Fast Offline Partial Evaluation of Large Logic Programs[⋆]

Michael Leuschel[1] and Germán Vidal[2]

[1] Institut für Informatik, Universität Düsseldorf, D-40225, Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de
[2] DSIC, Technical University of Valencia, E-46022, Valencia, Spain
gvidal@dsic.upv.es

1 Introduction

There are two main approaches to *partial evaluation* [6], a well-known technique for program specialisation. *Online* partial evaluators basically include an augmented interpreter that tries to evaluate the program constructs as much as possible—using the partially known input data—while still ensuring the termination of the process. *Offline* partial evaluators, on the other hand, require a *binding-time analysis* (BTA) to be run before specialisation, which annotates the source code to be specialised. Roughly speaking, the BTA annotates the various calls in the source program as either unfold (executed by the partial evaluator) or memo (executed at run time, i.e., memoized), and annotates the arguments of the calls themselves as static (known at specialisation time) or dynamic (only definitely known at run time).

In the context of logic programming, a BTA should ensure that the annotations of the arguments are correct, in the sense that an argument marked as static will be ground in all possible specialisations. It should also ensure that the specialiser will always terminate. This can be broadly classified into local and global termination [7]. The *local* termination of the process implies that no atom is infinitely unfolded. The *global* termination ensures that only a finite number of atoms are unfolded. Basically, the program annotations are *safe* when

- all atoms marked as unfold can be unfolded as much as possible (as indicated by the annotations) while still guaranteeing the local termination, and
- global termination is guaranteed by generalising the dynamic arguments whenever a new atom is added to the set of (to be) partially evaluated atoms; also, all arguments marked as static must indeed be ground.

In previous work, Craig et al [4] have presented a fully automatic BTA for logic programs, whose output can be used for the offline partial evaluator LOGEN [9]. Unfortunately, this BTA still suffers from some serious practical limitations:

- The current implementation does not guarantee global termination.

[⋆] This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008

- The technique and its implementation is quite complicated, consisting of a combination of various other analyses: model-based binding-type inference, binary clause generation, inter-argument size relation analysis with polyhedra, etc., running on different Prolog systems. As a consequence, the current implementation is quite fragile and hard to maintain.
- In addition to the implementation complexity, the technique is also very slow and does not scale to medium-sized examples.

Recently, Vidal [13] has introduced a quasi-termination analysis for logic programs that is independent of the selection rule. This approach is less precise than other termination analyses that take into account a particular selection strategy but, as a counterpart, is also faster and well suited for partial evaluation (where flexible selection strategies are often mandatory, see, e.g., [1, 7]). In this paper, we introduce a new BTA for logic programs with the following advantages:

- it is conceptually simpler and considerably faster, scaling to medium-sized or even large examples;
- the technique does ensure both local and global termination;
- the technique can be used to infer annotations for *semi-online* specialisers (e.g., LOGEN has semi-online features) combining the speed of the offline approach with the power of the online one.

The main source of improvement comes from using a size-change analysis for termination purposes rather than a termination analysis based on the *abstract binary unfoldings* [3] as in [4]. Basically, the main difference between both termination analyses is that the binary unfoldings consider a particular selection strategy (i.e., Prolog's leftmost selection strategy). As a consequence, every time the annotation of an atom changes from unfold to memo during the BTA of [4], the termination analysis should be redone from scratch in order to take into account that this atom would not be unfolded (thus avoiding the propagation of some bindings). On the other hand, the size-change analysis is independent of a particular selection strategy. As a consequence, it is less precise, since no variable bindings are propagated between the body atoms of a clause, but it should be run only *once*. In general the termination analysis is the most expensive component of a BTA.

We have implemented the new approach, and we will show on experimental results that the new technique is indeed much faster and much more scalable. On some examples, the accuracy is still sub-optimal, and we present various ways to improve this. Still, this is the first BTA for logic programs that can be applied to larger programs, and as such is an important step forward.

2 Size-Change Termination Analysis

In the remainder we assume basic knowledge of the fundamentals of logic programming [2, 12]. In this section, we recall the basis of the size-change analysis of [13], which is used to check the (quasi-)termination of a program.

We denote by $calls_P^{\mathcal{R}}(Q_0)$ the set of calls in the computations of a goal Q_0 within a logic program P and a computation rule $\mathcal{R}$. We say that a query Q is *strongly terminating* w.r.t. a program P if every SLD derivation for Q with P is finite. The query Q is *strongly quasi-terminating* if, for every computation rule $\mathcal{R}$, the set $call_P^{\mathcal{R}}(Q)$ contains finitely many nonvariant atoms. A program P is strongly (quasi-)terminating w.r.t. a set of queries $\mathcal{Q}$ if every $Q \in \mathcal{Q}$ is strongly (quasi-)terminating w.r.t. P. For conciseness, in the remainder of this paper, we write "(quasi-)termination" to refer to "strong (quasi-)termination."

Size-change analysis is based on constructing graphs that represent the decrease of the arguments of a predicate from one call to another. For this purpose, some ordering on terms is required. In [13], reduction orders $(\succsim, \succ)$ *induced* from symbolic norms $||\cdot||$ are used:

Definition 1 (symbolic norm [11]). *Given a term t,*

$$||t|| = \begin{cases} m + \sum_{i=1}^{n} k_i ||t_i|| & \textit{if } t = f(t_1,\ldots,t_n),\ n \geqslant 0 \\ t & \textit{if } t \textit{ is a variable} \end{cases}$$

where m and $k_1,\ldots,k_n$ are non-negative integer constants depending only on f/n. Note that we associate a variable over integers with each logical variable (we use the same name for both since the meaning is clear from the context).

The introduction of variables in the range of the norm provides a simple mechanism to express dependencies between the sizes of terms.

The associated induced orders $(\succsim, \succ)$ are defined as follows: $t_1 \succ t_2$ (respec. $t_1 \succsim t_2$) if $||t_1\sigma|| > ||t_2\sigma||$ (respec. $||t_1\sigma|| \geqslant ||t_2\sigma||$) for all substitution σ that makes $||t_1\sigma||$ and $||t_2\sigma||$ ground (e.g., an integer constant). Two popular instances of symbolic norms are the symbolic *term-size* norm $||\cdot||_{ts}$ (which counts the arities of the term symbols) and the symbolic *list-length norm* $||\cdot||_{ll}$ (which counts the number of elements of a list), e.g.,

$$f(X,Y,a) \succ_{ts} f(X,a,b) \text{ since } ||f(X,Y,a)||_{ts} = X+Y+3 > X+3 = ||f(X,a,b)||_{ts}$$
$$[X|R] \succsim_{ll} [s(X)|R] \text{ since } ||[X|R]||_{ll} = R+1 \geqslant R+1 = ||[s(x)|R]||_{ll}$$

Now, we produce a *size-change graph* $\mathcal{G}$ for every pair (H, B_i) of every clause $H \leftarrow B_1,\ldots,B_n$ of the program, with edges between the arguments of H and B_i when the size of the corresponding terms decrease w.r.t. a given reduction pair $(\succsim, \succ)$. Consider the following simple program:

(c_1) $incList([\,], \_, [\,])$.
(c_2) $incList([X|R], I, L) \leftarrow iList(X, R, I, L)$.
(c_3) $iList(X, R, I, [XI|RI]) \leftarrow add(I, X, XI), incList(R, I, RI)$.
(c_4) $add(0, Y, Y)$.
(c_5) $add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z)$.

Here, the size-change graphs associated to, e.g., clause c_3 are as follows:

$$\begin{tikzcd}
\mathcal{G} : iList \arrow[r] & add \\
1_{iList} \arrow[rd, "\succsim_{ts}"] & 1_{add} \\
2_{iList} & 2_{add} \\
3_{iList} \arrow[ruu, "\succsim_{ts}"] & 3_{add} \\
4_{iList} \arrow[ru, "\succ_{ts}"] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{G}' : iList \arrow[r] & incList \\
1_{iList} & 1_{incList} \\
2_{iList} \arrow[ru, "\succsim_{ts}"] & 2_{incList} \\
3_{iList} \arrow[ru, "\succsim_{ts}"] & 3_{incList} \\
4_{iList} \arrow[ru, "\succ_{ts}"] &
\end{tikzcd}$$

using a reduction pair $(\succsim_{ts}, \succ_{ts})$ induced from the symbolic term-size norm.

In order to identify the program *loops*, we should compute roughly a transitive closure of the size-change graphs by composing them in all possible ways. Basically, given two size-change graphs:

$$\mathcal{G} = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1) \qquad \mathcal{H} = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, l_r\}, E_2)$$

w.r.t. the same reduction pair $(\succsim, \succ)$, their concatenation is defined by

$$\mathcal{G} \bullet \mathcal{H} = (\{1_p, \ldots, n_p\}, \{1_r, \ldots, l_r\}, E)$$

where E contains an edge from i_p to k_r iff E_1 contains an edge from i_p to some j_q and E_2 contains an edge from j_q to k_r. Furthermore, if some of the edges are labelled with $\succ$, then so is the edge in E; otherwise, it is labelled with $\succsim$.

In particular, we only need to consider the *idempotent* size-change graphs $\mathcal{G}$ with $\mathcal{G} \bullet \mathcal{G} = \mathcal{G}$, because they represent the (potential) program loops.

For the previous example, we compute the following idempotent size-change graphs:

$$\begin{tikzcd}
\mathcal{G}_1 : \mathit{incList} \arrow[r] & \mathit{incList} & \mathcal{G}_2 : \mathit{iList} \arrow[r] & \mathit{iList} & \mathcal{G}_3 : \mathit{add} \arrow[r] & \mathit{add} \\
1_{\mathit{incList}} \arrow[r, "\succ_{ts}"] & 1_{\mathit{incList}} & 1_{\mathit{iList}} \arrow[r, bend left, "\succ_{ts}"] & 1_{\mathit{iList}} & 1_{\mathit{add}} \arrow[r, "\succ_{ts}"] & 1_{\mathit{add}} \\
2_{\mathit{incList}} \arrow[r, "\succsim_{ts}"] & 2_{\mathit{incList}} & 2_{\mathit{iList}} \arrow[r, "\succ_{ts}"] & 2_{\mathit{iList}} & 2_{\mathit{add}} \arrow[r, "\succsim_{ts}"] & 2_{\mathit{add}} \\
3_{\mathit{incList}} \arrow[r, "\succ_{ts}"] & 3_{\mathit{incList}} & 3_{\mathit{iList}} \arrow[r, "\succsim_{ts}"] & 3_{\mathit{iList}} & 3_{\mathit{add}} \arrow[r, "\succ_{ts}"] & 3_{\mathit{add}} \\
 & & 4_{\mathit{iList}} \arrow[r, "\succ_{ts}"] & 4_{\mathit{iList}} & &
\end{tikzcd}$$

that represent how the size of the arguments of the three potentially looping predicates changes from one call to another.

3 A Fully Automatic Binding-Time Analysis

3.1 Logen and Overview of the Algorithm

The LOGEN system [9] is an *offline* partial evaluator for Prolog, which works on an annotated version of the source program. LOGEN uses two kinds of annotations:

- *Filter declarations*, which declare which arguments to which predicates are static and which ones are dynamic. This influences the global control (only): dynamic arguments are always replaced by fresh variables, while static arguments are kept as they are. In this work, we consider a relatively simple domain of binding-types for predicate arguments:
  - static: the argument is definitely known at partial evaluation time;
  - nonvar: the argument is not a variable at partial evaluation time, i.e., the top-level function symbol is known;
  - list_nonvar: the argument is definitely bound to a finite list, whose elements are not variables;

  - list: the argument is definitely bound to a finite list of possibly unknown arguments at partial evaluation time;
  - dynamic: the argument is possibly unknown at partial evaluation time.

  In LOGEN, though, the user can define their own *binding-types* [4], and one can use the pre-defined list-constructor to define additional types such as list(dynamic) to denote a list of known length with dynamic elements, or list(nonvar) to denote a list of known length with non-variable elements.
- *Clause annotations*, which indicate for every call in the body of a clause how that call should be treated during unfolding. This thus influences the local control only, which is effectively hard-wired. Calls to user predicates are either annotated with **memo** — indicating that it should not be unfolded — or with **unfold** — indicating that it should be unfolded. Calls to built-ins are either annotated with **rescall** — indicating that it should not be called — or with **call** — indicating that it should be called during specialisation.

Our algorithm works in two phases. In the first phase a goal-independent size-change analysis is run to obtain selection-rule independent information about terminating and quasi-terminating calls. In the second phase, an abstract interpretation propagates abstract information about entry points thorough the program, making decisions about how to annotate the program. These decisions are based on the abstract patterns and the result of the size-change analysis.

3.2 Propagation of Binding-Types

Given a program and the specification of the input data for some initial predicate, a binding-time analysis first propagates the known information through all the program clauses. The propagation of binding-types works as follows:

- A call pattern is processed and the abstract information propagated through the program;
- For every program point, one has to decide
  - For built-ins, whether they should be called or not; this is decided solely on the abstract information. E.g., `functor(nonvar,dynamic,dynamic)` can be called, `functor(dynamic,static,dynamic)` cannot.
  - For calls to user-defined predicates, whether to unfold or memoize them. For this, the information of the size-change analysis is used. If the predicate is marked as unfold, its clauses will be analysed. If a predicate is marked as memo, then it is checked whether the current call is safe w.r.t. global termination. If not, arguments are generalised (marked dynamic). For this, again the size-change analysis results are used.

  Note that in both cases these decisions influence the propagation of abstract information to the right of the call under consideration. E.g., consider the goal `functor(X,F,N), p(F)`. If `X` is nonvar, then the `functor` call will be evaluated and `F` becomes static within `p(F)` (no matter what `F` was before).

We consider that the user provides a program and the binding-types for some predicate, and an iterative algorithm (like that of [4]) is used to propagate the binding-types to all predicates, i.e., to compute a program *division* of the form

$$div = \{p_1 \mapsto (bt_{11}, \ldots, bt_{1m_1}), \ldots, p_n \mapsto (bt_{n1}, \ldots, bt_{nm_n})\}$$

where $p_1, \ldots, p_n$ are the predicates of the considered program and bt_{ij} are binding-types. We write $div(p_i)$ to denote the binding-types associated to p_i in div and $div(p_i, j)$ to denote the binding-type of the j-th argument of p_i in div. For simplicity, we consider a *monovariant* binding-time analysis where a single sequence of binding-types is associated with each predicate.

3.3 Ensuring Local Termination

In this section, we consider the local termination of the specialisation process, i.e., we analyse whether the unfolding of an atom terminates for any selection strategy.

Let us first recall the notion of *instantiated enough* w.r.t. a symbolic norm from [11]: a term t is instantiated enough w.r.t. a symbolic norm $||\cdot||$ if $||t||$ is an integer constant. We now present a sufficient condition for termination. Basically, we require the decreasing parameters of (potentially) looping predicates to be instantiated enough w.r.t. a symbolic norm in the considered computations.

Theorem 1 (termination [13]). *Let P be a program and let $(\succsim, \succ)$ be a reduction pair induced by a symbolic norm $||\cdot||$. Let $\mathcal{A}$ be a set of atoms. If every idempotent size-change graph for P contains at least one edge $i_p \xrightarrow{\succ} i_p$ such that, for every atom $A \in \mathcal{A}$, computation rule $\mathcal{R}$, and atom $p(t_1, \ldots, t_n) \in calls_P^{\mathcal{R}}(A)$, t_i is instantiated enough w.r.t. $||\cdot||$, then P is terminating w.r.t. $\mathcal{A}$.*

For instance, according to the idempotent graphs in Sect. 2, calls to predicate *incList* terminate whenever its first or its last argument is instantiated enough w.r.t. $||\cdot||_{ts}$, either the second or the fourth argument of *iList* is instantiated enough w.r.t. $||\cdot||_{ts}$, and either the second or the third argument of *add* is instantiated enough w.r.t. $||\cdot||_{ts}$.

Note that the strictly decreasing arguments should be instantiated enough in every possible derivation w.r.t. any computation rule. Although this condition is undecidable in general, it can be approximated by using the binding-types of the computed division (cf. Section 3.2). From Theorem 1 above, one can design a simple annotation strategy as follows:

Definition 2 (local annotations). *Let P be a program and let $(\succsim, \succ)$ be a reduction pair induced by a symbolic norm $||\cdot||$. Let $\mathcal{G}$ be the idempotent size-change graphs from the size-change analysis and div be the computed division. The calls in the bodies of the clauses of P are annotated as follows:*

- *a call $p(t_1, \ldots, t_n)$ is annotated with* unfold *if every idempotent size-change graph for p/n in $\mathcal{G}$ contains at least one edge $i_p \xrightarrow{\succ} i_p$, $1 \leqslant i \leqslant n$, such that $div(p, i) \neq \mathsf{dynamic}$;*
- *otherwise, the call is annotated with* memo.

Consider, e.g., that the propagation of binding-types returns the following division for the *incList* program of Sect. 2:

$$\begin{array}{lll} div = \{ & incList & \mapsto (\mathsf{dynamic}, \mathsf{static}, \mathsf{dynamic}), \\ & iList & \mapsto (\mathsf{dynamic}, \mathsf{dynamic}, \mathsf{static}, \mathsf{dynamic}), \\ & add & \mapsto (\mathsf{static}, \mathsf{dynamic}, \mathsf{dynamic}) \quad \} \end{array}$$

Then, according to the idempotent size-change graphs of Sect. 2 and Theorem 1 above, only the call to *add* can be marked as unfold in the program.

3.4 Ensuring Global Termination

In order to ensure the global termination of the specialisation process, we should ensure that only a finite number of non-variant atoms are added to the set of (to be) partially evaluated atoms, i.e., that the sequence of atoms is *quasi-terminating*.

For this purpose, [13] adds an additional condition, namely that the considered norms $||\cdot||$ should be *bounded*, i.e., the set $\{s \mid ||t|| \geqslant ||s||\}$ should contain a finite number of nonvariant terms for any term t. Note that this condition excludes the use of some norms. For instance, the list-length norm is not bounded, a serious limitation of the results in [13]. Moreover, quasi-termination was only ensured if all calls at partial evaluation time were *linear*, i.e., if no call contained multiple occurrences of the same variable.

Basically, [13, Theorem 4.7] shows that quasi-termination is guaranteed when all idempotent size-change graphs have an edge $i_p \xrightarrow{R} i_p$ for every argument, with $R \in \{\succ, \succsim\}$, the considered norm is bounded, and all calls are linear. According to this result, we could define a simple annotation strategy as follows:

- the reduction pair is induced by the term-size norm (since the list-length norm is not bounded);
- for every call $p(t_1, \ldots, t_n)$, if there exists an idempotent size-change graph with no edge to the input argument i_p, then t_i is annotated as dynamic;
- otherwise, it is annotated as static.

This approach, however, is too coarse to produce useful results. Now, we show that arbitrary symbolic norms can still be used as long as a suitable generalisation is applied at the global level.

Let $||\cdot||$ be a symbolic norm. Given a term t, we denote by $mgg^{||\cdot||}(t)$ the most general generalisation of t such that $||t|| = ||mgg^{||\cdot||}(t)||$. E.g., given the term $t = [s(N), b]$, we have $mgg^{||\cdot||_{ll}}(t) = [X, Y]$. We also let $mgg^{||\cdot||}(p(t_1, \ldots, t_n)) = p(mgg^{||\cdot||}(t_1), \ldots, mgg^{||\cdot||}(t_n))$ and $mgg^{||\cdot||}(\mathcal{A}) = \{mgg^{||\cdot||}(A) \mid A \in \mathcal{A}\}$.

Definition 3 (quasi-termination up to a symbolic norm). *Let P be a program and $\mathcal{A}$ be a set of atoms. Given a symbolic norm $||\cdot||$, we say that P is quasi-terminating w.r.t. $\mathcal{A}$ up to $||\cdot||$ if, for every computation rule $\mathcal{R}$, the set $mgg^{||\cdot||}(call_P^{\mathcal{R}}(Q))$ contains finitely many nonvariant atoms.*

This notion of quasi-termination *up to a symbolic norm* is particularly useful in the context of partial evaluation since it takes into account that some generalisation is often performed in the global level. Consider, e.g., the program $\{p([X]) \leftarrow p([s(X)]).\}$. This program cannot be proved quasi-terminating according to [13] because the symbolic list-length norm cannot be used; actually, the program is not quasi-terminating:

$$p([a]) \rightsquigarrow p([s(a)]) \rightsquigarrow p([s(s(a))]) \rightsquigarrow \ldots$$

However, if we consider the sequence of calls in which every atom A is replaced by $mgg^{||\cdot||_{ll}}(A)$, then the computation is indeed quasi-terminating:

$$mgg^{||\cdot||_{ll}}(p([a])) = mgg^{||\cdot||_{ll}}(p([s(a)])) = \ldots = p(X) \qquad \text{(up to renaming)}$$

Therefore, non-bounded norms can be used as long as all symbols that are not taken into account by this norm are generalised in the global level.

In the following, we assume that the specialisation algorithm proceeds in this way, which allows us to exploit a stronger quasi-termination result and annotate less terms as dynamic.

Definition 4 (global annotations). *Let P be a program and let $(\succsim, \succ)$ be a reduction pair induced by a symbolic norm $||\cdot||$. Let $\mathcal{G}$ be the idempotent size-change graphs of the size-change analysis. For each predicate p/n and argument i, $1 \leqslant i \leqslant n$, we compute an initial division div such that:*

- *$div(p,i) =$ static if, for every idempotent size-change graph of P associated to p/n, there is an edge $j_p \xrightarrow{R} i_p$, $R \in \{\succ, \succsim\}$;*
- *otherwise, $div(p,i) =$ dynamic.*

Furthermore, if an argument is marked as static but the list-length norm was used, then it is changed to list(dynamic) to ensure that $mgg^{||\cdot||_{ll}}$ is applied to this argument in the global level.

For instance, for the *incList* example, given the idempotent size-change graphs of Sect. 2, all arguments of all predicates can be safely marked as static. This initial division is then used in the propagation of binding-types discussed in Sect. 3.2.

4 The BTA in Practice

Our new binding-time analysis is still being continuously extended and improved. We provide some preliminary experimental results below. The experiments were run on a MacBook Pro with a 2.33 GHz Core2 Duo Processor and 3 GB of RAM. Our BTA was using SICStus Prolog 4.02.

First, a simple example is the match-kmp from DPPD [10]. The original run time for 100,000 executions of the queries from [10] took 1.85 s. The run time of our BTA was below the measuring threshold and running LOGEN on the result also took very little time (0.00175 s). The specialised program took only 1.57 s (i.e., a speedup of 1.18). For comparison, ECCE took 0.02 s to specialise the program; the resulting specialised program is faster still (0.9 s), as the online specialiser was able to construct a KMP-like specialised pattern matcher. A better example is the regular expression interpreter from [10]. Here, the original took 1.82 s for 100,000 executions of the queries (r3 in [10]). Our BTA took 0.01 s, LOGEN took 0.004 s to specialise the program, which then took 0.97 s to run the queries (i.e., a speedup of 1.88). ECCE took 0.04 s to specialise the program; the specialised program only runs marginally faster (0.96 s).

To validate the scalability of our BTA we have tried our new BTA on a larger example, the PIC processor emulator from [5]. It consists of 137 clauses and 855 lines of code. The purpose is here was to specialise the PIC emulator for a particular PIC machine program, in order to run various static analyses on it. The old BTA from [4] took 1 m 39 s (on a Linux server which corresponds roughly to 67 seconds on the MacBook Pro).[3] Furthermore the generated annotation file is erroneous and could not be used for specialisation. With our new BTA a correct annotation is generated in 2 s; the ensuing specialisation by LOGEN took 4.26 s. Also, with ECCE it took 9 m 30 s to construct a (very large) specialised program.

Another example, is the lambda interpreter for a small functional language from [8]. This interpreter contains some side-effects and could not be run through ECCE. Our BTA took 1.28 s to generate an annotation. Specialisation with LOGEN then took 7 ms, but unfortunately resulting in no speedup over the original.

In conclusion, our BTA is well suited to be applied to larger programs. The accuracy of the annotations is not yet optimal, but note that at least we do obtain correct annotations which provide a good starting point for hand-tuning.

5 Future Work

In conclusion, we have presented a very fast BTA, able to cope with larger programs and for the first time ensuring both local and global termination. Compared to [13] we have a stronger quasi-termination result, allow non-bounded norms and have a new more precise annotation procedure. While the accuracy of our BTA is reasonable, there is still much room for improvement.

One way to improve the accuracy of the BTA is to generate not offline, but semi-online annotations. In other words, instead of generating `rescall` we produce `semicall` (which tries to call the built-in if enough static information is available), `online` instead of `memo` (which tries to unfold, if this is safe given the unfolding history), and marking arguments as `online` rather than `dynamic`. This should yield a fast but still precise partial evaluator: most of the decisions will hopefully already have been taken offline, the online overhead is only applied to those places in the source code where the BTA was imprecise.

Another source of improvement will come from refining the size-change analysis so that not all size-change graphs are *composable* but only those in which the considered atoms unify. Furthermore, we plan to label every call in the program so that different loops for the same predicate can be distinguished. Preliminary experiments in these directions are promising.

[3] It took us considerable time to get [4] working on a newer machine (due to the various components requiring differing — sometimes outdated — Prolog systems). Even there, this BTA sometimes ran out of memory on our MacBook Pro, and we had to resort to running the BTA on our webserver (this Linux server has 26 MLIPS, whereas our Mac has 38 MLIPS).

References

1. E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Deduction of Logic Programs with Impure Predicates. In *Proc. of LOPSTR'05*, pages 115–132. Springer LNCS 3901, 2006.
2. Krzysztof R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.
3. M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
4. S.-J. Craig, J. Gallagher, M. Leuschel, and K.S. Henriksen. Fully Automatic Binding Time Analysis for Prolog. In *Proc. of the Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 53–68. Springer LNCS 3573, 2005.
5. Kim S. Henriksen and John 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.
6. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
7. M. Leuschel and M. Bruynooghe. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming*, 2(4-5):461–515, 2002.
8. M. Leuschel, S.-J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising Interpreters Using Offline Partial Deduction. In *Program Development in Computational Logic*, pages 340–375. Springer LNCS 3049, 2004.
9. 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(1-2):139–191, 2004.
10. Michael Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
11. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *Proc. of Int'l Conf. on Logic Programming (ICLP'97)*, pages 63–77. The MIT Press, 1997.
12. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
13. G. Vidal. Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In *Proc. of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM'07)*, pages 51–60. ACM Press, 2007.