Abstract Partial Evaluation for Termination Analysis

Lior Tamary and Michael Codish

Department of Computer Science, Ben-Gurion University of the Negev, Israel
`(tamaryl|mcodish)@cs.bgu.ac.il`

1 Introduction

Current techniques for proving termination of logic programs focus on size and instantiation information. Structure information is ignored and for many examples this leads to imprecision in the analysis. There are two approaches to consider structure information: either to enhance the abstract domain over which the analysis is performed; or to transform the program so that structure in the concrete program is considered before abstract interpretation is applied. In [5], the authors propose a technique to transform a program so that the predicate names are adorned to reflect different structure patterns. In this approach, the adorned program is equivalent to the original program and termination analysis of the adorned program is more precise.

In this paper we present an alternative approach based on well-studied techniques of partial evaluation (unfolding) and abstract interpretation. Partial evaluation is a natural choice when the intention is to pre-evaluate just so much as to consider the structure present in the program. The advantage of this approach is that it relies completely on well-studied formal techniques.

In addition to partial evaluation we also abstract the program, maintaining structure, but possibly adding computations. This process preserves nontermination and hence is safe for termination analysis. This often simplifies the transformation.

We obtain results similar to those illustrated in [5]. We also compare our results to those obtained using ECCE [3] which is an off-the-shelf partial evaluator. Our results are more precise and analyses are considerably faster than those obtained using ECCE.

2 Partial Evaluation for Termination Analysis

When considering partial evaluation as a preprocessing for termination analysis it is important to keep in mind that unfolding does not preserve non-termination in general. Hence a preprocessed program might appear to terminate while the original does not. To illustrate the problem consider the following example.

Example 1. The program below is nonterminating for the query `?- r(A,B)`. It is intended to find two natural numbers the product and sum of which each are 0.

```
r(X,Y) :- times(X,Y,0), plus(X,Y,0).       times(0,X,0).
plus(0, X, X).                              times(s(X),Y,Z) :-
plus(s(X), Y, s(Z)) :- plus(X,Y,Z).           times(X,Y,W), plus(W,Y,Z).
```

Partial evaluation (using ECCE) results in the program consisting of the single fact `r(0,0)` which is clearly terminating for the query `?- r(A,B)`.

The ECCE [3] partial evaluator comes with a predefined setting `''t''` which is non-termination preserving and hence safe for termination analysis. This setting is used in all of our experiments.

3 Our method

Our method consists of three simple program transformations: **(a)** Binarisation — A program P is transformed to a binary program P' — this phase involves an abstraction, however termination of P' implies termination of P; **(b)** Binary unfolding — Unfolding a binary program is a simple transformation which preserves termination (and nontermination) — The main problem is to decide how much to unfold; and **(c)** Predicate renaming — This is a simple transformation along the lines of adorning [5] in which structural information is "encoded" in the predicate names. We illustrate that for the examples illustrated in [5] (and others) the TerminWeb analyzer [6] is able to prove termination after the application of the above transformations.

Binarisation: For a logic program P, the following (possibly infinite program) has the same termination behaviour (and makes observable the same answers and calls) as P [1].

$$bin(P) = \left\{ (h \leftarrow b_i)\theta \;\middle|\; \begin{array}{l} h \leftarrow b_1, \ldots, b_n \in P,\ 1 \leq i \leq n, \\ \theta \in ans_P(b_1, \ldots, b_{i-1}) \end{array} \right\}$$

where $ans_P(G)$ denotes the answer set for goal G with program P. As $bin(P)$ cannot be computed, we approximate it by taking an approximation of $ans_P(G)$. For the results of this paper we look at the approximation where $ans_P(G)$ maps all goals to the singleton $\{\epsilon\}$ (the empty substitution). We denote by $bin^a(P) = \left\{ h \leftarrow b_i \;\middle|\; h \leftarrow b_1, \ldots, b_n \in P,\ 1 \leq i \leq n \right\}$.

Example 2. The following Prolog program, quoted from [5], rewrites an expression in variable x by repeatedly applying laws of distributivity. The original statement of the problem is given in terms of rewrite systems and is described in [2].

```
dist(x,x).                           % (1)
dist(x*x,x*x).                       % (2)
dist(X+Y,U+V) :- dist(X,U), dist(Y,V).  % (3)
dist(X*(Y+Z),T) :- dist(X*Y+X*Z,T).     % (4)
dist((X+Y)*Z,T) :- dist(X*Z+Y*Z,T).     % (5)
```

The program $bin^a(P)$ is obtained by replacing the clause `(3)` by the two clauses

```
dist(X+Y,U+V) :- dist(X,U). % (3a)        dist(X+Y,U+V) :- dist(Y,V).  % (3b)
```

Theorem 1. *Termination of $bin^a(P)$ implies termination of P.*

Proof. The result follows because P and $bin(P)$ have the same termination behaviour and because $bin^a(P)$ is more general than $bin(P)$.

Binary unfolding: Binary unfolding, in contrast to general unfolding, is guaranteed to preserve termination and nontermination [1](theorem 4.3). A binary clause $a \leftarrow b$ in a binary program P can be replaced by the set of binary clauses

$$unf_P(a \leftarrow b) = \left\{ (a \leftarrow c)\theta \;\middle|\; b' \leftarrow c \in P, mgu(b, b') = \theta \right\}.$$

The unfolding operation can be applied repeatedly, however, in case the call graph of the program contains cycles, this process may not terminate. In practice, whenever it is possible to unfold a clause along a cyclic path, we do so only if the size of the clause body gets smaller according to a selected norm (we currently apply the termsize norm by default).

Example 3. Unfolding the clause `(4)` with `(3a)` from the binarised program $bin^a(P)$ of Example 2 yields: `dist(A*(B+C),D+E) :- dist(A*B,D).`

This demonstrates our strategy: Even though clause `(3a)` is cyclic, the body of the unfolded clause decreases when compared to `(4)` (in at least one of its arguments), hence we choose to follow this path for unfolding.
Unfolding $bin^a(P)$ results in the following program (where facts have been omitted as they do not contribute to non-termination).

```
dist(A+B,D+E) :- dist(A,D).          % (1')
dist(A+B,D+E) :- dist(B,E).          % (2')
dist(A*(B+C),D+E) :- dist(A*B,D).    % (3')
dist(A*(B+C),D+E) :- dist(A*C,E).    % (4')
dist((A+B)*C,D+E) :- dist(A*C,D).    % (5')
dist((A+B)*C,D+E) :- dist(B*C,E).    % (6')
```

Clauses `(3')`, `(4')`, `(5')` and `(6')` are obtained as a result of unfolding, whereas `(1')` and `(2')` remain unchanged. TerminWeb succeeds to prove termination of this program, where with the original it fails.

Predicate renaming: This is a simple program transformation to remove from the program calls to clauses with incompatible structure. Once program analysis will be applied these inconsistencies will disappear and precision will be lost.

Example 4. The program on the left is presented in [5]. We first rename each predicate occurrence to obtain the program on the right.

`loop(0,X,s(Y)) :- loop(0,s(X),Y).`	$loop_1$`(0,X,s(Y)) :-` $loop_2$`(0,s(X),Y).`
`loop(0,X,Y) :- loop(s(0),X,Y).`	$loop_3$`(0,X,Y) :-` $loop_4$`(s(0),X,Y).`
`loop(s(0),s(X),Y) :- loop(s(0),X,s(Y)).`	$loop_5$`(s(0),s(X),Y) :-` $loop_6$`(s(0),X,s(Y)).`

We then add: (a) entry points of the form $loop$`(A,B,C) :-` $loop_j$`(A,B,C)` corresponding to the new head atoms; and (b) call graph edge clauses of the form $loop_i$`(A,B,C) :-` $loop_j$`(A,B,C)` if the body atom $loop_i(\ldots)$ is unifiable with the head atom $loop_j(\ldots)$.

$loop$`(A,B,C) :-` $loop_1$`(A,B,C).`
$loop$`(A,B,C) :-` $loop_3$`(A,B,C).`
$loop$`(A,B,C) :-` $loop_5$`(A,B,C).`
$loop_2$`(A,B,C) :-` $loop_1$`(A,B,C).`
$loop_2$`(A,B,C) :-` $loop_3$`(A,B,C).`
$loop_4$`(A,B,C) :-` $loop_5$`(A,B,C).`
$loop_6$`(A,B,C) :-` $loop_5$`(A,B,C).`

TerminWeb can prove the termination of the transformed program (but not the original).

4 Experimentation

We have applied the approach to a set of benchmarks which can be found at `http://www.cs.bgu.ac.il/~mcodish/TerminWeb/` (under "Examples"). All of the transformations applied are fully implemented. Table 1 summarizes the results. The rows in the table correspond to the test programs. The columns indicate: `program` - the program, `query` - the query pattern ('b' means bound and 'f' means free), `*.pl` - termination analysis for the original program, `*.pe` - termination analysis after applying ECCE (non-termination preserving mode), `*.bin` - termination analysis after applying binarisation, `*.bin_pe` - termination analysis after applying ECCE to the binarised program, and `*.bin_unf` - termination analysis after applying binarisation and binary unfolding (as described in Section 3).

For each application of the termination analyzer the table indicates the total termination analysis time (in milliseconds) if the analysis was successful, or "no" if termination was not

[1] Selected norm is "termsize edges".
[2] The analyzer timed-out.
[3] The original program is binary to start with.

Program	Query	*.pl	*.pe	*.bin	*.bin_pe	*.bin_unf
dist	dist(b,f)	no	no	no	3040	10
	dist(f,b)	no	650	no	3490	10
d	d(b,f,f)	no	no	no	1280	20
	d(f,f,b)	no	no	no	1280	20
thrm1[1]	false(b)	no	no	no	no	20
thrm2[1]	true(b)	no	1010	no	450	20
balance	balance(b,f)	75780	∞^2	63250	43290	2880
plus	reduce(b,f)	no	1570[3]	no[3]	1570	20

Table 1. Results

proven. The analyses were performed through the web interface to TerminWeb using the default options of the analyzer unless indicated otherwise. We have focused on the cost of the termination analysis of the transformed programs, ignoring for the time being the cost of the transformations themselves. We have not compared the cost of unsuccessful analyses (also to get a "no" has a cost).

The results indicate that off-the-shelf partial evaluation is not suitable for termination analysis. This is apparent from the columns `*.pe` and `*.bin_pe`, illustrating that the analyses of the partial evaluated programs are less precise and are more costly than those obtained with simple unfolding. The purpose of the column `*.bin` is to illustrate that binarisation on its own does not solve the problem.

5 Conclusions and Ongoing Work

We illustrate that simple techniques of partial evaluation can be applied to improve the precision of termination analysis. Most likely, partial evaluators can be better tuned to perform this task. Meanwhile we have implemented straightforward transformations to illustrate the point.

An advantage in comparison to the adornments approach is that all is based on simple and well-studied transformations. A disadvantage is that it is not clear how to generalize ours for numeric computation.

We note that unfolding has been recognized important for improving the precision of termination analysis already in [4]. We note also that TerminWeb can analyse the mergesort program directly without applying unfolding.

References

1. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
2. N. Dershowitz. 33 examples of termination. In H. Comon and J.-P. Jouannaud, editors, *French Spring School of Theoretical Computer Science Advanced Course on Term Rewriting (Font Romeux, France, May 1993)*, volume 909, pages 16–26, Berlin, 1995. Springer-Verlag.
3. M. Leuschel. The ecce partial deduction system. `http://www.ecs.soton.ac.uk/~mal/systems/`.
4. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Unfolding the mystery of mergesort. In N.E. Fuchs, editor, *Proceedings of the 7th International Workshop on Logic Program Synthesis and Transformation*, volume 1463 of *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg, July 1997.
5. A. Serebrenik and D . De Schreye. Proving termination with adornments. In M. Bruynooghe, editor, *Pre-proceedings of International Symposium on Logic-based Program Synthesis and Transformation*, pages 139–154, 2003. `http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW365.pdf`.
6. C. Taboch, S. Genaim, and M. Codish. Terminweb: Semantic based termination analyser for logic programs, 2002. `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`.