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

Abstract. In this paper, we present a fast binding-time analysis (BTA) by integrating a size-change analysis, which is independent of a selection rule, into a classical BTA for offline partial evaluation of logic programs. In contrast to previous approaches, the new BTA is conceptually simpler and considerably faster, scaling to medium-sized or even large examples and, moreover, it ensures both the so called local and global termination. We also show that through the use of selective hints, we can achieve both good specialisation results and a fast BTA and specialisation process.

1 Introduction

There are two main approaches to *partial evaluation* [8], 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 [9]. 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. In previous work, Craig et al [4] have presented a fully automatic BTA for logic programs, whose output can be used for the

[⋆] 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 DAAD PPP D/06/12830 together with *Acción Integrada* HA2006-0008

offline partial evaluator LOGEN [11]. Unfortunately, this BTA still suffers from some serious practical limitations:

– The current implementation does not guarantee global termination.
– The technique and its implementation are quite complicated, consisting of a combination of various other analyses: model-based binding-time inference, binary clause generation, inter-argument size relation analysis with polyhedra. To make matters more complicated, these analyses also run 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 [18] 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, 9]).

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 make use of user-provided hints to obtain better specialisation.

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.

The paper is organized as follows. Section 2 introduces a fully automatic BTA which is parameterized by the output of a termination analysis, which is then presented in Sect. 3. Section 4 presents a summary of experimental results which demonstrate the usefulness of our approach. Finally, Sect. 5 concludes and presents some directions for future work.

2 A Fully Automatic Binding-Time Analysis

In the remainder we assume basic knowledge of the fundamentals of logic programming [2, 17]. In this section, we present a fully automatic BTA for (definite) logic programs. Our algorithm is parametric w.r.t.

– a domain of binding-times and
– a function for propagating binding-times through the atoms of clause bodies.

For instance, one can consider a simple domain with the basic binding-times static (definitely known at partial evaluation time) and $\mathsf{dynamic}$ (possibly unknown at partial evaluation time). More refined binding-time domains could also include, e.g., the following elements:

– nonvar: the argument is not a variable at partial evaluation time, i.e., the top-level function symbol is known;
– $\mathsf{list\_nonvar}$: 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.

In the LOGEN system [11], an *offline* partial evaluator for Prolog, the user can also define their own binding-times [4] (called *binding-types* in this context), and one can use the pre-defined list-constructor to define additional types such as $\mathsf{list(dynamic)}$ to denote a list of known length with dynamic elements, or $\mathsf{list(nonvar)}$ to denote a list of known length with non-variable elements.

For any binding-time domain $(\mathcal{D}, \sqsubseteq)$, we let $b_1 \sqsubseteq b_2$ denote that b_1 is *less dynamic* than b_2; also, we denote the least upper bound of binding-times b_1 and b_2 by $b_1 \sqcup b_2$ and the greatest lower bound by $b_1 \sqcap b_2$. For instance, if $\mathcal{D} = \{\mathsf{static}, \mathsf{dynamic}\}$ and $\mathsf{static} \sqsubseteq \mathsf{dynamic}$, we have

$$\begin{array}{l}\mathsf{static} \sqcup \mathsf{static} = \mathsf{static}\\ \mathsf{static} \sqcup \mathsf{dynamic} = \mathsf{dynamic} \sqcup \mathsf{static} = \mathsf{dynamic} \sqcup \mathsf{dynamic} = \mathsf{dynamic}\end{array}$$

$$\begin{array}{l}\mathsf{static} \sqcap \mathsf{static} = \mathsf{static} \sqcap \mathsf{dynamic} = \mathsf{dynamic} \sqcap \mathsf{static} = \mathsf{static}\\ \mathsf{dynamic} \sqcap \mathsf{dynamic} = \mathsf{dynamic}\end{array}$$

Given a set of binding-times W, we define $\sqcup W$ as follows (in a similar way we define $\sqcap W$):

$$\sqcup W \;=\; \begin{cases} \mathsf{static} & \text{if } W = \emptyset \\ b & \text{if } W = \{b\} \\ b_1 \sqcup (b_2 \sqcup (\ldots \sqcup b_{n-1}) \sqcup b_n) & \text{if } W = \{b_1, \ldots, b_n\},\ n > 0 \end{cases}$$

In the following, a *pattern* is defined as an expression of the form $p(b_1, \ldots, b_n)$ where p/n is a predicate symbol and $b_1, \ldots, b_n$ are binding-times. Given a binding-time domain, we consider an associated domain of *abstract substitutions* that map variables to binding-times. We introduce the following auxiliary functions that deal with patterns and abstract substitutions:

- Given a pattern $p(b_1, \ldots, b_n)$, the function $\bot(p(b_1, \ldots, b_n))$ returns a new pattern $p(\mathsf{static}, \ldots, \mathsf{static})$ where all arguments are static.
- Given an atom A and a pattern pat with $\mathit{pred}(A) = \mathit{pred}(\mathit{pat})$,[3] the partial function $\mathit{asub}(A, \mathit{pat})$ returns an abstract substitution for the variables of A according to pattern pat. For instance, for a simple binding-time domain $\mathcal{D} = \{\mathsf{static}, \mathsf{dynamic}\}$, function asub is defined as follows:[4]

  $$\begin{array}{l} \mathit{asub}(p(t_1, \ldots, t_n), p(b_1, \ldots, b_n)) \\ \qquad\qquad = \{x/b \mid x \in \mathcal{V}ar(p(t_1, \ldots, t_n)) \wedge b = \sqcap\{b_i \mid x \in \mathcal{V}ar(t_i)\}\} \end{array}$$

  Roughly speaking, a variable that appears only in one argument t_i will be mapped to b_i; if the same variable appears in several arguments, then it is mapped to the greatest lower bound of the corresponding binding-times of these arguments. For instance, we have

  $$\mathit{asub}(p(X, X), p(\mathsf{static}, \mathsf{dynamic})) = \{X/\mathsf{static}\}$$

  Observe that the greatest lower bound is used to compute the less dynamic binding-time of a variable when it is bound to different values.
- Given an atom $A = p(t_1, \ldots, t_n)$ and an abstract substitution σ, function $\mathit{pat}(A, \sigma)$ returns a pattern in which the binding-time of every argument is determined by the abstract substitution. For the binding-time domain $\mathcal{D} = \{\mathsf{static}, \mathsf{dynamic}\}$, this function can be formally defined as follows:

  $$\mathit{pat}(A, \sigma) = p(b_1, \ldots, b_n) \text{ where } b_i = \sqcup\{x\sigma \mid x \in \mathcal{V}ar(t_i)\},\ i = 1, \ldots, n$$

  E.g., we have $\mathit{pat}(p(X, X), \{X/\mathsf{static}\}) = p(\mathsf{static}, \mathsf{static})$ and $\mathit{pat}(q(f(X, Y)),$ $\{X/\mathsf{static}, Y/\mathsf{dynamic}\}) = q(\mathsf{dynamic})$.

Now, we present our BTA algorithm in a stepwise manner.

2.1 Call and Success Pattern Analysis

The first step towards a BTA is basically a simple call and success pattern analysis parameterized by the considered binding-time domain $(\mathcal{D}, \sqsubseteq)$. The algorithm is shown in Fig. 1. Here, we keep a *memo* table μ with the call and success patterns already found in the analysis, i.e., if $\mu(\mathit{cpat}) = \mathit{spat}$ then we have a call pattern cpat with associated success pattern spat; initially, all success patterns have static arguments. In order to add new entries to the memo table, we use the following function *addentry*:

$$\mathit{addentry}(\mathit{pattern}) = \textbf{if } \mathit{pattern} \notin \mathit{dom}(\mu) \textbf{ then } \mu(\mathit{pattern}) := \bot(\mathit{pattern}) \textbf{ fi}$$

where the notation $\mu(\mathit{pattern}) := \bot(\mathit{pattern})$ is used to denote an update of μ.

Basically, the algorithm takes a logic program P and an entry pattern *epat* and, after initializing the memo table, enters a loop until the memo table reaches a fixed point. Every iteration of the main loop proceeds as follows:

[3] Auxiliary function *pred* returns the predicate name and arity of an atom or pattern.
[4] $\mathcal{V}ar(s)$ denotes the set of variables in the term or atom s.

```
1. **Input:** a program $P$ and an entry pattern $epat$
2. **Initialisation:** $\mu := \emptyset$; $addentry(epat)$
3. **repeat**
   **for** all $cpat \in dom(\mu)$:
       for all clauses $H \leftarrow B_1, \ldots, B_n \in P$, $n \geqslant 0$, with $pred(H) = pred(cpat)$:
       (a) $\sigma_0 := asub(H, cpat)$
       (b) **for** i = 1 to n:
               $\sigma_i := get(B_i, \sigma_{i-1})$
       (c) $\mu(cpat) := \mu(cpat) \sqcup pat(H, \sigma_n)$
   **until** $\mu$ doesn't change
```

Fig. 1. Call and success pattern analysis

– for every call pattern *cpat* with a matching clause $H \leftarrow B_1, \ldots, B_n$, we first compute the entry abstract substitution $asub(H, cpat)$;
– then, we use function *get* to propagate binding-times through the atoms of the body, thus updating correspondingly the memo table with the call and (initial) success patterns for every atom;
– finally, we update in the memo table the success pattern associated to the call pattern *cpat* using the exit abstract substitution of the clause.

Function *get* is used to propagate binding-times through the atoms of the bodies as follows:

$$get(B, \sigma) \;=\; addentry(pat(B, \sigma));\ \textbf{return}\ asub(B, \mu(pat(B, \sigma)))$$

In the next section, we present a BTA that slightly extends this algorithm.

2.2 A BTA Ensuring Local and Global Termination

In contrast to the call and success pattern analysis of Fig. 1, a BTA should annotate every call with either unfold or memo so that

– 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.

Figure 2 shows a BTA that slightly extends the call and success pattern analysis of Fig. 1 (differences appear in a box). The main changes are as follows:

– We consider that each call B is uniquely identified by a program point $ppoint(B)$. The algorithm keeps track of the considered program points using the set *memo* (initially empty).
– The function for propagating binding-times, now called get', takes an additional parameter: the program point of the considered atom; function get' is

```
1. Input: a program $P$ and an entry pattern $epat$
2. Initialisation: $\mu := \emptyset$; $addentry(epat)$; $\boxed{memo := \emptyset}$
3. repeat
   for all $cpat \in dom(\mu)$:
       for all clauses $H \leftarrow B_1, \ldots, B_n \in P$, $n \geqslant 0$, with $pred(H) = pred(cpat)$:
       (a) $\sigma_0 := asub(H, cpat)$
       (b) for i = 1 to n:
               $\sigma_i := \boxed{get'(B_i, \sigma_{i-1}, ppoint(B_i))}$
       (c) $\mu(cpat) := \mu(cpat) \sqcup pat(H, \sigma_n)$
   until $\mu$ doesn't change
```

Fig. 2. A BTA ensuring local and global termination

defined as follows:

$$\begin{array}{lll} get'(B,\sigma,pp) & = & \textbf{if } unfold(pat(B,\sigma)) \wedge pp \notin memo \\ & & \quad \textbf{then return } get(B,\sigma) \\ & & \quad \textbf{else } addentry(gen(pat(B,\sigma))); memo := memo \cup \{pp\} \\ & & \qquad \textbf{return } \sigma \textbf{ fi} \\ get(B,\sigma) & = & addentry(pat(B,\sigma));\ \textbf{return } asub(B, \mu(pat(B,\sigma))) \end{array}$$

(note that function *get* is the same as the one used in Fig. 1).

- Auxiliary functions *gen* and *unfold* are defined as follows:
  - Given a pattern *pat*, function *unfold*(*pat*) returns true if *pat* is safe for unfolding, i.e., if it guarantees *local* termination.
  - Given a pattern *pat*, function *gen*(*pat*) returns a generalization of *pat* (i.e., $pat \sqsubseteq gen(pat)$) that ensures *global* termination.

  Precise definitions for *gen* and *unfold* will be presented in the next section.

3 Size-Change Termination Analysis

In this section, we first present the basis of the size-change analysis of [18], which is used to check the (quasi-)termination of a program. Then, we introduce appropriate definitions for functions *unfold* (local termination) and *gen* (global termination) which are based on the output of the size-change analysis.

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 [18], reduction orders $(\succsim, \succ)$ *induced* from symbolic norms $||\cdot||$ are used:

Definition 1 (symbolic norm [16]). *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 substitutions σ that make $||t_1\sigma||$ 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.,

$$\begin{array}{r} f(X,Y) \succ_{ts} f(X,a) \text{ since } ||f(X,Y)||_{ts} = X + Y + 2 > X + 2 = ||f(X,a)||_{ts} \\ [X|R] \succsim_{ll} [s(X)|R] \text{ since } ||[X|R]||_{ll} = R + 1 \geqslant R + 1 = ||[s(x)|R]||_{ll} \end{array}$$

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—the *output nodes*—and the arguments of B_i—the *input nodes*—when the size of the corresponding terms decrease w.r.t. a given reduction pair $(\succsim, \succ)$. A size-change graph is thus a bipartite labelled graph $\mathcal{G} = (V, W, E)$ where V and W are the labels of the output and input nodes, respectively, and $E \subseteq V \times W \times \{\succsim, \succ\}$ are the edges.

Example 1. Consider the following simple program:

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

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

$$\begin{tikzcd}
\mathcal{G} : iList & \longrightarrow & add & & \mathcal{G}' : iList & \longrightarrow & incList \\
1_{iList} \arrow[rrd, "\succsim_{ts}"] & & 1_{add} & & 1_{iList} & & 1_{incList} \\
2_{iList} & & 2_{add} & & 2_{iList} \arrow[rru, "\succsim_{ts}"] & & 2_{incList} \\
3_{iList} \arrow[rruu, "\succsim_{ts}"] & & 3_{add} & & 3_{iList} \arrow[rru, "\succsim_{ts}"] & & 3_{incList} \\
4_{iList} \arrow[rru, "\succ_{ts}"] & & & & 4_{iList} \arrow[rru, "\succ_{ts}"] & &
\end{tikzcd}$$

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.

Example 2. For the program of Example 1, 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.

3.1 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. Then, we present an appropriate definition for the function *unfold* of the BTA shown in Fig. 2.

Let us first recall the notion of *instantiated enough* w.r.t. a symbolic norm from [16]: 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 [18]). *Let P be a program and let $(\succsim, \succ)$ be a reduction pair induced from 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 $p(t_1, \ldots, t_n) \in \mathit{calls}_P^{\mathcal{R}}(A)$, 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 of Example 2, computations terminate whenever either the first or the last argument of *incList* 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 first 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-times of the computed call patterns. In the following, we extend the notion of "instantiated enough" to binding-times as follows: a binding-time b is instantiated enough w.r.t. a symbolic norm $||\cdot||$ if, for all term t approximated by the binding-time b, t is instantiated enough w.r.t. $||\cdot||$.

Now, we introduce an appropriate definition of *unfold*:

Definition 2 (local termination). *Let P be a program and let $(\succsim, \succ)$ be a reduction pair induced from a symbolic norm $||\cdot||$. Let $\mathcal{G}$ be the idempotent size-change graphs from the size-change analysis and $pat = p(b_1,\ldots,b_n)$ be a pattern. Function unfold(pat) returns true if every 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 b_i is instantiated enough w.r.t. $||\cdot||$.*

For instance, given the idempotent size-change graphs of Example 2, we have

$$unfold(incList(\mathsf{static}, \mathsf{dynamic}, \mathsf{dynamic})) = true$$

since there is an edge $1_{incList} \xrightarrow{\succ} 1_{incList}$ and the binding-time static is clearly instantiated enough w.r.t. any norm. In contrast, we have

$$unfold(incList(\mathsf{dynamic}, \mathsf{static}, \mathsf{dynamic})) = false$$

since only the edges $1_{incList} \xrightarrow{\succ} 1_{incList}$ and $3_{incList} \xrightarrow{\succ} 3_{incList}$ are labeled with $\succ$ and the binding-time dynamic is not instantiated enough w.r.t. any norm.

3.2 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.*

In principle, following the results in [18], function *gen* could be defined as follows: Given a pattern $pat = p(b_1,\ldots,b_n)$, function $gen(pat)$ returns $p(b'_1,\ldots,b'_n)$ where $b'_i = b_i$ if every idempotent size-change graph for p/n computed by the size-change analysis contains an edge $i_p \xrightarrow{R} i_p$, $R \in \{\succ, \succsim\}$, and dynamic otherwise. Furthermore, the considered reduction pair should be induced from a *bounded* symbolic norm $||\cdot||$, i.e., a norm such that the set $\{s \mid ||t|| \geqslant ||s||\}$ contains a finite number of nonvariant terms for any term t.

A main limitation of this approach comes from requiring $i_p \xrightarrow{R} i_p$, $R \in \{\succ, \succsim\}$, which is too restrictive for some examples. Consider, e.g., the size-change graphs of Example 2. Here, we would have $gen(iList(b_1, b_2, b_3, b_4)) = iList(\mathsf{dynamic}, b_2, b_3, b_4)$ for any binding-times b_1, b_2, b_3, b_4 since there is no edge $1_{iList} \xrightarrow{R} 1_{iList}$, $R \in \{\succ, \succsim\}$.

However, any sequence of calls to predicate *iList* is clearly quasi-terminating because the size of all four predicate arguments is bounded by the size of *some*—not necessarily the same—argument of the previous call. Therefore, in this paper, we consider the following improved definition of *gen*:

Definition 3 (global termination). *Let P be a program and let $(\succsim, \succ)$ be a reduction pair induced from a bounded symbolic norm $||\cdot||$. Let $\mathcal{G}$ be the idempotent size-change graphs computed by the size-change analysis. Given a pattern $pat = p(b_1, \ldots, b_n)$, function $gen(pat)$ returns $p(b'_1, \ldots, b'_n)$ where $b'_i = b_i$ if every size-change graph for p/n in $\mathcal{G}$ contains an edge $j_p \xrightarrow{R} i_p$, $R \in \{\succ, \succsim\}$, for some $j \in \{1, \ldots, n\}$, and* dynamic *otherwise.*

Now, given the idempotent size-change graphs of Example 2, we have $gen(pat) = pat$ for all pattern pat since there is an entry edge to every predicate argument, i.e., no generalisation is required at the global level.

Another limitation of [18] comes from the use of bounded norms, which excludes for instance the use of the well-known list-length norm. This approach, however, might be too coarse to produce useful results in some cases. In fact, by taking into account that some generalisation can be done during the specialisation process (i.e., at the global level), a weaker, more useful condition can be formulated.

Consider, e.g., the program $\{p([X]) \leftarrow p([s(X)]).\}$. This program cannot be proved quasi-terminating according to [18] because the symbolic list-length norm cannot be used; actually, the program is not quasi-terminating:

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

However, it can be acceptable for partial evaluation as long as those symbols that are not taken into account by this norm are generalised in the global level, i.e., as long as the list arguments are replaced with fresh variables at the global level.

Further details on this extension can be found in [12].

4 The BTA in Practice

Our new binding-time analysis is still being continuously extended and improved. The implementation was done using SICStus Prolog and provides a command-line interface. The BTA is by default polyvariant (but can be forced to be monovariant) and uses a domain with the following values: static, list_nv (for lists of non-variable terms), list, nv (for non-variable terms), and dynamic. The user can also provide hints to the BTA (see below). The implemented size-change analysis uses a reduction pair induced from the symbolic term-size norm.

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 run using SICStus Prolog 4.04, LOGEN and its generated specialisers were run using Ciao Prolog 1.13. We compared the results against the online specialiser ECCE [15], which was compiled using SICStus Prolog 3.12.8.

4.1 Results in Fully Automatic Mode

Figure 3 contains an overview of our empirical results, where all times are in seconds. A value of 0 means that the timing was below our measuring threshold. The first six benchmarks come from the DPPD [13] library, vanilla and lambdaint come from [10] and picemul from [7].

benchmark	original runtime	BTA analysis	LOGEN spec. time	specialised runtime	ECCE spec. time	specialised runtime
contains.kmp	0.18	0.01	0.002	0.18	0.11	0.02
imperative-power	0.33	2.35	0.009	0.38	0.82	0.15
liftsolve.app	0.46	0.02	0.014	0.28	0.10	0.02
match-kmp	1.82	0.00	0.002	1.52	0.02	0.91
regexp.r3	1.71	0.01	0.005	0.86	0.05	0.81
ssuply	0.23	0.01	0.001	0.00	0.02	0.00
vanilla	0.19	0.01	0.004	0.14	0.04	0.03
lambdaint	0.60	0.17	0.005	0.61	0.26	*0.02
picemul	-	0.31	4.805	-	576.370	-

Fig. 3. Empirical Results

DPPD. Let us first examine some the results from Fig. 3 for the DPPD benchmarks [13]. First, we consider a well-known match-kmp benchmark. The original run time for 100,000 executions of the queries from [13] was 1.82 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 1.52 s (i.e., a speedup of 1.20). For comparison, ECCE took 0.02 s to specialise the program; the resulting specialised program is faster still (0.91 s, i.e., a speedup of 2), as the online specialiser was able to construct a KMP-like specialised pattern matcher. So, our offline approach achieves some speedup, but the online specialiser is (not surprisingly) considerably better. However, the specialisation process itself is (again, not surprisingly) much faster using the offline approach.

A good example is the regular expression interpreter from [13] (benchmark regexp.r3 in Fig. 3). Here, the original took 1.71 s for 100,000 executions of the queries (r3 in [13]). Our BTA took 0.01 s, LOGEN took 0.005 s to specialise the program, which then took 0.86 s to run the queries (a speedup of 1.99). ECCE took 0.05 s to specialise the program; the specialised program runs slightly faster (0.81 s; a speedup of 2.11). So, here we obtain almost the same speedup as the online approach but using a much faster and predictable specialisation process. A similar outcome is achieved for the ssuply benchmark, where we actally obtain the same speedup as the online approach.

For liftsolve.app (an interpreter for the ground representation specialised for append as object program) we obtain a reasonable speedup, but the specialised program generated by ECCE is more than 10 times faster. The most disappointing benchmark is probably imperative-power, where the size change analysis takes over two seconds[5] and the specialised program is slower than the original.

In summary, for the DPPD benchmarks we obtain usually very good BTA speeds (apart from imperative-power), very good specialisation speeds, along with a mixture of some good and disappointing speedups.

PIC Emulator. To validate the scalability of our BTA we have tried our new BTA on a larger example, the PIC processor emulator from [7]. It consists of 137 clauses and 855 lines of code. The purpose here was to specialise the PIC emulator for a particular PIC machine program, in order to run various static analyses on it.[6] The old BTA from [4] took 1 m 39 s (on a Linux server which corresponds roughly to 67 seconds on the MacBook Pro).[7] Furthermore the generated annotation file is erroneous and could not be used for specialisation. With our new BTA a correct annotation is generated less than half a second; the ensuing specialisation by LOGEN took 4.8 s. The generated code is very similar to the one obtained using a manually constructed annotation in Section 3 of [7] or in [14]. In fact, it is slightly more precise and with a simple hint (see Sect. 4.2), we were able to *reduce* specialisation so as to obtain the exact same code as [14] for the main interpreter loop. Also, with ECCE it took 9 m 30 s to construct a (very large) specialised program.

In summary, this example clearly shows that we have attained our goal of being able to successfully analyse medium-sized examples with our BTA.

Vanilla and Lambda Interpreter. We now turn our attention to two interpreters from [10]. For vanilla (a variation of the vanilla metainterpreter specialised for double-append as object program) we obtain a reasonable speedup, but the specialised program generated by ECCE is more than 4 times faster.

A more involved example, is the lambda interpreter for a small functional language from [10]. This interpreter contains some side-effects and non-declarative features. It can still be run through ECCE, but there is actually no guarantee that

[5] This is mainly due to the large number (657) of size change graphs generated.

[6] The emulator cannot be run as is using an existing Prolog system, as the built-in arithmetic operations have to be treated like constraints.

[7] We were unable to get [4] working on the MacBook Pro and had to resort to using our webserver (with 26 MLIPS compared to the MacBook Pro's 38 MLIPS).

that ECCE will preserve the side-effects and their order. (This is identified by the asterisk in the table; however, in this particular case, the specialised program is correct.) Our BTA and LOGEN are very fast, but unfortunately resulting in no speedup over the original. Still, the BTA from [4] cannot cope with the program at all and we at least obtain a correct starting point. In the next subsection we show that through the use of some selective hints, we can actually obtain very good speedups for this example, as well as for all examples we have seen so far.

4.2 Improving the Results with Hints

While the above experiments show that we have basically succeeded in obtaining a fast BTA, the specialisation results are still unsatisfactory for many examples. There are several causes for this:

1. One is a limitation of the current approach which we plan to remedy in future work: namely that when the BTA detects a dangerous loop it is sufficient to "cut" the loop once somewhere in the loop (by inserting a memoisation point in the case of local termination) and not at all points on the loop. This idea for improvement is explored further in [12].
2. Another cause regards the particular binding-time domain used. For example, the lambdaint interpreter contains an environment which is a list of bindings from variables to values, such as `[x/2, y/3]`. During specialisation, the length of the list as well as the variable names are known, and the values are unknown. However, the closest binding-time value is `list_nv`, meaning that the BTA and the specialiser would throw away the variable names (i.e., the specialiser will work with `[A/B,C/D]` rather than with `[x/B,y/D]`). One solution is to improve our BTA to work with more sophisticated binding-time domains, possibly inferring the interesting abstract values using a type inference. Another solution is a so-called "binding-time improvement" (bti) [8], whereby we rewrite the interpreter to work with two lists (i.e., `[x,y]` and `[2,3]`) rather than one. The first list (i.e., `[x,y]`) can now be classified as static, thereby keeping the desired information and allowing the specialiser to remove the overhead of variable lookups. We have performed this transformation for lambdaint and liftsolve.app for Fig. 4.
3. The final reason is an inherent limitation of using size-change analysis, namely the fact that the selection rule is ignored. This both gives our BTA its speed and scalability, but it also induces a precision loss. One way to solve this issue is for the user to be able to selectively insert "hints" into the source code, overriding the BTA. For the moment we support hints that force unfolding (resp. memoisation) of certain calls or predicates, as well as ways to prevent generalisation of arguments of memoised predicates. These hints can also be used as temporary fix for point 1 above.

The main idea of using hints is to have just a few of them, in the original source code in a format that a user can comprehend. Compared to editing the annotation file generated by our BTA, the advantage of hints is that the source file can still be freely edited; there is no need to synchronise annotations with edited code as in earlier work (such as the Pylogen interface [5]). Also, the propagation of binding-times is still fully performed by the BTA (and no binding-time errors can be introduced). Also, unfolding and generalisation decisions for predicates without hints are also fully taken care of by our algorithm. There is obviously the potential for a user to override the BTA in such a way that the specialisation process will no longer terminate. Note, however, that one can still use the watchdog mode [14] to pinpoint such errors.

Figure 4 contains a selection of benchmarks from Fig. 3, where we have applied hints and sometimes also binding-time improvements. One new benchmark is ctl, a CTL model checker specialised for the formula `ef(p(unsafe))` and a parametric Petri net (see, e.g., [11]).

benchmark	original	BTA	LOGEN	specialised	ECCE	specialised
contains.kmp	0.18	0.01	0.002	0.18	0.11	0.02
+ hints		0.01	0.002	0.03		
imperative-power	0.33	2.35	0.009	0.38	0.82	0.15
+ hints		2.36	0.005	0.21		
liftsolve.app	0.46	0.02	0.014	0.28	0.10	0.02
bti + hints	0.46	0.02	0.002	0.03		
vanilla	0.19	0.01	0.004	0.14	0.04	0.03
+ hints		0.01	0.002	0.05		
lambdaint	0.60	0.17	0.005	0.61	0.26	*0.02
by hand		hand	0.002	0.05		
+hints		0.16	0.009	0.33		
bti	0.69	0.17	0.005	0.67	0.26	*0.02
bti+hints		0.19	0.006	0.05		
ctl	4.64	0.03	0.005	7.64	0.29	0.67
+ hints	-	0.03	0.003	0.63		

Fig. 4. Empirical Results with Hints

For vanilla, the following two hints are enough to get the Jones-optimal specialisation (i.e., the specialised program being isomorphic to the object program being interpreted):

```
'$MEMOANN'(solve_atom,1,[nv]).      '$UNFOLDCALLS'(solve(_)) :- true.
```

Without hints we do not get the optimal result for two reasons.

- First, because the termination analysis is overly conservative and does not realise that we can keep the top-level predicate symbol. The first hint remedies this. Its purpose is actually twofold, telling the BTA not to abstract nv (nonvar) binding-time values in the global control but also to raise an error if a binding-time value less precise than nv is encountered.
- Second, because the current algorithm breaks a loop at every point, while it is sufficient to break every loop once. The second hint remedies this problem, by forcing the unfold of `solve`.

For the liftsolve interpreter for ground representation, we have performed a binding-time improvement, after which one unfolding hint was sufficient to get a near optimal result. For the lambdaint interpreter, we again performed a bti, after which we get good compiled programs, corresponding almost exactly to the results obtained in [10], when hand-crafting the annotations with custom binding-times (this is the entry "hand" in Fig 4). Other large Prolog programs that we were able to successfully specialise this way were various Java Bytecode interpreters from [6] with roughly 100 clauses. Here, we were able to reproduce the decompilation from Java bytecode to CLP from [6] using our BTA together with LOGEN. In summary, our new BTA is very fast and together with some simple hints we can get both fast and good specialisation.

5 Discussion and 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 [18] 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, and excellent results can be obtained with the use of a few selective hints, there is still much room for improvement.

One way to improve the accuracy of the BTA consists in also running a standard left-termination analysis (such as, e.g., the termination analysis based on the abstract binary unfoldings [3]), so that left-terminating atoms (i.e., atoms whose computation terminate using the leftmost selection rule) are marked with a new annotation call (besides unfold and memo, which keep the same meaning). Basically, while atoms annotated with unfold allow us to perform an unfolding step and then the annotations of the derived goal must be followed, atoms annotated with call are *fully* unfolded. In principle, this extension may allow us to avoid some of the loss of accuracy due to considering a *strong* termination analysis during the BTA.

Another way to improve the accuracy of the BTA is to generate semi-online annotations. In other words, instead of generating `memo` we produce the annotation `online`, which tries to unfold an atom 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: the online overhead is only applied to those places in the source code where the BTA was imprecise.

In conclusion, our BTA is well suited to be applied to larger programs. The accuracy of the annotations is not yet optimal, but in conjunction with hints we have obtained both a fast BTA with very good specialisation results.

Acknowledgments. We would like to thank Maurice Bruynooghe for suggesting the introduction of the new annotation call, as discussed above. We also gratefully acknowledge the anonymous referees as well as the participants of LOPSTR 2008 for many useful comments and suggestions.

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. K. 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. LOPSTR'04*, pages 53–68. Springer LNCS 3573, 2005.
5. S.-J. Craig. *Practicable Prolog Specialisation.* PhD thesis, University of Southampton, U.K., June 2005.
6. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Improving the decompilation of java bytecode to prolog by partial evaluation. *Electr. Notes Theor. Comput. Sci.*, 190(1):85–101, 2007.
7. K. S. Henriksen and J. P. Gallagher. Abstract interpretation of pic programs through logic programming. In *SCAM*, pages 184–196. IEEE Computer Society, 2006.
8. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
9. 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.
10. 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.
11. 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.
12. M. Leuschel, S. Tamarit, and G. Vidal. Improving size-change analysis in offline partial evaluation. In P. Arenas and D. Zanardini, editors, *WLPE*, 2008.
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, S.-J. Craig, and D. Elphick. Supervising offline partial evaluation of logic programs using online techniques. In *Proceedings LOPSTR'06*, LNCS 4407, pages 43–59, 2006.
15. 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.
16. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *Proc. ICLP'97*, pages 63–77. The MIT Press, 1997.
17. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
18. G. Vidal. Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In *Proc. PEPM'07*, pages 51–60. ACM Press, 2007.