

# Controlling Conjunctive Partial Deduction of Definite Logic Programs

Robert Glück<sup>1</sup>, Jesper Jørgensen<sup>2</sup>, Bern Martens<sup>2</sup> and Morten H. Sørensen<sup>1</sup>

<sup>1</sup> Department of Computer Science, University of Copenhagen  
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark  
{glueck,rambo}@diku.dk

<sup>2</sup> Department of Computer Science, Katholieke Universiteit Leuven  
Celestijnenlaan 200A, B-3001, Heverlee, Belgium  
{jesper,bern}@cs.kuleuven.ac.be

## Abstract

“Classical” partial deduction, within the framework by Lloyd and Shepherdson, computes partial deduction for separate atoms independently. As a consequence, a number of program optimisations, known from unfold/fold transformations and supercompilation, cannot be achieved.

In this paper, we show that this restriction can be lifted through (polygenetic) specialisation of entire atom conjunctions. We present a generic algorithm for such partial deduction and discuss its correctness in an extended formal framework. We concentrate on novel control challenges specific to this “conjunctive” partial deduction. We refine the generic algorithm into a fully automatic concrete one that registers partially deduced conjunctions in a global tree, and prove its termination and correctness. We discuss some further control refinements and illustrate the operation of the concrete algorithm and/or some of its possible variants on interesting transformation examples.

## 1 Introduction

Partial deduction, one of the most well-studied transformation paradigms in logic programming, was originally introduced in the early eighties by Komorowski [13], and later put on a solid theoretical foundation by Lloyd and Shepherdson (LS) [19]. Recent surveys on partial deduction include [8,6].

However, partial deduction within the LS framework *cannot* accommodate certain important optimisations. As an example, consider the goal  $app(Xs, Ys, T), app(T, Zs, R)$ , where  $app$  is the usual append predicate:

$$\begin{aligned} app([], Ys, Ys). \\ app([H|Xs], Ys, [H|Zs]) &\leftarrow app(Xs, Ys, Zs). \end{aligned}$$

Given  $Xs, Ys, Zs$ , left-to-right execution of the goal  $app(Xs, Ys, T), app(T, Zs, R)$  first appends  $Xs$  and  $Ys$ , and appends to the result  $Zs$  through the local variable  $T$ . In other words, the predicate constructs from the lists  $Xs$  and  $Ys$  an intermediate list  $T$  which is then traversed to append  $Zs$  to it. While this goal is simple and elegant, it is rather inefficient since construction and traversal of such intermediate data structures is expensive.

The equivalent goal  $da(Xs, Ys, Zs, R)$  avoids the intermediate data structure and so is more efficient, but also less modular, and the definition of  $da$  is less obvious:

$$\begin{aligned} da([], Ys, Zs, R) &\leftarrow app(Ys, Zs, R). \\ da([X|Xs], Ys, Zs, [X|Rs]) &\leftarrow da(Xs, Ys, Zs, Rs). \\ app([], Ys, Ys). \\ app([X|Xs], Ys, [X|Zs]) &\leftarrow app(Xs, Ys, Zs). \end{aligned}$$

Through automatic program transformation, we should be allowed to program the first version, enjoying the benefits of elegance and modularity, and then automatically obtain the second version, thus reaping the benefits of efficiency too.

The reason why classical partial deduction cannot achieve the transformation of the original goal to *da* is simple: Atoms in conjunctive goals are transformed independently, whereas the above example requires merging a conjunction  $app(Xs, Ys, T)$ ,  $app(T, Zs, R)$  into one new atom  $da(Xs, Ys, Zs, R)$ . This limits partial deduction to a conservative *monogenetic*<sup>1</sup> specialisation of logic programs.

Unfold/fold transformations, similar to Burstall and Darlington’s classical work in functional programming [4], have been introduced to logic programming [26,22]. Through such unfold/fold transformations, so-called unnecessary variables *can* often be eliminated from logic programs [23], resulting in optimisations similar to *deforestation* [29] and *tupling* [5] known in functional programming.

The relationship between partial deduction, logic program specialisation and unfold/fold transformation has already been discussed in, among others, [2,24,22]. These discussions, however, focus on how specialisation of logic programs can be understood in an unfold/fold setting. In a companion paper to this one, De Schreye, Leuschel and de Waal have taken a different approach [15]: They have extended the LS framework such that the above mentioned limitations can be lifted and polygenetic “conjunctive” partial deduction is enabled.

In this paper, we address the question how such partial deduction can be actually *realised*. Indeed, the requirement that *partial deductions should be computed for conjunctions of atoms*, rather than for single, separate atoms leads to a host of novel issues concerning *control*. Below, we address these in detail. We clearly distinguish *local and global control*, discuss the relationship between the two, and specifically elaborate on the complications that arise from dealing with conjunctions at the global control level. As in [21,16], we will employ *tree structures to register dependencies at the global control level*, but, unlike in [21,16], these trees will now of course contain conjunctive goals instead of only single atoms. Finally, this also implies that we need to define *abstraction* (necessary to ensure termination) *on conjunctions of atoms*.

This leads to the following layout for the paper. Section 2 proposes a generic correct algorithm for conjunctive partial deduction. We derive a specific, fully automatic, concrete algorithm in Section 3. We extensively discuss its local and global control aspects, as well as outline some further possible variants and refinements. Finally, in Section 4, we show how the algorithm and its variants deal with several examples, and indeed substantially improve upon classical, monogenetic partial deduction. The paper ends with a discussion of related work and a brief conclusion (Sections 5 and 6).

## 2 A Generic Algorithm for Conjunctive Partial Deduction

We assume the reader is familiar with the basic concepts of logic programming and partial deduction, as presented in [17] and [19].

Throughout the rest of this paper, we presuppose some given logic language and consider only definite programs and goals in this language. Next,  $A$ ,  $A_0$ , etc. will always denote atoms,  $Q$ ,  $Q_0$ , etc. conjunctions of atoms and  $G$ ,  $G_0$ , etc. goals, i.e.  $\leftarrow Q$ , etc. Sometimes  $B$ ,  $B_0$ , etc. will also be used for conjunctions when these appear as bodies of clauses in some context. We will

---

<sup>1</sup>Mono- and polygeneticity are the dual notions of mono- and polyvariance. In a monogenetically produced program, every predicate definition derives from a single definition in the original program. Polygenetic transformation does not impose this restriction. See e.g. [11] for further details.

write  $Q \wedge Q'$  for the conjunction of two conjunctions  $Q$  and  $Q'$  (an atom is considered a special case of a conjunction). We will not distinguish between conjunctions that are identical (under associativity) modulo variable renaming, unless explicitly stated otherwise. We will use  $\equiv$  to denote syntactic identity (under associativity) while  $=$  (for conjunctions) will denote syntactic identity modulo reordering of atoms. Given a substitution  $\theta$  and a conjunction  $Q$  then  $\theta|Q$  will denote the restriction of  $\theta$  to the variables in  $Q$ .

As mentioned above, a correctness framework (modifying and extending the one in [19]) for conjunctive partial deduction is developed in [15]. For the benefit of the reader, we include (and occasionally slightly adapt) in Appendix A the main definitions and results from [15], needed as theoretical background for the algorithms below. For further details, comments and proofs, we refer to [15].

From now on, the term “partial deduction” will refer to polygenetic, conjunctive partial deduction, while familiar partial deduction within the LS framework will be designated as monogenetic or classical.

**Definition 2.1 (unfolding rule)** An *unfolding rule*  $U$  maps a program  $P$  and a goal  $G$  to a (possibly incomplete) non-trivial SLD-tree of  $G$  in  $P$ .

In fact, we will usually apply an unfolding rule not to a single goal, but to a set of goals, or even a set of conjunctions. Moreover, with further abuse of notation, we will often assume that an unfolding rule  $U$  does not return a set of SLD-trees, but the corresponding resultants, *i.e.* a pre-partial deduction.

**Definition 2.2 (abstraction operator)** An *abstraction operator*  $A$  is a function which for any finite set of conjunctions  $\mathcal{Q}$  returns a finite set of conjunctions  $A(\mathcal{Q})$  such that

1.  $Q \in A(\mathcal{Q})$  implies there exists  $Q' \in \mathcal{Q}$  such that  $Q' = Q\theta \wedge Q''$  for some  $Q''$  and  $\theta$ .
2.  $Q \in \mathcal{Q}$  implies there exist elements  $Q_1, \dots, Q_n \in A(\mathcal{Q})$  and  $\theta_1, \dots, \theta_n$  such that  $Q = Q_1\theta_1 \wedge \dots \wedge Q_n\theta_n$ .

Observe that abstracting a conjunction can involve splitting and/or generalising it.

The following basic algorithm for partial deduction is parameterised by an unfolding rule  $U$  and a family of abstraction operators  $A_{\mathcal{Q}_i}$ .

### Algorithm 2.3

**Input:** a program  $P$  and a goal  $\leftarrow Q$   
**Output:** a set of conjunctions  $\mathcal{Q}$   
**Initialisation:**  $i := 0$ ;  $\mathcal{Q}_0 := \{Q\}$   
**repeat**  $\mathcal{Q}_i := U(P, \mathcal{Q}_i)$   
     2.  $\mathcal{Q}_{i+1} := A_{\mathcal{Q}_i}(\mathcal{Q}_i \cup S^b)$   
     3.  $i := i + 1$   
**until**  $\mathcal{Q}_i = \mathcal{Q}_{i-1}$  (modulo variable renaming)  
**return**  $\mathcal{Q} = \mathcal{Q}_i$

In the special case where  $Q$  is an atom, and abstraction always splits all conjunctions into atoms, subsequently performing some generalisation on the resulting set, Algorithm 2.3 is similar to Gallagher’s Basic Algorithm [8] (restricted to definite programs).

From a program  $P$  and a goal  $\leftarrow Q$ , using some unfolding rule  $U$  and abstraction operators  $A_{\mathcal{Q}_i}$ , Algorithm 2.3 constructs a set of conjunctions  $\mathcal{Q}$ . This determines a pre-conjunctive partial deduction:

$$P_{\mathcal{Q}} = U(P, \mathcal{Q}) \cup P \setminus \{p(t_1, \dots, t_n) \leftarrow B \in P \mid p(s_1, \dots, s_n) \in \mathcal{Q}\}$$

The actual choice of  $U$  and the  $A_{Q_i}$  must be such that termination of the algorithm is ensured. Also, (correct) abstraction will guarantee the existence of a partitioning  $p$  such that, for goals  $G$  to be solved with the specialised program,  $P_Q \cup \{G\}$  is  $Q$ -covered wrt  $p$ . Then, through correct renaming, a conjunctive partial deduction of  $P$  wrt  $Q$  can be constructed which satisfies the conditions for Theorem A.15. Finally, an appropriate amount of specialisation should be obtained. Using a too cautious unfolding rule may lead to too much abstraction and therefore too little specialisation. All too eager unfolding, however, can cause code explosion, redundant work, slow specialisation and possibly non-termination. The next section addresses these *control problems* in detail. It also explains why we allow different abstraction operators in different iterations of the algorithm, rather than imposing a single one throughout.

### 3 A Concrete Algorithm

In this section, we refine the above generic algorithm for polygenetic partial deduction into a concrete one. Following [21,16] for the classical case, we first introduce a tree structure to register dependencies among conjunctions in the successive  $Q_i$ . Next, we choose specific unfolding and abstraction operators. Throughout, we will adhere to a clean conceptual separation between local and global control [8,21,16].

#### 3.1 Trees for Global Control

**Definition 3.1 (global tree)** A *global tree*  $\gamma$  is a labeled tree, where every node  $N$  is labeled with a conjunction of atoms  $Q_N$ .  $\mathcal{N}_\gamma$  will denote the set of its labels, and  $\mathcal{L}_\gamma \subseteq \mathcal{N}_\gamma$  the set of its leaf labels. For a branch  $\beta$  in  $\gamma$ ,  $\mathcal{N}_\beta$  will denote the set of conjunctions labeling  $\beta$ 's nodes, while  $\mathcal{S}_\beta$  will be the *sequence* of these labels, in the order they appear in  $\beta$ . Finally, for a leaf  $L \in \mathcal{L}_\gamma$ ,  $\beta_L$  will denote the (unique) branch containing  $L$ .

As in classical partial deduction, using global trees instead of just sets brings the ability to distinguish between unrelated goals during specialisation and thereby obtain a more specialised program. If two conjunctions in the global tree are on different branches, they will be considered unrelated, and an abstraction operator can be defined that takes this into account. This kind of precision seems to be even more crucial here than it is in a classical context (see Section 4.1).

Algorithm 2.3 is then refined as follows:

### Algorithm 3.2

**Input:** a program  $P$  and a goal  $\leftarrow Q$

**Output:** a set of conjunctions  $\mathcal{Q}$

**Initialisation:**  $i := 0$ ;  $\gamma_0 :=$  the global tree with a single node, labeled  $Q$

**repeat**  $L \in \mathcal{L}_{\gamma_i} : S_L := U(P, Q_L)$

2.  $\forall L \in \mathcal{L}_{\gamma_i}, \forall B \in S_L^b$ :

(a)  $\{Q_1, \dots, Q_n\} := A_{\beta_L}(B)$

(b)  $\forall Q_i \notin \mathcal{N}_{\beta_L}$ , add a child  $L'$  to  $L$  with label  $Q_{L'} = Q_i$

3.  $i := i + 1$

**until**  $\gamma_i = \gamma_{i-1}$

**output**  $\mathcal{N}_{\gamma_i}$

When we compare this algorithm with Algorithm 2.3, we first of all observe that each iteration no longer considers all conjunctions in “ $Q_i$ ”, but only those labeling leaves of  $\gamma_i$ . Obviously, all not yet partially deduced conjunctions in the global tree are indeed leaf labels.<sup>2</sup> Next, the abstraction operators  $A_\beta$  are applied to a single conjunction at a time, and they only take the conjunctions in the branch  $\beta$  which the new child nodes are potentially going to extend into account when abstracting the body of a new resultant. Also, a node  $N$  may be added to the global tree in spite of the fact that another node  $N'$  with a variant label already appears in it (but not as an ancestor of  $N$ ). Indeed,  $N'$  may have different ancestor labels, and if so, then the two label conjunctions, although they are variants, may be specialised in different ways.

It remains to fix specific choices for  $U$  and  $A_\beta$ , and discuss termination and correctness for the resulting concrete partial deduction algorithm.

### 3.2 Local Control

Recall that an unfolding rule  $U$  should construct, from a conjunction  $Q$  and a program  $P$ , a non-trivial SLD-tree for  $P \cup \{\leftarrow Q\}$ . The bodies of the associated resultants will, after abstraction, give rise to new conjunctions that may be added to the global tree  $\gamma$ . So the specific choice of unfolding rule will determine which new conjunctions will be considered as potential candidates for specialisation.

Determining  $U$  basically consists in defining how to extend an incomplete SLD-tree with new nodes. There exists a literature on this topic in classical partial deduction, see *e.g.* [1,3,20]. In this paper, we propose a method which is sophisticated enough to usually give good results for the kind of transformations we have in mind, and yet remains quite simple.

The following relation  $\trianglelefteq$ , adapted from [25,16] is called the (*strict*) *homeomorphic embedding* relation. As usual,  $e_1 \prec e_2$  denotes that  $e_2$  is a strict instance of  $e_1$ .

**Definition 3.3 (strict homeomorphic embedding)** Define  $\trianglelefteq$  on terms thus:

$$X \trianglelefteq Y$$

$$s \trianglelefteq f(t_1, \dots, t_n) \iff s \trianglelefteq t_i \text{ for some } i$$

$$f(s_1, \dots, s_n) \trianglelefteq f(t_1, \dots, t_n) \iff s_i \trianglelefteq t_i \text{ for all } i \text{ and } f(t_1, \dots, t_n) \not\prec f(s_1, \dots, s_n)$$

on atoms:

$$p(s_1, \dots, s_n) \trianglelefteq p(t_1, \dots, t_n) \iff s_i \trianglelefteq t_i \text{ for all } i \text{ and } p(t_1, \dots, t_n) \not\prec p(s_1, \dots, s_n)$$

---

<sup>2</sup>The reverse does not necessarily hold. The marks in [21,16] cater for this. They have been left out here to simplify the development, but can of course be incorporated in an implementation to avoid identical computation in successive iterations.

Next, we introduce a computation rule, based on  $\triangleleft$ .

**Definition 3.4 (selectable atom)** An atom  $A$  in a goal at the leaf of an SLD-tree is *selectable* if it does not descend from another selected atom  $A'$  in the SLD-tree which is embedded in  $A$ , i.e. for which  $A' \triangleleft A$  holds.

Finally, we can present our concrete unfolding rule:

**Definition 3.5 (concrete unfolding rule)** The concrete unfolding rule that we will use is the following: Unfold the left-most selectable atom in each goal of the SLD-tree under construction.

The following theorem, a variant of Kruskal's theorem [14], implies that  $U$  always constructs a finite, non-trivial SLD-tree. For a proof of the theorem, see [7].

**Theorem 3.6** *For any infinite sequence of atoms  $A_0, A_1, \dots$  there is a finite segment  $A_0, A_1, \dots, A_n$  such that  $A_i \triangleleft A_n$  for some  $i < n$ .*

**Corollary 3.7** *Let  $P$  be a program and  $G$  a goal. Let  $U$  be an unfolding rule as defined in Definition 3.5. Then  $U(P, G)$  is a finite, non-trivial SLD-tree for  $P \cup \{G\}$ .*

*Proof:* The result follows from Definitions A.10, 3.4 and 3.5, and Theorem 3.6.  $\square$

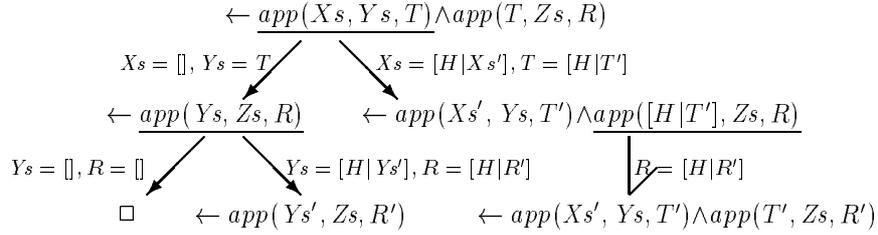


Figure 1: Unfolding the double append example

Figure 1 shows the SLD-tree obtained when using the concrete unfolding rule on the goal of the double append example presented in the introduction (selected atoms are underlined). Note that the tree is indeed non-trivial.

### 3.3 Global Control

The remaining unspecified aspects of global control reside in the abstraction operators  $A_\beta$ , deciding on which conjunctions get actually added to the global tree in order to ensure coveredness for bodies of newly derived resultants.

Obtaining coveredness is basically simple: Just add to the global tree all bodies of produced resultants as new “to be partially deduced” conjunctions. However, usually, this simple strategy does not terminate, and the need for abstraction derives from that. Essentially, for a given element  $B$  in some  $S_L^b$ , the abstraction operator  $A_{\beta_L}$  should consider whether adding  $B$  endangers termination. To this end, it should detect whether  $B$  is (in some sense) bigger than a label already in  $\mathcal{S}_{\beta_L}$ , since incorporating it might then lead to some systematic growing behaviour and thus non-termination.

However, since we are aiming at removing shared unnecessary variables from conjunctions, there is no point in keeping atoms together that do not share any variables in the first place. We will therefore always break up conjunctions into *maximal connected subparts* and abstraction will only consider these. In other words, resultant bodies will be automatically split into such connected chunks and it will actually be the latter that are considered by the abstraction operator proper.

According to Definition 2.2, abstraction consists of two components: conjunctions can be split and/or generalised. There are many ways this can be done and the concrete way will (usually) directly rely on the relation detecting growing behaviour. In this paper, we use homeomorphic embedding for both purposes. Global termination will then follow in a way similar to the local one.

In the rest of this section, to avoid some annoying technicalities, we will usually consider conjunctions as if they were terms and let the ordering of atoms matter.

**Definition 3.8 (maximal connected subconjunctions)** Given a conjunction  $Q \equiv A_1 \wedge \dots \wedge A_n$ , the collection  $\text{mcs}(Q) = \{Q_1, \dots, Q_m\}$  of *maximal connected subconjunctions* is defined through following conditions:

1.  $Q = Q_1 \wedge \dots \wedge Q_m$
2. If some variable  $X$  occurs in both  $A_i$  and  $A_j$  where  $i < j$ , then  $A_i$  occurs before  $A_j$  in the same  $Q_k$ .

**Definition 3.9 (ordered instance, generalisation)** A definite conjunction  $Q'$  is an (*ordered instance*) of another definite conjunction  $Q$ ,  $Q \leq Q'$ , iff  $Q' \equiv Q\theta$ . Given two conjunctions  $Q$  and  $Q'$ . An (*ordered generalisation*) of  $Q$  and  $Q'$  is a conjunction  $Q''$  such that  $Q'' \leq Q$  and  $Q'' \leq Q'$ . A *most specific (ordered) generalisation* of  $Q$  and  $Q'$  is an ordered generalisation  $Q''$  such that  $Q''$  is an ordered instance of every ordered generalisation of  $Q$  and  $Q'$ .

For two conjunctions  $Q \equiv A_1 \wedge \dots \wedge A_n$  and  $Q' \equiv A'_1 \wedge \dots \wedge A'_n$  where  $A_i$  and  $A'_i$  have the same predicate symbol for all  $i$ , a unique most specific generalisation  $[Q, Q']$  exists (modulo variable renaming).

We extend the definition of homeomorphic embedding to conjunctions of atoms.

**Definition 3.10 (homeomorphic embedding on conjunctions)** Define  $\triangleleft$  on conjunctions as follows :

$$Q \equiv A_1 \wedge \dots \wedge A_n \triangleleft Q' \equiv Q_1 \wedge A'_1 \wedge \dots \wedge Q_n \wedge A'_n \wedge Q_{n+1} \Leftarrow A_i \triangleleft A'_i \text{ for all } i\text{'s and } Q' \not\leq Q$$

Note that occurrences of the same variable in different atoms may be considered different.

We have the following two propositions. Their proofs are similar to the proofs of (analogous) Propositions 3.23 and 3.24 in the extended version of [16].

**Proposition 3.11** *Let  $Q_1, Q_2, Q_3$  be conjunctions such that  $Q_3 \leq Q_2$ . Then  $Q_1 \triangleleft Q_3 \Rightarrow Q_1 \triangleleft Q_2$ .*

So, a generalisation of a given conjunction will only embed conjunctions already embedded by the given one.

**Proposition 3.12** *Let  $Q_1, Q_2$  be conjunctions such that  $Q_2 \leq Q_1$ . Then  $Q_1 \triangleleft Q_2$  iff  $Q_1 \equiv Q_2$  (modulo variable renaming).*

The only remaining problem that has to be resolved before we can formally define the abstraction is how to split a maximal connected subconjunction  $Q'$  deriving from some  $B \in S_L^b$ , when it indeed embeds a goal  $Q$  on the branch  $\beta$  considered.

Assume that  $Q \equiv A_1 \wedge \dots \wedge A_n$  is embedded in  $Q'$ . Then the most obvious way to split  $Q'$  is into  $A'_1 \wedge \dots \wedge A'_n$  where  $A'_i$  embeds  $A_i$ , and  $R$  where  $R$  contains the remaining atoms of  $Q'$  in the order in which they appear in  $Q'$ . This may not be enough since  $R$  may still embed a goal in  $\mathcal{S}_{\beta_L}$ , but if we recursively repeat the splitting and (possibly) generalisation on  $R$ , we will end up with a set of conjunctions not embedding any label in  $\mathcal{S}_{\beta_L}$ .

Now, there may be several ways in which  $Q'$  embeds another conjunction and there may be several conjunctions in  $\mathcal{S}_{\beta_L}$  embedded in  $Q'$ . Tackling the latter aspect first, we will simply cut the gordian knot and abstract wrt the one closest to the leaf  $L$ . (This choice will also be instrumental in proving correctness of Algorithm 3.15 below.) Next, we will split in a way that tries to keep the best match wrt connecting variables. Consider the two conjunctions  $Q = p(X, Y) \wedge q(Y, Z)$  and  $Q' = p(X, T) \wedge p(T, Y) \wedge q(Y, Z)$ . Then  $Q'$  embeds  $Q$  and, to rectify this, we can either split  $Q'$  into  $p(X, T) \wedge q(Y, Z)$  and  $p(T, Y)$ , or into  $p(X, T)$  and  $p(T, Y) \wedge q(Y, Z)$ . Of these, the second option gives the best match and will be chosen by the splitting operation defined below. A simple method for approximating such best matches is the following:

**Definition 3.13 (best matching conjunction)** Given conjunctions  $Q, Q_1, \dots, Q_n$  all containing the same number of atoms and such that  $Q_i$  embeds  $Q$  for all  $i$ 's. A best matching conjunction  $Q_j$  is one for which  $[Q_j, Q]$  is equal to a minimally general element<sup>3</sup> in the set  $\{[Q_i, Q] \mid 1 \leq i \leq n\}$ .

**Definition 3.14 (splitting)** Given conjunctions  $Q \equiv A_1 \wedge \dots \wedge A_n$  and  $Q''$  such that  $Q \trianglelefteq Q''$ . Let  $Q'$  be the lexicographically leftmost subsequence<sup>4</sup> consisting of  $n$  atoms in  $Q''$  such that  $Q \trianglelefteq Q'$  and  $Q'$  is a best match among all subsequences  $Q^*$  consisting of  $n$  atoms in  $Q''$  such that  $Q \trianglelefteq Q^*$ . Then  $split_Q(Q'')$  is the pair  $(Q', R)$  where  $R$  is the conjunction containing the remaining atoms of  $Q''$  in the same order as they appear in  $Q''$ .

We can now make fully concrete the abstraction operators  $A_\beta$  in Algorithm 3.2.

---

<sup>3</sup>Among these, one can further select as follows. Consider graphs representing conjunctions where nodes represent occurrences of variables and there is an edge between two nodes iff they refer to occurrences of the same variable. A best match is then a  $Q_j$  which has a maximal number of edges in the graph for  $[Q_j, Q]$ .

<sup>4</sup>By the *lexicographically leftmost subsequence* of atoms in a conjunction we mean the one with the smallest tuple of position indexes when these are lexicographically ordered.

**Algorithm 3.15** For a global tree branch  $\beta$ ,  $\mathcal{S}_\beta = [B_0, \dots, B_n]$ , define the abstraction operator  $A_\beta$  as follows:

**Input:** a conjunction  $Q$

**Output:** a set of conjunctions  $\{Q_1, \dots, Q_n\}$  such that

$Q = Q_1 \theta_1 \wedge \dots \wedge Q_n \theta_n$  and  $\forall i, j : B_i \trianglelefteq Q_j \Rightarrow B_i \equiv Q_j$  (modulo variable renaming).

**Initialisation:** Let  $\mathcal{Q} = \emptyset$  and  $\mathcal{M} = \text{mcs}(Q)$ ;

**repeat** let  $M \in \mathcal{M}$ ,  $\mathcal{M} := \mathcal{M} \setminus \{M\}$ ;

2. If there exists a largest  $i$  such that  $B_i \trianglelefteq M$  and  $B_i \not\equiv M$  (modulo variable renaming), then

(a)  $(M_1, M_2) := \text{split}_{B_i}(M)$ ;

(b)  $W := [M_1, B_i]$ ;

(c)  $\mathcal{Q} := \mathcal{Q} \cup \text{mcs}(W)$ ;

(d)  $\mathcal{M} := \mathcal{M} \cup \text{mcs}(M_2)$ ;

3. Else  $\mathcal{Q} := \mathcal{Q} \cup \{M\}$ ;

**until**  $\mathcal{M} = \emptyset$ ;

**output**  $\mathcal{Q}$ ;

Note that  $A_\beta$  is indeed an abstraction operator in the sense of Definition 2.2, abstracting a singleton  $\{Q\}$ . It is this property which ensures the existence of a partitioning (and a renaming) such that the output of Algorithm 3.2 leads to a partial deduction satisfying the conditions of Theorem A.15. The issue of finding a good renaming is briefly addressed in [15]. We will not provide further details here. We do prove *termination* of Algorithm 3.2.

**Proposition 3.16** *Algorithm 3.15 terminates. A conjunction  $Q \in \mathcal{Q}$  either does not embed any  $B_i \in \mathcal{L}_\beta$ , or it is a variant of some such  $B_j$ .*

*Proof:* Upon every iteration, either a conjunction is removed from  $\mathcal{M}$ , or it is replaced by finitely many strictly smaller conjunctions, Termination follows.

For the second part of the proposition, a conjunction  $Q$  is added to  $\mathcal{Q}$  if either there is a  $B_i \equiv Q$ , or  $Q \equiv \text{mcs}([M_1, B_i])$  where  $B_i \trianglelefteq M_1$  and for no  $i < j$ ,  $B_j \trianglelefteq M_1$ . Since we also know that  $B_k \trianglelefteq B_i$  for no  $k < i$ , Proposition 3.11 implies that  $B_l \trianglelefteq Q$  for no  $l \neq i$ . Finally, Proposition 3.12 ensures that if  $B_i \trianglelefteq Q$ , then they are variants.  $\square$

So, abstraction according to Algorithm 3.15 is well defined: Its use ensures that no label in a branch of the global tree embeds an earlier label. The following theorem then is, again, a variant of Kruskal's Theorem.

**Theorem 3.17** *Algorithm 3.2 terminates if  $U$  is any terminating unfolding operator  $U$  and the  $A_\beta$ 's are defined through Algorithm 3.15.*

### 3.4 Alternative and Improved Strategies

In Sections 3.1 to 3.3, we have presented one concrete strategy for the control of conjunctive partial deduction. There are of course, as it is the case for classical partial deduction (see *e.g.* [20,16]), many variations of the ideas presented and we shall discuss some of these in this subsection. We will divide the discussion into three parts: the first part concerns alternative strategies for local control, the second alternative strategies for global control, and the third proposes techniques that involve connecting the two levels of control in some way.

For the local control, one might take most strategies from classical partial deduction and adapt them to ensure non-triviality. The simplest possible case will be a rule that simply unfolds (in some fixed order) every atom from the root goal and nothing more. This strategy

will often not be very successful when combined with the concrete abstraction used in this paper, since many conjunctions will appear at the global level too early and cause undesirable abstraction. Other strategies may involve unfolding further than our concrete strategy allows. One might also consider computation rules that incorporate determinacy or select atoms in a more intelligent manner than simply left-to-right.

Similarly, also for global control, there are many variations of the ideas presented above. One might use a different homeomorphic embedding relation on conjunctions, or some other well-quasi-ordering, or a well-founded ordering [21]. We expect that most methods used in classical partial deduction can be adapted to work for conjunctive partial deduction. The splitting strategy (Definition 3.14) used by the concrete abstraction operator is also somewhat arbitrary and may be improved. Consider the following two goals:  $G_1 \equiv p(X, T) \wedge q(T, Y)$  and  $G_2 \equiv p(X, T) \wedge r(T, R) \wedge q(R, Y)$ . Then if  $G_1$  occurs in the sequence of conjunctions used in abstracting  $G_2$ , then  $G_2$  embeds  $G_1$  and will be split into the three conjunctions:  $p(X, T)$ ,  $q(R, Y)$  and  $r(T, R)$ . This will cut the connections between the atoms and prevent any removal of redundant variables from  $G_2$ . To remedy this phenomenon, one can either try to improve the splitting operation, or aim at refining Definition 3.10. Indeed, a homeomorphic embedding relation on conjunctions taking variable sharing across atoms into account, would not give rise to such precision losses. However, in general, it would also not ensure termination.

Let us now consider *interdependent* strategies, that is, strategies that somehow mix the two levels of control by letting these be dependent on each other in a direct way. One way to do this is to extend the notion of “descendant of” (Definition A.9) to atoms in goals of the global tree, and then rephrase Definition 3.14 to first consider splits such that as many as possible of the  $A'_i$  are a descendant of the corresponding  $A_i$ . This will remove some of the arbitrariness of splitting so that not just the left-most subconjunction that embeds an earlier goal is split out, but one where the atoms are actually descendants of the atoms in the earlier goal. Refining the global control through characteristic trees as described by Leuschel and Martens [16] is also possible. Characteristic trees are abstractions of SLD-trees and in this way carriers of information about local control. Global tree nodes can be labeled with a *characteristic conjunction*, i.e. a pair of a conjunction and its characteristic tree. Abstraction then operates on characteristic conjunctions, taking the characteristic trees into account, which may lead to better specialisation as well as a more fine-grained polyvariance. Further details (in a classical setting) can be found in [16]. In Subsection 4.3 below, we discuss an example showing the usefulness of characteristic trees.

Finally, we describe an optimisation that may be applied on top of other strategies, which will in that case make these interdependent (if they not already are). There are several variants of this optimisation, but their common main objective is to make residual programs smaller without seriously damaging their quality (like *e.g.* getting less specialisation). The simplest version is the following: If a conjunction  $Q'$  at a leaf in an SLD-tree is a variant (instance) of a conjunction  $Q \in \mathcal{N}(\gamma_i)$ , then unfolding stops at that leaf. We call this refinement the *variant (instance) check rule*. Note that applying this rule may lead to different specialisation of  $Q'$ , since unfolding  $Q$  may have led to an SLD-tree, different from the subtree that can be built from  $Q'$ , and its leaves may have been abstracted in another way than those in the latter (sub)tree would. Another version applies variant (instance) checking in a post-processing phase. At the end of specialisation, it inspects the SLD-trees connected to the conjunctions in  $\mathcal{N}(\gamma)$ , and removes from them all subtrees rooted in nodes whose goal body is a variant (instance) of a conjunction in  $\mathcal{N}(\gamma)$ . This optimisation may occasionally also lead to less specialisation for essentially the same reasons as the one above. This second refinement will be named the *post variant (instance) check rule*. Of course, both techniques must be tuned to ensure non-triviality

of the SLD-trees obtained through gluing together the resulting smaller trees.

## 4 Examples

In this section, we present examples illustrating the optimisations that can be achieved through conjunctive partial deduction. We will, unless explicitly stated otherwise, use Algorithm 3.2 with the concrete strategy presented in Section 3 as well as the *variant check rule* and the *post variant check rule* described in Subsection 3.4. Some further examples can be found in [15].

### 4.1 Double Append

Initially, the global tree contains a single node labeled  $app(Xs, Ys, T), app(T, Zs, R)$ . Unfolding produces the SLD-tree shown in Figure 1. The fresh conjunctions to be considered are  $app(Ys', Zs, R')$  and  $app(Xs', Ys, T'), app(T', Zs, R')$ . The abstraction operator returns both unchanged. The second one, however, is a variant of the initial one, and therefore is not incorporated in the global tree. Since we use the post variant check rule from Subsection 3.4, we remove (safely) the subtree below  $app(Ys, Zs, R)$  in Figure 1 from the SLD-tree of the original goal. The SLD-tree of  $app(Ys', Zs, R')$  will be identical to the removed subtree (except for variable renaming). Then no more goals need to be considered, and the algorithm will terminate. From the result, one can construct the following partial deduction (for details on the renaming, see [15]):

$$\begin{array}{ll}
da([], Ys, Ys, Zs, R) & \leftarrow a(Ys, Zs, R). \\
da([H|Xs'], Ys, [H|T'], Zs, [H|R']) & \leftarrow da(Xs', Ys, T', Zs, R'). \\
a([], Ys, Ys) & \\
a([H|Xs'], Ys, [H|Zs']) & \leftarrow a(Xs', Ys, Zs').
\end{array}$$

This is almost the desired program except for the redundant third argument of  $da$ . Such redundant arguments can easily be removed by using a better renaming function, yielding the program shown in the introduction (Section 1).

This example also illustrates a point mentioned in Section 3.1: It is even more crucial to use global trees for conjunctive partial deduction than in a classical context. If we run an algorithm based on sets of conjunctions, then  $app(Xs', Ys, T'), app(T', Zs, R')$  embeds  $app(Ys, Zs, R)$  and abstraction splits  $app(Xs', Ys, T'), app(T', Zs, R')$  into two separate atoms. Consequently, no optimisation is obtained.

### 4.2 Rotate-prune

Consider the rotate-prune program, adopted from [23]:

$$\begin{array}{ll}
rotate(leaf(N), leaf(N)). & \\
rotate(tree(L, N, R), tree(L', N, R')) & \leftarrow rotate(L, L'), rotate(R, R'). \\
rotate(tree(L, N, R), tree(R', N, L')) & \leftarrow rotate(L, L'), rotate(R, R'). \\
prune(leaf(N), leaf(N)). & \\
prune(tree(L, 0, R), leaf(0)). & \\
prune(tree(L, s(N), R), tree(L', s(N), R')) & \leftarrow prune(L, L'), prune(R, R').
\end{array}$$

The predicate  $rotate(T1, T2)$  is true if the tree  $T2$  is the tree  $T1$  where the left subtree has been interchanged with the right subtree in zero or more nodes of  $T1$ , and  $prune(T1, T2)$  is true if  $T2$  is  $T1$  where each subtree of  $T1$  with label 0 has been replaced by a leaf labeled 0.

Given  $T1$ , the goal  $rotate(T1, U), prune(U, T2)$  first rotates and then prunes  $T1$  by means of an intermediate variable  $U$ . The following equivalent formulation, where  $rp(T1, T2)$  is the renaming of the original goal, arising by the technique in Section 3, avoids the intermediate data structure and so is more efficient.

$$\begin{aligned}
&rp(l(N), l(N)). \\
&rp(t(L, 0, R), l(0)) &&\leftarrow r(L), r(R). \\
&rp(t(L, s(N), R), t(L', s(N), R')) &&\leftarrow rp(L, L'), rp(R, R'). \\
&rp(t(L, s(N), R), t(R', s(N), L')) &&\leftarrow rp(L, L'), rp(R, R'). \\
&r(l(N)). \\
&r(t(L, N, R)) &&\leftarrow r(L), r(R).
\end{aligned}$$

This is equivalent to what can be obtained by unfold/fold transformations [23].

### 4.3 Remove

Our final example, adapted from [28], is particularly interesting since it sheds light on many aspects of conjunctive partial deduction. First, it is not actually a definite program, but the negation occurs in the form of an external predicate and these can be handled by a slight extension of our framework where such predicates are never touched. The predicate  $r$  in the following program succeeds if the second argument is a list obtained from the first argument by replacing each pair of identical elements by one of the elements.

$$\begin{aligned}
&r([], []). \\
&r([X], [X]). \\
&r([X, X|T], [X|T1]) \leftarrow r(T, T1). \\
&r([X, Y|T], [X|T1]) \leftarrow X \neq Y, r([Y|T], T1).
\end{aligned}$$

The goal that we will consider is  $r(X, T), r(T, Y)$ . This goal succeeds if  $Y$  is a list obtained from  $X$  by replacing all sequences of identical elements of length up to four by one of the elements. Applying our local unfolding strategy will produce the following conjunction at one of the leaves in the SLD-tree:  $r(X', [H|T']), r(T', Y')$ . This embeds the original conjunction and will therefore be split into its two constituting atoms. As a consequence, the possibility of removing the redundant variable  $T$  is lost. This shows the limitations of our concrete strategy, but fortunately they can be overcome by using an algorithm working with characteristic conjunctions, adapting the techniques presented in [16]. Indeed, the original conjunction,  $r(X, T), r(T, Y)$ , and the new one,  $r(X', [H|T']), r(T', Y')$ , have different characteristic trees (the left atom in the new conjunction does not match the first clause). Moreover, the latter characteristic tree does not embed the former, so no generalisation will take place. Unfolding continues with the conjunction  $r(X', [H|T']), r(T', Y')$ , leading to an SLD-tree which includes a leaf with conjunction  $r(X', T), r(T, Y)$  (using the variant check rule). One may show by completing the whole partial deduction of the example that this strategy yields a residual program in which the redundant variable  $T$  no longer occurs (see Appendix B).

## 5 Related Work

First, unfold/fold transformations [22] constitute a more general and more powerful transformation technique. They are based on unfold/fold rules as originally introduced in [4], and, for logic programming, [26].

Pettorossi and Proietti [22] describe a technique for classical partial deduction based on unfold/fold rules. It features *unfolding trees* which are similar to SLD-trees except that nodes are clauses and not goals, and a *u-selection rule* similar to our notion of a computation rule. Their technique does, however, rely on a quite simple folding strategy and no generalisation is involved, implying that termination is not guaranteed. Similar approaches are described in [23,24] where (in [24]) generalisation is present in the notion of “minimal foldable upper portion” of an unfolding tree.

The advantages of partial deduction over unfold/fold techniques are that partial deduction is conceptually simpler, control issues have been studied much more, and many fully automatic systems exist. So since conjunctive partial deduction is a generalisation of classical partial deduction it might be expected that many of the techniques will carry over successfully.

Next, supercompilation is a powerful transformation technique due to Turchin [27], which can also handle deforestation and tupling. It is based on unification-based information propagation and normal-order reduction. Supercompilation performs *driving* (unfolding and information propagation) and *generalisation* (a form of abstraction) [28]. Tree structures are used to record the history of configurations [9]. The connection between driving and classical partial deduction was established in [10]. See also [25,16].

Deforestation, as described in [29], can remove some intermediate data structures. It has been defined for “treeless terms”, but lacks sufficiently sophisticated local and global control to function properly in more general contexts.

Partial evaluation of functional languages [12] uses only constant propagation, as opposed to partial deduction and supercompilation. The usual evaluation strategy is applicative order which makes the specialisation monogenetic.

Finally, in classical partial deduction [19,8,6], the goals at the leaves of the SLD-trees are always cut up into atoms before being specialised further. Therefore, any information obtained by subsequently further specialising one atom in such a goal can never be used when specialising the other atoms in that same goal. Consequently, conjunctive partial deduction can have a major impact on the quality of specialisation even in cases where the objective is not elimination of unnecessary variables, but just specialisation of programs with respect to some known input. Benchmarks produced by a prototype implementation [15] show that this is in fact the case. The prototype uses determinate unfolding which for several of the considered examples leads to little or no specialisation with classical partial deduction, but to considerable specialisation with conjunctive partial deduction.

## 6 Conclusion

In this paper, we have investigated control for conjunctive partial deduction. The proposed strategies give good results on a range of examples, but clearly, further work will have to resolve a number of still open issues. We, however, feel very confident that the present paper provides a solid basis for the latter enterprise.

## Acknowledgements

We thank Danny De Schreye, André de Waal, Michael Leuschel, Torben Mogensen and Maurizio Proietti for interesting discussions on this work. Michael Leuschel also provided very valuable comments on a draft version of this paper.

Jesper Jørgensen was supported partly by the HCM Network “Logic Program Synthesis and Transformation” and partly by the Belgian GOA “Non-Standard Applications of Abstract Interpretation”. Bern Martens is a postdoctoral fellow of the K.U.Leuven Research Council. Finally, Robert Glück, Bern Martens and Morten Heine Sørensen were also supported by the DART project funded by the Danish Natural Sciences Research Council.

## References

- [1] R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.
- [2] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.
- [3] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.
- [4] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
- [5] W.-N. Chin. Towards an automated tupling strategy. In *Proceedings PEPM’93*, pages 119–132. ACM Press, 1993.
- [6] D. De Schreye, M. Leuschel, and B. Martens. Program specialisation for logic programs. Tutorial Presented at [18].
- [7] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 244–320. Elsevier, 1992.
- [8] J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings PEPM’93*, pages 88–98. ACM Press, 1993.
- [9] R. Glück and A.V. Klimov. Occam’s razor in metacomputation: The notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, and G. Rauzy, editors, *Proceedings SAS’93*, pages 112–123. Springer-Verlag, LNCS 724, 1993.
- [10] R. Glück and M.H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Proceedings PLILP’94.*, pages 165–181. Springer-Verlag, LNCS 844, 1994.
- [11] R. Glück and M.H. Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings Dagstuhl Seminar on Partial Evaluation*, Schloss Dagstuhl, Germany, February 1996. Springer-Verlag. To appear.
- [12] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
- [13] J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of prolog. In *Proceedings POPL’82*, pages 255–167. ACM Press, 1982.
- [14] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi’s conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
- [15] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. Technical Report CW225, Departement Computerwetenschappen, K.U.Leuven, Belgium, February 1996. Submitted for Publication. Accessible via <http://www.cs.kuleuven.ac.be/~lpai>.
- [16] 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 Dagstuhl Seminar on Partial Evaluation*, Schloss Dagstuhl, Germany, to appear 1996. Springer-Verlag. Extended version as Technical Report CW 220, K.U.Leuven. Accessible via <http://www.cs.kuleuven.ac.be/~lpai>.
- [17] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
- [18] J.W. Lloyd, editor. *Proceedings ILPS’95*. MIT Press, 1995.
- [19] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3-4):217–242, 1991.

- [20] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 1996. To Appear, abridged and revised version of Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, October 1993, accessible via <http://www.cs.kuleuven.ac.be/~lpai>.
- [21] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Kanagawa, Japan, June 1995. MIT Press.
- [22] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19 & 20:261–320, 1994.
- [23] M. Proietti and A. Pettorossi. Unfolding – definition – folding, in this order for avoiding unnecessary variables in logic programs. In *Proceedings PLILP'91*, pages 347–358. Springer-Verlag, LNCS 528, 1991.
- [24] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming*, 16:123–161, 1993.
- [25] M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In [18], pages 465–479.
- [26] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S-Å. Tärnlund, editor, *Proceedings ICLP'84*, pages 127–138, Uppsala, July 1984.
- [27] V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
- [28] V.F. Turchin. The algorithm of generalization in the supercompiler. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.
- [29] P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.

## A Foundations of Conjunctive Partial Deduction

**Definition A.1 (computed answer, resultant)** Let  $P$  be a definite program, and let  $\leftarrow Q_0$  be a definite goal. Let  $\leftarrow Q_0, \dots, \leftarrow Q_n$  be an SLD-derivation of  $P \cup \{\leftarrow Q_0\}$ , where  $\theta_1, \dots, \theta_n$  is the sequence of substitutions associated with the derivation. Let  $\theta = \theta_1 \circ \dots \circ \theta_n$ . Then the derivation has (*computed answer*)  $\theta|Q_0$  and (*resultant*)  $Q_0\theta \leftarrow Q_n$ .

If  $G$  is an atom, then the resultant is a definite program clause. In the LS framework, this is the key idea in obtaining new program clauses from an atomic goal and the original program. Notice that, in general, resultants are not program clauses. A set of resultants will also be called a *generalised program*.

**Definition A.2 (pre-conjunctive partial deduction)** Let  $P$  be a definite program,  $\mathcal{Q}$  a set of definite conjunctions,  $\mathcal{T}$  a set of finite, possibly incomplete, SLD-trees for  $P \cup \{\leftarrow Q\}$  for all  $Q \in \mathcal{Q}$ , and  $\mathcal{R}$  the set of associated sets of resultants. Let  $P_{\mathcal{Q}}$  be the generalised program obtained from  $P$  by removing the clauses defining predicates occurring in atomic elements of  $\mathcal{Q}$  and adding the resultants in the sets in  $\mathcal{R}$ .  $P_{\mathcal{Q}}$  is called a *pre-conjunctive partial deduction* of  $P$  wrt  $\mathcal{Q}$  (and  $\mathcal{T}$ ).

In general, when elements of  $\mathcal{Q}$  are conjunctions of several atoms, one needs *renaming* in order to get program clauses from resultants.

**Definition A.3 (pre-renaming)** A *pre-renaming*  $\sigma$  for a pre-conjunctive partial deduction  $P_{\mathcal{Q}}$  is a mapping from  $\mathcal{Q}$  to atoms such that for all  $Q \in \mathcal{Q}$ :

1.  $\text{vars}(\sigma(Q)) \subseteq \text{vars}(Q)$ .
2.  $\sigma(Q)$  has a predicate symbol which does not occur in  $P$  and is distinct from the predicate symbols of any  $\sigma(Q')$ ,  $Q' \in \mathcal{Q}$  and  $Q' \neq Q$ .

**Definition A.4 (renaming)** Let  $\sigma$  be a pre-renaming function for  $P_{\mathcal{Q}}$ . A *renaming*  $\rho$  based on  $\sigma$  is a mapping from conjunctions to atoms such that:

1. if  $Q$  is an instance of an element in  $\mathcal{Q}$ , then there exists some  $Q' \in \mathcal{Q}$  such that  $Q = Q'\theta$  and  $\rho(Q) = \sigma(Q')\theta$  for some  $\theta$ .
2. if  $Q$  is not an instance of an element in  $\mathcal{Q}$ , then  $\rho(Q) = Q$ .

If there exist elements of  $\mathcal{Q}$  with common instances, then there are several renamings associated with the same pre-renaming  $\sigma$  and the choice of a particular renaming is non-determinate.

**Definition A.5 (partitioning)** A *partitioning* of a set  $\mathcal{Q}$  of definite conjunctions is a function  $p$  such that for all  $Q \in \mathcal{Q}$ ,  $p(Q) = \{Q_1, \dots, Q_n\}$  satisfies  $Q = Q_1 \wedge \dots \wedge Q_n$ .

Note that both renaming and partitioning may treat two different conjunctions that are variants of each other in different ways.

**Definition A.6 (conjunctive partial deduction)** Let  $P$  be a definite program,  $\mathcal{Q}$  a finite set of definite conjunctions,  $P_{\mathcal{Q}}$  a pre-conjunctive partial deduction of  $P$  wrt  $\mathcal{Q}$  with associated resultant sets  $R_Q$ ,  $Q \in \mathcal{Q}$ ,  $\sigma$  a pre-renaming for  $P_{\mathcal{Q}}$ ,  $\rho$  a renaming based on  $\sigma$  and  $p$  a partitioning function. Then the *conjunctive partial deduction* of  $P$  (wrt  $\mathcal{Q}$ , under  $\sigma$ ,  $\rho$  and  $p$ ) is the program  $P_{\mathcal{Q}, \rho}$  which for each  $H \leftarrow B \in R_Q$  contains the clause:

$$\sigma(Q)\theta \leftarrow \bigwedge_{Q \in p(B)} \rho(Q)$$

where  $\theta$  is a substitution such that  $H = Q\theta$ , and for each other clause  $H \leftarrow B \in P_{\mathcal{Q}}$ , the clause:

$$H \leftarrow \bigwedge_{Q \in p(B)} \rho(Q)$$

**Definition A.7 (instance, more general)** A definite conjunction  $Q'$  is an *instance* of another definite conjunction  $Q$ ,  $Q \preceq Q'$ , iff  $Q' = Q\theta$ . We call  $Q$  *more general* than  $Q'$ .

**Definition A.8 (correct renaming)** Let  $\sigma$  be a pre-renaming for  $P_{\mathcal{Q}}$  and  $\rho$  a renaming based on  $\sigma$ . We say that  $\rho$  is a *correct* renaming (for  $P_{\mathcal{Q}}$ ) iff for every clause in  $H \leftarrow \bigwedge_{Q \in p(B)} \rho(Q)$  in  $P_{\mathcal{Q}, \rho}$ , every  $Q$  in  $p(B)$  with  $\rho(Q) = \sigma(Q')\theta$  and  $V = \text{vars}(Q) \setminus \text{vars}(\sigma(Q'))$  for some  $\theta$  and  $Q' \in \mathcal{Q}$ ,  $\theta|_V$  is a renaming substitution and the variables in  $V\theta$  are distinct from the variables in  $H$ ,  $\rho(Q)$  and  $p(B) \setminus \{Q\}$ .

**Definition A.9 (descendant of)** For the atoms in goals in an SLD-tree  $\tau$  we define a *descendant of* relation as the transitive closure of the following relation. Let  $G = \leftarrow A_1 \wedge \dots \wedge A_n$  be a goal in  $\tau$ ,  $A_m$  the selected atom in  $G$ ,  $A \leftarrow A'_1 \wedge \dots \wedge A'_k$  a clause of  $P$  such that  $\theta$  is an mgu of  $A_m$  and  $A$ . Then in  $\leftarrow (A_1 \wedge \dots \wedge A'_1 \wedge \dots \wedge A'_k \wedge \dots \wedge A_n)\theta$ , for each  $i \in \{1, \dots, k\}$ ,  $A'_i\theta$  is a descendant of  $A_m$ , and for each  $i \in \{1, \dots, m-1, m+1, \dots, n\}$ ,  $A_i\theta$  is a descendant of  $A_i$ . If  $A$  is a descendant of  $A'$ , we say that  $A$  descends from  $A'$ .

**Definition A.10 (non-trivial)** An SLD-tree is said to be *non-trivial* if for every conjunction  $Q$  in a leaf of the tree, every atom in  $Q$  is a descendant of a selected atom. A pre-conjunctive partial deduction  $P_{\mathcal{Q}}$  is called *non-trivial* if the SLD-trees in  $\mathcal{T}$  are.

**Definition A.11 (extract bodies from resultants)** For a finite set of resultants  $R$ , let  $R^b = \{B \mid Q \leftarrow B \in R\}$ .

**Definition A.12 ( $\mathcal{Q}$ -closed wrt  $p$ )** Let  $p$  be a partitioning and  $\mathcal{Q}$  a set of conjunctions. We say that a conjunction  $Q$  is  *$\mathcal{Q}$ -closed wrt  $p$*  iff every element of  $p(Q)$  is either an instance of an element of  $\mathcal{Q}$  or an atom whose predicate symbol is different from those of all atomic elements of  $\mathcal{Q}$ . A generalised program  $P$  is  *$\mathcal{Q}$ -closed wrt  $p$*  iff every element of  $P^b$  is  $\mathcal{Q}$ -closed wrt  $p$ .

Theorem A.15 below now follows from Theorem 3.10 of [15], in a way similar to what is described in Section 4.2 of [19].

We first introduce two further definitions:

**Definition A.13 ( $p$ -depends on)** Let  $P$  be a generalised program,  $G = \leftarrow Q$  a goal, and  $p$  a partitioning. We say that  $G$  *directly  $p$ -depends on* a (generalised) clause  $C$  in  $P$  if some  $Q' \in p(Q)$  unifies with the head of  $C$ . We say that  $G$   *$p$ -depends on* a (generalised) clause  $C$  in  $P$  if it either directly  $p$ -depends on  $C$  or it directly  $p$ -depends on a (generalised) clause  $C' = Q \leftarrow B$  in  $P$  and  $B$   $p$ -depends on  $C$ .

**Definition A.14 ( $\mathcal{Q}$ -covered wrt  $p$ )** Let  $P$  be a program,  $G$  a goal,  $p$  a partitioning,  $\mathcal{Q}$  a finite set of conjunctions,  $P_{\mathcal{Q}}$  a pre-conjunctive partial deduction of  $P$  wrt  $\mathcal{Q}$  and  $P^*$  the subprogram of  $P_{\mathcal{Q}}$  consisting of the (generalised) clauses on which  $G$   $p$ -depends. We say that  $P_{\mathcal{Q}} \cup \{G\}$  is  *$\mathcal{Q}$ -covered wrt  $p$*  if  $P^* \cup \{G\}$  is  $\mathcal{Q}$ -closed wrt  $p$ .

Finally, we state the following crucial correctness theorem:

**Theorem A.15** *Let  $P_{\mathcal{Q}, \rho}$  be a conjunctive partial deduction of  $P$  wrt  $\mathcal{Q}$ , under  $\rho$ ,  $\sigma$  and  $p$ . Let further  $\rho$  be correct for  $P_{\mathcal{Q}} \cup \{G\}$  and  $P_{\mathcal{Q}} \cup \{G\}$  be  $\mathcal{Q}$ -covered wrt  $p$ , then*

- $P \cup \{G\}$  has an SLD-refutation with computed answer  $\theta$ , such that  $\theta' = \theta|\rho(G)$ , iff  $P_{\mathcal{Q}, \rho} \cup \{\rho(G)\}$  has an SLD-refutation with computed answer  $\theta'$ .

*If in addition  $P_{\mathcal{Q}}$  is non-trivial, then*

- $P_{\mathcal{Q}} \cup \{G\}$  has a finitely failed SLD-tree iff  $P_{\mathcal{Q}, \rho} \cup \{\rho(G)\}$  has.

## B Residual Remove Program

This appendix contains the residual program for the remove example presented in Subsection 4.3. The program was produced by hand using a modification of our concrete algorithm incorporating characteristic conjunctions along the lines of [16]. The local control is unchanged, but the global tree was labeled with characteristic conjunctions. We have further used the *variant check rule* and the *post variant check rule* described in Subsection 3.4, as well as the post-processing described in the extended version of [16] which removes some “duplicated” polyvariance resulting from characteristic conjunctions with the same characteristic trees. This produces the following residual program:

$$\begin{aligned}
& rr([\ ], [\ ]). \\
& rr([H], [H]). \\
& rr([H, H|X'], [H]) \quad \leftarrow r'(X', [\ ]). \\
& rr([H, H|X'], [H|Y']) \quad \leftarrow rr'(X', H, Y'). \\
& rr([H, H|X'], [H|Y']) \quad \leftarrow rr''(X', I, Y'), H \neq I. \\
& rr([H, I|X'], [H]) \quad \leftarrow H \neq I, r''([I|X'], [\ ]). \\
& rr([H, I|X'], [H|Y']) \quad \leftarrow H \neq I, rr'([I|X'], H, Y'). \\
& rr([H, I|X'], [H|Y']) \quad \leftarrow H \neq I, rr''([I|X'], I', Y'), H \neq I'. \\
\\
& r'([\ ], [\ ]). \\
\\
& rr'([H], H, [\ ]). \\
& rr'([H, H|X'], H, Y) \quad \leftarrow rr(X', Y). \\
& rr'([H, I|X'], H, Y) \quad \leftarrow H \neq I, rr([I|X'], Y). \\
\\
& rr''([H], H, [H]). \\
& rr''([H, H|X'], H, [H]) \quad \leftarrow r'(X', [\ ]). \\
& rr''([H, H|X'], H, [H|Y']) \quad \leftarrow rr'(X', H, Y'). \\
& rr''([H, H|X'], H, [H|Y']) \quad \leftarrow rr''(X', I, Y'), H \neq I. \\
& rr''([H, I|X'], H, [H]) \quad \leftarrow H \neq I, r''([I|X'], [\ ]). \\
& rr''([H, I|X'], H, [H|Y']) \quad \leftarrow H \neq I, rr'([I|X'], H, Y'). \\
& rr''([H, I|X'], H, [H|Y']) \quad \leftarrow H \neq I, rr''([I|X'], I', Y'), H \neq I'.
\end{aligned}$$

This program can of course be further post-processed by unfolding the calls to  $r'$  and removing its definition, and by deleting the clauses of  $rr$  and  $rr''$  that contain a call to  $r''$  (because these will always fail).