Logic Program Specialisation: How To Be More Specific

Michael Leuschel[⋆] and Danny De Schreye[⋆⋆]

K.U. Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: {michael,dannyd}@cs.kuleuven.ac.be

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, 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 [14], 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 [14] 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 declarative built-ins like `\=`, `is`, ...). In that context, the following generic scheme (based on similar ones in e.g. [4, 12]) describes the basic layout of practically all algorithms for computing the sets S and $\{\tau_1, \ldots, \tau_n\}$.

Algorithm 1 (Standard Partial Deduction)

```
Initialise $i = 0$ , $S_i = \{A\}$
repeat
    for each $A_k \in S_i$, compute a finite SLD-tree $\tau_k$ for $A_k$ ;
    let $S_i' := S_i \cup \{B_l | B_l$ is an atom in a leaf of some tree $\tau_k$,
            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, and 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. $p(X) \leftarrow q(X), r(X) \qquad q(a) \leftarrow \qquad r(a) \leftarrow \qquad r(b) \leftarrow$

For a given query $\leftarrow p(X)$, one possible (although very unoptimal) outcome of the Algorithm 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.

$$\begin{tikzcd}
\tau_1\colon \leftarrow p(X) \arrow[d] & \tau_2\colon \leftarrow q(X) \arrow[d, "X/a"] & & \tau_3\colon \leftarrow r(X) \arrow[dl, "X/a"'] \arrow[dr, "X/b"] & & & \tau_2'\colon \leftarrow q(X) \arrow[dl, "X/a"'] \arrow[dr] & \\
\leftarrow q(X), r(X) & \square & \square & & \square & \square & & \leftarrow q(X)
\end{tikzcd}$$

Fig. 1. A possible outcome of Algorithm 1 for Ex. 1 and Ex. 1'

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, thereby making the call $\leftarrow r(X)$ more specific, namely $\leftarrow r(a)$. This information would then be used in the analysis of $r/1$, 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 [4]. There, a limited success-propagation, restricted to only one resolution step, is introduced and referred to as a *more specific resolution step.*

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, yielding Ex. 1'. A possible outcome of Algorithm 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 can be obtained.

In addition to pointing out further imprecision problems of the usual analysis scheme, the main contributions of the current paper are to propose a more refined analysis scheme that solves these problems and to illustrate the *applicability* of the new scheme to a *class of applications* in which they are *vital* for successful specialisation. 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 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. 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 underlying a specific specialisation technique: the one of *conjunctive partial deduction* [10, 6] and the one of *more specific programs* [15].[2]

Let us first present an abstract interpretation method based on [15] which calculates more specific versions of programs.

We first introduce the following notations. Given a set of logic formulas P, $\mathit{Pred}(P)$ denotes the set of predicates occuring in P. By $\mathit{mgu}^*(A, B)$ we denote the most general unifier of A and B', where B' is obtained from B by renaming apart wrt A. Next $\mathit{msg}(S)$ denotes the most specific generalisation of the atoms in S. We also define the predicate-wise application msg^* of the msg: $\mathit{msg}^*(S) = \{\mathit{msg}(S^p) \mid p \in \mathit{Pred}(P)\}$, 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 1. *For a definite logic program P and a set of atoms $\mathcal{A}$ we define:* $T_P(\mathcal{A}) = \{H\theta_1 \ldots \theta_n \mid H \leftarrow B_1, \ldots, B_n \in P \wedge \theta_i = \mathit{mgu}^*(B_i, A_i) \text{ with } A_i \in \mathcal{A}\}$. *We also define $U_P(\mathcal{A}) = \mathit{msg}^*(T_P(\mathcal{A}))$.*

One of the abstract interpretation methods of [15] can be seen (see also Sect. 6) as calculating $\mathit{lfp}(U_P) = U_P \uparrow^\infty (\emptyset)$. In [15] more specific versions of clauses and programs are obtained in the following way:

Definition 2. *Let $C = H \leftarrow B_1, \ldots, B_n$ be a definite clause and $\mathcal{A}$ a set of atoms. We define $\mathit{msv}_{\mathcal{A}}(C) = \{C\theta_1 \ldots \theta_n \mid \theta_i = \mathit{mgu}^*(B_i, A_i) \text{ with } A_i \in \mathcal{A}\}$. The* more specific version *$\mathit{msv}(P)$ of P is then obtained by replacing every clause $C \in P$ by $\mathit{msv}_{\mathit{lfp}(U_P)}(C)$ (note that $\mathit{msv}_{\mathit{lfp}(U_P)}(C)$ contains at most 1 clause).*

[2] These techniques are rather straightforward to integrate because they use the same abstract domain: a set of concrete atoms (or goals) is represented by the instances of an abstract atom (or goal).

In the light of the stated problems, an integration of partial deduction with the more specific program transformation seems a quite natural solution. In [15] such an integration was already suggested as a promising future direction. The following example however reveals that, in general, this combination is still too weak to deal with side-ways information propagation.

Example 2. **(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.

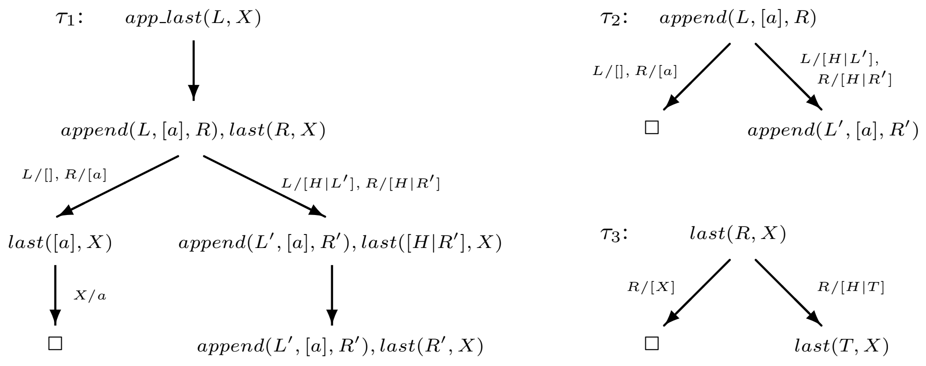

Fig. 2. SLD-trees for Ex. 2

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)` $\}$

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 *msg* 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* [8], *regular types* [5] or refined types for compile-time garbage collection of [16], 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), 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*, 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 [10] 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 leaf goal $\leftarrow B_1, \ldots, B_k$ of some SLD-tree τ_i, 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 [14].

Definition 3. (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 [14], the SLD-trees are restricted to *atomic* top-level goals. This restriction has been omitted in [10] 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 we refer to [10].

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 [17] 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 $msv(.)$ transformation *does* solve Ex. 2. Starting from the goal `app_last(X)` and using an analysis scheme similar to Algorithm 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 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')` or `last(R',X)` is 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 U_P-operator now produces the sets:

$U_P \uparrow 1 = \{$ `al([],[a],[a],a)`$\}$, $U_P \uparrow 2 = \{$ `al(X,[a],Y,a)`, `app_last(a)`$\}$, the latter being 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 ([9]).

```
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. When we later retrieve this value from the environment it is often vital to be able to recover this static value. This is very similar to the `append-last` problem of Ex. 2. So again, calculating $msv(P)$, even if we perform a magic-set transformation, does not give us any new information for a query like `store(E,k,2,E1),lookup(E1,k,Val)`. To solve this problem, one needs again to combine abstract interpretation with conjunctive partial deduction (to "deforest" [21] the intermediate environment E_1). The specialised program P' for the query `store(E,k,2,E1),lookup(E1,k,Val)` using e.g. the ECCE ([9]) conjunctive partial deduction system is the following:

```
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 be 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 later on control tests or loops.

3.2 Proving Functionality

The following is a generalisation of the standard definition of functionality (see e.g. [18] or [3]).

Definition 4. *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*

In the above definition we allow A, B to be used as atoms as well as terms (as arguments to $=/2$). Also, for simplicity of the presentation we restrict ourselves to correct answers. Therefore it can be easily seen that,[3] 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).

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

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. [18]). It can also be used to produce more efficient code (see e.g. [3]). 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:[4] `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 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 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` and `Env`.

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 next 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 can be found in [13]), 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(V,S,Bind)`

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

wrt a particular variable index V and a particular substitution S. 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 V is known to be 1. Taking the approach of the previous section we would add the definition gg(Sub,B1,B2) :- get_binding(1,Sub,B1),get_binding(1,Sub,B2) and apply conjunctive partial deduction and $msv(.)$ to obtain:

```
gg(Sub,B1,B2) :- get_binding_get_binding__1(Sub,B1,B2).
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 analysing 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,B,B)$\}$ 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 now recursively apply conjunctive partial deduction and $msv(.)$ to apply(V,X2,X3),apply(V,X2,X4) and then, again before reaching the fixpoint, to 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.[5]

We first introduce a special conjunction $\perp$ which is an instance of every conjunction, as well as the only instance of itself, and extend the msg such that $msg(S \cup \{\perp\}) = msg(S)$ and $msg(\{\perp\}) = \perp$. We also use the convention that if unification fails it returns a special substitution $fail$. Applying $fail$ to any conjunction Q in turn yields $\perp$. 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:

[5] 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).

Definition 5. (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 each 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 6. (refinement) *Let $I = \langle Q'_1, \ldots, Q'_s \rangle$ be an interpretation of a resultant tuple $RS = \langle (Q_1, R_1), \ldots, (Q_s, R_s) \rangle$ and $R = H \leftarrow Body$ be a resultant. 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 and all refinements of $R\theta$ are also called refinements of R under RS and I.*

Note that a least refinement does not always exist. Take for instance $R = q \leftarrow p(X, f(T)) \wedge p(T, X)$, $RS = \langle (p(X,Y), R_1) \rangle$ 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.[6] 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 [15], 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. 1 to work on interpretations of resultant tuples:

Definition 7. ($\mathbf{U}_{P,RS}$) *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 $M_i = msg(\{H \mid C \in R_i \wedge ref_{RS,I}(C) = H \leftarrow B\})$.*

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 [6] for some such operations).

Definition 8. (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.*

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

We need the following definition, before presenting the promised algorithm:

Definition 9. (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.*

Algorithm 2 (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[7] 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 and performs abstraction only when adding new conjunctions, the existing ones are not abstracted (it is trivial to adapt this). This is like in [12] but unlike Algorithm 1.

Algorithm 2 is powerful enough to prove e.g. functionality of `get_binding` of Sect. 3.3 *and* use it for further specialisation. A detailed execution of the algorithm, proving functionality of multiplication, can be found in [13].

Correctness of Algorithm 2 for preserving Least Herbrand Model as well as computed answers, follows from correctness of conjunctive partial deduction (see [10]) and of the more specific program versions for suitably chosen conjunctions (because [15] only allows one unfolding step, a lot of intermediate conjunctions have to be introduced) and extended for the more powerful refinements of Def 6. Termination, for a suitable abstraction operation (see [6]), 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 2 does not preserve finite failure. However, if the specialised program fails infinitely, then so does the original one (see [15]). 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 [19] for a suitable procedural semantics which is preserved.

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

5 Specialising the Ground Representation

A meta-program is a program which takes another program, the *object* program, as one of its inputs. An important issue in meta-programming is the representation of the object program. One approach is the *ground representation*, which encodes variables at the object level as ground terms. A lot of meta-programming tasks can only be written declaratively using the ground representation. This was e.g. the case for the application in [11], where a simplification procedure for integrity constraints in recursive deductive databases was written as a meta-program. The goal was to obtain a pre-compilation of the integrity checking via partial deduction of the meta-interpreter. However, contrary to what one might expect, partial deduction was then unable to perform interesting specialisation and no pre-compilation could be obtained. This problem was solved in [11] via a new implementation of the ground representation combined with a custom specialisation technique.

The crucial problem in [11] boiled down to a lack of information propagation at the object level. The ground representation entails the use of an explicit unification algorithm at the object level. For the application of [11] we were interested in deriving properties of the result `R` of calculating `unify(A,B,S),apply(H,S,R)` where `S` is a substitution (the *mgu*) and `A,B,H` are (representations of) partially known atoms. 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 `R` 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 Ex. 2. Therefore current abstract interpretation methods, as well as current partial deduction methods alone, fail to derive the desired information.

Fortunately Algorithm 2 provides an elegant and powerful solution to this problem. Some experiments, conducted with a prototype implementation of Algorithm 2 based on the ECCE system [9], are summarised in Table 1. A unification algorithm has been used, which encodes variables as `var(VarIndex)` and predicates/functors as `struct(p,Args)`. The full code can be found in [13]. All the examples were successfully solved by the prototype and the main ingredient of the success lay with proving functionality of `get_binding`.

The information propagations of Table 1 could neither be solved by regular approximations ([5]), nor by [15] alone, nor by set-based analysis ([7]) nor even by current implementations of the type graphs of [8]. In summary, Algorithm 2 also provides for a powerful abstract interpretation scheme as well as a full replacement of the custom specialisation technique in [11].[8]

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

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 1. Specialising the Ground Representation

6 Discussion and Conclusion

The approach presented in this paper can be seen as a practical realisation of a combined backwards and forwards analysis (see [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 [15] is not directly based on the T_P operator, but uses an operator on goal tuples which can handle conjunctions and which is (sometimes) 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 what conjunctive partial deduction can do).

In Algorithm 2 a conflict between efficiency and precision might arise. But Algorithm 2 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).

When using the unification algorithm from [11], instead of the one in [13] Algorithm 2 cannot yet handle all the examples of Table 1. The reason is that the substitutions in [11], in contrast to the ones in [13], are actually *accumulating* parameters, whose deforestation (see [20]) is still an open problem in general.

In conclusion, 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. 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 also thank Maurice Bruynooghe and anonymous referees for their helpful comments.

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. S. Debray and D. Warren. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, 1989.
4. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
5. J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.
6. R. Glück, J. Jørgensen, B. Martens, and M. Sørensen. Controlling conjunctive partial deduction of definite logic programs. *In this Volume*.
7. 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.
8. 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.
9. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~lpai`, 1996.
10. M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Bonn, Germany, September 1996. MIT Press. To Appear. Extended version as Technical Report CW 225, Departement Computerwetenschappen, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.
11. M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. Lloyd, editor, *Proceedings of the International Logic Programming Symposium*, pages 495–509, Portland, USA, December 1995. MIT Press.
12. 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, Schloß Dagstuhl, 1996. To Appear.
13. M. Leuschel and D. Schreye. Logic program specialisation: How to be more specific. Technical Report CW 232, Departement Computerwetenschappen, K.U. Leuven, Belgium, May 1996. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.
14. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.
15. K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990. Preliminary version in *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 909–923, Seattle, 1988. IEEE, MIT Press.
16. 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.

17. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19 & 20:261–320, May 1994.
18. 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 PLILP'91*, LNCS 528, Springer Verlag, pages 347–358, 1991.
19. T. C. Przymusinksi. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.
20. V. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.
21. P. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.