Controlling Conjunctive Partial Deduction

Robert Glück[2], Jesper Jørgensen[1], Bern Martens[1], Morten Heine Sørensen[2]

[1] Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001, Heverlee, Belgium. {jesper,bern}@cs.kuleuven.ac.be
[2] Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark. {glueck,rambo}@diku.dk

Abstract. Partial deduction within Lloyd and Shepherdson's framework transforms different atoms of a goal independently and therefore fails to achieve a number of unfold/fold transformations. A recent framework for *conjunctive partial deduction* allows unfold/fold transformations by specialisation of entire conjunctions, but does not give an actual algorithm for conjunctive partial deduction, and in particular does not address *control* issues (e.g. how to select atoms for unfolding). Focusing on novel challenges specific to local and global control, we describe a generic algorithm for conjunctive partial deduction, refine it into a fully automatic concrete algorithm, and prove termination and correctness.

1 Introduction

Partial deduction, introduced by Komorowski [17] and formalised by Lloyd and Shepherdson [24], takes a program and a query and returns a specialised program tuned towards answering any instance of the query. Partial deduction in Lloyd and Shepherdson's framework cannot achieve certain important optimisations. For example, consider the goal $app(X, Y, T), app(T, Z, R)$ in the program

$$\begin{array}{l} app([\,], Y, Y). \\ app([H\,|\,X], Y, [H\,|\,Z]) \leftarrow app(X, Y, Z). \end{array}$$

The goal is simple and elegant, but inefficient to execute. Given X, Y, Z and assuming left-to-right execution, $app(X, Y, T)$ constructs from X and Y an intermediate list T which is then traversed to append Z to it. Construction and traversal of intermediate data structures is expensive and redundant. The equivalent goal $da(X, Y, Z, R)$ is more efficient, but its definition is less obvious:

$$\begin{array}{ll} da([\,], Y, Z, R) & \leftarrow app(Y, Z, R). \\ da([H\,|\,X], Y, Z, [H\,|\,Rs]) & \leftarrow da(X, Y, Z, Rs). \\ app([\,], Y, Y). & \\ app([H\,|\,X], Y, [H\,|\,Z]) & \leftarrow app(X, Y, Z). \end{array}$$

Partial deduction in Lloyd and Shepherdson's framework cannot transform the original goal to the more efficient one, because atoms in conjunctive goals are transformed independently, while the example requires merging a conjunction $app(X, Y, T), app(T, Z, R)$ into one new atom $da(X, Y, Z, R)$.

To overcome such limitations, De Schreye, Leuschel and de Waal [19] introduced *conjunctive partial deduction*, an extension of Lloyd and Shepherdson's framework in which entire conjunctions of atoms are specialised, combining the power of unfold-fold transformations with the virtues of partial deduction. However, they did not give a concrete algorithm for conjunctive partial deduction.

This paper aims at a basis for the design of concrete algorithms within this extended framework. We present a generic algorithm for conjunctive partial deduction and refine it into a fully automatic one. Since partial deductions are computed for conjunctions of atoms, rather than for separate atoms, *novel control challenges* specific to conjunctive partial deduction arise. Our solutions to these challenges are the main contribution of the paper.

Section 2 recapitulates the extended framework of [19]. Section 3 gives a generic correct algorithm for conjunctive partial deduction. Section 4 derives a fully automatic, concrete algorithm. Section 5 illustrates the algorithm with examples, which demonstrate that it indeed substantially improves classical partial deduction. Section 6 discusses related work. We assume the reader is familiar with the basic notions of logic programming and partial deduction [22, 24, 10].

This paper is an abridged and somewhat revised version of a technical report [11] where the interested reader may find further comments and examples.

2 Foundations of Conjunctive Partial Deduction

We briefly recall the framework for conjunctive partial deduction by De Schreye, Leuschel and de Waal [19].

We presuppose a given logic language and consider only *definite programs and goals*. A clause has the form $A \leftarrow Q$, where A, A_0, etc. denote atoms and Q, Q_0, etc. conjunctions of atoms. G, G_0, etc. denote goals of the form $\leftarrow Q$, and B, B_0, etc. conjunctions when these appear as bodies of some clauses. $Q \wedge Q'$ denotes the conjunction of Q and Q'; an atom is considered a special case of a conjunction. If Q and Q' are identical modulo variable renaming (and associativity) we write $Q \equiv Q'$. If Q and Q' are identical modulo commutativity (and associativity), we write $Q = Q'$. For a substitution θ and a conjunction Q, $\theta|Q$ is the restriction of θ to the variables in Q. Q' is an instance of Q (Q is more general than Q'), written $Q \preceq Q'$, iff $Q' = Q\theta$ for some θ.

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

For example, if $Q_{app} \equiv app(X, Y, T) \wedge app(T, Z, R)$ and P_{app} is the program in Section 1, then $P_{app} \cup \{\leftarrow Q_{app}\}$ has SLD-derivation:

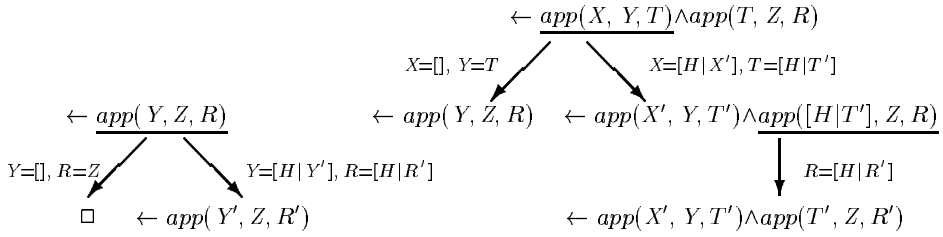
$$\begin{array}{c} \leftarrow app(X, Y, T) \wedge app(T, Z, R), \\ \leftarrow app(X', Y, T') \wedge app([H \mid T'], Z, R), \\ \leftarrow app(X', Y, T') \wedge app(T', Z, R') \end{array}$$

with computed answer $\{X \mapsto [H|X'], T \mapsto [H|T'], R \mapsto [H|R']\}$, and resultant $app([H|X'],\ Y, [H|T']) \wedge app([H|T'],\ Z, [H|R']) \leftarrow app(X',\ Y,\ T') \wedge app(T',\ Z,\ R')$.

Resultants are program clauses if the goal is atomic. In Lloyd and Shepherdson's framework, this restriction is used to obtain new program clauses from a program and an atomic goal. A *generalised program* is a set of resultants.

Definition 2. (pre-conjunctive partial deduction) Let P be a program, $\mathcal{Q}$ a set of conjunctions. For all $Q \in \mathcal{Q}$, let T_Q be a finite SLD-tree for $P \cup \{\leftarrow Q\}$, and R_Q the associated set of resultants. Let $P_\mathcal{Q}$ be the generalised program obtained from P by removing clauses defining predicates that occur in atomic elements of $\mathcal{Q}$ and adding the elements of R_Q, for all $Q \in \mathcal{Q}$. $P_\mathcal{Q}$ is a *pre-conjunctive partial deduction* of P wrt $\mathcal{Q}$ (derived from T_Q, $Q \in \mathcal{Q}$).

For instance, for P_{app} and $\mathcal{Q}_{app} = \{app(Y, Z, R), Q_{app}\}$ we have two SLD-trees (selected atoms are underlined):



The corresponding set of resultants is:

$$
\begin{array}{ll}
app([H|X'],\ Y, [H|T']) \wedge app([H|T'],\ Z, [H|R']) & \leftarrow app(X',\ Y, T') \wedge app(T',\ Z,\ R'). \\
app([],\ Y,\ Y) \wedge app(Y,\ Z,\ R) & \leftarrow app(Y,\ Z,\ R). \\
app([\,],\ Z,\ Z). & \\
app([H|Y'], Z, [H|R']) & \leftarrow app(Y',\ Z,\ R').
\end{array}
$$

This is also a pre-conjunctive partial deduction of P_{app} wrt $\mathcal{Q}$. If there were more clauses in P_{app} defining other predicates, these would also have been included.

In order to get from a pre-conjunctive partial deduction to a program, left hand sides must be renamed, so that they become atoms. This is done with *pre-renamings*. Moreover, in the right hand sides one must group atoms together by a *partitioning* into conjunctions that can be *renamed* correspondingly.

Definition 3. (pre-renaming) A *pre-renaming* σ 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. $vars(\sigma(Q)) \subseteq vars(Q)$.
2. $\sigma(Q)$'s predicate symbol occurs in neither P nor $\sigma(Q')$ for $Q' \in \mathcal{Q} \backslash \{Q\}$.

The simplest pre-renaming for $P_\mathcal{Q}$ maps every $Q \in \mathcal{Q}$ to an atom $p(X_1, \ldots, X_n)$, where p is a fresh predicate symbol and $X_1, \ldots, X_n$ are the distinct variables in Q. For example, with $P \equiv P_{app}$ and $\mathcal{Q} \equiv \mathcal{Q}_{app}$ we have as pre-renaming for $P_\mathcal{Q}$:

$$
\begin{array}{ll}
\sigma(app(X,\ Y,\ T) \wedge app(T,\ Z,\ R)) & = da(X,\ Y, T,\ Z,\ R) \\
\sigma(app(X,\ Z,\ R)) & = a(X,\ Z,\ R)
\end{array}
$$

Definition 4. (partitioning) A *partitioning* of a set $\mathcal{Q}$ of conjunctions is a map p such that for $Q \in \mathcal{Q}$, $p(Q) = \{Q_1, \ldots, Q_n\}$ and $Q = Q_1 \wedge \ldots \wedge Q_n$.

Definition 5. (renaming) Let σ be a pre-renaming function for $P_{\mathcal{Q}}$. A *renaming* ρ based on σ is a mapping from conjunctions to atoms such that:

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

If there exist elements of $\mathcal{Q}$ with common instances, then there are several renamings associated with the same pre-renaming. Both renaming and partitioning may treat conjunctions that are variants of each other in different ways.

Definition 6. (conjunctive partial deduction) Let P be a program, $\mathcal{Q}$ a set of conjunctions, $P_{\mathcal{Q}}$ a pre-conjunctive partial deduction of P wrt $\mathcal{Q}$ with associated resultant sets R_Q, $Q \in \mathcal{Q}$, σ a pre-renaming for $P_{\mathcal{Q}}$, ρ a renaming based on σ, and p a partitioning. Then the *conjunctive partial deduction of P (wrt $\mathcal{Q}$, under σ, ρ 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 θ is a substitution such that $H = Q\theta$, and for each other clause $H \leftarrow B \in P_{\mathcal{Q}}$, the clause: $H \leftarrow \bigwedge_{Q \in p(B)} \rho(Q)$.

For $P = P_{app}$ and $\mathcal{Q} = \mathcal{Q}_{app}$ we have the conjunctive partial deduction:

$$
\begin{array}{ll}
da([\,], Y, Y, Z, R) & \leftarrow a(Y, Z, R). \\
da([H|X'], Y, [H|T'], Z, [H|R']) & \leftarrow da(X', Y, T', Z, R'). \\
a([\,], Y, Y). & \\
a([H|X'], Y, [H|Z']) & \leftarrow a(X', Y, Z').
\end{array}
$$

To state the theorem guaranteeing correctness of conjunctive partial deduction, some terminology regarding closedness is required, see [19].

Definition 7. (correct renaming) Let σ be a pre-renaming for $P_{\mathcal{Q}}$ and ρ a renaming based on σ. Then ρ is a *correct* renaming (for $P_{\mathcal{Q}}$) iff for every clause $H \leftarrow \bigwedge_{Q \in p(B)} \rho(Q)$ in $P_{\mathcal{Q}_\rho}$, every Q in $p(B)$ with $\rho(Q) = \sigma(Q')\theta$ and $V = vars(Q) \backslash vars(\sigma(Q'))$ for some θ 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 $p(B) \backslash \{Q\}$.

Definition 8. (descends from) Given an SLD-tree τ. Let $G = \leftarrow A_1 \wedge \ldots \wedge A_n$ be a goal in τ, A_m the selected atom in G, $A \leftarrow A'_1 \wedge \ldots \wedge A'_k$ a clause of P such that θ is an mgu of A_m and A. Then in $\leftarrow (A_1 \wedge \ldots \wedge A'_1 \wedge \ldots \wedge A'_k \wedge \ldots \wedge A_n)\theta$, for each $i \in \{1, \ldots, k\}$, $A'_i\theta$ descends from A_m, and for each $i \in \{1, \ldots, n\} \backslash \{m\}$, $A_i\theta$ descends from A_i. If A' descends from A, and A'' descends from A', then A'' also descends from A, i.e. the relation is transitive.

Definition 9. (non-trivial) An SLD-tree is *non-trivial* if every atom in every conjunction in a leaf descends from a selected atom. A pre-conjunctive partial deduction $P_{\mathcal{Q}}$ derived from T_Q, $Q \in \mathcal{Q}$, is *non-trivial* if all T_Q are.

Definition 10. (bodies) For a set of resultants R, $R^b = \{B \mid Q \leftarrow B \in R\}$.

Definition 11. ($\mathcal{Q}$-closed wrt p) Let p be a partitioning and $\mathcal{Q}$ a set of conjunctions. 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.

Definition 12. (p-depends on) Let P be a generalised program, $G = \leftarrow Q$ a goal, and p a partitioning. G *directly p-depends on* a (generalised) clause C in P if some $Q' \in p(Q)$ unifies with the head of C. G *p-depends on* a (generalised) clause C in P if it 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 13. ($\mathcal{Q}$-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 set of clauses of $P_{\mathcal{Q}}$ on which G p-depends. $P_{\mathcal{Q}} \cup \{G\}$ is $\mathcal{Q}$-covered wrt p if $P^* \cup \{G\}$ is $\mathcal{Q}$-closed wrt p.

Theorem 14 below now follows from Theorem 3.10 of [19], in a way similar to what is described in Section 4.2 of [24].

Theorem 14. *Let $P_{\mathcal{Q}_\rho}$ be a conjunctive partial deduction of P wrt Q, under ρ, σ and p. Let ρ 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 θ, such that $\theta' = \theta|\rho(G)$,* iff *$P_{\mathcal{Q}_\rho} \cup \{\rho(G)\}$ has an SLD-refutation with computed answer θ'.*

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

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

3 A Generic Conjunctive Partial Deduction Algorithm

The framework in Section 2 provides conditions guaranteeing correctness of partial deduction, but does not give an actual algorithm. For this, *control issues* must be addressed, e.g. how to select atoms for unfolding.

We now present a generic algorithm which computes conjunctive partial deductions satisfying the conditions of Theorem 14. The algorithm uses (i) an *unfolding rule* for controlling local unfolding and (ii) an *abstraction operator* for controlling global termination, respectively.

Definition 15. (unfolding rule) An *unfolding rule* U maps a program P and a conjunction Q to the set of resultants derived from a non-trivial SLD-tree for $P \cup \{\leftarrow Q\}$. For a set of conjunctions $\mathcal{Q}$, $U(P, \mathcal{Q}) = \cup_{Q \in \mathcal{Q}} U(P, Q)$. Occasionally, $U(P, \mathcal{Q})$ will refer to the actual SLD-tree built.

Definition 16. (abstraction operator) An *abstraction operator* A maps any finite set of conjunctions $\mathcal{Q}$ to a finite set of conjunctions $A(\mathcal{Q})$ such that

1. if $Q \in A(\mathcal{Q})$, there exists $Q' \in \mathcal{Q}$ with $Q' = Q\theta \wedge Q''$ for some Q'' and θ.

2. if $Q \in \mathcal{Q}$, there exist $Q_i \in A(\mathcal{Q})$ and θ_i $(i = 1 \ldots n)$ with $Q = Q_1\theta_1 \wedge \ldots \wedge Q_n\theta_n$.

Note that abstraction of a conjunction can involve splitting and generalising.

The following basic algorithm for conjunctive partial deduction is parameterised by an unfolding rule U and an abstraction operator A.

Algorithm 1

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

until $\mathcal{Q}_i = \mathcal{Q}_{i-1}$ (modulo variable renaming)
output $\mathcal{Q}_i$

When Q is an atom, and abstraction operator A splits every conjunction into atoms, subsequently performing some generalisation on the resulting set, Algorithm 1 is essentially Gallagher's Basic Algorithm [10] restricted to definite programs.

From a program P and a goal $\leftarrow Q$, using some unfolding rule U and abstraction operator A, Algorithm 1 constructs a set of conjunctions $\mathcal{Q}$, which determines a pre-conjunctive partial deduction. From the abstraction, one can determine a partitioning p such that, for goals G to be solved with the specialised program, $P_{\mathcal{Q}} \cup \{G\}$ is $\mathcal{Q}$-covered wrt p. Then one can determine a correct renaming, and use this to construct a conjunctive partial deduction of P wrt $\mathcal{Q}$, satisfying the conditions for Theorem 14.

To obtain a concrete algorithm it remains to give an unfolding rule and an abstraction operator. These must ensure appropriate specialisation. A too cautious unfolding rule may entail too much abstraction and hence too little specialisation. Too eager unfolding, however, can cause code explosion, slow specialisation and non-termination. The next section addresses these *control problems*.

4 A Concrete Conjunctive Partial Deduction Algorithm

We now refine the above generic algorithm for conjunctive partial deduction into a concrete one. Following [26, 20] for the classical case, we introduce a *tree structure* to record dependencies among conjunctions in the successive $\mathcal{Q}_i$ and choose specific unfolding and abstraction operators. Throughout, we adhere to a conceptual *separation between local and global control* [10, 26, 20].

4.1 Trees for Global Control

Definition 17. (global tree) A *global tree* γ is a labeled tree, where every node N is labeled with a conjunction Q_N. $\mathcal{N}_\gamma$ denotes the set of its labels, and $\mathcal{L}_\gamma \subseteq \mathcal{N}_\gamma$

the set of its leaf labels. For a branch β in γ, $\mathcal{N}_\beta$ denotes the set of conjunctions labeling β's nodes, while $\mathcal{S}_\beta$ is the *sequence* of these labels, in the order they appear in β. For a leaf $L \in \gamma$, β_L denotes 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 are 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 in the classical context (c.f. Section 5.1).

Algorithm 1 is then refined as follows where each iteration no longer considers all conjunctions in "$\mathcal{Q}_i$", but only those labeling leaves of γ_i (all not yet partially deduced conjunctions in the global tree are indeed leaf labels).

Algorithm 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
  let γ_{i+1} arise from γ_i as follows:
  for all $L \in \mathcal{L}_{\gamma_i}$:
    for all $B \in (U(P, Q_L))^b$:
      let $\{Q_1, \ldots, Q_n\} := A_{\beta_L}(B) \backslash \mathcal{N}_{\beta_L}$
      and add n children to L with labels $Q_1, \ldots, Q_n$
  $i := i + 1$
until $\gamma_i = \gamma_{i-1}$
output $\mathcal{N}_{\gamma_i}$

The abstraction operators A_β are applied to a single conjunction at a time and, when abstracting the body of a new resultant, they only take the conjunctions in the branch β into account, which the new child nodes are potentially going to extend. Observe that 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, both may have different ancestor labels, and if so, then they may be specialised in different ways, although the two label conjunctions are variants.

It remains to fix specific choices for U and A_β, and discuss termination and correctness for the resulting concrete conjunctive partial deduction algorithm.

4.2 Unfolding Rule for Local Control

An unfolding rule U constructs, from a program P and a conjunction Q, the resultants of a non-trivial SLD-tree for $P \cup \{\leftarrow Q\}$. The bodies of the resultants (usually) give rise to new conjunctions that may be added to the global tree γ. So the choice of U for local control determines which new conjunctions will be considered as potential candidates for specialisation at the global level.

Determining U consists in defining how to extend an SLD-tree with new nodes. There exists an extensive literature on this topic in classical partial deduction, see e.g. [2, 4, 25]. We propose a method which is sophisticated enough to usually give good results for the kind of transformations we have in mind.

The following homeomorphic embedding relation $\trianglelefteq$ is adapted from [30, 20]. As usual, $e_1 \prec e_2$ denotes that e_2 is a strict instance of e_1.

Definition 18. (strict homeomorphic embedding) Let X, Y range over variables, f over functors, and p over predicates. Define $\trianglelefteq$ on terms and atoms:

$X \trianglelefteq Y$
$s \trianglelefteq f(t_1, \ldots, t_n) \quad \Leftarrow s \trianglelefteq t_i$ for some i
$f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n) \Leftarrow s_i \trianglelefteq t_i$ for all i
$p(s_1, \ldots, s_n) \trianglelefteq p(t_1, \ldots, t_n) \Leftarrow s_i \trianglelefteq t_i$ for all i and $p(t_1, \ldots, t_n) \not\prec p(s_1, \ldots, s_n)$

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

Definition 19. (selectable atom) An atom A in a goal at the leaf of an SLD-tree is *selectable* unless it descends from a selected atom A', with $A' \trianglelefteq A$.

Finally, we can present our concrete unfolding rule:

Definition 20. (concrete unfolding rule) Unfold the left-most selectable atom in each goal of the SLD-tree under construction. If no atom is selectable, do not unfold any atom in the goal.

The following theorem is a variant of Kruskal's theorem [18], see also [9].

Theorem 21. *For any infinite sequence $A_0, A_1, \ldots$, for some $0 \leq i < j$: $A_i \trianglelefteq A_j$.*

The following corollary follows from Definitions 9, 19 and 20, and Theorem 21.

Corollary 22. *Let P be a program, G a goal, and U the unfolding rule in Definition 20. Then $U(P, G)$ is a finite, non-trivial SLD-tree for $P \cup \{G\}$.*

4.3 Abstraction Operator for Global Control

It remains to specify the abstraction operators A_β, deciding which conjunctions are added to the global tree in order to ensure coveredness for bodies of newly derived resultants.

Ensuring coveredness is simple: add to the global tree all (unchanged) bodies of produced resultants as new, "to be partially deduced" conjunctions. However, this strategy leads usually to non-termination, and thus, the need for abstraction arises. For an element B in some $(U(P, Q_L))^b$, the abstraction operator A_{β_L} should consider whether adding B to the global level may endanger termination. To this end, A_{β_L} should detect whether B is (in some sense) bigger than another label already occurring in $\mathcal{S}_{\beta_L}$, since adding B might then lead to some systematic growing behaviour resulting in non-termination.

According to Definition 16, abstraction allows two operations: conjunctions can be split and 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.

Since we aim to remove shared, but unnecessary variables from conjunctions, there is no point in keeping atoms together that do not share variables. 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 be the latter that are considered by the abstraction operator proper.

Global termination then follows in a way similar to the local one.

Definition 23. (maximal connected subconjunctions) Given conjunction $Q \equiv A_1 \wedge \ldots \wedge A_n$, the collection $\mathrm{mcs}(Q) = \{Q_1, \ldots, Q_m\}$ of *maximal connected subconjunctions* is defined by:

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

Definition 24. (ordered instance, generalisation) A conjunction Q' is an *(ordered) instance* of another 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 $Q \equiv A_1 \wedge \ldots \wedge A_n$ and $Q' \equiv A'_1 \wedge \ldots \wedge A'_n$ where A_i and A'_i have the same predicate symbols for all i, a most specific generalisation $\lfloor Q, Q' \rfloor$ exists, which is unique modulo variable renaming.

We now extend the definition of homeomorphic embedding to conjunctions.

Definition 25. (homeomorphic embedding) Define $\trianglelefteq$ by:

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

Note that occurrences of the same variable in different atoms may be considered different. The proofs of the following two propositions are similar to the proofs of Propositions 3.23 and 3.24 in the extended version of [20].

Proposition 26. *If* $Q_3 \leq Q_2$, *then* $Q_1 \trianglelefteq Q_3 \Rightarrow Q_1 \trianglelefteq Q_2$.

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

Proposition 27. *Suppose* $Q_2 \leq Q_1$. *Then* $Q_1 \trianglelefteq Q_2$ *iff* $Q_1 \equiv Q_2$.

To complete the definition of abstraction, it remains to decide how to split a maximal connected subconjunction Q' deriving from some $B \in (U(P, Q_L))^b$, when it indeed embeds a goal Q on the branch β considered.

Assume that $Q \equiv A_1 \wedge \ldots \wedge A_n$ is embedded in Q'. An obvious way is to split Q' into $A'_1 \wedge \ldots \wedge A'_n$ and R, where A'_i embeds A_i, and R contains the remaining atoms of Q'. This may not suffice since R can still embed a goal in $\mathcal{S}_{\beta_L}$. Thus,

in order to obtain a set of conjunctions not embedding any label in $\mathcal{S}_{\beta_L}$, we *recursively repeat splitting and generalisation* on R.

There can be several conjunctions in $\mathcal{S}_{\beta_L}$ embedded in Q', and Q' can embed conjunctions in various ways. We cut the Gordian knot by abstracting wrt the node closest to leaf L. Next, we split in a way that is the *best match* wrt to connecting variables. Consider two conjunctions $Q \equiv p(X,Y) \wedge q(Y,Z)$ and $Q' \equiv p(X,T) \wedge p(T,Y) \wedge q(Y,Z)$. 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 way is the best match because it maintains the sharing of Y. A straightforward method for approximating best matches is the following.

Definition 28. (best matching conjunction) Given conjunctions $Q, Q_1, ..., Q_n$ 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 $\lfloor Q_j, Q \rfloor$ is equal to a minimally general element[3] [4] in the set $\{\lfloor Q_i, Q \rfloor | 1 \leq i \leq n\}$.

Definition 29. (splitting) Given conjunctions $Q \equiv A_1 \wedge \ldots \wedge A_n$ and Q'' such that $Q \trianglelefteq Q''$. Let Q' be the lexicographically leftmost subsequence[5] 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 now make the abstraction operators A_β fully concrete in Algorithm 2.

Algorithm 3 For a global tree branch β, $\mathcal{S}_\beta = [B_0, \ldots, B_n]$, define A_β by:

Input: a conjunction Q
Output: a set $\{Q_1, \ldots, Q_n\}$ with $Q = Q_1\theta_1 \wedge \ldots \wedge Q_n\theta_n$ and $\forall i, j : B_i \trianglelefteq Q_j \Rightarrow B_i \equiv Q_j$.
Initialisation: Let $\mathcal{Q} = \emptyset$ and $\mathcal{M} = \text{mcs}(Q)$;
repeat

1. 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 $B_i \not\equiv M$, then
   (a) $(M_1, M_2) := split_{B_i}(M)$;
   (b) $W := \lfloor M_1, B_i \rfloor$;
   (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 $\mathcal{Q}$;

[3] An element Q of a set $\mathcal{Q}$ is minimally general iff there does not exist another element Q' in $\mathcal{Q}$ such that Q' is an (strict) instance of Q.

[4] Among the minimally general elements, one can 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 with a maximal number of edges in the graph for $\lfloor Q_j, Q \rfloor$.

[5] By the *lexicographically leftmost subsequence* of atoms in a conjunction we mean the one with the smallest tuple of position indexes, ordered lexicographically.

Note that A_β is indeed an abstraction operator in the sense of Definition 16, abstracting a singleton $\{Q\}$. It is this property which ensures the existence of a partitioning (and a renaming) such that the output of Algorithm 2 leads to a conjunctive partial deduction satisfying the conditions of Theorem 14. The issue of finding a good renaming is briefly addressed in [19]. For lack of space, we cannot provide further details here. We do prove *termination* of Algorithm 2.

Proposition 30. *Algorithm 3 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, a conjunction is either removed from $\mathcal{M}$, or 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 $Q \equiv \mathrm{mcs}(\lfloor M_1, B_i \rfloor)$ where $B_i \trianglelefteq M_1$ and for no $i < j$, $B_j \trianglelefteq M_1$. Since $B_k \trianglelefteq B_i$ for no $k < i$, Proposition 26 implies that $B_l \trianglelefteq Q$ for no $l \neq i$. Finally, Proposition 27 ensures that if $B_i \trianglelefteq Q$, then they are variants. □

So, abstraction according to Algorithm 3 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 31. *Algorithm 2 terminates if U is a terminating unfolding operator and the A_β's are defined as in Algorithm 3.*

4.4 Refinements of the Algorithm

There are several ways in which the above algorithm can be refined further. Space does not allows a detailed discussion here, so we shall mention only two traditional techniques. Both techniques must be tuned to ensure non-triviality of the SLD-trees obtained by 'gluing' together the resulting smaller trees.

The simplest technique is as follows: 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 technique 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 can lead to less specialisation for essentially the same reasons as the one above. We call the second technique the *post variant (instance) check rule*.

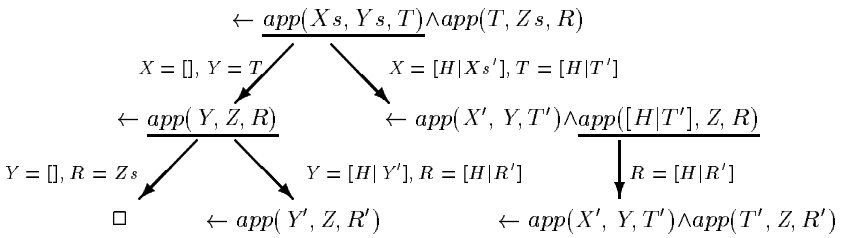
5 Examples

In this section, we present examples illustrating optimisations that can be achieved by conjunctive partial deduction. We will, unless explicitly stated otherwise, use

Algorithm 2 with the concrete strategy formulated in Section 4, as well as the *variant check rule* and the *post variant check rule* described in Subsection 4.4. Further examples can be found in [11] and [19].

5.1 Double Append

Initially, the global tree contains a single node labeled $app(X, Y, T), app(T, Z, R)$. Unfolding produces the SLD-tree shown below.



Note that the tree is indeed non-trivial. The fresh conjunctions to be considered are $app(Y', Z, R')$ and $app(X', Y, T'), app(T', Z, R')$. 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 4.4, we remove (safely) the subtree below $app(Y, Z, R)$ in the SLD-tree in Subsection 4.2. The SLD-tree of $app(Y', Z, R')$ 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 conjunctive partial deduction; for details on renaming, see [19].

$$
\begin{array}{ll}
da([\,], Y, Y, Z, R) & \leftarrow a(Y, Z, R). \\
da([H|X'], Y, [H|T'], Z, [H|R']) & \leftarrow da(X', Y, T', Z, R'). \\
a([\,], Y, Y). & \\
a([H|X'], Y, [H|Z']) & \leftarrow a(X', Y, Z').
\end{array}
$$

This is almost the desired program except for the redundant third argument of da. As shown in [21], such redundant arguments can easily be removed by using a better renaming function, yielding the program shown in Section 1.

This example also illustrates a point mentioned in Section 4.1: 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 $app(X', Y, T'), app(T', Z, R')$ embeds $app(Y, Z, R)$ and abstraction splits the conjunction $app(X', Y, T'), app(T', Z, R')$ into two separate atoms. Consequently, no optimisation is obtained.

5.2 Rotate-Prune

Consider the rotate-prune program, adopted from [28]:

$$
\begin{array}{l}
rotate(leaf(N), leaf(N)). \\
rotate(tree(L, N, R), tree(L', N, R')) \leftarrow rotate(L, L'), rotate(R, R'). \\
rotate(tree(L, N, R), tree(R', N, L')) \leftarrow rotate(L, L'), rotate(R, R').
\end{array}
$$

$prune(leaf(N), leaf(N))$.
$prune(tree(L, 0, R), leaf(0))$.
$prune(tree(L, s(N), R), tree(L', s(N), R')) \leftarrow prune(L, L'), prune(R, R')$.

The goal $rotate(T1, T2)$ is true if the tree $T2$ arises by interchanging the left and right subtree in zero or more nodes of $T1$, and $prune(T1, T2)$ is true if $T2$ arises be replacing each subtree of $T1$ with label 0 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 goal $rp(T1, T2)$ arising by the technique in Section 4, avoids the intermediate data structure and so is more efficient. This is equivalent to what unfold/fold transformations can do [28].

$rp(l(N), l(N))$.
$rp(t(L, 0, R), l(0)) \leftarrow r(L), r(R)$.
$rp(t(L, s(N), R), t(L', s(N), R')) \leftarrow rp(L, L'), rp(R, R')$.
$rp(t(L, s(N), R), t(R', s(N), L')) \leftarrow rp(L, L'), rp(R, R')$.
$r(l(N))$.
$r(t(L, N, R)) \leftarrow r(L), r(R)$.

6 Related Work

Burstall and Darlington introduced unfold/fold transformations in functional programming [5]; *deforestation* [34] and *tupling* [6] are two automatic instances of the technique. Similar transformations have been introduced in logic programming [31, 27] for removing redundant variables from logic programs [28].

The relationship between partial deduction and unfold/fold transformation has already been discussed [3, 29, 27] but with an emphasis on how specialisation of logic programs can be understood in an unfold/fold setting. Pettorossi and Proietti [27] describe a technique for classical partial deduction based on unfold/fold rules. Their technique relies on a simple folding strategy involving no generalisation, so termination is not guaranteed. Similar approaches are described in [28, 29]; in [29] generalisation is present in the notion of "minimal foldable upper portion" of an unfolding tree.

Turchin's supercompiler [32, 14] can also do deforestation. Supercompilation performs *driving* (normal-order unfolding and unification-based information propagation) and *generalisation* (a form of abstraction) [33, 30]. Tree structures are used to record the history of configurations [12]. The connection between driving and classical partial deduction was established in [13].

Recently, a transformation scheme has been proposed for functional-logic languages based on an automatic unfolding algorithm that builds narrowing trees [1]. A generic algorithm is provided that does not depend on the eager or lazy nature of the narrowing to be defined. Effects similar to driving and partial deduction can be achieved due to the unification-based mechanism of narrowing.

Finally, in classical partial deduction [24, 10, 8], 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. The experiments reported in [16] confirm this conjecture. In particular, determinate unfolding often leads to little or no specialisation with classical partial deduction, but performs much better with conjunctive partial deduction.

Acknowledgements. Special thanks to Danny De Schreye, André de Waal, Michael Leuschel, Torben Mogensen and Maurizio Proietti for discussions on this work. Michael Leuschel provided valuable feedback on an earlier version of this paper.

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

References

1. M. Alpuente, M. Falaschi, G. Vidal. Narrowing-driven partial evaluation of functional logic programs. *ESOP'96.* LNCS 1058, 45–61, Springer-Verlag, 1996.
2. R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.
3. A. Bossi, N. Cocco, S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.
4. M. Bruynooghe, D. De Schreye, B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.
5. R.M. Burstall, J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
6. W.-N. Chin. Towards an automated tupling strategy. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* 119–132. ACM Press, 1993.
7. O. Danvy, R. Glück, P. Thiemann (eds.). *Partial Evaluation*, LNCS 1110, Springer-Verlag, 1996.
8. D. De Schreye, M. Leuschel, B. Martens. Program specialisation for logic programs. Tutorial presented at [23].
9. N. Dershowitz, J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, 244–320, Elsevier, 1992.
10. J. Gallagher. Tutorial on specialisation of logic programs. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 88–98. ACM Press, 1993.
11. R. Glück, J. Jørgensen, B. Martens, and M. Sørensen. Controlling conjunctive partial deduction of definite logic programs. Technical Report CW 226, Departement Computerwetenschappen, K.U. Leuven, Belgium, February 1996.

12. R. Glück, A.V. Klimov. Occam's razor in metacomputation: The notion of a perfect process tree. In P. Cousot, et al. (eds.), *Static Analysis.* LNCS 724, 112–123, Springer-Verlag, 1993.
13. R. Glück, M.H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam (eds.), *Programming Language Implementation and Logic Programming.* LNCS 844, 165–181, Springer-Verlag, 1994.
14. R. Glück, M.H. Sørensen. A roadmap to metacomputation by supercompilation. In [7].
15. N.D. Jones, C.K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
16. J. Jørgensen, M. Leuschel, B. Martens. Conjunctive partial deduction in practice. Logic Program Synthesis and Transformation 1996. To appear.
17. 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 *Symposium on Principles of Programming Languages*, 255–167. ACM Press, 1982.
18. J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
19. M. Leuschel, D. De Schreye, A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration, JICSLP'96, 1996. To appear.
20. M. Leuschel, B. Martens. Global control for partial deduction through characteristic atoms and global trees. In [7].
21. M. Leuschel, M.H. Sørensen. Redundant argument filtering of logic programs, 1996. Logic Program Synthesis and Transformation 1996. To appear.
22. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
23. J.W. Lloyd (ed.). *Logic Programming: Proceedings of the 1995 International Symposium.* MIT Press, 1995.
24. J.W. Lloyd, J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3-4):217–242, 1991.
25. B. Martens, D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 1996. 28(2):89–146, 1996.
26. B. Martens, J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling (ed.), *International Conference on Logic Programming.* 597–611, MIT Press, 1995.
27. A. Pettorossi, M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19 & 20:261–320, 1994.
28. M. Proietti, A. Pettorossi. Unfolding – definition – folding, in this order for avoiding unnecessary variables in logic programs. In *Programming Language Implementation and Logic Programming.* LNCS 528, 347–358, Springer-Verlag, 1991.
29. M. Proietti, 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.
30. M.H. Sørensen, R. Glück. An algorithm of generalization in positive supercompilation. In [23], 465–479.
31. H. Tamaki, T. Sato. Unfold/fold transformation of logic programs. In S-Å. Tärnlund (ed.), *International Conference on Logic Programming.* 127–138, 1984.
32. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
33. V.F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, N.D. Jones (eds.), *Partial Evaluation and Mixed Computation.* 531–549. North-Holland, 1988.
34. P. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.