Logic Program Specialisation: How To Be More Specific

Michael Leuschel* Danny De Schreye†

Departement Computerwetenschappen
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail : {michael,dannyd}@cs.kuleuven.ac.be
Tel.: ++32 (0)16 - 32 7555 Fax: ++32 (0)16 - 32 7996

Abstract

Standard partial deduction suffers from several drawbacks when compared to top-down abstract interpretation schemes. Conjunctive partial deduction, an extension of standard partial deduction, remedies one of those, namely the lack of side-ways information passing. But two other problems remain: the lack of success-propagation as well as the lack of inference of global success-information. We illustrate these drawbacks and show how they can be remedied by combining conjunctive partial deduction with an abstract interpretation technique known as more specific program construction. We present a simple, as well as a more refined integration of these methods. Finally we illustrate the practical relevance of this approach for some advanced applications, like proving functionality or specialising certain meta-programs written in the ground representation, where it surpasses the precision of current abstract interpretation techniques.

1 Introduction

The heart of any technique for *partial deduction*, or more generally *logic program specialisation*, is a program analysis phase. Given a program P and an (atomic) goal $\leftarrow A$, one aims to analyse the computation-flow of P for all instances $\leftarrow A\theta$ of $\leftarrow A$. Based on the results of this analysis, new program clauses are synthesised.

In partial deduction, such an analysis is based on the construction of finite and usually incomplete[1], SLD(NF)-trees. More specifically, following the foundations for partial deduction developed in [21], one constructs

- a finite set of atoms $S = \{A_1, \ldots, A_n\}$, and
- a finite (possibly incomplete) SLD(NF)-tree τ_i for each $(P \cup \{\leftarrow A_i\})$,

such that:

1) the atom A in the initial goal $\leftarrow A$ is an instance of some A_i in S, and
2) for each goal $\leftarrow B_1, \ldots, B_k$ labelling a leaf of some SLD(NF)-tree τ_l, each B_i is an instance of some A_j in S.

*Supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation"

†Senior Research Associate of the Belgian Fund for Scientific Research

[1] As usual in partial deduction, we assume that the notion of an SLD-tree is generalised [21] to allow it to be incomplete: at any point we may decide not to select any atom and terminate a derivation.

The conditions 1) and 2) ensure that *together* the SLD(NF)-trees $\tau_1, \ldots, \tau_n$ form a *complete description* of all possible computations that can occur for all concrete instances $\leftarrow A\theta$ of the goal of interest. At the same time, the point is to propagate the available input data in $\leftarrow A$ as much as possible through these trees, in order to obtain sufficient accuracy. The outcome of the analysis is precisely the set of SLD(NF)-trees $\{\tau_1, \ldots, \tau_n\}$: a complete, and as precise as possible, description of the computation-flow.

Finally, a code generation phase produces a *resultant clause* for each non-failing branch of each tree, which synthesises the computation in that branch.

In the remainder of this paper we will restrict our attention to *definite* logic programs (possibly with some declarative built-ins like `\=`, `is`, ...). In that context, the following generic scheme (based on a similar one presented in [6, 5]) describes the basic layout of practically all proposed algorithms for computing the sets S and $\{\tau_1, \ldots, \tau_n\}$ (see also e.g. [13, 19]).

Algorithm 1.1 (Standard Partial Deduction)
Initialise $i = 0$, $S_i = \{A\}$
repeat
  for *each* $A_k \in S_i$, *compute a finite SLD-tree* τ_k *for* A_k ;
  let $S'_i := S_i \cup \{B_l | B_l$ *is an atom in a leaf of some tree* τ_k,
    which is not an instance of any $A_r \in S_i\}$;
  let $S_{i+1} :=$ abstract(S'_i)
until $S_{i+1} = S_i$

In this algorithm, *abstract* is a *widening operator*: $abstract(S'_i)$ is a set of atoms such that each atom of S'_i is an instance of atom in $abstract(S'_i)$. The purpose of the operator is to ensure termination of the analysis.

An analysis following this scheme focusses exclusively on a top-down propagation of call-information. In the separate SLD-trees τ_i, this propagation is performed through repeated unfolding steps. The propagation over different trees is achieved by the fact that for each atom in a leaf of a tree there exists another tree with (a generalisation of) this atom as its root. The decision to create a *set* of different SLD-trees — instead of just creating one single tree, which would include both unfolding steps *and* generalisation steps — is motivated by the fact that these individual trees determine how to generate the new clauses.

The starting point for this paper is that the described analysis scheme suffers from some clear imprecision problems. It has some obvious drawbacks compared to top-down abstract interpretation schemes, such as for instance the one in [1]. These drawbacks are related to two issues:

- the lack of *success-propagation*, both upwards and side-ways,
- the lack of inferring *global* success-information.

We discuss these issues in more detail.

1.1 Lack of success-propagation

Consider the following tiny program:

Example 1.2
$p(X) \leftarrow q(X), r(X)$ $\quad$ $q(a) \leftarrow$ $\quad$ $r(a) \leftarrow$ $\quad$ $r(b) \leftarrow$

For a given query $\leftarrow p(X)$, one possible (although very unoptimal) outcome of the Algorithm 1.1 is the set $S = \{p(X), q(X), r(X)\}$ and the SLD-trees τ_1, τ_2 and τ_3 presented in Fig. 1.

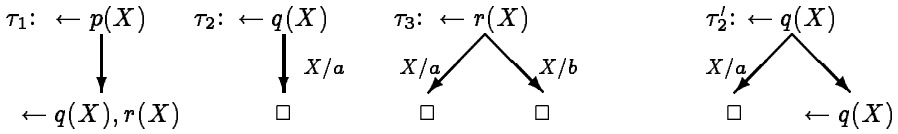


Figure 1: A possible outcome of Algorithm 1.1 for Ex. 1.2 and Ex. 1.2'

With this result of the analysis, the transformed program would be identical to the original one. Note that in τ_2 we have derived that the only answer for $\leftarrow q(X)$ is X/a. An abstract interpretation algorithm such as the one in [1] would propagate this success-information to the leaf of τ_1, yielding that (under the left-to-right selection rule) the call $\leftarrow r(X)$ becomes more specific, namely $\leftarrow r(a)$. This information would then be used in the analysis of the $r/1$ predicate, allowing to remove the redundant branch. Finally, the success-information, X/a, would be propagated up to the $\leftarrow p(x)$ call, yielding a specialised program:

$p(a) \leftarrow \qquad q(a) \leftarrow \qquad r(a) \leftarrow$

which is correct for all instances of the considered query $\leftarrow p(x)$.

Note that this particular example could be solved by the techniques in [5]. There, a limited success-propagation, restricted to only one resolution step, is introduced and referred to as a *more specific resolution step*. The particular example can of course also be solved by standard partial deduction and a sufficiently refined unfolding rule. More difficult and realistic examples, which cannot be solved by neither [5] nor standard partial deduction alone, will be presented later on in the paper.

1.2 Lack of inference of global success-information

Assume that we add the clause $q(x) \leftarrow q(x)$ to the program in Ex. 1.2, yielding Ex. 1.2'. A possible outcome of Algorithm 1.1 for the query $\leftarrow p(x)$ now is $S = \{p(X), q(X), r(X)\}$ and τ_1, τ_2', τ_3, where τ_2' is also depicted in Fig. 1.

Again, the resulting program is identical to the input program. In this case, simple bottom-up propagation of successes is insufficient to produce a better result. An additional fix-point computation is needed to detect that x/a is the only answer substitution. Methods as the one in [1] integrate such fix-point computations in the top-down analysis. As a result, the same more specialised program as for Ex. 1.2 can be obtained.

In addition to pointing out further imprecision problems of the usual analysis scheme, the main contributions of the current paper are:

1. To propose a more refined analysis scheme, building on the notions of *conjunctive partial deduction* (see [17, 9, 12, 20]) and *more specific programs* (see [23, 24]),[2] that solves the above mentioned problems.
2. To illustrate the *applicability* of the new scheme and to describe a *class of applications* in which they are *vital* for successful specialisation.

[2]The method of [23, 24] is the most straightforward to integrate with partial deduction because they both use the same abstract domain: a set of concrete atoms (or goals) is represented by all the instances of a given atom (or goal).

One class of problems is related to the specialisation of meta-programs that use the *ground-representation* for representing object programs. In [3] and [18] it was pointed out that current partial deduction techniques are incapable of specialising such programs satisfactorily. The extensions proposed in the current paper provide a general solution.

The remainder of the paper is organised as follows. In Sect. 2 we present the intuitions behind the proposed solution and illustrate the extensions on a few simple examples. In Sect. 3 we present more realistic, practical examples and and we justify the need for a more refined algorithm. This more refined Algorithm is then presented in Sect. 4 and used to specialise the ground representation in Sect. 5. We conclude with some discussions in Sect. 6.

2 Introducing More Specific Program Specialisation

There are different ways in which one could enhance the analysis to cope with the problems mentioned in the introduction. A solution that seems most promising is to just apply the abstract interpretation scheme of [1] to replace Algorithm 1.1. Unfortunately, this analysis is based on an AND-OR-tree representation of the computation, instead of an SLD-tree representation. As a result, applying the analysis for partial deduction causes considerable problems for the code-generation phase. It becomes very complicated to extract the specialised clauses from the tree. The alternative of adapting the analysis of [1] in the context of an SLD-tree representation causes considerable complications as well. The analysis very heavily exploits the AND-OR-tree representation to enforce termination.

The solution we propose here is based on the combination of two existing analysis schemes, each of which is underlying to a specific specialisation technique: the one of *conjunctive partial deduction* [17, 9, 12, 20] and the one of *more specific programs* [23, 24].

Let us first present an abstract interpretation method based on [23, 24] which calculates more specific versions of programs.

We first introduce the following notations. Given a set of logic formulas P, $Pred(P)$ denotes the set of predicates occuring in P. By $mgu^*(A, B)$ we denote the most general unifier of A and B', where B' is obtained from B by renaming apart wrt A. Next $msg(S)$ denotes the most specific generalisation of the atoms in S. We also define the predicate-wise application msg^* of the msg: $msg^*(S) = \{msg(S^p) \mid p \in Pred(P)\}$, where S^p are all the atoms of S having p as predicate.

In the following we define the well-known non-ground T_P operator along with an abstraction U_P of it.

Definition 2.1 *For a definite logic program P and a set of atoms $\mathcal{A}$ we define:* $T_P(\mathcal{A}) = \{H\theta_1 \ldots \theta_n \mid H \leftarrow B_1, \ldots, B_n \in P \wedge \theta_i = mgu^*(B_i\theta_1 \ldots \theta_{i-1}, A_i)$ *with* $A_i \in \mathcal{A}\}$. *We also define* $U_P(\mathcal{A}) = msg^*(T_P(\mathcal{A}))$.

One of the abstract interpretation methods of [23, 24] can be seen (for maximally general, atomic goal tuples, see Sect. 6) as calculating $lfp(U_P) = U_P \uparrow^\infty (\emptyset)$.[3] In [23, 24] more specific versions of clauses and programs are obtained in the following way:

[3] This in turn can be seen as an abstract interpretation method which infers top level functors for every predicate.

Definition 2.2 *Let $C = H \leftarrow B_1, \ldots, B_n$ be a definite clause and $\mathcal{A}$ a set of atoms. We define $msv_{\mathcal{A}}(C) = \{C\theta_1 \ldots \theta_n \mid \theta_i = mgu^*(B_i\theta_1 \ldots \theta_{i-1}, A_i)$ with $A_i \in \mathcal{A}\}$.*
The more specific version *$msv(P)$ of P is then obtained by replacing every clause $C \in P$ by $msv_{lfp(U_P)}(C)$ (note that $msv_{lfp(U_P)}(C)$ contains at most 1 clause).*

In the light of the stated problems, an integration of partial deduction with the more specific program transformation seems a quite natural solution. In [23, 24] such an integration was already suggested as a promising future direction. Take for instance the following program.

Example 2.3 (eqlist)

```
eqlist(X,Z) :- append(X,[],Z).
append([],L,L).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

Partial deduction for the goal `eqlist(X,Z)` followed by the more specific program transformation $msv(.)$ yields the specialised version:

```
eqlist(X,X) :- append(X,[],X).
```

while $msv(.)$ alone is incapable of doing so. The following example reveals that, in general, this combination is still too weak to deal with side-ways information propagation.

Example 2.4 (append-last)

```
app_last(L,X) :- append(L,[a],R), last(R,X).
append([],L,L).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
last([X],X).
last([H|T],X) :- last(T,X).
```

The hope is that the specialisation techniques are sufficiently strong to infer that a query `app_last(L,X)` produces the answer `X=a`. Partial deduction on its own is incapable of producing this result. An SLD-tree for the query `app_last(L,X)` takes the form of τ_1 in Fig. 2. Although the success-branch of the tree produces `X=a`, there are infinitely many possibilities for `L` and, without a bottom-up fixed-point computation, `X=a` cannot be derived for the entire computation. At some point the unfolding needs to terminate, and additional trees for `append` and `last`, for instance τ_2 and τ_3 in Fig. 2, need to be constructed. The resulting program is:

```
app_last([],a).
app_last([H|L'],X) :- append__1(L',[a],R'), last(R',X)
append([],[a],[a]).
append([H | X],[a],[H|Z]) :- append(X,[a],Z)
```

in addition to the original clauses for `last/2`.

Unfortunately, in this case, even the combination with the more specific program transformation is insufficient to obtain the desired result. We get:

$T_P(U_P \uparrow 1) = T_P \uparrow 2 =$
{ `app_last(a)`, `append([],[a],[a])`, `append([H],[a],[H,a])`, `last([X],X)`,`last([H,X],X)` }

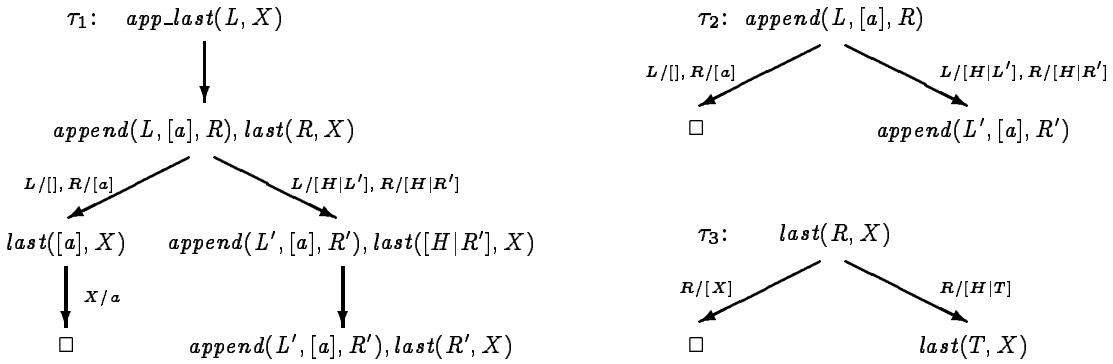


Figure 2: SLD-trees for Ex. 2.4

after which most specific generalisation yields

$U_P \uparrow 2 =$ { `app_last(a)`, `append(X,[a],[Y|Z])`, `last([X|Y], Z)` }

At this stage, all information concerning the last elements of the lists is lost and we reach the fix-point in the next iteration:

$U_P \uparrow 3 =$ { `app_last(Z)`, `append(X,[a],[Y|Z])`, `last([X|Y],Z)` }

One could argue that the failure is not due to the more specific programs transformation itself, but to a weakness of the most specific generalisation operator: it's inability to retain information at the end of a data-structure. Note however that even if we use other abstractions and their corresponding abstract operation proposed in the literature, such as *type-graphs* [11], *regular types* [8] or refined types for compile-time garbage collection of [25], the information still gets lost.

The heart of the problem is that in all these methods the abstract operator is applied to atoms of each predicate symbol *separately*. In this program (and *many, much more relevant others*, as we will discuss later in the paper), we are interested in analysing the conjunction `append(L,[a],R),last(R,X)` with a linking intermediate variable (whose structure is too complex for the particular abstract domain). If we could consider this conjunction as a *basic unit* in the analysis, and therefore not perform abstraction on the separate atoms, but only on conjunctions of the involved atoms, we would retain a precise side-ways information passing analysis.

In [17] we have developed a minimal extension to partial deduction, called *conjunctive partial deduction*. This technique extends the standard partial deduction approach by:

- considering a set $S = \{C_1, \ldots, C_n\}$ of *conjunctions of atoms* instead of individual atoms, and
- building an SLD-tree τ_i for each $P \cup \{\leftarrow C_i\}$,

such that the query $\leftarrow C$ of interest (which may now be a non-atomic goal) as well as each goal $\leftarrow B_1, \ldots, B_k$ labelling a leaf of some SLD-tree τ_i, C (resp. $B_1, \ldots, B_k$) can be partitioned into conjunctions $C'_1, \ldots, C'_r$, such that each C'_i is an instance of some $C_j \in S$.

The following basic notion is adapted from [21].

Definition 2.5 (resultant) *Let P be a program, $G = \leftarrow Q$ a goal, where Q is a conjunction of atoms, $G_0 = G, G_1, ..., G_n$ a finite derivation for $P \cup \{G\}$, with substitutions $\theta_1, ..., \theta_n$, and let G_n have the form $\leftarrow Q_n$. We say that $Q\theta_1...\theta_n \leftarrow Q_n$ is* the resultant *of the derivation $G_0, G_1, ..., G_n$.*

The notion can be generalised to SLD-trees. Given a finite SLD-tree τ for $P \cup \{G\}$, there is a corresponding set of resultants R_τ, including one resultant for each non-failed derivation of τ.

In the partial deduction notion introduced in [21], the SLD-trees are restricted to *atomic* top-level goals. This restriction has been omitted in [17] and therefore resultants of the SLD-trees are not necessarily clauses: their left-hand side may contain a conjunction of atoms. To transform such resultants back into standard clauses, conjunctive partial deduction involves a renaming transformation, from conjunctions to atoms with new predicate symbols, in a post-processing step. For further details on conjunctive partial deduction we refer to [17, 9, 12, 20].

Although this extension of standard partial deduction was motivated by totally different concerns than the ones in the current paper (the aim was to achieve a large class of unfold/fold transformations [26] within a simple extension of the partial deduction framework), experiments with conjunctive partial deduction on standard partial deduction examples also showed significant improvements. Only in retrospect we realised that these optimisations were due to considerably improved side-ways information-propagation.

Let us illustrate how conjunctive partial deduction combined with the more specific program transformation *does* solve Ex. 2.4. Starting from the goal `app_last(X)` and using an analysis scheme similar to Algorithm 1.1, but with the role of atoms replaced by conjunctions of atoms, we can obtain $S = \{$ `app_last(X)`, `append(L,[a],R)` $\wedge$ `last(R,X)` $\}$ and the corresponding SLD-trees, which are sub-trees of τ_1 of Fig. 2. Here, "$\wedge$" is used to denote conjunction in those cases where "," is ambiguous.

The main difference with the (standard) partial deduction analysis above is that the goal `append(L',[a],R'),last(R',X)` in the leaf of τ_1 is now considered as an undecomposed conjunction. This conjunction is already an instance of an element in S, so that no separate analysis for `append(L',[a],R')` and `last(R',X)` are required.

Using a renaming transformation: $rename($`append(x,y,z)`$\wedge$`last(z,u)`$) =$ `al(x,y,z,u)` the resulting transformed program is:

```
app_last(L,X) :- al(L,[a],R,X)
al([],[a],[a],a).
al([H|L'],[a],[H | R'], X) :- al(L',[a],R',X).
```

Applying the non-ground T_P-operator and more specific generalisation abstractions produces the sets:

$U_P \uparrow 1 =$ { `al([],[a],[a],a)` } $\quad U_P \uparrow 2 =$ { `al(X,[a],Y,a)`, `app_last(a)` }

which is a fix-point. Unifying the success-information with the body-atoms in the above program and performing argument filtering produces the desired more specific program:

```
app_last(L,a) :- al(L).
al([]).
al([H|L']) :- al(L').
```

3 Some Motivating Examples

In this section we illustrate the relevance of the introduced techniques by more realistic, practical examples.

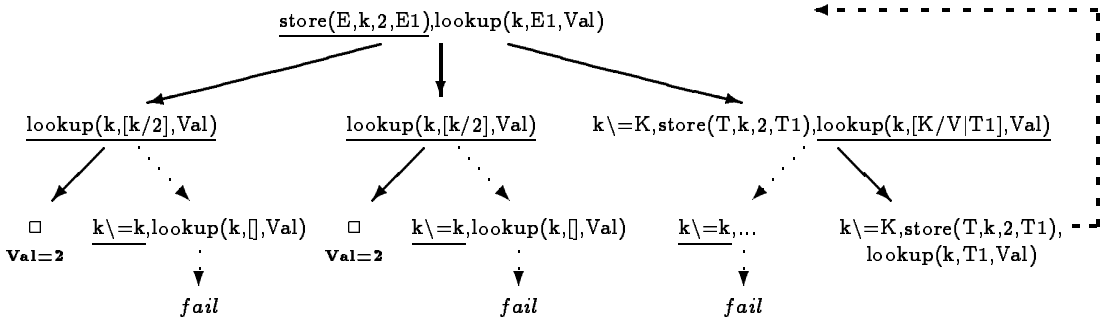
3.1 Storing values in an environment

The following piece of code P stores values of variables in an association list and is taken from a meta-interpreter for imperative languages ([14]).

```
store([],Key,Value,[Key/Value]).
store([Key/Value2|T],Key,Value,[Key/Value|T]).
store([K2/V2|T],Key,Value,[K2/V2|BT]) :- Key \= K2,store(T,Key,Value,BT).
lookup(Key,[Key/Value|T],Value).
lookup(Key,[K2/V2|T],Value) :- Key \= K2,lookup(Key,T,Value).
```

During specialisation it may happen that a known (static) value is stored in an unknown environment.[4] When we later on retrieve this value from the environment it is vital for good specialisation to be able to recover this static value. This is a problem quite similar to the `append-last` problem of Ex. 2.4. So again, calculating $msv(P)$ (even if we perform a magic-set transformation on P) does not give us any new information for a query like `store(E,k,2,E1),lookup(E1,k,Val)`. To be able to solve this problem one needs again to combine abstract interpretation with conjunctive partial deduction (to "deforest" [30] the intermediate environment E_1). The specialised program P' for the query `store(E,k,2,E1),lookup(E1,k,Val)` using the ECCE ([14]) system with determinate unfolding is the following (a duplicate `k\=X1` has been removed in the third clause):

```
store_lookup__1([],[k/2],2).
store_lookup__1([k/X1|X2],[key/2|X2],2).
store_lookup__1([X1/X2|X3],[X1/X2|X4],X5) :-
    k \= X1,store_lookup__1(X3,X4,X5).
```



If we now calculate $msv(P')$, we are able to derive that `Val` must have the value 2:

```
store_lookup__1([],[k/2],2).
store_lookup__1([k/X1|X2],[k/2|X2],2).
store_lookup__1([X1/X2|X3],[X1/X2,X4/X5|X6],2) :-
    k \= X1,store_lookup__1(X3,[X4/X5|X6],2).
```

Being able to derive this kind of information is of course even more relevant when one can continue specialisation with it. For instance in an interpreter for an imperative language there might be multiple static values which are stored and then examined again. These values might control tests or loops, and for good specialisation to take place one needs to be able to extract the kind of information above.

[4] The environment might be unknown for many reasons, e.g. due to abstraction or because part of the imperative program is unknown.

3.2 Proving Functionality

The following is a generalisation of the standard definition of functionality (see e.g. [27] or [4]).

Definition 3.1 *We say that a predicate p defined by a program P is* functional wrt the terms $t_1, \ldots, t_h$ *iff for every pair of atoms $A = p(t_1, \ldots, t_h, a_1, \ldots, a_k)$ and $B = p(t_1, \ldots, t_h, b_1, \ldots, b_k)$ we have that:*

- $\leftarrow A, B$ *has a correct answer θ iff* $\leftarrow A, A = B$ *has*
- $\leftarrow A, B$ *finitely fails iff* $\leftarrow A, A = B$ *finitely fails*

Note that in the above definition we allow A, B to be used as atoms as well as terms (as arguments to the predicate $= /2$). Also note that for simplicity of the presentation we restrict ourselves to correct answers. Therefore it can be easily seen that,[5] if the goal $\leftarrow A', A'$ is a more specific version of $\leftarrow A, B$ then p is functional wrt $t_1, \ldots, t_h$ (because $msv(.)$ preserves computed answers and removing syntactically identical calls preserves the correct answers for definite logic programs).

Functionality is useful for many transformations, and is often vital to get super-linear speedups. For instance it is needed to transform the naive (exponential) Fibonacci program into a linear one (see e.g. [27]). It can also be used to produce more efficient code (see e.g. [4]). Another example arises naturally from the `store-lookup` code of the previous section. In a lot of cases, specialisation can be greatly improved if functionality of `lookup(Key,Env,Val)` wrt a given key `Key` and a given environment `Env` can be proven (in other words if we lookup the same variable in the same environment we get the same value). For instance this would allow to replace, during specialisation, `lookup(Key,Env,V1),lookup(Key,Env,V2),p(V2)` by `lookup(Key,Env,V1),p(V1)`.

To prove functionality of `lookup(Key,Env,Val)` we simply add the following definition:[6]
`ll(K,E,V1,V2) :- lookup(K,E,V1),lookup(K,E,V2).` By specialising the query `ll(Key,Env,V1,V2)` using the ECCE system with determinate unfolding and then calculating $msv(.)$ for the resulting program, we are able to derive that `V1` must be equal to `V2`:

```
ll(K,E,V,V) :- lookup_lookup__1(K,E,V,V).
lookup_lookup__1(X1,[X1/X2|X3],X2,X2).
lookup_lookup__1(X1,[X2/X3,X4/X5|X6],X7,X7) :-
    X1 \= X2, lookup_lookup__1(X1,[X4/X5|X6],X7,X7).
```

In addition to obtaining a more efficient program the above implies (because conjunctive partial deduction preserves the computed answers) that `lookup(K,E,V),lookup(K,E,V)` is a more specific version of `lookup(K,E,V1),lookup(K,E,V2)`, and we have proven functionality of `lookup(Key,Env,Val)` wrt `Key,Env`.

In the same vein we can for example prove functionality of `plus/3`:

```
pp(X,Y,V,V) :- plus_plus__1(X,Y,V,V).
plus_plus__1(0,X1,X1,X1).
plus_plus__1(s(X1),X2,s(X3),s(X3)) :- plus_plus__1(X1,X2,X3,X3).
```

[5] The reasoning for computed answers is not so obvious.

[6] This is not strictly necessary but it simplifies spotting functionality.

3.3 The Need for a More Refined Integration

So far we have always completely separated the conjunctive partial deduction phase and the bottom-up abstract interpretation phase. The following example (which arose from a practical application described in Sect. 5) shows that this is not always sufficient.

Take a look at the following excerpt from a unification algorithm for the ground representation (the full code is in Appendix A), which takes care of extracting variable bindings out of (uncomposed) substitutions.

```
get_binding(V,empty,var(V)).
get_binding(V,sub(V,S),S).
get_binding(V,sub(W,S),var(V)) :- V \= W.
get_binding(V,comp(L,R),S) :- get_binding(V,L,VL), apply(VL,R,S).

apply(var(V),Sub,VS) :- get_binding(V,Sub,VS).
apply(struct(F,A),Sub,struct(F,AA)) :- l_apply(A,Sub,AA).

l_apply([],Sub,[]).
l_apply([H|T],Sub,[AH|AT]) :- apply(H,Sub,AH),l_apply(T,Sub,AT).
```

At first sight this looks very similar to the example of the previous section and one would think that we could easily prove functionality of `get_binding(VarIdx,Sub,Bind)` wrt a particular variable index `VarIdx` and a particular substitution `Sub`. Exactly this kind of information is required for the practical applications in Sect. 5.

Unfortunately this kind of information *cannot* be obtained by fully separated out phases. For simplicity we assume that the variable index `VarIdx` is known to be 1. Taking the approach of the previous section we would add the definition

```
gg(Sub,V1,V2) :- get_binding(1,Sub,V1),get_binding(1,Sub,V2).
```

and apply conjunctive partial deduction and $msv(.)$ to obtain:

```
gg(Sub,V1,V2) :- get_binding_get_binding__1(Sub,V1,V2).
get_binding_get_binding__1(empty,var(1),var(1)).
get_binding_get_binding__1(sub(1,X1),X1,X1).
get_binding_get_binding__1(sub(X1,X2),var(1),var(1)) :- 1 \= X1.
get_binding_get_binding__1(comp(X1,X2),X3,X4) :-
    get_binding_get_binding__1(1,X1,X5,X6),apply(X5,X2,X3),apply(X6,X2,X4).
```

By applying conjunctive partial deduction to `apply(X5,X2,X3),apply(X6,X2,X4)` of clause 5 we cannot derive that `X3` must be equal to `X4` because the variables indexes `X5` and `X6` are different (applying the same substitution on different terms can of course lead to differing results). However, if we re-apply conjunctive partial deduction *before* reaching the fixpoint of U_P, we can solve the above problem. Indeed after one application of U_P we obtain $\mathcal{A} = U_P(\emptyset) = \{$`get_binding_get_binding__1(S,V,V)`$\}$ and at that point we have that $msv_{\mathcal{A}}(.)$ of clause 5 looks like:

```
get_binding_get_binding__1(comp(X1,X2),X3,X4) :-
    get_binding_get_binding__1(1,X1,V,V),apply(V,X2,X3),apply(V,X2,X4).
```

If we recursively apply conjunctive partial deduction to `apply(V,X2,X3),apply(V,X2,X4)` and then similarly on `l_apply(V,X2,X3),l_apply(V,X2,X4)` we can derive functionality of `get_binding`. The details of this more refined integration are elaborated in the next section.

4 A More Refined Algorithm

We now present an algorithm which interleaves the least fixpoint construction of $msv(.)$ with conjunctive partial deduction unfolding steps. For that we have to adapt the more specific program transformation to work on incomplete SLDNF-trees obtained by conjunctive partial deduction instead of for completely constructed programs.[7]

We first introduce a special conjunction $\bot$ which is an instance of every conjunction, as well as the only instance of itself, and extend the msg such that $msg(S \cup \{\bot\}) = msg(S)$ and $msg(\{\bot\}) = \bot$. We also use the convention that if unification fails it returns a special substitution $fail$. Applying $fail$ to any conjunction Q in turn yields $\bot$. Finally by $\uplus$ we denote the concatenation of tuples (e.g. $\langle a \rangle \uplus \langle b, c \rangle = \langle a, b, c \rangle$).

In the following definition we associate conjunctions with resultants:

Definition 4.1 (resultant tuple) *Let $S = \{Q_1, ..., Q_s\}$ be a set of conjunctions of atoms, and $T = \{\tau_1, ..., \tau_s\}$ a set of finite, non-trivial SLD-trees for $P \cup \{\leftarrow Q_1\}, ..., P \cup \{\leftarrow Q_s\}$, with associated sets of resultants $R_1, ..., R_s$, respectively. Then the tuple of pairs $RS = \langle (Q_1, R_1), \ldots, (Q_s, R_s) \rangle$ is called a* resultant tuple *for P. An* interpretation *of RS is a tuple $\langle Q'_1, \ldots, Q'_s \rangle$ of conjunctions such that Q'_i is an instance of Q_i.*

The following defines how interpretations of resultant tuples can be used to create more specific resultants:

Definition 4.2 (refinement) *Let $R = H \leftarrow Body$ be a resultant and $I = \langle Q'_1, \ldots, Q'_s \rangle$ be an interpretation of a resultant tuple $RS = \langle (Q_1, R_1), \ldots, (Q_s, R_s) \rangle$. Let Q be a sub-goal of $Body$ such that Q is an instance of Q_i and such that $mgu^*(Q, Q'_i) = \theta$. Then $R\theta$ is called a* refinement *of R under RS and I. R itself, as well as any refinement of $R\theta$, is also called a refinement of R under RS and I.*

Note that a least refinement does not always exist. Take for instance $R = q \leftarrow p(X, f(T)) \wedge p(T, X)$, $RS = \langle (p(X, Y), R') \rangle$ and $I = \langle p(X, X) \rangle$. We can construct an infinite sequence of successive refinements of R under RS and I: $q \leftarrow p(f(X'), f(T)) \wedge p(T, f(X'))$, $q \leftarrow p(f(X'), f(f(T'))) \wedge p(f(T'), f(X'))$, ... Hence we denote by $ref_{RS,I}(R)$, a particular refinement of R under RS and I.[8] A pragmatic approach might be to allow any particular sub-goal to be unified only once with any particular element of I.

Note that in [23, 24], it is not allowed to further refine refinements and therefore only finitely many refinements exist and a least refinement can be obtained by taking the mgu^* of all of them. As we found out through several examples however, (notably the ones of Sect. 3.3 and Sect. 5) this approach turns out to be too restrictive in general. In a lot of cases, applying a first refinement might instantiate R in such a way that a previously inapplicable element of RS can now be used for further instantiation.

We can now extend the U_P operator of Def 2.1 to work on interpretations of resultant tuples:

Definition 4.3 ($\mathbf{U}_{P,RS}$) *Let $I = \langle Q'_1, \ldots, Q'_s \rangle$ be an interpretation of a resultant tuple $RS = \langle (Q_1, R_1), \ldots, (Q_s, R_s) \rangle$. Then $U_{P,RS}$ is defined by $U_{P,RS}(I) = \langle M_1, \ldots, M_s \rangle$, where $M_i = msg(\{H \mid C \in R_i \wedge ref_{RS,I}(C) = H \leftarrow B\})$.*

[7] This has the advantage that we do not actually have to apply a renaming transformation (and we might get more precision because several conjunctions might match).

[8] It is probably correct to use $\bot$ if the least refinement does not exist but we have not investigated this.

We refer the reader to Ex. 4.7 and Table 1 below for illustrations of the above concepts.

We can now present a generic algorithm which fully integrates the abstract interpretation $msv(.)$ with conjunctive partial deduction. Below, $=_r$ denotes syntactic identity, up to reordering.

We first define an abstraction operation, which is used to ensure termination of the conjunctive partial deduction process (see [9] for some such abstraction operations).

Definition 4.4 (abstraction) *A multi-set of conjunctions $\{Q_1, \ldots, Q_k\}$ is an* abstraction *of a conjunction Q iff for some substitutions $\theta_1, \ldots, \theta_k$ we have that $Q =_r Q_1\theta_1 \wedge \ldots \wedge Q_k\theta_k$. An* abstraction operation *is an operation which maps every conjunction to an abstraction of it.*

We also need the following definition:

Definition 4.5 (covered) *Let $RS = \langle (Q_1, R_1), \ldots, (Q_s, R_s)\rangle$ be a resultant tuple. We say that a conjunction Q is* covered *by RS iff there exists an abstraction $\{Q'_1, \ldots, Q'_k\}$ of Q such that each Q'_i is an instance of some Q_j.*

We can now present the promised algorithm.

Algorithm 4.6 (Conjunctive Msv)

Input: a program P, an initial query Q, an unfolding rule $unfold$ for P mapping conjunctions to resultants.
Output: A specialised and more specific program P' for Q.
Initialisation: $i := 0$; $I_0 = \langle \bot \rangle$, $RS_0 = \langle (Q, unfold(Q))\rangle$
repeat
  for every resultant R in RS_i such that the body B of $ref_{RS_i,I_i}(R)$ is not covered:
    /* perform conjunctive partial deduction: */
    calculate $abstract(B) = B_1 \wedge \ldots \wedge B_q$
    let $\{C_1, \ldots, C_k\}$ be the B_j's which are not instances[9] of conjunctions in RS_i
    $RS_{i+1} = RS_i \uplus \langle (C_1, unfold(C_1)), \ldots (C_k, unfold(C_k))\rangle$;
    $I_{i+1} = I_i \uplus \langle \underbrace{\bot \ldots \bot}_{k} \rangle$; $i := i + 1$.
  /* perform one bottom-up propagation step: */
  $I_{i+1} = U_{P,RS_i}(I_i)$; $RS_{i+1} = RS_i$; $i := i + 1$.
until $I_i = I_{i-1}$
return a renaming of $\{ref_{RS_i,I_i}(C) \mid (Q, R) \in RS_i \wedge C \in R\}$

Note that the above algorithm ensures coveredness. Also note that the above algorithm performs abstraction only when adding new conjunctions, the existing conjunctions are not abstracted (it is of course trivial to adapt this). This is like in [19] but unlike e.g. Algorithm 1.1.

Example 4.7 We now illustrate Algorithm 4.6 by proving functionality of `mul(X,Y,Z1)`, `mul(X,Y,Z2)` for the following program. Note that the general picture is very similar to showing functionality of `get_binding`, but leading to a shorter and simpler presentation.

```
mul(0,X,0).
mul(s(X),Y,Z) :- mul(X,Y,XY), plus(XY,Y,Z).
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

R_1	`mul(0,Y,0),mul(0,Y,0).`
R_2	`mul(s(X),Y,Z1),mul(s(X),Y,Z2) :- mul(X,Y,XY1),plus(XY1,Y,Z1),mul(X,Y,XY2),plus(XY2,Y,Z2).`
R_2'	`mul(s(0),Y,Z1),mul(s(0),Y,Z2) :- mul(0,Y,0),plus(0,Y,Z1),mul(0,Y,0),plus(0,Y,Z2).`
R_2''	`mul(s(0),Y,Y),mul(s(0),Y,Y) :- mul(0,Y,0),plus(0,Y,Y),mul(0,Y,0),plus(0,Y,Y).`
R_2'''	`mul(s(X),Y,Z1),mul(s(X),Y,Z2) :- mul(X,Y,Z),plus(Z,Y,Z1),mul(X,Y,Z),plus(Z,Y,Z2).`
R_2''''	`mul(s(X),Y,V),mul(s(X),Y,V) :- mul(X,Y,Z),plus(Z,Y,V),mul(X,Y,Z),plus(Z,Y,V).`
R_3	`plus(0,Y,Y),plus(0,Y,Y).`
R_4	`plus(s(X),Y,Z1),plus(s(X),Y,Z2) :- plus(X,Y,Z1),plus(X,Y,Z2).`

Table 1: Resultants and refinements

The resultants R_1, R_2, R_3, R_4 and their refinements for this example can be found in Table 1. Using an abstraction based on [9] and a determinate unfolding rule (see e.g. [7, 5, 15]) we obtain the following behaviour of Algorithm 4.6.

1. Initialisation: $I_0 = \langle \bot \rangle$, $RS_0 = \langle (\texttt{mul(X,Y,Z1)} \wedge \texttt{mul(X,Y,Z2)}, \{R_1, R_2\}) \rangle$
2. $ref_{RS_0,I_0}(R_2) = \bot$ and therefore all bodies are covered
3. We perform a bottom-up propagation step:
   $RS_1 = RS_0$, $I_1 = U_{P,RS_0}(I_0) = \langle \texttt{mul(0,Y,0)} \wedge \texttt{mul(0,Y,0)} \rangle \neq I_0$
4. Now $ref_{RS_1,I_1}(R_2) = R_2'$ and *abstract* of the body of R_2' yields:
   $\{\texttt{mul(0,Y,0)} \wedge \texttt{mul(0,Y,0)}, \texttt{plus(0,Y,Z1)} \wedge \texttt{plus(0,Y,Z2)}\}$ and we obtain:
   $I_2 = I_1 \uplus \langle \bot \rangle$, $RS_2 = RS_1 \uplus \langle (\texttt{plus(0,Y,Z1)} \wedge \texttt{plus(0,Y,Z2)}, \{R_3\}) \rangle$
5. We now go on with the bottom-up propagation: $RS_3 = RS_2$,
   $I_3 = U_{P,RS_2}(I_2) = \langle \texttt{mul(0,Y,0)} \wedge \texttt{mul(0,Y,0)}, \texttt{plus(0,Y,Y)} \wedge \texttt{plus(0,Y,Y)} \rangle$
6. The body of $ref_{RS_3,I_3}(R_2) = R_2''$ is covered and we go on with the bottom-up propagation: $RS_4 = RS_3$,
   $I_4 = U_{P,RS_3}(I_3) = \langle \texttt{mul(X,Y,Z)} \wedge \texttt{mul(X,Y,Z)}, \texttt{plus(0,Y,Y)} \wedge \texttt{plus(0,Y,Y)} \rangle$
7. Now $ref_{RS_4,I_4}(R_2) = R_2'''$ is no longer covered. *abstract* of the body of R_2''' yields:
   $\{\texttt{mul(X,Y,Z)} \wedge \texttt{mul(X,Y,Z)}, \texttt{plus(Z,Y,Z1)} \wedge \texttt{plus(Z,Y,Z2)}\}$ and $I_5 = I_4 \uplus \langle \bot \rangle$,
   $RS_5 = RS_4 \uplus \langle (\texttt{plus(Z,Y,Z1)} \wedge \texttt{plus(Z,Y,Z2)}, \{R_3, R_4\}) \rangle$
8. We do a bottom-up propagation step: $RS_6 = RS_5$, $I_6 = U_{P,RS_5}(I_5) =$
   $\langle \texttt{mul(X,Y,Z)} \wedge \texttt{mul(X,Y,Z)}, \texttt{plus(0,Y,Y)} \wedge \texttt{plus(0,Y,Y)}, \texttt{plus(Z,Y,V)} \wedge \texttt{plus(Z,Y,V)} \rangle$
9. The bodies of $ref_{RS_6,I_6}(R_2) = R_2''''$ and $ref_{RS_6,I_6}(R_4)$ are covered and we have reached the fixpoint: $I_7 = U_{P,RS_6}(I_6) = I_6$.

The final specialised program is as follows (an unreachable predicate has been removed, `mul(X,Y,Z1)`, `mul(X,Y,Z2)` has been renamed to `mul_mul(X,Y,Z1,Z2)` and `plus(X,Y,Z1)`, `plus(X,Y,Z2)` has been renamed to `plus_plus(X,Y,Z1,Z2)`; the program could be further improved by a better renaming) and functionality is obvious:

```
mul_mul(0,Y,0,0).
mul_mul(s(X),Y,Z,Z) :- mul_mul(X,Y,XY,XY),plus_plus(XY,Y,Z,Z).
plus_plus(0,X,X,X).
plus_plus(s(X),Y,s(Z),s(Z)) :- plus_plus(X,Y,Z,Z).
```

Note that when using the definitions of [23, 24] the least refinement of R_2 wrt RS_6 and I_6 is not R_2''' (because `plus(Z,Y,Z')`$\wedge$`plus(Z,Y,Z')` cannot be applied) but R_2''. Hence the fixpoint is not reached and in the next iteration the vital functionality information would be lost!

[9] Or *variants* to make the algorithm more precise.

Correctness of Algorithm 4.6 for preserving Least Herbrand Model as well as computed answers, follows from correctness of conjunctive partial deduction (see [17]) and of the more specific program versions for suitably chosen conjunctions (because [23, 24] only allows one unfolding step, a lot of intermediate conjunctions have to be introduced) and extended for the more powerful refinements of Def 4.2. Termination, for a suitable abstraction operation (see [9]), follows from termination of conjunctive partial deduction (for the **for** loop) and termination of $msv(.)$ (for the **repeat** loop).

Note that in contrast to conjunctive partial deduction, $msv(.)$ can replace infinite failure by finite failure, and hence Algorithm 4.6 does not preserve finite failure. However, if the specialised program fails infinitely, then so does the original one (see [23, 24]).

The above algorithm can be extended to work for normal logic programs. But, because finite failure is not preserved, neither are the SLDNF computed answers. One may have to look at SLS [28] for a suitable procedural semantics which is preserved.

5 Specialising the Ground Representation

Integrity constraints play a crucial role in deductive databases, abductive and inductive logic programs. From a practical viewpoint however, it can be quite expensive to check the integrity after each update. To alleviate this problem, special purpose integrity simplification methods have been proposed, taking advantage of the fact that the program/database was consistent prior to the update. In [16] such a simplification procedure for hierarchical databases was written as a meta-program, which was then specialised for update patterns, resulting in very efficient, pre-compiled integrity checks.

In [18] this approach was extended to *recursive* databases/programs. In contrast to the hierarchical case, the *ground representation* had to be used to write the simplification procedure. It was also shown that, contrary to what one might expect, partial deduction was then unable to perform interesting specialisation and no pre-compilation of integrity checks could be obtained. This problem was solved in [18] via a new implementation of the ground representation combined with a custom specialisation technique.

The crucial problem in [18] boiled down to a lack of information propagation at the object level. Indeed, the ground representation entails the use of an explicit unification algorithm (like the one in Appendix A) at the object level. A meta-interpreter which implements the specialised integrity checking of [22] for instance, will select an atom A from the original update pattern and unify this atom with an atom B_i in the body of a clause $H \leftarrow B_1 \ldots, B_n$ and then apply the unifier to the head H to obtain an induced, potential update. At pre-compilation time, the atom A, and maybe also H and B_i, are not fully known. If we want to obtain effective specialisation, it is vital that the information we do possess about A (and B_i) is propagated "through" unification towards H. If this knowledge is not carried along then it is impossible to obtain efficient pre-compiled integrity checks.

In this section we show that the techniques of this paper can solve this information propagation problem in a more general and sometimes more precise manner and therefore contribute to produce highly specialised and efficient pre-compiled integrity checks.

As already mentioned, we are interested in deriving properties of the result `Res` of calculating `unify(A,B,S),apply(H,S,Res)`. In a concrete example we might have `A = status(X,student,Age)`, `B = status(ID,E,A)`, `H = category(ID,E)` and we would like to derive

that `Res` must be an instance of `category(ID',student)`. However it turns out that, when using an explicit unification algorithm, the substitutions have a much more complex structure than e.g. the intermediate list of the append-last example 2.4. Therefore current abstract interpretation methods, as well as current partial deduction methods alone, fail to derive the desired information.

Fortunately Algorithm 4.6 provides an elegant and powerful solution to this problem. Some experiments, conducted with a prototype implementation of Algorithm 4.6 based on the ECCE system [14], are summarised in Table 2. The unification algorithm of Appendix A has been used, which encodes variables as `var(VarIndex)` and predicates/functors as `struct(p,Args)`. Notice that all the examples were successfully solved by the prototype.[10] The main ingredient of the success lay with proving functionality of `get_binding`.

Also note that the information propagations of Table 2 could neither be solved by regular approximations ([8]), nor by the abstract interpretation method of [23, 24] alone, nor by set-based analysis ([10]) nor even by current implementations of the type graphs of [11]. In summary, Algorithm 4.6 also provides for a powerful abstract interpretation scheme as well as a full replacement of the custom specialisation technique in [18].[11]

`unify(A,B,S),apply(H,S,Res)`			
`A`	`B`	`H`	`Res`
`struct(p,[var(1),X])`	`struct(p,[struct(a,[]),Y])`	`var(1)`	`struct(a,[])`
`struct(p,[X,var(1)])`	`struct(p,[Y,struct(a,[])])`	`var(1)`	`struct(a,[])`
`struct(p,[X,X])`	`struct(p,[struct(a,[]),Y])`	`X`	`struct(a,[])`
`struct(F,[var(I)])`	`X`	`X`	`struct(F,[A])`
`struct(p,[X1,var(1),X2])`	`struct(p,[Y1,struct(a,[]),Y2])`	`var(1)`	`struct(a,[])`

Table 2: Specialising the Ground Representation

6 Discussion

The approach presented in this paper can be seen as a practical realisation of a combined backwards and forwards analysis as outlined in [2], but using the sophisticated control techniques of (conjunctive) partial deduction to guide the analysis. Of course, in addition to analysis, our approach also constructs a specialised, more efficient program.

The method of [23, 24] is not directly based on the T_P operator, but uses an operator on goal tuples which can handle conjunctions and which is sufficiently precise if deforestation can be obtained by 1-step unfolding without abstraction. For a lot of practical examples this will of course not be the case. Also, apart from a simple pragmatic approach, no way to obtain these conjunctions is provided (this is exactly one of the things which conjunctive partial deduction can do). We also already mentioned a drawback in the calculation of refinements, which makes [23, 24] unsuitable to derive functionality of `get_binding` or `mul`.

In Algorithm 4.6 a conflict between efficiency and precision might arise. Indeed some deforestation can only be obtained at the cost of possible slowdowns. But Algorithm 4.6 can be easily extended to allow different trees for the same conjunction (e.g. use determinate unfolding for efficient code and a more liberal unfolding for a precise analysis).

[10] Notice that for the fourth example it would be incorrect to derive $Res = struct(F, [var(I)])$ (e.g. if $B = X = struct(f, struct(a, []))$ then $Res = X = struct(f, struct(a, []))$).

[11] It is sometimes even able to provide better results because it can handle structures with unknown functors or unknown number of arguments with no loss of precision.

When using the unification algorithm from [3, 18], instead of the one in Appendix A, Algorithm 4.6 cannot yet handle all the examples of Table 2. The reason is that the substitutions in [3, 18] are actually *accumulating* parameters which are first fully generated before they can be consumed! Deforestation of accumulators is still an open research problem (for functional languages, first, not yet automatic, approaches can be found in [29]). Let us adapt Ex. 2.4 into the `rev_last` example:

Example 6.1 (reverse-last)

```
rev_last(L,X) :- reverse(L,[a],R), last(R,X).
reverse([],L,L).
reverse([H|T],Acc,Res) :- reverse(T,[H|Acc],Res).
last([X],X).
last([H|T],X) :- last(T,X).
```

In the above program `reverse` is written using an accumulating parameter, and in that case neither conjunctive partial deduction nor any unfold/fold method we know of can deforest the intermediate variable `R`. Unfolding the goal `reverse(L,[a],R), last(R,X)` is depicted in Figure 3. Notice that no matter how we unfold we cannot obtain a recursive definition. Conjunctive partial deduction would detect the growing of the accumulator and produce the abstraction `reverse(L,A,R), last(R,X)`. Unfolding can now produce a recursive definition, as can be seen in Figure 4. However the partial input `a` has been abstracted away, and we are not able to deduce that `X=a`.

On the other hand, if we use the naive reverse without an accumulator, defined by:

```
nrev([],[]).
nrev([H|T],Res) :- nrev(T,TR),append(TR,[H],Res).
```

then, after unfolding `nrev([a|L],R),last(R,X)` once, we obtain `nrev(L,TR), append(TR,[a],R), last(R,X)`. We are now in the situation of the append-last example Ex. 2.4 and `X=a` ca be easily obtained via Algorithm 4.6.

In conclusion, in this paper we have illustrated limitations of both partial deduction and abstract interpretation on their own. We have argued for a tighter integration of these methods and presented a refined algorithm, interleaving a least fixpoint construction with conjunctive partial deduction. The practical relevance of this approach has been illustrated by several examples and we have shown its usefulness in proving functionality. Finally, a prototype implementation of the algorithm was able to achieve sophisticated specialisation *and* analysis for meta-interpreters written in the ground representation, outside the reach of current specialisation or abstract interpretation techniques.

Acknowledgments

The authors greatly benefited from stimulating discussions with Jesper Jørgensen and Bern Martens. We are grateful to Ulf Nilsson for pointing out a mistake in the formulation of the T_P operator. We also thank anonymous referees for their helpful comments.
The work on conjunctive partial deduction grew out of joint work with Robert Glück, Jesper Jørgensen, Bern Martens, Morten Heine Sørensen and André de Waal.
An abridged version of this paper will appear in the Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96), LNCS, Springer Verlag.

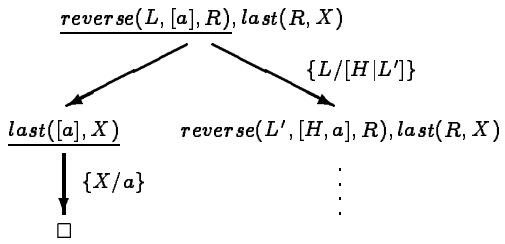


Figure 3: Unfolding for Example 6.1

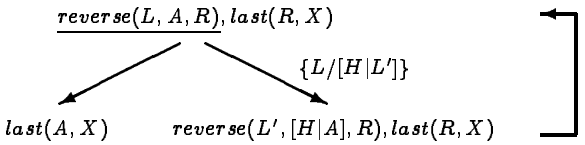


Figure 4: Unfolding of Generalisation for Example 6.1

References

[1] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10:91–124, 1991.

[2] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[3] D. A. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'91*, pages 205–220, Manchester, UK, 1991.

[4] S. Debray and D. Warren. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, 1989.

[5] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[6] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[7] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.

[8] J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.

[9] R. Glück, J. Jørgensen, B. Martens, and M. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS, Aachen, Germany, September 1996. To Appear. Extended version as Technical Report CW 226, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[10] N. Heintze. Practical aspects of set based analysis. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 765–779, Washington D.C., 1992. MIT Press.

[11] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *The Journal of Logic Programming*, 13(2 & 3):205–258, 1992.

[12] J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallager, editor, *Pre-Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, pages 46–62, Stockholm, Sweden, August 1996. Also in the Proceedings of BENELOG'96.

[13] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, Lecture Notes in Computer Science 1048, pages 1–16, Utrecht, Netherlands, September 1995. Springer-Verlag.

[14] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~lpai`, 1996.

[15] M. Leuschel and D. De Schreye. An almost perfect abstraction operation for partial deduction using characteristic trees. Technical Report CW 215, Departement Computerwetenschappen, K.U. Leuven, Belgium, October 1995. Submitted for Publication. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[16] M. Leuschel and D. De Schreye. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 253–263, La Jolla, California, June 1995. ACM Press.

[17] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press. Extended version as Technical Report CW 225, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[18] M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 495–509, Portland, USA, December 1995. MIT Press. Extended version as Technical Report CW 210, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[19] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Extended version as Technical Report CW 220, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[20] M. Leuschel and M. H. Sørensen. Redunant argument filtering of logic programs. In J. Gallager, editor, *Pre-Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, pages 63–77, Stockholm, Sweden, August 1996.

[21] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[22] J. W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity checking in stratified databases. *The Journal of Logic Programming*, 4(4):331–343, 1987.

[23] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 909–923, Seattle, 1988. IEEE, MIT Press.

[24] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.

[25] A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-structure data-flow analysis for prolog. *ACM Transactions on Programming Languages and Systems*, 16(2):205–258, 1994.

[26] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19 & 20:261–320, May 1994.

[27] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming, PLILP'91*, LNCS 528, Springer Verlag, pages 347–358, 1991.

[28] T. C. Przymusinksi. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.

[29] V. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

[30] P. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990. Preliminary version in ESOP'88, LNCS 300.

A A Ground Unification Algorithm

The following is a unification algorithm for the ground representation which does not use accumulating parameters and delays composition as well as application of substitutions as long as possible. Also for simplicity we have not added an occurs check (adding an occur check can only increase the precision of our analysis).

```
unify(T1,T2,MGU) :-
   unify(T1,empty,T2,empty,MGU).

unify(struct(F,A1),S1,struct(F,A2),S2,MGU) :-
   l_unify(A1,S1,A2,S2,MGU).
unify(var(V),S1,struct(F,A2),S2,MGU) :-
   get_binding(V,S1,VS),
   unify2(VS,S1,struct(F,A2),S2,MGU).
unify(struct(F,A1),S1,var(V),S2,MGU) :-
   get_binding(V,S2,VS),
   unify2(struct(F,A1),S1,VS,S2,MGU).
unify(var(V),S1,var(W),S2,MGU) :-
   get_binding(V,S1,VS),
   get_binding(W,S2,WS),
   unify2(VS,S1,WS,S2,MGU).

unify2(struct(F,A1),S1,struct(F,A2),S2,MGU) :-
   l_unify(A1,S1,A2,S2,MGU).
unify2(var(V),S1,struct(F,A2),S2,sub(V,struct(F,A2))).
unify2(struct(F,A1),S1,var(V),S2,sub(V,struct(F,A1))).
unify2(var(V),S1,var(V),S2,empty).
unify2(var(V),S1,var(W),S2,sub(V,var(W))) :-
   V\=W.

l_unify([],S1,[],S2,[]).
l_unify([H|T],S1,[H2|T2],S2,comp(HMGU,TMGU)) :-
   unify(H,S1,H2,S2,HMGU),
   l_unify(T,comp(S1,HMGU),T2,comp(S2,HMGU),TMGU).

apply(var(V),Sub,VS) :-
   get_binding(V,Sub,VS).
apply(struct(F,A),Sub,struct(F,AA)) :-
   l_apply(A,Sub,AA).

l_apply([],Sub,[]).
l_apply([H|T],Sub,[AH|AT]) :-
   apply(H,Sub,AH),
   l_apply(T,Sub,AT).

get_binding(V,empty,var(V)).
get_binding(V,sub(V,S),S).
get_binding(V,sub(W,S),var(V)) :-
   V \= W.
get_binding(V,comp(L,R),S) :-
   get_binding(V,L,VL),
   apply(VL,R,S).
```