A Polyvariant Binding-Time Analysis for Off-line Partial Deduction

Maurice Bruynooghe, Michael Leuschel, and Konstantinos Sagonas

Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{maurice,michael,kostis}@cs.kuleuven.ac.be

Abstract. We study the notion of binding-time analysis for logic programs. We formalise the unfolding aspect of an on-line partial deduction system as a Prolog program. Using abstract interpretation, we collect information about the run-time behaviour of the program. We use this information to make the control decisions about the unfolding at analysis time and to turn the on-line system into an off-line system. We report on some initial experiments.

1 Introduction

Partial evaluation and partial deduction are well-known techniques for specialising respectively functional and logic programs. While both depart from the same basic concept, there is quite a divergence between their application and overall approach. In functional programming, the most widespread approach is to use *off-line specialisers*. These are typically very simple and fast specialisers which take (almost) no control decisions concerning the degree of specialisation. In this context, the specialisation is performed as follows: First, a *binding-time analysis* (BTA) is performed on the program which annotates all its statements as either “reducible” or “non-reducible”. The annotated program is then passed to the off-line specialiser, which executes the statements marked reducible and produces residual code for the statements marked non-reducible. In logic programming, the *on-line* approach is almost the only one used. All work is done by a complex on-line specialiser which monitors the whole specialisation process and decides on the degree of specialisation while specialising the program. A few researchers have explored off-line specialisation, but lacking an appropriate notion of BTA, they worked with hand-annotated programs, something which is far from being practical. Until now, it was unclear how to perform BTA for logic programs.

The current paper remedies this situation. It develops a BTA for logic programs, not by translating the corresponding notions from functional programming to logic programming, but by departing from first principles. Given a logic program to be specialised, we develop a logic program which performs its on-line specialisation. The behaviour of this program is analysed and the results are used to take all decisions w.r.t. the degree of specialisation off-line. This turns the on-line specialiser into an off-line specialiser. A prototype has been built and the quality and speed of the off-line specialisation has been evaluated.

2 Background

2.1 Partial Deduction

In contrast to ordinary (full) evaluation, a *partial evaluator* receives a program P along with only *part* of its input, called the *static input*. The remaining part of the input, called the *dynamic input*, will only be known at some later point in time. Given the static input S, the partial evaluator then produces a *specialised* version P_S of P which, when given the dynamic input D, produces the same output as the original program P. The goal is to exploit the static input in order to derive a more efficient program.

In the context of logic programming, full input to a program P consists of a goal G and evaluation corresponds to constructing a complete SLDNF-tree for $P \cup \{G\}$. The static input is given in the form of a *partially instantiated* goal G' (and the specialised program should be correct for all instances of G').

A technique which produces specialised programs is known under the name of *partial deduction* [18]. Its general idea is to construct a finite set of atoms $\mathcal{A}$ and a finite set of finite, but possibly *incomplete* SLDNF-trees (one for every[1] atom in $\mathcal{A}$) which "cover" the possibly infinite SLDNF-tree for $P \cup \{G'\}$. The derivation steps in these SLDNF-trees correspond to the computation steps which have been performed beforehand and the specialised program is then extracted from these trees by constructing one specialised clause per non-failing branch.

In partial deduction one usually distinguishes two levels of control: the *global control*, determining the set $\mathcal{A}$, thus deciding *which* atoms are to be partially deduced, and the *local control*, guiding construction of the finite SLDNF-trees for each individual atom in $\mathcal{A}$ and thus determining *what* the definitions for the partially deduced atoms look like.

2.2 Off-line vs. On-line Control

The (global and local) control problems of partial evaluation and deduction in general have been tackled from two different angles: the so-called *on-line* versus *off-line* approaches. The *on-line* approach performs all the control decisions *during* the actual specialisation phase. The *off-line* approach on the other hand performs a (binding-time) analysis phase *prior* to the actual specialisation phase. This analysis starts from a description of which parts of the inputs will be "*static*" (i.e. sufficiently known) and provides *binding-time annotations* which encode the control decisions to be made by the specialiser, so that the specialiser becomes much more simple and efficient.

Partial evaluation of functional programs [8, 15] has mainly stressed off-line approaches, while supercompilation of functional [32, 31] and partial deduction of logic programs [13, 3, 1, 30, 25, 20] have concentrated on on-line control.

On-line methods, usually obtain better specialisation, because no control decisions have to be taken beforehand, i.e. at a point where the full specialisation

[1] Formally, an SLDNF-tree is obtained from an atom or goal by what is called an *unfolding rule*.

information is not yet available. The main reasons for using the off-line approach are to make specialisation itself more efficient and, due to a simpler specialiser algorithm, enable effective self-application (specialisation of the specialiser) [16].

Few authors discuss off-line specialisation in the context of logic programming [27, 17], mainly because so far no automated binding-time analysers have been developed. This paper aims to remedy this problem.

3 Towards BTA for partial deduction

3.1 An on-line specialiser

The basic idea of BTA in functional programming is to model the flow of static input: the arguments of a function call flow to the function body, the result of a function flows back to the call expression. The expressions are annotated *reducible* when enough of their parameters are static, i.e. will be known at specialisation time, to allow the (partial) computation of the expression. Modelling the dataflow gives a system of inequalities over variables in a domain $\{static, dynamic\}$ whose least solution yields the best annotation.

This approach does not immediately translate to logic programs. Problems are that the dataflow in unification is bidirectional and that the degree of instantiation of a variable can change over its lifetime (see also [17]).

We follow a different approach and reconstruct binding-time analysis from first principles. We start with a Prolog program which performs the unfolding decisions of an on-line specialiser. However, whereas real on-line specialisers base their unfolding decisions on the history of the specialisation phase, ours bases its decisions solely on the actual arguments of the call (which can be more easily approximated off-line). This is in agreement with the off-line specialisers for functional languages which base their decision to evaluate or residualise an expression on the availability of the parameters of that expression. The next step will be to analyse the behaviour of this program (the binding-time analysis) and to use the results to make the unfolding decisions at compile time.

First we develop the on-line specialiser. Assuming that for each predicate p/m a *test* predicate $unfold\_p/m$ exists which decides whether to unfold a call or not, we obtain an on-line specialiser by replacing each call $p(\bar{t})$ by

$$(unfold\_p(\bar{t}) \rightarrow p(\bar{t}); memoise\_p(\bar{t}))$$

A call to $memoise\_p(\bar{t})$ informs the specialiser that the call $p(\bar{t})$ has to be residualised. The specialiser has to check whether (a generalisation of) $p(\bar{t})$ has already been specialised —if not it has to initiate the specialisation of (a generalisation of) $p(\bar{t})$— and has to perform appropriate renaming of predicates to ensure that residual code calls the proper specialised version of the predicate it calls.

Example 1 (Funny append). Consider the following on-line specialiser for a variant, *funnyapp/3* of the *append/3* predicate in which the first two arguments of the recursive call have been swapped:

```
funnyapp([],X,X).
funnyapp([X|U],V,[X|W]) :-
    ( unfold_funnyapp(V,U,W) -> funnyapp(V,U,W)
                              ; memoise_funnyapp(V,U,W) ).
unfold_funnyapp(X,Y,Z) :- ground(X).
```

Specialising this program for a query `funnyapp([a,b],L,R)` results in the specialised clause (the residual call is renamed as `funnyapp_1`)

```
funnyapp([a,b],L,[a|R1]) :- funnyapp_1(L,[b],R1).
```

Specialising the `funnyapp` program for the residual call `funnyapp(L,[b],R1)` gives (after renaming) the clauses

```
funnyapp_1([],[b],[b]).
funnyapp_1([X|U],[b],[X,b|R]) :- funnyapp_2(U,[],R).
```

Once more specialising, now for the residual call `funnyapp(U,[],R)`, gives

```
funnyapp_2([],[],[]).
funnyapp_2([X|U],[],[X|U]).
```

This completes the specialisation. Note that the sequence of residual calls is terminating in this example. In general, infinite sequences are possible. They can be avoided by generalising some arguments of the residual calls before specialising.

In the above example, instead of using `ground(X)` as condition of unfolding, one could also use the test `nilterminated(X)`.This would allow to obtain the same level of specialisation for a query `funnyappend([X,Y],L,R)`. This test is another example of a so called *rigid* or *downward closed* property: if it holds for a certain term, it holds also for all its instances. Such properties are well suited for analysis by means of abstract interpretation.

3.2 From on-line to off-line

Turning the on-line specialiser into an off-line one requires to determine the *unfold_p/n* predicates during a preceding analysis and to decide on whether to replace the $(\mathit{unfold\_p}(\bar{t}) \rightarrow p(\bar{t}); \mathit{memoise}(p(\bar{t})))$ construct either by $p(\bar{t})$ or by $\mathit{memoise}(p(\bar{t}))$. The decision has to be based on a safe estimate of the calls $\mathit{unfold\_p}(\bar{t})$ which will occur during the specialisation. Computing such safe approximations is exactly the purpose of *abstract interpretation* [9].

```
:- {grnd(L1)} fap1(L1,L2,R) {grnd(L1)}.
fap1([],X,X).
fap1([X|U],V,[X|W]) :- {grnd(X,U)}
    ( unf_fap1(V,U,W) {grnd(X,U,V)} ->
          {grnd(X,U,V)} fap2(V,U,W) {grnd(X,U,V,W)}
        ; {grnd(X,U)} memo_fap3(V,U,W) {grnd(X,U)}
    ) {grnd(X,U)}.
unf_fap1(X,Y,Z) :- {grnd(Y)} ground(X) {grnd(X,Y)}.
memo_fap3(X,Y,Z).
```

By adding to the code of Example 1 the fact `memoise_funnyapp(X,Y,Z).`, and an appropriate handling of the Prolog built-in `ground/1`, one can run a goal-dependent *polyvariant* groundness analysis (using e.g. PLAI coupled with the set-sharing domain) for a query where the first argument is ground and obtain the above annotated program. The annotated code for the version `fap2` is omitted because it is irrelevant for us. Indeed, inspecting the annotations for `unf_fap1` we see that the analysis cannot infer the groundness of its first argument. So we decide off-line not to unfold, we cancel the test and the *then* branch and simplify the code into:

```
:- {grnd(L1)} fap1(L1,L2,R) {grnd(L1)}.
fap1([],X,X).
fap1([X|U],V,[X|W]) :- {grnd(X,U)} memo_fap3(V,U,W) {grnd(X,U)}.
```

The residual call to `funnyappend` has a different call pattern than the original call: its second argument is now ground. Thus we perform a second analysis and obtain (the annotated code for `fap4` is omitted):

```
:- {grnd(L2)} fap3(L1,L2,R) {grnd(L2)}.
fap3([],X,X).
fap3([X|U],V,[X|W]) :- {grnd(V)}
    ( unf_fap2(V,U,W) {grnd(V)} ->
            {grnd(V)} fap4(V,U,W) {grnd(V)}
          ; {grnd(V)} memo_fap5(V,U,W) {grnd(V)}
    ) {grnd(V)}.
unf_fap2(X,Y,Z) :- {grnd(X)} ground(X) {grnd(X)}.
memo_fap5(X,Y,Z).
```

This time, the annotations for `unf_fap2` show that the groundness test will definitely succeed. So we decide off-line always to unfold and only keep the *then* branch. Moreover, the `fap4` call has the same call pattern as the original call to `funnyapp`, so we also rename it as `fap1`. This yields the second code fragment:

```
:- {grnd(L2)} fap3(L1,L2,R) {grnd(L2)}.
fap3([],X,X).
fap3([X|U],V,[X|W]) :- {grnd(V)} fap1(V,U,W) {grnd(V)}
```

Applying the specialiser on these two code fragments for a query `fap1([a,b],L,R)` gives the same specialised code as in Example 1. However, this time, no calls to `unfold_funnyapp` have to be evaluated during specialisation.

3.3 Automation

To weave the step by step analysis sketched above in a single analysis, a special purpose tool has to be built. We implemented a system based on the abstract domain POS, also called PROP [24]. It describes the state of the program variables by means of *positive* boolean formulas, i.e., formulas built from ↔, $\wedge$ and $\vee$. Its most popular use is for groundness analysis. In that case, the formula X expresses that the program variable X is (definitely) bound to a ground term, $X \leftrightarrow Y$ expresses that X is bound to a ground term iff Y is, so an eventual binding of X to a ground term will be propagated to Y. This domain is extended with *false* as bottom element and is ordered by boolean implication. Groundness analysis corresponds to checking the rigidity[2] of program variables w.r.t. the *termsize norm*[3] and abstracts a unification such as $X = [Y|Z]$ by the boolean formula $X \leftrightarrow Y \wedge Z$. However POS can also be used with other semi-linear norms [4]. In e.g. normalised programs, it only requires to redefine the abstraction of the unifications. For example, with the *listlength norm*[4], unification of $X = [Y|Z]$ is abstracted as $X \leftrightarrow Z$, and a formula X means that the program variable X is bound to a term with a bounded listlength, i.e. either the term is a nil-terminated list, or has a main functor which is not a list constructor.

The analyser has to decide the outcome of the *unfold_p* test and has to decide which branch to take for further analysis while doing the analysis. Also it has to launch the analysis of the generalisations of the memoised calls. The generalisation we employ is to replace an argument which is not rigid under the norm used in the analysis by the abstraction of a fresh variable. These requirements exclude the direct use of the abstract compilation technique in the way advocated by e.g. [5]. One problem of the scheme of [5] is that it handles constructs $(\mathit{ground}(X) \rightarrow p(\bar{t}); \mathit{memoise}(p(\bar{t})))$ too inaccurately. The boolean formula is represented as a truth table, i.e. a set of tuples, and the analyser processes the truth table a tuple at a time. Therefore it cannot infer in a program point that X is true, i.e. that X is definitely ground, so it can never conclude that the *else* branch cannot be taken. The other problem is that the analyses launched for the *memoised* calls should not interfere (i.e. output should not flow back) with the analysis of the clauses containing the memoised calls. Note that defining *memoise* as $\mathit{memoise\_p}(X_1, \ldots, X_n) \text{ :- } \mathit{copy}(X_1, Y_1), \ldots, \mathit{copy}(X_n, Y_n)$, and abstracting $\mathit{copy}(X, Y)$ as $X \leftrightarrow Y$ does not work: The abstract success state of executing $p(Y_1, \ldots, Y_n)$ will update the abstractions of $X_1, \ldots, X_n$.

Our prototype binding-time analyser currently consists of ≈ 800 lines of Prolog code and uses XSB [29] as a generic tool for semantic-based program analysis [6]. The boolean formulas from the POS domain are represented by their truth tables. This representation enables abstract operations to have straightforward implementations based on the projection and equi-join operations of the relational algebra. The disadvantage is that the size of truth tables quickly increases with the number of variables in a clause. The use of dedicated data structures like BDDs to represent the boolean formulas as in [33] often results in better performance but at the expense of substantial programming efforts.

The main part of the analyser can be seen as a source-to-source transformation (i.e. abstract compilation) that given the program P to be analysed,

[2] A term is rigid w.r.t. a norm if all its instances have the same size w.r.t. the norm.

[3] The termsize norm [11] includes all subterms in the *measure* of the term.

[4] The listlength norm [11] includes only the tail of the list in the measure of the list (and measures other terms as 0); nil-terminated lists are rigid under this norm.

produces an abstract program P^α with suitable annotations. The abstract program can be directly run under XSB (using tabling to ensure termination). The execution leaves the results of the analysis in the XSB *tables*. Each predicate p/n of P is abstracted to a predicate $p^\alpha/2$ whose arguments carry input and output *sets* of tuples. The core part of setting up the analysis is then to define the code for the abstract interpretation of each call (at a program point $\mathtt{PP}_\#$ of interest):

$$(\mathit{unfold\_p}(\overline{X}) \rightarrow p(\overline{X}); \mathit{memoise\_p}(\overline{X})). \qquad (1)$$

is abstracted by the following code fragment:

```
$\mathit{project}$(Args,TPP$_{in}$,TC),
( $\mathit{unfold\_p}$(TC) ->
        $\mathit{unfold}$(TC,PP$_\#$), $p^\alpha$(TC,TR)
      ; TR=TC, $\mathit{generalise}$(TC,TCG), $\mathit{memo}$(TCG,PP$_\#$), $p^\alpha$(TCG,_) ),
$\mathit{equi\_join}$(Args,TPP$_{in}$,TR,TPP$_{out}$),
```

Predicates $unfold/2$ and *memo*/2 which abstract the behaviour of each call in the form of (1) above are *tabled* predicates which have no effect on the computation, but only record information containing the results of the analysis. Their arguments are the current abstraction and the current program point. This information is then dumped from the XSB tables and is fed to the off-line specialiser. The variable $\mathtt{TPP}_{in}$ holds the truth table which represents the abstraction of the program state in the point prior to the call. The call to *project*/3 projects the truth table on the positions `Args` of the variables $\overline{X}$ participating in the call. The result is `TC` (Tuples of the Call). The predicate $unfold\_p/1$ (currently supplied by the user for each predicate p/n to be analysed) inspects `TC` to decide whether there is sufficient information to unfold the call. If it succeeds the *then* branch is taken which analyses the effects of unfolding p/n. This is done by executing $p^\alpha/2$ with `TC` as abstraction of the call state. The analysis returns `TR` as abstraction of the program state reached after unfolding p/n. If the call to $unfold\_p/1$ fails, the call is memoised, and the program state remains unchanged, so `TR = TC`. The generalisation of the memoised call also needs to be analysed; therefore the *else* branch first generalises the current state `TC` into `TCG` by erasing all dependencies for non-rigid arguments[5] and then calls $p^\alpha/2$ with `TCG` as initial state, but takes care not to use the returned abstract state as the bindings resulting from specialising memoised calls do not flow back. These actions effectively realise the intended functionality of *memoise_p*/1. Finally, the new program state `TR` over the variables $\overline{X}$ has to be propagated to the other program variables described by the initial state $\mathtt{TPP}_{in}$. This is achieved simply by taking the equi-join over the `Args` of $\mathtt{TPP}_{in}$ and `TR`. The new program state is described by $\mathtt{TPP}_{out}$.

One of our examples (see Section 4.2) uses two different norms in the *unfold* tests: the term norm which tests for groundness, and the listlength norm which tests for the boundedness of lists (whether lists are nil-terminated). This does not pose a problem for our framework, we simply use a truth table which encodes two boolean formulas, one for the term norm and one for the listlength norm.

[5] A position is rigid if it has an "s" in each tuple e.g. *generalise*(`[p(s,s,s), p(s,d,d)],TCG`) yields `TCG = [p(s,s,s), p(s,s,d), p(s,d,s), p(s,d,d)]`.

4 Some Experiments and Benchmarks

We first discuss the parser and liftsolve examples from [17].

4.1 The parser example

A small generic parser for languages defined by grammars of the form $S ::= aS|X$ (X is a placeholder for a terminal symbol as well as the first argument to nont/3; arguments 2 and 3 represent the string to be parsed as a difference list):

```
nont(X,T,R) :- t(a,T,V),nont(X,V,R).
nont(X,T,R) :- t(X,T,R).
t(X,[X|Es],Es).
```

A termination analysis can easily determine that calls to t/3 always terminate and that calls to nont/3 terminate if their second argument is ground. One can therefore derive the following unfold predicates:

```
unfold_t(X,S1,S2).
unfold_nont(X,T,R) :- ground(T).
```

Performing our analysis for the entry point :- $\{grnd(X)\}$ nont(X,_,_) we obtain the following annotated program (dynamic arguments [i.e. non-ground ones] and non-reducible predicates [i.e. memoised ones] are underlined):

```
nont(X,T,R) :- t(a,T,V), nont(X,V,R).
nont(X,T,R) :- t(X,T,R).
t(X,[X|Es],Es).
```

Feeding this information into the off-line system LOGEN [17] and specialising nont(c,T,R), we obtain:

```
nont__0([a|B],C) :- nont__0(B,C).
nont__0([c|D],D).
```

Analysing the same specialiser for :- $\{grnd(T)\}$ nont(_,T,_) yields:

```
nont(X,T,R) :- t(a,T,V), nont(X,V,R).
nont(X,T,R) :- t(X,T,R).
t(X,[X|Es],Es).
```

Feeding this information into LOGEN and specialising nont(X,[a,a,c],R) yields:

```
nont__0(c,[]).
nont__0(a,[c]).
nont__0(a,[a,c]).
```

4.2 The liftsolve example

The following program is a meta-interpreter for the ground representation, in which the goals are "lifted" to the non-ground representation for resolution. To perform the lifting, an accumulating parameter is used to keep track of the variables that have already been encountered and generated. The predicate mng and l_mng transform (a list of) ground terms (the first argument) into (a list of) non-ground terms (the second argument; the third and fourth arguments represent the incoming and outgoing accumulator respectively). The predicate `solve` uses these predicates to "lift" clauses of a program in ground representation (its first argument) and then use them for resolution with a non-ground goal (its second argument) to be solved.

```
solve(GrP,[]).
solve(GrP,[NgH|NgT]) :-
        non_ground_member(term(clause,[NgH|NgBdy]),GrP),
        solve(GrP,NgBdy), solve(GrP,NgT).
non_ground_member(NgX,[GrH|_GrT]) :- make_non_ground(GrH,NgX).
non_ground_member(NgX,[_GrH|GrT]) :- non_ground_member(NgX,GrT).
make_non_ground(G,NG) :- mng(G,NG,[],_Sub).
mng(var(N),X,[],[sub(N,X)]).
mng(var(N),X,[sub(N,X)|T],[sub(N,X)|T]).
mng(var(N),X,[sub(M,Y)|T],[sub(M,Y)|T1]) :- N \== M, mng(var(N),X,T,T1).
mng(term(F,Args),term(F,IArgs),InS,OutS) :- lmng(Args,IArgs,InS,OutS).
lmng([],[],Sub,Sub).
lmng([H|T],[IH|IT],InS,OutS) :-
        mng(H,IH,InS,InS1), lmng(T,IT,InS1,OutS).
```

The following unfold predicates can be derived by a termination analysis:

```
    unfold_lmng(Gs,NGs,InSub,OutSub) :- ground(Gs), bounded_list(InSub).
    unfold_mng(G,NG,InSub,OutSub) :- ground(G), bounded_list(InSub).
    unfold_make_non_ground(G,NG) :- ground(G).
    unfold_non_ground_member(NgX,L) :- ground(L).
    unfold_solve(GrP,Query) :- fail.
```

Analysing the specialiser for the entry point `solve(ground,_)` we obtain:

```
    solve(GrP,[]).
    solve(GrP,[NgH|NgT]) :-
            non_ground_member(term(clause,[NgH|NgBdy]), GrP),
            solve(GrP,NgBdy), solve(GrP,NgT).
    non_ground_member(NgX,[GrH|_GrT]) :- make_non_ground(GrH,NgX).
    non_ground_member(NgX,[_GrH|GrT]) :- non_ground_member(NgX,GrT).
    make_non_ground(G,NG) :- mng(G,NG, [],_Sub).
    mng(var(N),X,[],[sub(N,X)]).
    mng(var(N),X,[sub(N,X)|T], [sub(N,X)|T]).
    mng(var(N),X,[sub(M,Y)|T], [sub(M,Y)|T1]) :- N \== M, mng(var(N),X,T,T1).
    mng(term(F,Args),term(F,IArgs), InS,OutS) :- lmng(Args,IArgs, InS,OutS).
    lmng([],[],Sub,Sub).
    lmng([H|T],[IH|IT], InS,OutS) :- mng(H,IH,InS,InS1), lmng1(T,IT, InS1,OutS).
    lmng1([],[],Sub,Sub).
    lmng1([H|T],[IH|IT], InS,OutS) :- mng1(H,IH,InS,InS1), lmng1(T,IT, InS1,OutS).
    mng1(var(N),X,[],[sub(N,X)]).
    mng1(var(N),X, [sub(N,X)|T],[sub(N,X)|T]).
    mng1(var(N),X, [sub(M,Y)|T],[sub(M,Y)|T1]) :- N \== M, mng1(var(N),X,T,T1).
    mng1(term(F,Args),term(F,IArgs), InS,OutS) :- lmng1(Args,IArgs, InS,OutS).
```

One can observe that the call `lmng1(T,IT, InS1,OutS)` has not been unfolded. Indeed, the third argument `InS1` is considered to be dynamic (non-ground) and the call to `unfold_lmng` will thus not always succeed. However, based on the termination analysis, it is actually sufficient for termination if the third arguments to mng and lmng are bounded lists (as the listlength norm can be used in the termination proof). If we use our prototype to also keep track of bounded lists we obtain the desired result: the call lmng1(T,IT,InS1,OutS) can be unfolded as the first argument is ground and third argument can be inferred to be a bounded list. By feeding the so obtained annotations into LOGEN [17] we obtain a specialiser which removes (most of) the meta-interpretation overhead. E.g. specialising

```
solve([term(clause,[term(q,[var(1)]), term(p,[var(1)])]),
     term(clause,[term(p,[term(a,[])])])],G)
```

yields the following residual program:

```
solve__0([]).
solve__0([term(q,[B])|C]) :- solve__0([term(p,[B])]),solve__0(C).
solve__0([term(p,[term(a,[])])|D]) :- solve__0([]),solve__0(D).
```

4.3 Some Benchmarks

We now study the efficiency and quality of our approach on a set of benchmarks. Except for the parser benchmark all benchmarks come from the DPPD benchmark library [19]. We ran our prototype analyser, BTA, that performs binding-time analysis and fed the result into the off-line compiler generator LOGEN [17] in order to derive a specialiser for the task at hand. The ECCE on-line partial deduction system [19] has been used for comparison (settings are the same as for ECCE-X in [20], i.e. a mixtus like unfolding, a global control based upon characteristic trees but no use of conjunctive partial deduction). The interested reader can consult [20] to see how ECCE compares with other systems.

All experiments were conducted on a Sun Ultra-1 running SunOS 5.5.1. ECCE and LOGEN were run using Prolog by BIM 4.1.0. BTA was run on XSB 1.7.2.

Benchmark	ECCE - PD	BTA	LOGEN	PD	Ratio
depth.lam	0.34 s	0.05 + 0.579 s	0.05 s	0.003 s	113
liftsolve.app	1.00 s	0.079* + 1.841 s	0.05 s	0.006 s	167
liftsolve.app4	12.32 s	″	″	0.014 s	880
match.kmp	0.18 s	0.06 + 0.031 s	0.01 s	0.006 s	30
parser	0.06 s	0.03 + 0.01 s	0.02 s	0.001 s	60
regexp.r1	0.17 s	0.039 + 0.031 s	0.06 s	0.006 s	28

Table 1. Analysis and Specialisation Times

In Table 1 one can see a summary of the transformation times. The columns under BTA contain: the time to abstract and compile the program + the time for execution of the abstracted program (both under XSB). The column under LOGEN contains the time to generate the specialiser with LOGEN using the so obtained annotations. Observe, that for any given initial annotation, this has only to be performed *once*: the so obtained specialiser can then be used over and over again for different specialisation tasks. E.g. the same specialiser was used for the liftsolve.app and liftsolve.app4 benchmark. The '*' for liftsolve.app indicates the time for the abstract compilation only producing code for the groundness analysis. The extra arguments and instructions for the bounded list analysis were added by hand (but will be generated automatically in the next version of the prototype). The column under PD gives the time for the off-line specialisation. The last column of the table contains the ratio of running ECCE over running the specialisers generated by BTA + LOGEN. As can be seen, the specialisers produced by BTA + LOGEN run 28 – 880 times faster than ECCE. We conjecture that for larger programs (e.g liftsolve with a very big object program) this difference can get even bigger. Also, for 3 benchmarks the combined time of running BTA + LOGEN and then the so obtained specialiser was less than running ECCE, i.e. our off-line approach fares well even in "one-shot" situations. Of course, to arrive at a fully automatic (terminating) system one will still have to add the time for the termination analysis, needed to derive the "unfold" predicates.

Benchmark	Original	ECCE	BTA + LOGEN
`depth.lam`	0.08 s	0.00 s	0.06 s
	1	≈ 32	1.33
`liftsolve.app`	0.13 s	0.01 s	0.01 s
	1	13	13
`liftsolve.app4`	0.17 s	0.00 s	0.02 s
	1	> 34	8.5
`match.kmp`	0.58 s	0.34 s	0.51 s
	1	1.71	1.14
`parser`	0.20 s	0.12 s	0.12 s
	1	1.74	1.74
`regexp.r1`	0.29 s	0.10 s	0.20 s
	1	2.9	1.5

Table 2. Absolute Runtimes and Speedups

Table 2 compares the efficiency of the specialised programs (for the run time queries see [19]; for the `parser` example we ran $nont(c, [a^{17}, c, b], [b])$ 100 times). As was to be expected, the programs generated by the on-line specialiser ECCE outperform those generated by our off-line system. E.g. for the `match.kmp` benchmark ECCE is able to derive a Knuth-Morris-Pratt style searcher, while off-line systems (so far) are unable to achieve such a feat. However, one can see that the specialised programs generated by BTA + LOGEN are still very satisfactory. The most satisfactory application is `liftsolve.app` (as well as `liftsolve.app4`), where the specialiser generated by BTA + LOGEN runs 167 (resp. 880) times faster than ECCE while producing residual code of equal (resp. almost equal) efficiency. In fact, the specialiser compiled the append object program from the ground representation into the non-ground one in just 0.006 s (to be compared with e.g. the compilers generated by SAGE [14] which run in the order of minutes). Furthermore, the time to produce the residual program and then running it is less than the time needed to run the original program for the given set of runtime queries. This nicely illustrates the potential of our approach for applications such as runtime code generation, where the specialisation time is (also) of prime importance.

5 Discussion

We have formulated a binding-time analysis for logic programs, and have reported on a prototype implementation and on an evaluation of its effectiveness. To develop the binding-time analysis, we have followed an original approach: Given a program P to be analysed we transform it into an on-line specialiser program P', in which the unfolding decision are explicitly coded as calls to predicates `unfold_p`. The on-line specialiser is different from usual ones in the sense that it — like off-line specialisers — uses the availability of arguments to decide on the unfolding of calls. Next, we apply abstract interpretation —a binding-time analysis— to gather information about the run-time behaviour of P'. The information in the program points related to `unfold_p` allows to decide whether the test will definitely succeed —in which case the unfolding branch is retained— or will possibly fail —in which case the branch yielding residual code is retained. The resulting program now behaves as an off-line specialiser as all unfolding decisions have been taken at analysis time.

An issue to be discussed in more detail is the termination of the specialisation. First, a specialiser has a global control component. It must ensure that only a finite number of atoms are specialised. In our prototype, we generalise the residual calls before generating a specialised version: arguments which are not rigid[6] w.r.t. the norm used in the unfolding condition are replaced by fresh variables. This works well in practice but is not a sufficient condition for termination. In principle one could define the `memoise_p` predicates as:

`memoise_p(`$\overline{X}$`) :- copy_term(`$\overline{X}$`,`$\overline{Y}$`), generalise(`$\overline{Y}$`,`$\overline{Z}$`), p(`$\overline{Z}$`).`

and then generalise such that quasi-termination [21] of the program, where calls to `p` are tabulated, can be proven. In practice, the built-in `copy_term/2` and the built-ins needed to implement `generalise/2` will make this a non-trivial task. Secondly, there is the local control component. It must ensure that the unfolding of a particular atom terminates. This is decided by the code of the transformed program. Defining the `unfold_p` predicates by hand is error-prone and consequently not entirely reliable. In principle, one could replace the calls `memoise_p` by `true` and apply off-the-shelf tools for proving termination of logic programs [22, 7]. Whether these will do well depends on how well they handle the $if - then - else$ construct used in deciding on the unfolding and the built-ins used in the rigidity test (e.g. the analysis has to infer that `X` is bounded and rigid w.r.t. the norm in the program point following a test `ground(X)`). It is likely that small extensions to these tools will suffice to apply them successfully in proving termination of the unfolding[7], at least when the unfolding conditions are based on rigidity tests with respect to the norms used by those termination analysis tools.

A more interesting approach for the local control problem is to automatically generate unfolding conditions by program analysis. Actually, one could apply a

[6] I.e., "*static*" from functional programming becomes "*rigid* w.r.t. a given norm."

[7] After a small extension by its author, the system of [22] could handle small examples. However, so far we have not done exhaustive testing.

more general scheme for handling the unfolding than the one used so far. Having for each predicate p/n the original clauses with head p/n and transformed clauses with head pt/n, the transformed clauses could be derived from the original by replacing each call q/m by:

```
( terminates_q(t̄) -> q(t̄)
      ; ( unfold_q(t̄) -> qt(t̄) ; memoise_q(t̄) ) )
```

In [10], Decorte and De Schreye describe how the constraint-based termination analysis of [11] can be adapted to generate a finite set of "most general" termination conditions (e.g. for append/3 they would generate rigidity w.r.t. the listlength norm of the first argument and rigidity w.r.t. the listlength norm of the third argument as the two most general termination conditions; for our funnyapp/3 they would generate rigidity of the first and second argument w.r.t. the listlength norm as the most general termination condition.). These conditions can be used to define the terminates_q predicates. If they succeed, the call q($\bar{t}$) can be executed with the original code and is guaranteed to terminate. Moreover, as they are based on rigidity, they are very well suited to be approximated by our binding-time analysis. Actually, in all our benchmarks programs, we were using termination conditions for controlling the unfolding, so in fact we could have further improved the speed of the specialiser by not checking the condition on each iteration but using the above scheme.

Generating unfold_q definitions is a harder problem. It is related to the generation of "safe" (i.e. termination ensuring) delay declarations in languages such as MU-Prolog and Gödel. This is a subtle problem as discussed in [28, 23]. For example, the condition (nonvar(X); nonvar(Z)) is not safe for a call append(X,Y,Z); execution, and in our case unfolding, could go on infinitely for some non-linear calls (e.g. append([a|L],Y,L)). Also the condition nonvar/1 is not rigid. (For funnyapp/3 we had rigid conditions, however this is rather the exception than the rule.) A safe unfolding condition for append(X,Y,Z) is linear(append(X,Y,Z)), (nonvar(X); nonvar(Z)). Linearity is well suited for analysis (e.g. [2]), but a test nonvar(X) is not. Moreover, unless X is ground, the test is typically not invariant over the different iterations of a recursive predicate. A solution could be to switch to a hybrid specialiser: deciding the linearity test at analysis-time and the simple nonvar tests at run-time. But as said above, perhaps due to lack of a good application (for languages with delay, speed is more important than safety), there seems to be no work on generating such conditions.

Another hybrid approach is taken in a recent work independent of ours [26]. This work also starts from the termination condition. When it is violated, the size of the term w.r.t. the norm used in the termination condition and the maximal reduction of the size in a single iteration is used to compute the number of unfolding steps. The program is transformed and calls to be unfolded are given an extra argument initialised with the allowed number of unfolding steps. An on-line test checks the value of the counter and the call is residualised when the counter reaches zero.

Acknowledgements

M. Bruynooghe and M. Leuschel are supported by the Fund for Scientific Research - Flanders Belgium (FWO). K. Sagonas is supported by the Research Council of the K.U. Leuven. Some of the present ideas originated from discussions and joint work with Jesper Jørgensen, and from the PhD. work of Dirk Dussart [12], to both of whom we are very grateful. We thank Bart Demoen, Stefaan Decorte, Bern Martens, Danny De Schreye and Sandro Etalle for interesting discussions, ideas and comments.

References

1. R. Bol. Loop Checking in Partial Deduction. *The Journal of Logic Programming*, 16(1&2):25–46, May 1993.
2. M. Bruynooghe, M. Codish, and A. Mulkers. Abstracting unification: a key step in the design of logic program analyses. In *Computer Science Today*, pages 406–442. Springer-Verlag, LNCS Vol. 1000, 1995.
3. M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding During Partial Deduction. *New Generation Computing*, 11(1):47–79, 1992.
4. M. Codish and B. Demoen. Deriving Polymorphic Type Dependencies for Logic Programs Using Multiple Incarnations of Prop. In B. Le Charlier, editor, *Proceedings of the First International Symposium on Static Analysis*, number 864 in LNCS, pages 281–297, Namur, Belgium, September 1994. Springer-Verlag.
5. M. Codish and B. Demoen. Analysing Logic Programs using "Prop"-ositional Logic Programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, December 1995.
6. M. Codish, B. Demoen, and K. Sagonas. Semantic-Based Program Analysis for Logic-Based Languages using XSB. K.U. Leuven TR CW 245. December 1996.
7. M. Codish and C. Taboch. A Semantic Basis for Termination Analysis of Logic Programs and its Realization using Symbolic Norm Constraints. In *Proceedings of the Sixth International Conference on Algebraic and Logic Programming*, number 1298 in LNCS, pages 31–45. Springer-Verlag, September 1997.
8. C. Consel and O. Danvy. Tutorial Notes on Partial Evaluation. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
9. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977. ACM.
10. S. Decorte and D. De Schreye. Termination Analysis: Some Practical Properties of the Norm and Level Mapping Space. TR, Dept. Comp. Science, K.U. Leuven.
11. S. Decorte and D. De Schreye. Demand-driven and Constraint-based Automatic Termination Analysis for Logic Programs. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 78–92, Leuven, Belgium, July 1997. The MIT Press.
12. D. Dussart. *Topics in Program Specialisation and Analysis for Statically Typed Functional Languages.* PhD thesis, Katholieke Universiteit Leuven, May 1997.
13. J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialisation. *New Generation Computing*, 9(3,4):305–333, 1991.
14. C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, Department of Computer Science, University of Bristol.

15. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science, 1993.
16. N. D. Jones, P. Sestoft, and H. Søndergaard. MIX: a Self-applicable Partial Evaluator for experiments in Compiler Generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
17. J. Jørgensen and M. Leuschel. Efficiently Generating Efficient Generating Extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, number 1110 in LNCS, pages 238–262, Schloß Dagstuhl, February 1996. Springer-Verlag.
18. J. Komorowski. Partial Evaluation as a means for inferencing data structures in an Applicative Language: A Theory and Implementation in the case of Prolog. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 255–267, Albuquerque, New Mexico, January 1982. ACM.
19. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~lpai`, 1996.
20. M. Leuschel, B. Martens, and D. De Schreye. Controlling Generalisation and Polyvariance in Partial Deduction of Normal Logic Programs. *ACM Trans. Prog. Lang. Syst.*, 20, 1998. To Appear.
21. M. Leuschel, B. Martens, and K. Sagonas. Preserving Termination of Tabled Logic Programs While Unfolding. In N. Fuchs, editor, *Proceedings of LOPSTR'97: Logic Program Synthesis and Transformation*, LNCS, Leuven, Belgium, July 1997.
22. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 63–77, Leuven, Belgium, July 1997. The MIT Press.
23. E. Marchiori and F. Teusink. Proving Termination of Logic Programs with Delay Declarations. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium*, pages 447–461, Portland, Oregon, December 1995.
24. K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Progr. Lang. and Syst.*, 2(1–4):181–196, 1993.
25. B. Martens and D. De Schreye. Automatic Finite Unfolding Using Well-Founded Measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996.
26. J. Martin. Sonic Partial Deduction. Technical Report, Dept. Elec. and Comp. Sc., University of Southampton, January 1998.
27. 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.
28. L. Naish. Coroutining and the Construction of Terminating Logic Programs. Technical Report TR 92/5, Dept. Computer Science, University of Melbourne, 1992.
29. K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.
30. D. Sahlin. Mixtus: An Automatic Partial Evaluator for Full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
31. M. H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium*, pages 465–479, Portland, Oregon, December 1995.
32. V. F. Turchin. The Concept of a Supercompiler. *ACM Trans. Prog. Lang. Syst.*, 8(3):292–325, July 1986.
33. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the Domain Prop. *Journal of Logic Programming*, 23(3):237–278, June 1995.