Supervising Offline Partial Evaluation of Logic Programs using Online Techniques*

Michael Leuschel, Stephen-John Craig and Dan Elphick

Institut für Informatik, Universität Düsseldorf
D-40225, Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de

Abstract. A major impediment for more widespread use of offline partial evaluation is the difficulty of obtaining and maintaining annotations for larger, realistic programs. Existing automatic binding-time analyses still only have limited applicability and annotations often have to be created or improved and maintained by hand, leading to errors. We present a technique to help overcome this problem by using online control techniques which supervise the specialisation process in order to detect such errors. We discuss an implementation in the LOGEN system and show on a series of examples that this approach is effective: very few false alarms were raised while infinite loops were detected quickly. We also present the integration of this technique into a web interface, which highlights problematic annotations directly in the source code. A method to automatically fix incorrect annotations is presented, allowing the approach to be also used as a pragmatic binding time analysis. Finally we show how our method can be used for efficiently locating errors with built-ins inside Prolog source code.

1 Introduction

Partial evaluation [11] is a source-to-source program transformation technique which specialises programs by fixing part of the input of some source program P and then pre-computing those parts of P that only depend on the fixed part of the input. The so-obtained transformed programs are less general than the original but often more efficient. The part of the input that is fixed is referred to as the *static* input, while the remainder of the input is called the *dynamic* input. The research into controlling partial evaluation can be broadly partitioned into two schools of thought: the *offline* and the *online* approach. In the online approach all control decisions (i.e., deciding which parts of the input are static and which parts of the program should be pre-computed) are made online, during the specialisation process. The idea of the offline approach is to separate the specialisation process into two phases (cf. Fig. 1):

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

* This research has been carried out as part of the EU funded project IST-2001-38059 ASAP (Advanced Specialization and Analysis for Pervasive Systems).

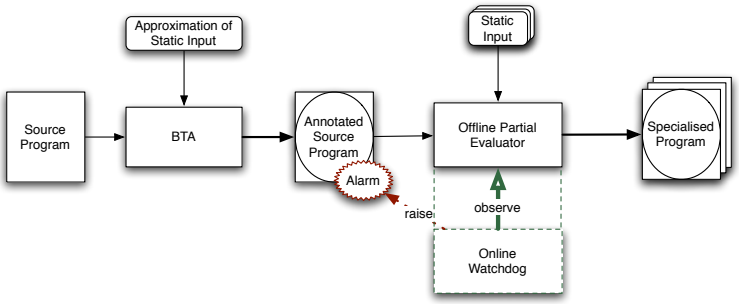

Fig. 1. Offline Partial Evaluation and the new watchdog mode

– A (simplified) *specialisation phase*, which is guided by the result of the BTA.

A short summary of the advantages and disadvantages of offline specialisation wrt to online specialisation is as follows:

– The offline approach is in principle less precise (see, however, [6]) as it has to make decisions before the actual static values are known.
– The offline approach leads to simpler specialisers, thus making self-application easier and leading to more efficient specialisation. Especially the specialisation phase proper (i.e., after the BTA has been performed) is usually considerably faster than specialisation using an online specialiser. This is relevant in situations where the same source program is re-specialised multiple times w.r.t. the same approximation of the static data.
– The offline approach is more predictable, as it is relatively clear from the annotation which parts of the code will be pre-computed. This also means that it is easier to tune the offline approach by editing the annotations.

An offline system for logic programming is the LOGEN system [19]. LOGEN has successfully been applied to non-trivial interpreters, and can be used to achieve Jones-optimality [23]. for a variety of interpreters [17], i.e., completely removing the interpretation overhead.[1] As such, LOGEN is of potential interest for many logic programming areas and applications; for example, LOGEN has been applied to optimise access control checks in deductive databases [2], to compile denotational semantics language specifications into abstract machine code [31], or to pre-compile Object Petri nets for later model checking. However, the learning curve for LOGEN is still considerable and the LOGEN system has up until now still proven to be too difficult to be used by non-experts in partial evaluation. The main difficulty lies in coming up with the correct annotations (and then maintaining them as the source program evolves). Indeed, while some errors (i.e., annotating an argument as static even though it is dynamic) can be easily identified by various abstract interpretation schemes (see, e.g., [5, 8]), ensuring

[1] Achieving this predictably for a variety of interpreters using online approaches is not yet fully understood; see, however, [30].

termination of the specialisation process is a major obstacle. Recent work has led to a fully automatic BTA [8], but unfortunately the BTA still only provides partial termination guarantees[2]; is sometimes overly conservative, especially for the more involved (and more interesting) applications; and can be too costly to apply for larger, real-life Prolog programs. Finally, the BTA of [8] does not yet deal with many of Prolog's built-ins and non-logical control constructs.

In this paper we present a way to tackle and solve this problem from a new angle. The main idea is to use online techniques to *supervise* an offline specialiser. The central idea is that the user can turn on a *watchdog* mode which activates powerful online control methods to supervise the offline specialiser (see Fig. 1). If the online control detects a potential infinite loop (or some other problem such as incorrectly calling a built-in) an *alarm* is raised, helping the user to identify and fix errors in the annotation. This watchdog mode will obviously slow down the specialisation process, invalidating one of the advantages of the offline approach. However, it is the intention that this watchdog would only be activated in the initial development or maintenance phase of the annotation or when an error (e.g., apparent non-termination) arises: it is not our intention to have the watchdog mode permanently enabled (in that case an online partial evaluator would be more appropriate). In this paper we formally develop this idea, present an implementation inside the LOGEN system [19] and evaluate its performance on a series of examples. We show that on most correct annotations no false alarms are raised, while on incorrect annotations the problems are spotted quickly and useful feedback is given. We also present a web interface that can further help the user to quickly spot and automatically fix the problems identified by the watchdog. We thus hope that this new technique will make it possible for users to quickly find errors in their annotations. This hope is underpinned by several initial case studies within the ASAP project.

2 Offline Partial Evaluation

We now describe the process of offline partial evaluation of logic programs. Throughout this paper we suppose familiarity with basic notions in logic programming. We follow the notational conventions of [22]. Formally, evaluating a logic program P for an atom A consists in building a so-called *SLD-tree* and then extracting the *computed answer substitutions* from every non-failing branch of that tree. Take for example the following program to match a regular expression against a (difference) list of characters:

```
re(empty,T,T).                          re(ch(X),[X|T],T).
re(or(X,Y),H,T) :- re(X,H,T).           re(or(X,Y),H,T) :- re(Y,H,T).
re(star(X),T,T).
re(star(X),H,T) :- re(X,H,T1),re(star(X),T1,T).
re(cat(X,Y),H,T) :- re(X,H,T1),re(Y,T1,T).
```

[2] [9] does provide full termination guarantees for functional programs but is not available in a running system and does seem not cope very well with interpreters.

As an example, the SLD-tree for `re(star(ch(a)),[C],[])` is presented on the left in Fig. 2. The underlined atoms are called selected atoms. Here there is only one branch, and its computed answer is `C = a`.

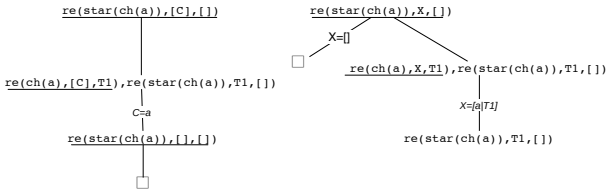

Fig. 2. Complete and Incomplete SLD-trees for the regular expression program

Partial evaluation (also sometimes called partial deduction) for logic programs proceeds by building possibly *incomplete* SLD-trees, i.e., trees in which it is possible *not* to select certain atoms. The right side of Fig. 2 contains such an incomplete SLD-tree, where the call `re(star(ch(a)),T1,[])` is not selected. Formally, partial evaluation builds a series of incomplete SLD-trees for a set of atoms $\mathcal{A}$ that is chosen in such a way that all unselected leaf atoms (such as `re(star(ch(a)),T1,[])`in Fig. 2) as well as all user queries of interest are an instance of some atom in $\mathcal{A}$. The specialised program is then extracted from those trees by producing one new specialised predicate for every atom in $\mathcal{A}$, with one clause constructed per non-failing branch. The arguments of the specialised predicate are the variables of the corresponding atom in $\mathcal{A}$. E.g., for $\mathcal{A}$ = { `re(star(ch(a)),X,[])`} and for the SLD-tree in Fig. 2, we would get:

```
re__0([]).              re__0([a|A]) :-  re__0(A).
```

Partial evaluation techniques for logic programs often start off with an initial atom A_0 of interest: $\mathcal{A} = \{A_0\}$. For every atom in $\mathcal{A}$ an SLD-tree is built, and then all unselected leaf atoms which are not an instance of an atom in $\mathcal{A}$ are added to $\mathcal{A}$. This is repeated until all unselected leaf atoms are an instance of some atom in $\mathcal{A}$. To ensure termination, generalisation techniques have to be applied; i.e., atoms in $\mathcal{A}$ may be replaced by a more general atom. The control of partial evaluation for logic programs is thus naturally separated into two components [24] (see also [16]): The *local control* controls the construction of the SLD-trees for the atoms in $\mathcal{A}$ and thus determines *what* the residual clauses for the atoms in $\mathcal{A}$ are. The process of constructing these trees is also called *unfolding*. The *global control* controls the content of $\mathcal{A}$, it decides *which* specialised predicates are present in the residual program and ensures that all unselected leaf atoms are an instance of some atom in $\mathcal{A}$.
In offline partial deduction the local and global control are guided by annotations. The LOGEN system [19] uses two kinds of annotations for this:

– *Filter declarations*, which declare which arguments (or subarguments) to which predicates are static and which ones dynamic. This influences the global control only. More precisely, for unselected leaf atoms the dynamic (sub-)arguments are replaced by fresh variables; it is then checked whether a variant of this generalised atom already exists in $\mathcal{A}$; if not the generalised atom is added to $\mathcal{A}$.
– *Clause annotations*, which indicate for every call in the body how that call should be treated during unfolding; i.e., it influences the local control only. For now, we assume that a call is either annotated by **memo** — indicating that it should not be selected – or by **unfold** — indicating that it should be selected. Built-ins (or predicates whose source is not available) can be annotated as either **call** — indicating that the call should be executed at specialization time — or as **rescall** — indicating that the call should not be executed at specialization time.

First, let us consider, e.g., an annotated version of the regular expression program above in which the filter declarations annotate the first and third arguments as static while the second one is dynamic: `:- filter re(static,dynamic,static).` Then let the clause annotations annotate the call `re(star(X),T1,T)` in the last clause as **memo** and all the other calls as **unfold**. Given a specialisation query `re(star(ch(a)),X,[])`, offline partial deduction would proceed as follows:

1. The atom `re(star(ch(a)),X,[])` is generalised by replacing the dynamic arguments by variables. In this case, the second argument is already a variable.
2. The generalised atom is added to $\mathcal{A}$ and then unfolded. This generates exactly the right SLD-tree depicted in Fig. 2.
3. The leaf atoms of the tree are again generalised and are added to $\mathcal{A}$ if no variant is already in $\mathcal{A}$. In this case there is only one leaf atom—namely `re(star(ch(a)),T1,[])`—whose second argument is again already a variable and a variant of which is already in $\mathcal{A}$. Thus no further unfolding is required.
4. The specialised code is produced by mapping each atom in $\mathcal{A}$ to a fresh predicate whose arguments are the variables of the atoms. In this case `re(star(ch(a)),X,[])` would be mapped to, e.g., `re__0(X)` resulting in the same specialised code as above:

```
re__0([]).              re__0([a|A]) :-  re__0(A).
```

3 Watchdog Mode

Below we show how offline partial evaluation can be supervised by online techniques, in order to identify non-terminating annotations. We first need the concept of a well-quasi order, which is used for many online techniques:

Definition 1. *A* quasi order $\leq_S$ *on a set* S *is a reflexive and transitive binary relation on* $S \times S$*. A sequence of elements* $s_1, s_2, \ldots$ *in* S *is called* **admissible with respect to** $\leq_S$ *iff there are no* $i < j$ *such that* $s_i \leq_S s_j$*. The relation* $\leq_S$ *is a* **well-quasi order (wqo)** *on* S *iff there are no infinite admissible sequences with respect to* $\leq_S$*.*

A widely used wqo is the homeomorphic embedding relation $\trianglelefteq$. The following is an adaptation of the definition from [28] (see, e.g., [14, 15] for a summary of its use in online control). In what follows, we define an *expression* to be either a term, an atom, a conjunction, or a goal.

Definition 2. *The* (pure) homeomorphic embedding *relation* $\trianglelefteq$ *on expressions is inductively defined as follows (i.e.* $\trianglelefteq$ *is the least relation satisfying the rules):*

1. $X \trianglelefteq Y$ *for all variables* X, Y
2. $s \trianglelefteq f(t_1, \ldots, t_n)$ *if* $s \trianglelefteq t_i$ *for some* i
3. $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ *if* $n \geq 0$ *and* $\forall i \in \{1, \ldots, n\} : s_i \trianglelefteq t_i$.

Notice that n is allowed to be 0 and we thus have $c \trianglelefteq c$ for all constant and proposition symbols. When $s \trianglelefteq t$ we also say that s is *embedded in* t or t is *embedding* s. By $s \triangleleft t$ we denote that $s \trianglelefteq t$ and $t \not\trianglelefteq s$. The intuition behind the above definition is that $A \trianglelefteq B$ iff A can be obtained from B by "striking out" certain parts. E.g., we have $p(0) \triangleleft p(s(0))$ and $f(a, b) \triangleleft h(f(g(a)), b)$.

For a finite set of function symbols, $\trianglelefteq$ is a well-quasi order, i.e., for every infinite sequence of expressions $s_1, s_2, \ldots$ there exists $i < j$ such that $s_i \trianglelefteq s_j$. This property has been used in various online control algorithms (first in [28] for supercompilation and then in [20] for partial evaluation of logic programs and then in various other techniques, e.g., [1]). Its main use is to ensure termination by stopping unfolding/specialisation when a new expression to specialise s_j embeds some earlier expression s_i of the specialisation history.

In the case of specialisation we know that the function symbols occurring within a given program (text) are finite. Thus for pure logic programs without built-ins, $\trianglelefteq$ is a well-quasi order for calls that can occur at runtime or at specialisation time. However, certain built-ins (such as `is/2` or `functor/3`) permit a program to generate an unbounded number of new function symbols. For this we employ the solution from [15, 20]: all function symbols not occurring within the original program text are classified as *dynamic* and we add the rule: $f(s_1, \ldots, s_n) \trianglelefteq g(t_1, \ldots, t_m)$ if f/n and g/m are dynamic function symbols.[3]

We now show how we have used $\trianglelefteq$ to act as a "watchdog" in offline specialisation which is used to supervise both the unfolding process and the memoisation. Let us first discuss the supervision of the local control. Suppose that we are constructing the SLD-tree for a given atom A. A simple solution would be, whenever an atom is unfolded, to check whether the sequence of selected literals starting from A up to (and including) the currently selected atom is admissible wrt $\trianglelefteq$.

However, it is well known ([4], see also [16]) in online partial evaluation of logic programs that examining the sequence of selected atoms does give rise to suboptimal techniques. Indeed, this sequence does not contain the information which selected atom actually descends from which other selected atom. This shortcoming can be remedied by working on the sequence of *covering ancestors* of the selected atom, i.e., only those atoms from which the selected atom descends

[3] It would be possible to refine this slightly by adding the requirement that there exists a subsequence of $t_1, \ldots, t_m$ which embeds the arguments to $s_1, \ldots, s_n$.

(via resolution). More formally, covering ancestors [4] can be captured in the following definitions.

Definition 3. *If a program clause $H \leftarrow B_1, \ldots, B_n$ is used in a derivation step with selected atom A then, for each i, A is the* **parent** *of the instance of B_i in the resolvent and in each subsequent goal where an instance originating from B_i appears (up to and including the goal where B_i is selected). The* **ancestor** *relation is the transitive closure of the parent relation. Let G_0, G_1, ..., G_n be an SLD-derivation with selected atoms A_1,A_2, ..., A_n. The* **covering ancestor sequence of** *A_i, a selected atom, is the maximal subsequence A_{j_1}, A_{j_2}, $\ldots A_{j_m} = A_i$ of A_1, A_2, ..., A_i such that all atoms in the sequence have the same predicate symbol and,$\forall 1 \leq k < m$ it holds that A_{j_k} is an ancestor of $A_{j_{k+1}}$.*

For every atom that is unfolded the supervisor will check whether the covering ancestor sequence of the selected atom is admissible wrt $\trianglelefteq$. If it is then specialization will proceed normally. Otherwise, an alarm will be raised: e.g., a warning message will be printed and the specialization process will suspend, allowing the user to choose between aborting or continuing the specialisation process. For example, in the SLD-tree on the left in Fig. 2, when selecting the call `re(star(ch(a)),[],[])`) the watchdog will check whether the sequence $\langle$`re(star(ch(a)),[C],[]) re(star(ch(a)),[],[])`$\rangle$ is admissible wrt $\trianglelefteq$. As it is admissible, no alarm is raised.

Let us now examine the global control, which builds up the set $\mathcal{A}$ of atoms to be specialised. To achieve more refined control, the set $\mathcal{A}$ is often structured as a tree [20, 24], called a *specialisation tree*. Basically, if after unfolding some atom A_j we have to add one of the unselected leaf atoms A_k in the SLD-tree to the set $\mathcal{A}$, then we register A_k as a child of A_j in the specialisation tree. We can thus do the following for atoms annotated as **memo**: we first build up the global specialisation tree, i.e., when a call A_k gets memoed during unfolding of A_i, and A_k is not an instance of another atom that has already been specialised, then we add A_k as a child of A_i in the specialisation tree. Furthermore, we check whether the sequence of ancestors of A_k in the tree is admissible wrt $\trianglelefteq$. If it is not, we raise an alarm and allow the user to choose between aborting or continuing the specialisation process.

The Implementation We have integrated the above idea and technique into the LOGEN system. The LOGEN system uses the so-called "cogen" approach to specialisation, i.e., given an annotated source program it directly generates a specialised specialiser for this source program and the annotation (called a *generating extension*). In particular, for every clause of the source program LOGEN derives an "unfolder" clause in the generating extension, having an extra argument to compute the residual code. Similarly, memoisation predicates are constructed for the memoised predicates. To implement our watchdog mode the unfolder predicates do not carry enough information to determine whether unfolding the body literals is actually safe or not: we need access to the covering

ancestor sequence. For memoised calls, we additionally need the global specialisation tree. We have adapted the compilation strategy of LOGEN so that in watchdog mode an extra argument is maintained by the unfolder and memoisation predicates, where both the covering ancestor sequence and the specialisation tree are built up. In Section 4, we examine empirically whether our approach is efficient enough to be practical and precise enough to be useful.

Let us now use the watchdog on our regular expression example. First, we annotate all calls as unfold and run LOGEN from the command line with the watchdog mode enabled:

```
   %  logen re.pl "re(star(ch(a)),X,[])" -w
<| HOMEOMORPHIC WARNING |> : UNFOLDING re(star(ch(a)),A,[]),
History: [re(star(ch(a)),B,[])]
A predicate is possibly being unfolded infinitely often.
You may want to annotate the call as memo.
Type 'c' to continue unfold, 'F' to fail branch,
'C' to continue without further intervention, anything else to abort:
```

As can be seen, the watchdog has correctly spotted (after a few ms) that we were about to infinitely unfold the atom `re(star(ch(a)),X,[])`. If we now correct our annotation, as suggested, by memoing the last call to `re/2` in the last clause we get the following:

```
   %  logen re.pl "re(star(ch(a)),X,[])" -w
   /* re(star(ch(a)),A,[]) :-re__0(A). */
   re__0([]).
   re__0([a|A]) :- re__0(A).
```

I.e., this time the watchdog has not raised an alarm, and indeed our specialisation now terminates.

4 Experiments

In the first series of experiments we ran our watchdog technique on a series of *correctly annotated*, terminating examples. We have gathered some simple programs as well as a variety of successful applications of the LOGEN system documented in the literature. The purpose was twofold: first, test whether, despite the overhead, the approach is practical, and second, whether the number of false alarms is low enough for the approach to be useful.

The results of the experiments are summarised in Table 1. All experiments were run using Ciao Prolog 1.13 on a Macintosh PowerPC Dual G5, 2.7 GHz with 4.5 GB of RAM running Mac OS X 10.4. The cogen time in the second column is the time needed to run the cogen of the LOGEN system to generate the generating extension. Column three contains the same figure for the watchdog mode. The fourth column then contains the time needed to run the generating extension on a single specialisation query. Column five contains the same figure for the watchdog mode. The sixth column contains the number of alarms, and thus the number

Benchmark	Cogen	Cogen watch	Spec.	Spec. watch	False Alarms	Overhead One Shot	Overhead Just Spec.
match	43 ms	65 ms	1.4 ms	1.7 ms	0	1.50	1.21
transpose	44 ms	66 ms	0.5 ms	0.8 ms	0	1.50	1.60
ex_depth	45 ms	68 ms	1.7 ms	2.3 ms	0	1.51	1.35
inter_medium	48 ms	71 ms	0.4 ms	12.5 ms	0	1.73	31.25
vanilla_list	44 ms	67 ms	1.0 ms	1.3 ms	0	1.52	1.30
liftsolve	49 ms	74 ms	1.7 ms	72.3 ms	0	2.89	42.53
lambdaint	60 ms	92 ms	1.6 ms	11.7 ms	0	1.68	7.31
db_access	115 ms	145 ms	1.2 ms	10.8 ms	1	1.34	9.00
matlab	94 ms	146 ms	3.4 ms	40.0 ms	0	1.91	11.76
pascal	70 ms	107 ms	3.0 ms	27.3 ms	0	1.84	9.10
picsim	258 ms	390 ms	145.8 ms	1999.4 ms	0	5.92	13.71
Average					0.09	2.12	11.83

Table 1. The watchdog approach for correct annotations

of false alarms (as all annotations ensure termination). The overhead of the watchdog mode in "one-shot" situations (i.e., a single specialisation) is presented in column seven, while the overhead of the specialisation process without the cogen time is presented in the last column. The benchmark programs are as follows. First, `match` is the semi-naive pattern matcher from [13], specialised for the pattern `[a,a,b]`. `transpose` and `ex_depth` are also taken from [13], while `inter_medium` is taken from [8]. `vanilla_list` is a variation of the vanilla interpreter (see [17]) specialised for an object program which is in turn the same interpreter but with the append as object program. `liftsolve` is the interpreter for the ground representation from [13] specialised for the append program. `lambdaint` is the interpreter for a small functional language presented in [17], specialised for the Fibonacci function. `db_access` is the interpreter for access control from [2] specialised for a particular policy and query pattern (query Q4 in [2]). `matlab` is the interpreter for a subset of the Matlab language also used in [21], specialised for the factorial function. `pascal` is the denotational semantics interpreter for a small Pascal like language used in [31], specialised for a small Pascal program so as to obtain assembly like code. `picsim` is an emulator for the machine language of PIC processors written by Kim Henriksen and John Gallagher [10], specialised for a particular machine program (so as to extract analysis information by further abstract interpretation).

In summary the results are very satisfactory. The overhead on the specialisation is usually an order of magnitude (this is to be expected, as every unfolding step and memoisation step is supervised and checked against the history of unfoldings and earlier memoisations respectively), even though the overhead on the total time in "one-shot" situations (i.e. time for both the cogen and the specialisation) is often much less, e.g., 84 % for the `pascal` experiment or 50 % for the `match` benchmark. What is most encouraging, however, is the low number of false alarms: on only one of the experiments false alarms were raised, and even there only a single alarm was raised.

We now examine how our approach fares when the annotations are *erroneous* and do not ensure termination of the specialiser. This is probably a more typical use case, as the watchdog would usually be turned on in exactly those circumstances. It is, however, more difficult to present empirical data in that setting: the notion of overhead makes no sense as the offline approach does not terminate; it is also difficult to quantify the earliest possible moment when non-termination can be "detected." Still, we will try to show on a series of examples that our watchdog technique does find the problem and does so quickly.

In Section 3 we have already looked at a simple example. Let us now examine the behaviour of the watchdog method on some more realistic examples. Take the `ex_depth` interpreter used in the previous section, counting the depth of SLD-trees, but marking this time the depth argument as static rather than dynamic.[4] Termination is no longer guaranteed, and this is a common annotation mistake in offline partial evaluation. This is spotted quickly by our technique (after less than 5 ms):

```
  % logen ex_depth_nonterm.pl "solve([inboth(X,Y,Z)],0,Depth)." -w
<| HOMEOMORPHIC WARNING|> : MEMO Atom solve([member(A,B)],s(s(s(0))),C),
History: [solve([member(D,E)],s(s(0)),F),solve([member(G,H),member(I,J)],
 s(0),K), solve([inboth(L,M,N)],0,0)]
```

Let us now take the same annotation, but this time unfold all calls to `solve`. As above, this is correct from the point of view of binding times (i.e., all arguments marked as static are really static). However, termination is not guaranteed, something which our technique spots quickly (again after less than 6 ms):

```
   % logen ex_depth_nonterm_local.pl "solve([inboth(X,Y,Z)],0,Depth)." -w
 <| HOMEOMORPHIC WARNING |> : UNFOLDING solve([member(A,B)],s(s(s(C))),D),
History: [solve([member(E,F)],s(s(G)),H),solve([member(I,J),member(I,K)],
 s(L),M), solve([inboth(N,0,P)],Q,R)]
```

The problems above were spotted at a very early stage, where it is easy for the user to identify the causes. In both cases the specialiser without watchdog mode will not terminate. Finally, we have tried two bigger examples: the `lambdaint` and `pascal` interpreters from the previous subsection. In the former we annotated the `apply` construct and in the latter the `while` construct as unfoldable. After less than 25 ms and 60 ms and 9 and 11 unfolding steps respectively the problem was detected by the watchdog.

5 The Web Interface and Semi-Automatic Correction

Further Error Conditions In addition to non-termination there are various other errors that often arise in hand-crafted annotations. First, a common mistake is to annotate built-ins as **call** even though they are not guaranteed to be sufficiently instantiated at specialisation time. Another common mistake is to

[4] This required adapting one clause for the binding times to be correct.

make the filter declarations too narrow, so that not all memoised calls are covered by the filter declaration. We have extended our watchdog technique so that these conditions are detected. This means that all calls to built-ins are explicitly checked by the watchdog, and the filter errors are also caught and presented to the user.[5] Another common mistake relates to *backpropagation* of bindings [26] in the context of non-logical built-ins and connectives. Here the watchdog uses co-routining to detect those backpropagations.

Graphical Web interface In [18] we have presented a graphical web interface for the LOGEN system, which allows the user to edit annotations for a given Prolog program in a user friendly way: the Prolog program to be annotated is presented with comments and formatting intact and colour coding (as well as "mouse over" information) is used to display the annotation. The annotations can be edited using an intuitive point and click interface.

In order to make it easier for users to understand and act upon the feedback provided by the watchdog, it would make sense to provide the watchdog information by highlighting the problematic annotations directly in the source code frame of the web interface. For this we had to extend the scheme presented in Section 3, in that the generating extensions also need to keep track of program points (in addition to the covering ancestor sequence and the specialisation tree). This information (along with a description of the error) is then fed back in XML format to the web interface to locate the source of the error and highlight it. Once this framework was in place, it was possible to extend the XML format to convey further information, such as how to fix an incorrect annotation. Below we show how these suggestions can be computed, again using online control techniques. To use this information the web interface uses XSLT to translate LOGEN's XML suggestions into Javascript statements for fixing the annotations, which are executed if the user presses the "fix" button.

Correcting Annotations Let us first summarise the four classes of problems that our watchdog can catch, along with a summary of the fixes that can be applied. Note that there could be alternate ways to fix the problems below; e.g., by unfolding more user predicates to make more things static. Our underlying assumption here is that the user will progress from more aggressive annotations to less aggressive ones (with less calls marked as unfold and call).

- Problem 1: dangerous unfolding is detected by $\trianglelefteq$. Fix: mark this call as **memo**.
- Problem 2: a built-in is marked as **call** but is not sufficiently instantiated (or throws an exception). Fix: mark the built-in as **rescall**.
- Problem 3: a call marked as **memo** is not covered by its filter declaration. Fix: generalise the filter declaration to cover the call. Details are presented below.

[5] The filter errors are now actually also caught in normal mode as this extra checking does not incur a significant overhead.

– Problem 4: $\trianglelefteq$ has detected a potential infinite memoisation. Fix: generalise the filter declaration to throw part of the static information away.

For the first two entries the fix is straightforward; for the latter two the computation of the updated filter declarations is more subtle. There are various ways this could be achieved. Below we present solutions inspired by online control techniques.

We first need to recall some background on LOGEN's filter declarations: a filter declaration assigns every argument of every predicate a binding-type. A *binding type* is a generalisation of the classical binding-times (static, dynamic; see, e.g., [11]), making it possible to precisely specify which subarguments are static or dynamic (rather than having to declare the entire argument as either static or dynamic). LOGEN's binding types are expressed using the standard formalism employed by polymorphically typed languages (e.g. [27]). Formally, a *type* is either a *type variable* or a *type constructor* of arity $n \geq 0$ applied to n types. We presuppose the existence of three 0-ary type constructors: `static`, `dynamic`, and `nonvar`. These constructors are given a pre-defined meaning.

Definition 4. *A* type definition *for a type constructor c of arity n is of the form*

$$c(V_1, \ldots, V_n) \longrightarrow f_1(T_1^1, \ldots, T_1^{n_1}) \ ; \ \ldots \ ; \ f_k(T_k^1, \ldots, T_k^{n_k})$$

with $k \geq 1, n, n_1, \ldots, n_k \geq 0$ and where $f_1, \ldots, f_k$ are distinct function symbols, $V_1, \ldots, V_n$ are distinct type variables, and T_i^j are types which only contain type variables in $\{V_1, \ldots, V_n\}$.
A type system *Γ is a set of type definitions, exactly one for every type constructor c different from* `static`, `dynamic`, *and* `nonvar`.

From now on we will suppose that the underlying type system Γ is fixed. LOGEN also allows function symbols to be used as type constructors and we thus also suppose that every function symbol of arity n is also a type constructor of arity n, defined by $f(V_1, \ldots, V_n) \longrightarrow f(V_1, \ldots, V_n)$ in Γ. As an example, the parametric type `list(T)` can be declared as follows in LOGEN (following the notations of Mercury): `:- type list(T) ---> [ ] ; [T | list(T)].`

We define *type substitutions* to be finite sets of the form $\{V_1/\tau_1, \ldots, V_k/\tau_k\}$, where every V_i is a type variable and τ_i a type. Type substitutions can be applied to types (and type definitions) to produce *instances* in exactly the same way as substitutions can be applied to terms. For example, $list(V)\{V/\texttt{static}\} = list(\texttt{static})$. A type or type definition is called *ground* if it contains no type variables.

Definition 5. *We now define* type judgements *relating terms to types in Γ.*
– $t : \texttt{dynamic}$ *holds for any term t*
– $t : \texttt{static}$ *holds for any ground term t*
– $t : \texttt{nonvar}$ *holds for any non-variable term t*
– $f(t_1, \ldots, t_n) : c(\tau'_1, \ldots, \tau'_k)$ *if there exists a ground instance of a type definition in Γ which has the form $c(\tau'_1, \ldots, \tau'_k) \longrightarrow \ldots f(\tau_1, \ldots, \tau_n) \ldots$ and where $t_i : \tau_i$ for $1 \leq i \leq n$.*

Here are a few examples, using the type system Γ_1 above. First, we have $s(0) : \texttt{static}$, $s(0) : \texttt{nonvar}$, and $s(0) : \texttt{dynamic}$. Also, $s(X) : \texttt{nonvar}$, $s(X) : \texttt{dynamic}$ but not $s(X) : \texttt{static}$. A few examples with lists are: $[s(0)] : \mathit{list}(\texttt{static})$, $[X, Y] : \mathit{list}(\texttt{dynamic})$.

The following fixes problem 3 identified earlier, i.e., it computes a new binding type for an argument t which has incorrectly been assigned a binding type τ:

Definition 6. *Let t be a term and τ a binding type. $\mathit{tgen}(t, \tau) =$*
- *τ if $t : \tau$;*
- *$\texttt{dynamic}$ if $\neg(t : \tau)$ and t is a variable;*
- *$f(\mathit{tgen}(t_1, \tau_1), \ldots, \mathit{tgen}(t_k, \tau_k))$ if $t = f(t_1, \ldots, t_n)$ and $\tau = f(\tau'_1, \ldots, \tau'_k)$ and $\neg(t : \tau)$;*
- *$\texttt{nonvar}$ otherwise.*

For example, $\mathit{tgen}(s(0), \texttt{static}) = \texttt{static}$, $\mathit{tgen}(p(X), \texttt{static}) = \texttt{nonvar}$, and $\mathit{tgen}(X, \texttt{static}) = \texttt{dynamic}$. Also $\mathit{tgen}(p(s(X), 0), p(\texttt{static}, \texttt{static})) = p(\texttt{nonvar}, \texttt{static})$, and $\mathit{tgen}([a, X], \mathit{list}(\texttt{nonvar})) = \texttt{nonvar}$. The above algorithm does not try to invent new types and the last example shows that there are ways to make the algorithm more precise (by inferring $\mathit{list}(\texttt{dynamic})$ rather than $\texttt{nonvar}$). However, the algorithm does guarantee termination and correctness in the following sense:

Proposition 1. *For every infinite sequence of terms $t_1, t_2, \ldots$ and for every binding type τ_0 the sequence $\tau_1, \tau_2, \ldots$ with $\tau_i = \mathit{tgen}(t_i, \tau_{i-1})$ stabilises and there exists a $k > 0$ such that for all $j > k$ we have $t_j : \tau_j$.*

The proposition follows from the fact that by construction $t : \mathit{tgen}(t, \tau)$ and for any given type τ only finitely many more general types can be obtained by applying *tgen*.

Dangerous Memoisation The watchdog flags a memoisation of a call as dangerous if it can find an ancestor in the specialization tree which is embedded in the call. To fix this (potential) problem detected by the watchdog we make use of the following definition:

Definition 7. *Let $a = p(a_1, \ldots, a_n)$ and $b = p(t_1, \ldots, t_n)$ be two atoms such that $a \trianglelefteq b$. Then the* growing argument positions *of b wrt a are all indices i such that t_i is not a variant of a_i.*

It can be seen that for every growing argument position i we have $a_i \triangleleft b_i$. A simple solution is now to compute all growing argument positions and adapt the filter declaration so that the corresponding arguments are given the binding-type $\texttt{dynamic}$ (i.e., these arguments will be replaced by fresh variables during memoisation). This is the solution that we have currently implemented within LOGEN. A more subtle solution could be developed by employing most specific generalisation (*msg*) [12] a common technique used for controlling generalisation in online specialisation [16]: the *msg* of a set of terms S is the most specific term such that all expressions in S are instances of it. We can now compute the *msg* on the

growing arguments and then only replace the variables by `dynamic`. For example, given the memoised call $p(b, p(s(s(0)), 1)), [W])$ with filter declaration `:- filter p(static,static,list(dynamic))` and with covering ancestor $p(a, p(s(0), 1), [V])$ we have $msg(\{p(s(s(0)), 1), p(s(0), 1)\}) = p(s(Z), 1)$ and thus obtain the new declaration: `:- filter p(static,p(s(dynamic),static),list(dynamic))`.

Pragmatic BTA, Debugging and ASAP Case Studies The above new methods and the accompanying web interface make it now much easier for users to fix their incorrect annotations. Furthermore, it also enables a *pragmatic BTA* to be performed. Basically, the idea is to start off with an initial annotation where all user predicates are marked as **unfold** and all built-ins as **call**. The specialiser is then run in watchdog mode on a series of representative specialisation queries. Every time a problem is detected the fix computed by LOGEN is applied. This is repeated until all sample queries can be specialised without error. One should also combine this process with the filter propagation algorithm of [8] to ensure that the annotations are correct wrt `static` information. It is easy to see that this process must terminate. However, the approach does obviously not guarantee that specialisation will terminate for all queries. Still, this approach has proven to be successful on some larger case studies within the European project ASAP. In the first study, LOGEN was used to specialise an interpreter for a process language with the aim of automatic task scheduling on pervasive devices. In another case study a complete emulator for PIC assembly [10] was successfully specialised by LOGEN for arbitrary PIC programs (for further analysis such as dead code detection). In both cases the automatic BTA was not applicable (due to the size of the interpreters and due to the various built-ins used), but the watchdog mode enabled us to annotate the programs with much less effort (e.g. a few hours for the process language interpreter) than was previously possible. Note that once the annotation was developed, the watchdog mode was turned off allowing the offline specialiser to run at full speed for the various applications.

Another application of our method and web interface is debugging. For instance, we had a version of the task scheduling interpreter containing a built-in error (calling `T2 =.. [Op,V0|V1]` instead of `T2 =.. [Op,V0,V1]`). When one executes the main method of this program one simply gets the following message: `ERROR: illegal arithmetic expression`, without any indication about the call or the location of the error. Using a debugger to locate such errors is often not practical: the error was reached after 175 steps (and more tricky problems will easily require thousands or millions of steps given that current Prolog systems can exceed 20 Million logical inferences per second) and when using the debugger's leap command one gets the same message as above, without any indication about the problematic built-in nor its location. Our watchdog approach can be used as an automatic debugger to locate those problems (as well as locating loops). The idea is to use a simple BTA which annotates all calls to user predicates as unfold and all calls to built-ins as call (this BTA is available via the web interface). Specializing then corresponds to supervised execution, where checks and program point information has been weaved into the source

program. We thus get information about the actual call that causes the problem (as well as precise program point information which is used by the web interface to highlight the location of the error):

```
  % logen task_csp_scheduler_err.pl "main([2,3,4])" -wb
<| BUILT-IN ERROR |> : CALL Atom _7966=..[+,2|10]
```

The `-wb` option tells LOGEN to check only built-ins and not user predicates for potential loops (in order to reduce the overhead). Some experimental data for the overhead of this approach can be seen in the table below. As can be seen, the overhead is very reasonable.

Benchmark	original (consulted)	original (compiled)	LOGEN (normal)	LOGEN -wb (watch)
lambdaint	2.18 ms	1.22 ms	1.29 ms	4.48 ms
task_scheduler	1.15 ms	0.61 ms	0.59 ms	2.83 ms

6 Related Work and Conclusion

We believe that our idea to use online techniques to supervise an offline specialiser to be new. However, in the past several researchers have investigated hybrid strategies[6] (e.g., [3, 7, 11]) where offline partial evaluators were augmented with online constructs. The aim there was different (to augment specialisation power) and only very few actual techniques or implementations exist ([29] is one). Another related work is [25] where program transformations are used to construct justifications for computed answers.

We have presented the idea of using online techniques in general and the homeomorphic embedding relation in particular to supervise an offline specialiser, in an effort to help the development of correct annotations by identifying error conditions. We have implemented this technique within the LOGEN system and have shown that this technique turned out to be very effective: very few false alarms were raised and the overhead was low enough for the technique to be practically usable. We have presented an improved web interface that feeds back this information to the user in an intuitive way, and we have presented techniques to automatically computed fixes for the spotted problems. We have applied our ideas to various case studies, and the techniques have enabled us to annotate and specialise much larger programs than was previously possible. These new features of LOGEN can also be tried out at using a web interface at `http://stups.cs.uni-duesseldorf.de/~pe/weblogen`.

Our technique can also be used as a pragmatic BTA: one simply starts with a maximally aggressive annotation and then lets the watchdog find and fix the errors. While this does not produce termination guarantees, it has proven very effective in practice and can easily deal with larger source programs and with many built-ins. Another application of our watchdog mode is to locate built-in

[6] Sometimes called mixline annotations.

errors or non-termination problems in user programs, and highlight those errors directly within the user's source program.

The LOGEN system has found many uses; from specialising PIC assembly code emulators, to point cut languages for aspect orientation over to CTL model checkers. But so far using LOGEN required considerable expertise in partial evaluation, hampering a more widespread usage. With this work we hope to make LOGEN and the underlying technology accessible to a broader community.

Acknowledgements We would like to thank Marc Fontaine for useful feedback.

References

1. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
2. S. Barker, M. Leuschel, and M. Varea. Efficient and flexible access control via logic program specialisation. In *Proceedings PEPM'04*, pages 190–199. ACM Press, 2004.
3. A. Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 27–50. North-Holland, 1988.
4. M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.
5. M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Proceedings ESOP'98*, LNCS 1381, pages 27–41. Springer-Verlag, April 1998.
6. N. H. Christensen and R. Glück. Offline partial evaluation can be as accurate as online partial evaluation. *ACM Transactions on Programming Languages and Systems*, 26(1):191–220, 2004.
7. C. Consel. Binding time analysis for high order untyped functional languages. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 264–272, New York, NY, USA, 1990. ACM Press.
8. S.-J. Craig, J. Gallagher, M. Leuschel, and K. S. Henriksen. Fully automatic binding-time analysis for Prolog. In S. Etalle, editor, *Proceedings LOPSTR 2004*, LNCS 3573, pages 53–68. Springer-Verlag, August 2004.
9. A. J. Glenstrup and N. D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, LNCS 1181, pages 273–284. Springer-Verlag, 1996.
10. K. S. Henriksen and J. P. Gallagher. Analysis and specialisation of a PIC processor. In *Proceedings of the IEEE International Conference on Systems, Man & Cybernetics (2)*, pages 1131–1135, The Hague, The Netherlands, 2004.
11. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
12. J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan-Kaufmann, 1988.
13. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
14. M. Leuschel. On the power of homeomorphic embedding for online termination. In G. Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

15. M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In T. Æ. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation - Essays dedicated to Neil Jones*, LNCS 2566, pages 379–403. Springer-Verlag, 2002.
16. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.
17. M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specializing interpreters using offline partial deduction. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, LNCS 3049, pages 341–376. Springer-Verlag, 2004.
18. M. Leuschel, D. Elphick, M. Varea, S. Craig, and M. Fontaine. The Ecce and Logen partial evaluators and their web interfaces. In F. T. John Hatcliff, editor, *Proceedings of PEPM'06*, pages 88–94. IBM Press, Januar 2006.
19. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.
20. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
21. M. Leuschel and G. Vidal. Forward slicing by conjunctive partial deduction and argument filtering. In M. Sagiv, editor, *Proceedings ESOP 2005*, LNCS 3444, pages 61–76. Springer-Verlag, April 2005.
22. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
23. H. Makholm. On Jones-optimal specialization for strongly typed languages. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, LNCS 1924, pages 129–148. Springer-Verlag, 2000.
24. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press.
25. G. Pemmasani, H.-F. Guo, Y. Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online justification for tabled logic programs. In *Proceedings FLOPS 2004*, LNCS 2998, pages 24 – 38. Springer-Verlag, January 2004.
26. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
27. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1–3):17–64, 1996.
28. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479, Portland, USA, 1995. MIT Press.
29. M. Sperber. Self-applicable online partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar*, LNCS 1110, pages 465–480, Schloß Dagstuhl, 1996. Springer-Verlag.
30. W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs, editor, *Logic Program Synthesis and Transformation. Proceedings of LOPSTR'97*, LNCS 1463, pages 322–342, Leuven, Belgium, July 1997.
31. Q. Wang, G. Gupta, and M. Leuschel. Towards provably correct code generation via Horn logical continuation semantics. In M. V. Hermenegildo and D. Cabeza, editors, *Proceedings PADL'05*, of *LNCS 3350*, pages 98–112. Springer, 2005.