Fully Automatic Binding Time Analysis for Prolog

Stephen-John Craig[1], John P. Gallagher[2], Michael Leuschel[1], and Kim S. Henriksen[2]

[1] Department of Electronics and Computer Science, University of Southampton,
Highfield, Southampton, SO17 1BJ, UK
{sjc02r,mal}@ecs.soton.ac.uk
[2] Department of Computer Science, University of Roskilde,
P.O. Box 260, D-4000 Roskilde, Denmark
{jpg,kimsh}@ruc.dk

Abstract. Offline partial evaluation techniques rely on an annotated version of the source program to control the specialisation process. These annotations guide the specialisation and have to ensure termination of the partial evaluation. We present an algorithm for generating these annotations automatically. The algorithm uses state-of-the-art termination analysis techniques, combined with a new type-based abstract interpretation for propagating the binding types. This algorithm has been implemented as part of the LOGEN partial evaluation system, and we report on performance of the algorithm on a series of benchmarks.

1 Introduction

The offline approach to specialisation has proven to be very successful for functional and imperative programming, and more recently for logic programming. Most offline approaches perform a *binding-time analysis* (BTA) prior to the specialisation phase. Once this has been performed, the specialisation process itself can be done very efficiently [10] and with a predictable outcome. Compared to online specialisation, offline specialisation is in principle less powerful (as control decisions are taken by the BTA *before* the actual static input is available), but much more efficient (once the BTA has been performed). This makes offline specialisation very useful for compiling interpreters [9]; a key application of partial evaluation. However, up to now, no automatic BTA for logic programs has been fully implemented (there are some partial implementations, cf, Section 4), requiring users to manually annotate the program. This is an error-prone process, and requires considerable expertise. Hence, to make offline specialisation accessible to a wider audience, a fully automatic BTA is essential.

Binding-Time Analysis

In essence, a *binding-time analysis* does the following: given a program and a generalisation of the input available for specialisation, it approximates all values within the program and generates annotations that steer the specialisation process. The partial evaluator (or the compiler generator generating the specialised partial evaluator) then uses the generated annotated program to guide the specialisation process. (This process is illustrated in Fig. 2 in the appendix).

Within the annotated program there are two types of annotations.

- *Filter declarations*: these give each argument of a predicate a *binding-type*. A binding-type indicates something about the structure of an argument. For example, it could

indicate which parts of the argument are *dynamic* (possibly unbound at specialisation time) and which parts are *static* (definitely known at specialisation time). Other more precise binding types are also possible. These annotations influence the *global control*, in that dynamic parts are generalised away (i.e., replaced by fresh variables) and known parts are kept unchanged.
- *Clause annotations*: these indicate how every call in the body should be treated during specialisation. Essentially, these annotations determine whether a call is unfolded at specialisation time or at run time. These influence the *local control* [12]. For the LOGEN system [10] the main annotations are: *unfold* (the call is unfolded), *memo* (the call is not unfolded but a specialised version is produced), *call* (the call is fully executed without further intervention by the partial evaluator), *rescall* (the call is left unmodified in the residual code).

2 The Algorithm

We now outline the main steps of the automatic algorithm as depicted in **Fig. 1**. The core of the algorithm is a loop which propagates the binding types with respect to the current clause annotations (step 1), generates the abstract binary program (steps 2 and 3) and checks for termination conditions (step 4), modifying the annotations accordingly. Initially, all calls are annotated as *unfold* (or *call* for built-ins), with the exception of imported predicates which are annotated as *rescall* (step 0). Annotations can be changed to *memo* or *rescall*, until termination is established. Termination of the main loop is ensured since there is initially only a finite number of *unfold* or *call* annotations, and each iteration of the loop eliminates one or more *unfold* or *call* annotation.

2.1 Binding Type Propagation (Filter Propagation)

The basis of the BTA is a classification of arguments using abstract values. Using this classification, we obtain a binding-time division, which defines the pattern of binding types occurring in each predicate call in an execution of the program for a given initial goal. This information in turn determines which calls to unfold and which to keep in the residual programs.

The use of *static-dynamic* binding types was introduced for functional programs, and has been used in BTAs for logic programs [13]. A simple classification of arguments into "fully known" or "totally unknown" is often unsatisfactory in logic programs, where partially unknown terms occur frequently at runtime, and would prevent specialising of many "natural" logic programs such as the vanilla meta-interpreter [7, 11] or most of the benchmarks from the DPPD library [8].

We outline a method of describing more expressive binding types and propagating them. The analysis framework is described elsewhere [5]. The method is flexible, allowing standard static and dynamic binding types to be freely mixed with program-specific types.

Regular Binding Types Instantiation modes can be coded as regular types. A regular type t is defined by a rule $t = f_1(t_{1,1}, \ldots, t_{1,m_1}); \ldots; f_n(t_{n,1}, \ldots, t_{n,m_n})$ where $f_1, \ldots, f_n$ are function symbols (possibly not distinct) and for all $1 \leq i \leq n$ and $1 \leq j \leq m_i$, m_i is the arity of f_i, and $t_{i,j}$ are regular types. The interpretation of such rules is well understood in the framework of regular tree grammars or finite tree automata [3].

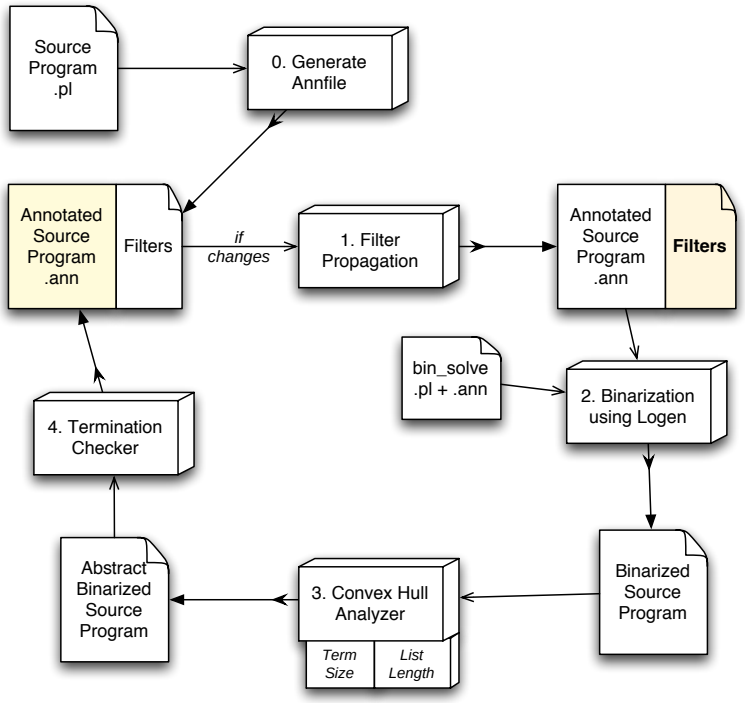

Fig. 1. Overview

The set of ground terms over a given signature can be described using regular types, as can the set of non-ground terms, the set of variables, and the set of non-variable terms. If we assume that the signature is $\{[], [.|.], s, 0, v\}$ with the usual arities, then the definitions of the types *ground term* (*static*), *variable* (*var*) and *any term* (*dynamic*) are as follows.

$$static = 0; []; [static|static]; s(static)$$
$$var = v$$
$$dynamic = 0; []; [dynamic|dynamic]; s(dynamic); v$$

The constant v is assumed to be a special constant not occurring in programs or goals. Therefore any argument of type *var* is possibly a variable. In addition to modes, regular types can describe common data structures. The set of all lists, for instance, is given as $list = []; [dynamic|list]$. We can describe the set of lists of list by the type $listlist = []; [list|listlist]$.

Type Determinization We take a given set of regular types, and transform them into a set of *disjoint* regular types. Our method is completely independent of the actual types, apart from requiring that they should include the types *static* and *dynamic*. This process is called *determinization* and is a standard operation on finite tree automata [3]. Given the types *dynamic*, *static*, *var* and *list* as shown above, determinization yields

definitions of the disjoint sets of terms, which are (1) ground lists, (2) non-ground lists, (3) ground non-lists, (4) non-ground, non-variable non-lists and (5) variables.

The advantage of determinized types is that we can propagate them more precisely than non-disjoint types. Further, we can use standard techniques to propagate them. The rules for the disjoint types define a *pre-interpretation* of the signature Σ. An abstract interpretation based on this pre-interpretation gives the least model over the given types. From the model and an initial typed goal we compute accurate call patterns (*filters*) for each body atom.

Analysing Annotated Programs The standard methods for computing an abstract model and abstract call patterns have to be modified in our procedure, since some body calls may be marked as *memo*. That is, they are not to be unfolded but rather kept in the specialised program. This obviously affects propagation of binding types. When building the abstract model of a program, we simply delete memo calls from the program, as they cannot contribute anything to the model. However, we do require call patterns for memo calls: when deriving the call pattern, say for atom B_j in clause $H \leftarrow B_1, \ldots, B_j, \ldots, B_n$, we ignore the answers to memo calls occurring in $B_1, \ldots, B_{j-1}$.

2.2 Termination Checking

There are two separate termination requirements during partial evaluation. *Local termination* concerns the avoidance of infinite derivations in which atoms annotated as *unfold* or *call* are selected. *Global termination* concerns the set of calls that are unfolded in the course of the partial evaluation, which should be finite. Finiteness of the set is ensured by a generalisation operation driven by the filter declarations.

Local Termination Based On Binary Clause Semantics We approach the *local termination* problem using the *binary clause semantics* [2], a representation of a program's computations that makes it possible to reason about loops and hence termination.

Informally, the binary clause semantics of a program P is the set of all pairs of atoms (called binary clauses) $p(\bar{X})\theta \leftarrow q(\bar{t})$ such that p is a predicate, $p(\bar{X})$ is a most general atom for p, and there is a finite derivation (with leftmost selection rule) $\leftarrow p(\bar{X}), \ldots, \leftarrow (q(\bar{t}), Q)$ with computed answer substitution θ. In other words a call to $p(\bar{X})$ is followed some time later by a call to $q(\bar{t})$, computing a substitution θ. We modify the semantics to include *program point* information for each call in the program. A clause $p(ppM, \bar{X})\theta \leftarrow q(ppN, \bar{t})$ details that the call $p(\bar{X})$ at program point ppM is followed sometime later by a call to $q(\bar{t})$ at program point ppN. This extra precision is required to correctly identify the actual unsafe call.

The binary semantics is in general infinite, but we make a safe approximation of the set of binary clauses using abstract interpretation. We use a domain of convex hulls (currently the convex hull analyser is derived from one kindly supplied by Samir Genaim and Michael Codish [6]) to abstract the set of binary clauses with respect to a selected norm. Using such an abstraction, we aim to obtain a finite set of binary clauses and a set of constraints representing a linear relationship between the sizes of the respective concrete arguments. In particular, loops are represented by binary clauses with the same predicate occurring in the head and body. Termination proofs require that for every abstract binary clause between p and p (at the same program point) there is a strict reduction in the size for some rigid argument.

3 Experimental Results

The automatic binding time analysis detailed in this paper is implemented as part of the LOGEN partial evaluation system. The system has been tested using benchmarks derived from the DPPD benchmark library [8]. The figures in Table 1 present the timing results[3] from running the BTA on an unmodified program given an initial filter declaration.

Benchmark	BTA	Original	Specialised	Speedup	Relative
match	690ms	160ms	110ms	1.45	0.69
vanilla	210ms	130ms	60ms	2.17	0.46
advisor	280ms	40ms	20ms	2.00	0.50
regexp	340ms	560ms	340ms	1.67	0.60
contains	1130ms	550ms	320ms	1.72	0.58
simple interpreter	820ms	90ms	10ms	9.09	0.11
medium interpreter	2720ms	430ms	10ms	33.33	0.03
Average					0.42

Table 1. Benchmark Figures for Automatic Binding Time Analysis

4 Related Work and Conclusion

To the best of our knowledge, the first binding-time analysis for logic programming is [1]. The approach of [1] obtains the required annotations by analysing the behaviour of an *online* specialiser on the subject program. Unfortunately, the approach was overly conservative. Indeed, [1] decides whether or not to unfold a call based on the original program, not taking current annotations into account. This means that a call can either be completely unfolded or not at all. Also, the approach was never fully implemented and integrated into a partial evaluator.

In Section 6 of [10] a more precise BTA has been presented, which has been partially implemented. It is actually the precursor of the BTA here. However, the approach was not fully implemented and did not consider the issue of filter propagation (filters were supposed to be correct). Also, the identification of unsafe calls was less precise as it did not use the binary clause semantics with program points (i.e., calls may have been classified as unsafe even though they were not part of a loop).

[16] is probably the most closely related work to ours. This work has a lot in common with ours, and we were unaware of this work while developing our present work.[4] Let us point out the differences. Similar to [10], [16] was not fully implemented (as far as we know, based on the outcome of the termination analysis, the user still had to manually update the annotations by hand) and also did not consider the issue of filter propagation. Also, [16] cannot handle the `nonvar` annotation (this means that, e.g., it can only handle the vanilla interpreter if the call to the object program is fully static). However, contrary to [10], and similar to our approach, [16] does use the binary clause semantics. It even uses program point information to identify non-terminating calls. However, we have gone one step further in using program point information, as we will only look for loops from one program point back to itself. Take for example the following program:

[3] The execution time for the Original and Specialised code is based on executing the benchmark query 50,000 times on a 2.4Ghz Pentium with 512Mb running SICStus Prolog 3.11.1. The specialisation times for all examples was under 10ms.

[4] Thanks for reviewers of LOPSTR'04 for pointing this work out to us.

```
p(a) :- q(a).
q(a) :- q(b).
q(b) :- q(b).
```

Both our approach and [16] will mark the call `q(a)` as unfoldable and the call `q(b)` in clause 3 as unsafe. However, due to the additional use of program points, we are able to mark the call `q(b)` in clause 2 as unfoldable (as there is no loop from that program point back to itself), whereas we believe that [16] will mark it as unsafe. We believe that this extra precision may pay off for interpreters. Finally, due to the use of our meta-programming approach we can handle the full LOGEN annotations (such as `call`, `rescall`, `resif`,...) and can adapt our approach to compute memoisation loops and tackle global termination.

The papers [14, 15, 17] describe various BTAs for Mercury, even addressing issues such as modularity and higher-order predicates. An essential part of these approaches is the classification of unifications (using Mercury's type and mode information) into tests, assignments, constructions and deconstructions. Hence, these works cannot be easily ported to a Prolog setting, although some ideas can be found in [17].

Currently our implementation guarantees correctness and termination at the local level, and correctness but not yet termination at the global level. However, the framework can very easily be extended to ensure global termination as well. Indeed, our binary clause interpreter can also compute memoisation loops, and so we can apply exactly the same procedure as for local termination. Then, if a memoised call is detected to be unsafe we have to mark the non-decreasing arguments as dynamic. Finally, as has been shown in [4], one can actually relax the strict decrease requirement for global termination (i.e., one can use $\leq$ rather than $<$), provided so-called "finitely partitioning" norms are used.

Acknowledgements Thanks to Michael Codish and Samir Genaim for the convex hull analyser. Thanks to Dan Elphick for his work on the Python mode for Logen.

References

1. M. Bruynooghe, M. Leuschel, and K. Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Proceedings of the European Symposium on Programming (ESOP'98)*, LNCS 1381, pages 27–41. Springer-Verlag, April 1998.
2. M. Codish and C. Taboch. A semantic basic for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
3. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille3.fr/tata, 1999.
4. S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K. Sagonas. Termination analysis for tabled logic programming. In N. Fuchs, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, LNCS 1463, pages 111–127, Leuven, Belgium, July 1998.
5. J. P. Gallagher and K. Henriksen. Abstract domains based on regular types. In V. Lifschitz and B. Demoen, editors, *Proceedings of the International Conference on Logic Programming (ICLP'2004)*, LNCS. Springer Verlag, 2004. (to appear).
6. S. Genaim and M. Codish. Inferring termination conditions of logic programs by backwards analysis. In *International Conference on Logic for Programming, Artificial intelligence and reasoning*, volume 2250 of *Springer Lecture Notes in Artificial Intelligence*, pages 681–690, 2001.

7. P. Hill and J. Gallagher. Meta-programming in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–497. Oxford Science Publications, Oxford University Press, 1998.
8. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.ecs.soton.ac.uk/~mal`, 1996-2002.
9. 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.
10. 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.
11. B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995.
12. 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.
13. T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.
14. W. Vanhoof. Binding-time analysis by constraint solving: a modular and higher-order approach for mercury. In M. Parigot and A. Voronkov, editors, *Proceedings of LPAR'2000*, LNAI 1955, pages 399–416. Springer-Verlag, 2000.
15. W. Vanhoof and M. Bruynooghe. Binding-time analysis for mercury. In D. De Schreye, editor, *Proceedings of the International Conference on Logic Programming ICLP'99*, pages 500–514. MIT Press, 1999.
16. W. Vanhoof and M. Bruynooghe. Binding-time annotations without binding-time analysis. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference*, LNCS 2250, pages 707–722. Springer-Verlag, 2001.
17. W. Vanhoof, M. Bruynooghe, and M. Leuschel. Binding-time analysis for Mercury. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, LNCS this Volume. Springer-Verlag, 2004.

A Specialisation and BTA

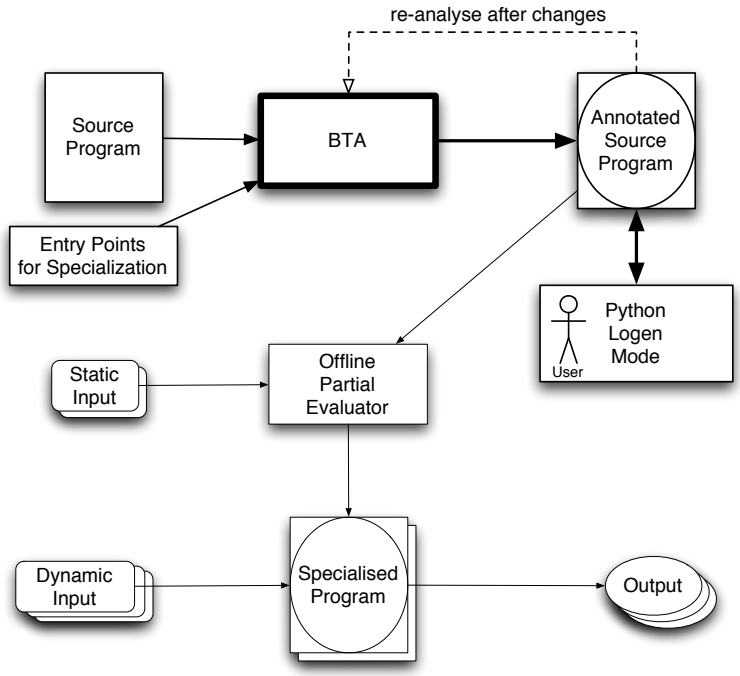

Fig. 2. The role of the BTA for offline specialisation using LOGEN

B Screenshot of LOGEN

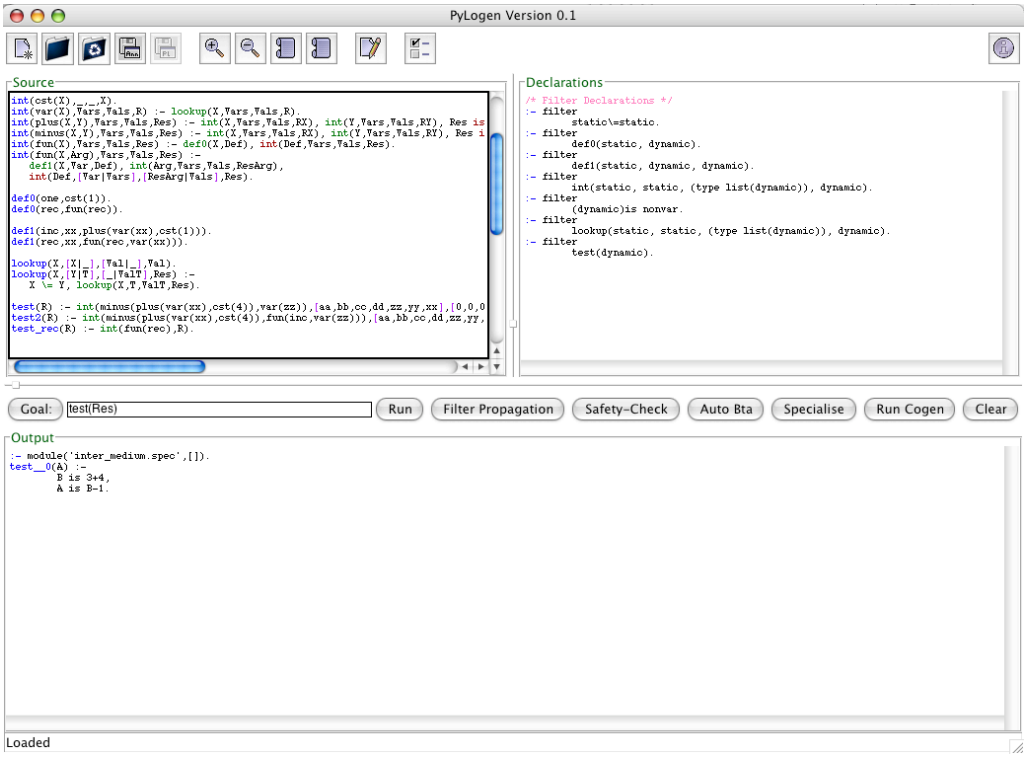

Fig. 3. Snapshot of a LOGEN session