Self-Tuning Resource Aware Specialisation for Prolog

Stephen-John Craig
University of Southampton, United Kingdom
University of Düsseldorf, Germany
steve.craig@gmail.com

Michael Leuschel*
University of Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de

ABSTRACT

The paper 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, tailouring the specialisation to the particular problem at hand. We present experimental results for our implemented technique on a series of benchmarks. The results are compared against the aggressive termination based binding-time analysis and optimised using different measures for the quality of code. We also show that our technique avoids some classical pitfalls of partial evaluation.

Keywords

Partial Evaluation, Partial Deduction, Binding-Time Analysis,Logic Programming

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.

In this paper we present 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 recent fully automatic binding-time analysis (BTA)[5]. The insight on which this paper 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 tradeoff 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 paper, shows, empirically and through examples, how it avoids pitfalls which other specialisers such as ECCE [16] or MIXTUS [21] fall into. We also show how we can

*The authors have been partially supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 ASAP project.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

achieve a good tradeoff between various resource considerations. It is also demonstrated on a series of benchmark programs the practicality and performance of the approach.

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 [25]. In [25] 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.

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 [1, 2, 24, 4]. 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 paper 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.

2. CONTROLLING PARTIAL DEDUCTION

In the remainder of the paper we assume some basic knowledge of logic programming [17].

2.1 Basics of Partial Deduction

Partial deduction [18] is a program specialisation technique for logic programs: given a logic program P and some partially instantiated query Q, it derives a new program P', which is specialised for answering the query Q and all its possible instantiations. Partial deduction proceeds by deriving a set of atoms $\mathcal{A}$ and by building for each element of $\mathcal{A}$ a possibly incomplete SLDNF-tree using an *unfolding rule*. For every element of $\mathcal{A}$, partial deduction produces a specialised predicate definition by extracting a specialised clause from every non-failing branch of the tree built for it.

An important issue in partial deduction is the control. Here we distinguish between [19]

- global control: deciding which atoms to be put into $\mathcal{A}$, and
- local control: deciding which trees to build for the elements of $\mathcal{A}$.

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 [14]), 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.

2.2 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 1.

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

Listing 1: The `inboth/3` example

Let $\mathcal{A}$ = {`inboth(a,L,[X,Y])`, `member(a,L)`}. By performing non-leftmost non-determinate unfolding for the call `inboth(a,L,[X,Y])` in Figure 1 (and doing a single unfolding step for `member(a,L)`), we obtain the partial deduction P' (Listing 2) of P with respect to $\mathcal{A}$.

```
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 2: Specialising Listing 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 $\mathcal{A}$). 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'.

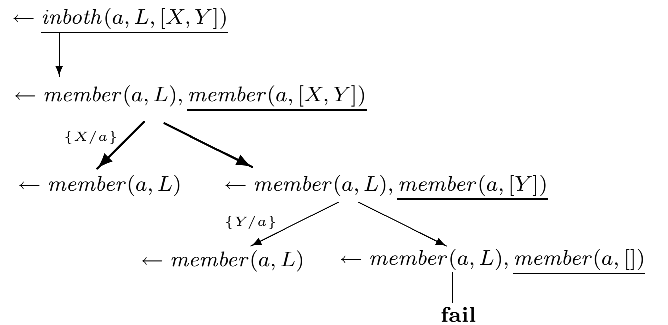

Figure 1: Non-leftmost non-determinate unfolding for Listing 2

The classical solution to this problem is to disallow non-leftmost unfolding unless it is deterministic (SP [7, 8, 9], ECCE [16]), or allow non-leftmost unfolding but not left-propagate bindings (PADDY [20], MIXTUS [21]). Some partial evaluators, for instance, SAGE [11, 10] do not prevent such work duplication. This can result in huge slowdowns (see, e.g., [3]).

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 1 with the set $\mathcal{A} = \{$`inboth(X,[Y],[V,W])`$\}$. The query can be fully unfolded producing the partial deduction P' (Listing 3) of P with respect to $\mathcal{A}$.

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

Listing 3: Specialising Listing 1 for {inboth(X,[Y],[V,W])}

No goal has been duplicated by the leftmost non-determinate unfolding, but the unification `X=Y` for $\leftarrow$ `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 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.

```
 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 4: Example using clause 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 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 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).

```
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 5: Specialising Listing 4 for `index_test(A,B,C)`. The useful clause indexing has been lost

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 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 [16]).[2] 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 runtime 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 [23], most of which are still valid today). Also, even when partial deduction

[2]This is less of an issue in conjunctive partial deduction, as variable links between calls are not automatically lost when one stops unfolding; see [12].

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 paper 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.

3. OFFLINE PARTIAL DEDUCTION

The main idea of offline partial deduction is to separate the specialisation process into two phases:

- First a *binding-time analysis* (*BTA*) is performed which, given a program and an approximation of the input available for specialisation, approximates all values within the program and generates an *annotated program*.
- A (simplified) *specialisation phase*, which is guided by the annotations of the BTA.

This approach is illustrated in Figure 2 and is called *offline* because most control decisions are taken beforehand.

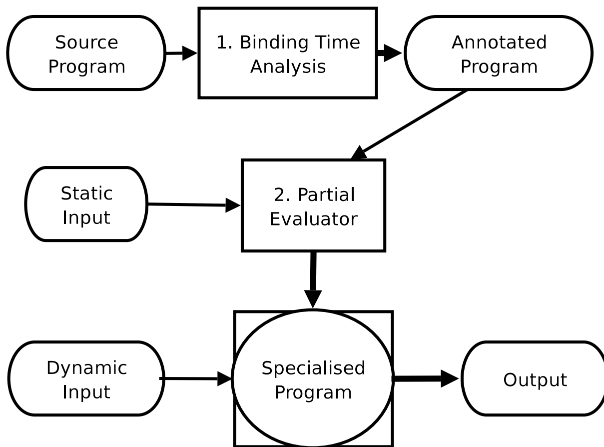

Figure 2: Offline Partial Evaluation

In the remainder of the paper we will use the LOGEN specialisation system [15]. In LOGEN every program point in every clause is annotated with a clause annotation, telling the specialiser what to do when reaching this program point. Furthermore, every argument of every predicate is annotated with a binding type, which tells the specialiser to what extent this argument will be known at specialisation time.

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* [19]. For the LOGEN system [15] the main annotations are as follows:

- **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. (Note: the predicate body is itself annotated and will be re-examined by the partial evaluator.)
- **memo**: The call is not unfolded, but 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.

Binding Types

Each argument of a predicate in an annotated program is given a *binding type* by means of *filter declarations*. A binding type indicates something about the structure of an argument at specialisation time. This information is used when the predicate is "memoed." The basic binding types are usually known as *static* and *dynamic* which are defined as follows:

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

More precise binding types can be defined by means of regular type declarations, and combined with basic binding types. For example, one can define types such as list skeletons.

The filter declarations influence the *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.

The paper [5] introduced an automatic binding-time analysis for LOGEN. 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 was 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 paper is designed to complement the binding-time analysis of [5], providing control over the quality of the produced specialised programs.

Figure 3 is the match program taken from the DPPD library of benchmarks [13]. The program is a naïve string matcher; the `match/2` succeeds if the given pattern occurs in the string. The program has been annotated for the LOGEN system 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.

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

Figure 3: Annotated match program

Using the above annotation and the specialisation query `match([a,c], A)`, LOGEN will produce the following specialised 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: Specialising match/2 using the annotations in Figure 3

4. MUTATIONS

Offline specialisation takes an annotated program as input. In this section we examine 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 1).

DEFINITION 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 3) 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 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 2 (MUTATION). *Let C be a 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 3 (SET OF MUTATIONS). *$mutations(C)$ is defined as the set of all possible mutations of C.*

Table 1 shows the initial set of mutations for the match program in Figure 3. 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 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 1 produces the new *safe* annotations in Table 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 2: Mutation after filter propagation

Two of the mutations have been detected as unsafe and have been modified accordingly. Figure 4 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 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 3: Mutation after full automatic binding-time analysis

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 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 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.[3] A fitness function (Definition 5) is used to determine the score given the characteristics.

DEFINITION 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 tradeoff 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

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

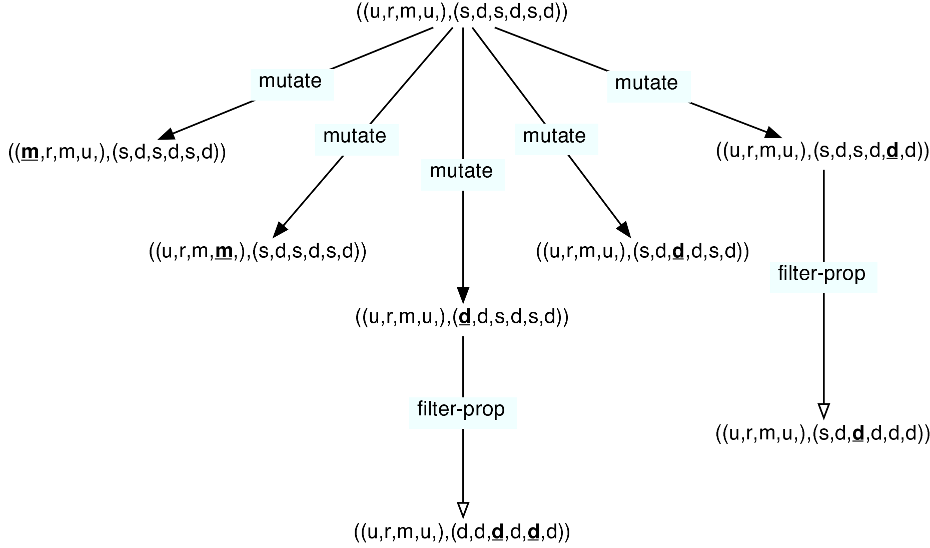

Figure 4: Safe annotation configurations after filter propagation

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 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. 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 5 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 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.

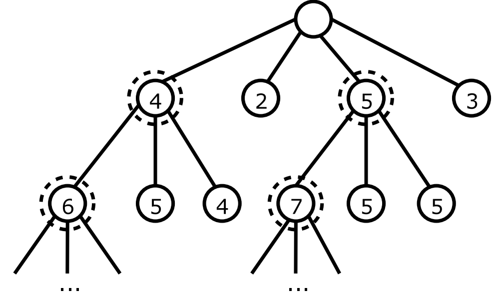

Figure 5: Beam search for $W = 2$

Algorithm 1 describes the self-tuning algorithm in psuedo code. It uses Definition 6 to measure the characteristics of an annotation configuration.

Definition 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 4) produces the annotations in Figure 7. 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 performace) and has marked it as memo. The tuned annotations produced specialised code that is twice as fast as the aggressive annotations.

Figure 8 is the self-tuned output for the `match/2` program (Figure 3), optimised for both size and time. The algorithm

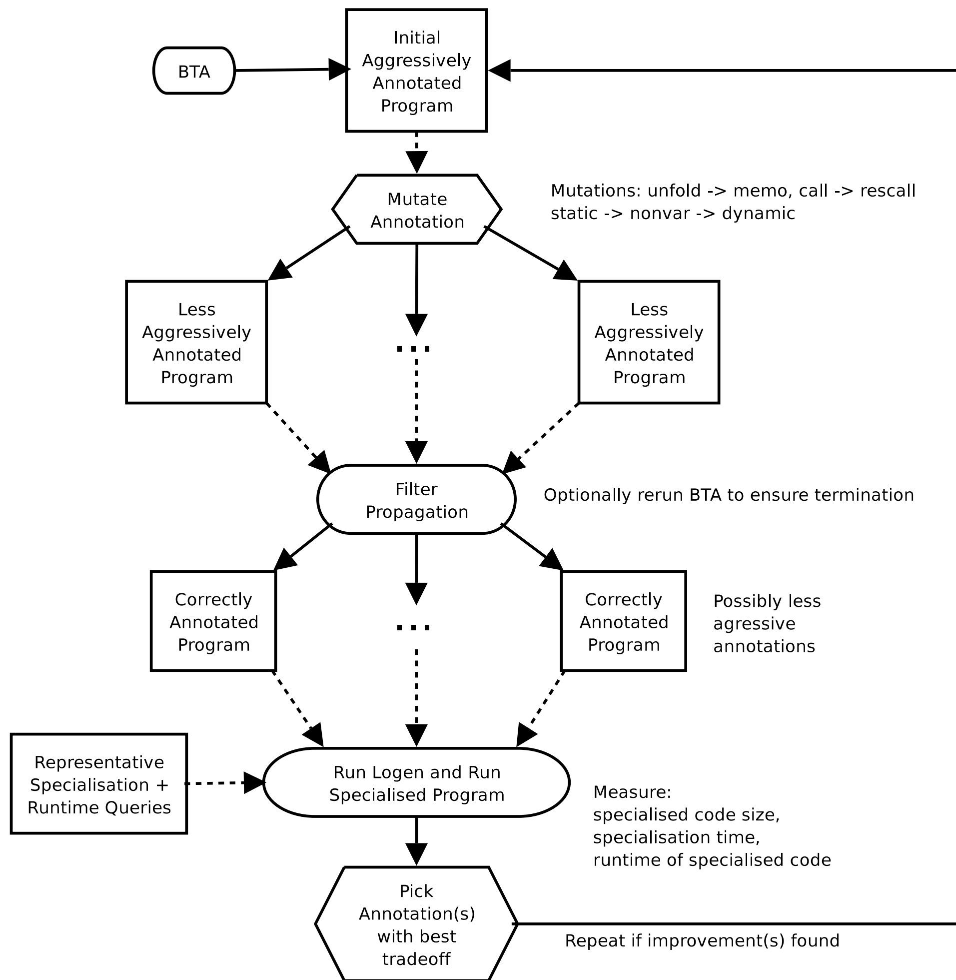

Figure 6: Self-tuning overview

```
index_test(f(_), Y, Z) :- p(Y, Z).
                          \____/
                           memo
...
```

Figure 7: Final annotations for `index_test/3`, optimised for time and size

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 4).

7. EXPERIMENTS

Table 4 presents the results of running[4] the self-tuning algorithm on a series of benchmarks taken from the DPPD library [13]:

advisor – A simple expert system.

inboth – The inboth example form Section 2.2.

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

```
match(Pat, T) :-
        match1(Pat, T, Pat, T).
        \_____________________/
                 memo
match1([], _Ts, _P, _T).
match1([A|_Ps], [B|_Ts], P, [_X|T]) :-
        A \== B, match1(P, T, P, T).
        \_____/  \________________/
        rescall        memo
match1([A|Ps], [A|Ts], P, T) :-
        match1(Ps, Ts, P, T).
        \___________________/
               unfold
:- filter match(static, dynamic).
:- filter match1(static, dynamic, static, dynamic).
```

Figure 8: Final annotations for `match/2`, optimised for time and size

index_test – The indexing example from Section 2.2.

match – A simple naïve pattern matcher.

missionaires – A program for the missionaries and cannibals problem.

Algorithm 1 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** $CS = Previous$

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 [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. $score = reduction$.

time & size – An equally weighting of the normalised execution time and program size. $score = speedup \times reduction$.

The execution time, compiled code size and specialisation time are the non-normalised characteristics from Section 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. Our 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 comprimise solution finds configurations which perform marginally slower than the fastest, but without the code explosion.

8. SUMMARY AND FUTURE WORK

This paper 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 a recent fully automatic binding-time analysis [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 of [5] can lead to large code explosion for little gain in efficiency, while our algorithm finds a much better tradeoff.

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 con-

Benchmark Program	Fitness Function	Execution Time	Compiled Code Size (bytes)	Specialisation Time	Optimisation Time	Attempted Configurations
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 4: Experimental results for the self-tuning algorithm

figuration, 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*,[5] 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 scal-

[5]Strictly speaking our current algorithm is actually closer to an evolutionary algorithm rather than a genetic algorithm [6].

ability 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 [5] is an iterative algorithm. During the algorithm described in this paper, 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 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 previous work [22] the specialisation process itself was parallelised, distributing the work over a network of work stations.

Acknowledgements

We would like to thank Michael Roskopf, Bart Demoen, John Gallagher, and all the other partners of the ASAP project for their help and input.

9. REFERENCES

[1] E. Albert, S. Antoy, and G. 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.

[2] E. Albert and G. 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.

[3] A. F. Bowers and C. A. Gurr. Towards fast and declarative meta-programming. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 137–166. MIT Press, 1995.

[4] 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.

[5] S.-J. Craig, M. Leuschel, J. Gallagher, and K. Henriksen. Fully automatic Binding Time Analysis for Prolog. In S. Etalle, editor, *Logic Based Program Synthesis and Transformation, 14th International Workshop*, pages 61–70, 2004.

[6] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, 2003.

[7] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[8] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

[9] 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.

[10] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[11] C. A. Gurr. Specialising the ground representation in the logic programming language Gödel. In Y. Deville, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'93*, Workshops in Computing, pages 124–140, Louvain-La-Neuve, Belgium, 1994. Springer-Verlag.

[12] J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'96*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag.

[13] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2004.

[14] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[15] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.

[16] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[17] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[18] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.

[19] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613. MIT Press, June 1995.

[20] S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[21] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[22] M. Sperber, P. Thiemann, and H. Klaeren. Distributed partial evaluation. In *Proceedings of the*

second international symposium on Parallel symbolic computation, pages 80–87. ACM Press, 1997.

[23] R. Venken and B. Demoen. A partial evaluation system for Prolog: Theoretical and practical considerations. *New Generation Computing*, 6(2 & 3):279–290, 1988.

[24] G. Vidal. Cost-Augmented Partial Evaluation of Functional Logic Programs. *Higher-Order and Symbolic Computation*, 17(1-2):7–46, 2004.

[25] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1–2):3–35, 2001.