Binding-time Annotations without Binding-time Analysis
(Extended Abstract)

Wim Vanhoof and Maurice Bruynooghe

Department of Computer Science,
K.U.Leuven, Belgium.
e-mail: {wimvh,maurice}@cs.kuleuven.ac.be

Abstract. In this work, we argue that the use of termination conditions as unfolding criteria is insufficient to obtain adequate specialisation of logic programs. We describe how a binding-time analysis for logic programs that allows more liberal unfoldings can be obtained by altering termination analysis in such a way that it proves termination *at specialisation-time* rather than *at run-time*.

1 Introduction and Motivation

Partial evaluation is a well-studied source-to-source transformation, capable of specialising a program P with respect to a part s of its input. The result is a program P_s that computes, when provided with the remaining part d of the input, the same result as the original program P on the complete input $s + d$. The general effect of partial evaluation is that the computations performed by a program are *staged*: some (ideally all) operations in P that depend only on s are performed by the specialiser; the remaining computations (those depending on d) by the residual program P_s. Partial evaluation can be used to speed up the computation of a program, in particular when the program must be run a number of times while part of its input (the part denoted by s) remains constant. Indeed, using partial evaluation, the latter computations must be performed only once to construct P_s, which can then be run any number of times with different inputs $d_1, \ldots, d_n$.

The heart of any partial evaluator is an evaluation mechanism for the language under consideration. In a logic programming setting, "evaluation" of a program corresponds to building an SLD-tree for a program/query pair $\langle P, Q \rangle$. If the program terminates, the corresponding SLD-tree is finite. In this setting, *partially available* input corresponds to a query Q' that is less instantiated than Q. Due to the nature of logic programming, the program could, in principle, simply be evaluated with respect to Q'. Most likely, however, the SLD-tree built for $\langle P, Q' \rangle$ will be infinite. Indeed, if the control flow is determined by a value that is unknown in $\langle P, Q' \rangle$, SLD-derivations of infinite length may be created resulting in a non-terminating specialisation process. Instead of building such a possibly infinite SLD-tree, a partial evaluator for logic programs builds a finite number of *finite* SLD-trees that together cover the complete computation for $\langle P, Q' \rangle$ [5]. The resulting SLD-trees are *partial*, in the sense that, while building the SLD-tree, the partial evaluator unfolds some predicate calls whereas it does not unfold others. The predicate calls that are not unfolded are said to be *residualised* – they will appear as code in the residual program.

Most work on partial evaluation in logic programming concentrates on the so-called *on-line* approach to partial evaluation [4]: during the construction of a partial SLD-tree, the partial evaluator selects each call occurring in an SLD-derivation and decides whether or not to unfold it; usually basing its decision on the structure of the SLD-tree built so far. In the *off-line* approach on the other hand, the program is first analysed by a so-called binding-time analysis (BTA). Binding-time analysis is a global analysis that takes a program and (an abstraction of) the query and generates an *annotated* version of the original program, in which every predicate call is accompanied by an instruction stating whether or not instances of this call must be unfolded. The actual specialiser builds the partial SLD-trees simply by following the instructions generated by BTA.

In general, on-line systems tend to achieve better specialisation results, since they consider each call occurring in an SLD-derivation in isolation. This differs from the off-line approach, in which a control decision is associated to a syntactic occurrence of a call in the program, based on an abstraction of the concrete input. Off-line systems, however, also offer a number of advantages over on-line systems. First, the separation of the process in a binding-time analysis followed by a specialisation phase makes the process conceptually easier to reason about, and results in a fairly simple (and efficient!) specialiser from which the burden of continuously monitoring the evaluation process has been removed. Also, the analysis output can be represented by annotations on the original source program, and provide as such excellent feedback to the user providing clues to why an optimisation was (not) performed. In spite of these advantages and the extensive work on off-line partial evaluation in other paradigms, only few efforts have been made to construct an off-line partial evaluation system for logic programming [1].

The basic task of binding-time analysis is to mark as much predicate calls as possible *unfoldable* in the program while guaranteeing that building a partial SLD-tree by following the annotations is a terminating process. Termination of the process is guaranteed if only calls that are guaranteed to terminate are marked *unfoldable*. This is basically the approach taken in [1]: termination analysis (either by hand, or by an automatic system) is used to derive conditions – expressed in terms of availability of a predicate's arguments – under which a call to the particular predicate is guaranteed to terminate. Next, the program is analysed by a classical groundness analysis [6], the results of which (expressing what arguments in a predicate call are available) are combined with the termination conditions to settle the unfolding conditions during analysis. However, appealing as the approach may seem, the use of *run-time* termination of a call as an unfold condition imposes considerable restrictions on the unfolding possibilities. The fact that a call is marked unfoldable only in case it terminates under normal evaluation implies that only calls that can *completely* be unfolded to *true* or *fail* are unfolded. In other words, a specialiser relying on the result of such a binding-time analysis is restricted to building *complete* SLD-trees, rather than *partial* SLD-trees. We illustrate this limitation with the following example.

Example 1. Consider the meta interpreter depicted in the left-hand side of Fig. 1. The interpreter has two predicates implementing the classical `member/2` and `append/3` predicates as object program. Suppose we want to annotate this program with respect to the query `solve([mem(X,Xs)])`.

Meta interpreter

```
solve([]).
solve([A|Gs]):- solve_atom(A), solve(Gs).

solve_atom(A):- cl(A,Body), solve(Body).

cl(mem(X,Xs), [app(_,[X|_],_Xs)]).
cl(app([],L,L), []).
cl(app([X|Xs],Y,[Z|Zs]),[app(Xs,Y,Zs)]).
```

Specialised meta interpreter

```
solve([mem(X,Xs)]):- solve_atom(mem(X,Xs)).

solve_atom(mem(X,Xs)):-
     solve_atom(app(A,[X|_], Xs)).

solve_atom(app([],[X|B],[X|B])).
solve_atom(app([E|Es],[X|B],[Z|Zs])):-
     solve_atom(app(Es, [X|B], Zs)).
```

Fig. 1. Vanilla meta interpreter

Using termination as a criterium for unfolding results in no calls to `solve/1` at all being marked unfoldable, and practically no specialisation would be obtained. Indeed, all calls to `solve/1` arising during unfolding would be of the form `solve([mem(X,Xs)])` or `solve([app(A,[X|_],Xs)])`, none of which is terminating.

2 From Termination Analysis to Binding-time Analysis

In this work, we devise a binding-time analysis that allows more liberal unfoldings than those based on termination conditions *tout-court*. The general idea is to construct step-wise an annotated version of the program and to use termination analysis to prove that building a *partial* SLD-tree for an initial goal with respect to the *annotated program under construction* (as opposed to the original program) terminates. Hence, the process of building a partial SLD-tree according to the annotations terminates.

Example 2. Reconsider the vanilla interpreter from Example 1. Intuitively, from a specialisation point of view, we can see that it is perfectly safe to unfold all calls to the `solve/1` predicate as long as the intermediate calls to `solve_atom/1` are residualised. The idea is that the `solve/1` predicate in a sense only performs the parsing of an object goal (deconstructing a list of atoms), which is terminating in Example 1. Thus, residualising the calls to `solve_atom/1` and unfolding the others results in the specialised program depicted in the right-hand side of Fig. 1 which corresponds, modulo a standard structure filtering transformation [3] with the standard definitions for the `append/3` and `member/2` predicates. Hence, all meta interpretation overhead has been removed.

In order to construct such a binding-time analysis, the key observation is the following: the termination behaviour of building a partial SLD-tree for a query Q with respect to an *annotated* program P_{ann} is equivalent with the termination behaviour of building a complete SLD-tree for a program that is derived from P by *removing* those calls that are annotated to residualise in P_{ann}, keeping only those that are annotated unfoldable in P_{ann}. Our binding-time analysis then proceeds as follows: Assume we have to annotate a program P with respect to an initial query Q. If termination of Q with respect to P can be proven by an automatic termination analysis, all calls in P can safely be annotated unfoldable. As a result, every call occurring during specialisation of Q will be unfolded, and specialisation of Q boils down to plain evaluation of Q in P (constructing a complete SLD-tree for $\langle P, Q\rangle$). If, on the other hand, termination of Q can not be proven, due to the presense of a "suspect" call in P, a new program is derived from P by *deleting* the particular call from P. Once more, termination analysis is performed on the transformed program, and the process is repeated until enough calls are removed from the program such that it terminates. Deleting a call from the program corresponds with marking the particular call to residualise in the annotated program: the call under consideration is not unfolded, and no bindings are created by it.

Such a binding-time analysis can easily be implemented starting from a termination analysis that is capable of identifying "problematic" calls, i.e. those calls due to which the analysis was unable to prove termination of a goal. An example of such a termination analysis is [2]. The analysis constructs a finite approximation of the *binary unfoldings semantics* of the program under consideration. The binary unfoldings semantics of a program consists of a set of binary clauses that relates the head atom with a particular body atom of one of the program's clauses. To prove termination of a query Q with respect to a program P, it suffices to show [2] that a particular condition holds on (some of) the binary clauses constructed by the termination analysis. If the condition does not hold on a particular binary clause, its single body atom identifies the problematic call in the original program.

References

1. Maurice Bruynooghe, Michael Leuschel, and Kostis Sagonas. A polyvariant binding-time analysis for off-line partial deduction. In C. Hankin, editor, *Programming Languages and Systems, Proc. of ESOP'98, part of ETAPS'98*, pages 27–41, Lisbon, Portugal, 1998. Springer-Verlag. LNCS 1381.
2. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
3. J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 229–244, Leuven, April 1990.
4. 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), 1998.
5. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3&4):217–242, 1991.
6. Kim Marriott and Harald Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM letters on Programming Languages and Systems*, 2(1-4):181–196, 1993.