. The algorithm has been implemented as part of the partial evaluation system and we show experimental results for a series of benchmarks.

5.1 Introduction

The offline approach to specialisation has proven to be very successful for functional and imperative programming, and more recently for logic programming. Most offline approaches perform a *binding-time analysis* (BTA) prior to the specialisation phase. Once this has been performed, the specialisation process itself can be done very efficiently (Leuschel et al., 2004b) and with a predictable outcome. Compared to online specialisation, offline specialisation is in principle less powerful (as control decisions are taken by the BTA *before* the actual static input is available), but much more efficient (once the BTA has been performed). This makes offline specialisation very useful for compiling interpreters (Leuschel et al., 2004a), a key application of partial evaluation.

However, up until now no automatic BTA for logic programs has been fully implemented (though there are some partial implementations, discussed in Section 5.8), requiring users to manually annotate the program. This is an error-prone and tedious process requiring considerable expertise. Hence, to make offline specialisation accessible to a wider audience, a fully automatic BTA is essential.

In essence, a *binding-time analysis* does the following: given a program and a description of the input available for specialisation, it approximates all values within the program and generates annotations that steer the specialisation process. The partial evaluator (or the compiler generator generating the specialised partial evaluator) then uses the generated annotated program to guide the specialisation process. This process is illustrated in Figure 5.1. The figure also shows the graphical editor which allows a user to inspect the annotations and fine tune them if necessary.

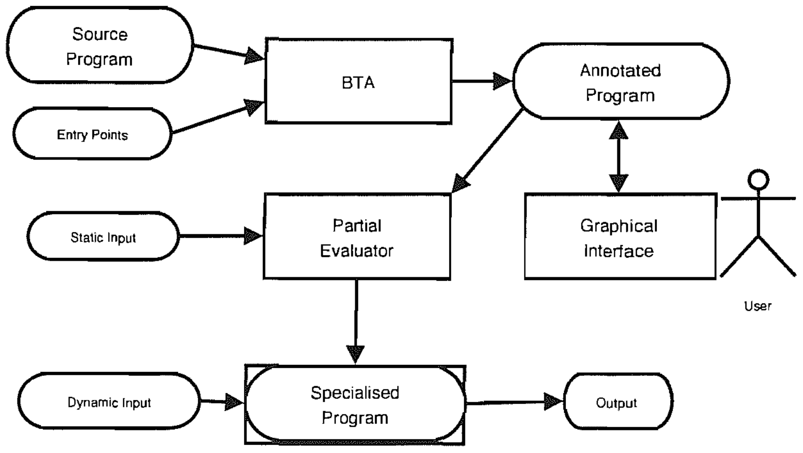


FIGURE 5.1: The role of the BTA for offline specialisation using LOGEN

To guide our partial evaluator the binding-time analysis must provide *binding types* and *clause annotations*, which will now be described.

Binding Types

Each argument of a predicate in an annotated program is given a *binding type* by means of a *filter declaration*. A binding type indicates something about the structure of an argument at specialisation time. The basic binding types are usually known as *static* and *dynamic* defined as follows.

- **static** — The argument is *definitely known* at specialisation time.
- **dynamic** — The argument is *possibly unknown* at specialisation time.

We will see in Section 5.3 that more precise binding types can be defined by means of regular type declarations and combined with basic binding types. For example, an interpreter may use an environment that is a partially static data structure at partial evaluation time. To model the environment, e.g., as a list of static names mapped to dynamic variables we could use the following definition:

```
:- type binding = static / dynamic.
:- type list_env = [] ;  [binding | list_env].
```

Through the filter declarations we associate binding types with arguments of particular predicates, as in the following example (taken from the `inter_binding` from Section 5.7):

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

The filter declarations influence *global control*, since dynamic parts of arguments are generalised away (that is, replaced by fresh variables) and the known, static parts are left unchanged. They also influence whether arguments are "filtered out" in the specialised program. Indeed, static parts are already known at specialisation time and hence do not have to be passed around at runtime.

Clause Annotations

Clause annotations indicate how each call in the program should be treated during specialisation. Essentially, these annotations determine whether a call in the body of a clause is performed at specialisation time or at run time. Clause annotations influence the *local control* (Martens and Gallagher, 1995). The developed system has four basic annotations. These annotations were already discussed in previous chapters but are reiterated here for completeness.

- **unfold** — The call is unfolded under the control of the partial evaluator. The call is replaced with the predicate body, performing all the needed substitutions.
- **memo** — The call is not unfolded, instead the call is generalised using the filter declaration and specialised independently.
- **call** — The call is fully executed without further intervention by the partial evaluator.
- **rescall** — The call is left unmodified in the residual code.

5.2 Algorithm Overview

Implementing a fully automatic BTA is a challenging task for several reasons. First, the binding type information about the static and dynamic parts of arguments has to be propagated throughout the program. Second, one has to decide how to treat each body call in the program. This has to be guided by termination issues (avoiding infinite unfolding) but also safety issues (avoiding calling built-ins that are not sufficiently instantiated). Furthermore, the decisions made about how to treat body calls in turn affect the propagation of the binding types, which in turn affect how body calls are to be treated. In summary, we need

- a precise way to propagate binding types, allowing for new types and partially static data,
- a way to detect whether the current annotations ensure safety and termination at specialisation time,
- and an overall algorithm to link the above two together.

Also, if the current annotations do not ensure termination we need a way to identify the body calls that are causing the (potential) non-termination in order to update the annotations. For this we have implemented our own termination analyser, based on the binary clause semantics (Codish and Taboch, 1999). To achieve a precise propagation of binding types we have used a new analysis framework (Gallagher and Henriksen, 2004) based on regular types and type determinization.

Figure 5.2 outlines the main steps of the BTA algorithm. The input to the algorithm consists of a program, a set of binding types, and a filter declaration giving binding types to the entry query (the query with respect to which the program is to be partially evaluated). The core of the algorithm is a loop which propagates the binding types from the entry query with respect to the current clause annotations (step 1), generates the abstract binary program (steps 2 and 3) and checks for termination conditions (step 4).

If a call is found to be unsafe at step 4 (e.g. might not terminate) the annotations are modified accordingly. Initially, all calls are annotated as **unfold** (or **call** for built-ins), with the exception of imported predicates which are annotated as **rescall** (step 0). Annotations can be changed to **memo** or **rescall**, until termination is established. Termination of the main loop is ensured since there is initially only a finite number of **unfold** or **call** annotations, and each iteration of the loop eliminates one or more **unfold** or **call** annotation.

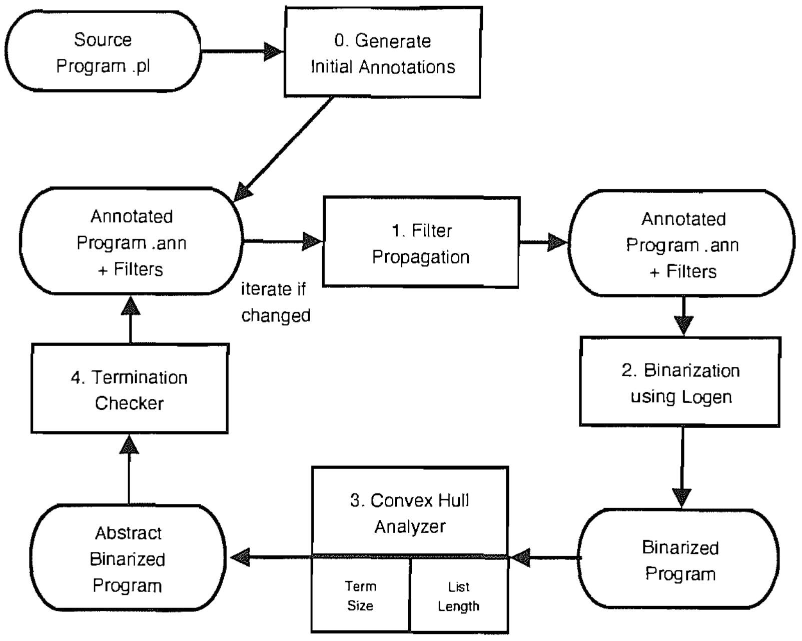

Source Program .pl
0. Generate Initial Annotations
Annotated Program .ann + Filters
1. Filter Propagation
Annotated Program .ann + Filters
iterate if changed
4. Termination Checker
2. Binarization using Logen
3. Convex Hull Analyzer
Term Size
List Length
Abstract Binarized Program
Binarized Program

FIGURE 5.2: Overview of the BTA algorithm

5.3 Binding Type Propagation

The basis of the BTA is a classification of arguments using abstract values. In this section we explain how to obtain such a classification for a given program and initial goal. The method is completely independent of the actual binding types, apart from requiring that they should include the type **dynamic**. Usually **static** and **nonvar** are also included. A *binding-time division* is a set of filter declarations of the form $p(t_1, \ldots, t_n)$, where p/n is a program predicate and $t_1, \ldots, t_n$ are binding types. For the purpose of explanation we consider only *monovariant* binding-time divisions, namely those in which there is at most one filter declaration for each predicate. However, the algorithm has been extended to *polyvariant* binding-time divisions, which allow several filter declarations for each predicate.

A binding-time division defines the binding types occurring in each predicate call in an execution of the program for a given initial goal. This information in turn is used when determining which calls to unfold and which to keep in the residual programs. A binding-time division should be *safe* in the sense that every possible concrete call is described by one of the filter declarations, e.g., a call p/n will never be made with arguments that violate the filter declaration for p/n.

The use of *static-dynamic* binding types was introduced for functional programs, and has been used in BTAs for logic programs (Mogensen and Bondorf, 1992). However, a simple classification of arguments into "fully known" or "totally unknown" is often unsatisfactory in logic programs, where partially unknown terms occur frequently at runtime, and would prevent specialisation of many "natural" logic programs such as the vanilla meta-interpreter (Hill and Gallagher, 1998; Martens and De Schreye, 1995a) or most of the benchmarks from the DPPD library (Leuschel, 1996-2004).

We outline a method of describing more expressive binding types and propagating them. The analysis framework is described in Gallagher and Henriksen (2004). The method is flexible, allowing standard static and dynamic binding types to be freely mixed with program-specific types.

Regular Binding Types

Definition 5.1 (regular type rule). A regular type t is defined by a rule of the form: $t = f_1(t_{1,1}, \ldots, t_{1,m_1}); \ldots; f_n(t_{n,1}, \ldots, t_{n,m_n})$ where $f_1, \ldots, f_n$ are function symbols (possibly not distinct) and for all $1 \leq i \leq n$ and $1 \leq j \leq m_i$, m_i is the arity of f_i, and $t_{i,j}$ are regular types.

Definition 5.2 (set of regular types). The set of types $t_1, \ldots, t_p$ define a collection of type rules. These rules can be used to assign terms to a particular type. A set of regular types $\{t_1, \ldots, t_p\}$ is *covered* iff $\forall t : t \in \{t_1, \ldots, t_p\} \wedge t = f_1(t_{1,1}, \ldots, t_{1,m_1}); \ldots; f_n(t_{n,1}, \ldots, t_{n,m_n}) \Rightarrow ((\forall i : 1 \leq i \leq n \wedge \forall j : 1 \leq j \leq m_i) \Rightarrow t_{i,j} \in \{t_1, \ldots, t_p\})$. A set of regular types is *complete* if all terms of interest can be assigned a type.

For example, Figure 5.3 defines the regular types t_1 and t_2. Using these rules the term $s(0)$ is of type t_2 and $s(s(0))$ is of type t_1. The notation $t : z$ is used to denote that the term t is of type z.

$$\begin{aligned} t_1 &= 0; s(t_2) \\ t_2 &= s(t_1) \end{aligned}$$

FIGURE 5.3: A set of regular types defining the rules for t_1 and t_n

The analysis builds upon the use of regular type rules. The interpretation of such rules is well understood in the framework of regular tree grammars or finite tree automata (Comon et al., 1997).

Instantiation modes including *static* (ground term), *nonvar* (non-variable) and *dynamic* (any term) can be coded as regular types for a given fixed signature. For example, Figure 5.4 gives the definitions of static, nonvar and dynamic for the signature $\{[], [\cdot|\cdot], s/1, 0/0, v\}$. The constant v, representing a free variable, is a distinguished constant not occurring in programs or goals. Note that v is not included in the types *static* and *nonvar*. Therefore any term of type *dynamic* is possibly a variable. The classical partial evaluation definition of static can also includes variables (if some instantiation criteria are satisfied), here we use a definition that restricts static to ground terms.

$$\begin{aligned}
&static = 0; []; [static|static]; s(static)\\
&nonvar = 0; []; [dynamic|dynamic]; s(dynamic)\\
&dynamic = 0; []; [dynamic|dynamic]; s(dynamic); v
\end{aligned}$$

FIGURE 5.4: Definitions for static, nonvar and dynamic for the signature $\{[], [\cdot|\cdot], s/1, 0/0, v\}$

In addition to modes, regular types can describe common data structures. The set of all lists, for instance, is given as $list = []; [dynamic|list]$. The set of lists of lists can be described by the type $listlist = []; [list|listlist]$. As already seen, program-specific types such as the type of environments are also regular types.

$$\begin{aligned}
&binding = static/dynamic\\
&list\_env = []; [binding|list\_env]
\end{aligned}$$

The definitions in Figure 5.4 define overlapping types. The term $s(0)$ matches all three type rules and therefore occurs in all three types. This can lead to a loss or precision during propagation. Suppose types t_1 and t_2 are not disjoint; then the terms that are in the intersection can be represented by both t_1 and t_2 and hence the two types will not be distinguishable wherever terms from the intersection can arise.

Type Determinization

Definition 5.3 (set of disjoint regular types). The set of regular types, $t_1, \ldots, t_p$, is *disjoint* if it is covered and $\forall$ terms $t, t : z_1 \wedge t : z_2 \Rightarrow z_1 = z_2$. That is the type definitions do not overlap and each term can be assigned to only one type.

A given set of regular types is transformed into a set of *disjoint* regular types. This process is called *determinization* and is a standard operation on finite tree automata (Comon et al., 1997). Such a set of rules corresponds to a bottom-up deterministic finite tree automata (Comon et al., 1997). The inclusion of the type *dynamic* ensures that the set of rules is *complete*, that is, that every term is a member of exactly one of the disjoint types.

The advantage of determinized types is that they can be propagated more precisely than non-disjoint types. In the case of the overlapping types t_1 and t_2 the set of disjoint types would contain separate types representing $t_1 \cap t_2$, $t_1 \setminus t_2$ and $t_2 \setminus t_1$. In the worst case, it can thus be seen that there is an exponential number of disjoint types for a given set of types. In practice, many of the intersections and set complements are empty and we find usually that the number of disjoint types is similar to the number of given types. Thus with disjoint types, we can obtain a more accurate representation of the set of terms that can appear in a given argument position, while retaining the intuitive, user-oriented notation of arbitrary types. In fact, the type declarations of LOGEN and LIX can be used without modification to construct an abstract domain.

For example, given the types *dynamic*, *static*, *nonvar* and *list* as shown above, determinization yields definitions of the disjoint sets of terms: (1) non-ground, non-variable non-lists, (2) non-ground lists, (3) ground lists, (4) variables and (5) ground non-lists. These are represented in Figure 5.5 the numbers indicate the areas intersecting the various types. The rules defining these disjoint types are typically hard to read and would be difficult to write directly. For example, naming the above 5 types $q_1, \ldots, q_5$ respectively, the type rule for non-ground lists is $q_2 = [q_1|q_2]; [q_2|q_2]; [q_3|q_2]; [q_4|q_2]; [q_5|q_2]; [q_2|q_3]; [q_1|q_3]; [q_4|q_3]$. A more compact representation is actually used (Gallagher and Henriksen, 2004).

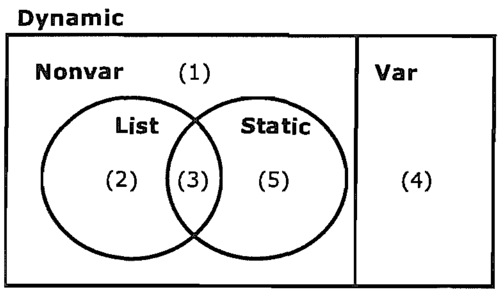


FIGURE 5.5: Set of disjoint terms from the initial set *dynamic*, *static*, *nonvar* and *list*: (1) non-ground, non-variable non-lists, (2) non-ground lists, (3) ground lists, (4) variables and (5) ground non-lists.

Propagating the Binding Types

Types can be viewed as an abstraction of terms, i.e. each type represents a set of terms. They can be used to construct a domain for abstract interpretation (Bruynooghe and Janssens, 1988; Codish and Demoen, 1994; Codish and Lagoon, 2000; Horiuchi and Kanamori, 1987). The rules for a complete set of disjoint types define a *pre-interpretation*, whose domain is the set of disjoint types. A pre-interpretation with domain D maps all constants to D and all functions f/n to $\underbrace{D \times \ldots \times D}_{n} \rightarrow D$.

An abstract interpretation based on this pre-interpretation gives the least model over the domain. (Boulanger and Bruynooghe, 1994; Boulanger et al., 1994; Gallagher et al., 1995). Using bottom up evaluation yields the success patterns for each program predicate, over the disjoint types. The termination of the evaluation is guaranteed as the domain is finite. The success patterns define the set of all possible ways the predicate can succeed; that is, each predicate p/n has a set of possible success pattern $\{p(t_1^1, \ldots, t_n^1), \ldots, p(t_1^m, \ldots, t_n^m)\}$ where t_j^i are all disjoint regular types.

The success patterns describe *all* possible instantiation modes that can occur after success of each predicate in the program. The analysis is interested in the calling patterns that can occur during specialisation, that is the instantiation modes for each predicate given an initial typed goal. The *magic-set* approach is used to obtain the calls, as described in Codish and Demoen (1993). This yields the set of call patterns for each predicate p/n, $\{p(s_1^1, \ldots, s_n^1), \ldots, p(s_1^k, \ldots, s_n^k)\}$ where s_j^i are all disjoint regular types.

Each user predicate in the program is assigned a single filter declaration. This gives the binding type of all calls at specialisation time. A single predicate may be called with many different binding types, the different binding types are combined to form a single binding type. For example, the predicate $p/1$ called with the pattern $p(t_1)$ and $p(t_2)$ can be described by a single call pattern $p(t_1 \cup t_2)$. The set of call patterns $\{p(s_1^1, \ldots, s_n^1), \ldots, p(s_1^k, \ldots, s_n^k)\}$ yields the single filter declaration $p(s_1^1 \cup \ldots \cup s_1^k, \ldots, s_n^1 \cup \ldots \cup s_n^k)$. The notation $[t_1, t_2]$ is used to represent the union $t_1 \cup t_2$. For example, the set of call patterns $\{p(q_1, q_2), p(q_2, q_2)\}$ would derive the filter $p([q1, q2], [q2])$ for the predicate $p/2$. For displaying to the user, if required, these filters can be translated back to a description in terms of the original types, rather than the disjoint types.

Analysing Annotated Programs

The standard methods for computing an abstract model and abstract call patterns have to be modified in our algorithm, since some body calls may be marked as **memo** or **rescall**. That is, they are not to be unfolded but rather kept in the specialised program. This obviously affects propagation of binding types, since a call annotated as **memo** or **rescall** cannot contribute any answer bindings.

When building the abstract model of a program, all calls marked **memo** and **rescall** are deleted from the program, as they cannot contribute anything to the model. If C is a conjunction of calls, $\overline{C}$ denotes the conjunction obtained deleting **memo**-ed and **rescall**-ed atoms from C. Let P be an annotated program; the success patterns are computed for the program $\overline{P} = \{H \leftarrow \overline{B} \mid H \leftarrow B \in P\}$.

When deriving a call pattern, for an atom B_j in clause $H \leftarrow B_1, \ldots, B_j, \ldots$, the answers to **memo**-ed and **rescall**-ed calls occurring in $B_1, \ldots, B_{j-1}$ are ignored. That is, only the clause $H \leftarrow \overline{B_1, \ldots, B_{j-1}}, B_j, \ldots$ is considered when computing the calls to B_j.

5.4 Safety of Built-in Calls

The decision on how to annotate calls to built-in predicates cannot be handled by the termination checker, but is guided by a definition of the allowed calling patterns, with respect to the given set of binding types. The allowed calling patterns must ensure a *safe* execution of the call at specialisation time, that is it must have the same behaviour as it would at runtime. For instance, considering the simple binding types **static** and

dynamic the call `X > Y` can only be guaranteed safe at specialisation time when both `X` and `Y` are **static**. The call `X is Y` can be executed whenever `Y` is **static** but `X` is **dynamic** (either known or unknown). Some built-ins have more than one allowed calling pattern; for example `functor(T,F,N)` can be executed if either `T` is **static** or both `F` and `N` are **static**.

Whenever the binding types for a call to a built-in predicate do not match one of the allowed calling patterns, the call is marked **rescall**. Thus if no calling patterns are supplied for some built-in, then all calls to that built-in will be annotated **rescall**. As well as safe calling patterns for built-ins we also provide success patterns for propagating the results.

Listing 5.1 defines the abstract call and success patterns for `is/2` and `functor/3` using the binding types **static**, **nonvar** and **dynamic**.

```
/* The call can only be made if it matches an allowed call pattern */
abstractCall(dynamic is static).
abstractCall(functor(nonvar,dynamic,dynamic)).
abstractCall(functor(dynamic,static,static)).

/* If called it will propagate the following patterns on success */
abstractSuccess(static is static).
abstractSuccess(functor(nonvar,static,static)).
```

LISTING 5.1: Abstract call and success patterns are provided for built-ins

5.5 Termination Checking

Without proper annotations in the source program, the specialiser may fail to terminate. There are two reasons for non-termination:

- **Local Termination:** Unfolding an unsafe call may fail to terminate or provide infinitely many answers.
- **Global Termination:** Even if local termination is ensured, the specialisation may still fail to terminate if it attempts to build infinitely many specialised versions of some predicate for infinitely many different static values.

Global termination is not addressed in this algorithm (Section 5.8 discusses some of the issues related to global termination). The *local termination* problem is approached using the *binary clause semantics* (Codish and Taboch, 1999), a representation of a program's computations that makes it possible to reason about loops and hence termination.

Binary Clause Semantics

Informally, the binary clause semantics of a program is the set of all pairs of atoms (called binary clauses) $p(\bar{X})\theta \leftarrow q(\bar{t})$ such that p is a predicate, $p(\bar{X})$ is a most general atom for p, and there is a finite derivation (with leftmost selection rule) $\leftarrow p(\bar{X}), \ldots, \leftarrow (q(\bar{t}), Q)$ with computed answer substitution θ. In other words a call to $p(\bar{X})$ is followed some time later by a call to $q(\bar{t})$, computing a substitution θ.

The semantics are modified to include *program point* information for each call in the program. A clause $p(ppM, \bar{X})\theta \leftarrow q(ppN, \bar{t})$ details that the call $p(\bar{X})$ at program point ppM is followed sometime later by a call to $q(\bar{t})$ at program point ppN, computing a substitution θ. This extra precision is required to correctly identify the actual unsafe call.

To create the binary clause semantics a modified vanilla interpreter is specialised with respect to the source program. The interpreter for the binary clause semantics is given in Chapter 9. Using an interpreter allows us to easily adapt the semantics for the annotations by adding rules to the interpreter.

For example, take the classic append program shown in Listing 5.2 containing a single recursive call in the second clause. In this program there is only one possible loop, the recursive call to append/3 at program point 0. This would be represented in the binary clause semantics as $append(A, B, C)\theta_0 \leftarrow append(D, E, F)$ where $\theta_0 = \{A/[G|F], B/E, C/[G|F]\}$, a call to append(A,B,C) is followed sometime later by the same call with the substitution θ_0. In fact there are an infinite number of binary clauses $append(A, B, C)\theta_0 \leftarrow append(D, E, F), append(A, B, C)\theta_1 \leftarrow append(D, E, F), \ldots$, where $\theta_0 = \{A/[G|F], B/E, C/[G|F]\}, \theta_1 = \{A/[G, H|F], B/E, C/[G, H|F]\}, \ldots$, representing the infinite number of loops possible through the same program point.

The transformation of append/3 to binary clause semantics is shown in Listing 5.3. The first clause represents a loop from the call `app([A|B], C, [A|D])` at program point 0 back to itself with the arguments `app(B, C, D)`, the second clause represents the infinite number of possible loops through the same point. The binary clause semantics can be obtained by calling `bin_solve_atom__2(ProgramPoint, Head, Body)`, producing computed answers representing the binary clauses (Listing 5.4). The printed clauses show the first three binary clauses for append/3.

```
app([], B, B).
app([A|As], B, [A|Cs]) :- /* program point 0 */ app(As, B, Cs).
```

LISTING 5.2: The append program

```
bin_solve_atom__2(0, app([A|B], C, [A|D]), app(B, C, D)).
bin_solve_atom__2(0, app([A|B], C, [A|D]), app(E, F, G)) :-
        bin_solve_atom__2(0, app(B, C, D), app(E, F, G)).
```

LISTING 5.3: The binary clause version of append from Listing 5.2

```
| ?- bin_solve_atom__2(P, Head,Body), portray_clause((Head:-Body)).
app([A|B], C, [A|D]) :-
        app(B, C, D).
P = 0,
Body = app(_A,_B,_C),
Head = app([_D|_A],_B,[_D|_C]) ? ;

app([A,B|C], D, [A,B|E]) :-
        app(C, D, E).
P = 0,
Body = app(_A,_B,_C),
Head = app([_D,_E|_A],_B,[_D,_E|_C]) ? ;

app([A,B,C|D], E, [A,B,C|F]) :-
        app(D, E, F).
P = 0,
Body = app(_A,_B,_C),
Head = app([_D,_E,_F|_A],_B,[_D,_E,_F|_C]) ?
yes
```

LISTING 5.4: Output from Listing 5.3, produces an infinite number of answers representing the infinite number of loops through the same point

Convex Hull Abstraction

The binary semantics is in general infinite, but a safe approximation of the set of binary clauses is made using abstract interpretation. We use a domain of convex hulls (the convex hull analyser used in our implementation is derived from ones kindly supplied by Genaim and Codish (2001)Benoy et al. (2004)) to abstract the set of binary clauses with respect the size of their arguments, as defined below.

Definition 5.4 (norm). A norm is a size function, mapping an arbitrary term to a natural number. The norm of t is represented as $|t|$ and is usually subscripted with the name of the norm.

A norm is used to measure the size of terms, this maps terms to natural numbers. Two well known norms are based on the term size and list length. Term size measures the number of functors in an atom (Equation 5.1) and list length counts the number of elements in a list Equation 5.2.

Definition 5.5 (rigidity). A term t is rigid w.r.t a norm $|.|_n$ iff for all instances of t, $|t|_n$ maps to the same value.

The notion of rigidity ensures that the term is instantiated enough to be accurately measured. For example, the term `[A,B,C]` is rigid w.r.t the list length norm, it will evaluate to three for all instantiations of `A`,`B` and `C`. However, the term `[A,B|Z]` is not rigid w.r.t the list length norm as different instantiations of `Z` will result in different calculated norms.

$$|t|_{term} = \begin{cases} 1 + \sum_{i=1}^{n} |t_i|_{term} & \text{if } t = f(t_1, ..., t_n) \\ 0 & otherwise \end{cases} \tag{5.1}$$

$$|t|_{list} = \begin{cases} 1 + |ts|_{list} & \text{if } t = [t|ts] \\ 0 & otherwise \end{cases} \tag{5.2}$$

Abstracting the binary clause program w.r.t a chosen norm produces a finite set of binary clauses and a set of constraints representing linear relationships between the sizes of the respective concrete arguments. Listing 5.5 is the binary clause program for append, Listing 5.3, abstracted using the domain of convex hulls with respect to the list norm.

```
bin_solve_atom(0, app(A,B,C), app(D,E,F)) :-
        [A = 1 + D, B = E, C = 1 + F,  D > = 0, E >= 0, F >= 0]
```

LISTING 5.5: Abstract convex hull of Listing 5.3 using the List norm

The constraints represent the relation that a single element of the first and third arguments is removed each iteration while the size of the second argument remains unchanged.

We are investigating properties of the program at specialisation time. The propagated binding types describe the possible values during specialisation time, for an argument to be rigid w.r.t to a norm at specialisation time its binding type must be rigid w.r.t the norm (Definition 5.6). The term size norm is only rigid if the term itself is fully ground, thus only binding types guaranteeing groundness are rigid w.r.t the term size norm. The list length norm requires that at least a list skeleton is present, therefore only binding types matching this description are rigid.

Definition 5.6 (rigidity for binding types). A binding type s is rigid w.r.t to a norm n iff all terms of type s are rigid w.r.t to the norm n.

Checking termination criteria

Loops are represented by binary clauses with the same predicate occurring in the head and body. Termination can be proven if for every abstract binary clause between p and p (at the same program point) there is a strict reduction in the size for some rigid argument (i.e. its binding type must be rigid w.r.t the abstracted norm).

For the loop, $p(t_1, \ldots, t_n) \leftarrow p(t'_1, \ldots, t'_n)$, to be safe we must show that $\exists i$ where $1 \leq i \leq n \wedge |t_i| > |t'_i|$ and the binding type for t_i is rigid w.r.t the chosen norm. The constraints shown in Listing 5.5 show a decrease in the first ($A = 1 + D$) and third argument ($C = 1 + F$). Given the initial filter declaration:

```
:- filter app(type list(dynamic), dynamic, dynamic).
```

Only the first argument is rigid w.r.t the list norm, as `type list(dynamic)` guarantees a valid list skeleton at specialisation time. Termination is proven for this loop using the first argument and these binding types. However if the filter specified was

```
:- filter app(dynamic,type list(dynamic), dynamic).
```

then the call would have to be marked unsafe and would be changed from **unfold** to **memo**, as there is no strict decrease in any rigid arguments.

5.6 Example

We demonstrate the binding-time analysis using the transpose example shown in Listing 5.6. The program takes a matrix, represented as a list of lists, and transposes the rows and columns.

```
/* Created by Pylogen */
/* file: transpose.pl */
transpose(Xs,[]) :-  nullrows(Xs).
transpose(Xs,[Y|Ys]) :-   makerow(Xs,Y,Zs), transpose(Zs,Ys).

makerow([],[],[]).
makerow([[X|Xs]|Ys],[X|Xs1],[Xs|Zs]) :-  makerow(Ys,Xs1,Zs).

nullrows([]).
nullrows([[]|Ns]) :-  nullrows(Ns).
```

LISTING 5.6: Program for transposing a matrix

The initial filter declaration, providing the binding types of the entry point is:
`:- filter transpose((type list(dynamic)), dynamic).`
The first argument is a list of **dynamic** elements, the length of the list will be known but the individual elements will not be known at specialisation time. The second argument is fully **dynamic**; it will not be given at specialisation time.

All calls in the program are initially annotated as **unfold**. Using this initial annotation and the entry types for transpose the binding types are propagated throughout the program. The resultant binding types are shown in Listing 5.7. The list structure has been successfully propagated through the clauses of the program.

```
:- filter   transpose((type list(dynamic)), dynamic).
:- filter   makerow((type list(dynamic)), dynamic, dynamic).
:- filter   nullrows((type list(dynamic))).
```

LISTING 5.7: Propagated filters for Listing 5.6 using the initial filter `transpose((type list(dynamic)), dynamic)`

The next stage of the algorithm looks for possibly non-terminating loops in the annotated program. The result is shown in Listing 5.8. The binary clause representation of the program has been abstracted with respect to the list norm over the domain of convex hulls. For termination of each of the loops in Listing 5.8 to be proven it must be shown that there is a strict decrease in any rigid argument. Based on the propagated binding types only the first argument of each predicate is rigid with respect to the list norm. The predicate `makerow/3` has a strict decrease (`A=1.0+D`); `nullrows/1` also has a strict decrease (`A=1.0+B`), but the recursive call to `transpose/2` has no decrease in a rigid argument and is therefore unsafe.

```
bin_solve_atom(3, makerow(A,B,C), makerow(D,E,F)) :-
        [A=1.0+D,D>=0.0,B=1.0+E,E>=0.0,C=1.0+F,F>=0.0].
bin_solve_atom(4, nullrows(A), nullrows(B)) :-
        [A=1.0+B,B>=0.0].
bin_solve_atom(2, transpose(A,B), transpose(C,D)) :-
        [B>D,C>=0.0,D>=0.0,A=C,B=1.0+D].

%% Loop at program point 2 is unsafe (transpose/2)
```

LISTING 5.8: Binary clause representation of Listing 5.6 abstracted over the domain of convex hulls with respect to the List norm

Marking the offending unsafe call as **memo** removes the potential loop and further iterations through the algorithm produce no additional unsafe calls. The final output of the BTA algorithm is shown in Listing 5.9. The result is a correctly annotated program for specialising transpose for a given list length.

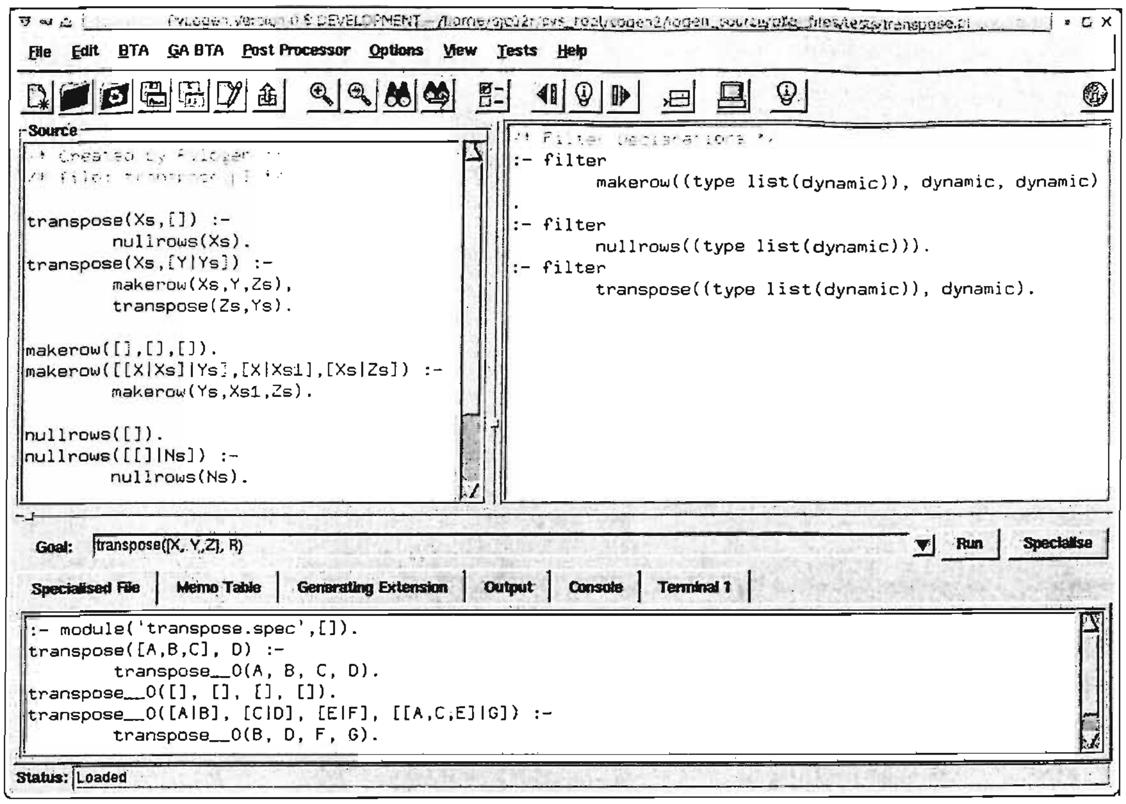
```
logen(transpose, transpose(A,[])) :-
        logen(unfold, nullrows(A)).
logen(transpose, transpose(A,[B|C])) :-
        logen(unfold, makerow(A,B,D)),
        logen(memo, transpose(D,C)).
logen(makerow, makerow([],[],[])).
logen(makerow, makerow([[A|B]|C],[A|D],[B|E])) :-
        logen(unfold, makerow(C,D,E)).
logen(nullrows, nullrows([])).
logen(nullrows, nullrows([[]|A])) :-
        logen(unfold, nullrows(A)).

:- filter    makerow((type list(dynamic)), dynamic, dynamic).
:- filter    nullrows((type list(dynamic))).
:- filter    transpose((type list(dynamic)), dynamic).
```

LISTING 5.9: Annotated version of transpose from Listing 5.6

5.7 Experimental Results

The automatic binding-time analysis detailed in this chapter is implemented as part of the PYLOGEN system. The system has been tested using benchmarks derived from the DPPD benchmark library (Leuschel, 1996-2004). The figures in Table 5.1 present



```
transpose(Xs,[]) :-
        nullrows(Xs).
transpose(Xs,[Y|Ys]) :-
        makerow(Xs,Y,Zs),
        transpose(Zs,Ys).

makerow([],[],[]).
makerow([[X|Xs]|Ys],[X|Xs1],[Xs|Zs]) :-
        makerow(Ys,Xs1,Zs).

nullrows([]).
nullrows([[]|Ns]) :-
        nullrows(Ns).
```

```
:- filter
        makerow((type list(dynamic)), dynamic, dynamic)
.
:- filter
        nullrows((type list(dynamic))).
:- filter
        transpose((type list(dynamic)), dynamic).
```

Goal: transpose([X, Y,Z], R)

```
:- module('transpose.spec',[]).
transpose([A,B,C], D) :-
        transpose__0(A, B, C, D).
transpose__0([], [], [], []).
transpose__0([A|B], [C|D], [E|F], [[A,C,E]|G]) :-
        transpose__0(B, D, F, G).
```

Status: Loaded

FIGURE 5.6: Screenshot of transpose example from Listing 5.9

the timing results[1] from running the BTA on an unmodified program given an initial filter declaration. These benchmark examples along with the PYLOGEN system can be downloaded from the website[2].

Benchmark	BTA	Original	Specialised	Relative Time
combined	3220ms	110ms	30ms	0.27
inter binding	1380ms	60ms	10ms	0.17
inter medium	1440ms	140ms	10ms	0.07
inter simple	2670ms	80ms	30ms	0.38
match	400ms	90ms	70ms	0.78
regexp	780ms	220ms	60ms	0.28
transpose	510ms	80ms	10ms	0.13

TABLE 5.1: Benchmark figures for the Automatic Binding-Time Analysis

- **combined** - A test case combining the inter simple, inter medium and regular expression interpreters.
- **inter binding** - An interpreter using a partially static data structure for an environment. In this example we combine the list and term norms.

[1] The execution time for the Original and Specialised code is based on executing the benchmark query 20,000 times on a 2.4Ghz Pentium with 512MB running SICStus Prolog 3.11.1. The specialisation times for all examples was under 20ms.

[2]http://www.asap.soton.ac.uk/logen

- **inter medium** - An interpreter with the environment split into two separate lists, one for the static names the other for the dynamic values.
- **inter simple** - A simple interpreter with no environment, but contains a selection of built-in arithmetic functions.
- **match** - A string pattern matcher.
- **regexp** - An interpreter for regular expressions.
- **transpose** - A matrix transpose program.

5.8 Summary

To the best of our knowledge, the first binding-time analysis for logic programming was Bruynooghe et al. (1998). The approach of Bruynooghe et al. (1998) obtains the required annotations by analysing the behaviour of an *online* specialiser on the subject program. Unfortunately, the approach was overly conservative. Indeed, Bruynooghe et al. (1998) decides whether or not to unfold a call based on the original program without taking the current annotations into account. This means that a call can either be completely unfolded or not at all. Also, the approach was never fully implemented and integrated into a partial evaluator.

In Section 6 of Leuschel et al. (2004b) a more precise BTA is presented, which has been partially implemented. It is actually the precursor of the BTA here. However, the approach was not fully implemented and did not consider the issue of filter propagation (filters were supposed to be correct). Also, the identification of unsafe calls was less precise as it did not use the binary clause semantics with program points (i.e., calls may have been classified as unsafe even though they were not part of a loop).

Vanhoof and Bruynooghe (2001a) is probably the most closely related work to ours. This work has a lot in common with ours, and we were unaware of this work while developing our present work. Let us point out the differences. Similar to Leuschel et al. (2004b), Vanhoof and Bruynooghe (2001a) was not fully implemented (as far as we know, based on the outcome of the termination analysis, the user still had to manually update the annotations by hand) and also did not consider the issue of filter propagation. Also, Vanhoof and Bruynooghe (2001a) cannot handle the **nonvar** annotation (this means that, e.g., it can only handle the vanilla interpreter if the call to the object program is fully static). However, contrary to Leuschel et al. (2004b), and similar to our approach, Vanhoof and Bruynooghe (2001a) do use the binary clause semantics. It even uses program point information to identify non-terminating calls. However, we have gone one step further in using program point information, as we will only look for loops from one program point back to itself. Take for example the following program:

```
p(a) :- q(a).
q(a) :- q(b).
q(b) :- q(b).
```

Both our approach and Vanhoof and Bruynooghe (2001a) will mark the call `q(a)` as unfoldable and the call `q(b)` in clause 3 as unsafe. However, due to the additional use of program points, we are able to mark the call `q(b)` in clause 2 as unfoldable (as there is no loop from that program point back to itself), whereas we believe that Vanhoof and Bruynooghe (2001a) will mark it as unsafe. We believe that this extra precision may pay off when specialising interpreters. Finally, due to the use of our meta-programming approach we can handle the full LOGEN annotations (such as **call**, **rescall** and **resif**) and can adapt our approach to compute memoisation loops and tackle global termination.

The papers Vanhoof (2000); Vanhoof and Bruynooghe (1999); Vanhoof et al. (2004) describe various BTAs for Mercury, even addressing issues such as modularity and higher-order predicates. An essential part of these approaches is the classification of unifications (using Mercury's type and mode information) into tests, assignments, constructions and deconstructions. Hence, these works cannot be easily ported to a Prolog setting, although some ideas can be found in Vanhoof et al. (2004).

A promising avenue for future work involves extending the system to derive the norms automatically from the propagated binding types. Decorte et al. (1993); Vanhoof and Bruynooghe (2001b) discuss the need for automatic inference of norms and in particular deriving norms from type information.

Currently our implementation guarantees correctness and termination at the local level, and correctness but not yet termination at the global level. However, the framework can easily be extended to ensure global termination as well. Indeed, our binary clause interpreter can also compute memoisation loops, and we can apply exactly the same procedure as for local termination. Then, if a memoised call is detected to be unsafe we have to mark the non-decreasing arguments as dynamic. Finally, as has been shown in Decorte et al. (1998), one can actually relax the strict decrease requirement for global termination (i.e., one can use $\leq$ rather than $<$), provided so-called "finitely partitioning" norms are used.

Chapter 6

Self-tuning Specialisation

The chapter develops a self-tuning resource aware partial evaluation technique for Prolog programs, which derives its own control strategies tuned for the underlying computer architecture and Prolog compiler using a genetic algorithm approach. The algorithm is based on mutating the annotations of offline partial evaluation. Using a set of representative sample queries it decides upon the fitness of annotations, controlling the trade-off between code explosion, speedup gained and specialisation time. The user can specify the importance of each of these factors in determining the quality of the produced code, tailoring the specialisation to the particular problem at hand. Experimental results for the implemented technique on a series of benchmarks are presented. The results are compared against the aggressive termination based binding-time analysis from Chapter 5 and optimised using different measures for the quality of code. It is shown that the technique avoids some classical pitfalls of partial evaluation.

6.1 Introduction

Despite over 10 years of research on the specialisation of logic programs, there still exist research challenges related to improving the actual specialisation capabilities (this is also true for specialisation of other programming paradigms). For example, existing specialisers do not use a sufficiently precise model of the compiler for the target system to guide their decisions during specialisation. This means that specialisers can produce specialised code that is actually slower than the original. Also, most specialisers focus solely on improving the execution speed, sacrificing other resources such as code size and memory consumption. This means that the code size and specialisation effort can be out of proportion with the actual improvement in speed.

Developing control techniques that are predictable, with reasonable specialisation complexity and that can provide a good balance between resources, is a challenging but worthwhile research objective.

This chapter presents a *self-tuning* system, which derives its own specialisation control using a genetic algorithm approach. Fitness scores are derived by actually running the specialised code and hence the particular Prolog compiler and architecture are automatically taken into account.

More precisely, we use an offline approach based on the fully automatic binding-time analysis (Chapter 5). The insight on which this self-tuning technique is based, is that the annotations can form the genes for a genetic algorithm.[1] Indeed, annotations can easily be mutated, or even merged. The key ingredients of success in our approach are:

- The fully automatic BTA provides a starting point for the algorithm. The BTA can be used to check the safety of new annotation configurations. Alternatively, based on the starting point provided by the BTA, a time-out value can be computed which can be used to discard unsuccessful mutations (where specialisation takes too long or does not terminate).
- Overall termination and convergence is guaranteed as mutations only "generalise" (**unfold** into **memo**, **static** into **dynamic**).
- Through the use of a representative sample of queries, actual figures for the particular compiler and architecture are obtained. This allows for resource aware specialisation.
- The overall trade-off between execution time, code size (and other factors such as specialisation time) can be influenced by tuning the fitness function, used to discard bad mutations.

This chapter, shows, empirically and through examples, how it avoids pitfalls which other specialisers such as ECCE (Leuschel et al., 1998) or MIXTUS (Sahlin, 1993) fall into. We also show how we can achieve a good trade-off between various resource considerations. It is also demonstrated on a series of benchmark programs the practicality and performance of the approach.

6.1.1 Other Approaches and Related Work

Such an approach has already proven to be highly successful in the context of optimising scientific linear algebra software (Whaley et al., 2001). In (Whaley et al., 2001) part of the installation procedure includes a test and feedback cycle which optimises internal parameters to give the best performance for the processor architecture, memory and cache.

[1]It is much less obvious to us how one could use a genetic algorithm to effectively optimise online specialisation.

A suitable *low-level cost model* would allow a partial evaluation system to make more informed choices about the local control (e.g., is this unfolding step going to be detrimental to performance) and global control (e.g., does this extra polyvariance really pay off).

There has been some promising initial work on cost models for logic and functional programs in Albert et al. (2001); Albert and Vidal (2002); Brassel et al. (2004); Vidal (2004). However, such a low-level cost model will depend on both the particular Prolog compiler and on the target architecture and it is hence unlikely that one can find an elegant mathematical model that is easy to manipulate and precise. It is also not entirely clear how such a cost model could be used in practice to guide specialisation. It is possible that the approach we present in this chapter could make use of a low-level cost model to determine the quality of specialised code, but a cost model may prove too inaccurate to give reliable results.

6.2 Controlling Partial Deduction

This thesis has already discussed issues relating to the control of partial deduction. The issue of control is important as it affects the correctness and termination of the specialisation process, as well as the quality of the specialised program. Considerable effort has been devoted to this crucial issue (see, e.g., the references in Leuschel and Bruynooghe (2002)), and the issue of correctness is well understood and several powerful techniques (such as homeomorphic embedding) can be used to ensure termination. However, the issue of the quality of the specialised program is still relatively open. While it is well understood that unrestricted unfolding can be detrimental to the efficiency of the specialised program, and that *determinate* unfolding can be used to avoid most pitfalls related to this, the overall picture is unclear. Indeed, using just determinate unfolding will prevent substantial efficiency gains in certain cases, and still may not prevent program slowdowns and code explosion (with a limited efficiency gain). Below we elaborate on some of the pitfalls of partial deduction in more detail, showing where it can go wrong and produce undesirable results.

6.2.1 Some Pitfalls of Partial Deduction

One pitfall related to the local control (unfolding) is known as *work duplication*. The problem is illustrated in the following example.

Let P be the program defined in Listing 6.1.

Let $A = \{$`inboth(a,L,[X,Y])`, `member(a,L)`$\}$. Performing non-leftmost non-determinate unfolding for the call `inboth(a,L,[X,Y])` in Figure 6.1 (and doing a single unfolding

```
member(X,[X|T]).
member(X,[Y|T]) :- member(X,T).
inboth(X,L1,L2) :- member(X,L1),
                   member(X,L2).
```

LISTING 6.1: The `inboth/3` example

step for `member(a,L)`), we obtain the partial deduction P' (Listing 6.2) of P with respect to A.

```
member(a,[a|T]).
member(a,[Y|T]) :- member(a,T).
inboth(a,L,[a,Y]) :- member(a,L).
inboth(a,L,[X,a]) :- member(a,L).
```

LISTING 6.2: Specialising Listing 6.1 for {`inboth(a,L,[X,Y])`, `member(a,L)`}

Let us examine the run-time goal $G = \leftarrow$ `inboth(a, [h,g,f, e,d,c,b,a], [X,Y])` (which is an instance of an atom in A). Using the Prolog left-to-right computation rule the expensive sub-goal $\leftarrow$ `member(a,[h,g,f,e,d,c,b,a])` is only evaluated once in the original program P, while it is executed twice in the specialised program P'.

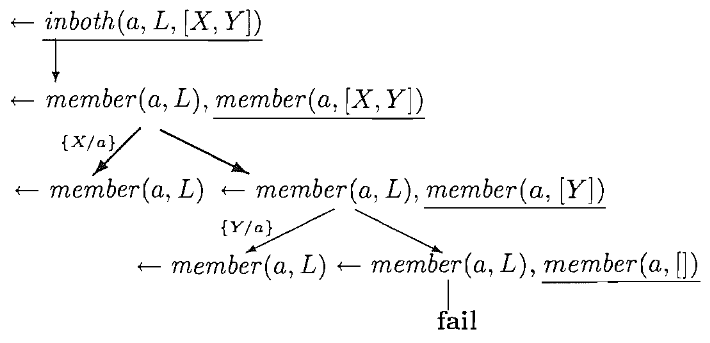

← inboth(a, L, [X, Y])
← member(a, L), member(a, [X, Y])
{X/a}
← member(a, L)
← member(a, L), member(a, [Y])
{Y/a}
← member(a, L)
← member(a, L), member(a, [])
fail

FIGURE 6.1: Non-leftmost non-determinate unfolding for Listing 6.2

The classical solution to this problem is to disallow non-leftmost unfolding unless it is deterministic (SP (Gallagher, 1993, 1991; Gallagher and Bruynooghe, 1991), ECCE (Leuschel et al., 1998)), or allow non-leftmost unfolding but not left-propagate bindings (PADDY (Prestwich, 1992), MIXTUS (Sahlin, 1993)). Some partial evaluators, for instance, SAGE (Gurr, 1994a,b) do not prevent such work duplication. This can result in huge slowdowns (see Bowers and Gurr (1995)).

However, non-leftmost non-determinate unfolding can sometimes have the opposite effect and lead to big speed-ups, which are thus prevented. Furthermore, even determinate unfolding can still lead to duplication of work, namely in unification with multiple heads:

Let us return to the program in Listing 6.1 with the set A = {`inboth(X,[Y],[V,W])`}. The query can be fully unfolded producing the partial deduction P' (Listing 6.3) of P with respect to A.

```
inboth(X,[X],[X,W]).
inboth(X,[X],[V,X]).
```

LISTING 6.3: Specialising Listing 6.1 for {`inboth(X, [Y], [V,W])`}

```
index_test(f(_),Y,Z) :- p(Y,Z).
p(a,1).
p(b,2).
p(c,3).
p(d,4).
p(e,5).
p(f,6).
p(g,7).
p(h,8).
p(i,9).
p(j,10).
```

LISTING 6.4: Example using clause indexing

No goal has been duplicated by the leftmost non-determinate unfolding, but the unification X=Y for ← inboth(X, [Y], [V,W]) has been duplicated in the residual code. This unification can have a substantial cost when the corresponding actual terms are large.

Another trap of partial deduction is the possible *loss of indexing*. Indeed, Prolog systems spend a lot of their time looking up clauses that match the current goal. When all calling arguments are free, the system has no choice but to go through the clauses one by one. However, if some of the arguments are (at least partially) instantiated then some clauses that do not match can be skipped. This is achieved using argument indexing and takes analogy from indexing in database systems. The standard Prolog indexing techniques rely on *first argument clause indexing*; that is they by default index on the first argument. Indexing can provide an important performance boost when searching over a large set of clauses.

Listing 6.4 is a a simple program with a collection of facts represented by p/2. By default indexing will be performed on the first argument of p/2, and as long as the first argument in the call to p/2 is instantiated we will benefit from the speedups of indexing.

During specialisation unfolding may change the behaviour of the clause indexing. Through unfolding, facts may be subsumed by calling predicates, whose argument orderings differ. When specialising Listing 6.4 for index_test(A,B,C) it is safe to fully unfold the call to p/2, as termination is guaranteed and it removes a level of redirection. Unfortunately in the newly created index_test__0/3 predicate (Listing 6.5), the first argument is no longer a useful basis for clause indexing and as a result, the specialised code is substantially slower than the original program (taking twice as long to complete the same benchmark).

In Ciao Prolog (and some others), the indexer allows programmers to select the argument(s) to index on. This would be an alternative to not unfolding the call, but would still require that the specialiser changes the indexing information. The classical solution

```
index_test__0(f(_), a, 1).
index_test__0(f(_), b, 2).
index_test__0(f(_), c, 3).
index_test__0(f(_), d, 4).
index_test__0(f(_), e, 5).
index_test__0(f(_), f, 6).
index_test__0(f(_), g, 7).
index_test__0(f(_), h, 8).
index_test__0(f(_), i, 9).
index_test__0(f(_), j, 10).
```

LISTING 6.5: Specialising Listing 6.4 for `index_test(A,B,C)`. The useful clause indexing has been lost

is to avoid any reordering of arguments, but this is not enough to prevent this problem. Using pure determinate unfolding (no non-determinate unfolding except at the root of an SLD-tree) together with no argument reordering avoids most of the problems. However, most determinate unfolding rules are not pure and allow one non-determinate step, this is often important for precision (see benchmarks in Leuschel et al. (1998)). This is less of an issue in conjunctive partial deduction, see Jørgensen et al. (1996).

Another related problem is the loss of indexing due to argument filtering. For example, take the following program:

```
p(f(a,b)).
p(f(b,c)).
p(f(d,e)).
p(f(e,a)).
```

Specialising for `p(f(X,Y))` produces the following specialised code:

```
p__1(a,b).
p__1(b,c).
p__1(d,e).
p__1(e,a).
```

Filtering has removed the `f/2` structure and replaced it with two arguments representing the substructure. Now, potentially the specialised program will run slower for a run-time query such as `p(f(X,a))`, provided the underlying Prolog system provides "deep" indexing (e.g., Ciao Prolog does allow this with the indexer package). This is because only the first argument is indexed, and the lookup is on the second argument in the specialised program. However, most Prolog systems only index on the top-level functor (e.g., SICStus) and hence there is actually no slow-down. In fact the program can run faster as the functor `f/2` no longer needs to be deconstructed.

The behaviour of the indexing in different Prologs is a case where depending on the Prolog the specialiser could behave differently to produce better quality code. Prolog systems also impose a maximum number of arguments. Some Prolog systems do not, but after a certain limit (e.g., 32) all further arguments are simply put into a list. As argument filtering can increase the number of arguments, this must be taken into account by the specialiser. Other differences may exist between Prologs and platforms, for example features such as tabling may influence the performance of specialised programs.

In this section we have only scratched the surface of various ways in which existing partial deduction techniques can go wrong (more pitfalls can be found in Venken and Demoen (1988), most of which are still valid today) . Also, even when partial deduction does achieve some speed improvement, this may ensue an unacceptable explosion in the code size. It is clear that deriving a good specialised program is a non-trivial pursuit, covered with many pitfalls and difficult to put into a simple mathematical model.

The motivation of this chapter is to provide a method for deriving specialisation control based on the underlying architecture guided by trial and error, providing the user with the ability to balance execution time against code explosion, or other program properties. The algorithm uses empirical measurements to tackle issues that could prove difficult to handle using a purely mathematical model. We concentrate on offline partial deduction as it provides a clear separation between specialisation and control.

6.3 Annotated Programs

As emphasised throughout this thesis, offline partial evaluation uses annotated programs to control the specialisation process. Chapter 5 introduced an automatic binding-time analysis. The analysis used state-of-the-art termination analysis techniques, combined with a type-based abstract interpretation for propagating the binding types combined. Safety of built-ins is guaranteed using a database of allowed calling patterns (with respect to the propagated binding types). The analysis was designed to be as aggressive as possible and is guided only by termination, it contains no heuristics for quality of code. The algorithm described in this chapter is designed to complement the binding-time analysis, providing control over the quality of the produced specialised programs.

Figure 6.2 is the match program taken from the DPPD library of benchmarks (Leuschel, 1996-2004). The program is a naïve string matcher; the `match/2` succeeds if the given pattern occurs in the string. The program has been annotated using the automatic binding-time analysis, the specialisation query will contain a **static** pattern but the string to search will be **dynamic**. The analysis has concluded that the first and last calls can be safely unfolded, i.e. they are guaranteed to terminate at specialisation time. The recursive call in the second `match1/4` clause has been marked memo and cannot be safely unfolded.

Using the annotations in Figure 6.2 and the specialisation query `match([a,c], A)`, the specialiser will produce Listing 6.6.

$$\begin{array}{l}
\texttt{match(Pat,T) :-}\\
\qquad \underbrace{\texttt{match1(Pat,T,Pat,T)}}_{unfold}.\\
\texttt{match1([],\_Ts,\_P,\_T).}\\
\texttt{match1([A|\_Ps],[B|\_Ts],P,[\_X|T]) :-}\\
\qquad \underbrace{\texttt{A\textbackslash== B}}_{rescall},\ \underbrace{\texttt{match1(P,T,P,T)}}_{memo}.\\
\texttt{match1([A|Ps],[A|Ts],P,T) :-}\\
\qquad \underbrace{\texttt{match1(Ps,Ts,P,T)}}_{unfold}.\\
\\
\texttt{:-filter match(static,dynamic).}\\
\texttt{:-filter match1(static,dynamic,static,dynamic).}
\end{array}$$

FIGURE 6.2: Annotated match program

```
match([a,c], A) :-  match__0(A).
match__0([A|B]) :-
        a\==A,  match1__1(B, B).
match__0([a,A|B]) :-
        c\==A,  match1__1([A|B], [A|B]).
match__0([a,c|_]).
match1__1([A|_], [_|B]) :-
        a\==A, match1__1(B, B).
match1__1([a,A|_], [_|B]) :-
        c\==A, match1__1(B, B).
match1__1([a,c|_], _).
```

LISTING 6.6: Specialising `match/2` using the annotations in Figure 6.2

6.4 Mutations

This sections examines how annotations can be mutated and thus form the basis of a genetic algorithm aimed at improving annotations.

A single set of annotations for a program is represented by an annotation configuration (Definition 6.1).

Definition 6.1 (annotation configuration). (α, β) is an annotation configuration for some program P where $\alpha \in \Sigma_c^*, \Sigma_c = \{u, m, c, r\}$, $\beta \in \Sigma_f^*, \Sigma_f = \{s, d\}$

The length of α is the number of body literals in P and the length of β is the sum of the arity of the predicates in P. A configuration represents a set of annotations for the program P. With u, m, c, r, s, and d representing **unfold**, **memo**, call, **rescall**, **static** and **dynamic** respectively.

For example, the annotations from the match program (Figure 6.2) are represented by the annotation configuration $((u,\ r,\ m,\ u,),\ (s,\ d,\ s,\ d,\ s,\ d))$.

The binding-time analysis concentrates on termination and provides a set of aggressive annotations, doing as much work as possible at specialisation time. However, this does not always produce the best specialised programs. As already discussed, there are some circumstances where it is better not to perform an operation at specialisation time or to discard some static information.

The algorithm presented searches for "better" annotation configurations which, while less aggressive than the configuration provided by the binding-time analysis, may produce better specialised code. The algorithm explores the possible *mutations* (Definition 6.2) of the current annotation configuration. A mutation of a configuration is defined as a new annotation configuration but with *one* of the annotations modified. The mutations produce new, less aggressive annotations. For example, a call marked as **unfold** can be turned into **memo**, or an argument that was previously **static** is treated as **dynamic**. This changes the behaviour of the specialiser.

Definition 6.2 (mutation). Let C be an annotation configuration for P, f_c and f_f are mapping functions defined as $f_c = \{u \mapsto m, c \mapsto r\}, f_f = \{s \mapsto d\}$. If C is of the form $(\alpha X \alpha', \beta)$ and $X \in dom(f_c)$ then the annotation configuration $(\alpha f_c(X) \alpha', \beta)$ is a mutation of C. If C is of the form $(\alpha, \beta X \beta')$ and $X \in dom(f_f)$ then the annotation configuration $(\alpha, \beta f(X) \beta')$ is a mutation of C.

Definition 6.3 (set of mutations). $mutations(C)$ is defined as the set of all possible mutations of C.

Table 6.1 shows the initial set of mutations for the match program in Figure 6.2. The initial configuration of match has five possible mutations, the mutated element has been underlined in each mutation.

Original	$((u,r,m,u),(s,d,s,d,s,d))$
1	$((\underline{m},r,m,u),(s,d,s,d,s,d))$
2	$((u,r,m,\underline{m}),(s,d,s,d,s,d))$
3	$((u,r,m,u),(\underline{d},d,s,d,s,d))$
4	$((u,r,m,u),(s,d,\underline{d},d,s,d))$
5	$((u,r,m,u),(s,d,s,d,\underline{d},d))$

TABLE 6.1: Initial set of mutations for match

It is possible that a mutated annotation configuration may be unsafe. Generalising more arguments, or memoising rather than unfolding calls, may have repercussions throughout the rest of the program. The annotation configuration may be unsafe for a number of reasons:

- The filter information may be incorrect. Marking an argument as **dynamic** or memoing a call rather than unfolding may change the propagation of static data throughout the program.
- A built-in that was previously safe to call, may now not be sufficiently instantiated at specialisation time.

- The specialisation process may fail to terminate. Information that previously guaranteed termination may have been generalised away.

Unsafe annotations will not produce valid specialised programs and are therefore of little use. Given an unsafe annotation configuration the automatic binding-time analysis algorithm can be used to find the next safe configuration. This may require that further calls are marked as memo or that the filter information is propagated correctly.

The entire binding-time analysis algorithm is complex; however, it is sufficient to run only the filter propagation and built-in safety checking. Non-termination of the specialisation process can then be monitored using timeouts. A sensible value for the timeout can be estimated using the specialisation and runtime of the original annotated program as a base.

Using the filter propagation and built-in checking on the annotations in Table 6.1 produces the new *safe* annotations in Table 6.2.

Original	$((u, r, m, u), (s, d, s, d, s, d))$
1	$((m, r, m, u), (s, d, s, d, s, d))$
2	$((u, r, m, m), (s, d, s, d, s, d))$
3'	$((u, r, m, u), (d, d, \underline{d}, d, \underline{d}, d))$
4	$((u, r, m, u), (s, d, d, d, s, d))$
5'	$((u, r, m, u), (s, d, \underline{d}, d, d, d))$

TABLE 6.2: Mutation after filter propagation

Two of the mutations have been detected as unsafe and have been modified accordingly. Figure 6.3 shows the tree of these mutations. Running the filter propagation has further mutated the annotation configuration producing new configurations with multiple mutated elements.

It is also possible to run the full binding-time analysis algorithm to find the safe set of mutations (Table 6.3). The termination analysis has detected that, in additional to the filters, one of the annotations must be changed from unfold to memo.

Original	$((u, r, m, u), (s, d, s, d, s, d))$
1	$((m, r, m, u), (s, d, s, d, s, d))$
2	$((u, r, m, m), (s, d, s, d, s, d))$
3'	$((u, r, m, \underline{m}), (d, d, \underline{d}, d, \underline{d}, d))$
4'	$((u, r, m, \underline{m}), (s, d, d, d, s, d))$
5'	$((u, r, m, \underline{m}), (s, d, \underline{d}, d, d, d))$

TABLE 6.3: Mutation after full automatic binding-time analysis

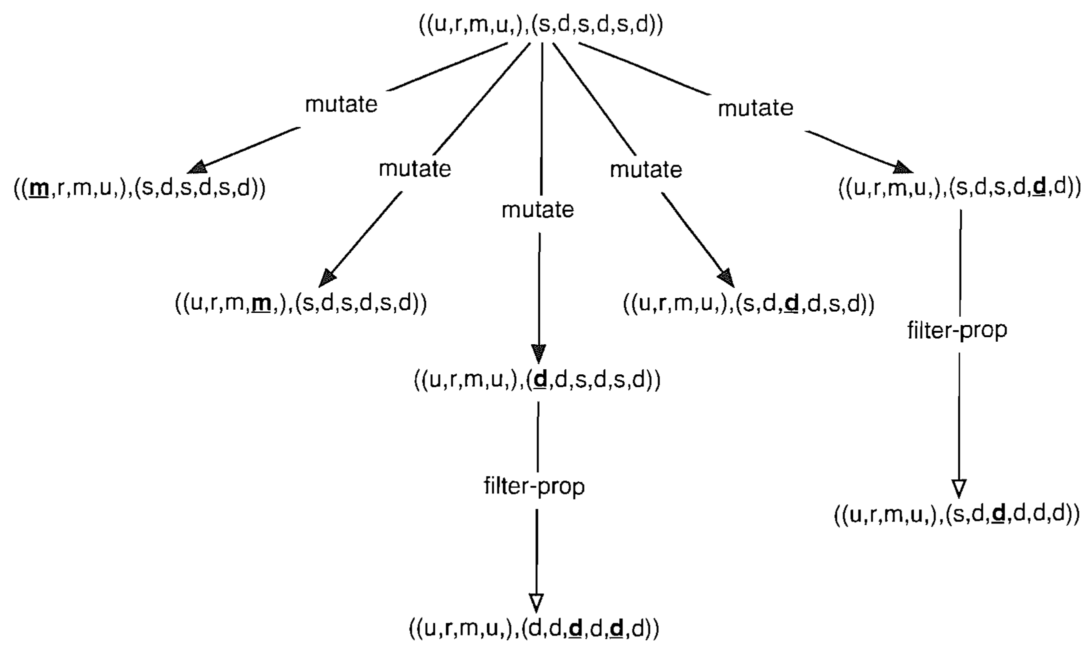

((u,r,m,u,),(s,d,s,d,s,d))
mutate
mutate
mutate
mutate
mutate
((m,r,m,u,),(s,d,s,d,s,d))
((u,r,m,m,),(s,d,s,d,s,d))
((u,r,m,u,),(d,d,s,d,s,d))
((u,r,m,u,),(s,d,d,d,s,d))
((u,r,m,u,),(s,d,s,d,d,d))
filter-prop
filter-prop
((u,r,m,u,),(d,d,d,d,d,d))
((u,r,m,u,),(s,d,d,d,d,d))

FIGURE 6.3: Safe annotation configurations after filter propagation

6.5 Deciding Fitness

To explore the search space effectively, it is essential to be able to assess the quality of a particular annotation configuration. Empirical testing is used to determine the quality of the specialised code. However, each annotation configuration can be used to specialise the same program for different sets of static data. It is impractical to test for all possible sets of of static data, so instead a representative set of sample queries is used. These queries are provided by the user. It is important that the sample queries accurately reflect the type of queries of interest as the program will be optimised with these queries in mind.

The quality of the annotation configuration is calculated using characteristics from the specialisation process:

execution time – The actual execution time of the sample queries. The sample queries are benchmarked over a number of executions to obtain a final execution time. This allows the algorithm to optimise for the fastest program.

compiled code size – The size of the produced specialised code. The size is taken after compilation into byte code. Specialisation can result in large code explosion, sometimes for a very small gain.

specialisation time – The time taken to specialise the program for the sample queries. In situations where the program is to be re-specialised frequently it may be desirable to take into account the actual specialisation time during optimisation.

It would be possible to measure additional characteristics that may be of interest to the user. For example, the memory usage during execution.

The different characteristics contain different units and cannot easily be combined. To allow comparison between the different characteristics, they are first normalised. Normalising the values against a common base case produces a new value, where 1.0 signifies it is the same as the base case, a value of 2.0 indicates it is twice as good as the base case and a value of 0.5 indicates it is twice as bad as the base case.

A fully dynamic annotation configuration (Definition 6.4) with all calls marked as rescall or memo is used as a base case. The fully dynamic annotation configuration produces specialised code which has the same behaviour as the original program, as all static data is discarded during specialisation and no calls are made at specialisation time. Each characteristic is normalised by dividing the value with the same characteristic from the dynamic annotation configuration.

Definition 6.4 (dynamic annotation configuration)**.** The annotation configuration (α, β) is fully dynamic if $\alpha \in \Sigma_{c'}^{*}, \Sigma_{c'} = \{m, r\}, \beta \in \Sigma_{f'}^{*}, \Sigma_{f'} = \{d\}$.

Where the length of α is the number of body literals in P and the length of β is the sum of the arity of the predicates in P.

While it would be possible to optimise the program for a single characteristic, much more interesting optimisations can be made by combining the different characteristics into a single score.[2] A fitness function (Definition 6.5) is used to determine the score given the characteristics.

Definition 6.5 (fitness function)**.** The fitness function is used to determine the *quality* of an annotation configuration based on its measured characteristics. The function takes as input the normalised values for specialisation time (*spectime*), execution (*speedup*) and code size reduction (*reduction*).

The choice of fitness function is important in determining the quality of code for the particular requirements. The fitness function is used to balance the trade-off between the different characteristics. A simple scoring function to find fastest specialised program would only take into account the execution time. However, sometimes the most aggressive annotations can cause dramatic code explosion with little actual gain in execution time. Using a scoring function based on both the execution time and compiled code size ensures a balance is maintained between the two characteristics.

For example, say the original program executes in $200ms$ and is $4,000$ bytes. Annotation configuration A executes in $100ms$ and is $30,000$ bytes while annotation configuration

[2]It may also be possible to use a multi-objective genetic algorithm with multiple fitness functions. Further research is needed to investigate this possibility.

B executes in $120ms$ but is only 5,000 bytes. It may be desirable to choose B, which while slightly slower is much smaller than A.

Currently the default fitness function is defined as $score = speedup^{\alpha} \times reduction^{\beta} \times spectime^{\gamma}$ where the α, β and γ values reflect the importance of the characteristics.

6.6 Algorithm

Using the concepts defined in the previous sections the complete algorithm is now presented. The algorithm is given an initial starting annotation configuration and returns the best annotation configurations found according to the set fitness function.

To explore the search space the algorithm uses a *beam search.* The beam search explores the neighbours at each node (in this case the single mutations), and only descends into the W best nodes for each level, where W is described as the *width* of the beam. The search terminates when the W best nodes remain unchanged through an iteration.

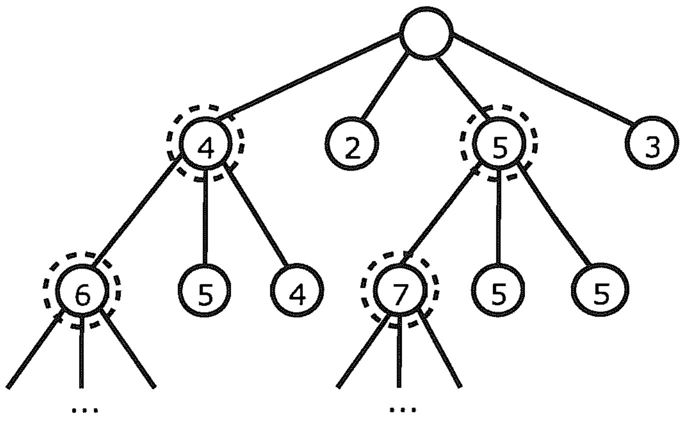


FIGURE 6.4: Beam search for $W = 2$

Figure 6.4 demonstrates the beam search for $W = 2$. The values in the nodes represent the scores, a higher score representing a better selection. At each level the search proceeds by selecting the best two solutions.

Figure 6.5 outlines the algorithm. Starting with an initial annotated program, the algorithm proceeds to find mutations of the initial configuration. Each mutation is checked for safety by running the filter propagation and then the safe configurations are benchmarked. At each iteration the best annotations are chosen and the algorithm continues. When no further improvements are found, the algorithm terminates. The depth of the search tree is bounded by the number of annotations, as at each generation at least one annotation must be made less aggressive. The filter propagation allows multiple annotations to be modified in a single step, effectively skipping levels in the search tree.

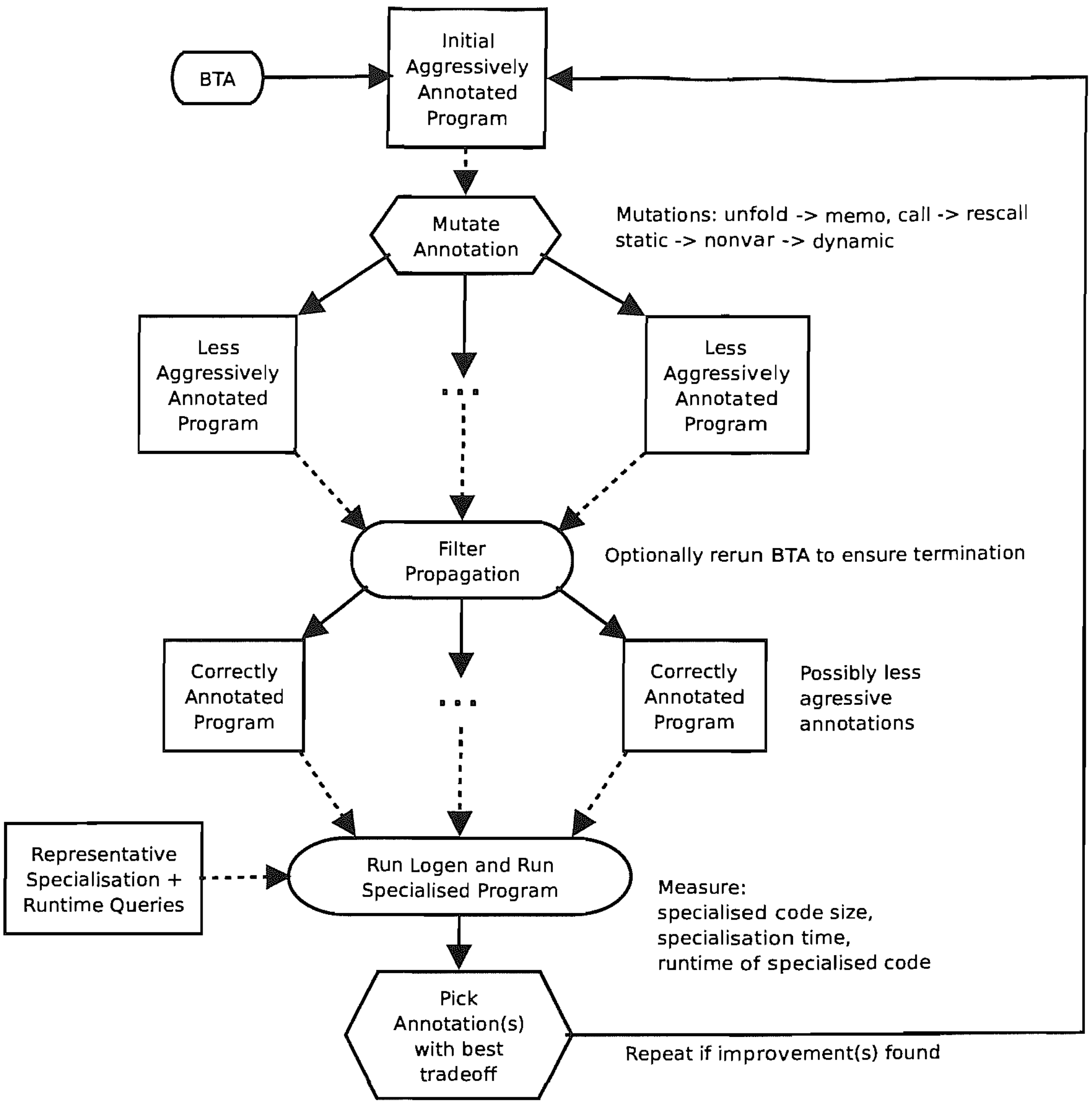

BTA
Initial Aggressively Annotated Program
Mutate Annotation
Mutations: unfold -> memo, call -> rescall
static -> nonvar -> dynamic
Less Aggressively Annotated Program
Less Aggressively Annotated Program
Filter Propagation
Optionally rerun BTA to ensure termination
Correctly Annotated Program
Correctly Annotated Program
Possibly less agressive annotations
Representative Specialisation + Runtime Queries
Run Logen and Run Specialised Program
Measure:
specialised code size,
specialisation time,
runtime of specialised code
Pick Annotation(s) with best tradeoff
Repeat if improvement(s) found

FIGURE 6.5: Self-tuning overview

Algorithm 2 describes the self-tuning algorithm in psuedo code. It uses Definition 6.6 to measure the characteristics of an annotation configuration.

Definition 6.6 (test_conf). Given a program P, an annotation configuration C, a specialisation goal G_{sp} and a runtime query G_{rt}, $test\_conf(P, C, G_{sp}, G_{rt})$ returns the tuple (ST, RT, SS). Where ST is the time taken to specialise P for the goal G_{sp}, producing the specialised program P'. RT is the execution time of P' for the goal G_{rt} and SS is the compiled code size of P'

For example, running the algorithm on the `index_test/3` example (Listing 6.4) produces the annotations in Figure 6.6. The annotations have be tuned for time and speed. The algorithm has discovered that the call should not be unfolded (as it is detrimental to

Algorithm 2 Self-tuning algorithm

```
Input:Program $P$
Input:Initial annotation configuration $C_{init}$
Input:Specialisation goal $G_{sp}$
Input:Runtime goal $G_{rt}$
Input:Beam width $W$

1: $C_{dyn}$ = fully dynamic annotation configuration for $P$
2: $(ST_{dyn}, RT_{dyn}, SS_{dyn})$ = TestConf($P$,$C_{dyn}$,$G_{sp}$,$G_{rt}$)
3: $Cache = \{C_{dyn} \mapsto fitness\_func(1, 1, 1)\}$
4: $CS = \{C_{init}\}$
5: repeat
6:   $CS_{safe} = CS$
7:   for all $C \in CS$ do
8:     $m_{safe}$ = safe set of $mutations(C)$
9:     $CS_{safe} = CS_{safe} \cup m_{safe}$
10:  end for
11:  for all $C \in CS_{safe}$ do
12:    if $C \notin dom(Cache)$ then
13:      $(ST, RT, SS)$ = TestConf($P$,$C$,$G_{sp}$,$G_{rt}$)
14:      $ST' = ST/ST_{dyn}$
15:      $RT' = RT/RT_{dyn}$
16:      $SS' = SS/SS_{dyn}$
17:      $Score = fitness\_func(ST', RT', SS')$
18:      $Cache = Cache \cup \{C \mapsto Score\}$
19:    end if
20:  end for
21:  $Previous = CS$
22:  $CS$ = Choose best $W$ configurations based on scores from $Cache$
23: until $CS = Previous$
```

performace) and has marked it as **memo**. The tuned annotations produced specialised code that is twice as fast as the aggressive annotations.

$\texttt{index\_test(f(\_),Y,Z) :- } \underbrace{\texttt{p(Y,Z)}}_{memo}\texttt{.}$

...

FIGURE 6.6: Final annotations for `index_test/3`, optimised for time and size

Figure 6.7 is the self-tuned output for the `match/2` program (Figure 6.2), optimised for both size and time. The algorithm has decided that while the first call can be safely unfolded, better code can be produced by memoing the call instead. The produced code is nearly two times smaller than the aggressive annotations and runs faster (full details can be found in Table 6.4).

$$\begin{array}{l}
\texttt{match(Pat,T) :-} \\
\qquad \underbrace{\texttt{match1(Pat,T,Pat,T).}}_{memo} \\
\texttt{match1([],\_Ts,\_P,\_T).} \\
\texttt{match1([A|\_Ps],[B|\_Ts],P,[\_X|T]) :-} \\
\qquad \underbrace{\texttt{A}\backslash\texttt{== B,}}_{rescall}\ \underbrace{\texttt{match1(P,T,P,T).}}_{memo} \\
\texttt{match1([A|Ps],[A|Ts],P,T) :-} \\
\qquad \underbrace{\texttt{match1(Ps,Ts,P,T).}}_{unfold}
\end{array}$$

```
:-filter match(static,dynamic).
:-filter match1(static,dynamic,static,dynamic).
```

FIGURE 6.7: Final annotations for `match/2`, optimised for time and size

6.7 Experiments

Table 6.4 presents the results of running[3] the self-tuning algorithm on a series of benchmarks taken from the DPPD library Leuschel (1996-2004):

advisor – A simple expert system.

inboth – The inboth example form Section 6.2.1.

index_test – The indexing example from Section 6.2.1.

match – A simple naïve pattern matcher.

missionaires – A program for the missionaries and cannibals problem.

regexp – A program testing whether a string matches a regular expression (using difference lists).

relative – A simple expert system.

vanilla_bd – A vanilla meta-interpreter, with a "contrived" object program invented by Bart Demoen.

Each test program has five enteries in the table: the original program, the program after specialising it using the annotations derived by the BTA of Chapter 5, and the results from the self-tuning algorithm with three different fitness functions:

time – The normalised time to execute the specialised program. *score = speedup*.

size – The normalised size of the byte compiled specialised program. *score = reduction*.

[3]Benchmarks were performed on a 2.5Ghz Pentium with 512MB running SICStus Prolog 3.11.1

Benchmark Program	Fitness Function	Execution Time	Compiled Size (bytes)	Specialisation Time	Optimisation Time	Attempted
advisor	original	700ms	4098	-	-	-
advisor	BTA	700ms	13929	20ms	-	-
advisor	time	430ms	9256	20ms	21s	14
advisor	size	700ms	4098	20ms	10s	16
advisor	time & size	440ms	4784	20ms	23s	16
inboth	original	850ms	1453	-	-	-
inboth	BTA	450ms	4717	20ms	-	-
inboth	time	370ms	3942	20ms	21s	20
inboth	size	820ms	1289	20ms	17s	26
inboth	time & size	470ms	1673	20ms	24s	23
index_test	original	2570ms	1753	-	-	-
index_test	BTA	5270ms	1675	20ms	-	-
index_test	time	2570ms	1753	20ms	21s	4
index_test	size	5270ms	1675	20ms	3s	4
index_test	time & size	2570ms	1753	20ms	21s	4
match	original	800ms	1037	-	-	-
match	BTA	510ms	2204	20ms	-	-
match	time	440ms	1487	20ms	7s	7
match	size	800ms	1037	20ms	5s	8
match	time & size	440ms	1487	20ms	10s	8
missionaries	original	4710ms	6701	-	-	-
missionaries	BTA	4710ms	55956	80ms	-	-
missionaries	time	3490ms	11802	60ms	2332s	505
missionaries	size	3880ms	6259	80ms	413s	688
missionaries	time & size	3830ms	6263	60ms	3386s	715
regexp	original	3540ms	1620	-	-	-
regexp	BTA	810ms	1417	20ms	-	-
regexp	time	810ms	1417	20ms	44s	19
regexp	size	810ms	1417	20ms	16s	24
regexp	time & size	810ms	1417	20ms	55s	24
relative	original	1400ms	2544	-	-	-
relative	BTA	320ms	2356	20ms	-	-
relative	time	270ms	5411	20ms	47s	33
relative	size	280ms	2364	20ms	37s	40
relative	time & size	280ms	2364	20ms	28s	22
vanilla_bd	original	430ms	9891	-	-	-
vanilla_bd	BTA	760ms	8369	20ms	-	-
vanilla_bd	time	260ms	9092	20ms	142s	21
vanilla_bd	size	760ms	8369	20ms	87s	14
vanilla_bd	time & size	260ms	8938	20ms	142s	21

TABLE 6.4: Experimental results for the self-tuning algorithm

time & size – An equally weighting of the normalised execution time and program size. *score* = *speedup* × *reduction*.

The execution time, compiled code size and specialisation time in the table are the non-normalised characteristics from Section 6.5. The optimisation time is the total time taken to find the annotation configuration, the starting configurations were provided by the BTA. The number of attempted configurations is the actual number of different annotations that were tested during the search. Note, the three enteries for the different fitness functions were timed independently of each other, in practice the cache could be reused for the different searches.

The results show that the highly aggressive configurations provided by the termination driven binding-time analysis do not neccessarily produce the best code, either in terms of code size or execution time. In both the *missionaries* and *advisor* examples the BTA configuration suffers from a code explosion for no actual gain. The *missionaries* example suffers an eight-fold increase in size, while the *advisor* example is three times larger; with neither program running any faster. The aggressive unfolding in the *index_test* example also suffers a performance penalty, the loss of the clause indexing causes the BTA configuration to run two times slower than the original. Another interesting example is *vanilla_demoen*. The purpose of the example was to show that under some circumstances meta-interpretation has the advantage of creating terms late and that removing the meta-interpretation can actually slow down the program. The algorithm here has avoided the pitfall and has actually found a specialisation that improves upon the original but does not suffer from the problem of creating terms too early.

Solely using execution time as a measure for the quality of code is not always ideal either. The *advisor*, *inboth*, *missionaries* and *relative* examples all suffer from an explosion in code size when optimised only for execution time. Balancing execution time against code size produces some interesting results. For example, the *missionaries* program's fastest solution is 35% faster than the original with an 75% increase in code size; balancing code size with execution time finds a solution which is 23% faster than the original and is also actually 7% smaller. In the three other examples, the compromise solution finds configurations which perform marginally slower than the fastest, but without the code explosion.

6.8 Summary and Future Work

This chapter has presented a self-tuning, resource-aware offline specialisation technique. The main insight was that the annotations of offline partial evaluation can be used as the basis of a genetic algorithm. Indeed, the fitness of annotations can be evaluated by trial and error using a set of representative sample queries on some target Prolog system and hardware, taking properties such as execution time and code size into account. This makes our approach both resource aware and able to fine-tune itself to new hardware or Prolog systems. Furthermore, annotations can be mutated by toggling individual clause or predicate annotations. To reduce the search space we make use of the fully automatic binding-time analysis (Chapter 5) in order to adapt unsafe mutations (of which there are many) into safe ones. The binding-time analysis also provides a valid starting point for our algorithm.

The empirical evaluation or our technique has been very encouraging. We have shown that our self-tuning algorithms avoids pitfalls of ordinary partial evaluation, while being able to find better specialised code in terms of speedup, code size or both. For example, the results show that the binding-time analysis can lead to large code explosion for little gain in efficiency, while our algorithm finds a much better trade-off.

In future it would be useful to examine whether one can use a cost model in place of the representative sample queries to evaluate the runtime of the specialised programs. Another important area of future research is the efficiency of the genetic algorithm. While searching for the final configuration, the algorithm may try many different configurations. This is costly as each configuration must be tested for safety, specialised and then benchmarked. To optimise the algorithm we must either speed up the total time taken per configuration, or reduce the number of configurations that are tested.

The benchmarking itself must produce timings with enough granularity to distinguish between the best cases, meaning that the time taken to benchmark each configuration cannot easily be reduced. In the case where a benchmark is run multiple times to produce reliable results, it may be possible to change the measurement taken, instead using the number of iterations possible in a given time period.

At each iteration in the beam search, single stage mutations are added to the set of configurations. There is currently no attempt at genetic *crossover*,[4] combining configurations with good performance in the hope of finding a better one. Of course, naïvely breeding configurations may not produce better answers, but there are situations where combining two *independent* mutations will allow the algorithm to converge on the final solution faster. Further work is needed to determine when configurations can be combined and an initial starting point could be mutations affecting different predicates, or by using some form of dependency analysis. It may also be possible to divide large programs into smaller sections for optimisation. While this can remove possible optimisations, it increases the scalability of the algorithm. Another possible way to improve the scalability is to introduce randomness into our algorithm (i.e., not compute and evaluate all possible mutations but only some random subset).

The binding-time analysis is an iterative algorithm. During the algorithm described in this chapter, the BTA is run on many different configurations to ensure that they are safe. Most of the configurations differ only slightly from ones previously analysed. The BTA algorithm, along with the specialisation process itself, could be modified to reuse previous intermediate results. If a subset of a program has been seen before (with the same annotations) then it is possible some of the analysis can be reused. This should provide good opportunities to speed up the safety analysis for each configuration.

The system lends itself well to parallelisation. The different configurations can be tested on different machines. Care must be taken in the interpretation of the results, since the algorithm tunes towards the performance of the installed Prolog system and underlying architecture. While the results can be normalised between machines of differing speeds

[4]Strictly speaking our current algorithm is actually closer to an evolutionary algorithm rather than a genetic algorithm Eiben and Smith (2003).

providing a fair indication of speed, it will not take into account any differences in the actual architecture, which may affect performance. Initial results of parallelisation look promising; running the *missionaries* example on two computers (with similar specifications) produces a 96% improvement in execution time compared with the execution time on a single machine. Further investigation is needed to fully explore this avenue. In work of Sperber et al. (1997), the specialisation process itself was parallelised, distributing the work over a network of work stations producing some good results.

Chapter 7

Extending Specialisation Techniques

To take specialisation to a wider audience of real users it is important to support the features of modern Prolog implementations. As well as performing traditional pure declarative partial deduction LIX has been extended to support many non-declarative features as well as major Prolog extensions. Coroutining and constraint logic programming are two of the recent major extensions to modern Prolog implementations. The specialisation of constraint logic programs will be discussed separately in Chapter 8. This chapter focuses on coroutining and introduces a new annotation based on the idea of delayed execution.

7.1 Coroutines

Coroutines extend the traditional Prolog selection rule from 'leftmost goal' to the 'leftmost *unblocked* goal'. A goal is *blocked* as long as the blocking condition remains unsatisfied. The definition from the SICStus Prolog manual is given in Figure 7.1.

```
when(+Condition,:Goal)
    Blocks Goal until the Condition is true,
    where Condition is a goal with the restricted syntax:
    nonvar(X)
    ground(X)
    ?=(X,Y)
    Condition,Condition
    Condition;Condition
```

FIGURE 7.1: when/2 definition from the SICStus Prolog manual

Delaying the execution of a goal allows, for example, the user to program in a declarative manner without worrying about the instantiation patterns of potentially unsafe or non-terminating calls. For example, is/2 can only be called with the second argument ground, otherwise an instantiation exception will be thrown at runtime. As a result the user often has to carefully decide the order of statements so that the arguments are correctly instantiated.

7.1.1 Coroutining Example

We demonstrate a common use of coroutining using the is/2 built in. The ground_max/3 predicate given in Listing 7.1 instantiates the third argument with the maximum of the first two arguments.

```
ground_max(X,Y,Z) :-
          Z is max(X,Y).
```

LISTING 7.1: Predicate to find maximum of X and Y providing they are both ground

However, ground_max/2 is only safe if both the first two arguments are ground when the call is encountered. Otherwise an exception will be generated as demonstrated in Listing 7.2.

```
| ?- ground_max(X,Y,Z), X = 2, Y = 3.
! Instantiation error in argument 2 of is/2
! goal:  _79 is max(_76,_77)
```

LISTING 7.2: Listing 7.1 throws an instantiation error if called with unground variables

The second argument to is/2 must be ground before the call is made. Using the leftmost selection rule produces an exception in the example, reordering the clauses would fix this error (Listing 7.3).

```
| ?- X = 2, Y =3, ground_max(X,Y,Z).
X = 2,
Y = 3,
Z = 3 ?
yes
```

LISTING 7.3: Reordering the goals from Listing 7.2 fixes the error

Instead of manually reordering the clauses, which is not always possible, a more declarative solution is to use coroutines. If we define a blocking condition for the call to is/2 (Listing 7.4) we do not have to worry about the order of the literals. When the call becomes instantiated enough it will be executed and the results correctly propagated. The definition of ground_max/2 can be rewritten to use when/2 (Listing 7.5).

```
| ?- when(ground((X,Y)), Z is max(X,Y)), X = 2, Y = 3.
X = 2,
Y = 3,
Z = 3 ?
```

```
yes
```

LISTING 7.4: Using coroutines delays the execution of the is/2 until is is correctly instantiated

```
max(X,Y,Z) :-
    when(ground((X,Y)), Z is max(X,Y)).
```

LISTING 7.5: Predicate to find maximum of X and Y using coroutines to guarantee groundness

7.1.2 Specialising Coroutines

The specialisation techniques introduced in this thesis must be extended to support coroutines. Using the existing annotations one must either annotate the `when/2` predicate as call or **rescall**.

For these annotation to be safe a `when/2` marked as **call** must only contain built ins, and must be called and triggered during the execution of the current branch.

A `when/2` marked as **rescall** must again contain only built ins, as calls to user predicates would not be specialised, and the `when/2` would simply become part of the residual code. This condition can be relaxed to allow user defined predicates using the same technique as `findall/3` in Section 3.3.2. The LIX code can easily be extended, Listing 7.6, to handle the residual `when/2`, **reswhen**, annotation.

```
body(reswhen(Condition,Goal), when(Condition,RGoal)) :-
    body(Goal,RGoal).
```

LISTING 7.6: Extending LIX for the **reswhen** annotation

Annotating the call to `when/2` in Listing 7.5 as **reswhen** and specialising the program for the call `max(X,3,Z)` produces the residual code in Listing 7.7. There is room for further improvement by reducing the condition `ground((A,3))` to `ground(A)`.

```
max__0(A, B) :-
        when(ground((A,3)), B is max(A,3)).
```

LISTING 7.7: Specialising `when_max/2` from Listing 7.5 for the goal `max(X,3,Z)`

However, Listing 7.6 will never remove the overhead of the `when/2` even when it is safe to do so. For instance, specialising the call `max(2,3,Z)` will produce an unneeded `when/2` (Listing 7.8). By checking the blocking condition before producing the residual code, Listing 7.9, we can produce better code. If the condition is satisfied only the goal is produced, otherwise the entire `when/2` is reproduced.

```
max__0(B) :-
        when(ground((2,3)), B is max(2,3)).
```

LISTING 7.8: Specialising `when_max/2` from Listing 7.5 for the goal `max(2,3,Z)`, produces unneeded `when/2`

```
body(reswhen(Condition,Goal), ResidualCode) :-
     body(Goal,RGoal),
     (Condition ->
               ResidualCode = RGoal
     ;
               ResidualCode = when(Condition,RGoal)
     ).
```

LISTING 7.9: Extending Listing 7.6 for simple case where condition is already satisfied

The static **when** annotation executes the `when/2` during specialisation (Listing 7.10). The user must guarantee that the `when/2` condition will be satisfied (it will become unblocked) during specialisation or the code will be lost. The limitations of the static **when** annotation will be addressed in Section 7.1.3.

```
body(when(Cond,Call), Res) :-
        when(Cond, body(Call, BRes)).
```

LISTING 7.10: Extending LIX for the **when** annotation

7.1.3 semiwhen

The extension of Listing 7.9 allows for dynamic use of the `when/2` predicate. However executing the `when/2` during specialisation time is more complicated, requiring the user to carefully check the `when/2` will be satisfied during the specialisation of the current branch. A better solution is to use `call_residue/2` to check for blocked coroutines, providing a more flexible method for specialising coroutines.

We extend the specialiser to support a **semiwhen** annotation. The `when/2` will be executed during specialisation and if the condition becomes satisfied then the call will be executed under the control of the specialiser. If the condition is not satisfied while specialising the current branch then the goal will be removed from the blocked state, and the correct residual code will be generated to produce the `when/2` predicate in the final residual code.

The `body/2` predicate from LIX is extended to execute the `when/2` during specialisation. Depending on whether or not the `when/2` succeeds a different action is performed.

- If the `when/2` succeeds then `body/2` is called on the blocked call. The call is performed during specialisation and `Res=BRes` links up any residual code.
- If the `when/2` has not been triggered after all the branch has been computed then `call_residue/2` will return any blocked goals. The predicate `inspect_residual/1` matches blocked goals and produces a new `when/2` declaration for the residual code. The `Res=BRes` acts as a place holder, and `Res` is instantiated with the final residual code.

```
body(semiwhen(Cond,Call), Res) :-
        when(Cond, (Res=BRes,body(Call, BRes))).
memo(A, B) :-
        (   memo_table(A, B) ->
            true
        ;   generalise_and_filter(A, C, D),
            assert(memo_table(C,D)),
            findall((D:-E)
            (
                call_residue(unfold(C,E), Blocked),  %%% Catch residual
                inspect_residual(Blocked),           %%% search residual for when
            ),
            F),
            format('/*~k=~k*/~n', [D,C]),
            pretty_print_clauses(F),
            memo_table(A, B)
        ).

inspect_residual([]).
inspect_residual([Res|Tail]) :-
        Res = _-(prolog:when(_,Cond,_Module:(Res=BRes,Call))),  %% untriggered when
        !,
        make_residual(Call, ResCall),  %%% produces residual code from call
        Res = when(Cond,ResCall)       %%% rebuild the when statement
        inspect_residual(Tail).
```

LISTING 7.11: Extending LIX for the **semiwhen** annotation

7.1.4 Specialisation Example

We demonstrate the **semiwhen** annotation using the definition of not_eq/2 in Listing 7.12. Execution of the inequality is delayed until both arguments are ground, ensuring the call will be correctly instantiated.

```
not_eq(A,B) :-
    when(ground((A,B)), A \= B).
```

LISTING 7.12: **semiwhen** example: not_eq/2 will delay until both arguments are ground

If the call to when/2 is annotated as **semiwhen** then the specialiser will attempt to perform the operation if the goal becomes unblocked during specialisation. If however the goal remains blocked the when/2 will be regenerated as part of the residual code. The results of specialising Listing 7.13 for different queries are shown in Listing 7.14, Listing 7.15 and Listing 7.16.

```
test(A,B,C) :-
    not_eq(A, a),
    not_eq(B,b),
    not_eq(C,c),
    B = d. %call made at specialisation time
```

LISTING 7.13: Test query for not_eq/2 in Listing 7.12

In Listing 7.14 the program has been specialised for the goal `test(A,B,C)`. One of the calls to `not_eq(B,b)` has been removed as it became fully ground during specialisation, triggering the `when/2` and as the values are not equal it succeeded. The remaining `when/2` calls have been rebuilt in the residual code.

```
test__0(A, d, B) :-
        when(ground((A,a)), A\=a),
        when(ground((B,c)), B\=c).
```

LISTING 7.14: Specialising Listing 7.13 for the goal `test(A,B,C)`

Listing 7.15 is specialised for the goal `test(b,B,C)`. This time the first argument is ground and an additional `when/2` can be removed.

```
test__0(d, A) :-
        when(ground((A,c)), A\=c).
```

LISTING 7.15: Specialising Listing 7.13 for the goal `test(b,B,C)`

Listing 7.16 is specialised for the goal `test(a,B,C)`. The first argument is again ground, the `when/2` is triggered but this time the inequality test fails. This produces correctly failing residual code.

```
test__0(_, _) :-
        fail.
```

LISTING 7.16: Specialising Listing 7.13 for the goal `test(a,B,C)`

7.2 Online Annotation

The **semiwhen** annotation introduces some online control to the partial evaluator. If the blocking condition is satisfied then the call will be specialised otherwise it will be left in the residual code. This idea can be extended into a new annotation. This section introduces the **online** annotation.

Offline partial evaluators make the majority of their decisions based not on the actual static data values but on their binding types. The partial evaluator is then driven by an annotated source file, providing predictable results and offering the annotator full control over the specialisation process.

In contrast online partial evaluators make their decisions based on the actual values of the static data. This makes them potentially more powerful but much less predictable in practice. The **online** annotation attempts to combine some of the predictability and control over specialisation offered by offline techniques but using the actual static data values rather than an approximation.

The idea behind the **online** annotation is to annotate a safe call pattern for the calling of a predicate rather than annotating each individual call to the predicate. Specifying the instantiation pattern for making a call during specialisation is very similar to specifying the guard condition for coroutining. Combining the power of coroutining with the annotation also creates a very flexible selection strategy, we simply specialise the call for which we have sufficient information to do something useful. This hopefully propagates some results which may trigger a guard for another blocked call.

Take for example the classical append program (Listing 7.17). In our offline partial evaluation setting the recursive call to app/3 is either marked **unfold** or **memo** depending on how the program is called. Using a mono-variant analysis the call must be marked **memo** if any call in the program does not provide sufficiently instantiated information, in the case of app/3 the first or last argument must be a well formed list.

```
app([],B,B).
app([H|T], B,[H|T1]) :- app(T,B,T1).
```

LISTING 7.17: app/3 is the classical predicate to append two lists together

Using our **online** annotation we specify a safety condition for making the call to app/3 once and it is checked in an online fashion for all calls to app/3 in the program. As the recursive call will be subject to the same safety criteria we can relax the need for a well formed list. It is sufficient that either the first or last argument is **nonvar**, implying that at least the top level functor is known. Listing 7.18 specifies the two safety requirements for a call to app/3.

```
:-is_safe(app(A,_,_),nonvar(A)).
:-is_safe(app(_,_,A),nonvar(A)).
```

LISTING 7.18: Safety declarations for app/3 in Listing 7.17

The offline specialisation algorithm is now augmented with some decisions based on actual static data values. Specialising app/3 for the call app(A, B, C), supplying no static information correctly produces Listing 7.19. As no useful static data was supplied the code is identical, after renaming, to the original program. This is the same as marking the recursive call as **memo** but the decision has been made based on the actual data.

```
app__0([], A, A).
app__0([A|B], C, [A|D]) :-
        app__0(B, C, D).
```

LISTING 7.19: Specialising Listing 7.17 for the goal app(A, B, C)

Specialising app/3 for the call app([a,b,c], B, C), supplying the first argument to be a valid list produces Listing 7.20. The recursive call has been fully unfolded and the redundant argument has been removed. This is the same as marking the recursive call as **unfold**.

```
app([a,b,c], A, B) :-
        app__0(A, B).
app__0(A, [a,b,c|A]).
```

LISTING 7.20: Specialising Listing 7.17 for the goal app([a,b,c], B, C)

Specialising app/3 for the call app(A, B, [a,b,c|C]), supplying a partial list for the third argument, produces Listing 7.21. The recursive call has been unfolded where enough information was available but has been correctly memoed when there was not sufficient information to safely unfold. This would not be possible without the **online** annotation as the call would have to be marked as **memo** and the first argument would be classed as dynamic. It would therefore not be possible to unfold the partial list.

```
app__0([], [a,b,c|A], A).
app__0([a], [b,c|A], A).
app__0([a,b], [c|A], A).
app__0([a,b,c|A], B, C) :-
        app__1(A, B, C).
app__1([], A, A).
app__1([A|B], C, [A|D]) :-
        app__1(B, C, D).
```

LISTING 7.21: Specialising Listing 7.17 for the goal app(A, B, [a,b,c|C])

In addition to introducing online decisions into the specialiser the **online** annotation also provides delayed unfolding. We use test_delay/3 defined in Listing 7.22 to demonstrate this. The predicate contains calls to app/3 followed by a unification which will instantiate one of the arguments. It is only when the argument is instantiated that the app/3 is safe to unfold.

```
test_delay(A,D,E) :-
    app(A,B,C),
    app(C,D,E),
    C = [a,b,c,d].
```

LISTING 7.22: Predicate to test delayed unfolding of Listing 7.17

Specialising Listing 7.22 for the goal test(A, B, C) produces Listing 7.23. When first encountered, the calls to app/3 were not sufficiently instantiated to safely unfold. After instantiating C, the guard condition for unfolding is triggered on both blocked calls. This allows the code to be correctly unfolded producing the specialised program (Listing 7.23). Doing this without the **online** annotation would require careful modification of the selection strategy to ensure the correct propagation.

```
test_delay__0([], A, [a,b,c,d|A]).
test_delay__0([a], A, [a,b,c,d|A]).
test_delay__0([a,b], A, [a,b,c,d|A]).
test_delay__0([a,b,c], A, [a,b,c,d|A]).
test_delay__0([a,b,c,d], A, [a,b,c,d|A]).
```

LISTING 7.23: Specialising Listing 7.22 for the goal test(A, B, C)

7.3 Summary

This chapter explored coroutining and specialisation. The partial evaluator has been extended to handle the when/2 predicate. A call to when/2 can be annotated as **when**,

reswhen or **semiwhen** depending of if the call is **static**, **dynamic** or in the case of **semiwhen** a decision will be made during specialisation.

Using delayed execution the **online** annotation was introduced. The **online** annotation allows the user to annotate the program by specifying *safe* call patterns for unfolding a call instead of annotating every call in the program. The specialiser uses this information to check the actual static values at specialisation time, and makes an online decision. The **online** annotation also extends the selection rule used during specialisation, instead of choosing the left most goal for specialisation we specify guard conditions for each call. The specialiser can then choose the call for which it has sufficient information.

Chapter 8 will explore constraint logic programming (clp) and extend the specialisation techniques to handle clp programs.

Chapter 8

Specialisation of Constraint Logic Programming Languages

The work in this chapter has been previously published as Craig and Leuschel (2003) for the Andrei Ershov Fifth International Conference, Perspectives of System Infomatics.

Constraint logic programming extends traditional logic programming to include reasoning about relationships or 'constraints' in a particular domain. CLP(Q) offers a powerful constraint solver for the domain of rational numbers. The basic specialisation technique for CLP(Q) programs is given and is shown to handle non-declarative features. This chapter presents implementation details along with experimental results.

8.1 Introduction

Constraint logic programming (CLP) over the real domain, CLP(R), and the rational domain, CLP(Q), offer a powerful mathematical solver for the domains of real and rational numbers. The CLP(R) and CLP(Q) schemes used in this chapter and related tool are instances of the general constraint logic programming scheme introduced by Jaffar & Michaylov Jaffar et al. (1991).

CLP languages allow the programmer to express the problem in a very high level language, specifying relationships between objects, while the underlying engine uses powerful incremental constraint solvers.

Expressing a problem as a set of relations can be a more natural and declarative way of solving the problem. For example, Newton's second law (Figure 8.1) expresses the relationship between the force, mass and acceleration of an object.

Listing 8.1 is a Prolog encoding of Newton's 2^{nd} law using constraints. The encoding is very natural, the actual constraint exactly matches the initial definition in Figure 8.1.

$$Force = Mass \times Acceleration$$

FIGURE 8.1: Newton's 2^{nd} law specifies a relationship between force, mass and acceleration.

Two additional predicates have been added which build upon the encoding of Newton's law. This example has quickly built up a more complex set of relations by adding more constraints in a declarative manner.

```
:- use_module(library(clpq)).
newton(Force, Mass, Acceleration) :-
        {Force = Mass * Acceleration}.

moon(Weight, Mass) :-
        {G = 9.8, MoonG = G/6},
        newton(Weight, Mass, MoonG).

earth(Weight, Mass) :-
        {G = 9.8},
        newton(Weight, Mass, G).
```

LISTING 8.1: CLP version of Newston's 2^{nd} law. Two new relations, earth/2 and moon/2, that use newton/3 have been created.

The code in Listing 8.1 can be used to compare the weight of an object on the Moon and on the Earth (Listing 8.2). CLP can be used in a declarative manner, either the weight or the mass can be provide to calculate an answer. In fact `moon/2` can be called with no instantiated arguments giving a new relationship between the variables (Listing 8.3).

```
| ?- moon(WM, Mass), earth(WE, Mass), {Mass = 30}.
WE = 294,
WM = 49,
Mass = 30 ?
yes
```

LISTING 8.2: Listing 8.1 can be used to compare the weight of an objection on the Earth and the Moon

```
| ?- moon(W, M).
{M=30/49*W} ?
yes
```

LISTING 8.3: Specifying two uninstantiated variables produces a relation between the variables without needing to ground them

Despite some recent interest, there has been surprisingly little work on the specialisation of constraint logic programs. Indeed after some work in the early 90's (Smith, 1991; Smith and Hickey, 1990) there has been a long period of relative inactivity, especially compared to the success that constraint logic programming has encountered for practical applications. Only very recently, new research is emerging (Fioravanti et al., 2000, 2001; Peralta and Gallagher, 2002; Tao et al., 1997) which is trying to tackle this difficult but practically relevant problem.

This chapter presents an introduction to partial evaluation of constraint logic programs (CLP) and presents a newly developed technique and its implementation. The technique is implemented and demonstrated using several examples to evaluate the power and efficiency of the system. This work presents the first offline specialiser for CLP, and it is also the first compiler generator for CLP. The goal was to develop a system with fast and predictable specialisation times and to ensure wide applicability it also caters for non-declarative features.

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

The partial evaluator is extended to handle full CLP(R) or CLP(Q) programs. Supporting constraint specialisation across predicates by memoising constraints and retains the full power of the specialiser on ordinary logic programming constructs. The next section explains how this is achieved.

8.2 Specialisation of pure CLP(R) and CLP(Q) programs

Algorithm 1 from Chapter 2 is extended to specialise CLP programs. The modified algorithm is split into two: the main loop (Algorithm 3) and the `STEP` function (Algorithm 4).

This section uses a *projection* operation, this "projects" a set of constraints onto a set of variables. This removes irrelevant constraints, and returns a set of constraints relating to the projected variables. For example, projecting the constraints {`X = Y` $\wedge$ `Y >2`} onto the variable `X` would give the constraint {`X>2`}. The projection operation also performs constraint simplification.

8.2.1 Memoisation

Algorithm 1 in Chapter 2 used a memoisation table to store pairs of generalised and filtered atoms ($\langle A^G, A^F \rangle$). The filtered atom, A^F, is a unique identifier to the residual predicate for the specialised query A^G. The residual predicate A^F can only be reused if the current goal, after generalisation, is an instance of the stored generalised goal A^G.

For example, a memo table entry ⟨ `p(2,X,Y)`, `p_1(X,Y)` ⟩ can be reused by a memoed call to `p(2,a,A)` but not by the call `p(A,B,C)` (as `p(A,B,C)` is *more general* than `p(2,X,Y)`). If there is no matching entry in the memoisation table then a new entry must be added. The current goal is generalised and filtered and added to the table. The main loop selects unmarked entries in the table and specialises them independently, creating the required residual code.

In the CLP setting the memoisation table must be extended to take the current constraint store into account. As a CLP variable may be uninstantiated but still bound to constraints it cannot always be clearly marked as **static** or **dynamic**. A new binding type, **constraint**, is introduced so that constraints can be propagated throughout the program. The **constraint** binding type is effectively the same as the **semi** binding type described in Chapter 4. The binding type **semi** guarantees that the variable will not be less instantiated at runtime, the variable is kept during generalisation and is only filtered if the value is completely ground. A variable can still be marked **dynamic** to limit the propagation of constraints.

Algorithm 3 Offline Partial Deduction with Constraints

```
Input: A Program $P$ and an atom $A$
Global: $MemoTable = \varnothing$
1: generalise $A$ to give $A^G$
2: filter $A^G$ to give $A^F$
3: add $\langle A^G, A^F, \varnothing \rangle$ to $MemoTable$
4: repeat
5:     select an unmarked tuple $\langle A^G, A^F, CS \rangle$ in $MemoTable$ and mark it
6:     {CS represents a stored set of constraints with the memo entry}
7:     STEP($A^F$, $\langle\rangle$, $\langle\langle$unfold , $A^G\rangle\rangle$, $CS$)
8: until all tuples in $MemoTable$ are marked
```

When a goal, Q, is memoised for the first time the current constraints, projected onto the arguments of Q, must also be stored. When the goal stored in the memoisation table is selected for specialisation the initial constraints stored in the table will be reused, therefore a memoised call can only be reused if the current constraint store is at least as restrictive as the stored constraints.

For example, a call `p(X,Y,a)` is memoised with the constraint store {`X>2, T<7`}. The filter for p/2 declares the first argument as a **constraint**, the second **dynamic** and the last argument **static**. The generalised call is `p(X,Z,a)`, the filtered call is `p_1(X,Z)` and the projected constraint store is {`X>2`}. The entry ⟨`p(X,Z,a)`, `p_1(X,Z)` , {`X>2`}⟩ is stored in the memoisation table and `p(X,Z,a)` will be specialised with the initial constraints {`X>2`}. This entry can only be used by matching goals, i.e. goals that are an instance of `p(X,Z,a)` *and* have constraints that are at least as restrictive as {`X>2`} (e.g. {`X>3`} but not {`X>0`}).

Algorithm 4 STEP function with Constraints

```
function STEP($Q, B, C, CS$)
  {$Q$ is current goal}
  {$B$ is current residual code}
  {$C$ is remaining annotated atoms}
  {$CS$ is the current constraint store}
  if $C$ is $\varepsilon$ then
    $CS' = project_{vars(Q,B)}(CS)$
    $\exists\langle A^{G'}, A^{F'}, A^{CS}\rangle$ s.t. $Q$ unifies with $A^{F'}$ with substitution $\theta$
    $CS''$ = remove constraints from $CS'\theta$ entailed by $A^{CS}\theta$
    $B' = CS'' \wedge B\theta$
    pretty print the clause $Q$:-$B'$
  else
    let $B = \langle A_1, ..., A_i\rangle$
    let $C = \langle\langle Ann_1, AA_1\rangle, ..., \langle Ann_j, AA_j\rangle\rangle$
    if $Ann_1$ is memo then
      generalise $AA_1$ to give $A^G$
      if $\exists\langle A^{G'}, A^{F'}, CS'\rangle \in MemoTable$ s.t. $AA_1$ is a variant of $A^{G'} \wedge CS \rightarrow CS'$ then
        {$A^G$ has been previously added with an entailed set of constraints. Compute call to residual predicate}
        $\theta = mgu(AA_1, A^{G'})$
        $A^F = A^{F'}\theta$
      else
        {Compute residual predicate head and add call to pending list}
        filter $A^G$ to give $A^F$
        remove non-linear constraints from $CS$ and project onto variables of $A^F$ to give constraints $CS'$
        add $\langle A^G, A^F, CS'\rangle$ to $MemoTable$
      end if
      STEP($Q, \langle A_1, ..., A_i, A^F\rangle, \langle\langle Ann_2, AA_2\rangle..\langle Ann_j, AA_j\rangle\rangle, CS$)
    else if $Ann_1$ is unfold then
      for all $Head$:-$AnnBody$ in program $P$ do
        if $AA_1$ unifies with $Head$ giving $mgu$ $\theta$ then
          $\theta = mgu(Head, AA_1)$
          let $BA'$ = concat($AnnBody, \langle Ann_2, AA_2\rangle, \ldots, \langle Ann_n, AA_n\rangle$)
          STEP($Q\theta, B\theta, BA'\theta, CS$)
        end if
      end for
    else if $Ann_1$ is constraint then
      if $CS \wedge AA_1$ is consistent then
        STEP($Q, B, \langle\langle Ann_2, AA_2\rangle..\langle Ann_j, AA_j\rangle\rangle, CS \wedge AA_1$)
      end if
    end if
  end if
end function
```

Non-linear constraints

The CLP(Q) system is restricted to solve only linear constraints because the decision algorithms for general non-linear constraints are prohibitively expensive to run (Jaffar et al., 1991). However non-linear constraints are collected by the CLP(Q) engine in the hope that through the addition of further constraints they might become simple enough to solve. In Listing 8.4 the constraint store has not failed but has become inconsistent, there are no values of X that will satisfy the set of equations. During memoisation a part of the constraint store is stored along with the residual call and subsequent calls are checked against the stored constraint set. Non-linear constraints cannot be tested for entailment and calculating a convex hull for non-linear constraints is expensive. Therefore only linear constraints are stored inside the memoisation points, non-linear constraints are simplified and added to the residual code.

```
| ?- {X * X = Y , X * X = Z, Z + 2 = Y }.
{Z= -2+Y},
clpq:{-(Z)+X^2=0},
clpq:{X^2-Y=0} ?
yes
```

LISTING 8.4: Non-linear constraints can lead to an inconsistent constraint store

Collecting constraints

During unfolding constraints are propagated throughout the program. In memoisation these constraints are collected in constraint collection points, simplified and added to the residual program. Listing 8.5 is an extract from a CLP program, CS_{Label} represents a set of constraints. The call p/2 in r/3 has been annotated as **memo** and the calls to foo/1 in both the p/2 clauses have been annotated as **unfold**. The clauses for foo/1 are not shown in the example.

```
r(X,Y,Z) :- CS_r1, memo(p(X,Y)), CS_r1.1.
p(A,B) :- CS_p1, unfold(foo(A)).
p(A,B) :- CS_p2, unfold(foo(B)).
```

LISTING 8.5: Extract from a CLP program, CS_{Label} is a set of constraints

For simplicity it is assumed that the initial constraint store projected onto X,Y,Z is empty when the call to r(X,Y,Z) is made. The constraints CS_{r1} are added to the constraint store and the memo-ed call to p(X,Y) is encountered. In Listing 8.6 the linear constraints from CS_{r1} projected onto the variables in p(X,Y) are stored in the memo table along with the initial entry point for r(X,Y,Z).

$CS_{mp} = removeNonLinear(project_{vars(p(X,Y))}(CS_{r1}))$

```
memo_table(p(X,Y) , p_1(X,Y), CS_mp).
memo_table(r(X,Y,Z), r_1(X,Y,Z), ∅).
```

LISTING 8.6: Memoisation table entry for p in Listing 8.5

The constraints $CS_{r1.1}$ are added to the constraint store and the residual clause for r_1/3 is created. All constraints are collected and projected onto the variables occurring in the clause and added to the residual code. Listing 8.7 is the final residual code for r_1/3.

$CS_{rspec} = project_{X,Y,Z}(CS_{r1} \wedge CS_{r1.1})$

```
r_1(X,Y,Z) :- CSrspec, p_1(X,Y).
```

LISTING 8.7: Specialised fragment of Listing 8.5

The two clauses for p/2 are specialised using the initial constraints CS_{mp}, the calls to foo/1 are unfolded and they become part of the residual code. As the constraint set CS_{mp} specifies a precondition for all calls to the residual code, p_1(X,Y), all residual constraints entailed by CS_{mp} can be safely removed from the final code. If a subsequent call is made to p(X,Y) it may reuse the specialised code p_1(X,Y) if and only if the current linear constraints projected onto the variables X and Y are at least as restrictive as CS_{mp}. The final code for p_1/2 is shown in Listing 8.8.

$CS_{pspec1} = removeEntailed(CS_{mp}, CS_{p1} \wedge unfold(foo(A), CS_{p1} \wedge CS_{mp}))$
$CS_{pspec2} = removeEntailed(CS_{mp}, CS_{p2} \wedge unfold(foo(B), CS_{p2} \wedge CS_{mp}))$

```
p_spec[1](A,B) :- CSpspec1.
p_spec[2](A,B) :- CSpspec2.
```

LISTING 8.8: Specialised fragment of Listing 8.5

8.2.2 Unfolding with Constraints

The classical unfold transformation replaces a predicate call with the predicate body, performing all the needed substitutions. In CLP the state of the uninstantiated variables is held in the constraint store. During unfolding constraints are collected and propagated through the program. The constraints are then collected and simplified at the enclosing memoisation point (the top-level entry point is treated as a memoised call).

Let us examine the trivial CLP(Q) program in Listing 8.9, which naïvely multiplies X by an integer Y to give R. Figure 8.2 demonstrates how to unfold this program for the call multiply(X,2,R). After each recursive call to multiply, a new constraint is added to the constraint store ($\mathrm{C}_{1..3}$). After the unfolding is complete it is not only necessary to extract the computed answer substitution but also the final residual constraints held in C_3. These constraints are then projected onto the variables X and R of the top-level query and simplified to produce the residual program in Listing 8.10.

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

LISTING 8.9: Trivial CLP(Q) multiplication predicate

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

LISTING 8.10: Specialisation of CLP multiply Listing 8.9

Careful attention must be paid to the simplification of the residual constraints. During unfolding an entailment check ensures that redundant clauses are removed from the specialised program. Marriott and Stuckey (1993) demonstrates the optimisations available through constraint reordering and removal when the removal does not effect control flow. If a constraint is likely to fail and hence cause backtracking then it should be added to the constraint store as early as possible to ensure less time is wasted in unneeded calculations.

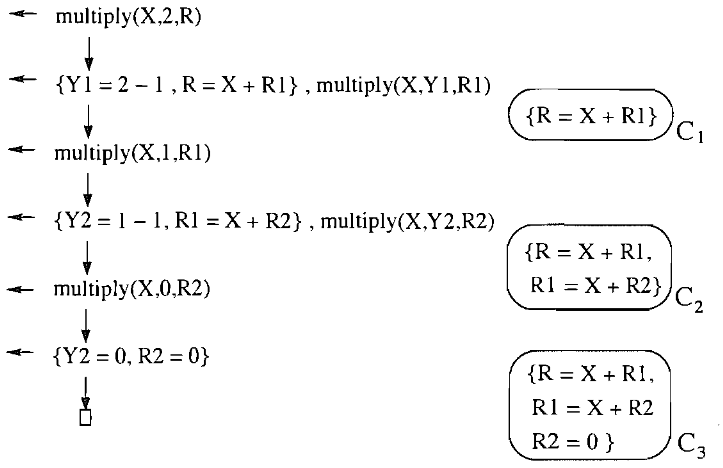

multiply(X,2,R)
{Y1 = 2 - 1 , R = X + R1} , multiply(X,Y1,R1)
multiply(X,1,R1)
{Y2 = 1 - 1, R1 = X + R2} , multiply(X,Y2,R2)
multiply(X,0,R2)
{Y2 = 0, R2 = 0}
{R = X + R1}
C1
{R = X + R1, R1 = X + R2}
C2
{R = X + R1, R1 = X + R2 R2 = 0 }
C3

FIGURE 8.2: The multiply predicate is unfolded, producing the residual constraints C_3

8.2.3 Convex Hull and Widening

If there is a subsequent call to a memoised predicate and it cannot reuse the existing residual call then a new entry is added to the memoisation table and the call is respecialised for the new constraints. It may also be possible to widen the constraint to encompass both the existing and the new call, reducing residual code size.

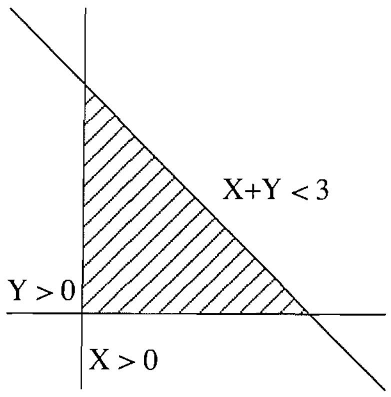
For example, consider the memoisation entry in Listing 8.11, the constraints CS_{mpq} define the convex hull in Figure 8.3(a). A subsequent call to `q(X,Y)` can only reuse the entry if its constraints lie inside the defined hull.

$CS_{mpq1} = \{X > 0, Y > 0, X + Y < 3\}$
`memo_table(q(X,Y), q__1(X,Y),` CS_{mpq1}`).`

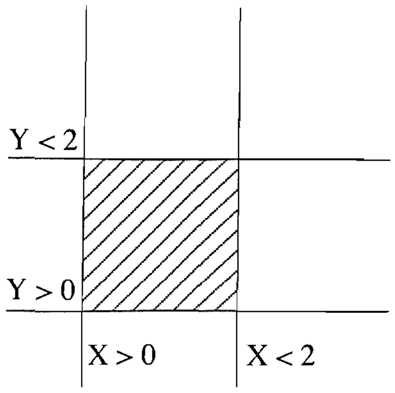
LISTING 8.11: Example memoisation entry

A call to `q(X,Y)` is made with constraints {`X>0, Y>0, X<2, Y<2`} (Figure 8.3(b)), the existing memo entry *cannot* be reused and the call to `q(X,Y)` must be respecialised. Two strategies are available:

1. Respecialise the predicate for the new constraints {`X>0, Y>0, X<2, Y<2`} resulting in the memoisation table Listing 8.12. Two residual predicates for q are created, q__1 and q__2, possibly resulting in duplicated code in the specialised program.



(a) Convex hull defined by the constraints $\{X > 0,\, Y > 0,\, X + Y < 3\}$



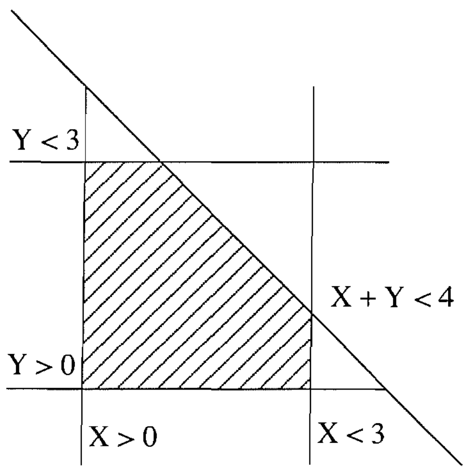
(b) Convex hull defined by the constraints $\{X > 0,\, Y > 0,\, X < 2,\, Y < 2\}$

FIGURE 8.3: Constraint sets specify a convex hull in space

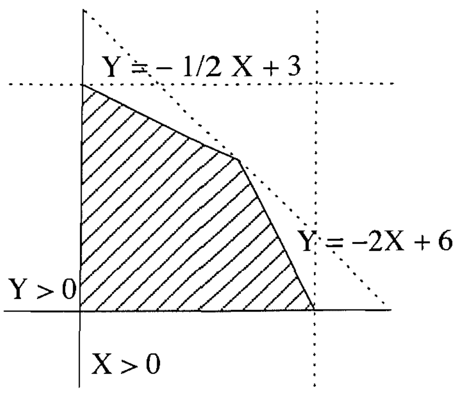
2. Creating a new set of constraints that encompass both the existing and new constraints. The new constraint set is a convex hull encompassing the constraints Figure 8.3(a) ∪ Figure 8.3(b) as shown in Figure 8.4. Once the predicate has been specialised for these constraints the old residual call can be discarded and the new one used by both calls. It may also be necessary to widen the set by removing constraints to ensure termination.

```
CS_mpq1 = {X > 0, Y > 0, X + Y < 3}
CS_mpq2 = {X > 0, Y > 0, X < 2, Y < 2}
memo_table(q(X,Y) ,q__1(X,Y), CS_mpq1).
memo_table(q(X,Y) ,q__2(X,Y), CS_mpq2).
```

LISTING 8.12: Memoisation table after respecialising for CS_{mpq2}



(a) Approximation of convex hull



(b) Optimal convex hull

FIGURE 8.4: Convex hull for constraints Figure 8.3(a) ∪ Figure 8.3(b)

Calculating the optimal convex hull (Figure 8.4(b)) is computationally expensive but it can be approximated by shifting the existing constraints (possibly to infinity) until they enclose both of the original constraint spaces (Figure 8.4(a)).

8.2.4 Rounding Errors with CLP(R)

Currently the specialisation phase uses the Rational domain, CLP(Q), to generate specialised code for the CLP(R) engine. During the specialisation phase the residual constraint store becomes part of the specialised program. Listing 8.13 demonstrates that it is not always possible to retrieve exact numbers from the CLP(R) engine and therefore truncation errors can be introduced into the specialised program. A CLP(R) program can be specialised using the CLP(Q) engine however it may take quite big rationals to accommodate the required level of precision.

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

LISTING 8.13: Demonstration of CLP(R) rounding problems, the output from the CLP engine is dependent on the ordering of the variables

8.3 Non-declarative Programs

To properly handle Prolog programs with non-declarative features one has to pay special attention to the left-propagation of bindings and of failure Prestwich (1992); Sahlin (1993). Indeed, for calls `c` to predicates with side-effects (such as `nl/0`) "`c,fail`" is not equivalent to "`fail,c`". Other predicates are called "propagation sensitive" Sahlin (1993). For calls `c` to such predicates, even though `c,fail` $\equiv$ `fail` may hold, the equivalence `(c, X=t)` $\equiv$ `(X=t, c)` does not. One such predicate is `var/1`, e.g. `(var(X),X=a)` $\not\equiv$ `(X=a,var(X))`. Predicates can both be propagation sensitive and have side-effects (such as `print/1`). The way this problem is overcome (Leuschel et al., 2004b) is via special annotations which selectively prevent the left-propagation of bindings and failure. This allows the system to handle almost full Prolog[1], while still being able to left-propagate bindings whenever this is guaranteed to be safe. In a CLP setting, the whole issue gets more complicated in that one also has to worry about the left-propagation of constraints. Take for instance the clause `p(X) :- var(X),X=<2` and suppose we transform it into `p_1(X) :- X=<2,var(X)`. The problem is now that the query `X>=2,p_1(X)` to the specialised program fails while the original query `X>=2,p(X)` succeeds with a computed answer `X=2.0`. To overcome this problem we have extended the scheme to enable us to selectively prevent the left-propagation of constraints. Using our new system we are now in a position to handle full CLP programs with non-declarative features. Take for example the following simple CLP(Q) program:

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

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

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

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

8.4 Examples and Experiments

In this section the technique is illustrated on a non-trivial example. Listing 8.14 calculates the balance of a loan over N periods. Balances is a list of length N. The interest rate is controlled by the loan scheme and decided by the amount of the initial loan. The map/3 predicate is used to apply the loan scheme over the list of balances.

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

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

map(_,[_],_).
map(SCHEME,[H1,H2|Tail], Repayment) :- Call =.. [SCHEME,H1,H2, Repayment],
            call(Call), map(SCHEME,[H2|Tail], Repayment).
calcLoan(Amount,NewTotal, Interest, Repayment) :-
          {NewTotal = Amount + (Amount * Interest) - Repayment}.
```

LISTING 8.14: Loan.pl, calculates the balance of a loan over N periods for a given loan scheme and repayment.

8.4.1 Unfolding Example

In Listing 8.15 the loan/3 predicate has been specialised to calculate the balances over two periods for a principal loan over 4000. As the length of the list is known all of the recursive calls can be executed at specialisation time. The map/3, scheme/3, and calcLoan/4 calls have been unfolded and the resultant code has been inlined into the specialised code. The two redundant loan schemes have been removed from the final code as they dealt with loans of less than 4000. The specialised predicate (Listing 8.15) runs 68% faster than the original predicate in Listing 8.14.

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

LISTING 8.15: Specialised loan predicate for `loan(X,[P1,P2], R)` where {X>4000}

8.4.2 Memoisation Example

In Listing 8.16 the map predicate from the loan program has been specialised to use either `scheme1` or `scheme2`. The length of the list has not been specified so the recursive call must be memoised. The calls in the body of `map/3` have been unfolded and the residual code inlined in the specialised code. The removal of the overhead from the `univ` (`=..`) and call operators combined with the simplification of the loan calculation to include the hard coded interest rate produces a 57% speed up over the original predicate.

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

LISTING 8.16: Specialised version of the loan example for calls *map*(*scheme*1, *T*, *R*) and *map*(*scheme*2, *T*, *R*). In this example the recursive call to *map*_1 is memoed as the length of the list is not known at specialisation time

8.5 Summary

8.5.1 Experimental Results

Table 8.1 summarises our experimental results. The timings were obtained by using SICStus Prolog 3.11 on a 2.4 Ghz Pentium 4. The second column contains the time spent by cogen to produce the generating extension. The third column contains the time that the generating extension needed to specialise the original program for a particular specialisation query. The fourth column contains the time the specialised program took for a series of runtime queries and the fifth column contains the results from the original programs. The final column contains the speedup of the specialised program as compared to the original. Full details of the experiments (source code and queries) can be found at the DPPD library (Leuschel, 1996-2004).

The multiply example is the naïve multiply from Section 8.2.2, the two loan examples are taken from the previous section. Ctl_clp is a computational tree logic (CTL) model checker written in CLP, it is based upon a CTL model checker written for XSB-Prolog from the DPPD library. It is specialised for an infinite state Petri net and a safety property.

Program	Cogen Time	Genex Time	Runtime	Original	Relative Runtime
multiply	≤10ms	20 ms	10 ms	3780 ms	0.003
loan_unfold	≤10ms	≤10ms	385 ms	647 ms	0.59
loan_map	≤10ms	≤10ms	411 ms	647 ms	0.63
ctl_clp	≤10ms	100 ms	17946 ms	24245 ms	0.74

TABLE 8.1: Experimental results

8.5.2 Summary

There has been some early work on specialisation of CLP programs Smith (1991); Smith and Hickey (1990) and optimisation Marriott and Stuckey (1993). There has been some recent interest in online specialisation techniques for constraint logic programs Fioravanti et al. (2000, 2001); Peralta (2000); Peralta and Gallagher (2002); Tao et al. (1997). To the best of our knowledge there is no work on offline specialisation of CLP programs, and to our knowledge none of the above techniques can handle non-declarative programs.

There is still scope to improve the code generator of the system, e.g., by using more sophisticated reordering as advocated in Marriott and Stuckey (1993). Other possibilities might be to convert CLP operations into standard Prolog arithmetic (e.g., using `is/2`) when this is safe.

The specialisation of CLP programs is an important research area for partial evaluation. This chapter presented a working version of an offline CLP specialiser and first results look promising. Supporting CLP along with the other feature already developed in this thesis allows the developed specialiser to handle a large class of full Prolog programs.

Chapter 9

Specialising Interpreters

This chapter presents the results of specialisation using the system on a series of examples. The focus is on the specialisation of interpreters and in addition to presenting experimental results it is shown that interesting transformations can also be achieved. In particular, the Lloyd-Topor transformation (Lloyd and Topor, 1984) is performed by partially evaluating a modified vanilla interpreter. The interpreter used to calculate the binary clause semantics used for the binding-time analysis in Chapter 5 is also presented.

An interesting application for partial evaluation is the specialised of interpreters. The object program to interpret is typically **static** and known at specialisation time, while the runtime goal remains **dynamic**. Partial evaluation can remove the overhead of interpretation, performing all the interpretation tasks at specialisation time and leaving behind a much more efficient "compiled" program. The ultimate goal is to remove a full layer of interpretation and achieve the *Jones Optimality* criterion (Jones et al., 1993). The Jones Optimality criterion is discussed in Chapter 1.

9.1 Vanilla Self-interpreter

A classic benchmark for partial evaluation of logic programs is the vanilla self-interpreter (Listing 9.1) . It is a self-interpreter as it is written in the same language it interprets. While it may look like a simple program it still presents enough challenges for partial evaluation and will require careful annotation. Once the partial evaluator can successfully specialise the simple vanilla interpreter it is possible to extend the functionality and present more powerful transformations.

```
:- use_module(prolog_reader).

solve([]).
solve([A|T]) :- solve_atom(A), solve(T).

solve_atom(A) :-
```

```
    prolog_reader:get_clause(A,B),
    solve(B).

solve_file(File, Goal) :-
    prolog_reader:load_file(File),
    solve_atom(Goal).
```

LISTING 9.1: Vanilla self-interpreter for Prolog

The predicate `solve_atom/1` looks up the clause definition in the clause database and calls `solve/1` its body. Clauses are represented by a `head` and a list of body literals. An auxiliary module (`prolog_reader`) is used to load clauses from a file, it defines `load_file/1` and `get_clause/2`. Listing 9.1 has the entry point `solve_file/2` which takes as arguments the name of the file containing the clauses and a entry goal.

The program in Listing 9.2 is used to demonstrate the interpreters in this chapter. It contains predicates for reversing and appending lists If the partial evaluator is powerful enough it should be able to achieve Jones optimality, specialising the vanilla interpreter for Listing 9.2 and produce code which is as at least as efficient as the original program.

```
rev_app(A,B,C) :-
        rev(A,D), append(D,B,C).

rev(A,B) :-
        rev(A,[],B).

append([],B,B).
append([A|As],Bs, [A|Cs]) :-
        append(As, Bs, Cs).

rev([],A,A).
rev([A|B], C, D) :-
        rev(B,[A|C],D).
```

LISTING 9.2: Example file containing clauses for append and reversing lists

Both the object program and the entry goal pattern will be known at specialisation time. Listing 9.3 is the annotated version of the interpreter using the annotation format introduced in Chapter 2. At specialisation time the object program is given along with the entry pattern (not a fully instantiated goal). The goal given to `solve_atom/1` is marked as **nonvar**, the top level functor will be kept but the arguments will be replaced with fresh variables. To ensure termination at specialisation time for all specialisation goals *either* the recursive call to `solve/1` or the call to `solve_atom/1` must be marked as **memo**. In these annotations `solve_atom/1` is marked as **memo**, as this will provide more natural specialised programs. The remaining calls are marked as **unfold** if they are user defined or call otherwise. The filename is **static** and the actual file should be available during specialisation (the clauses will be loaded at specialisation time).

```
:-module(vanilla_list ,[]).
:-use_module(prolog_reader).
logen(solve, solve([])).
logen(solve, solve([A|B])) :-
```

```
        logen(memo, solve_atom(A)),
        logen(unfold, solve(B)).
logen(solve_atom, solve_atom(A)) :-
        prolog_reader:logen(call, get_clause(A,B)),
        logen(unfold, solve(B)).
logen(solve_file, solve_file(A,B)) :-
        prolog_reader:logen(call, load_file(A)),
        logen(memo, solve_atom(B)).
:- filter
        solve_atom(nonvar).
:- filter
        solve_file(static, nonvar).
```

LISTING 9.3: Annotated version of the vanilla interpreter

The vanilla interpreter is specialised for the entry goal rev_app(A,B,C) and the object program from Listing 9.2. The residual program contains all of the code reachable from rev_app(A,B,C). Notice that the interpreter has been specialised away and only the translated object clauses appear in the residual code.

```
:- module('vanilla_list_inc.spec',[]).
solve_file('append.pl', rev_app(A,B,C)) :-
        solve_file__0(A, B, C).
solve_atom__1(A, B, C) :-
        solve_atom__2(A, D),
        solve_atom__3(D, B, C).
solve_atom__2(A, B) :-
        solve_atom__4(A, [], B).
solve_atom__3([], A, A).
solve_atom__3([A|B], C, [A|D]) :-
        solve_atom__3(B, C, D).
solve_atom__4([], A, A).
solve_atom__4([A|B], C, D) :-
        solve_atom__4(B, [A|C], D).
```

LISTING 9.4: Specialising the vanilla interpreter for rev_app/3. The output is Jones Optimal, all interpretation overhead has been removed

Listing 9.5 is the memoisation table produced while specialising the vanilla interpreter. The table/3 predicate contains the mappings between the original and the specialised predicates. For example solve_atom__1(A,B,C) represents rev_app(A,B,C) from the original program.

```
table(solve_atom(rev_app(A,B,C)), solve_atom__1(A,B,C), []).
table(solve_atom(rev(A,B)),       solve_atom__2(A,B),   []).
table(solve_atom(append(A,B,C)),  solve_atom__3(A,B,C), []).
table(solve_atom(rev(A,B,C)),     solve_atom__4(A,B,C), []).
```

LISTING 9.5: Memo table for Listing 9.4

Renaming Listing 9.4 using these mappings produces the residual program in Listing 9.6. This program is identical to the original example file, Jones optimality for the vanilla interpreter has been demonstrated using these annotations, as the specialised program is identical to the original source program after trivial renaming.

```
rev_app(A, B, C) :-                 %solve_atom__1
        rev(A, D),                  %solve_atom__2
        append(D, B, C).            %solve_atom__3
rev(A, B) :-                        %solve_atom__2
        rev(A, [], B).              %solve_atom__4
append([], A, A).                   %solve_atom__3
append([A|B], C, [A|D]) :-          %solve_atom__3
        append(B, C, D).            %solve_atom__3
rev([], A, A).                      %solve_atom__4
rev([A|B], C, D) :-                 %solve_atom__4
        rev(B, [A|C], D).           %solve_atom__4
```

LISTING 9.6: Renaming Listing 9.4 using the mappings from Listing 9.5 reproduces the original source program Listing 9.2. Jones optimality has been achieved.

To benchmark this program the prolog_reader module has been removed and the clauses from the example file are inlined. This removes the overhead of reloading the example file for every iteration (as the specialised program does not suffer this overhead).

Benchmarks for the specialised version of the vanilla interpreter can been seen in Table 9.1. The specialised program is five times faster than the original interpreted version as the overhead of interpretation has been removed.

Benchmark	Iterations	Specialisation Time	Original Runtime	Specialised Runtime	Relative Runtime
Vanilla	3000000	20ms	26280ms	4940ms	0.19

TABLE 9.1: Benchmark figures for the vanilla interpreter

Table 9.2 compares the compiled program size of the original and specialised program. The specialised code is 30% smaller than the original interpreter and clauses. The original size included the interpreter along with the program clauses and is therefore should always be larger than the specialised code (given that Jones Optimality has been shown).

Benchmark	Original Program Size	Specialised Program Size	Relative Program Size
Vanilla	2082 bytes	1462 bytes	0.70

TABLE 9.2: Program size comparison for vanilla interpreter

In this example Jones optimality was demonstrated for the vanilla interpreter. Online partial evaluators such as ECCE (Leuschel et al., 1998) or MIXTUS (Sahlin, 1993) 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 Martens (1994) (discussing the parsing problem) and the more recent Vanhoof and Martens (1997) and Leuschel (2002) for more details. The offline approach provides precise control over the specialisation process in a predictable manner. Predictability is important, the specialisation of the interpreter should behave independently of the complexity of the object program. Online techniques may work well for many object programs but can be "fooled" by other (often contrived) object programs. It should be noted that online techniques can be capable of removing several layers of interpretation in one go, while an offline approach will typically only be able to remove one layer at a time.

Just having a simple offline partial evaluator is not sufficient to remove all interpretation, the specialiser must also provide expressive annotations. It would not have been possible to achieve this criterion without the **nonvar** annotation. The argument to `solve_atom/1` cannot be marked as **static** (as it can contain variables), and marking it as **dynamic** would mean that no useful specialisation could be achieved. Without the **nonvar** annotation considerable rewriting of the interpreter would have been required.

The interpreter in Listing 9.1 is written is such a way that the specialiser can distinguish between conjunctions and object level calls. The body of the clauses are represented as a list of calls, this allows the specialiser to treat actual object calls differently to program structure (the enclosing list skeleton). The examples in this chapter will extend the interpreter to handle the natural representation of Prolog programs.

9.2 A Debugging Vanilla Interpreter

Jones optimality has been demonstrated for the vanilla interpreter, the interpreter is now extended for debugging purposes. The benefit of having a specialiser capable of removing the interpretation overhead is that it allows the programmer to extend the interpreter in an easy fashion with a minimal overhead.

During debugging it is often useful to look at program traces, that is a step by step guide of the execution. In Prolog this is often represented by tracking *Calls*, *Exits* and *Failures* of the predicates of interest. Modern Prolog implementations generally come with built-in debugging and trace facilities but here the vanilla interpreter is used to show the extension. This could also be added to modified vanilla interpreters or interpreters for other custom languages.

The vanilla interpreter, Listing 9.1, is extended with an extra argument containing a list of predicates to trace. Each time a call to `solve/3` is made the call is first checked against the list of predicates to be traced; if found, the debugging information is printed on calling, exiting and failure. The indentation level increases and decreases as the program descends into and exits from calls, the current level is stored in the dynamic predicate `indent/1`.

```
:- use_module(prolog_reader).
:- use_module(library(lists)).
:- dynamic indent/1.

current_indent_level(X) :- indent(X),!.
current_indent_level(0) :- assert(indent(0)).
```

```
increase_indent_level :- indent(X), retractall(indent(_)),assert(indent(s(X))).
decrease_indent_level :- indent(s(X)), retractall(indent(_)),assert(indent(X)).

solve([],_).
solve([A|T],Trace) :-
    (trace_call(A,Trace) ->
             debug_print('Call:',A),
             (
             (
              increase_indent_level,
              solve_atom(A, Trace),
              decrease_indent_level, debug_print('Exit:',A)
             )
             ;
             (decrease_indent_level, debug_print('Fail:',A))
             )
     ;
             solve_atom(A,Trace)

     ), solve(T,Trace).

solve_atom(A, Trace) :-
     prolog_reader:get_clause(A,B),
     solve(B, Trace).

solve_file(File, Goal, Trace) :-
    reset_indent_level,
    prolog_reader:load_file(File),
    solve_atom(Goal,Trace).

debug_print(Type, Call) :-
    current_indent_level(Indent),
    print_indent(Indent),
    print(user_error,Type),
    print(user_error,Call),
    nl(user_error).

trace_call(Call, CallsToTrace) :-
    functor(Call, P, A),
    member(P/A, CallsToTrace).

print_indent(0).
print_indent(s(A)) :- print(user_error,'>'),print_indent(A).
```

LISTING 9.7: Extended version of the vanilla interpreter. Prints debugging information for watched predicates.

The debugging interpreter is first specialised for the goal `rev_app(A,B,C)`, and provided with an empty list of goals to trace. The residual code, Listing 9.8, has been specialised with tracing disabled. The resultant code contains no debugging statements. This produces identical code to the specialised vanilla code (Listing 9.4), the only difference is the inclusion of the `:- dynamic indent/1`. In this example all of the overhead of debugging, for example the checking of each call against the list of predicates to trace, has been removed and no clauses related to debugging have been generated.

```
:- module('vanilla_debug_inc.spec',[]).
:- dynamic indent/1.
solve_file('append.pl', rev_app(A,B,C), []) :-
        solve_atom__1(A, B, C).          % rev_app/3
solve_atom__1(A, B, C) :-                % rev_app/3
        solve_atom__2(A, D),             % rev/2
        solve_atom__3(D, B, C).          % append/3
solve_atom__2(A, B) :-                   % rev/2
        solve_atom__4(A, [], B).         % rev/3
solve_atom__3([], A, A).                 % append/3
solve_atom__3([A|B], C, [A|D]) :-        % append/3
        solve_atom__3(B, C, D).          % append/3
solve_atom__4([], A, A).                 % rev/3
solve_atom__4([A|B], C, D) :-            % rev/3
        solve_atom__4(B, [A|C], D).      % rev/3
```

LISTING 9.8: Specialising the vanilla debugging interpreter for rev_app(A,B,C), no predicates have been specified for tracing. No debugging information is produced.

A more interesting specialisation query specialises the program with tracing enabled. In Listing 9.9 the interpreter is specialised for the goal rev_app(A,B,C) and the predicate [append/3] is marked for tracing. The residual code contains identical fragments to the original code but tracing print statements have been weaved into the calls to append/3. Each time a call to append/3 is made it is surrounded by print statements and a guard condition to catch failure. All predicates without tracing have been left unmodified, except renaming, introducing no additional overhead for the rest of the program.

```
:- module('vanilla_debug_inc.spec',[]).
:- dynamic(indent / 1).
solve_atom(rev_app(A,B,C), [append/3]) :-
        solve_atom__0(A, B, C).                % rev_app/3

solve_atom__0(A, B, C) :-                      % rev_app/3
        solve_atom__1(A, D),                   % rev/2
        debug_print__2('Call:', append(D,B,C)),
        (   increase_indent_level__3,
            solve_atom__4(D, B, C),            % app/3
            decrease_indent_level__5,
            debug_print__2('Exit:', append(D,B,C))
        ;   decrease_indent_level__5, debug_print__2('Fail:', append(D,B,C))
        ).
solve_atom__1(A, B) :-                         % rev/2
        solve_atom__8(A, [], B).               % rev/3
solve_atom__4([], A, A).                       % append/3
solve_atom__4([A|B], C, [A|D]) :-              % append/3
        debug_print__2('Call:', append(B,C,D)),  % DEBUG Entry
        (   increase_indent_level__3,
            solve_atom__4(B, C, D),            % append/3
            decrease_indent_level__5,
            debug_print__2('Exit:', append(B,C,D))% DEBUG Exit
        ;   decrease_indent_level__5,
            debug_print__2('Fail:', append(B,C,D))% DEBUG Fail
        ).
solve_atom__8([], A, A).                       % rev/3
solve_atom__8([A|B], C, D) :-                  % rev/3
        solve_atom__8(B, [A|C], D).            % rev/3
```

```
decrease_indent_level__5 :-
        indent(s(A)), retractall(indent(_)), assert(indent(A)).
increase_indent_level__3 :-
        indent(A), retractall(indent(_)), assert(indent(s(A))).
debug_print__2(A, B) :-
        current_indent_level__6(C), print_indent__7(C),
        format(user_error, "~w ~w~n", [A,B]).
current_indent_level__6(A) :- indent(A), !.
current_indent_level__6(0) :- assert(indent(0)).
print_indent__7(0).
print_indent__7(s(A)) :-
        print(user_error, >), print_indent__7(A).
```

LISTING 9.9: Specialising the vanilla debugging interpreter for rev_app(A,B,C), append/3 has been marked for tracing.

The output from running the specialised code (Listing 9.9) can be seen in Listing 9.10. Every call to append/3 has been traced on calling and exiting. The indentation levels line up the call and exit patterns of each individual call. Only append/3 has been traced, the other predicates are unmodified.

```
Call: append([h,g,f,e,d,c,b,a],[i,j,k,l,m,n,o,p],_11928)
>Call: append([g,f,e,d,c,b,a],[i,j,k,l,m,n,o,p],_12128)
>>Call: append([f,e,d,c,b,a],[i,j,k,l,m,n,o,p],_12371)
>>>Call: append([e,d,c,b,a],[i,j,k,l,m,n,o,p],_12678)
>>>>Call: append([d,c,b,a],[i,j,k,l,m,n,o,p],_13049)
>>>>>Call: append([c,b,a],[i,j,k,l,m,n,o,p],_13484)
>>>>>>Call: append([b,a],[i,j,k,l,m,n,o,p],_13983)
>>>>>>>Call: append([a],[i,j,k,l,m,n,o,p],_14546)
>>>>>>>>Call: append([],[i,j,k,l,m,n,o,p],_15173)
>>>>>>>>Exit: append([],[i,j,k,l,m,n,o,p],[i,j,k,l,m,n,o,p])
>>>>>>>Exit: append([a],[i,j,k,l,m,n,o,p],[a,i,j,k,l,m,n,o,p])
>>>>>>Exit: append([b,a],[i,j,k,l,m,n,o,p],[b,a,i,j,k,l,m,n,o,p])
>>>>>Exit: append([c,b,a],[i,j,k,l,m,n,o,p],[c,b,a,i,j,k,l,m,n,o,p])
>>>>Exit: append([d,c,b,a],[i,j,k,l,m,n,o,p],[d,c,b,a,i,j,k,l,m,n,o,p])
>>>Exit: append([e,d,c,b,a],[i,j,k,l,m,n,o,p],[e,d,c,b,a,i,j,k,l,m,n,o,p])
>>Exit: append([f,e,d,c,b,a],[i,j,k,l,m,n,o,p],[f,e,d,c,b,a,i,j,k,l,m,n,o,p])
>Exit: append([g,f,e,d,c,b,a],[i,j,k,l,m,n,o,p],[g,f,e,d,c,b,a,i,j,k,l,m,n,o,p])
Exit: append([h,g,f,e,d,c,b,a],[i,j,k,l,m,n,o,p],[h,g,f,e,d,c,b,a,i,j,k,l,m,n,o,p])
```

LISTING 9.10: Output from the specialised code Listing 9.9.

Table 9.3 contains the benchmark results for running the debugging interpreter. With tracing disabled the specialised code runs without overhead, while the interpreted code is much slower. When tracing is enabled the actual printing and indenting of the debugging statements take up a substantial amount of the computation time so there is little speed up. However, only predicates that are marked for tracing will suffer a performance penalty, so the speed increase is dependent on how much of the execution is taken up with traced calls.

Table 9.4 compares the compiled program sizes of the specialised programs and the interpreter (with clauses inlined). The specialised program without tracing contains none of the interpreters code while the tracing program contains extra calls surrounding all traced calls.

Benchmark	Iterations	Specialisation Time	Original Runtime	Specialised Runtime	Relative Runtime
Debug (No debug)	3000000	30ms	42180ms	5090ms	0.12
Debug (trace rev/2)	30000	30ms	13470ms	12350ms	0.92

TABLE 9.3: Benchmark figures for the vanilla debugging interpreter

Benchmark	Original Program Size	Specialised Program Size	Relative Program Size
Debug (No debug)	6237 bytes	1625 bytes	0.26
Debug (trace rev/2)	6356 bytes	5032 bytes	0.79

TABLE 9.4: Program size comparison for vanilla interpreter

9.3 A Profiling Vanilla Interpreter

The previous section produced specialised tracing code. The vanilla interpreter is now extended to profile code. The profiling information will track the number of calls to a clause, this example produces profiling information for each program point. An example of the profiled output is given in Listing 9.11, here append/3 and rev/2 have been profiled. Each program point of interest is commented with the number of *hits* to that program point. In the example append([],A,A) succeeded once, and the recursive call to append/3 succeeded eight times.

```
append([],A,A).          /* 1 */
append([A|B],C,[A|D]) :- /* 8 */
     append(B,C,D).      /* 8 */
rev(A,B) :-              /* 1 */
     rev(A,[],B).        /* 1 */
```

LISTING 9.11: Output from the profiling vanilla interpreter on the example, Listing 9.2, for the goal rev_app([a,b,c,d,e,f,g,h],[i,j,k,l,m,n,o,p] ,C) profiling all calls to rev/2 and append/3.

The original vanilla interpreter, Listing 9.1, is modified to include program point information. Each call to get_clause/3 returns a unique clause identifier and as solve/3 progresses it counts the number of body literals. This produces a unique identifier in the form id(ClauseNumber, LiteralNumber) for each point in the program. The debugging interpreter passes around a list of predicates of interest, i.e. the predicates will be profiled. The special case all will profile all predicates in the program. Each time solve/3 is called the current goal, which is passed as an additional argument, is checked against the list of predicates to profile. If the predicate is being profiled then a *hit* is counted against the current program point, note that an additional *hit* is made at the end of the clause to tell if the clause succeeded.

After an execution the dynamic predicate profile_data/2 contains the number of *hits* for each program point of interest. To present this to the user in a useful fashion the clauses of interest are pretty printed with the profile information contained in comments. The entry point solve_and_print/2 executes the goal and prints the profile information.

```
:- use_module(library(lists)).   :- use_module(library(charsio)).
:- dynamic profile_data/2.

/* Extended Vanilla solve */
solve([],ID,ToProfile, CurrentGoal) :-
    (profile_call(CurrentGoal,ID, ToProfile) ->
             hit_pp(ID)
     ;
             true
     ).

solve([A|T],ID,ToProfile, CurrentGoal) :-
    (profile_call(CurrentGoal,ID,ToProfile) ->
            hit_pp(ID),
            solve_atom(A,ToProfile)
    ;
            solve_atom(A,ToProfile)
    ), ID = id(CID,LID), NID is LID+1, NEWID = id(CID, NID),
    solve(T,NEWID,ToProfile, CurrentGoal).

/* Extended Vanilla solve_atom, keeps current goal */
solve_atom(A,ToProfile) :-
     get_clause(ID,A,B), functor(A, Func, Arity),
     solve(B,id(ID,0),ToProfile, Func/Arity).

/* Entry point: First calls goal then prints results of profiler */
solve_and_print(Goal, ToProfile) :-
    solve_atom(Goal, ToProfile),
    ( ToProfile = [] -> true
    ;
      print_profile_data(user_error, ToProfile)
    ).

/* Track Program Point hits */
hit_pp(ID) :-
    (profile_data(ID, Old) ->
            New is Old +1,
            retractall(profile_data(ID,_))
    ;
            New = 1
    ),assert(profile_data(ID, New)).

get_profile_data(ID, Hits) :- profile_data(ID, Hits).

/* Is this call marked for profiling? */
profile_call(_Call, _ID,ToProfile) :- member(all, ToProfile), !.
profile_call(Call, _, ToProfile)    :- member(Call, ToProfile).

/* Pretty Printing Clauses */
print_profile_data(Stream,ToProfile) :-
    get_clause(CID,Head,Body), numbervars((Head,Body), 0,_),
    functor(Head, Func,Arity), profile_call(Func/Arity, _,ToProfile),
    print_clause(Stream, Head, Body, CID), fail.
print_profile_data(_,_).

print_clause(Stream, Head, [], CID) :-
```

```
    !,
    write_to_chars(Head,CallS), get_profile_data(id(CID,0), Hits),
    format(Stream,"~s./* ~w */ ~n",[CallS, Hits]).
print_clause(Stream, Head, Body, ID) :-
    write_to_chars(Head,CallS), get_profile_data(id(ID,0), Hits),
    format(Stream,"~s :-/* ~w */",[CallS, Hits]),
    print_profile_body(Stream,ID,0,Body).

print_profile_body(_,_,_,[]).
print_profile_body(Stream,ID,LID,[Call|Rest]) :-
    write_to_chars(Call, CallS),
    NLID is LID + 1,  get_profile_data(id(ID,NLID), Hits),
    (Rest = [] ->
            format(Stream, "~n      ~s. /* ~w */~n", [CallS, Hits])
    ;
            format(Stream, "~n      ~s, /* ~w */", [CallS, Hits]),
            print_profile_body(Stream,ID,NLID, Rest)
    ).
```

LISTING 9.12: Extended version of the vanilla interpreter. Collects profiling information and pretty prints results.

As in the case of the debugging interpreter the program is first specialised with profiling disabled. The resulting code is shown in Listing 9.13, note that no profiling information has been added to the program and the results are again the same as the original program. No overhead has been introduced on the non-profiled code.

```
:- module('vanilla_profile_inc.spec',[]).
:- dynamic(profile_data / 2).
solve_and_print(rev_app(A,B,C), []) :-
        solve_and_print__0(A, B, C).
solve_and_print__0(A, B, C) :-
        solve_atom__1(A, B, C).          % rev_app/3
solve_atom__1(A, B, C) :-                % rev_app/3
        solve_atom__2(A, D),             % rev/2
        solve_atom__3(D, B, C).          % append/3
solve_atom__2(A, B) :-                   % rev/2
        solve_atom__4(A, [], B).         % rev/3
solve_atom__3([], A, A).                 % append/3
solve_atom__3([A|B], C, [A|D]) :-        % append/3
        solve_atom__3(B, C, D).          % append/3
solve_atom__4([], A, A).                 % rev/3
solve_atom__4([A|B], C, D) :-            % rev/3
        solve_atom__4(B, [A|C], D).      % rev/3
```

LISTING 9.13: The vanilla profiling interpreter is specialised for solve_and_print/2 for the goal rev_app(A,B],C) without profiling. No profiling overhead is introduced.

Listing 9.12 is next specialised for the same goal rev_app(A,B,C) but profiling the predicates rev/2 and append/3. The specialised code is shown in Listing 9.14, when executed this code produces the output already seen in Listing 9.11. Only the predicates of interest have been modified, non-profiled predicates remain unchanged. The specialised code also contains a specialised pretty printer for producing the results with minimal overhead.

```
:- module('vanilla_profile_inc.spec',[]).
:- dynamic(profile_data / 2).
solve_and_print(rev_app(A,B,C), [rev/2,append/3]) :-
        solve_and_print__0(A, B, C).
solve_and_print__0(A, B, C) :-
        solve_atom__1(A, B, C),
        print_profile_data__2.
solve_atom__1(A, B, C) :-                 % rev_app/3
        solve_atom__3(A, D),              % rev/2
        solve_atom__4(D, B, C).           % append/3
solve_atom__3(A, B) :-                    % rev/2
        hit_pp__5(id(2,0)),
        solve_atom__6(A, [], B),          % rev/3
        hit_pp__5(id(2,1)).
solve_atom__4([], A, A) :-                % append/3
        hit_pp__5(id(0,0)).
solve_atom__4([A|B], C, [A|D]) :-         % append/3
        hit_pp__5(id(1,0)),
        solve_atom__4(B, C, D),           % append/3
        hit_pp__5(id(1,1)).
solve_atom__6([], A, A).                  % rev/3
solve_atom__6([A|B], C, D) :-             % rev/3
        solve_atom__6(B, [A|C], D).       % rev/3

/* Specialised Pretty Printer */
print_profile_data__2 :-
        profile_data(id(0,0), A),
        format(user_error, "~s./* ~w */ ~n", ["append([],A,A)",A]),
        fail.
print_profile_data__2 :-
        profile_data(id(1,0), A),
        format(user_error, "~s :-/* ~w */", ["append([A|B],C,[A|D])",A]),
        profile_data(id(1,1), B),
        format(user_error, "~n      ~s. /* ~w */~n", ["append(B,C,D)",B]),
        fail.
print_profile_data__2 :-
        profile_data(id(2,0), A),
        format(user_error, "~s :-/* ~w */", ["rev(A,B)",A]),
        profile_data(id(2,1), B),
        format(user_error, "~n      ~s. /* ~w */~n", ["rev(A,[],B)",B]),
        fail.
print_profile_data__2.

hit_pp__5(A) :-
        (   profile_data(A, B) ->
            C is B+1, retractall(profile_data(A,_))
        ;   C=1
        ),
        assert(profile_data(A,C)).
```

LISTING 9.14: The vanilla profiling interpreter is specialised for solve_and_print/2 and the goal rev_app(A,B,C) with profiling on rev/2 and append/3.

Table 9.5 contains the benchmark results for the specialised and original profiling interpreters. With profiling removed the specialised program runs without an overhead compared to the original program, while the non-specialised interpreted version is dramatically slower. Even with profiling enabled the specialised program executes in 66%

of the time of the original interpreted version, in this example a large proportion of the program is being profiled (append/3 takes up almost half of the execution time). With larger programs the speed increase will be more dramatic depending on how much of the code is profiled, as non-profiled code runs without overhead.

Benchmark	Iterations	Specialisation Time	Original Runtime	Specialised Runtime	Relative Runtime
Profile (no Profile)	3000000	30ms	91280ms	5280ms	0.06
Profile (rev/2,append/3)	30000	30ms	31630ms	20920ms	0.66

TABLE 9.5: Benchmark figures for the vanilla profiling interpreter

Table 9.6 compares code size of the original and specialised programs. Without profiling all of the interpreter can be removed and smaller code is produced, but as more profiled predicates are added the size of the code increases. Each profiled predicate will produce a specialised pretty printer along with the additional *hit* counters. The annotations could be modified to produce a single pretty printer instead of the specialised printers shown in Listing 9.14.

Benchmark	Original Program Size	Specialised Program Size	Relative Program Size
Profile (no Profile)	9594 bytes	1799 bytes	0.19
Profile (rev/2,append/3)	9594 bytes	7222 bytes	0.75

TABLE 9.6: Program size comparison for the vanilla Profiling interpreters

9.4 A Caching Vanilla Interpreter

The debugging and profiling interpreters both provided extra information to the programmer about the execution of the program. The interpreter is now extended to use cached values, and as a result produce caching specialised object programs. The following examples use a naïve implementation of fib/2 to calculate the Fibonacci sequence (Listing 9.15). It is well known that this implementation of Fibonacci is exponential but by caching previously values it can be made linear.

```
fib(0,0).
fib(1,1).
fib(X,Y) :-
        X1 is X-1,
        X2 is X-2,
        fib(X1, Y1),
        fib(X2, Y2),
        Y is Y1+Y2.
```

LISTING 9.15: fib/2 naïvely calculates the Fibonacci sequence

The existing vanilla interpreter only handles user predicates, the first simple extension is to support built-ins. The predicate `is_builtin/1` is added which specifies which predicates are built-ins, i.e. not implemented by the user, these calls can then made directly.

The global dynamic predicate `cache/2` is used to hold the cached values. A new predicate `solve_atom_cache/2` checks if the current call has caching enabled. If caching is enabled the answer is first looked up in the cache, if no matching solution is found the original `solve_atom/2` is called and the result is added to the cache. The cache is reset before execution to ensure the benchmark comparisons are fair. The modified interpreter is shown in Listing 9.16.

```
:- use_module(library(lists)).
:- dynamic cache/2.

solve([], _).
solve([A|T], ToCache) :- is_builtin(A),!, call(A), solve(T, ToCache).
solve([A|T], ToCache) :- solve_atom_cache(A, ToCache), solve(T, ToCache).

solve_atom(A, ToCache) :-
    get_clause(A,B),
    solve(B,ToCache).

solve_atom_cache(A, ToCache) :-
     functor(A, F, Arity),
     (cache_pred(F/Arity, ToCache) ->
           (cache(F/Arity, A) ->
                 true
           ;
                  solve_atom(A,ToCache),
                  assert(cache(F/Arity, A))

           )
     ;
           solve_atom(A,ToCache)
     ).

solve_file(File, Goal, ToCache) :-
    prolog_reader:load_file(File),
    retractall(cache(_,_)),
    solve_atom_cache(Goal, ToCache).

cache_pred(Pred, ToCache) :-
    member(Pred, ToCache).

is_builtin(is(_,_)).
```

LISTING 9.16: Extended version of vanilla interpreter to implement answer caching.

As in the previous examples the interpreter is first specialised with the additional features disabled. In this case we specialise for calling `fib/2` with no clauses marked for caching. The resulting specialised program, Listing 9.17, is identical (after renaming) to our original definition of `fib/2` from Listing 9.15. Non-cached code has no additional overhead when used with the caching interpreter.

```
solve_atom_cache__1(0, 0).                % fib/2
solve_atom_cache__1(1, 1).                % fib/2
solve_atom_cache__1(A, B) :-              % fib/2
        C is A-1,
        D is A-2,
        solve_atom_cache__1(C, E),        % fib/2
        solve_atom_cache__1(D, F),        % fib/2
        B is E+F.
```

LISTING 9.17: The caching interpreter is specialised with caching disabled. No overhead is introduced.

The definition of fib/2 without caching is not only slow it is also very memory intensive, running the program on queries greater that fib(25,X) produces out of memory errors. This can be improved by introducing a cache of previously calculated values. The vanilla caching interpreter is specialised again with caching enabled on all calls to fib/2. In the resulting code, Listing 9.18, a new predicate has been introduced, wrapping the original definition of fib/2. If the goal has been seen before then the cached value is returned, otherwise the original definition is called and the answer is stored. Note that all calls to the original fib/2 in the program will be automatically translated to call the cached version (as in the case of the two recursive calls).

```
solve_file__0(A, B) :-
        retractall(cache(_,_)),
        solve_atom_cache__1(A, B).

solve_atom_cache__1(A, B) :-            % fib_cache/2
        (   cache(fib/2, fib(A,B)) ->
            true
        ;   solve_atom__2(A, B),        % fib/2
            assert(cache(fib/2,fib(A,B)))
        ).

solve_atom__2(0, 0).                    % fib/2
solve_atom__2(1, 1).                    % fib/2
solve_atom__2(A, B) :-                  % fib/2
        C is A-1,
        D is A-2,
        solve_atom_cache__1(C, E),      % fib_cache/2
        solve_atom_cache__1(D, F),      % fib_cache/2
        B is E+F.
```

LISTING 9.18: The caching interpreter is specialised with caching enabled for fib/2. The code has inlined cache checking and generation.

Table 9.7 shows the benchmark timings for running the original and specialised programs. With caching disabled only a few number of iterations were possible due to the high execution times of the naïve implementation. The cached version performs far better than the uncached version (note the number of iterations in addition to the execution times). The specialised version runs nearly twice as fast as the interpreted version.

Table 9.8 compares the size of the specialised program against the original interpreter and clauses. Each cached predicate will contain an extra caching entry point (like

Benchmark	Iterations	Specialisation Time	Original Runtime	Specialised Runtime	Relative Runtime
Cache (no Cache)	100	20ms	76950ms	4780ms	0.06
Cache (fib/2)	100000	20ms	42880ms	23320ms	0.54

TABLE 9.7: Benchmark figures for the vanilla caching interpreter

`solve_atom_cache__1/2` in Listing 9.18). All other predicates in the program that are not being cached will remain unchanged (apart from renaming).

Benchmark	Original Program Size	Specialised Program Size	Relative Program Size
Cache (no Cache)	5810 bytes	1485 bytes	0.26
Cache (fib/2)	5810 bytes	2512 bytes	0.43

TABLE 9.8: Program size comparison for the vanilla caching interpreter

9.5 Binary Clause Semantics

The previous examples produced specialised code that was runnable, either augmented with extra output or extended for caching. The focus now switches to an example of code transformation for analysis. Chapter 5 introduced an algorithm for an automatic binding-time analysis, a part of this algorithm involved converting the program into binary clause semantics. This was then used to prove properties about the original program.

The binary clause semantics is a representation of the loops in a program, it is used in the binding-time analysis algorithm to reason about termination. Informally, the binary clause semantics of a program P is the set of all pairs of atoms (called binary clauses) $p(\bar{X})\theta \leftarrow q(\bar{t})$ such that p is a predicate, $p(\bar{X})$ is a most general atom for p, and there is a finite derivation (with leftmost selection rule) $\leftarrow p(\bar{X}), \ldots, \leftarrow (q(\bar{t}), Q)$ with computed answer substitution θ. In other words a call to $p(\bar{X})$ is followed some time later by a call to $q(\bar{t})$, computing a substitution θ.

For example take the program fragment in Listing 9.19 for the predicates `p/1` and `r/2`.

```
p(0).
p(X) :- r(X,Y),p(Y).

r(s(X), X).
```

LISTING 9.19: Example program for binary clause transformation.

The analysis is only interested in computing the possible loops in the program, for a loop to exist there must be a sequence of calls from one program point back to the same program point. In this case there is one possible loop, the recursive call to `p/1`.

This is represented in the binary clause semantics as $p(X)\theta_0 \leftarrow p(Y)$ where $\theta_0 := \{X/s(Y)\}$. A call from `p(X)` is followed sometime later by another call to `p(Y)` with the substitution $\{X/s(Y)\}$. In fact there are an infinite number of binary clauses, $p(X)\theta_1 \leftarrow p(Y), p(X)\theta_2 \leftarrow p(Y), \ldots, p(X)\theta_n \leftarrow p(Y), \ldots$, with the substitutions $\theta_1 := \{X/s(s(Y))\}, \theta_2 := \{X/s(s(s(Y)))\}, \ldots, \theta_n := \{X/s(s(s(\ldots s(Y))))\}, \ldots$

The binding-time analysis algorithm works on an annotated versions of the source program. In each iteration the annotations are refined. The vanilla interpreter is first extended to handle annotated programs (Listing 9.20). All calls in the program are surrounded by an annotation, the first argument represents a unique program point and the second argument the actual call. In Listing 9.20 all calls marked as **rescall** are ignored as they are not executed during specialisation and can therefore play no part in the possible non-termination of the specialisation process. The ease of handling the different annotation types here demonstrates one of the benefits of writing program transformation as interpreters. The **memo** and **unfold** annotations are treated in the same manner as they are both effectively executed at specialisation time, though their behaviour will differ when the interpreter is extended to find possible loops.

```
solve([]).
solve([unfold(PP,H)|T]) :-
        solve_atom(H),
        solve(T).
solve([call(PP,H)|T]) :-
        call(H),
        solve(T).
solve([memo(PP,H)|T]) :-
        solve_atom(H), solve(T).
solve([rescall(PP,H)|T]) :-
        solve(T).

solve_atom(H) :-
        get_clause(H,Bdy), solve(Bdy).
```

LISTING 9.20: The vanilla interpreter is extended to handle annotated programs with program point information.

The next addition to the interpreter is to look for possible program loops. A possible loop exists in an annotated program if there is a finite derivation (through unfolded calls) from a program point back to the same program point. Listing 9.21 contains `bin_solve/3` an extended `solve/1` for calculating binary clauses. Again each annotation in the program is handled differently, **memo** and **rescall** annotations are ignored (they can not contribute to loops in an annotated program), **call** annotations are called and **unfold** annotations are treated specially to find potential loops. Calls marked as **unfold** are handled in three different ways:

1. If the analysis is currently looking for a loop to the *same* program point as it is unfolding it simply succeeds (it has found a loop from the program point back to itself).

2. It descends into the unfolded call looking for any additional loops.

3. The predicate `solve_atom/1` is called to perform any needed substitutions and continue looking for loops on the current body.

```
bin_solve(PP,[unfold(PP,H)|_T],H).
bin_solve(PP,[unfold(PP1,H)|_T],RecCall) :-
        bin_solve_atom(PP,H,RecCall).
bin_solve(PP,[unfold(_,H)|T],RecCall) :-
        solve_atom(H),  /* solve it and then find recursive calls for T */
        bin_solve(PP,T,RecCall).
bin_solve(PP,[memo(_,_)|T],RecCall) :-
        bin_solve(PP,T,RecCall).
bin_solve(PP,[call(_,Call)|T],RecCall) :-
        call(Call),
        bin_solve(PP,T,RecCall).
bin_solve(PP,[rescall(_,_)|T],RecCall) :-
        bin_solve(PP,T,RecCall).

bin_solve_atom(PP,H,Rec) :-
        get_clause(H,Bdy), bin_solve(PP,Bdy,Rec).
```

LISTING 9.21: The vanilla interpreter is extended to look for loops.

Running the extended interpreter on the example program Listing 9.19 produces the set of clauses shown in Listing 9.22. The `bin_solve_atom__1/3` clauses represent the loops from the recursive call in `p/1` back to itself. The `bin_solve_atom__2/3` predicate fails as there are no possible loops from `r/2` to `p/1`. The `solve_atom__1` and `solve_atom__2` map directly to the original program clauses in Listing 9.19, but all calls marked **memo** or **rescall** are removed.

```
bin_solve_atom__1(1, p(A), p(B)) :-
        solve_atom__2(A, B).
bin_solve_atom__1(1, p(A), p(B)) :-
        solve_atom__2(A, C),
        bin_solve_atom__1(1, p(C), p(B)).
bin_solve_atom__1(1, p(A), p(B)) :-
        bin_solve_atom__2(0, r(A,_), p(B)).

bin_solve_atom__2(0, r(_,_), p(_)) :-
        fail.

solve_atom__1(0).
solve_atom__1(A) :-
        solve_atom__2(A, B),
        solve_atom__1(B).
solve_atom__2(s(A), A).
```

LISTING 9.22: Output from specialising binary clause interpreter for Listing 9.19

Calling the specialised program, Listing 9.22, for the goal `bin_solve_atom__1(A,B,C).` produces the answers in Listing 9.23. This corresponds to the infinite number of binary clauses already demonstrated (a loop exists from the program point back to itself and a loop exists going through the loop an arbitrary number of times). In Chapter 5 this output is then analysed by a convex hull analyser to prove termination properties of the annotated program.

```
| ?- bin_solve_atom__1(A,B,C).
A = 0,
B = p(s(_A)),
C = p(_A) ? ;

A = 0,
B = p(s(s(_A))),
C = p(_A) ? ;

A = 0,
B = p(s(s(s(_A)))),
C = p(_A) ? ;
...
```

LISTING 9.23: Running the specialised binary clause program

This section extended the vanilla interpreter to produce binary clauses for input programs. The output from specialisation is not a program that is meant to be executed as in the previous examples but is instead analysed. Using the interpreter approach allows the programmer to easily and quickly develop quite complex transformation, in this example the handling of the different annotations is done by adding simple rules to the interpreter.

Benchmarks are not presented for this interpreter, as the results are program transformations that are not meant to be executed (and would produce an infinite number of answers).

9.6 Lloyd Topor Transformation

The paper *Making Prolog More Expressive* (Lloyd and Topor, 1984) introduced extended programs and goals for logic programming. The extended programs can contain clauses that have an arbitrary first-order formula in their body. The only requirement for executing the transformed programs is a sound form of the negation as failure rule. This example will demonstrate that it is possible to achieve the same program transformation by specialising an intuitive Prolog interpreter.

In Lloyd and Topor (1984) an extended program P is transformed into a general program P', called the *general form* of P, using a set of transformation rules. The rules (a) ... (j) are applied until no more transformations can be applied. Lloyd and Topor (1984) proves this process terminates and always gives a general program.

(a) Replace $A \leftarrow \alpha \wedge \neg(V \wedge W) \wedge \beta$
by $A \leftarrow \alpha \wedge \neg V \wedge \beta$
and $A \leftarrow \alpha \wedge \neg W \wedge \beta$

(b) Replace $A \leftarrow \alpha \ \wedge \forall x_1 \ldots x_n W \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge \neg\exists x_1 \ldots x_n \neg W \wedge \ \beta$

(c) Replace $A \leftarrow \alpha \ \wedge \neg\forall x_1 \ldots x_n W \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge \exists x_1 \ldots x_n \neg W \wedge \ \beta$

(d) Replace $A \leftarrow \alpha \ \wedge V \leftarrow W \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge V \wedge \ \beta$
and $A \leftarrow \alpha \ \wedge \neg W \wedge \ \beta$

(e) Replace $A \leftarrow \alpha \ \wedge \neg(V \leftarrow W) \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge W \wedge \neg V \wedge \ \beta$

(f) Replace $A \leftarrow \alpha \ \wedge (V \vee W) \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge V \wedge \ \beta$
and $A \leftarrow \alpha \ \wedge W \wedge \ \beta$

(g) Replace $A \leftarrow \alpha \ \wedge \neg(V \vee W) \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge \neg V \wedge \neg W \wedge \ \beta$

(h) Replace $A \leftarrow \alpha \ \wedge \neg\neg W \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge W \wedge \ \beta$

(i) Replace $A \leftarrow \alpha \ \wedge \exists x_1 \ldots x_n W \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge W \wedge \ \beta$

(j) Replace $A \leftarrow \alpha \ \wedge \neg\exists x_1 \ldots x_n W \wedge \ \beta$
by $A \leftarrow \alpha \ \wedge \neg p(y_1, \ldots, y_k \wedge \ \beta$
) and $p(y_1, \ldots, y_k) \leftarrow \exists x_1 \ldots x_n W$
where $y_1, \ldots, y_k$ are the free variables in $\exists x_1 \ldots x_n W$ and p is a new predicate not already appearing in the program.

For example, take the definition of subset in Figure 9.1. The definition is written in a clear mathematical way and has been expressed in a form similar to the specification of the problem. However this specification cannot be executed directly in Prolog as it contains $\forall$ and $\leftarrow$ in its body.

$$x \subseteq y \leftarrow \forall u(u \in y \leftarrow u \in x)$$

FIGURE 9.1: A definition of subset

Transforming the extended program in Figure 9.1 using the transformation rules produces the general program Figure 9.2. The general program uses negation and requires a sound implementation of negation as failure. The general program, rewritten in standard Prolog syntax is given in Listing 9.24. The $\in$ operator has been replaced by calls to `mem/2`.

$$x \subseteq y \leftarrow \neg p(x,y)$$
$$p(x,y) \leftarrow \neg(u \in y) \wedge u \in x$$

FIGURE 9.2: The transformed general program for subset

```
subset(X,Y) :-
        \+p(Y,X).
p(X,Y) :-
        \+mem(A,X),
        mem(A,Y).
mem(A, [A|_]).
mem(A, [_|B]) :-
        mem(A, B).
```

LISTING 9.24: Prolog version of general subset program Figure 9.2

A safe computation rule for negation as failure can be implemented by delaying selected negative literals until they have become ground. This delay can be achieved by using coroutines. Specifying a guard condition on the negation that all variables must be ground will ensure a safe computation. Listing 9.24 is converted to use `when/2` (Listing 9.25), the call to `subset/2` can now be safely made and will delay until both input arguments are ground.

```
subset(X,Y) :-
        when(ground([Y,X]),\+p(Y,X)).
p(X,Y) :-
        when(ground([A,X]),\+mem(A,X)),
        mem(A,Y).
mem(A, [A|_]).
mem(A, [_|B]) :-
        mem(A, B).
```

LISTING 9.25: Prolog version of general subset program Figure 9.2 using coroutining to delay the negation.

```
| ?- subset([a,b,c], S), S = [d,e,f].
no
| ?- subset([a,b,c], S), S = [d,e,f,a,b,c].
S = [d,e,f,a,b,c] ?
yes
```

The vanilla interpreter is extended to handle a more natural form of input programs. Instead of using list skeletons as in the previous examples, it supports standard conjunctions ((_,_)) of literals. The predicate `solve/1` is extended to decompose conjunctions, recursively calling `solve/1` on each part. The second clause matches the actual calls in clause body and calls `solve_literal/1`.

```
solve(',' (A,T)) :- solve(A), solve(T).
solve(A) :- nonvar(A), A\= ','(_,_), solve_literal(A).
```

New operators are defined for implication and negation in the interpreter. The functions `forall/2` and `exists/2` are reserved for ∀ and ∃.

```
:- op(950,yfx,'=>'). % implies right
:- op(950,yfx,'<='). % implies left
:- op(850,yfx,'or'). % or
:- op(800,yfx,'&').  % and
:- op(750,fy,'~').   % not
```

Two new predicate are created for handling body literals, a positive (solve_literal/1) and a negative one (not_solve_literal/1). The definition for solve_literal/1 contains the basic clauses for dealing with true and false, if a *not* operator is encountered control is passed to not_solve_literal/1. The & operator performs a conjunction of the two arguments and the or operator performs a disjunction.

```
solve_literal(true).
solve_literal(false) :- fail.
solve_literal('~'(L)) :- not_solve_literal(L).  %% Call negative version of solve_literal
solve_literal('&'(A,B)) :- solve_literal(A), solve_literal(B).
solve_literal(or(A,_)) :- solve_literal(A).
solve_literal(or(_,B)) :- solve_literal(B).
solve_literal(A) :- is_user_pred(A),solve_atom(A).
solve_literal(A) :- is_built_in(A),call(A).

solve_atom(A) :- my_clause(A,B), solve(B).
```

The clauses for not_solve_literal/1 are similar, but define the negated counterparts of solve_literal/1. Notice the handling of & and or, DeMorgan's laws ($\neg(A \vee B) \equiv (\neg A \wedge \neg B)$ and $\neg(A \wedge B) \equiv (\neg A \vee \neg B)$) are applied and solve_literal/1 is called. If a Prolog negation is required then coroutining is used to delay the negation until it is safe.

```
not_solve_literal(true) :- solve_literal(false).
not_solve_literal(false) :- solve_literal(true).
not_solve_literal('~'(L)) :- solve_literal(L).
not_solve_literal(or(A,B)) :- solve_literal('&'('~'(A), '~'(B))).
not_solve_literal('&'(A,B)) :- solve_literal(or('~'(A), '~'(B))).
not_solve_literal(A) :- is_user_pred(A), not_solve_atom(A).
not_solve_literal(A) :- is_built_in(A),A \= =(_,_), A \= \=(_,_),
    term_variables(A,Vars),
    when(ground(Vars), \+(call(A))).

not_solve_atom(A) :-
    term_variables(A,Vars),
    when(ground(Vars), \+(solve_atom(A))).
```

Implication is handled by transforming it into a disjunction (using $A \rightarrow B \equiv (\neg A \vee B)$ and $A \leftarrow B \equiv (A \vee \neg B)$) , which will in turn be handled by the previous clauses.

```
solve_literal('=>'(A,B)) :- solve_literal(or('~'(A),B)).
solve_literal('<='(A,B)) :- solve_literal(or(A,'~'(B))).

not_solve_literal('=>'(A,B)) :- solve_literal('~'(or('~'(A),B))).
not_solve_literal('<='(A,B)) :- solve_literal('~'(or(A,'~'(B)))).
```

The exists(X,A) operator is implemented by making a copy of the atom, A, and renaming all occurrences of X. This is done to avoid name clashes and solve_literal/1 is called on the copy. The `forall(X,A)` is transformed into a `exists(X, A)` and the negative version of `forall(X,A)` is transformed into a positive `exists` using standard logic laws. The negative version of `exists/2` must be handled directly, a `when/2` declaration is added to ensure the negation is only selected when the arguments are instantiated.

```
solve_literal(exists(X,A)) :- rename(X,A,CopyA),solve_literal(CopyA).
solve_literal(forall(X,A)) :- not_solve_literal(exists(X,'~'(A))).

not_solve_literal(forall(X,A)) :- solve_literal(exists(X,'~'(A))).
not_solve_literal(exists(X,A)) :- force_not_solve_literal(exists(X,A)).

force_not_solve_literal(Formula) :-
    get_free_variables(Formula ,[] ,[] ,Vars),
    when(ground(Vars), \+(solve_literal(Formula))).
```

Specialising the interpreter for the subset extended program in Listing 9.26 produces the residual program Listing 9.27.

```
subset(A,B) :-
        forall(C,member(C,B)<=member(C,A)).
```

LISTING 9.26: Subset extended program in interpreter form

The specialised program is almost identical to the hand crafted program in Listing 9.25. All negations have been enclosed by `when/2`, this will ensure the negation will only be performed when the goal is properly ground. This transformation was performed by specialising an interpreter. Importantly the interpreter was not simply an encoding of the transformation rules from Lloyd and Topor (1984) but an intuitive interpreter for Prolog handling the extended program syntax.

```
solve_atom(subset(A,B)) :-
        solve_atom__0(A, B).
solve_atom__0(A, B) :-
        when(ground([A,B]), \+solve_literal__1(A,B,_)).
solve_literal__1(A, B, _) :-
        when(ground([B,C]), \+member(C,B)),
        member(C, A).
```

LISTING 9.27: Specialising the interpreter for the subset extended program (Listing 9.26)

Benchmark	Iterations	Specialisation Time	Original Runtime	Specialised Runtime	Relative Runtime
Lloyd Topor	100000	60ms	8580ms	2370ms	0.28

TABLE 9.9: Benchmark figures for the Lloyd Topor interpreter

Benchmark	Original Program Size	Specialised Program Size	Relative Program Size
Lloyd Topor	19877 bytes	1685 bytes	0.08

TABLE 9.10: Program size comparison for the Lloyd Topor interpreter

9.7 Summary

This chapter focused on the specialisation of interpreters. A series of different interpreters were presented and specialised with good results. The interpreters in this chapter are all extensions of the basic vanilla interpreter for Prolog. It was shown that in addition to extending the base language (e.g. adding caching), specialised interpreters can also be used for program analysis (e.g. in the case of the binary clause interpreter). All the examples given in this chapter were interpreters for logic languages. Chapter 3 presented an interpreter for a functional language.

When specialising interpreters, the interpreter is partially evaluated with respect to the source program, the hope is to specialise away the overhead of interpretation and produce a "compiled" object program. These specialised object programs combine the features and style of the interpreter with the algorithm of the source program.

Jones optimality (called the "optimality criterion" in Jones et al. (1993)) was demonstrated for the vanilla self-interpreter. A self-interpreter was chosen as it is easy to judge to what extent the interpretive overhead has been removed, as both the object and specialised program are in the same language. It is easy to see that the specialisation in Section 9.1 is Jones optimal as the object and specialised programs are identical up to predicate and variable naming. The derived extensions to the vanilla interpreter were also Jones optimal, e.g. when specialising the debugging interpreter for program P with none of its predicates being spied on we will always get a program equivalent to P (with no overhead from interpretation). In the functional community there has been a lot of recent interest in Jones Optimality. In particular Glück (2002) shows the theoretical interest of having Jones optimal specialiser and the results should also be relevant for logic programming.

Perhaps the most complicated interpreter specialised in this thesis is LIX itself. The partial evaluator, LIX, can be viewed as an interpreter for annotated source programs. Chapter 3 discussed the extensions required for self-application and many of these same extensions are used in specialising the interpreters in this chapter. Using the Futamura projections (Futamura, 1971) it is possible to build compilers for a particular interpreter by self-application (as demonstrated in Chapter 3). These compilers can be reused on any object program for a given interpreter and provide a convenient means to make efficient program transformers.

The interpreters in this chapter are written in a "natural" way and produce good specialised programs. However, it is possible to write programs in a "natural" manner that do not specialise well. Jones (2004, 1996) presents an interesting discussion about writing interpreters for specialisation. One must be careful to write the program with specialisation in mind, and ensure a clean separation of binding times so as much as possible can be done at specialisation time.

The experimental results highlight the speedups that can be obtained through the specialisation of interpreters and show that the system can be a useful basis for generating compilers for high-level languages. The speedup is often a result of removing the parsing overhead of the interpreter, on the more complicated examples an additional speedup is obtained by removing the associated checks for the language extensions.

Benchmark	Iterations	Specialisation Time	Original Runtime	Specialised Runtime	Relative Runtime
Vanilla	3000000	20ms	26280ms	4940ms	0.19
Debug (No debug)	3000000	30ms	42180ms	5090ms	0.12
Debug (trace rev/2)	30000	30ms	13470ms	12350ms	0.92
Profile (no Profile)	3000000	30ms	91280ms	5280ms	0.06
Profile (rev/2,append/3)	30000	30ms	31630ms	20920ms	0.66
Cache (no Cache)	100	20ms	76950ms	4780ms	0.06
Cache (fib/2)	100000	20ms	42880ms	23320ms	0.54

TABLE 9.11: Benchmark figures for the interpreters

The relative code sizes of the specialised and original (combined interpreter and source program) program can be seen in Table 9.12. When no special features are used the specialised code size is generally smaller. For example, in the debugging interpreter specialising P with no debugging information produces a program equivalent to P. The code size of the original program will always have the additional size of the interpreter. In the examples where extra code is added to the specialised program, for example when tracing predicates in the debugging interpreter, the relationship varies depending on how many predicates are traced. In comparison the original program only grows by the constant size of the interpreter.

Benchmark	Original Program Size	Specialised Program Size	Relative Program Size
Vanilla	2082 bytes	1462 bytes	0.70
Debug (No debug)	6237 bytes	1625 bytes	0.26
Debug (trace rev/2)	6356 bytes	5032 bytes	0.79
Profile (no Profile)	9594 bytes	1799 bytes	0.19
Profile (rev/2,append/3)	9594 bytes	7222 bytes	0.75
Cache (no Cache)	5810 bytes	1485 bytes	0.26
Cache (fib/2)	5810 bytes	2512 bytes	0.43

TABLE 9.12: Program size comparison for interpreters

Chapter 10

Conclusion and Future Work

The main stated aim of this thesis is to make Prolog partial evaluation *practicable*. In itself this represents many different and worthwhile challenges. Partial evaluation is a complex process, unlike standard evaluation control decisions are based on partial sections of the full input data. For a partial evaluation system to be usable by a wider audience it must be powerful enough to specialise real life programs, including the extensions found in modern Prolog implementations. It should also be simple to use but still provide the user with enough control to get the most out of the specialisation process. The overall aim was to develop an accessible framework for partial evaluation of Prolog programs.

To make the system simple to use it should be as automated as possible, but should not compromise the ability of experienced users to control the specialisation process. Online partial evaluators are in general more automatic than offline systems as they do not require an annotated source program. Online systems make all of their decisions during the specialisation process and therefore are potentially more precise as, in contrast to offline techniques, the decision can be based on actual static data values. However, specialising the same program for slightly different static data may have a dramatic effect on the control decisions and cause unpredictable behaviour. While source programs do not have to be annotated, online systems generally offer a large selection of different control strategies. Choosing the correct strategy for the problem at hand can require as much intimate knowledge of the specialiser as hand annotating in an offline setting.

This thesis concentrated on offline partial evaluation techniques. An offline partial evaluator was developed, combined with an automatic binding-time analysis to annotate source programs and a self-tuning algorithm to optimise these annotations. This not only provides a high degree of automation but also allows the user to manually modify the annotations and steer the specialisation process. This steering is of particular importance in the specialisation of interpreters, since previous research using automated online techniques has shown that it is difficult to achieve consistent results when specialising interpreters (in general and especially trying to achieve Jones optimality).

As already discussed, offline partial evaluators require a binding-time analysis phase. The role of binding-time analysis is shown in Figure 10.1, given a source program and a description of the entry points it produces an annotated program.

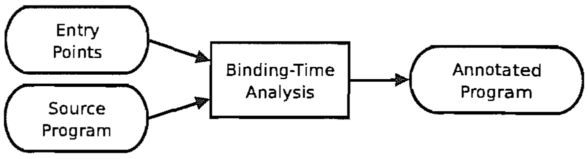


FIGURE 10.1: The role of the binding-time analysis

This annotated program guides the specialisation process and should guarantee the partial evaluation terminates with a correct residual program. Classically the binding-time analysis was performed manually, requiring considerable expertise. The static data must be propagated from the entry query throughout the program, and then based on this information each call should be correctly annotated. The decision of how to annotate a call is usually based on how much information will be available at specialisation time. The binding-time analysis algorithm developed in Chapter 5 is fully automatic and allows for the rapid annotation of source programs. It combines state of the art termination analysis techniques with a type-based abstract interpretation for propagating binding types. The algorithm supports built-ins as well as user defined predicates. Unlike many previous algorithms, the one presented in Chapter 5 is fully implemented and integrated into the PYLOGEN interface. The algorithm was tested on a series of examples (Section 5.7) with promising results, it correctly produced valid annotated programs in a reasonable amount of time.

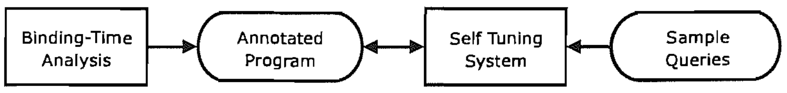


FIGURE 10.2: The role of the self-tuning algorithm

The binding-time analysis guarantees a safe set of annotations, performing as many operations at specialisation time as possible. However, the most aggressive annotations do not always create the best quality specialised code. Chapter 6 presented a self-tuning system, which derived its own specialisation control for the particular Prolog compiler and underlying architecture by trial and error. The algorithm takes an annotated program and produces an annotated program (Figure 10.2). Using a set of representative sample queries it refines the annotations, controlling the trade-off between code explosion, speedup gained and the actual specialisation time. The user can specify the importance of each of the factors in determining the quality of the produced code, tailoring the specialisation to the particular problem at hand. The main insight of the technique was that the annotations can be used as a basis of a genetic algorithm.

The separation of the binding-time analysis and self-tuning proves to be useful. The role of the binding-time analysis can be clearly define to simply "do as much as possible" while ensuring termination of the specialisation. There is no need to include heuristics for quality of code. The optimisation and performance of the annotations is left to the self-tuning algorithm which in turn makes use of the binding-time analysis to ensure the final annotations are correct. The empirical evaluation of the self-tuning algorithm has been very encouraging, it successfully avoided many of the pitfalls of partial evaluation.

For a partial evaluation system to be useful it must support the specialisation of real life languages. This includes the extensions used in modern Prolog systems. This thesis introduced extensions for coroutining and constraint logic programming.

Coroutining allows the delayed execution of Prolog goals using guards. The goal will not be executed until the guard condition has been satisfied. This gives the programming control over the Prolog selection rule in a declarative fashion. Chapter 7 discussed the partial evaluation of coroutines, in particular the Prolog `when/2` predicate. New annotations were introduced to the system to handle `when/2` calls. These annotations were used in the specialisation of the Lloyd-Topor interpreter in Chapter 9.

Based on the idea of delayed execution a new annotation was introduced using guard conditions. The **online** annotation delays unfolding until a guard criteria is satisfied during specialisation, if the guard is not satisfied during specialisation the goal is residualised and added to the specialised program. The benefit of this annotation is that it provides an online control to the offline algorithm, the guard condition is evaluated at specialisation time so can makes decisions based on the actual values of the static data.

The constraint logic programming (CLP) extensions demonstrate that the system is extensible and provides support for an important paradigm in logic programming. CLP allows the user to model a problem using a set of constraints in a particular domain. The system is capable of non-trivial offline specialisation of non-declarative programs containing CLP expressions, to the best of our knowledge the first of its kind.

Chapter 9 presented a series of increasingly complicated interpreters, all of which were specialised with good results. The chapter demonstrated that specialising interpreters can be used to extend languages with new features, for example the caching interpreter. Specialising interpreters can also be used for program analysis. The binary clause semantics is used to verify termination properties of programs during the binding-time analysis, in the algorithm a program is transformed into binary clauses by partially evaluating an interpreter. Writing interpreters to manipulate programs can be a high level and natural method for program transformation, and when combined with partial evaluation it can produce efficient program transformers. The interpreters specialised demonstrate the expressive power of the developed system. Jones Optimality was shown, removing all of the overhead of interpretation through specialisation.

Self-application can be used to improve the efficiency of the specialisation process, especially when the same program is to be specialised for different sets of static input. The separation of the different phases in offline partial evaluation is favourable when self-applying as only the final simplified specialisation phase has to be self-applied. The LIX system proves to be an effective and surprisingly simple self-applicable partial evaluator system for Prolog. Chapter 3 demonstrated that contrary to popular belief the ground representation is not required for self-application. The LIX system can produce specialised specialisers which are highly optimised through self-application and can handle non-declarative and higher order predicates. Using the 2^{nd} Futamura projection the LIX system can produce compilers for a given interpreter and a compiler generator via the 3^{rd} projection.

The PYLOGEN graphical interface combines the techniques developed in this thesis into an integrated environment for writing and specialising Prolog programs. Prolog source programs can be loaded into the interface and annotated without needing to modify the underlying source code, all the annotations are stored separately. The annotating itself can either be done by hand or using the automatic binding-time analysis. Annotating by hand is performed using the colour coded interface, where each annotation is represented by a colour overlay on the original source program and can be modified using point and click. The automatic binding-time analysis and self-tuning algorithm can be invoked directly from PYLOGEN, the results can then be modified by hand if needed. As is often the case a combination of automatic analysis and the ability to modify the annotations by hand can prove to be the most powerful technique.

The interface provides a quick and easy way to specialise programs. Once the entry goal has been entered, a click of a button specialises the program and displays the results back to the user. The specialised code can be compared against the original using the built-in benchmarking suite. The ability to quickly specialise programs and see the results of the specialisation in the interface allows users to quickly pick up the effects of the different annotations on the specialisation process. A built-in Prolog shell can be used to execute the specialised programs and inspect their behaviour.

10.1 Future Work

The graphical interface is still being developed and improvements are ongoing to make the system more usable and to add new features. The handling of error conditions is an area for improvement. When specialisation fails to terminate due to either local or global termination issues few specialisation systems provide any useful feedback. Providing understandable error messages that pinpoint possible causes would help to improve the difficult learning curve associated with partial evaluation.

Another aim is to make the system as accessible as possible. TerminWeb[1] provides a webpage interface to a powerful termination analyser, it is hoped to develop a similar system for PYLOGEN. This would provide new users with a simple way to experiment with the specialiser.

The self-tuning algorithm and the binding-time analysis are both iterative algorithms. There is room for optimisation by improving caching between iterations so previous answers can be reused. This should prove especially useful in the convex hull analyser used in the binding-time analysis. The self-tuning technique lends itself well to parallelisation, and initial tests look promising. The system can be extended to work over a cluster of computers, another level of normalisation would be needed to compensate for the different machines.

Currently the implementation of the binding-time analysis guarantees correctness and termination at the local level, and correctness but not yet termination at the global level. However the framework can be extended to ensure global termination as well. Indeed, the binary clause interpreter used for local termination can also compute memoisation loops, and so it can apply exactly the same procedure. Then, if a memoised call is detected to be unsafe the non-decreasing arguments must be marked as dynamic (or weakened to a more general filter).

The implemented BTA algorithm supports multiple norms by combining the results from two separate executions, the technique could be adapted to use the combined norms from Bruynooghe et al. (2003), allowing the detection of more complex termination conditions based on the relationship between different norms. The BTA should also be extended to include more complex binding types and the more advanced annotations. This would allow it to handle complex interpreters, an ideal application for specialisation.

The examples in this thesis concentrated on the specialisation of interpreters, which is often cited as an ideal application for partial evaluation. We hope to develop the system to provide more features specially designed for the development and partial evaluation of interpreters. Such features may include templates for common base languages (for example the vanilla interpreter) and better support for debugging interpreters used in specialisation.

To identify the problems associated with the specialisation of larger industrial style applications we hope to successfully apply our techniques to more complex examples. Notably the B interpreter from ProB (Leuschel and Butler, 2003), a model checker for the B formal specification language. The interpreter is used to interpret B specifications

[1]`http://www.cs.bgu.ac.il/~mcodish/TerminWeb/`

during model checking and we expect to able to achieve considerable speedups through partial evaluation.

There is still scope to improve the code generated during CLP specialisation. Using sophisticated constraint reordering techniques or converting CLP operations into standard Prolog arithmetic (e.g., using `is/2`) when it is safe.

An *offline* partial evaluator was discussed in this thesis and we hope to develop the system to include *online* techniques. This hybrid approach could include features from the ECCE (Leuschel et al., 1998) online partial evaluation system, to act as a watchdog monitoring the offline system. This could be developed into a more integrated algorithm allowing the specialiser to cope with a mixture of annotated and unannotated code.

Chapter 7 introduced the **online** annotation, using coroutining to delay unfolding. It is possible that this technique could be developed further and used as a basis for a partial evaluation system. Each atom in the body could be given a guard condition to prevent unfolding until the correct criteria has been met. During partial evaluation an atom is only selected if the guard condition is satisfied, any atoms that remain unselected are reconstructed into the final residual code (with the correct answer substitutions calculated during partial evaluation). More research needs to be done to see if this is a viable method.

Appendix A

PyLogen Tutorial

The PYLOGEN system is an implemented tool for specialising Prolog programs. The specialisation engine is written in SICStus Prolog and the interface is a mixture of Python and Tk. This section will explain the basic functionality through a simple tutorial.

A.1 Starting PYLOGEN

Follow the online instructions for installing PYLOGEN . To start PYLOGEN :

- OS X:
  `[~] pythonw logen.py`
- Windows and Linux:
  `[~] python logen.py`

Regular Expression Example

For this tutorial we use a simple regular expression parser (Listing A.1). The interpreter takes a basic regular expression and a string (represented by a list of atoms) and succeeds if the string matches the regular expression (Listing A.2). The empty pattern, ϵ, is represented by the special constant eps.

```
match(Regexp,String) :- regexp(Regexp,String,[]).

regexp(eps,T,T).
regexp(X,[X|T],T) :- atomic(X).
regexp(+(A,_B),Str,DStr) :- regexp(A,Str,DStr).
regexp(+(_A,B),Str,DStr) :- regexp(B,Str,DStr).
regexp(.(A,B),Str,DStr) :- regexp(A,Str,I), regexp(B,I,DStr).
regexp(*(A),S,DS) :- regexp(.(A,*(A)),S,DS).
regexp(*(A),S,S).
```

LISTING A.1: An interpreter for regular expressions

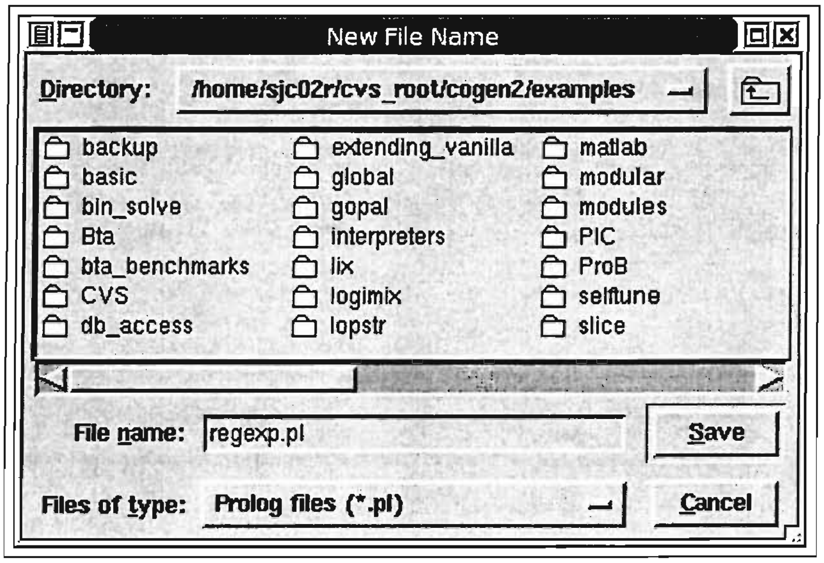
```
| ?- match(.(*(a),b), [a,a,a,b]).
yes
% source_info
| ?- match(.(*(a),b), [a,a,a,b,c]).
no
```

LISTING A.2: Using the regular expression interpreter

A.2 Specialising the Regular Expression Interpreter

Create a new file

Click on the **new** icon or select **new** from from the **File** menu. In the dialog box select a location for the new file and call it *regexp.pl.*


New File Name

Directory: /home/sjc02r/cvs_root/cogen2/examples

backup, basic, bin_solve, Bta, bta_benchmarks, CVS, db_access, extending_vanilla, global, gopal, interpreters, lix, logimix, lopstr, matlab, modular, modules, PIC, ProB, selftune, slice

File name: regexp.pl

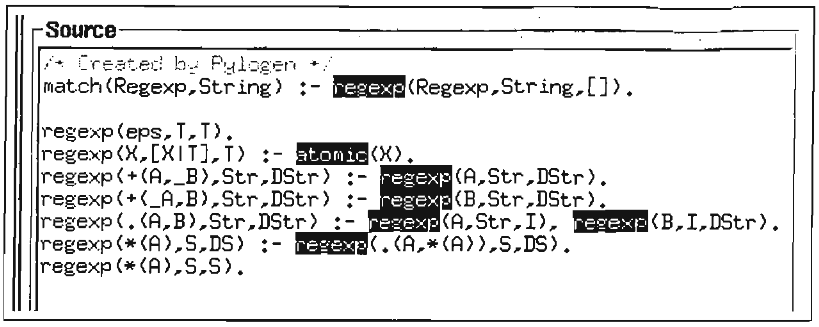
Files of type: Prolog files (*.pl)

Edit the new file

The default mode in PYLOGEN edits the annotations associated with the current source code. The top left pane contains the sourcecode, the top right pane contains the filter declarations and the lower pane displays the different output modes. To actually edit the sourcecode we must first enter **sourcecode mode**. Click on the **edit** icon or select **sourcecode mode** from the **Edit** menu.

Once in **sourcecode mode** add the sourcecode from Listing A.1 into the top left pane. When you have finished typing entering the sourcecode click the **save** icon or select

annotation mode from the **Edit** menu. If there is a parse error you will be notified by an error message, if everything is correct the source code we be reloaded and annotated using the **unknown** annotation.



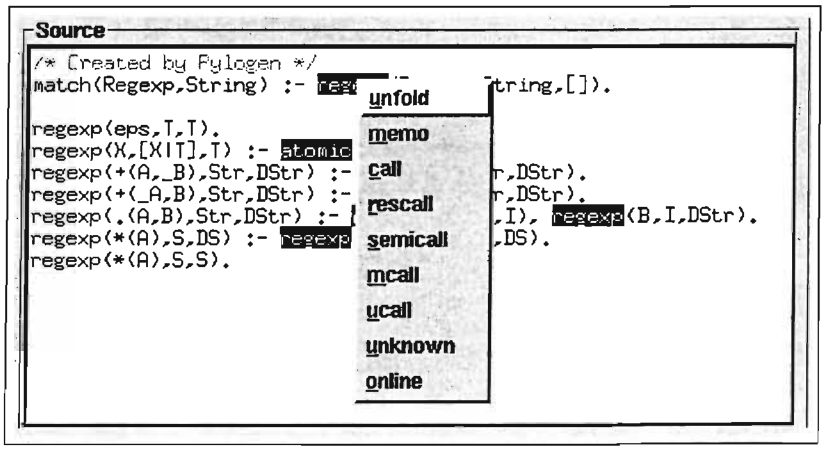
```
/* Created by Pylogen */
match(Regexp,String) :- regexp(Regexp,String,[]).

regexp(eps,T,T).
regexp(X,[X|T],T) :- atomic(X).
regexp(+(A,_B),Str,DStr) :- regexp(A,Str,DStr).
regexp(+(_A,B),Str,DStr) :- regexp(B,Str,DStr).
regexp(.(A,B),Str,DStr) :- regexp(A,Str,I), regexp(B,I,DStr).
regexp(*(A),S,DS) :- regexp(.(A,*(A)),S,DS).
regexp(*(A),S,S).
```

Annotate the new file

The **unknown** annotation is used to identify unannotated calls is the program. To specialise the regular expression interpreter we must first properly annotate the program. We assume that the regular expression will be known at specialisation time, **static**, but the string to match against will be **dynamic**.

The predicate `match/2` is an entry point into the regular expression interpreter, it simply calls `regexp/3` with an empty list as the third argument. The third argument contains the "left over" part of the string, so `match/2` only succeeds on an exact match. We choose to **unfold** the call to `regexp/3`, clicking on the call will display the annotation menu. Select **unfold** from the menu to annotate this call.

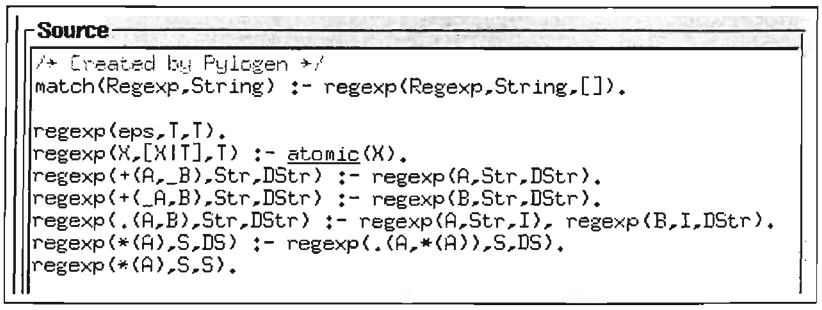


```
/* Created by Pylogen */
match(Regexp,String) :- reg             tring,[]).

regexp(eps,T,T).
regexp(X,[X|T],T) :- atomic
regexp(+(A,_B),Str,DStr) :-             r,DStr).
regexp(+(_A,B),Str,DStr) :-             r,DStr).
regexp(.(A,B),Str,DStr) :-              ,I), regexp(B,I,DStr).
regexp(*(A),S,DS) :- regexp             ,DS).
regexp(*(A),S,S).
```

Now we move onto annotate the `regexp/3` predicate. The first call is to the built-in predicate `atomic/1`. If we have an atomic item in our pattern then we simply look for that item in the input string. As the first argument is **static** (it was passed directly from `match/2`), we can safely make this call at specialisation time. Mark the call to `atomic/1` as call, again by clicking on the call and selecting `call`.

The remaining calls are all recursive calls to the `regexp/3` predicate. The annotations in a program ensure it will terminate at specialisation time. When annotating a program by hand it is important to keep in mind which calls are safe to **unfold** and which must be marked **memo**. In the case of the regular expression interpreter we know the pattern is **static**, so as long as we are decreasing the pattern each call we are going to eventually terminate. Inspecting the clauses shows that the only unsafe call is in handling of the `*(Pattern)`, this allows an unbounded number of matches against `Pattern`. As we do not have the string to match against we must mark the recursive call to `regexp(.(A,*(A)),S,DS)` as **memo**. The rest of the calls can be marked **unfold**.

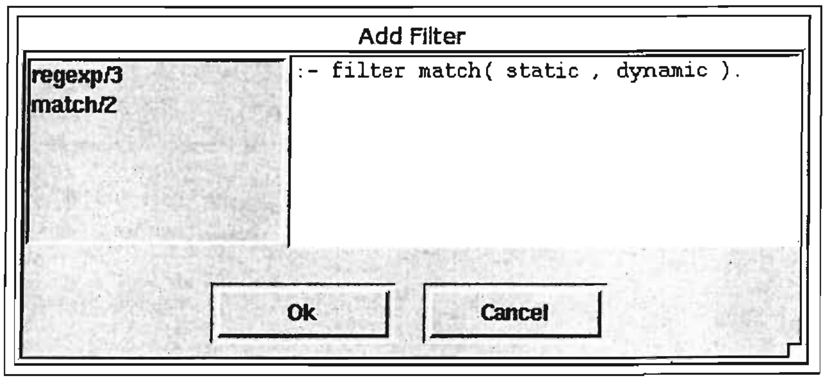


```
/* Created by Pylogen */
match(Regexp,String) :- regexp(Regexp,String,[]).

regexp(eps,T,T).
regexp(X,[X|T],T) :- atomic(X).
regexp(+(A,_B),Str,DStr) :- regexp(A,Str,DStr).
regexp(+(_A,B),Str,DStr) :- regexp(B,Str,DStr).
regexp(.(A,B),Str,DStr) :- regexp(A,Str,I), regexp(B,I,DStr).
regexp(*(A),S,DS) :- regexp(.(A,*(A)),S,DS).
regexp(*(A),S,S).
```

Add an entry point

We have now annotated all of the clauses in the regular expression program. Now we must tell the specialiser something about the entry point of the program. We intend to call `match/2` with a **static** first argument and a **dynamic** second argument. Click the **insert filter** icon or select **insert filter** from the **Edit** menu.



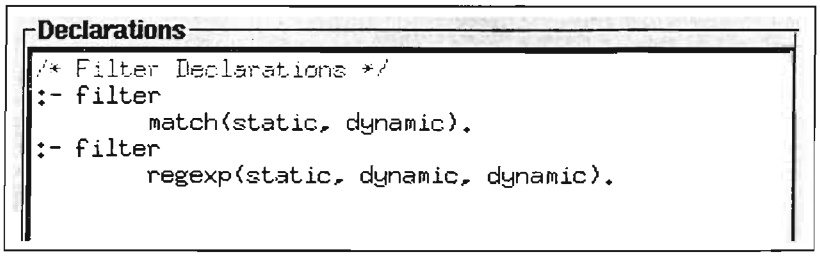
```
:- filter match( static , dynamic ).
```

The left hand side contains a list of predicates appearing in the source program. Double click on `match/2` to create an empty filter declaration. Change the declaration to make the first argument **static**.

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

Filter Propagation

As a call to regexp/3 is marked as **memo** we will also need to provide a filter declaration for regexp/3. This can be done manually, inferring that regexp/3 is **static**, **dynamic**, **dynamic** from the initial call in match/2. We can also use the filter propagation discussed in Chapter 5. Save the file and select **propagate filters** from the **BTA** menu.



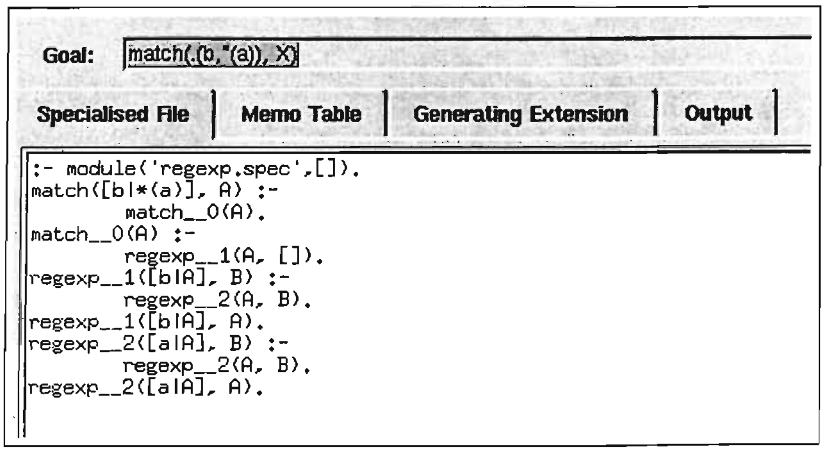
```
/* Filter Declarations */
:- filter
        match(static, dynamic).
:- filter
        regexp(static, dynamic, dynamic).
```

Specialising the regular expression interpreter

Now we have annotated the interpreter we can can specialise it for different regular expressions. Save the file and enter a specialisation query in the **Goal** entry box.

```
match(.(b,*(a)), X)
```

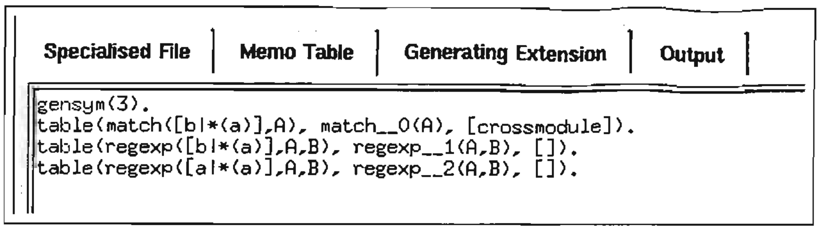
This will specialise the interpreter for matching a string beginning with a b followed by zero or more a's. Click **Specialise** or press return to specialise the program.



```
:- module('regexp.spec',[]).
match([b|*(a)], A) :-
        match__0(A).
match__0(A) :-
        regexp__1(A, []).
regexp__1([b|A], B) :-
        regexp__2(A, B).
regexp__1([b|A], A).
regexp__2([a|A], B) :-
        regexp__2(A, B).
regexp__2([a|A], A).
```

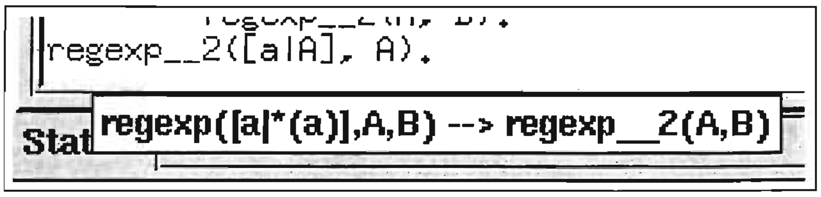
The specialise code contains an entry point `match([b|*(a)], A)` which will call the corresponding specialised predicate. The overhead of interpreting the regular expression has been removed and only the string matcher remains.

The memo table maintains the list of specialised predicates and their original call patterns. It is used internally during specialisation and is saved to a file when specialisation is complete. Selecting the **Memo Table** tab displays the table.



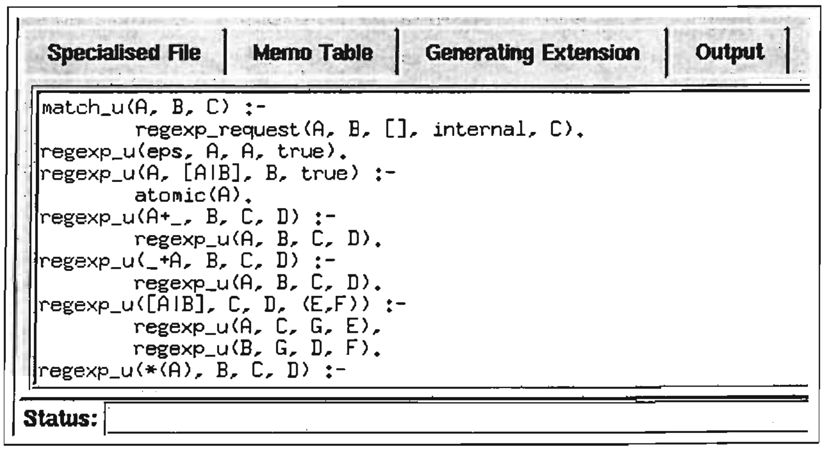
```
gensym(3).
table(match([b|*(a)],A), match__0(A), [crossmodule]).
table(regexp([b|*(a)],A,B), regexp__1(A,B), []).
table(regexp([a|*(a)],A,B), regexp__2(A,B), []).
```

The two entries for regexp/3 correspond to the two specialised versions of regexp/3 generated during specialisation, called regexp__1 and regexp__2. regexp__1 is specialised for a b followed by some a's, and regexp__2 is specialised for an a follow by some more a's. Hovering over a call in the specialised file displays the original mapping from the memo table in a balloon window.



```
regexp__2([a|A], A).
regexp([a|*(a)],A,B) --> regexp__2(A,B)
```

A *cogen* specialiser first creates a generating extension, a specialised specialiser, which is then used to specialise a file for a particular query. Clicking on the **generating extension** tab will display this file. The generating extension only needs to be regenerated if the annotations change, it can be reused for different specialisation queries.



```
match_u(A, B, C) :-
        regexp_request(A, B, [], internal, C).
regexp_u(eps, A, A, true).
regexp_u(A, [A|B], B, true) :-
        atomic(A).
regexp_u(A+_, B, C, D) :-
        regexp_u(A, B, C, D).
regexp_u(_+A, B, C, D) :-
        regexp_u(A, B, C, D).
regexp_u([A|B], C, D, (E,F)) :-
        regexp_u(A, C, G, E),
        regexp_u(B, G, D, F).
regexp_u(*(A), B, C, D) :-
```

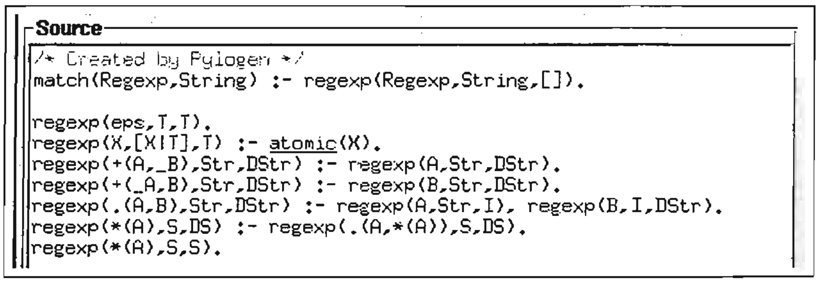
A.3 Using the Automatic Binding-time Analysis

In the last section we annotated the file by hand, manually checking each annotation. Chapter 5 introduces the automatic binding-time analysis (bta). The bta automatically annotates a file with a correct set of annotations. From the **BTA** menu select **unfold all**, this will reset the file, annotating it to perform *all* of the operations at specialisation time. Now add an entry point for the bta, this is done using a filter declaration.

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

The regular expression interpreter manipulates terms as it parses the regular expression. Select **List Norm** from the **BTA** menu. Save the file and then select **Auto bta** from the **BTA** menu.

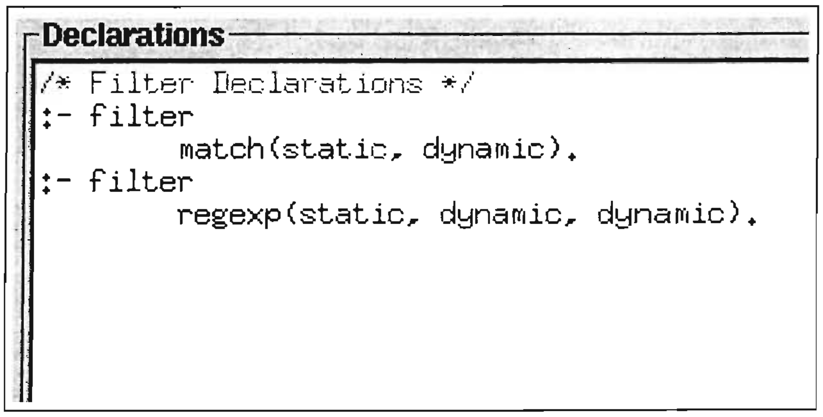
The bta should provide the same annotations we selected manually. Only the recursive call to `regexp/3` handling the * will be marked as **memo**.



```
/* Created by Pylogen */
match(Regexp,String) :- regexp(Regexp,String,[]).

regexp(eps,T,T).
regexp(X,[X|T],T) :- atomic(X).
regexp(+(A,_B),Str,DStr) :- regexp(A,Str,DStr).
regexp(+(_A,B),Str,DStr) :- regexp(B,Str,DStr).
regexp(.(A,B),Str,DStr) :- regexp(A,Str,I), regexp(B,I,DStr).
regexp(*(A),S,DS) :- regexp(.(A,*(A)),S,DS).
regexp(*(A),S,S).
```

The filter declarations should be correctly propagated throughout the program.



```
/* Filter Declarations */
:- filter
        match(static, dynamic).
:- filter
        regexp(static, dynamic, dynamic).
```

Appendix B

Annotated Lix

```
(:-module(lix,[])).
(:-op(1150,fx,type)).
:- dynamic
        memo_table/2.
:- dynamic
        flag/2.
(:-use_module(library(terms))).
logen(print_memo_table , print_memo_table) :-
        logen(rescall , memo_table(A,B)),
        logen(rescall , portray_clause(memo_table(A,B))),
        logen(rescall , fail).
logen(print_memo_table , print_memo_table).
logen(lix_load, lix_load(A,B,C)) :-
        logen(rescall , print('%loading file ')),
        logen(rescall , print(A)),
        logen(rescall , nl),
        logen(unfold , lix(B,C)).
logen(lix, lix(A,B)) :-
        logen(rescall , retractall(memo_table(_,_))),
        logen(rescall , print('/*Generated by Lix*/\n')),
        logen(rescall , print(':- dynamic flag/2, memo_table/2.\n')),
        logen(rescall , print(':- use_module(library(lists)).\n')),
        logen(rescall , print(':- use_module(library(terms)).\n')),
        logen(rescall , print(':- op(1150, fx, type).\n')),
        logen(rescall , print(':- op(1150, fx, filter).\n')),
        logen(rescall , print(':- dynamic ann_clause/3, filter/2.\n')),
        logen(memo, memo(A,B)).
logen(memo, memo(A,B)) :-
        resif(logen(rescall,memo_table(A,B)),
              logen(rescall,true),
              ( logen(unfold,generalise_and_filter(A,C,D)),
                logen(rescall,assert(memo_table(C,D))),
                resfindall((D:-E),
                logen(memo,unfold(C,E)),F),
                logen(rescall,format('/*~k=~k*/~n',[D,C])),
                logen(memo,pretty_print_clauses(F)),
                logen(rescall,memo_table(A,B))
               )).
logen(unfold, unfold(ann_clause(A,B,C),true)) :-
        logen(call, !),
        logen(rescall, ann_clause(A,B,C)).
logen(unfold, unfold(filter(A,B),true)) :-
        logen(call, !),
        logen(rescall, filter(A,B)).
logen(unfold, unfold(A,B)) :-
        logen(unfold, ann_clause(_,A,C)),
        logen(unfold, body(C,B)).
logen(body, body(true,true)).
logen(body, body((A,B),(C,D))) :-
        logen(unfold, body(A,C)),
        logen(unfold, body(B,D)).
logen(body, body(logen(call,A),true)) :-
        logen(rescall, call(A)).
logen(body, body(logen(rescall,A),A)).
logen(body, body(logen(memo,A),B)) :-
        logen(memo, memo(A,B)).
logen(body, body(logen(unfold,A),B)) :-
        logen(memo, unfold(A,B)).
logen(body, body(resif(A,B,C),(D->E;F))) :-
        logen(unfold, body(A,D)),
        logen(unfold, body(B,E)),
        logen(unfold, body(C,F)).
logen(body, body(if(A,B,C),D)) :-
        resif(logen(unfold,body(A,_)),
              (logen(unfold,body(B,E)),
               logen(rescall,E=D)
              ),(logen(unfold,body(C,F)),logen(rescall,F=D)
              )).
logen(body, body(resfindall(A,B,C),findall(A,D,C))) :-
        logen(unfold, body(B,D)).
logen(body, body(hide_nf(A),B)) :-
        logen(rescall, term_variables(A,C)),
        resfindall((D,C), logen(unfold,body(A,D)), E),
        resif(logen(rescall,E=[]),
              logen(rescall,B=fail),
              (logen(memo,make_disj(E,C,F)),
               logen(memo,flatten(F,B))
              )).
logen(make_disj, make_disj([(A,B)],C,D)) :-
        logen(memo, simplify_eq(B,C,E)),
        logen(rescall, D=(E,A)).
logen(make_disj, make_disj([(A,B)|C],D,(E;F))) :-
        logen(memo, make_disj(C,D,F)),
        logen(memo, simplify_eq(B,D,G)),
        logen(rescall, E=(G,A)).
logen(simplify_eq, simplify_eq(A,B,fail)) :-
        logen(rescall, A\=B).
logen(simplify_eq, simplify_eq(A,B,true)) :-
        logen(rescall, A==B).
logen(simplify_eq, simplify_eq(A,B,A=B)) :-
        logen(rescall, var(A)),
        logen(rescall, !).
logen(simplify_eq, simplify_eq(A,B,A=B)) :-
        logen(rescall, var(B)),
        logen(rescall, !).
logen(simplify_eq, simplify_eq(A,B,C)) :-
        logen(rescall, nonvar(A)),
        logen(rescall, nonvar(B)),
        logen(rescall, functor(A,D,E)),
        logen(rescall, functor(B,D,E)),
        logen(rescall , A=..[D|F]),
        logen(rescall , B=..[D|G]),
        logen(memo, simplify_eqL(F,G,C)).
logen(simplify_eqL , simplify_eqL([A],[B],C)) :-
        logen(memo, simplify_eq(A,B,C)).
logen(simplify_eqL , simplify_eqL([A|B],[C|D],(E,F))) :-
        logen(memo, simplify_eq(A,C,E)),
        logen(memo, simplify_eqL(B,D,F)).
logen(generalise_and_filter , generalise_and_filter(A,B,C)) :-
        logen(call, functor(A,D,E)),
        logen(call, functor(B,D,E)),
        logen(unfold, filter(A,F)),
        logen(call, A=..[G|H]),
        logen(unfold, gen_filter(F,H,I,J)),
        logen(call, B=..[G|I]),
        logen(memo, gensym(G,K)),
        logen(rescall , C=..[K|J]).
logen(typedef , typedef(list(A),(struct([],[]);struct('.',[A,(type list(A))])))).
logen(gen_filter, gen_filter([],[],[],[])).
logen(gen_filter, gen_filter([(A;_)|B],C,D,E)) :-
        logen(unfold, gen_filter([A|B],C,D,E)).
logen(gen_filter, gen_filter([(_;A)|B],C,D,E)) :-
        logen(unfold, gen_filter([A|B],C,D,E)).
logen(gen_filter, gen_filter([static|A],[B|C],[B|D],E)) :-
        logen(unfold, gen_filter(A,C,D,E)).
logen(gen_filter, gen_filter([static_nf|A],[B|C],[B|D],[B|E])) :-
        logen(unfold, gen_filter(A,C,D,E)).
logen(gen_filter, gen_filter([dynamic|A],[_|B],[C|D],[C|E])) :-
        logen(unfold, gen_filter(A,B,D,E)).
logen(gen_filter, gen_filter([nonvar|A],[B|C],[D|E],F)) :-
        logen(rescall , B=..[G|H]),
        logen(rescall , length(H,I)),
        logen(rescall , length(J,I)),
        logen(rescall , D=..[G|J]),
        logen(unfold, gen_filter(A,C,E,K)),
        logen(rescall , append(J,K,F)).
logen(gen_filter, gen_filter([(type A)|B],C,D,E)) :-
        logen(unfold, typedef(A,F)),
        logen(memo, gen_filter([F|B],C,D,E)).
logen(gen_filter, gen_filter([struct(A,B)|C],[D|E],[F|G],H)) :-
        logen(rescall , D=..[A|I]),
        logen(unfold, gen_filter(B,I,J,K)),
        logen(rescall , F=..[A|J]),
        logen(unfold, gen_filter(C,E,G,L)),
        logen(rescall , append(K,L,H)).
logen(pretty_print_clauses , pretty_print_clauses([])).
logen(pretty_print_clauses , pretty_print_clauses([A|B])) :-
        logen(memo, flatten(A,C)),
        logen(rescall , portray_clause(C)),
        logen(memo, pretty_print_clauses(B)).
logen(flatten , flatten((A:-B),(A:-C))) :-
        logen(rescall , !),
        logen(memo, flatten(B,C)).
logen(flatten , flatten((A,B),C)) :-
        logen(rescall , !),
        logen(memo, flatten(A,D)),
        logen(memo, flatten(B,E)),
        resif(logen(rescall ,D=true),
              logen(rescall ,C=E),
                resif(logen(rescall,E=true),
                      logen(rescall,C=D),
                      logen(rescall,C=(D,E))
                      )
                ).
logen(flatten, flatten((A;B),C)) :-
        logen(rescall, !),
        logen(memo, flatten(A,D)),
        logen(memo, flatten(B,E)),
        resif(logen(rescall,D=true),
              logen(rescall,C=E),
              resif(logen(rescall,E=true),
                    logen(rescall,C=D),
                    logen(rescall,C=(D;E))
                   )
             ).
logen(flatten, flatten((A->B;C),(D->E;F))) :-
        logen(rescall, !),
        logen(memo, flatten(A,D)),
        logen(memo, flatten(B,E)),
        logen(memo, flatten(C,F)).
logen(flatten, flatten(call(A),A)) :-
        logen(rescall, nonvar(A)),
        logen(rescall, !).
logen(flatten, flatten(A,A)).
logen(gensym, gensym(A,B)) :-
        logen(rescall, var(B)),
        logen(call, atom(A)),
        logen(memo, oldvalue(A,C)),
        logen(rescall, D is C+1),
        logen(memo, set_flag(gensym(A),D)),
        logen(call, name(E,[95,95])),
        logen(unfold, string_concat(A,E,F)),
        logen(unfold, string_concat(F,D,B)).
logen(oldvalue, oldvalue(A,B)) :-
        logen(rescall, flag(gensym(A),B)),
        logen(rescall, !).
logen(oldvalue, oldvalue(_,0)).
logen(set_flag, set_flag(A,B)) :-
        logen(call, nonvar(A)),
        logen(rescall, retract(flag(A,_))),
        logen(rescall, !),
        logen(rescall, asserta(flag(A,B))).
logen(set_flag, set_flag(A,B)) :-
        logen(call, nonvar(A)),
        logen(rescall, asserta(flag(A,B))).
logen(append, append([],A,A)).
logen(append, append([A|B],C,[A|D])) :-
        logen(unfold, append(B,C,D)).
logen(string_concat, string_concat(A,B,C)) :-
        logen(call, name(A,D)),
        if(logen(call,var(B)), logen(rescall,name(B,E)), logen(call,name(B,E))),
        logen(unfold, append(D,E,F)),
        if(logen(call,var(B)), logen(rescall,name(C,F)), logen(call,name(C,F))).
logen(filter, filter(app(_,_,_),[dynamic,static,dynamic])).
logen(ann_clause, ann_clause(1,app([],A,A),true)).
logen(ann_clause, ann_clause(2,app([A|B],C,[A|D]),logen(memo,app(B,C,D)))).
logen(filter, filter(test(_),[dynamic])).
logen(ann_clause, ann_clause(5,test(A),hide_nf(logen(unfold,p(A))))).
logen(ann_clause, ann_clause(3,p(a),true)).
logen(ann_clause, ann_clause(4,p(b),true)).
logen(ann_clause, ann_clause(0,l_eval([],_,[]),true)).
logen(ann_clause, ann_clause(1,
                  l_eval([A|B],C,[D|E]),
                  (logen(unfold,eval(A,C,D)),logen(unfold,l_eval(B,C,E))))).
logen(ann_clause, ann_clause(2,eval(cst(A),_,constr(A,[])),true)).
logen(ann_clause, ann_clause(3,
                  eval(constr(A,B),C,constr(A,D)),
                  logen(unfold,l_eval(B,C,D)))).
logen(ann_clause, ann_clause(4,eval(var(A),B,C),logen(unfold,lookup(A,B,C)))).
logen(ann_clause, ann_clause(5,
                  eval(plus(A,B),C,constr(D,[])),
                  (logen(unfold,eval(A,C,constr(E,[]))),
                   logen(unfold,eval(B,C,constr(F,[]))),
                   logen(rescall,D is E+F)))).
logen(ann_clause, ann_clause(6,
                  eval(minus(A,B),C,constr(D,[])),
                  (logen(unfold,eval(A,C,constr(E,[]))),
                   logen(unfold,eval(B,C,constr(F,[]))),
                   logen(rescall,D is E-F)))).
logen(ann_clause, ann_clause(7,
                  eval(times(A,B),C,constr(D,[])),
                  (logen(unfold,eval(A,C,constr(E,[]))),
                   logen(unfold,eval(B,C,constr(F,[]))),
                   logen(rescall,D is E*F)))).
logen(ann_clause, ann_clause(8,
                  eval(eq(A,B),C,constr(D,[])),
                  (logen(unfold,eval(A,C,E)),
                   logen(unfold,eval(B,C,F)),
                   resif(logen(rescall,E=F),
                         logen(rescall,D=true),
                         logen(rescall,D=false))
                        ))).
logen(ann_clause, ann_clause(9,
                  eval(let(A,B,C),D,E),
                  (logen(unfold,eval(B,D,F)),
                   logen(unfold,store(D,A,F,G)),
                   logen(unfold,eval(C,G,E))))).
logen(ann_clause, ann_clause(10,
                  eval(if(A,B,C),D,E),
                  logen(unfold,eval_if(A,B,C,D,E)))).
logen(ann_clause, ann_clause(11,
                  eval(if2(A,B,C),D,E),
                  (logen(unfold,eval(A,D,F)),
                   resif(logen(rescall,F=constr(true,[])),
                         hide_nf(logen(unfold,eval(B,D,E))),
                         hide_nf(logen(unfold,eval(C,D,E))))))).
logen(ann_clause, ann_clause(12,eval(lambda(A,B),_,lambda(A,B)),true)).
logen(ann_clause, ann_clause(13,
                  eval(apply(A,B),C,D),
                  (logen(unfold,eval(B,C,E)),
                   logen(unfold,rename(E,C,lambda(F,G))),
                   logen(unfold,eval(A,C,H)),
                   logen(unfold,store(C,F,H,I)),
                   logen(memo,eval(G,I,D))))).
logen(ann_clause, ann_clause(14,
                   eval(fun(A),_,B),
                   logen(unfold,function(A,B)))).
logen(ann_clause, ann_clause(15,
                  eval(print(A),_,constr(true,[])),
                  (logen(rescall,print(A)),
                   logen(rescall,nl)))).
logen(ann_clause, ann_clause(16,
                  eval_if(A,B,_,C,D),
                  (logen(unfold,test(A,C)),
                   logen(rescall,!),
                   logen(unfold,eval(B,C,D))))).
logen(ann_clause, ann_clause(17,eval_if(_,_,A,B,C),logen(unfold,eval(A,B,C)))).
logen(ann_clause, ann_clause(18,
                  test(eq(A,B),C),
                  (logen(unfold,eval(A,C,D)),
                   logen(unfold,eval(B,C,D))))).
logen(ann_clause, ann_clause(19,rename(A,_,B),logen(call,B=A))).
logen(ann_clause, ann_clause(20,
                  function(fib,
                           lambda(x,
                            if(eq(var(x),cst(0)),cst(1),
                             if(eq(var(x),cst(1)),cst(1),
                               plus(apply(minus(var(x),cst(1)),fun(fib)),
                                    apply(minus(var(x),cst(2)),fun(fib)))))),
                            true)).
logen(ann_clause, ann_clause(21,store([],A,B,[A/B]),true)).
logen(ann_clause, ann_clause(22,store([A/_|B],A,C,[A/C|B]),true)).
logen(ann_clause, ann_clause(23,
                  store([A/B|C],D,E,[A/B|F]),
                  (logen(call,D\==A),
                   logen(unfold,store(C,D,E,F))))).
logen(ann_clause, ann_clause(24,lookup(A,[A/B|_],B),true)).
logen(ann_clause, ann_clause(25,
                   lookup(A,[B/_|C],D),
                   (logen(rescall,A\==B),
                    logen(unfold,lookup(A,C,D))))).
logen(ann_clause, ann_clause(26,
                    fib(A,B),
                    (logen(unfold,store([],x,A,C)),
                     logen(unfold,eval(apply(cst(A),fun(fib)),C,constr(B,_)))))).
logen(ann_clause, ann_clause(27,
                  bench(A,B),
                  (logen(rescall,A>B),
                   logen(rescall,print('Done')),
                   logen(rescall,nl)))).
logen(ann_clause, ann_clause(28,
                  bench(A,B),
                  (logen(rescall,A=<B),
                   logen(unfold,fib(A,C)),
                   logen(rescall,!),
                   logen(rescall,print(fib(A))),
                   logen(rescall,print(' == ')),
                   logen(rescall,print(C)),
                   logen(rescall,nl),
                   logen(rescall,D is A+1),
                   logen(memo,bench(D,B))))).
logen(filter, filter(
              l_eval(_,_,_),
              [static,(type list(struct(/,[static,dynamic]))),dynamic])).
logen(filter, filter(
              eval(_,_,_),
                [static,(type list(struct(/,[static,dynamic]))),dynamic])).
logen(filter, filter(rename(_,_,_),[dynamic,dynamic,dynamic])).
logen(filter, filter(function(_,_),[dynamic,dynamic])).
logen(filter, filter(store(_,_,_,_),[dynamic,static,static,dynamic])).
logen(filter, filter(lookup(_,_,_),[struct(static,[]),dynamic,dynamic])).
logen(filter, filter(fib(_,_),[dynamic,dynamic])).
logen(filter, filter(bench(_,_),[dynamic,dynamic])).
logen(filter, filter(bench2(_,_),[dynamic,dynamic])).
logen(filter, filter(
                eval_if(_,_,_,_,_),
                [static,static,static,(type list(struct(/,[static,dynamic]))),dynamic])).
:- filter
        lix_load(static, nonvar, dynamic).
:- filter
        lix(nonvar, dynamic).
:- filter
        memo(nonvar, dynamic).
:- filter
        unfold(nonvar, dynamic).
:- filter
        generalise_and_filter(nonvar, dynamic, dynamic).
:- filter
        pretty_print_clauses(dynamic).
:- filter
        flatten(dynamic, dynamic).
:- filter
        gensym(static, dynamic).
:- filter
        oldvalue(dynamic, dynamic).
:- filter
        set_flag(nonvar, dynamic).
:- filter
        make_disj(dynamic, dynamic, dynamic).
:- filter
        simplify_eq(dynamic, dynamic, dynamic).
:- filter
        simplify_eqL(dynamic, dynamic, dynamic).
:- filter
        gen_filter(static, dynamic, dynamic, dynamic).
```

Appendix C

Lix-Cogen

```
/*Generated by Lix*/

:- dynamic flag/2, memo_table/2.
:- use_module(library(lists)).
:- use_module(library(terms)).
:- op(1150, fx, type).
:- dynamic filter/2.
:- dynamic ann_clause/3.


lix(Call, R1,R2) :- lix__2(Call,R1,R2).



/*oldvalue__2(_6626,_6627)=oldvalue(_6626,_6627)*/
oldvalue__2(A, B) :-
        flag(gensym(A), B), !.
oldvalue__2(_, 0).
/*set_flag__2(_8423,_8313)=set_flag(gensym(_8423),_8313)*/
set_flag__2(A, B) :-
        retract(flag(gensym(A),_)), !,
        asserta(flag(gensym(A),B)).
set_flag__2(A, B) :-
        asserta(flag(gensym(A),B)).
/*gensym__13(_5350)=gensym(lix,_5350)*/
gensym__13(A) :-
        var(A),
        oldvalue__2(lix, B),
        C is B+1,
        set_flag__2(lix, C),
        name(C, D),
        name(A, [108,105,120,95,95|D]).
/*gensym__14(_13281)=gensym(memo,_13281)*/
gensym__14(A) :-
        var(A),
        oldvalue__2(memo, B),
        C is B+1,
        set_flag__2(memo, C),
        name(C, D),
        name(A, [109,101,109,111,95,95|D]).
/*unfold__23(_19939,_19941,_19827)=unfold(filter(_19939,_19941),_19827)*/
```

```
unfold__23(A, B, true) :-
        filter(A, B).
unfold__23(app(_,_,_), [dynamic,static,dynamic], true).
unfold__23(test(_), [dynamic], true).
unfold__23(l_eval(_,_,_), [static,(type list(struct(/,[static,dynamic]))),dynamic], true).
unfold__23(eval(_,_,_), [static,(type list(struct(/,[static,dynamic]))),dynamic], true).
unfold__23(rename(_,_,_), [dynamic,dynamic,dynamic], true).
unfold__23(function(_,_), [dynamic,dynamic], true).
unfold__23(store(_,_,_,_), [dynamic,static,static,dynamic], true).
unfold__23(lookup(_,_,_), [struct(static,[]),dynamic,dynamic], true).
unfold__23(fib(_,_), [dynamic,dynamic], true).
unfold__23(bench(_,_), [dynamic,dynamic], true).
unfold__23(bench2(_,_), [dynamic,dynamic], true).
unfold__23(eval_if(_,_,_,_,_),
          [static,static,static,(type list(struct(/,[static,dynamic]))),dynamic], true).
unfold__23(lix_load(_,_,_), [static,nonvar,dynamic], true).
unfold__23(lix(_,_), [nonvar,dynamic], true).
unfold__23(memo(_,_), [nonvar,dynamic], true).
unfold__23(unfold(_,_), [nonvar,dynamic], true).
unfold__23(generalise_and_filter(_,_,_), [nonvar,dynamic,dynamic], true).
unfold__23(pretty_print_clauses(_), [dynamic], true).
unfold__23(flatten(_,_), [dynamic,dynamic], true).
unfold__23(gensym(_,_), [static,dynamic], true).
unfold__23(oldvalue(_,_), [dynamic,dynamic], true).
unfold__23(set_flag(_,_), [nonvar,dynamic], true).
unfold__23(make_disj(_,_,_), [dynamic,dynamic,dynamic], true).
unfold__23(simplify_eq(_,_,_), [dynamic,dynamic,dynamic], true).
unfold__23(simplify_eqL(_,_,_), [dynamic,dynamic,dynamic], true).
unfold__23(gen_filter(_,_,_,_), [static,dynamic,dynamic,dynamic], true).
/*unfold__25(_28360,_28362,_28248)=unfold(typedef(_28360,_28362),_28248)*/
unfold__25(list(A), (struct([],[]);struct('.',[A,(type list(A))])), true).
/*gensym__15(_32177)=gensym(gen_filter,_32177)*/
gensym__15(A) :-
        var(A),
        oldvalue__2(gen_filter, B),
        C is B+1,
        set_flag__2(gen_filter, C),
        name(C, D),
        name(A, [103,101,110,95,102,105,108,116,101,114,95,95|D]).
/*flatten__2(_36773,_36774)=flatten(_36773,_36774)*/
flatten__2((A:-B), (A:-C)) :- !,
        flatten__2(B, C).
flatten__2((A,B), C) :- !,
        flatten__2(A, D),
        flatten__2(B, E),
        (   D=true ->
            C=E
        ;   E=true ->
            C=D
        ;   C=(D,E)
        ).
flatten__2((A;B), C) :- !,
        flatten__2(A, D),
        flatten__2(B, E),
        (   D=true ->
            C=E
        ;   E=true ->
            C=D
        ;   C=(D;E)
        ).
flatten__2((A->B;C), (D->E;F)) :- !,
        flatten__2(A, D),
        flatten__2(B, E),
        flatten__2(C, F).
flatten__2(A, A).
/*pretty_print_clauses__2(_35068)=pretty_print_clauses(_35068)*/
pretty_print_clauses__2([]).
pretty_print_clauses__2([A|B]) :-
        flatten__2(A, C),
        portray_clause(C),
        pretty_print_clauses__2(B).
/*memo__15(_30031,_30033,_30035,_30037,_29915)=
  memo(gen_filter(_30031,_30033,_30035,_30037),_29915)*/
memo__15(A, B, C, D, E) :-
        (   memo_table(gen_filter(A,B,C,D), E) ->
            true
        ;   gensym__15(F),
            G=..[F,H,I,J],
            assert(memo_table(gen_filter(A,H,I,J),G)),
            findall((G:-K), unfold__24(A,H,I,J,K), L),
            format('/*~k=~k*/~n', [G,gen_filter(A,H,I,J)]),
            pretty_print_clauses__2(L),
            memo_table(gen_filter(A,B,C,D), E)
        ).
/*unfold__24(_26796,_26798,_26800,_26802,_26680)=
  unfold(gen_filter(_26796,_26798,_26800,_26802),_26680)*/
unfold__24([], [], [], [], true).
unfold__24([(A;_)|B], C, D, E, F) :-
        unfold__24([A|B], C, D, E, F).
unfold__24([(_;A)|B], C, D, E, F) :-
        unfold__24([A|B], C, D, E, F).
unfold__24([static|A], [B|C], [B|D], E, F) :-
        unfold__24(A, C, D, E, F).
unfold__24([dynamic|A], [_|B], [C|D], [C|E], F) :-
        unfold__24(A, B, D, E, F).
unfold__24([nonvar|A], [B|C], [D|E], F,
            (B=..[G|H],length(H,I),length(J,I),D=..[G|J],K,append(J,L,F))) :-
        unfold__24(A, C, E, L, K).
unfold__24([(type A)|B], C, D, E, (F,G)) :-
        unfold__25(A, H, F),
        memo__15([H|B], C, D, E, G).
unfold__24([struct(A,B)|C], [D|E], [F|G], H, (D=..[A|I],J,F=..[A|K],L,append(M,N,H))) :-
        unfold__24(B, I, K, M, J),
        unfold__24(C, E, G, N, L).
/*gensym__16(_33084)=gensym(gensym,_33084)*/
gensym__16(A) :-
        var(A),
        oldvalue__2(gensym, B),
        C is B+1,
        set_flag__2(gensym, C),
        name(C, D),
        name(A, [103,101,110,115,121,109,95,95|D]).
/*gensym__17(_39756)=gensym(oldvalue,_39756)*/
gensym__17(A) :-
        var(A),
        oldvalue__2(oldvalue, B),
        C is B+1,
        set_flag__2(oldvalue, C),
        name(C, D),
        name(A, [111,108,100,118,97,108,117,101,95,95|D]).
/*unfold__27(_42554,_42556,_42442)=unfold(oldvalue(_42554,_42556),_42442)*/
unfold__27(A, B, (flag(gensym(A),B),!)).
unfold__27(_, 0, true).
/*memo__17(_37834,_37836,_37722)=memo(oldvalue(_37834,_37836),_37722)*/
memo__17(A, B, C) :-
        (   memo_table(oldvalue(A,B), C) ->
            true
        ;   gensym__17(D),
            E=..[D,F,G],
            assert(memo_table(oldvalue(F,G),E)),
            findall((E:-H), unfold__27(F,G,H), I),
            format('/*~k=~k*/~n', [E,oldvalue(F,G)]),
            pretty_print_clauses__2(I),
            memo_table(oldvalue(A,B), C)
        ).
/*gensym__18(_43026)=gensym(set_flag,_43026)*/
gensym__18(A) :-
        var(A),
        oldvalue__2(set_flag, B),
        C is B+1,
        set_flag__2(set_flag, C),
        name(C, D),
        name(A, [115,101,116,95,102,108,97,103,95,95|D]).
/*unfold__28(_45824,_45826,_45712)=unfold(set_flag(_45824,_45826),_45712)*/
unfold__28(A, B, (true,retract(flag(A,_)),!,asserta(flag(A,B)))) :-
        call(nonvar(A)).
unfold__28(A, B, (true,asserta(flag(A,B)))) :-
        call(nonvar(A)).
/*memo__18(_40836,_40838,_40724)=memo(set_flag(_40836,_40838),_40724)*/
memo__18(A, B, C) :-
        (   memo_table(set_flag(A,B), C) ->
            true
        ;   A=..[D|E],
            length(E, F),
            length(G, F),
            H=..[D|G],
            append(G, [I], J),
            gensym__18(K),
            L=..[K|J],
            assert(memo_table(set_flag(H,I),L)),
            findall((L:-M), unfold__28(H,I,M), N),
            format('/*~k=~k*/~n', [L,set_flag(H,I)]),
            pretty_print_clauses__2(N),
            memo_table(set_flag(A,B), C)
        ).
/*unfold__30(_46692,_46694,_46696,_46578)=
  unfold(append(_46692,_46694,_46696),_46578)*/
unfold__30([], A, A, true).
unfold__30([A|B], C, [A|D], E) :-
        unfold__30(B, C, D, E).
/*unfold__29(_44314,_44316,_44318,_44200)=
  unfold(string_concat(_44314,_44316,_44318),_44200)*/
unfold__29(A, B, C, (true,D,E,F)) :-
        call(name(A,G)),
        (   call(var(B)) ->
            true,
            name(B,H)=D
        ;   call(name(B,H)),
            true=D
        ),
        unfold__30(G, H, I, E),
        (   call(var(B)) ->
            true,
            name(C,I)=F
        ;   call(name(C,I)),
            true=F
        ).
/*unfold__26(_35878,_35880,_35766)=unfold(gensym(_35878,_35880),_35766)*/
unfold__26(A, B, (var(B),true,C,D is E+1,F,true,G,H)) :-
        call(atom(A)),
        memo__17(A, E, C),
        memo__18(gensym(A), D, F),
        call(name(I,[95,95])),
        unfold__29(A, I, J, G),
        unfold__29(J, D, B, H).
/*memo__16(_31164,_31166,_31052)=memo(gensym(_31164,_31166),_31052)*/
memo__16(A, B, C) :-
        (   memo_table(gensym(A,B), C) ->
            true
        ;   gensym__16(D),
            E=..[D,F],
            assert(memo_table(gensym(A,F),E)),
            findall((E:-G), unfold__26(A,F,G), H),
            format('/*~k=~k*/~n', [E,gensym(A,F)]),
            pretty_print_clauses__2(H),
            memo_table(gensym(A,B), C)
        ).
/*unfold__22(_17946,_17948,_17950,_17832)=
  unfold(generalise_and_filter(_17946,_17948,_17950),_17832)*/
unfold__22(A, B, C, (true,true,D,true,E,true,F,C=..[G|H])) :-
        call(functor(A,I,J)),
        call(functor(B,I,J)),
        unfold__23(A, K, D),
        call(A=..[L|M]),
        unfold__24(K, M, N, H, E),
        call(B=..[L|N]),
        memo__16(L, G, F).
/*gensym__19(_22885)=gensym(unfold,_22885)*/
gensym__19(A) :-
        var(A),
        oldvalue__2(unfold, B),
        C is B+1,
        set_flag__2(unfold, C),
        name(C, D),
        name(A, [117,110,102,111,108,100,95,95|D]).
/*unfold__32(_27188,_27190,_27192,_27074)=
  unfold(ann_clause(_27188,_27190,_27192),_27074)*/
unfold__32(A, B, C, true) :-
        ann_clause(A, B, C).
unfold__32(1, app([],A,A), true, true).
unfold__32(2, app([A|B],C,[A|D]), logen(memo,app(B,C,D)), true).
unfold__32(5, test(A), hide_nf(logen(unfold,p(A))), true).
unfold__32(3, p(a), true, true).
unfold__32(4, p(b), true, true).
unfold__32(0, l_eval([],_,[]), true, true).
unfold__32(1, l_eval([A|B],C,[D|E]),
               (logen(unfold,eval(A,C,D)),logen(unfold,l_eval(B,C,E))), true).
unfold__32(2, eval(cst(A),_,constr(A,[])), true, true).
unfold__32(3, eval(constr(A,B),C,constr(A,D)), logen(unfold,l_eval(B,C,D)), true).
unfold__32(4, eval(var(A),B,C), logen(unfold,lookup(A,B,C)), true).
unfold__32(5, eval(plus(A,B),C,constr(D,[])),
              (logen(unfold,eval(A,C,constr(E,[]))),
               logen(unfold,eval(B,C,constr(F,[]))),
               logen(rescall,D is E+F)), true).
unfold__32(6, eval(minus(A,B),C,constr(D,[])),
              (logen(unfold,eval(A,C,constr(E,[]))),
               logen(unfold,eval(B,C,constr(F,[]))),
               logen(rescall,D is E-F)), true).
unfold__32(7, eval(times(A,B),C,constr(D,[])),
              (logen(unfold,eval(A,C,constr(E,[]))),
               logen(unfold,eval(B,C,constr(F,[]))),
               logen(rescall,D is E*F)), true).
unfold__32(8, eval(eq(A,B),C,constr(D,[])),
              (logen(unfold,eval(A,C,E)),
               logen(unfold,eval(B,C,F)),
               resif(logen(rescall,E=F),
                     logen(rescall,D=true),
                     logen(rescall,D=false))), true).
unfold__32(9, eval(let(A,B,C),D,E),
              (logen(unfold,eval(B,D,F)),
               logen(unfold,store(D,A,F,G)),
               logen(unfold,eval(C,G,E))), true).
unfold__32(10, eval(if(A,B,C),D,E),
               logen(unfold,eval_if(A,B,C,D,E)), true).
unfold__32(11, eval(if2(A,B,C),D,E),
               (logen(unfold,eval(A,D,F)),
                resif(logen(rescall,F=constr(true,[])),
                hide_nf(logen(unfold,eval(B,D,E))),
                hide_nf(logen(unfold,eval(C,D,E))))), true).
unfold__32(12, eval(lambda(A,B),_,lambda(A,B)), true, true).
unfold__32(13, eval(apply(A,B),C,D),
               (logen(unfold,eval(B,C,E)),
                logen(unfold,rename(E,C,lambda(F,G))),
                logen(unfold,eval(A,C,H)),
                logen(unfold,store(C,F,H,I)),logen(memo,eval(G,I,D))), true).
unfold__32(14, eval(fun(A),_,B), logen(unfold,function(A,B)), true).
unfold__32(15, eval(print(A),_,constr(true,[])),
               (logen(rescall,print(A)),logen(rescall,nl)), true).
unfold__32(16, eval_if(A,B,_,C,D),
               (logen(unfold,test(A,C)),
                logen(rescall,!),
                logen(unfold,eval(B,C,D))), true).
unfold__32(17, eval_if(_,_,A,B,C), logen(unfold,eval(A,B,C)), true).
unfold__32(18, test(eq(A,B),C),
               (logen(unfold,eval(A,C,D)),logen(unfold,eval(B,C,D))), true).
unfold__32(19, rename(A,_,B), logen(call,B=A), true).
unfold__32(20, function(fib,
                  lambda(x,
                   if(eq(var(x),cst(0)),cst(1),
                    if(eq(var(x),cst(1)),cst(1),
                     plus(apply(minus(var(x),cst(1)),fun(fib)),
                         apply(minus(var(x),cst(2)),fun(fib))))))),
               true, true).
unfold__32(21, store([],A,B,[A/B]), true, true).
unfold__32(22, store([A/_|B],A,C,[A/C|B]), true, true).
```

```
unfold__32(23, store([A/B|C],D,E,[A/B|F]),
               (logen(call,D\==A),logen(unfold,store(C,D,E,F))), true).
unfold__32(24, lookup(A,[A/B|_],B), true, true).
unfold__32(25, lookup(A,[B/_|C],D),
               (logen(rescall,A\==B),logen(unfold,lookup(A,C,D))), true).
unfold__32(26, fib(A,B),
               (logen(unfold,store([],x,A,C)),
                logen(unfold,eval(apply(cst(A),fun(fib)),C,constr(B,_)))), true).
unfold__32(27, bench(A,B),
               (logen(rescall,A>B),logen(rescall,print('Done')),logen(rescall,nl)), true).
unfold__32(28, bench(A,B),
               (logen(rescall,A=<B),
                logen(unfold,fib(A,C)),
                logen(rescall,!),
                logen(rescall,print(fib(A))),
                logen(rescall,print(' == ')),
                logen(rescall,print(C)),
                logen(rescall,nl),
                logen(rescall,D is A+1),
                logen(memo,bench(D,B))), true).
/*gensym__20(_40266)=gensym(make_disj,_40266)*/
gensym__20(A) :-
        var(A),
        oldvalue__2(make_disj, B),
        C is B+1,
        set_flag__2(make_disj, C),
        name(C, D),
        name(A, [109,97,107,101,95,100,105,115,106,95,95|D]).
/*gensym__21(_46645)=gensym(simplify_eq,_46645)*/
gensym__21(A) :-
        var(A),
        oldvalue__2(simplify_eq, B),
        C is B+1,
        set_flag__2(simplify_eq, C),
        name(C, D),
        name(A, [115,105,109,112,108,105,102,121,95,101,113,95,95|D]).
/*gensym__22(_54085)=gensym(simplify_eqL,_54085)*/
gensym__22(A) :-
        var(A),
        oldvalue__2(simplify_eqL, B),
        C is B+1,
        set_flag__2(simplify_eqL, C),
        name(C, D),
        name(A, [115,105,109,112,108,105,102,121,95,101,113,76,95,95|D]).
/*unfold__36(_56893,_56895,_56897,_56779)=
  unfold(simplify_eqL(_56893,_56895,_56897),_56779)*/
unfold__36([A], [B], C, D) :-
        memo__21(A, B, C, D).
unfold__36([A|B], [C|D], (E,F), (G,H)) :-
        memo__21(A, C, E, G),
        memo__22(B, D, F, H).
/*memo__22(_52050,_52052,_52054,_51936)=
  memo(simplify_eqL(_52050,_52052,_52054),_51936)*/
memo__22(A, B, C, D) :-
        (   memo_table(simplify_eqL(A,B,C), D) ->
            true
        ;   gensym__22(E),
            F=..[E,G,H,I],
            assert(memo_table(simplify_eqL(G,H,I),F)),
            findall((F:-J), unfold__36(G,H,I,J), K),
            format('/*~k=~k*/~n', [F,simplify_eqL(G,H,I)]),
            pretty_print_clauses__2(K),
            memo_table(simplify_eqL(A,B,C), D)
        ).
/*unfold__35(_49451,_49453,_49455,_49337)=
  unfold(simplify_eq(_49451,_49453,_49455),_49337)*/
unfold__35(A, B, fail, A\=B).
unfold__35(A, B, true, A==B).
unfold__35(A, B, A=B, (var(A),!)).
unfold__35(A, B, A=B, (var(B),!)).
unfold__35(A, B, C,
          (nonvar(A),nonvar(B),functor(A,D,E),functor(B,D,E),
           A=..[D|F],B=..[D|G],H)) :-
        memo__22(F, G, C, H).
/*memo__21(_44610,_44612,_44614,_44496)=
  memo(simplify_eq(_44610,_44612,_44614),_44496)*/
memo__21(A, B, C, D) :-
        (   memo_table(simplify_eq(A,B,C), D) ->
            true
        ;   gensym__21(E),
            F=..[E,G,H,I],
            assert(memo_table(simplify_eq(G,H,I),F)),
            findall((F:-J), unfold__35(G,H,I,J), K),
            format('/*~k=~k*/~n', [F,simplify_eq(G,H,I)]),
            pretty_print_clauses__2(K),
            memo_table(simplify_eq(A,B,C), D)
        ).
/*unfold__34(_43068,_43070,_43072,_42954)=
  unfold(make_disj(_43068,_43070,_43072),_42954)*/
unfold__34([(A,B)], C, D, (E,D=(F,A))) :-
        memo__21(B, C, F, E).
unfold__34([(A,B)|C], D, (E;F), (G,H,E=(I,A))) :-
        memo__20(C, D, F, G),
        memo__21(B, D, I, H).
/*memo__20(_38231,_38233,_38235,_38117)=
  memo(make_disj(_38231,_38233,_38235),_38117)*/
memo__20(A, B, C, D) :-
        (   memo_table(make_disj(A,B,C), D) ->
            true
        ;   gensym__20(E),
            F=..[E,G,H,I],
            assert(memo_table(make_disj(G,H,I),F)),
            findall((F:-J), unfold__34(G,H,I,J), K),
            format('/*~k=~k*/~n', [F,make_disj(G,H,I)]),
            pretty_print_clauses__2(K),
            memo_table(make_disj(A,B,C), D)
        ).
/*gensym__23(_42889)=gensym(flatten,_42889)*/
gensym__23(A) :-
        var(A),
        oldvalue__2(flatten, B),
        C is B+1,
        set_flag__2(flatten, C),
        name(C, D),
        name(A, [102,108,97,116,116,101,110,95,95|D]).
/*unfold__37(_45685,_45687,_45573)=unfold(flatten(_45685,_45687),_45573)*/
unfold__37((A:-B), (A:-C), (!,D)) :-
        memo__23(B, C, D).
```

```
unfold__37((A,B), C, (!,D,E,(F=true->C=G;G=true->C=F;C=(F,G)))) :-
        memo__23(A, F, D),
        memo__23(B, G, E).
unfold__37((A;B), C, (!,D,E,(F=true->C=G;G=true->C=F;C=(F;G)))) :-
        memo__23(A, F, D),
        memo__23(B, G, E).
unfold__37((A->B;C), (D->E;F), (!,G,H,I)) :-
        memo__23(A, D, G),
        memo__23(B, E, H),
        memo__23(C, F, I).
unfold__37(A, A, true).
/*memo__23(_40967,_40969,_40855)=memo(flatten(_40967,_40969),_40855)*/
memo__23(A, B, C) :-
        (   memo_table(flatten(A,B), C) ->
            true
        ;   gensym__23(D),
            E=..[D,F,G],
            assert(memo_table(flatten(F,G),E)),
            findall((E:-H), unfold__37(F,G,H), I),
            format('/*~k=~k*/~n', [E,flatten(F,G)]),
            pretty_print_clauses__2(I),
            memo_table(flatten(A,B), C)
        ).
/*unfold__33(_35799,_35801,_35687)=unfold(body(_35799,_35801),_35687)*/
unfold__33(true, true, true).
unfold__33((A,B), (C,D), (E,F)) :-
        unfold__33(A, C, E),
        unfold__33(B, D, F).
unfold__33(logen(call,A), true, call(A)).
unfold__33(logen(rescall,A), A, true).
unfold__33(logen(memo,A), B, C) :-
        memo__14(A, B, C).
unfold__33(logen(unfold,A), B, C) :-
        memo__19(A, B, C).
unfold__33(resif(A,B,C), (D->E;F), (G,H,I)) :-
        unfold__33(A, D, G),
        unfold__33(B, E, H),
        unfold__33(C, F, I).
unfold__33(if(A,B,C), D, (E->F,G=D;H,I=D)) :-
        unfold__33(A, _, E),
        unfold__33(B, G, F),
        unfold__33(C, I, H).
unfold__33(resfindall(A,B,C), findall(A,D,C), E) :-
        unfold__33(B, D, E).
unfold__33(hide_nf(A), B,
           (term_variables(A,C),findall((D,C),E,F),(F=[]->B=fail;G,H))) :-
        unfold__33(A, D, E),
        memo__20(F, C, I, G),
        memo__23(I, B, H).
/*unfold__31(_25679,_25681,_25567)=unfold(unfold(_25679,_25681),_25567)*/
unfold__31(ann_clause(A,B,C), true, (true,ann_clause(A,B,C))) :-
        call(!).
unfold__31(filter(A,B), true, (true,filter(A,B))) :-
        call(!).
unfold__31(A, B, (C,D)) :-
        unfold__32(_, A, E, C),
        unfold__33(E, B, D).
/*memo__19(_20695,_20697,_20583)=memo(unfold(_20695,_20697),_20583)*/
memo__19(A, B, C) :-
```

```
        (   memo_table(unfold(A,B), C) ->
            true
        ;   A=..[D|E],
            length(E, F),
            length(G, F),
            H=..[D|G],
            append(G, [I], J),
            gensym__19(K),
            L=..[K|J],
            assert(memo_table(unfold(H,I),L)),
            findall((L:-M), unfold__31(H,I,M), N),
            format('/*~k=~k*/~n', [L,unfold(H,I)]),
            pretty_print_clauses__2(N),
            memo_table(unfold(A,B), C)
        ).
/*gensym__24(_25967)=gensym(pretty_print_clauses,_25967)*/
gensym__24(A) :-
        var(A),
        oldvalue__2(pretty_print_clauses , B),
        C is B+1,
        set_flag__2(pretty_print_clauses , C),
        name(C, D),
        name(A, [112,114,101,116,116,121,95,112,114,105,
                 110,116,95,99,108,97,117,115,101,115,95,95|D]).
/*unfold__38(_28787,_28677)=unfold(pretty_print_clauses(_28787),_28677)*/
unfold__38([], true).
unfold__38([A|B], (C,portray_clause(D),E)) :-
        memo__23(A, D, C),
        memo__24(B, E).
/*memo__24(_24158,_24048)=memo(pretty_print_clauses(_24158),_24048)*/
memo__24(A, B) :-
        (   memo_table(pretty_print_clauses(A), B) ->
            true
        ;   gensym__24(C),
            D=..[C,E],
            assert(memo_table(pretty_print_clauses(E),D)),
            findall((D:-F), unfold__38(E,F), G),
            format('/*~k=~k*/~n', [D,pretty_print_clauses(E)]),
            pretty_print_clauses__2(G),
            memo_table(pretty_print_clauses(A), B)
        ).
/*unfold__21(_16071,_16073,_15959)=unfold(memo(_16071,_16073),_15959)*/
unfold__21(A, B,
           (memo_table(A,B)->true
            ;
            C,assert(memo_table(D,E)),
            findall((E:-F),G,H),
            format('/*~k=~k*/~n',[E,D]),I,memo_table(A,B))) :-
        unfold__22(A, D, E, C),
        memo__19(D, F, G),
        memo__24(H, I).
/*memo__14(_11091,_11093,_10979)=memo(memo(_11091,_11093),_10979)*/
memo__14(A, B, C) :-
        (   memo_table(memo(A,B), C) ->
            true
        ;   A=..[D|E],
            length(E, F),
            length(G, F),
            H=..[D|G],
            append(G, [I], J),
            gensym__14(K),
            L=..[K|J],
            assert(memo_table(memo(H,I),L)),
            findall((L:-M), unfold__21(H,I,M), N),
            format('/*~k=~k*/~n', [L,memo(H,I)]),
            pretty_print_clauses__2(N),
            memo_table(memo(A,B), C)
        ).
/*unfold__20(_8135,_8137,_8023)=unfold(lix(_8135,_8137),_8023)*/
unfold__20(A, B,
          (retractall(memo_table(_,_)),
           print('/*Generated by Lix*/\n'),
           print(':- dynamic flag/2, memo_table/2.\n'),
           print(':- use_module(library(lists)).\n'),
           print(':- use_module(library(terms)).\n'),
           print(':- op(1150, fx, type).\n'),
           print(':- dynamic filter/2.\n'),print(':- dynamic ann_clause/3. \n'),C)) :-
        memo__14(A, B, C).
/*memo__13(_3160,_3162,_3048)=memo(lix(_3160,_3162),_3048)*/
memo__13(A, B, C) :-
        (   memo_table(lix(A,B), C) ->
            true
        ;   A=..[D|E],
            length(E, F),
            length(G, F),
            H=..[D|G],
            append(G, [I], J),
            gensym__13(K),
            L=..[K|J],
            assert(memo_table(lix(H,I),L)),
            findall((L:-M), unfold__20(H,I,M), N),
            format('/*~k=~k*/~n', [L,lix(H,I)]),
            pretty_print_clauses__2(N),
            memo_table(lix(A,B), C)
        ).
/*lix__2(_1675,_1677,_1563)=lix(lix(_1675,_1677),_1563)*/
lix__2(A, B, C) :-
        retractall(memo_table(_,_)),
        print('/*Generated by Lix*/\n'),
        print(':- dynamic flag/2, memo_table/2.\n'),
        print(':- use_module(library(lists)).\n'),
        print(':- use_module(library(terms)).\n'),
        print(':- op(1150, fx, type).\n'),
        print(':- dynamic filter/2.\n'),
        print(':- dynamic ann_clause/3. \n'),
        memo__13(A, B, C).
```

Bibliography

Sergei M. Abramov and Robert Glück. From standard to non-standard semantics by semantics modifiers. *International Journal of Foundations of Computer Science*, 12 (2):171–211, 2001.

Elvira Albert, Sergio Antoy, and Germán Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of 10th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'2000)*, pages 103–124. Springer LNCS 2042, 2001.

Elvira Albert and Germán Vidal. Symbolic profiling for multi-paradigm declarative languages. In *Logic-Based Program Synthesis and Transformation (Proc. of LOPSTR'01)*, pages 148–167. Springer LNCS 2372, 2002.

Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

Krzysztof R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.

F. Benoy, A. King, and F. Mesnard. Computing Convex Hulls with a Linear Solver. *Theory and Practice of Logic Programming*, January 2004.

Anders Bondorf, Frank Frauendorf, and Michael Richter. An experiment in automatic self-applicable partial evaluation of Prolog. Technical Report 335, Lehrstuhl Informatik V, University of Dortmund, 1990.

D. Boulanger and M. Bruynooghe. A systematic construction of abstract domains. In B. Le Charlier, editor, *Proc. First International Static Analysis Symposium, SAS'94*, volume 864 of *Lecture Notes in Computer Science*, pages 61–77, 1994.

D. Boulanger, M. Bruynooghe, and M. Denecker. Abstracting s-semantics using a model-theoretic approach. In M. Hermenegildo and J. Penjam, editors, *Proc. 6^{th} International Symposium on Programming Language Implementation and Logic Programming, PLILP'94*, volume 844 of *Lecture Notes in Computer Science*, pages 432–446, 1994.

Antony F. Bowers and Corin 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.

B. Brassel, M. Hanus, F. Huch, J. Silva, and G. Vidal. Runtime Profiling of Functional Logic Programs. In *Proc. of the 14th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 178–189, 2004.

M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.

M. Bruynooghe and G. Janssens. An instance of abstract interpretation integrating type and mode inferencing. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of ICLP/SLP*, pages 669–683. MIT Press, 1988.

Maurice Bruynooghe, Michael Codish, John Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis through combination of type based norms. Technical report, Department of Computer Science; Ben-Gurion University, May 2003.

Maurice Bruynooghe, Michael Leuschel, and Kostis Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In Chris Hankin, editor, *Proceedings of the European Symposium on Programming (ESOP'98)*, LNCS 1381, pages 27–41. Springer-Verlag, April 1998.

M. Codish and B. Demoen. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In D. Miller, editor, *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver.* MIT Press, 1993.

M. Codish and B. Demoen. Deriving type dependencies for logic programs using multiple incarnations of Prop. In B. Le Charlier, editor, *Proceedings of SAS'94, Namur, Belgium*, 1994.

M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. *Theoretical Computer Science*, 238(1-2):131–159, 2000.

Michael Codish and Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. Release October 2002.

Stephen-John Craig and Michael Leuschel. A compiler generator for constraint logic programs. In M Broy and A Zamulin, editors, *Perspectives of System Informatics*, volume 2890 of *LNCS*, pages 148–161. Springer, 2003.

Stephen-John Craig and Michael Leuschel. Lix: an effective self-applicable partial evaluator for Prolog. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings*, pages 85–99, 2004.

Stephen-John Craig, Michael Leuschel, John Gallagher, and Kim Henriksen. Fully automatic Binding Time Analysis for Prolog. In Sandro Etalle, editor, *Logic Based Program Synthesis and Transformation, 14th International Workshop*, pages 61–70, 2004.

Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 41(2 & 3):231–277, November 1999.

Stefaan Decorte, Danny De Schreye, Michael Leuschel, Bern Martens, and Konstantinos Sagonas. Termination analysis for tabled logic programming. In Norbert Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, pages 111–127, Leuven, Belgium, July 1998. ISBN 3-540-65074-1.

Stefaan Decorte, Danny De Schreye, and Massimo Fabris. Automatic inference of norms: A missing link in automatic termination analysis. In *ILPS*, pages 420–436, 1993.

N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.

A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing.* Springer-Verlag, 2003.

Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Automated strategies for specializing constraint logic programs. In *Logic Based Program Synthesis and Transformation. Proceedings of Lopstr'2000*, LNCS 1207, pages 125–146, 2000.

Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verifying ctl properties of infinite-state systems by specializing constraint logic programs. In *Proceedings of VCL'2001*, Florence, Italy, September 2001.

H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2 & 3):91–118, 1988.

Y Futamura. Partial evaluation of computation process– an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In J. W. Lloyd, editor, *Proc. of International Logic Programming Symposium*, pages 351–365, 1995.

J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 88–98. ACM Press, 1993. ISBN 0-89791-594-1.

John Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

John Gallagher and Maurice Bruynooghe. Some low-level transformations for logic programs. In M Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 229–244, 1990.

John Gallagher and Maurice Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

John Gallagher and Kim Henriksen. Abstract domains based on regular types. In V Lifschitz and Bart Demoen, editors, *Proceedings of the International Conference on Logic Programming (ICLP'2004)*, LNCS, page to appear. Springer Verlag, 2004.

Samir Genaim and Michael Codish. Inferring termination conditions for logic programs using backwards analysis. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of the Eighth International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 681–690. Springer-Verlag, December 2001.

Robert Glück. On the generation of specialisers. *Journal of Functional Programming*, 4 (4):499–514, 1994.

Robert 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. ISBN 1-58113-458-4.

Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.

Robert Glück and Morten Heine Sørensen. A roadmap to metacomputation by supercompilation. In *Selected Papers from the Internaltional Seminar on Partial Evaluation*, pages 137–160. Springer-Verlag, 1996. ISBN 3-540-61580-6.

C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994a.

C. A. Gurr. Specialising the ground representation in the logic programming language Gödel. In Y. Deville, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'93*, Workshops in Computing, pages 124–140, Louvain-La-Neuve, Belgium, 1994b. Springer-Verlag.

Patricia Hill and John 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.

Carsten Kehler Holst. Syntactic currying: yet another approach to partial evaluation. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1989.

K. Horiuchi and T. Kanamori. Polymorphic type inference in prolog by abstract interpretation. In *Proc. 6th Conference on Logic Programming*, volume 315 of *Lecture Notes in Computer Science*, pages 195–214, 1987.

Joxan Jaffar, Spiro Michaylov, and Roland H. C. Yap. A methodology for managing hard constraints in CLP systems. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 306–316. ACM Press, 1991. ISBN 0-89791-428-7.

N.D. Jones. Transformation by interpreter specialisation. *Science of Computer Programming*, 52:307–339, 2004.

Neil D. Jones. What not to do when writing an interpreter for specialisation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 216–237. Springer-Verlag, 1996.

Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Æ. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *1990 International Conference on Computer Languages, New Orleans, Louisiana, March 1990*, pages 49–58. IEEE Computer Society, March 1990.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., 1993. ISBN 0-13-020249-5.

Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. *SIGPLAN Not.*, 20(8):82–87, 1985. ISSN 0362-1340.

Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler. *List and Symbolic Computation*, 2(1):9–50, 1989.

J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Springer-Verlag.

Jesper Jørgensen, Michael Leuschel, and Bern Martens. Conjunctive partial deduction in practice. In John Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag.

S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

Jan Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 49–69. Springer-Verlag, 1992.

Michael Leuschel. Partial evaluation of the "real thing". In Laurent Fribourg and Franco Turini, editors, Logic Program Synthesis and Transformation — Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, LNCS 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.

Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2004.

Michael Leuschel. Logic program specialisation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, LNCS 1706, pages 155–188 and 271–292, Copenhagen, Denmark, 1999. Springer-Verlag. ISBN 3-540-66710-5.

Michael Leuschel. Homeomorphic embedding for online termination of symbolic methods. In Torben Æ. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation - Essays dedicated to Neil Jones*, LNCS 2566, pages 379–403. Springer-Verlag, 2002. ISBN 3-540-00326-6.

Michael Leuschel and Maurice Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

Michael Leuschel and Michael Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003. ISBN 3-540-40828-2.

Michael Leuschel, Stephen-John Craig, Maurice Bruynooghe, and Wim Vanhoof. Specializing interpreters using offline partial deduction. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, LNCS 3049. Springer-Verlag, 2004a.

Michael Leuschel and Danny De Schreye. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 253–263, La Jolla, California, June 1995. ACM Press.

Michael Leuschel and Danny De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S.D. Swierstra, editors, *Proceedings of PLILP'96*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996. Springer-Verlag.

Michael Leuschel, Jesper Jørgensen, Wim Vanhoof, and Maurice Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004b.

Michael Leuschel, Bern Martens, and Danny 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.

J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

Henning Makholm. On Jones-optimal specialization for strongly typed languages. In Walid Taha, editor, *Semantics, Applications and Implementation of Program Generation*, volume 1924 of *Lecture Notes In Computer Science*, pages 129–148, Montreal, Canada, 20 September 2000. Springer-Verlag.

K. Marriott and P. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, removal, and reordering. In *Proceedings of POPL'93*, pages 334–344. ACM Press, 1993.

Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998. ISBN 0-262-13341-5.

B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995a.

B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *The Journal of Logic Programming*, 22(1):47–99, 1995b.

Bern Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming.* PhD thesis, K.U. Leuven, February 1994.

Bern Martens and John Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L Sterling, editor, *Proceedings ICLP'95*, pages 597–613. MIT Press, June 1995.

T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In Kung-Kiu Lau and Tim Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.

Julio C. Peralta. *Analysis and Specialisation of Imperative Programs: An approach using CLP.* PhD thesis, Department of Computer Science, University of Bristol, July 2000.

Julio C. Peralta and John P. Gallagher. Towards semantics-based partial evaluation of imperative programs. Technical Report CSTR-97-003, Department of Computer Science, University of Bristol, April 1997.

Julio C. Peralta and John P. Gallagher. Convex hull abstractions in specialization of CLP programs. In Michael Leuschel, editor, *Logic-based Program Synthesis and Transformation (LOPSTR'2002)*, LNCS 2664, pages 90–108, Madrid, Spain, September 2002. Springer-Verlag.

Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.

Steven Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

Donald A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In N. D. Jones and P. Hudak, editors, *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–71. ACM Press Sigplan Notices 26(9), 1991.

Donald A. Smith and Timothy Hickey. Partial evaluation of a CLP language. In S.K. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 119–138. MIT Press, 1990.

Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1–3):17–64, 1996.

Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In John W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479, Portland, USA, December 1995. MIT Press.

Michael Sperber, Peter Thiemann, and Hervert Klaeren. Distributed partial evaluation. In *Proceedings of the second international symposium on Parallel symbolic computation*, pages 80–87. ACM Press, 1997. ISBN 0-89791-951-3.

Leon Sterling and Ehud Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques.* MIT Press, 1994. ISBN 0-262-19338-8.

Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248:211–242, 2000.

Yi Tao, William I. Grosky, and Chunnian Liu. An automatic partial deduction system for constraint logic programs. In *ICTAI*, pages 149–156, 1997.

W. Vanhoof and B. Martens. To parse or not to parse. In Norbert Fuchs, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'97*, LNCS 1463, pages 322–342, Leuven, Belgium, July 1997. ISBN 3-540-65074-1.

Wim Vanhoof. Binding-time analysis by constraint solving: a modular and higher-order approach for mercury. In M. Parigot and Andrei Voronkov, editors, *Proceedings of LPAR'2000*, LNAI 1955, pages 399–416. Springer-Verlag, 2000.

Wim Vanhoof and Maurice Bruynooghe. Binding-time analysis for mercury. In Danny De Schreye, editor, *Proceedings of the International Conference on Logic Programming ICLP'99*, pages 500–514. MIT Press, 1999.

Wim Vanhoof and Maurice Bruynooghe. Binding-time annotations without binding-time analysis. In R. Nieuwenhuis and A Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference*, LNCS 2250, pages 707–722. Springer-Verlag, 2001a.

Wim Vanhoof and Maurice Bruynooghe. When size does matter. In *LOPSTR '01: Selected papers from the 11th International Workshop on Logic Based Program Synthesis and Transformation*, pages 129–147. Springer-Verlag, 2001b. ISBN 3-540-43915-3.

Wim Vanhoof, Maurice Bruynooghe, and Michael Leuschel. Binding-time analysis for Mercury. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, LNCS 3049. Springer-Verlag, 2004.

Raf Venken and Bart Demoen. A partial evaluation system for Prolog: Theoretical and practical considerations. *New Generation Computing*, 6(2 & 3):279–290, 1988.

G. Vidal. Cost-Augmented Partial Evaluation of Functional Logic Programs. *Higher-Order and Symbolic Computation*, 17(1-2):7–46, 2004.

R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1–2):3–35, 2001.

Index