ASAP

IST-2001-38059

Advanced Analysis and Specialization for Pervasive Systems

Self-tuning Specialization Systems

Deliverable number:	D12
Workpackage:	Resource-Oriented Specialization (WP4)
Preparation date:	1 July 2004
Due date:	1 July 2004
Classification:	Public
Lead participant:	Univ. of Southampton
Partners contributed:	Tech. Univ. of Madrid (UPM),Univ. of Bristol, Univ. of Southampton

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

Short description:

This deliverable studies the development of self-tuning specialization techniques.

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.

2 Offline Self-Tuning

In the first part of the deliverable we present a *self-tuning* system, which derives its own specialisation control using a genetic algorithm approach.

More precisely, we use an offline approach based on the recent fully automatic binding-time analysis (BTA). The insight on which this paper is based, is that the annotations can form the genes for a genetic algorithm.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 deliverable, shows, empirically and through examples, how it avoids pitfalls which other specialisers such as ECCE or MIXTUS fall into. We also show how we can 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.

3 Online Self-tuning

In this part of the deliverable we propose a framework for on-line partial deduction which, on one hand allows finer-grained handling of control decisions by allowing the use of several control strategies during a partial deduction session, and on the other hand allows the possibility of exploring different candidate partial deductions.

Existing algorithms for partial deduction, given a local and a global control rule, deterministically produce a specialized program from an initial program and description of run-time queries. In this work we propose a novel framework for partial deduction of logic programs which is *poly-controlled* in that it can take into account repertoires of global control and local control rules instead of a single, predetermined one. This framework is more flexible than existing ones since it allows assigning *different* global and local control rules to different atoms. In addition, this modification potentially transforms partial deduction from a *greedy* algorithm into a *search-based* algorithm. As a result, sets of candidate specialized programs can be achieved, instead of a single one.

In order to make the algorithm fully automatic, it is required the use of self-tuning techniques which allow measuring the quality of the different candidate programs automatically. The framework is resource aware in that it uses fitness functions which consider multiple factors such as run-time and code size for the specialized programs. Finally, and in order to make the proposed framework effective, we present pruning mechanisms which allow reducing the search space by allowing code generation even for configurations which are not final and discuss the possibility of using branch and bound explorations of the search-space. We also mention the possibility or re-using previous selections in order to speed-up the specialization process for different queries to a same program. The framework has been implemented in CiaoPP and the preliminary experiments performed so far are promising.

Contents

Part I

Self-Tuning Resource Aware Specialisation for Prolog

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.

4 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)[CLGH04]. 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 [LMDS98] or MIXTUS [Sah93] fall into. We also show how we can 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.

4.1 Other Approaches and Related Work

Such an approach has already proven to be highly successful in the context of optimising scientific linear algebra software [WPD01]. In [WPD01] 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 [AAV01, AV01, Vid04, BHH+04]. 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

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

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.

5 Controlling Partial Deduction

In the remainder of the paper we assume some basic knowledge of logic programming [Llo87].

5.1 Basics of Partial Deduction

Partial deduction [LS91] 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 [MG95]

- 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 [LB02]), 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.

5.2 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'.

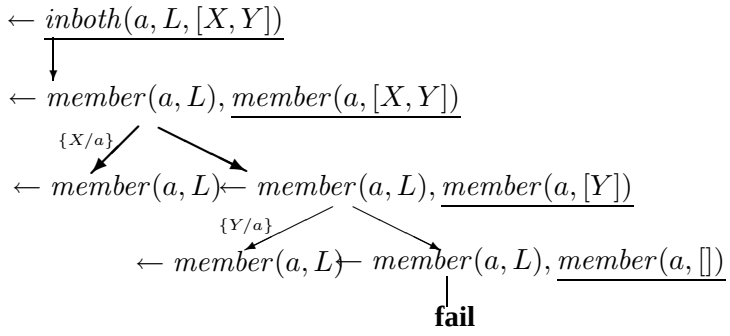

Figure 1: Non-leftmost non-determinate unfolding for Listing 2

The classical solution to this problem is to disallow non-leftmost unfolding unless it is deterministic (SP [Gal91, GB91, Gal93], ECCE [LMDS98]), or allow non-leftmost unfolding but not left-propagate bindings (PADDY [Pre92], MIXTUS [Sah93]). Some partial evaluators, for instance, SAGE [Gur94b, Gur94a] do not prevent such work duplication. This can result in huge slowdowns (see, e.g., [BG95]).

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 ← `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 [LMDS98]). This is less of an issue in conjunctive partial deduction, see [JLM96].

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 [VD88], 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 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.

6 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.

In the remainder of the paper we will use the LOGENspecialisation system [LJVB04]. In LOGENevery program point in every clause is annotated with a clause annotation, telling the

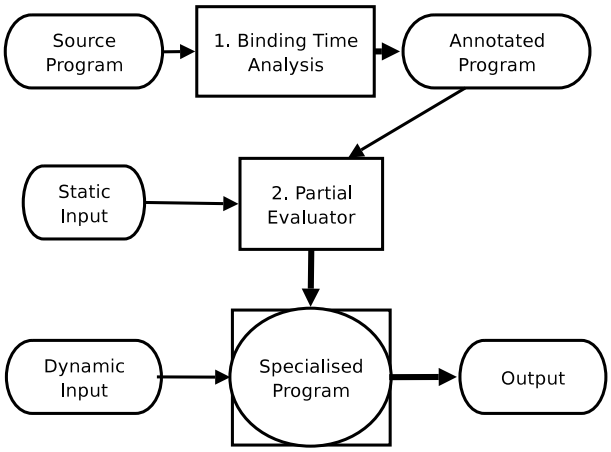

Figure 2: Offline Partial Evaluation

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* [MG95]. For the LOGENsystem [LJVB04] 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 [CLGH04] 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 [CLGH04], providing control over the quality of the produced specialised programs.

Figure 3 is the match program taken from the DPPD library of benchmarks [Leu02]. 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 LOGENsystem 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 above annotation and the specialisation query `match([a,c], A)`, LOGEN will produce the following specialised program:

$\texttt{match}(\texttt{Pat}, \texttt{T}) :-$

$\quad \underbrace{\texttt{match1}(\texttt{Pat}, \texttt{T}, \texttt{Pat}, \texttt{T})}_{unfold}.$

$\texttt{match1}([], \texttt{\_Ts}, \texttt{\_P}, \texttt{\_T}).$

$\texttt{match1}([\texttt{A}|\texttt{\_Ps}], [\texttt{B}|\texttt{\_Ts}], \texttt{P}, [\texttt{\_X}|\texttt{T}]) :-$

$\quad \underbrace{\texttt{A} \backslash == \texttt{B}}_{rescall}, \underbrace{\texttt{match1}(\texttt{P}, \texttt{T}, \texttt{P}, \texttt{T})}_{memo}.$

$\texttt{match1}([\texttt{A}|\texttt{Ps}], [\texttt{A}|\texttt{Ts}], \texttt{P}, \texttt{T}) :-$

$\quad \underbrace{\texttt{match1}(\texttt{Ps}, \texttt{Ts}, \texttt{P}, \texttt{T})}_{unfold}.$

$:-\texttt{filter}\ \texttt{match}(\texttt{static}, \texttt{dynamic}).$

$:-\texttt{filter}\ \texttt{match1}(\texttt{static}, \texttt{dynamic}, \texttt{static}, \texttt{dynamic}).$

Figure 3: 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: Specialising `match/2` using the annotations in Figure 3

7 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.

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.

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

- 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.

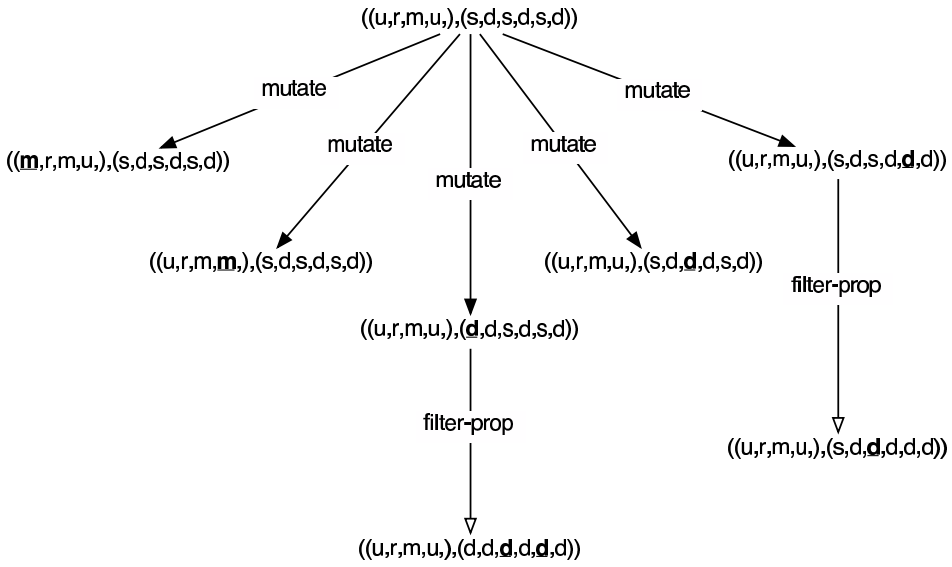

Figure 4: Safe annotation configurations after filter propagation

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

8 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.[2] 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 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.

9 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.

[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.

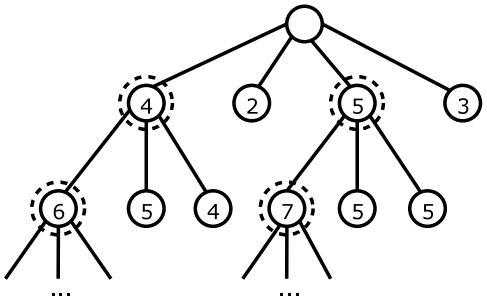

Figure 5: Beam search for $W = 2$

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.

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 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).

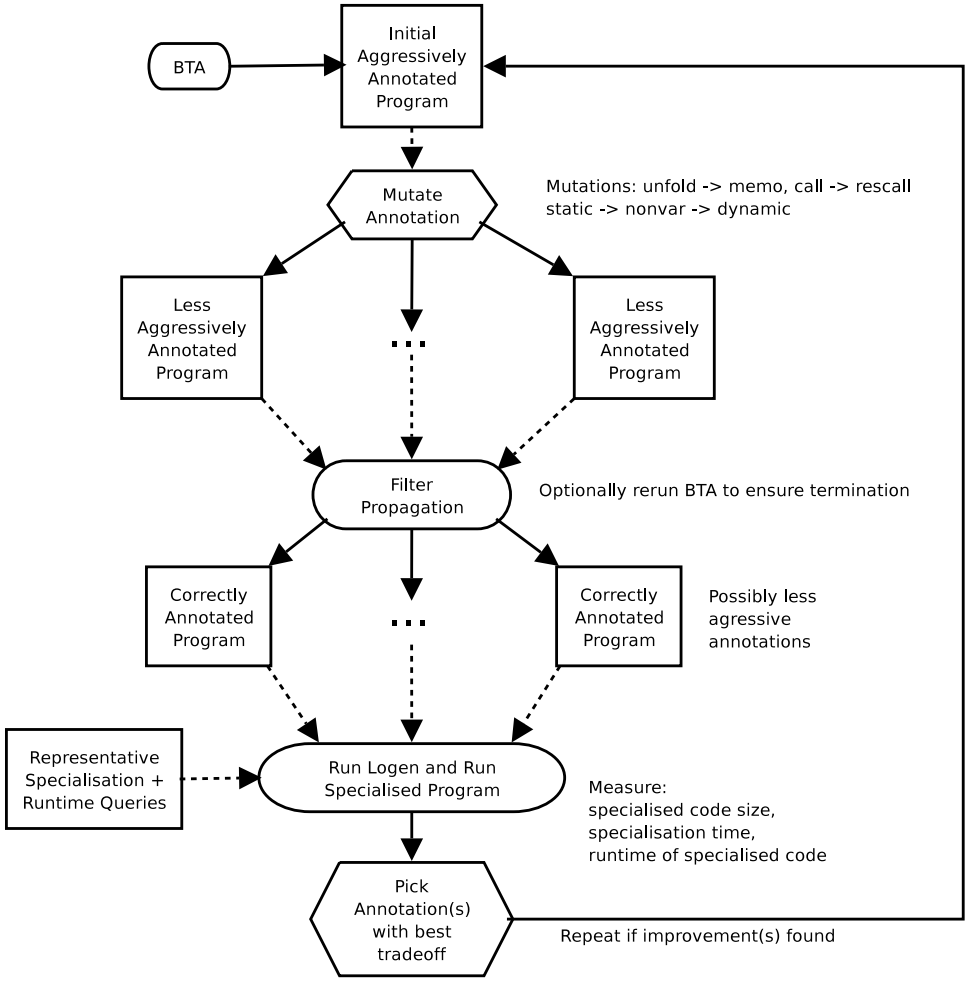

Figure 6: Self-tuning overview

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

...

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

10 Experiments

Table 4 presents the results of running[3] the self-tuning algorithm on a series of benchmarks taken from the DPPD library [Leu02]:

advisor – A simple expert system.

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

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: $Cache$ = $\{C_{dyn} \mapsto \mathit{fitness\_func}(1, 1, 1)\}$
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: $Score$ = $\mathit{fitness\_func}(ST', RT', SS')$
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** $CS = Previous$

inboth – The inboth example form Section 5.2.

index_test – The indexing example from Section 5.2.

match – A simple naïve pattern matcher.

```
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

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 [CLGH04], 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 8. 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

Benchmark Program	Fitness Function	Execution Time	Compiled Code Size (bytes)	Specialisation Time	Optimisation Time	Attempted Confi gurations
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

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.

11 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 [CLGH04] 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 [CLGH04] 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 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 granuality 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 [CLGH04] 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

[4]Strictly speaking our current algorithm is actually closer to an evolutionary algorithm rather than a genetic algorithm [ES03].

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 [STK97] 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.

Part II

Poly-Controlled Partial Deduction and its application to Self-Tuning Specialization

Abstract

Given a local and a global control rule, existing algorithms for partial deduction deterministically produce a specialized program from an initial program and description of run-time queries. In this work we propose a novel framework for partial deduction of logic programs which is *poly-controlled* in that it can take into account repertoires of global control and local control rules instead of a single, predetermined one. This framework is more flexible than existing ones since it allows assigning *different* global and local control rules to different atoms. In addition, this modification potentially transforms partial deduction from a *greedy* algorithm into a *search-based* algorithm. As a result, sets of candidate specialized programs can be achieved, instead of a single one. In order to make the algorithm fully automatic, it is required the use of self-tuning techniques which allow measuring the quality of the different candidate programs automatically. The framework is resource aware in that it uses fitness functions which consider multiple factors such as run-time and code size for the specialized programs. Finally, and in order to make the proposed framework effective, we present pruning mechanisms which allow reducing the search space by allowing code generation even for configurations which are not final and discuss the possibility of using branch and bound explorations of the search-space.

12 Introduction

The goal of *partial deduction* is to speedup programs by performing certain computation steps at compile-time rather than at run-time. The quality of the code generated by partial deduction greatly depends on the *control strategy* used. Traditional partial deduction algorithms are parametric w.r.t. the so-called *global control* and *local control* rules. The issue of devising good control rules has received considerable attention.

However, it is well known that sometimes partial deduction can slow-down programs and it is rather difficult to predict the performance results of different control strategies. If, in addition to time-performance, we also take into account the size and memory usage of the residual program, then selecting good control rules becomes an extremely difficult task. The existence of sophisticated control rules which behave (almost) optimally for all programs is still far from

reality. Thus, it seems interesting to be able to develop a program transformation framework which (1) can flexibly use different control strategies for different atoms and (2) can generate several candidate specializations which can be experimentally compared for efficiency, in terms of multiple factors such as size of the specialized program and time- and memory-efficiency of such specialized program.

In order to do so, several difficulties have to be overcome. Traditional on-line techniques for partial deduction of logic programs are not very well suited for self-tuning specialization techniques for several reasons. One is that the algorithm does not provide a self-contained specialized program until the partial deduction algorithm has computed a set of atoms which is *closed*, moment in which partial deduction terminates. Another one is that the algorithm assumes the existence of a global control and a local control rule which behaves well for all atoms in the specialized program. However, in practical situations it can be very useful to be able to use *different* global and local control rules for different atoms.

In this work we propose a framework for on-line partial deduction which allows using different global and local control rules for different atoms. Optionally, an algorithm can be devised which, instead of a single specialized program, generates a set of candidate specialized programs. The framework is self-tuning in that it uses empirical evaluations for selecting the best candidates by means of a *fitness function*. It is also resource-aware in that multiple factors, such as size of specialized programs and their memory consumption can be taken into account by the fitness function in addition to the natural consideration of time-efficiency of the specialized programs. In order to be able to effectively explore the search space introduced by the use of several control strategies, the framework allows pruning intermediate states thanks to the use of a *closure* operation which allows generating code even for sets of atoms which are not closed. We also sketch the possibility of using branch and bound in order to efficiently explore the search space while obtaining specialized programs which are guaranteed to behave optimally w.r.t. the fitness function of interest.

13 Background

We assume some basic knowledge on the terminology of logic programming. See for example [Llo87] for details. Very briefly, an *atom* A is a syntactic construction of the form $p(t_1, \ldots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \ldots, t_n$ are terms. The function $pred$ applied to atom A, i.e., $pred(A)$, returns the predicate symbol p/n for A. A *clause* is of the form $H \leftarrow B$ where its head H is an atom and its body B is a conjunction of atoms. A *definite program* is a finite set of clauses. A *goal* (or query) is a conjunction of atoms. The con-

cept of *computation rule* is used to select an atom within a goal for its evaluation. A *computation rule* is a function $\mathcal{R}$ from goals to atoms. Let G be a goal of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$, $k \geq 1$. If $\mathcal{R}(G) = A_R$ we say that A_R is the *selected* atom in G. The operational semantics of definite programs is based on derivations. We provide the definition of a derivation step below. See [Llo87] for more details.

Definition 7 (derivation step) *Let G be $\leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause in P. Then G' is* derived *from G and C via $\mathcal{R}$ if the following conditions hold:*

$$\theta = mgu(A_R, H)$$

$$G' \textit{ is the goal } \leftarrow \theta(A_{R-1}, B_1, \ldots, B_m, A_1, A_{R+1}, \ldots, A_k)$$

As customary, given a program P and a goal G, an *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees*. A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer for goal G. Such a derivation is called *failed* if it is not possible to perform a derivation step with G_n.

In order to compute a *partial deduction* (PD) [LS91], given an input program and a set of atoms (goal), the first step consists in applying an *unfolding rule* to compute finite (possibly incomplete) SLD trees for these atoms. Given an atom A, an unfolding rule computes a set of finite SLD derivations $D_1, \ldots, D_n$ (i.e., a possibly incomplete SLD tree) of the form $D_i = A, \ldots, G_i$ with computer answer substitution θ_i for $i = 1, \ldots, n$ whose associated *resultants* are $\theta_i(A) \leftarrow G_i$. Therefore, this step returns the set of resultants, i.e., a program, associated to the root-to-leaf derivations of these trees. The set of resultants for the computed SLD tree is called a *partial deduction* (PD) for the initial goal (query). We refer to [LB02] for details. In order to ensure the local termination of the PD algorithm while producing useful specializations, the unfolding rule must incorporate some non-trivial mechanism to stop the construction of SLD trees. Nowadays, well-founded orderings (wfo) [BSM92, MD96] and well-quasi orderings (wqo) [SG95, Leu98] are broadly used in the context of on-line PE techniques (see, e.g., [Gal93, LMDS98, SG95]). In addition to local termination, an *abstraction operator* is applied to properly add the atoms in the right-hand sides of resultants to the set of atoms to be partially evaluated. This abstraction operator performs the *global control* and is in charge of guaranteeing that the number of atoms which are generated remains finite by replacing atoms by more general ones, i.e., by losing precision in order to guarantee termination.

14 The Dilemma of Controlling Partial Deduction

As mentioned above, there exist many powerful local and global control rules to choose from. Just as an example, in the case of local control, one of the decisions which have to be taken is whether to allow non-leftmost unfolding or not. It is well known that performing unfolding steps w.r.t. atoms which are not leftmost can slow-down programs. Also, in the presence of *impure* predicates, non-leftmost unfolding can even produce incorrect results. A detailed discussion on the correctness on non-leftmost unfolding can be found in [APG05], which is included as part of Deliverable D06. On the other hand, performing non-leftmost unfolding can provide important speedups in other cases. See the program in Listing 7.

```
exp(Base,Exp,Res):- exp_ac(Exp,Base,1,Res).

exp_ac(0,_,Res,Res).
exp_ac(Exp,Base,Tmp,Res):-
        Exp > 0,
        Exp1 is Exp - 1,
        NTmp is Tmp * Base,
        exp_ac(Exp1,Base,NTmp,Res).
```

Listing 7: The `exponential/3` example

If we specialize it w.r.t. the query `exp(Base,2,Res)`, enabling non-leftmost unfolding allows to unroll the recursive calls and the residual code is:

```
exp(A,2,B) :- C is 1*A, B is C*A.
```

Consider now the program in Listing 8 below:

```
p(B):-  C is B + 1, q(C).

q(1).
q(2).
q(3).
q(4).
q(5).
q(6).
```

Listing 8: The `p/1` example

Since the call `C is B + 1` to the built-in predicate `is/2` is not sufficiently instantiated to be executed (`B` is not yet bound to an arithmetic expression), it is required to enable non-leftmost unfolding in order to unfold the call `q(C)`. However, such unfolding generates the following residual code shown in Listing 9 below:

```
p(A) :- 1 is A+1.
p(A) :- 2 is A+1.
p(A) :- 3 is A+1.
p(A) :- 4 is A+1.
p(A) :- 5 is A+1.
```

```
p(A) :- 6 is A+1.
```

Listing 9: The residual code for `p/1`

This code is less efficient than the original definition of `p/1`, since the indexing for predicate `q/1` is lost and the calls to built-in `is/2` have to be speculatively performed until a success is found, if any. In summary, the same feature of a local control rule, i.e., whether to allow non-leftmost unfolding, can be beneficial for certain calls (atoms) and can be counter productive in others.

15 Partial Deduction as a Greedy Algorithm

As it is well known, *greedy algorithms* are characterized by starting from an initial *configuration* (or state) *Conf*$_0$ and repeatedly applying a *transformation rule T* which given a configuration *Conf*$_i$ produces a successor configuration *Conf*$_{i+1}$ s.t. *Conf*$_{i+1}$=T(*Conf*$_i$) until a configuration *Conf*$_n$, $n \geq 0$, is reached which satisfies certain condition which guarantees that *Conf*$_f$ is *final*.

It is possible to consider traditional partial deduction frameworks as greedy algorithms. See Algorithm 2. A configuration *Conf*$_i$ is a pair $\langle S_i, H_i \rangle$ s.t. S_i is the set of atoms yet to be handled by the algorithm and H_i is the set of atoms already handled by the algorithm. Indeed, in H_i not only we store atoms but also the result of applying global control to such atoms, i.e., members of H_i are pairs of the form $\langle A_i, A'_i \rangle$. Correctness of the algorithm requires that each A'_i is an *abstraction* of A_i, i.e., $A_i = A'_i\theta$.

Given a set of atoms S which describe the potential queries to the program, the initial configuration is of the form $\langle S, \emptyset \rangle$. In each iteration of the algorithm, an atom A_i from S is selected (line 5). Then, global control and local control as defined by the *Abstract* and *Unfold* rules, respectively, are applied (lines 6 and 7). This builds an SLD-tree for A'_i, a generalization of A_i as determined by *Abstract*, using the predefined unfolding rule *Unfold*. Once the SLD-tree τ_i is computed, the leaves in its resultants, i.e., the atoms in the residual code for A'_i are collected by the function *leaves*. Those atoms in *leaves*(τ_i) which are not a variant of an atom handled in previous iterations of the algorithm are added to the set of atoms to be considered. We use $B \equiv A$ to denote that B and A are *variants*, i.e., they are equal modulo variable renaming. A configuration is final when it is of the form $\langle \emptyset, H \rangle$. The specialized program corresponds to $\bigcup_{\langle A,A' \rangle \in H_n} resultants(A')$. Note that this algorithm differs from those in [Gal93, LB02] in that once an atom A_i is abstracted into A'_i, code for A'_i will be generated, and it will not be abstracted any further no matter which other atoms are handled in later iterations of the algorithm. As a result, the set of atoms it generates are not guaranteed to be *independent*. Two atoms are independent when they have no common instance. However, the pairs in H allow to uniquely

Algorithm 2 Greedy Partial Deduction algorithm

Input: Program P

Input: Set of atoms of interest S

Input: A global control rule *Abstract*

Input: A local control rule *Unfold*

Output: A partial deduction for P and S, encoded by H_i

1: $i = 0$
2: $H_0 = \emptyset$
3: $S_0 = S$
4: **repeat**
5: $A_i = Select(S_i)$
6: $A'_i = \mathit{Abstract}(H_i, A_i)$
7: $\tau_i = \mathit{Unfold}(P, A'_i)$
8: $H_{i+1} = H_i \cup \{\langle A_i, A'_i \rangle\}$
9: $S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall \langle B, \_ \rangle \in H_{i+1} \,.\, B \not\equiv A\}$
10: $i = i + i$
11: **until** $S_i = \emptyset$

determine the version to use at each program point. Since code generation produces a new predicate name per entry in H, independence is guaranteed, and thus the specialized program will not produce more solutions than the original one. The ECCE system [Leu02] can be made to behave as Algorithm 2 by setting the *parent abstraction* flag to *off*.

16 Poly-Controlled Partial Deduction

As we have seen, in the greedy algorithm given a configuration $\langle S_i, H_i \rangle$, and once we decide to continue the computation using $A_i \in S_i$, there only one successor configuration which is $T(\langle S_i, H_i \rangle)$. However, it is well known that several control strategies exist which can be of interest in different circumstances. It is indeed a rather difficult endeavour to find a pair of global control and local control rules which behaves well in all settings. Thus, rather than considering a single global control and local control rule, at least in principle one can be interested in applying *different* local and global control rules to *different* atoms. Unfortunately this is something which existing algorithms for partial deduction do not cater for.

If we allow different combinations of global and local control rules, given a configuration,

there is no longer a single successor in the computation of the algorithm but possibly several ones. In fact, given a set of unfolding rules $\mathcal{U} = \{\mathit{Unfold}_1, \ldots, \mathit{Unfold}_i\}$ and a set of abstraction functions $\mathcal{G} = \{\mathit{Abstract}_1, \ldots, \mathit{Abstract}_j\}$, there are $i \times j$ possible combinations.

Algorithm 3 Poly-Controlled Partial Deduction algorithm

Input: Program P
Input: Set of atoms of interest S
Input: Set of unfolding rules $\mathcal{U}$
Input: Set of generalization functions $\mathcal{G}$
Input: Selection function $Pick$
Output: A partial deduction for P and S, encoded by H_i

1: $i = 0$
2: $H_0 = \emptyset$
3: $S_0 = S$
4: **repeat**
5: $\quad A_i = Select(S_i)$
6: $\quad \langle \mathit{Abstract}, \mathit{Unfold} \rangle = Pick(A_i, H_i, \mathcal{G}, \mathcal{U})$
7: $\quad A'_i = \mathit{Abstract}(H_i, A_i)$
8: $\quad \tau_i = \mathit{Unfold}(P, A'_i)$
9: $\quad H_{i+1} = H_i \cup \{\langle A_i, A'_i, \mathit{Unfold} \rangle\}$
10: $\quad S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall \langle B, \_, \_ \rangle \in H_{i+1} \,.\, B \not\equiv A\}$
11: $\quad i = i + 1$
12: **until** $S_i = \emptyset$

Algorithm 3 shows a *poly-controlled* partial deduction algorithm. We refer to this algorithm as poly-controlled because it allows the use of multiple control strategies and use different ones for different atoms. The choice of the control strategy to apply during the handling of each atom is performed by the $Pick$ function which given an atom A_i, a history H_i, and a set of global control rules and a set of local control rules, picks up a pair $\langle \mathit{Abstract}, \mathit{Unfold} \rangle$ among all possible ones. This algorithm differs from the greedy algorithm seen in Section 15 in several ways. One is that rather than receiving as input an abstraction function and an unfolding rule, it receives *a set* of global control rules and *a set* of local control rules. Another difference is that the tuples in set H_i now contain not only an atom and the result of abstracting it, but also the unfolding rule which has been picked for unfolding such atom. This is needed in order to use exactly such unfolding rule during the code generation phase. Indeed, the specialized program corresponds

to $\bigcup_{\langle A,A',Unfold\rangle \in H_n} resultants(A', Unfold)$, where the function *resultants* is now parametric w.r.t. the unfolding rule. This allows applying the same unfolding rule during unfolding as it was applied during the execution of Algorithm 3. The third and final difference corresponds to the addition of the non-deterministic function *Pick* used in line 6, and whose behaviour has already been described above.

Clearly, different choices for the *Pick* function will result in different specialized programs. It is important to note that the finer-grained control of poly-controlled partial deduction can potentially produce specialized programs which are hard or even impossible to obtain by using off-the-shelf control strategies. Also, the addition of the *Pick* function conceptually makes the poly-controlled partial deduction algorithm being composed of three levels of control, the local control, the global control, and the *search control*, which is determined by the function *Pick*. Note that the inclusion of the history as an input argument to *Pick* allows to make informed decisions and not only random ones.

17 A motivating Example

We now show in Listing 10 a program which defines the predicate `main/3` which contains calls to the predicates `exp/3` and `p/1` defined as before:

```
main(A,B,C):- exp(B,2,Result), p(A).
```

Listing 10: A motivating example

If we specialize this program w.r.t. the query `main(A,B,C)` using leftmost unfolding, the residual code we get is:

```
main(A,B,C) :-
        D is 1*B,
        exp_ac_1(1,B,D,C),
        p_1(A).

exp_ac_1(1,A,B,C) :- D is B*A, exp_ac_2(0,A,D,C).

exp_ac_2(0,_1,A,A).

p_1(A) :- B is A+1, q_1(B).

q_1(1).
q_1(2).
q_1(3).
q_1(4).
q_1(5).
q_1(6).
```

Listing 11: Result with leftmost unfolding

Note that none of the calls to the built-in predicate `is/2` is sufficiently instantiated to be executed at specialization time. Since only leftmost unfolding is allowed, the unfolding trees computed are not very deep, resulting in a large number of residual predicates. On the other hand, if we choose to enable non-leftmost unfolding, the residual program we obtain is:

```
main(A,B,C) :- D is 1*B, C is D*B, 1 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 2 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 3 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 4 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 5 is A+1.
main(A,B,C) :- D is 1*B, C is D*B, 6 is A+1.
```

Listing 12: Result with leftmost unfolding

where only an SLD tree has been required, and thus no auxiliary predicates are defined. Unfortunately, neither the program in Listing 11 nor the one in Listing 12 is optimal. This is because, in order to achieve an optimal result, non-leftmost unfolding should be used for atoms for predicate `exp/3`, but only leftmost unfolding should be used for atoms for predicate `p/1`, as already discussed above.

18 Searching for all Specializations

As we have seen, the poly-controlled algorithm can provide better specializations that those achievable by traditional partial deduction algorithms by assigning different control strategies to different atoms. However, the improvements achieved rely on the behaviour of the function *Pick*. Unfortunately, choosing a good *Pick* function can be a very hard task. Another alternative is, instead of deciding *a priori*, the control strategy to apply to each atom, to generate several candidate partial deductions and then decide *a posteriori* which specialized program to use. This can be done by computing all possible combinations of global and local control rules and explore the whole search space in order to generate not only a specialized program but rather a collection of specialized programs.

Algorithm 4 shows an all-solutions search-based algorithm. In this case, there is no longer a single successor configuration state for each atom to unfold, but several of them. This can be interpreted as, given $\mathcal{G}$=$\{A_1, \ldots, A_j\}$ and $\mathcal{U}$=$\{U_1, \ldots, U_i\}$, we now have a *set* of transformation operators $T_{U_1}^{A_1}, \ldots, T_{U_i}^{A_1}, \ldots, T_{U_i}^{A_j}$. Obviously, in general we will be interested in selecting only one specialized program out of all final programs obtained. However, doing this a posteriori allows to make much more informed decisions.

The main difference of this algorithm w.r.t. Algorithm 3 is that there are now two additional data structures. One is *Confs*, which contains the configurations which are currently being explored. Another one is *Sols*, which stores the set of solutions found by the algorithm. As it is well

Algorithm 4 All-candidates Search-based Partial Deduction algorithm

Input: Program P

Input: Set of atoms of interest S

Input: Set of unfolding rules $\mathcal{U}$

Input: Set of generalization functions $\mathcal{G}$

Output: Set of partial deductions *Sols*

1: $H_0 = \emptyset$
2: $S_0 = S$
3: create(*Confs*); *Confs* = push($\langle S_0, H_0 \rangle$, *Confs*)
4: *Sols* = $\emptyset$
5: **repeat**
6: $\langle S_i, H_i \rangle$ = pop(*Confs*)
7: $A_i = Select(S_i)$
8: $Candidates = \{\langle Abstract, Unfold \rangle \mid Abstract \in \mathcal{G}, Unfold \in \mathcal{U}\}$
9: **repeat**
10: $Candidates = Candidates - \{\langle Abstract, Unfold \rangle\}$
11: $A'_i = Abstract(H_i, A_i)$
12: $\tau_i = Unfold(P, A'_i)$
13: $H_{i+1} = H_i \cup \{\langle A_i, A'_i, Unfold \rangle\}$
14: $S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall \langle B, \_, \_ \rangle \in H_{i+1} \,.\, B \not\equiv A\}$
15: **if** S_{i+1}=$\emptyset$ **then**
16: *Sols* = *Sols* $\cup \{H_{i+1}\}$
17: **else**
18: push($\langle S_{i+1}, H_{i+1} \rangle$,*Confs*)
19: **end if**
20: **until** $Candidates = \emptyset$
21: $i = i + 1$
22: **until** empty_stack(*Confs*)

known, the use of different data structures for *Confs* provides different traversals of the search space. Currently, Algorithm 4 uses a stack, which means that the search space will be traversed in a depth-first fashion. Now the algorithm does not work with single configurations but rather with stacks of configurations. The process terminates when the stack of configurations to handle is empty, i.e. all final configurations have been reached.

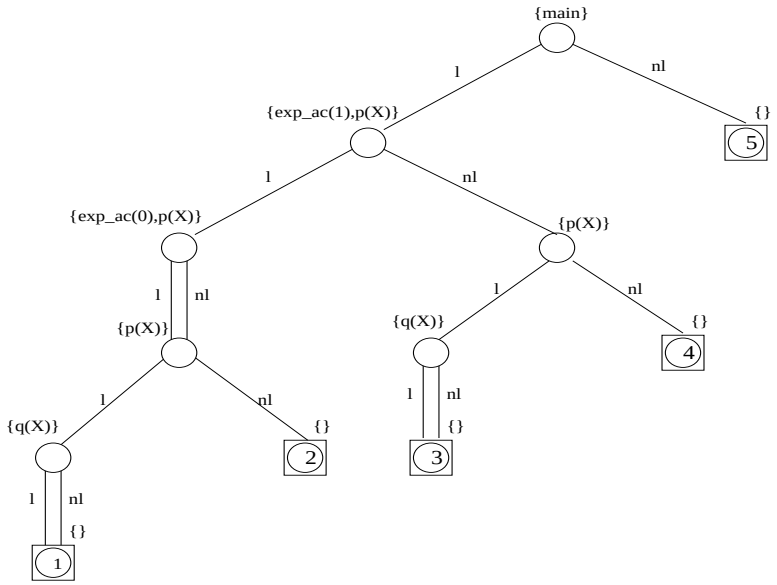

Figure 9: Search space for the motivating example

19 Searching for All Specializations in our Motivating Example

Consider again the motivating example in Listing 10. Consider also two local control rules, one which performs leftmost unfolding only, and the other one which also performs non-leftmost, i.e., $\mathcal{U}$=$\{$*leftmost*,*nonleftmost*$\}$ and one global control rule *id* which always returns the same atom, i.e., $\mathcal{G}$=$\{id\}$. By applying our all solution poly-controlled Algorithm 4 we get five different specialized programs. In particular, *Solution1* corresponds to the program in Listing 11 and *Solution5* to the program in Listing 12. In addition, our algorithm also produces three other candidate programs which are not easily achieved automatically unless poly-controlled specialization is used.

The search space for this example is shown in Figure 9. There, each configuration is represented with a circle. Configurations which are final are marked with a square around the circle. As can be seen, the whole search space for the example consists of 12 configurations, 7 of which are not final and 5 are final, and thus correspond to different candidate solutions, as already mentioned. Each configuration is adorned with the set of atoms yet to be handled, i.e., S_i in the algorithm. Each node can have two descendants, which are indicated with arcs. Arcs are labeled either `l`, for *leftmost* or `nl` for *non-leftmost*. The set of nodes already handled is not shown explicitly in each node, but it is implicitly represented by traversing the tree from each node upwards up to the root, since an atom is handled in each node. For example, in the case of *Solution3*,

Program	Run Time	Code Size	Speedup	Code Reduction
Original	5890	1606	1.00	1.00
Solution1	3652	1596	1.61	1.01
Solution2	5138	1543	1.15	1.04
Solution3	2931	1379	2.01	1.16
Solution4	3962	1326	1.49	1.21
Solution5	7223	1321	0.82	1.22

Table 5: Comparison of Solutions

the history is $\{\langle q(B), q(B), nl\rangle, \langle p(A), p(A), l\rangle, \ \langle exp\_ac(1, A, B, C), exp\_ac(1, A, B, C), nl\rangle, \langle main(A, B, C), main(A, B, C), l\rangle\}$. Also, some nodes only have one descendant linked by two arcs to its parent. This indicates that the two control strategies considered produce equivalent configurations. This allows reducing the search space and is discussed in more detail in Section 21 below.

Table 5 provides a comparison of the different candidate solutions together with the original program. The first column indicates the program we refer to in each row. The second column provides an indication of the run-time efficiency of the different programs. Since the overall execution time of this program is very small, and in order to obtain measurements which make sense, this time has been obtained by running the query `main(8,9,Result)` a million times and subtracting the time required by an empty loop which performs a million iterations. The third column compares the sizes of the different programs. This size is in number of bytes of the program compiled into byte-code using Ciao-1.11 and after subtracting the size of an empty program. Finally, the last two columns compare the run-time and code-size of the different programs with that of the original program.

As it can be seen, not all programs obtained by partial deduction are necessarily faster than the original one. In particular, *Solution5*, the one obtained using non-leftmost unfolding for all cases is less efficient than the original one. This is indicated by an speedup lower than 1, which is 0.82 in this case. On the other hand, the speedup obtained by *Solution1* is 1.61, but it is still far from the fastest program, which is *Solution3* with an speedup of 2.01. As regards code size, in this particular case all solutions achieved are smaller than the original program, though it is well-known that in some cases partial deduction can produce programs which are significantly larger than the original one. The smallest program is *Solution5*, with a code reduction of 1.22, but which happens to be the slowest program of all, including the original one.

If both the speedup and code reduction factors are taken into account, the most promising programs are *Solution3* and *Solution4*, neither of which are achievable by using one unfolding

rule for all atoms. If code size is not a very pressing issue, then *Solution3* is probably the best one, but otherwise *Solution4* should be used, since a relative small increase in program size provides significant time performance improvement. The choice between the two solutions mentioned will depend on the fitness function used, which can put more emphasis in one factor or another.

20 Self-Tuning, Resource-Aware Partial Deduction

Though Algorithm 4 can be used to automatically generate a large number of candidate specialized programs to choose from, we need some mechanism to select just one of them since the goal of partial deduction is to obtain a specialized program, not many.

There are obviously several criteria which can be used to decide how good a specialized program is. Possibly, the most obvious one is the time-efficiency of the specialized program. However, there are some complications associated to evaluating such time-efficiency. There are several possibilities which could be taken into account. One can be the use of static cost analysis. Cost analysis can aim at obtaining upper or lower bounds on computational cost or even average cost. Each of these three possibilities are of interest, and can be applied in different situations. Yet another possibility is to measure run-times. This approach, though simpler, has the problem that we need a set of test cases which are representative of the class of run-time queries which will be performed.

Another factor which we are probably interested in considering is the size of the resulting programs. This feature of programs is fairly easy to measure. However, depending on the resources available in the platform in which the specialized code will run, this can be of higher or lower importance. Also, even in cases where code size is not much of an issue, it can happen that different specialized programs have similar time-efficiency but some of them can be significantly larger than others.

Finally the memory-consumption of the specialized code is also an issue, especially if the program will run on devices with limited resources, as is the case in embedded systems and pervasive computing. Similarly to the case of time-efficiency, both static and dynamic approaches can be used, with similar pros and cons as discussed above for time-efficiency. The framework we propose in this work is *resource-aware* since it can take all the above criteria into account.

Another goal of our framework is that it should be fully automatic. I.e., there should be no need for human intervention in order to decide which of the candidate specializations is the best. We refer to this as a *self-tuning* approach. As we will see later on, our framework currently uses test cases in order to obtain information about the quality of the specialized programs. This information is then summarized by a *fitness function* which assigns a numeric value to each can-

didate specialization and which reflects how good the corresponding program is. The framework is parametric w.r.t. the fitness function so that the method can be applied with different aims in mind. Some times we may be interested in achieving code which is as time-efficient as possible, whereas in other cases memory-efficiency can be a primary aim. It is important to note that this search-based approach to partial deduction is also of interest when only run-time is taken into account. Even in such case there is no control strategy alone which is guaranteed to always produce the most-efficient code for all compilers and architectures.

21 Speeding up Search in Poly-Controlled Partial Deduction

Though the search-based approach presented in Section 18 above is definitely appealing, it can present important drawbacks in practice. One is that even for relatively small programs, the number of candidate programs can be too large to make the approach effective in practice. One first obvious optimization is to eliminate configurations which are direct descendants in the search tree from the same node and which happen to be equivalent. I.e., it will relatively often be the case that given a configuration *Conf* there are more than one T_U^A and $T_{U'}^{A'}$ with $(A, U) \neq (A', U')$ s.t. $T_U^A(Conf) = T_{U'}^{A'}(Conf)$. This optimization is easy to implement, not very costly to execute, and reduces search space significantly. For example, in the search space in Figure 9, which already includes this optimization, if this optimization were not applied then it would contain 19 configurations, instead of 12 and there would be 9 candidate solutions instead of 5.

In addition to the elimination of redundant configurations, and as is often done in many search-based algorithms, in order to reduce the search space, one would like to be able to prune away branches which are not promising. For this, we need to be able to apply the fitness function not only on configurations which are final, but also on intermediate configurations. The problem here is that the partial evaluation algorithm is not devised in such a way that a consistent program can be obtained for any set of atoms. If such set of atoms is not closed, then the union of the partial deductions for the atoms in the set does not correspond to a self-contained program.

The solution we propose in this case is the use of an additional function *close*, which given any configuration deterministically produces a new configuration which is closed and which uses the original definitions of the corresponding procedures for all atoms not handled yet by the algorithm.

Interestingly, this function can be achieved as a particular case of Algorithm 3 by using $\mathcal{G}$= $\{dynamic\}$ and $\mathcal{U}$= {*one-step*}. The abstraction function *dynamic* abstracts away the value of all arguments and the unfolding rule *one-step* performs just one unfolding step. Once the close function terminates, the set of atoms obtained is closed and we can apply the usual code

generation phase, as presented in Section 16 to achieve the specialized program. This allows applying the evaluation function to all configurations in the search space, and not only those which are final.

21.1 Search Strategy

Once we have the possibility of generating code and applying the fitness functions to non-final configuration, the frequency with which we apply the *close* function and evaluate configurations is an open issue. Also, the number of *candidate* configurations which we may decide to have simultaneously is also open.

It is important to note that the process of applying the fitness function to a configuration can help prune branches and thus reduce the search space to be explored, it will in general be a costly task since it involves generating code, compiling, and running the program for the given set of test cases. There is thus a clear trade-off between how often we apply the fitness function and the number of nodes explored. By applying the fitness function too often, can even slow-down the search process, since the time taken to close configurations and evaluate them can actually be higher than the time needed to further expand the configuration and apply the fitness function only in the final states reachable from such configuration. Also, pruning non-final configurations from the search tree can avoid finding the optimal program possible, since the fitness function is not guaranteed to be increasing nor decreasing. It can be the case that a configuration with low value in the fitness function can lead to a final configuration which is optimal.

Our implementation is parametric w.r.t. two values. One is the number of levels in the search tree which should be expanded before applying the fitness function. The other is the maximum number of configurations which can remain "active" after a selection process. Note that this number does not necessarily correspond to the maximum number of candidates active in any point in time. It only affects states in depth-levels as indicated by the depth limit.

21.2 Reducing the Branching Factor

In spite of the possibility of eliminating redundant configurations and non-promising branches, an interesting possibility which is worthwhile to explore in practice is the use poly-controlled partial deduction with more restrictive capabilities in order to reduce the cost of exploring the search space. For example, rather than allowing different local and global control rules for different atoms with the same predicate symbol, we can also restrict ourselves to configurations which always use the same unfolding and abstraction rule for all atoms of a predicate. This restriction will greatly reduce the branching factor of our algorithm since, handling of an atom

A_i will become deterministic as soon as we have previously considered an atom for the same predicate: it is compulsory to use exactly the same local and global control as before.

Though this simplification may appear too restrictive, it is also often the case that though it can be interesting to use different control rules for different predicates, it is likely that the same control rule will behave well for all instances of a predicate. In this case, it is important that we do not prune too early, since otherwise the fact that we have applied a particular control strategy in an atom for the predicate can force the application of the same control strategy all over the descendants in the search tree from that configuration. An intermediate solution which is worth exploring is to define control at the level of *modes* for a predicate. This means that two calls to a predicate with the same instantiation level in their arguments have to use the same control strategy, but not if they have different instantiation patterns.

22 Branch and Bound

A well known technique for pruning configurations which definitely cannot improve over existing solutions is known as *Branch and Bound*. This technique involves the use of two components for estimating the fitness of an intermediate configuration. One which is actual and another which maximizes or minimizes the possible value in case of completing the configuration. Since in our case we aim at maximizing the value of the fitness function, the approximation has to provide an upper bound over the incomplete part of the configuration. If the combination of such upper bound and the current fitness of the configuration are lower than some of the existing solutions, then there is no point in further expanding the given configuration.

It is important to note that the original program is a final configuration, so we can take the value of the fitness function over the initial program as a lower bound of the value which the fitness function will take in the best solution found. In addition, and as the search continues, further final configurations will be found which will possibly increase the fitness value of the best solution so far.

If the fitness function takes the size of the resulting program as one factor, we can take for a configuration *conf*=$\langle S, H\rangle$ then we can take as current value the size of the resultant in the atoms in H and take as estimate zero for the atoms in S plus any further atom added by the *close* function.

When taking execution time as a factor one possibility is to use a profiler such as that described in Deliverable D16, which allows defining *cost centers*. The profiler splits the total execution time among the different predicates in the program. When a cost center is defined, it accumulates the execution time of all computations started from such predicate. Thus, given a

configuration $\langle S, H\rangle$ by defining a cost center in every atom in *Close*$(\langle S, H\rangle) = C_P$ which is not already in H, when running the C_P, the time reported by the profiler for predicates in H does not include the time actually required for atoms which are not in H. If the total time taken by predicates related to atoms in H is higher than the best time already found, again there is no point in further expanding this node. A similar reasoning could be applied for memory-consumption of the specialized program, though our profiler does not currently provide the required support.

23 Discussion and Future Work

The poly-controlled partial deduction framework opens up the door to many interesting possibilities. As we have seen in our motivating example, it can achieve specialized programs which are difficult to find otherwise given a set of existing control strategies.

However, it can be very costly unless good search strategies are used. In Sections 21 and 22 we have proposed several ways of reducing the cost of traversing the search space. In addition to this, a relatively simple idea but which, to the best of our knowledge, is not exploited in actual partial evaluation systems is that different procedures have different relative importance on the overall time efficiency of the program. Thus, it can be a good idea to obtain data on the cost of the different procedures by means of profiling in order to be able to make more informed decisions at partial evaluation time.

For example, for procedures with little impact on the run-time of the program, less aggressive control strategies can be used, whereas in calls to predicates with an important cost, more aggressive strategies should be used. Also, the branching factor could be varied for different atoms according to the importance of the atom being handled. If the atom has important weight, we should probably try out more different alternatives than in other less important predicates.

Our implementation of poly-controlled partial deduction seems to provide very promising results. It remains as future work to experimentally try poly-controlled partial deduction on a larger set of programs, using different search control and evaluate thoroughly what the cost and benefits of the proposed approach are. Also, in a context in which a program can be repeatedly specialized for different call patterns it seems feasible to reuse previous results of poly-controlled specialization in order to help us reduce the branching factor. This can be achieved by storing the best control strategy found for previous atoms, i.e., the ones which are selected in the program which is chosen by the fitness function. This idea, combined with the restriction of having just one control strategy per instantiation pattern for a predicate, as discussed above, seem most promising in order to implement a system which can efficiently specialize different initial call patterns for the same program.

References

[AAV01] 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)*, LNCS 2042, pages 103–124. Springer-Verlag, 2001.

[APG05] E. Albert, G. Puebla, and J. Gallagher. A Partial Deducer Assisted by Predefined Assertions and a Backwards Analyzer. In *5th International Workshop on the Implementation of Logics (WIL'04)*, March 2005.

[AV01] Elvira Albert and Germán Vidal. Source-Level Abstract Profiling for Multi-Paradigm Declarative Programs. In *Proc. of 11th Int'l Workshop on Logic-based Program Synthesis and Transformation, LOPSTR'2001*, 2001.

[BG95] 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.

[BHH+04] 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.

[BSM92] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding during Partial Deduction. *New Generation Computing*, 1(11):47–79, 1992.

[CLGH04] 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.

[ES03] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, 2003.

[Gal91] John Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[Gal93] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[GB91] John Gallagher and Maurice Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

[Gur94a] 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.

[Gur94b] 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.

[JLM96] 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.

[LB02] 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.

[Leu98] Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[Leu02] Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.

[LJVB04] 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, 2004.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[LMDS98] 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.

[LS91] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[MD96] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.

[MG95] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Japan, June 1995. MIT Press.

[Pre92] Steven Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[Sah93] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[SG95] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479. The MIT Press, 1995.

[STK97] 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.

[VD88] Raf Venken and Bart Demoen. A partial evaluation system for Prolog: Theoretical and practical considerations. *New Generation Computing*, 6(2 & 3):279–290, 1988.

[Vid04] G. Vidal. Cost-Augmented Partial Evaluation of Functional Logic Programs. *Higher-Order and Symbolic Computation*, 17(1-2):7–46, 2004.

[WPD01] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1–2):3–35, 2001.