CONJUNCTIVE PARTIAL DEDUCTION: FOUNDATIONS, CONTROL, ALGORITHMS, AND EXPERIMENTS

DANNY DE SCHREYE, ROBERT GLÜCK, JESPER JØRGENSEN, MICHAEL LEUSCHEL, BERN MARTENS, AND MORTEN HEINE SØRENSEN

▷ Partial deduction in the Lloyd-Shepherdson framework cannot achieve certain optimisations which are possible by unfold/fold transformations. We introduce *conjunctive partial deduction*, an extension of partial deduction accommodating such optimisations, e.g., tupling and deforestation.

We first present a *framework for conjunctive partial deduction,* extending the Lloyd-Shepherdson framework by considering conjunctions of atoms (instead of individual atoms) for specialisation and renaming. Correctness results are given for the framework with respect to computed answer semantics, least Herbrand model semantics, and finite failure semantics.

Maintaining the well-known distinction between local and global control, we describe a *basic algorithm* for conjunctive partial deduction, and refine it into a *concrete algorithm* for which we prove termination. The problem of finding suitable renamings which *remove redundant arguments* turns out to be important, so we give an independent technique for this.

A fully automatic *implementation* has been undertaken, which always terminates. Differences between the abstract semantics and Prolog's left-to-right execution motivate deviations from the abstract technique in the actual implementation, which we discuss. The implementation has been tested on an extensive set of *benchmarks* which demonstrate that conjunctive partial deduction indeed pays off, surpassing conventional partial deduction on a range of small to medium-size programs, while remaining manageable in an automatic and terminating system.

Address correspondence to D. De Schreye, M. Leuschel, B. Martens, Dept. of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001, Heverlee, Belgium, email:`dannyd@cs.kuleuven.ac.be` `mal@ecs.soton.ac.uk` `bern@cs.kuleuven.ac.be`; R. Glück, M.H. Sørensen, Dept. of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, email:{`glueck,rambo`}`@diku.dk`; J. Jørgensen, Dept. of Mathematics and Physics, Royal Veterinary and Agricultural University, Thorvaldsensvej 40, DK-1871 Frederiksberg C, Denmark, email:`jesper@dina.kvl.dk`.

THE JOURNAL OF LOGIC PROGRAMMING

© Elsevier Science Inc., 1994

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

Keywords: Program Specialisation and Transformation, Partial Evaluation, Partial Deduction, Unfold/fold, Supercompilation, Logic Programming, Well-quasi orders. ◁

1. INTRODUCTION

Two approaches to transformation of logic programs have received considerable attention over the last few decades: the *unfold/fold* approach and *partial deduction.* Unfold/fold transformations have been studied by Tamaki and Sato, Pettorossi and Proietti, and others [74, 62], and was originally introduced by Burstall and Darlington [12] in functional programming. Partial deduction was developed by Komorowski [33] and formalised by Lloyd and Shepherdson [55]. Partial deduction is also referred to, in slightly different contexts, as partial evaluation or program specialisation [20, 29, 14], and was also introduced in functional programming first.

The relation between these two streams of work has been a matter of research, discussion, and controversy over the years. It has already been studied by several authors [5, 65, 62, 70], but with an emphasis on how specialisation of logic programs can be understood in an unfold/fold setting. Pettorossi and Proietti [62] describe a technique for 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 [64, 65] (in [65] generalisation is present in the notion of "minimal foldable upper portion" of an unfolding tree).

In the context of definite logic programs, partial deduction is a strict subset of the unfold/fold transformation. In essence, partial deduction refers to the class of unfold/fold transformations in which "unfolding" is the only basic[1] transformation rule; other rules, such as "definition", "lemma application" or "goal replacement", are not supported.

Most partial deduction methods make use of renaming transformations. Again, renaming is closely related to unfold/fold. Roughly stated, renaming can be formalised as a two-step unfold/fold transformation involving a "definition" step (the new predicate is defined to have the truth-value of the old one), immediately followed by a number of folding steps (appropriate occurrences of the old predicate are replaced by the corresponding new one).

In spite of these connections, there are still important differences between the unfold/fold and partial deduction methods. One is that there is a large class of transformations which are achievable through unfold/fold, but not through partial deduction. Typical instances of this class are transformations that eliminate "redundant variables" (see [62, 64]). For example, consider the following program for appending two lists.

$$app([\,], Ys, Ys).$$
$$app([H|Xs], Ys, [H|Zs]) \leftarrow app(Xs, Ys, Zs).$$

One way to append *three* lists is to use the goal $app(Xs, Ys, T), app(T, Zs, R)$, which is simple and elegant, but inefficient to execute. Given Xs, Ys, Zs and assuming left-to-right execution, $app(Xs, Ys, T)$ constructs from Xs and Ys an intermediate list

[1] One might argue that a (weak) implicit folding step is used by partial deduction, which becomes explicit when one wants to reuse e.g. the unfold/fold correctness results.

T which is then traversed to append Zs to it. Construction and traversal of such intermediate data structures is expensive. In the following less obvious program, the goal $da(Xs, Ys, Zs, R)$ appends three lists more efficiently.

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

Partial deduction techniques within the framework of Lloyd and Shepherdson [55] cannot substantially improve the conjunction $app(Xs, Ys, T), app(T, Zs, R)$ because they transform the two atoms independently. The transformation requires merging the conjunction $app(Xs, Ys, T), app(T, Zs, R)$ into one new atom $da(Xs, Ys, Zs, R)$. For some other illuminating discussions, concerning limitations of partial deduction, we refer to [61, 76, 36].

On the other hand, partial deduction has advantages over unfold/fold as well. Due to its more limited applicability, and its resulting lower complexity, the transformation can be more effectively and easily controlled. For instance, while the unfold/fold approach allows an *arbitrary mixture* of transformation rules which can make use of *any* prior program within the transformation sequence, partial deduction performs operations in a more tractable manner, never making use of the intermediate programs (leading to a much lower space and time complexity).

Furthermore, control issues have obtained considerable attention in partial deduction research, and, in the current state-of-the-art, have obtained a level of refinement which goes beyond mere heuristic strategies, as we find in unfold/fold. Indeed, formal frameworks have been developed, analysing issues of termination and of code- and search-explosion, and efficiency gains have been obtained [11, 58, 23, 24, 40, 50]. Several fully automated systems (SP [22], SAGE [27], PADDY[63], MIXTUS [67], ECCE [40, 50, 51, 53]) as well as semi-automated ones (LOGIMIX [60], LEUPEL [39, 46], COGEN [30]) have been developed and successfully applied to at least medium-size applications [46, 49, 18, 37]. A similar development of automated techniques and systems has not been undertaken in the context of unfold/fold transformations.

The aim of this paper is to bring the advantages of these two approaches to program transformation together. We therefore develop an extension of conventional partial deduction, called *conjunctive partial deduction*. The close relationship with partial deduction implies that some of the well-studied automated techniques from partial deduction can be extended to the new setting, although new problems do arise. At the same time, the new setting accommodates powerful unfold/fold transformations. Thus, we combine the benefits of automated techniques known from partial deduction with the power of unfold/fold transformations.

As the name suggests, conjunctive partial deduction does not automatically split up goals into constituting atoms, but attempts to specialise the program with respect to entire conjunctions of atoms. Sometimes, splitting a goal into subparts is still necessary to guarantee termination, but, in general, it is avoided when possible. The technique approaches more closely techniques for the specialisation and transformation of functional programs, such as deforestation [77], and supercompilation [75, 72]. Especially the latter constituted, together with unfold/fold transformations, a source of inspiration for the conception and design of conjunctive partial deduction.

The present paper offers an up to date, comprehensive, and uniform presentation of conjunctive partial deduction as developed in [48, 26, 52, 31]. We proceed in a top-down manner presenting conjunctive partial deduction from theory to practice. First, foundation and correctness results are given, then control and algorithmic topics are discussed in depth, and finally, implementation issues and benchmark results round the presentation off. In detail, the paper is organised as follows.

1. **Framework and correctness.** We present a framework for conjunctive partial deduction, analogous to the Lloyd-Shepherdson framework for conventional partial deduction, and give correctness results with respect to computed answer semantics, least Herbrand model semantics, and finite failure semantics.

   The most important aspect of the extension is that we consider conjunctions of atoms, instead of individual atoms, for specialisation and renaming. This provides a setting that—based on our current empirical evaluation—seems powerful enough to achieve the results of most unfold/fold transformations involving unfolding, folding and definition only. We also present improved correctness conditions, which, in contrast to current results for unfold/fold, allow the preservation of finite failure while not imposing certain potentially disastrous (non-determinate) unfolding steps.

2. **Basic and concrete algorithms.** We develop a basis for the design of concrete algorithms within our extended framework. Since partial deductions are computed for conjunctions of atoms, rather than for separate atoms, novel control challenges specific to conjunctive partial deduction arise.

   We present a basic algorithm for conjunctive partial deduction and refine it into a fully automatic one and prove termination; correctness follows from the framework. The algorithm uses an unfolding rule for controlling local unfolding and an abstraction operator for controlling global termination.

3. **Redundant argument filtering.** Automatically generated programs often contain redundant parts, and the above mentioned concrete algorithm for conjunctive partial deduction, even though it performs argument filtering and redundant clause removal, fails to remove all redundant arguments. This means that, in contrast to many unfold/fold techniques, the above method cannot get rid of *all* the overhead of unnecessary variables.

   To remedy this problem, we formalise the notion of a redundant argument and give a safe, effective approximation for redundant argument filtering.

4. **Implementation and benchmarks.** After having elaborated foundations and algorithms, we endeavour to put conjunctive partial deduction on trial. We report on extensive experiments with an implementation of our algorithms, describe various concrete control options used, look at abstraction in a practical Prolog context, and discuss an extensive set of benchmark results.

5. **Related work and overall conclusions.** After the presentation of foundations, algorithms and results, we thoroughly discuss the relationship between conjunctive partial deduction and some of the most well-known unfold/fold transformation techniques, as well as point out our main achievements.

1.1. Preliminaries

We assume familiarity with basic notions in logic programming, e.g. as presented in [54]. Throughout the paper, we consider *definite programs and goals*, except when explicitly stated otherwise.

A Horn 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', where $\wedge$ is assumed associative throughout the paper. An atom is considered a special case of a conjunction. Q' is an instance of Q (Q is more general than Q'), written $Q \preceq Q'$, iff $Q' = Q\theta$ for some θ. Similarly, $Q \prec Q'$ denotes that Q' is a strict instance of Q and $Q \equiv Q'$ denotes that Q is a variant of Q'.

2. FOUNDATIONS

In this section we provide extensions of the basic definitions in the Lloyd-Shepherdson framework and of renaming transformations. We also illustrate how these extensions are sufficient to support the transformations referred to in the introduction.

2.1. Resultants

Let us first recapitulate essential notions from conventional partial deduction. As usual in partial deduction, we assume that the standard notions of SLD-trees and SLD-derivations are generalised [55] to allow them to be incomplete: at any point we may decide not to select any atom and terminate a derivation. Within an SLD-tree, leaves of this kind will be called *dangling* [58]. Also, we will call an SLD-tree *trivial* iff its root is a dangling leaf. The following basic notion is adapted from [55] and associates a first-order formula with a finite SLD-derivation.

Definition 2.1. Let P be a program, $\leftarrow Q$ a goal and D a finite SLD-derivation for $P \cup \{\leftarrow Q\}$ with computed answer θ and leaf goal $\leftarrow B$. Then the formula $Q\theta \leftarrow B$ is called the *resultant* of D.

This concept can be extended to SLD-trees in the following way:

Definition 2.2. Let P be a program, G a goal and let τ be a finite SLD-tree for $P \cup \{G\}$. Let $D_1, \ldots, D_n$ be the non-failing SLD-derivations associated with the branches of τ. Then the set of resultants $resultants(\tau)$ is the union of the resultants of $D_1, \ldots, D_n$. We also define the set of bodies, $bodies(\tau)$, to be the conjunctions Q_i of the leaf goals $\leftarrow Q_i$ of $D_1, \ldots, D_n$.

Note that, in general, resultants are not clauses: their left-hand side may contain a conjunction of atoms. In the partial deduction notion introduced in [55] (there referred to as *partial evaluation*), the SLD-trees and resultants are restricted to *atomic* top-level goals. This restriction ensured that the resultants are indeed clauses. We omit this restriction here and define partial deduction of conjunctions.

Definition 2.3. Let P be a program and Q a conjunction. Let τ be a finite, non-trivial and possibly incomplete SLD-tree for $P \cup \{\leftarrow Q\}$. The set of resultants $resultants(\tau)$ is called a *conjunctive partial deduction of Q in P (via τ).*

Example 2.4. Let P be the following program.

(C_1) $max_length(X, M, L) \leftarrow max(X, M) \wedge length(X, L)$
(C_2) $max(X, M) \leftarrow max1(X, 0, M)$
(C_3) $max1([\,], M, M) \leftarrow$
(C_4) $max1([H|T], N, M) \leftarrow H \leq N \wedge max1(T, N, M)$
(C_5) $max1([H|T], N, M) \leftarrow H > N \wedge max1(T, H, M)$
(C_6) $length([\,], 0) \leftarrow$
(C_7) $length([H|T], L) \leftarrow length(T, K) \wedge L\ is\ K + 1$

Let $\mathcal{Q} = \{max_length(X, M, L), max1(X, N, M) \wedge length(X, L)\}$. Consider the finite SLD-trees τ_1, τ_2 depicted in Figure 1 (where arcs have been labelled with clause numbers used for the derivation steps) for the elements of $\mathcal{Q}$. The associated conjunctive partial deductions are then $resultants(\tau_1) = \{R_{1,1}\}$ and $resultants(\tau_2) = \{R_{2,1}, R_{2,2}, R_{2,3}\}$ respectively, where the individual resultants are as follows:

$(R_{1,1})$ $max_length(X, M, L) \leftarrow max1(X, 0, M) \wedge length(X, L)$
$(R_{2,1})$ $max1([\,], N, N) \wedge length([\,], 0) \leftarrow$
$(R_{2,2})$ $max1([H|T], N, M) \wedge length([H|T], L) \leftarrow$
$H \leq N \wedge max1(T, N, M) \wedge length(T, K) \wedge L\ is\ K + 1$
$(R_{2,3})$ $max1([H|T], N, M) \wedge length([H|T], L) \leftarrow$
$H > N \wedge max1(T, H, M) \wedge length(T, K) \wedge L\ is\ K + 1$

If we take the union of the conjunctive partial deductions of the elements of $\mathcal{Q}$ we obtain the set of resultants $P_{\mathcal{Q}} = \{R_{1,1}, R_{2,1}, R_{2,2}, R_{2,3}\}$. Clearly $P_{\mathcal{Q}}$ is not a Horn clause program. Apart from that $P_{\mathcal{Q}}$ has the desired tupling structure (except that the variable X still has multiple occurrences). The two functionalities ($max/3$ and $length/2$) in the original program have been merged into single traversals.

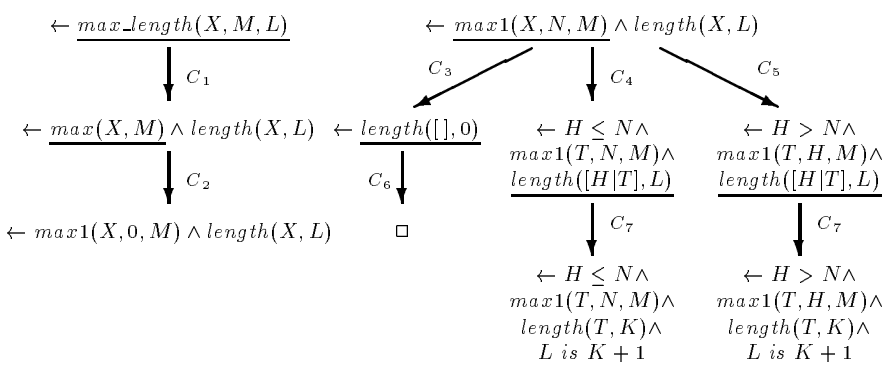


FIGURE 1. SLD-trees τ_1 and τ_2 for Example 2.4

2.2. Partitioning and Renaming

In order to convert resultants into a standard logic program, we will rename conjunctions of atoms by new atoms. Such renamings require some care. For one thing, given a set of resultants $P_{\mathcal{Q}}$, obtained by taking the conjunctive partial deduction of the elements of a set $\mathcal{Q}$, there may be ambiguity concerning which conjunctions in the bodies to rename. For instance, if $P_{\mathcal{Q}}$ contains the clause $p(X,Y) \leftarrow r(X) \wedge q(Y) \wedge r(Z)$ and $\mathcal{Q}$ contains $r(U) \wedge q(V)$, then either the first two, or the last two atoms in the body of this clause are candidates for renaming. To formally fix such choices, we introduce the notion of a *partitioning function*.

To this end we introduce a bit of terminology. If A is a set, $\mathcal{M}(A)$ denotes all multisets composed of elements of A and $=_r$ denotes syntactic identity, up to reordering, on conjunctions. If M is a multiset then we also use notations like $\wedge_{Q \in M} Q$ to denote some conjunction constructed from the elements in M, taking their multiplicity into account. Of course, this notation is defined only up to reordering. For instance, for the multiset $M = \{p, p, q\}$, $\wedge_{Q \in M} Q$ refers to either of the conjunctions $p \wedge p \wedge q$, $p \wedge q \wedge p$, or $q \wedge p \wedge p$.

Definition 2.5. Let $\mathcal{C}$ denote the set of all conjunctions of atoms over the given alphabet. A *partitioning function* is a mapping $p : \mathcal{C} \rightarrow \mathcal{M}(\mathcal{C})$, such that for any $C \in \mathcal{C}$: $C =_r \wedge_{Q \in p(C)} Q$.

For the *max_length* example, let p be the partitioning function which maps any conjunction $C =_r max1(X,N,M) \wedge length(X,L) \wedge B_1 \wedge ... \wedge B_n$ to $\{max1(X,N,M) \wedge length(X,L), B_1, ..., B_n\}$, where $B_1, ..., B_n$, $n \geq 0$, are atoms with predicates different from $max1$ and $length$. We leave p undefined on other conjunctions.

Note that the multiplicity of literals is relevant for the computed answer semantics[2] and we therefore have to use multisets, instead of just simple sets, for full generality in Definition 2.5 above.

Even with a fixed partitioning function, a range of different renaming functions could be introduced, all fulfilling the purpose of converting conjunctions into atoms (and therefore, resultants into Horn clauses). The differences are related to potentially added functionalities of these renamings, such as:

- elimination of multiply occurring variables (e.g. $p(X,X) \mapsto p'(X)$),
- elimination of redundant data structures (e.g. $q(a, f(Y)) \mapsto q'(Y)$),
- elimination of existential or unused variables.

Below we introduce a class of generalised renaming functions, supporting the first two functionalities stated above, but we abstract from details of whether and how they are performed. We will also present a post-processing, supporting the third functionality, in Section 4.

Definition 2.6. An *atomic renaming* α for a given set of conjunctions $\mathcal{Q}$ is a mapping from $\mathcal{Q}$ to atoms such that

- for each $Q \in \mathcal{Q}$: $vars(\alpha(Q)) = vars(Q)$ and
- for $Q, Q' \in \mathcal{Q}$ such that $Q \neq Q'$: the predicate symbols of $\alpha(Q)$ and $\alpha(Q')$ are distinct.

Note that with this definition, we are actually also renaming the atomic elements

[2] Take for example $P = \{p(a,X) \leftarrow, p(X,b) \leftarrow\}$. Then $P \cup \{\leftarrow p(X,Y) \wedge p(X,Y)\}$ has an SLD-refutation with computed answer semantics $\{X/a, Y/b\}$ while $P \cup \{\leftarrow p(X,Y)\}$ has not.

of $\mathcal{Q}$. This is not really essential for converting resultants into clauses, but it proves useful for various other aspects (e.g. dealing with independence).

Definition 2.7. Let α be an atomic renaming for $\mathcal{Q}$ and p a partitioning function. A *renaming function* $\rho_{\alpha,p}$ for $\mathcal{Q}$ (based on α and p) is a mapping from conjunctions to conjunctions such that:

$$\rho_{\alpha,p}(B) =_r \bigwedge_{C_i \in p(B)} \alpha(Q_i)\theta_i \text{ where each } C_i = Q_i\theta_i \text{ for some } Q_i \in \mathcal{Q}.$$

If some $C_i \in p(B)$ is not an instance of an element in $\mathcal{Q}$ then $\rho_{\alpha,p}(B)$ is undefined. Also, for a goal $\leftarrow Q$, we define $\rho_{\alpha,p}(\leftarrow Q) = \leftarrow \rho_{\alpha,p}(Q)$.

Observe that we do not necessarily have that $\alpha(Q) = \rho_{\alpha,p}(Q)$. Indeed, there are two degrees of non-determinism in defining $\rho_{\alpha,p}$ once α and p are fixed. First, if $\mathcal{Q}$ contains elements Q and Q' which share common instances, then there are several possible ways to rename these common instances and a multitude of renaming functions based on the same atomic renaming α and partitioning p exist. Secondly, the order in which the atoms $\alpha(Q_i)\theta_i$ occur in $\rho_{\alpha,p}(B)$ is not fixed beforehand and may therefore vary from one renaming function to another. Usually one would like to preserve the order in which the unrenamed atoms occurred in the original conjunction B. This is however not always possible, namely when the partitioning function assembles non-contiguous chunks from B. Take for instance the conjunction $B = q_1 \wedge q_2 \wedge q_3$, a partitioning p such that $p(B) = \{q_1 \wedge q_3, q_2\}$ and an atomic renaming α such that $\alpha(q_1 \wedge q_3) = qq$ and $\alpha(q_2) = q$. Then $\rho_{\alpha,p}(B) = qq \wedge q$ and $\rho'_{\alpha,p}(B) = q \wedge qq$ are the only possible renamings and in both of them q_2 has changed position. Fortunately the order of the atoms is of no importance for the usual declarative semantics, i.e. it does neither influence the least Herbrand model, the computed answers obtainable by SLD-resolution nor the set of finitely failed queries. The order might, however, matter if we restrict ourselves to some specific selection rule (like LD-resolution). We return to this issue in Section 5.1.

2.3. Conjunctive Partial Deduction

We are now in a position to give a definition of conjunctive partial deduction.

Definition 2.8. Let P be a program, $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ be a finite set of conjunctions, and $\rho_{\alpha,p}$ be a renaming for $\mathcal{Q}$ based on the atomic renaming α and the partitioning function p. For each $i \in \{1, \ldots, n\}$, let R_i be a conjunctive partial deduction of Q_i in P and let $P_{\mathcal{Q}} = \{R_i \mid i \in \{1, \ldots, n\}\}$. Then the program

$$\{\alpha(Q_i)\theta \leftarrow \rho_{\alpha,p}(B) \mid Q_i\theta \leftarrow B \in R_i \wedge 1 \leq i \leq n \wedge \rho_{\alpha,p}(B) \text{ is defined } \}$$

is called the *conjunctive partial deduction of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$*.

Returning to our example, we introduce a different predicate for each of the two elements in $\mathcal{Q}$ via the atomic renaming α:

- $\alpha(max_length(X, M, L)) = max_length(X, M, L)$ and
- $\alpha(max1(X, N, M) \wedge length(X, L)) = ml(X, N, M, L)$.

$\mathcal{Q}$ does not contain elements with common instances and for the resultants at hand there exists only one renaming function $\rho_{\alpha,p}$ for $\mathcal{Q}$ based on α and p. The conjunctive partial deduction wrt $\mathcal{Q}$ is now obtained as follows.

The head $max_length(X, M, L)$ in the single clause of R_1 is replaced by itself. The head-occurrences $max1([], N, N) \wedge length([], 0)$ and $max1([H|T], N, M) \wedge length([H|T], L)$

are replaced by $ml([\,], N, N, 0)$ and $ml([H|T], N, M, L)$.

The body occurrences $max1(X, 0, M) \wedge length(X, L)$, $max1(T, N, M) \wedge length(T, K)$ as well as $max1(T, H, M) \wedge length(T, K)$ are replaced by $ml(X, 0, M, L)$, $ml(T, N, M, K)$ and $ml(T, H, M, K)$ respectively.

The resulting program is:

$$
\begin{array}{l}
max_length(X, M, L) \leftarrow ml(X, 0, M, L) \\
ml([\,], N, N, 0) \leftarrow \\
ml([H|T], N, M, L) \leftarrow H \leq N \wedge ml(T, N, M, K) \wedge L \ is \ K + 1 \\
ml([H|T], N, M, L) \leftarrow H > N \wedge ml(T, H, M, K) \wedge L \ is \ K + 1
\end{array}
$$

Example 2.9. (*double append, revisited*) Let $P = \{C_1, C_2\}$ be the well known *append* program.

$$
\begin{array}{ll}
(C_1) & app([\,], L, L) \leftarrow \\
(C_2) & app([H|X], Y, [H|Z]) \leftarrow app(X, Y, Z)
\end{array}
$$

As discussed in Section 1, the goal $G = \leftarrow app(X, Y, I) \wedge app(I, Z, R)$ can be used to (inefficiently) concatenate three lists. We now show how conjunctive partial deduction, as defined above, offers salvation.

Let $\mathcal{Q} = \{app(X, Y, I) \wedge app(I, Z, R), app(X, Y, Z)\}$ and assume that we construct the finite SLD-tree τ_1 depicted in Figure 2, again labelling arcs with applied clause numbers, for the query $\leftarrow app(X, Y, I) \wedge app(I, Z, R)$ as well as a simple tree τ_2 with a single unfolding step for $\leftarrow app(X, Y, Z)$. Let $P_{\mathcal{Q}}$ consist of the clauses $resultants(\tau_2) = \{C_1, C_2\}$ as well as the resultants $resultants(\tau_1)$:

$$
\begin{array}{ll}
(R_1) & app([\,], Y, Y) \wedge app(Y, Z, R) \leftarrow app(Y, Z, R) \\
(R_2) & app([H|X'], Y, [H|I']) \wedge app([H|I'], Z, [H|R']) \leftarrow \\
 & \qquad app(X', Y, I') \wedge app(I', Z, R')
\end{array}
$$

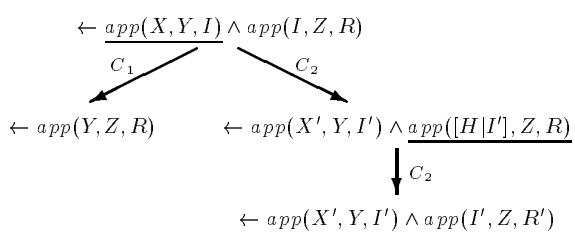


FIGURE 2. SLD-tree for Example 2.9

Suppose that we use a partitioning function p such that $p(B) = \{B\}$ for all conjunctions B. If we now take an atomic renaming α for $\mathcal{Q}$ such that $\alpha(app(X, Y, I) \wedge app(I, Z, R)) = da(X, Y, I, Z, R)$ and $\alpha(app(X, Y, Z)) = app(X, Y, Z)$ (i.e. the distinct variables have been collected and have been ordered according to their first appearance), the conjunctive partial deduction P' of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ will contain the clauses C_1, C_2 as well as:

$$
\begin{array}{ll}
(C_3') & da([\,], Y, Y, Z, R) \leftarrow app(Y, Z, R) \\
(C_4') & da([H|X'], Y, [H|I'], Z, [H|R']) \leftarrow da(X', Y, I', Z, R')
\end{array}
$$

In the conjunctive partial deduction, the inefficiency caused by the unnecessary traversal of I is avoided as the elements encountered while traversing X and

Y are stored directly in R. However, the intermediate list I is still constructed, and if we are not interested in its value, then this is an unnecessary overhead. This can be remedied through a post-processing phase to be presented in Section 4. The resulting specialised program then contains the clauses C_1, C_2 as well as:

(C_3) $\quad da([\,], Y, Z, R) \leftarrow app(Y, Z, R)$
(C_4) $\quad da([H|X'], Y, Z, [H|R']) \leftarrow da(X', Y, Z, R')$

It coincides with the desired program as shown in Section 1; the unnecessary variable I, as well as the inefficiencies caused by it, have now been completely removed.

Now that we have *defined* conjunctive partial deduction, the next few sections establish its *correctness* (Proposition 2.15 and Theorem 2.19).

2.4. Mapping to Transformation Sequences

Standard partial deduction is a strict subset of the (full) unfold/fold transformation techniques as defined for instance in the survey paper [62] by Pettorossi and Proietti. It is therefore not surprising that correctness can be established by showing that a conjunctive partial deduction can (almost) be obtained by a corresponding unfold/fold transformation sequence and then re-using correctness results from [62].

Note that, in contrast to [62], we treat programs as sets of clauses and not as sequences of clauses. The order of clauses makes no difference for the semantics we are interested in. In the remainder of this section, we will use the notations $hd(C), bd(C)$ of [62] to refer to the head and the body of a clause C respectively.

As stated in [62], a program transformation process starting from an initial program P_0 is a sequence of programs $P_0, \ldots, P_n$, called a *transformation sequence*, such that program P_{k+1}, with $0 \leq k < n$, is obtained from P_k by the application of a *transformation rule*, which may depend on $P_0, \ldots, P_k$. The following transformation rules and concepts are defined in [62]:

- *Unfolding* rule [62, (R1)]. We will use the terminology of "unfolding a clause C wrt a literal A (in the body of C), using (clauses $D_1, \ldots, D_n$ in) the program P_j". For example, given $P_0 = \{p \leftarrow q \wedge r, q \leftarrow r\}$, we can unfold $p \leftarrow q \wedge r$ wrt q using P_0, giving as a result the clause $p \leftarrow r \wedge r$.

- *Folding* rule [62, (R2)]. We will apply this rule only for single clauses and therefore use the terminology of "folding a clause C wrt $bd(D)\theta$ (in the body of C), using a clause D in the program P_j". In essence, folding is the inverse of the unfolding operation. For example we can fold $p \leftarrow q \wedge r$ wrt r using $q \leftarrow r$ in P_0 above, giving as result the clause $p \leftarrow q \wedge q$.

  The *T&S-Folding* rule [62, (R3)] is a restricted form of folding. We refer to it as "T&S-folding a clause C wrt $bd(D)\theta$, using a clause D in the program P_j".

- *Definition* [62, (R4)] and *T&S-Definition* rule [62, (R15)]; as well as the associated concepts of *old* and *new* predicates.

- The concept of *fold-allowing* [62, Definition 7].

Due to space restrictions, we have to refer the reader to [62] for the actual definitions (the T&S-folding and -definition rules are initially from the paper [74] by Tamaki and Sato).

In Definition 2.10 below, we map a conjunctive partial deduction to a transformation sequence. Basically the conjunctive partial deduction P' of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ can be obtained from P using 4 transformation phases. In the first phase, one introduces definitions for every conjunction in $\mathcal{Q}$, using the same predicate symbol as in α. In the second phase, these new definitions get unfolded according to the SLD-trees for the elements in $\mathcal{Q}$: exactly one unfolding step for each corresponding resolution step in the SLD-trees. In the third phase, conjunctions in the bodies of clauses are folded using the definitions introduced in phase 1. Finally, in the fourth phase, the original definitions in P are removed. The first three phases can be mapped to the unfold/fold transformation framework of [62] in a straightforward manner. Phase 4 will have to be treated separately because the clause removals do not meet the requirements of definition elimination transformations as defined in [62].

In Definition 2.6 of an atomic renaming, we did not require that the predicate symbols of the renamings were fresh, i.e. it is possible to reuse predicate symbols that occur in the original program P. This is of no consequence, because the original program is "thrown away". However, in unfold/fold, the original program is not systematically thrown away and in definition steps one can usually only define fresh predicates. To simplify the presentation, we restrict ourselves in a first phase to atomic renamings which only map to fresh predicate symbols, not occurring in the original program P. Those atomic renamings will be called *fresh*. At a later stage, we will extend the result to any atomic renaming satisfying Definition 2.6.

Definition 2.10. Let P' be the conjunctive partial deduction of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$. A *transformation sequence for* P' (given P, $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$) is a transformation sequence $P_0, \ldots, P_d, \ldots, P_u, \ldots, P_f$, such that $P_0 = P$, and

1. $P_0, \ldots, P_d$ contains only definition introductions, namely exactly one for every element $Q \in \mathcal{Q}$: $P_i = P_{i-1} \cup \{\alpha(Q) \leftarrow Q\}$.

2. $P_d, \ldots, P_u$ contains only unfolding steps using clauses of P_0, namely exactly one for every resolution step in the SLD-trees constructed (in order to obtain $P_{\mathcal{Q}}$) for the elements of $\mathcal{Q}$: i.e. if this resolution step in a tree for $Q \in \mathcal{Q}$ resolves a selected literal A with clauses $D_1, \ldots, D_n$, we perform an unfolding step of a clause $\alpha(Q)\theta \leftarrow F, A, G$ in some P_i wrt A, using $D_1, \ldots, D_n$ in P_0.

3. $P_u, \ldots, P_f$ contains only folding steps, namely exactly one for every renamed conjunction C in the body of a clause of $P_{\mathcal{Q}}$: i.e. for $C = Q\theta$, such that $Q \in \mathcal{Q} \wedge C \in p(B) \wedge \rho_{\alpha,p}(C) = \alpha(Q)\theta$, where $H \leftarrow B \in P_{\mathcal{Q}}$, we fold a corresponding clause $H \leftarrow B'$ wrt $Q\theta$, (where $B' =_r Q\theta \wedge R$) using the definition $\alpha(Q) \leftarrow Q$ in P_d, yielding the new clause $H \leftarrow B''$ (with $B'' =_r \alpha(Q)\theta \wedge R$).

Observe that unfolding always uses clauses in P_0 and folding always uses new definitions in P_d. The following example illustrates the above definition.

Example 2.11. Let $P = P_0 = \{C_1, C_2\}$ be the *append* program of Example 2.9 and let $\mathcal{Q}$, $\rho_{\alpha,p}$, $P_{\mathcal{Q}}$ and P' be defined as in that example except that we adapt

α slightly such that $\alpha(app(X,Y,Z)) = app'(X,Y,Z)$ (to make α fresh). Then the transformation sequence $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_f$, shown below, is a transformation sequence for P'. $P_1 = P_0 \cup \{Def_1\}$ and $P_2 = P_1 \cup \{Def_2\}$ are obtained by a definition introduction, where

(Def_1) $da(X,Y,I,Z,R) \leftarrow app(X,Y,I) \wedge app(I,Z,R)$
(Def_2) $app'(X,Y,Z) \leftarrow app(X,Y,Z)$

$P_3 = P_0 \cup \{U_1, U_2, Def_2\}$ is obtained by unfolding clause Def_1 wrt $app(X,Y,I)$, using P_0, where

(U_1) $da([\,],Y,Y,Z,R) \leftarrow app(Y,Z,R)$
(U_2) $da([H|X'],Y,[H|I'],Z,R) \leftarrow app(X',Y,I') \wedge app([H|I'],Z,R)$

$P_4 = P_0 \cup \{U_1, U_3, Def_2\}$ is obtained by unfolding clause U_2 wrt the atom $app([H|I'],Z,R)$, using P_0, where

(U_3) $da([H|X'],Y,[H|I'],Z,[H|R']) \leftarrow app(X',Y,I') \wedge app(I',Z,R')$

$P_5 = P_0 \cup \{U_1, U_3, U_4, U_5\}$ is now obtained by unfolding clause Def_2 wrt the atom $app(X,Y,Z)$ using P_0, where

(U_4) $app'([\,],L,L) \leftarrow$
(U_5) $app'([H|X],Y,[H|Z]) \leftarrow app(X,Y,Z)$

$P_6 = P_0 \cup \{U_1, U_3', U_4, U_5\}$ is obtained by folding the clause U_3 wrt $(app(X,Y,I) \wedge app(I,Z,R))\theta$, using clause Def_1 from P_1, where $\theta = \{X/X', I/I', R/R'\}$ and

(U_3') $da([H|X'],Y,[H|I'],Z,[H|R']) \leftarrow da(X',Y,I',Z,R')$

Finally, after two more folding steps using Def_2 from P_1 we obtain the final program $P_f = P_0 \cup \{U_1', U_3', U_4, U_5'\}$:

(C_1) $app([\,],L,L) \leftarrow$
(C_2) $app([H|X],Y,[H|Z]) \leftarrow app(X,Y,Z)$
(U_1') $da([\,],Y,Y,Z,R) \leftarrow app'(Y,Z,R)$
(U_3') $da([H|X'],Y,[H|I'],Z,[H|R']) \leftarrow da(X',Y,I',Z,R')$
(U_4) $app'([\,],L,L) \leftarrow$
(U_5') $app'([H|X],Y,[H|Z]) \leftarrow app'(X,Y,Z)$

The steps from P_0 to P_1 and P_1 to P_2 are applications of the T&S definition introduction rule. The last 3 steps are T&S-folding steps as defined in ([62];R3). However, e.g. the folding step from P_5 to P_6 is not an instance of the reversible folding rule (R13) of [62] (which would require $app(X',Y,I') \wedge app(I',Z,R')$ to be folded with a clause in P_5 and different from U_3). Also note that $P_f \setminus P = P'$.

2.5. Fair SLD-Trees

In Definition 2.3, (as in standard partial deduction [55]) we required the SLD-trees to be non-trivial. In the context of (standard) partial deduction of atoms, this condition avoids problematic resultants of the form $A \leftarrow A$ and is sufficient for total correctness (given independence and coveredness). In the context of conjunctive partial deductions, we need (for correctness wrt the finite failure semantics) an extension of this condition:

Definition 2.12. (*inherited, fair*) Let the goal $G' = \leftarrow (A_1 \wedge \ldots A_{i-1} \wedge B_1 \wedge \ldots B_k \wedge A_{i+1} \wedge \ldots A_n)\theta$ be derived via an SLD-resolution step from the goal $G = \leftarrow A_1 \wedge \ldots A_i \wedge \ldots A_n$, and the clause $H \leftarrow B_1 \wedge \ldots B_k$, with selected atom A_i. We say that the atoms $A_1\theta, \ldots, A_{i-1}\theta, A_{i+1}\theta, \ldots, A_n\theta$ in G are *inherited from G in*

G'. We extend this notion to derivations by taking the transitive and reflexive closure.

A finite SLD-tree τ for $P \cup \{G\}$ is said to be *fair* iff no atom in a dangling leaf goal L of τ is inherited from G in L.

The conjunctive partial deduction P' of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ is *fair* iff all the SLD-trees used to construct $P_{\mathcal{Q}}$ are fair.

The above means that every atom occurring in the top-level goal of an SLD-tree has to be selected at some point in every non-failing branch. For SLD-trees for atomic goals this notion coincides with the one of non-trivial trees (i.e. trees whose root is not a dangling leaf). Also, for the folding steps that we will perform (in the transformation sequence associated with a conjunctive partial deduction), this corresponds to conditions of *fold-allowing* in [62] or *inherited* in [69]. All these conditions ensure that we do not encode an unfair selection rule in the transformation process, which is vital when trying to preserve the finite failure semantics (for a more detailed discussion see e.g. [69]).

Sometimes however, this definition, as well as the one of fold-allowing in [62] or the one of inherited in [69], imposes more unfolding than strictly necessary. In some cases this forces one to perform non-leftmost, non-determinate unfolding, possibly leading to disastrous effects on the efficiency of the specialised program. Also, the tree τ_1 of Example 2.9 depicted in Figure 2 does not satisfy Definition 2.12, although the resulting program is actually totally correct. In order to make τ_1 fair, one would have to perform one more unfolding step on $\leftarrow app(Y, Z, R)$.

The following, weaker, notion of fairness remedies this problem.

Definition 2.13. (*weakly fair*) Let P' be the conjunctive partial deduction of P wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$. For $Q \in \mathcal{Q}$ let $Leaves_Q$ denote the dangling leaf goals of the SLD-tree for $P \cup \{\leftarrow Q\}$ used to construct the corresponding resultants in $P_{\mathcal{Q}}$. We first define the following (increasing) series $\mathcal{WF}_i$ of subsets of $\mathcal{Q}$:

- $Q \in \mathcal{WF}_0$ iff $Q \in \mathcal{Q}$ and for each $L \in Leaves_Q$ no atom is inherited from $\leftarrow Q$ in L.
- $Q \in \mathcal{WF}_{k+1}$ iff $Q \in \mathcal{Q}$ and for each $L \in Leaves_Q$ and each $C \in p(L)$ which contains an atom inherited from $\leftarrow Q$ in L and which gets renamed into $\alpha(Q')\theta$ (with $C = Q'\theta$ and $Q' \in \mathcal{Q}$) inside $\rho_{\alpha,p}(L)$ we have that $Q' \in \mathcal{WF}_k$.

Then P' is *weakly fair* iff there exists a number $0 \leq k < \infty$ such that $\mathcal{Q} = \mathcal{WF}_k$.

Note that if every SLD-tree τ_Q is fair (i.e. P' is fair) then P' is weakly fair, independently of the renaming function $\rho_{\alpha,p}$ (because no atom in a leaf is inherited from the root goal and thus $\mathcal{WF}_0 = \mathcal{Q}$). Intuitively, the above definition ensures that every atom in a conjunction Q in $\mathcal{Q}$ is either unfolded directly in the tree $\tau(Q)$ or it is folded on a conjunction Q' in which the corresponding atom is guaranteed to be unfolded (again either directly or indirectly by folding and so on in a well-founded manner).

Example 2.14. Let $P_{\mathcal{Q}}$ be the resultants for the set $\mathcal{Q} = \{app(X, Y, I) \wedge app(I, Z, R), app(X, Y, Z)\}$ of Example 2.9. The simple tree for $P \cup \{\leftarrow app(X, Y, Z)\}$ is fair. Therefore $app(X, Y, Z) \in \mathcal{WF}_0$ independently of $\rho_{\alpha,p}$.

Let τ_1 be the SLD-tree for $P \cup \{G\}$, with $G = \leftarrow app(X, Y, I) \wedge app(I, Z, R)$, of Figure 2. The conjunctions in the dangling leaves of τ are $\{L_1, L_2\}$, with $L_1 =$

$app(Y, Z, R)$ and $L_2 = app(X', Y, I') \wedge app(I', Z, R')$. The SLD-tree τ_1 is not fair, but for $\rho_{\alpha,p}$ of Example 2.9, we have that $app(X, Y, I) \wedge app(I, Z, R) \in \mathcal{WF}_1$:

- $app(X', Y, I')$ and $app(I', Z, R')$ are not inherited from G in $\leftarrow L_2$.
- $app(Y, Z, R)$ is inherited from G in $\leftarrow L_1$, but $\rho_{\alpha,p}(L_1) = \alpha(app(X, Y, Z))\theta$ and, as we have seen above, $app(X, Y, Z) \in \mathcal{WF}_0$.

So for $k = 1$ we have that $\mathcal{Q} = \mathcal{WF}_k$ and P' is thus weakly fair.

2.6. Correctness of Conjunctive Partial Deduction

We now have the necessary apparatus to actually prove correctness of conjunctive partial deduction.

Proposition 2.15. Let $P_0, \ldots, P_f$ be a transformation sequence for the conjunctive partial deduction P' of P_0 wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ based on a fresh atomic renaming. Then, for every goal G, such that its predicates occur in P_0, we have that

- *$P_0 \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P_f \cup \{G\}$ has.*

If in addition P' is weakly fair then

- *$P_0 \cup \{G\}$ has a finitely failed SLD-tree iff $P_f \cup \{G\}$ has.*

Proof. The following proof frequently refers to [62]. A more self-contained, but considerably longer, presentation can be found in [41]. First we show two lemmas.

In Lemma 1 below, we first establish a necessary condition in order to apply some of the theorems from [62], namely that the definition steps in Definition 2.10 are *T&S-Definition* steps [74, 62].

Lemma 1. Let $P_0, \ldots, P_f$ be a transformation sequence for P' constructed using a fresh atomic renaming. All the definition introduction steps of $P_0, \ldots, P_f$ are T&S definition steps.

All definition steps are of the form: $P_k = P_{k-1} \cup \{\alpha(Q) \leftarrow Q\}$. The conditions imposed on α guarantee that the predicate of $\alpha(Q)$ does not occur in $P_0, \ldots, P_k$ (point 1 of R15 in [62]). Point 2 of R15 requires that all the predicates in the body of the introduced clause are only old predicates. According to the definitions in [74], all predicates in P_0 are old and hence this condition is trivially satisfied. However, for the slightly modified definitions used in [62], this is not always the case, but we can use the following simple construction to make every predicate in P an old predicate. Let the predicates occurring in P be $p_1, \ldots, p_j$, and let $fresh$ and $fail$ be distinct propositions not occurring in P, nor in the image of α. We simply define $P_0 = P \cup \{C_f\}$ where $C_f = fresh \leftarrow fail, p_1(\bar{t_1}), \ldots, p_j(\bar{t_j})$ and the t_i are sequences of terms of correct length. By doing so, we do not modify any of the semantics we are interested in, but ensure that all the predicates in P are old according to the definition in [62]. We implicitly assume the presence of such a C_f in the following lemmas and propositions as well (in case we want to apply the modified definitions used in [62]). This concludes the proof of Lemma 1.

The following shows that, due to our particular way of defining renamings, the folding steps in a transformation sequence are *T&S-Folding* steps [74, 62].

Lemma 2. Let $P_0, \ldots, P_f$ be a transformation sequence for the conjunctive partial deduction P' of P_0 wrt $\mathcal{Q}$, $P_{\mathcal{Q}}$ and $\rho_{\alpha,p}$ based on a fresh atomic renaming. Then the folding steps in $P_0, \ldots, P_f$ are T&S-folding steps which satisfy the requirements of Theorems 8 and 10 in [62]. If in addition P' is fair, then the T&S-folding steps also satisfy the requirements of Theorem 12 in [62].

In order to prove that the folding steps of Definition 2.10 are T&S-folding steps, we have to show that points 2 and 3 of R3 in [62] hold. Point 3 states that the predicate symbol of $\alpha(Q)$ should occur only once in P_d (where P_d is the program of Definition 2.10 obtained after all definitions have been introduced), which holds trivially by construction of the definitions in P_d and because the atomic renaming α is fresh. Also, point 2 of R3 states that variables removed by the renaming should be existential variables. Because we imposed $vars(\alpha(Q)) = vars(Q)$ for atomic renamings, no variables are removed and the criterion is trivially satisfied.

The fact that we have non-trivial SLD-trees ensures that at least one atom in B is fold-allowing and hence, the requirements of Theorem 8 and 10 hold. Furthermore, if P' is fair, then every atom in B wrt which T&S-folding is performed (i.e. every atom in Q) is fold-allowing, and the requirements of Theorem 12 are met.

This concludes the proof of Lemma 2. If P' is only weakly fair then the requirements of Theorem 12 in [62] are not met. We will have to deal with that special case separately later on.

Lemmas 1 and 2 ensure that the prerequisites of Theorem 10 in [62] are met. Hence the computed answer semantics Sem_{CA} is preserved under the above conditions and $P_0 \cup \{G\}$ has an SLD-refutation with computed answer θ iff $P_f \cup \{G\}$ has.[3]

If P' is fair we can use the same Lemmas 1 and 2 combined with Theorem 12 of [62] to deduce that the finite failure semantics Sem_{FF} is preserved, i.e. $P_0 \cup \{G\}$ has a finitely failed SLD-tree iff $P_f \cup \{G\}$ has.

As already mentioned earlier, in case P' is only weakly fair we cannot directly apply Theorem 12 of [62]. We therefore do a specific proof by induction on the minimum number min such that $\mathcal{Q} = \mathcal{WF}_{min}$, where the $\mathcal{WF}_i$ are defined as in Definition 2.13.

Base Case: If $min = 0$ then for every $Q \in \mathcal{Q}$ no leaf contains an atom inherited from $\leftarrow Q$ and thus P' is fair. Hence, by the above reasoning, we can deduce the preservation of finite failure.

Induction Hypothesis: The finite failure semantics is preserved for all P' which are weakly fair and such that $min \leq k$.

Induction Step: Let $min = k + 1$ and let $\mathcal{W} \subset \mathcal{Q}$ be defined as $\mathcal{W} = \mathcal{WF}_{k+1} \setminus \mathcal{WF}_k$. The idea of the proof is to unfold the clauses for the elements of $\mathcal{W}$ so that, according to Definition 2.13, they become elements of $\mathcal{WF}_k$ in the unfolded program P'_f. This will allow us to apply the induction hypothesis on P'_f. The details are elaborated in the following. As in Definition 2.13, we denote by $Leaves_Q$ (with $Q \in \mathcal{Q}$) the dangling leaf goals of the SLD-tree for $P \cup \{\leftarrow Q\}$ used to construct the corresponding resultants in $P_{\mathcal{Q}}$. Let P'_f be obtained from P_f by performing the following unfolding steps for every element $Q \in \mathcal{W}$:

[3] Note that Lemmas 1 and 2 also ensure that Theorem 8 of [62] can be applied and thus, given a fixed first-order language $\mathcal{L}_P$, the least Herbrand model semantics Sem_H is also preserved (restricted to the predicates occurring in the original program P_0). We will not use this property in the remainder of this paper however.

for each $L \in Leaves_Q$ and each $C \in p(L)$ which contains an atom inherited from $\leftarrow Q$ in L and which gets renamed into $\alpha(Q')\theta$, unfold the clause corresponding to L wrt $\alpha(Q')\theta$ (i.e. the renamed version of C inside $\rho_{\alpha,p}(L)$) using the definition of $\alpha(Q')$ in P_f.

Note that P'_f can be obtained by a transformation sequence for a partial deduction P'' based on the same atomic renaming α and the same set $\mathcal{Q}$ as P' but based on SLD-trees with a deeper unfolding for the elements of $\mathcal{W}$[4]. Each element of $\mathcal{Q} \setminus \mathcal{W}$ is still in $\mathcal{WF}_k$ (as well as in $\mathcal{WF}_i, i < k$ if it was in $\mathcal{WF}_i$ for P_f) because the associated trees and resultants remain unchanged. We also know that every Q' above must be in $\mathcal{Q} \setminus \mathcal{W} = \mathcal{WF}_k$, because the second rule of Definition 2.13 could be applied to deduce that $Q \in \mathcal{WF}_{k+1}$. Hence in P'' associated with P'_f, each element of $\mathcal{W}$ is now in $\mathcal{WF}_k$, due to the unfolding. Hence we can apply the induction hypothesis to deduce that finite failure is preserved in P'_f wrt P_0. Now because unfolding is totally correct wrt the finite failure semantics Sem_{FF}, we know that P_f and P'_f are equivalent under Sem_{FF}. Thus the induction hypothesis holds for $max = k + 1$. □

We are now in a position to state a correctness result similar to the one in [55]. In contrast to [55], we do not need an independence condition (because of the renaming), but we still need an adapted coveredness condition:

Definition 2.16. ($\mathcal{Q}$-*covered wrt* p) Let p be a partitioning function and $\mathcal{Q}$ a set of conjunctions. We say that a conjunction Q is $\mathcal{Q}$-*covered wrt* p iff every conjunction $Q' \in p(Q)$ is an instance of an element in $\mathcal{Q}$. Furthermore a set of resultants R is $\mathcal{Q}$-*covered wrt* p iff every body of every resultant in R is $\mathcal{Q}$-covered wrt p.

The above coveredness condition ensures that the renamings performed in Definition 2.8 are always defined and that the original program P can be thrown away from the end result of a transformation sequence for the associated conjunctive partial deduction.

Example 2.17. Let $\mathcal{Q} = \{q(x) \wedge r, q(a)\}$, $Q = q(a) \wedge q(b) \wedge r$. Then, for a partitioning function p such that $p(Q) = \{q(b) \wedge r, q(a)\}$, Q is $\mathcal{Q}$-covered wrt p. However, for p' with $p'(Q) = \{q(a) \wedge r, q(b)\}$, Q is not $\mathcal{Q}$-covered wrt p'.

Proposition 2.18 establishes a correspondence between the result P_f of the above transformation sequence and the corresponding conjunctive partial deduction.

Proposition 2.18. Let P' be the conjunctive partial deduction of P wrt $\mathcal{Q}$, $P_\mathcal{Q}$ and $\rho_{\alpha,p}$ such that $P_\mathcal{Q}$ is $\mathcal{Q}$-covered wrt p. Also let $P_0, \ldots, P_f$ be a transformation sequence for P'. Then $P_f \setminus P = P'$, where P is the original program.

Theorem 2.19. Let P' be the conjunctive partial deduction of P wrt $\mathcal{Q}$, $P_\mathcal{Q}$ and $\rho_{\alpha,p}$. If $P_\mathcal{Q} \cup \{G\}$ is $\mathcal{Q}$-covered wrt p then

- *$P \cup \{G\}$ has an SLD-refutation with computed answer semantics θ iff $P' \cup \{\rho_{\alpha,p}(G)\}$ has an SLD-refutation with computed answer semantics θ.*

If in addition P' is weakly fair then

- *$P \cup \{G\}$ has a finitely failed SLD-tree iff $P' \cup \{\rho_{\alpha,p}(G)\}$ has.*

[4] Possibly a slightly adapted renaming function is needed to ensure that the renamings of the new leaves of these deeper SLD-trees coincide with the clause bodies obtained by the unfolding performed on P_f.

PROOF. Let us first prove the theorem for conjunctive partial deductions constructed using a fresh atomic renaming α. Let $x_1, \ldots, x_n$ be the variables of G ordered according to their first appearance and let *query* be a fresh predicate of arity n. We then define $P_0 = P \cup \{query(x_1, \ldots, x_n) \leftarrow Q_G\}$ where $G = \leftarrow Q_G$. The conjunctive partial deduction of P_0 will be identical to the one of P except for the extra clause for *query*. We can now construct a transformation sequence $P_0, \ldots, P_f$ for the conjunctive partial deduction of P_0 and then apply Proposition 2.15 to deduce that $query(x_1, \ldots, x_n)$ has the same computed answer and finite failure behaviour in P_0 and P_f. Note that *query* is defined in P_f by the clause $query(x_1, \ldots, x_n) \leftarrow \rho_{\alpha,p}(Q_G)$. Hence $P \cup \{G\}$ has the same behaviour wrt computed answers and finite failure as $P_f \cup \{\rho_{\alpha,p}(G)\}$. Finally $P' = P_f \setminus \{P \cup query(x_1, \ldots, x_n) \leftarrow \rho_{\alpha,p}(Q)\}$. Hence, the theorem follows from the fact that, due to $\mathcal{Q}$-coveredness wrt p, the predicates defined in P, as well as the predicate *query*, are inaccessible from $\rho_{\alpha,p}(Q)$ in the predicate dependency graph.

Let us now prove the result for unrestricted renaming. For that we simply introduce a fresh intermediate renaming and prove the result by two applications of the above theorem. More precisely, let α' and α'' be such that $\alpha(Q) = \alpha''(\alpha'(Q))$ for every $Q \in \mathcal{Q}$ and such that α' is a fresh atomic renaming for $P_{\mathcal{Q}}$ and also such that α'' is a fresh atomic renaming wrt the range of α'. Such renamings can always be constructed. We can now apply the above result to deduce that the conjunctive partial deduction P'', obtained from $P_{\mathcal{Q}}$ under $\rho_{\alpha',p}$, is totally correct for the query $\rho_{\alpha',p}(G)$. The conjunctive partial deduction P' (as well as the query $\rho_{\alpha,p}(G)$) can be obtained from P'' by performing a (standard) partial deduction wrt the set $\mathcal{A} = \{\alpha'(Q) \mid Q \in \mathcal{Q}\}$ and by unfolding every atom in $\mathcal{A}$ exactly once. Hence we can re-apply the above theorem and we obtain total correctness of P' wrt P. □

Example 2.20. Let $P_{\mathcal{Q}}$, P and P' be taken from Example 2.9. Consider $G = \leftarrow app([1,2],[3],I) \wedge app(I,[4],R)$. We have that $\rho_{\alpha,p}(G) = da([1,2],[3],I,[4],R)$. It can be seen that $P_{\mathcal{Q}} \cup \{G\}$ is $\mathcal{Q}$-covered wrt p and indeed $P \cup \{G\}$ and $P_{\rho_{\alpha,p}} \cup \{\rho_{\alpha,p}(G)\}$ have the same set of computed answers: $\{\{I/[1,2,3], R/[1,2,3,4]\}\}$. Note that P', as mentioned in Example 2.14 above, is weakly fair and therefore finite failure is also preserved.

2.7. Negation and Normal Programs

To conclude Section 2, we briefly discuss how the foundations of conjunctive partial deduction might be extended to provide for negation.

Now, when negative literals cannot be selected during the transformation, this is not so difficult: a lot of results from the literature can be reused. For instance, we can recycle results from the unfold/fold literature to prove preservation of the perfect model semantics for stratified programs [69] and preservation of the well-founded semantics for normal programs [70]. If in addition we have fairness, the conditions of modified (T&S) folding of [69] hold, and we can use preservation of the SLDNF success and finite failure set for stratified programs. Some further results from [2] can also be applied. In [8], the correctness results of [69] are adapted for Fitting's semantics and the results might also be applicable in our case. The results of [25] do not seem to be applicable because they use a different folding rule (which requires the clauses involved in the folding process to be all in the same program P_i).

Lifting the above restriction, however, and allowing general SLDNF-trees during transformation is more difficult. Note that [69, 70, 62, 2, 8] do not allow the unfolding of negative literals. Selecting negative literals might be obtained by *goal replacement* or *clause replacement*, but Theorems 15 and 16 from [62] cannot be applied because different folding rules are used. [68] allows unfolding inside negation and works with first order formulas, but it still has to be investigated whether its results (for Kleene's 3-valued logic) can be used. Also [32] has a negative unfolding rule (under certain termination conditions), but this rule (along with the correctness theorems) is situated in the context of deriving *definite* logic programs from first order specifications. So, for the moment, there seems to be no conjunctive equivalent to the correctness theorem of [55] for normal logic programs and partial deduction based on constructing finite SLDNF-trees. Further work will be needed to extend the correctness results of the previous section.

3. CONTROL ISSUES AND ALGORITHMS

The framework presented in the previous section incorporates unfold/fold-like transformations through specialisation of entire conjunctions, but does not give an actual algorithm for conjunctive partial deduction, and in particular does not address *control* issues. Focusing on novel control challenges, we will in this section present a basic algorithm for conjunctive partial deduction, refine it into a fully automatic concrete algorithm, and prove termination and correctness of the latter.

3.1. Controlling Partial Deduction

In recent years considerable progress has been achieved on the issue of controlling standard partial deduction. In that context, a conceptual distinction was introduced between *local* and *global control* [24, 59].

Local Control

The local control level deals with the construction of (possibly incomplete) SLD-trees for the atoms to be partially deduced. In essence, it consists of an unfolding strategy. This may be done by specifying a rule for selecting atoms to unfold, and unfold until no more atoms are select by the rule. Requirements are: termination, good specialisation, and avoiding search space explosion as well as work duplication. Existing approaches have been based on one or more of the following elements:

- *determinacy* [23, 22]
  Only (except once) select atoms that match a single clause head. The strategy can be refined with a so-called "look-ahead" to detect failure at a deeper level. Methods solely based on this heuristic, apart from not guaranteeing termination, are often somewhat too conservative.

- *well-founded measures* [11, 58]
  Imposing some (essentially) well-founded order on selected atoms guarantees termination, but, on its own, can lead to overly eager unfolding.

- *homeomorphic embedding* [72, 50]
  Instead of well-founded ones, well-quasi-orders can be used [67, 4]. Homeomorphic embedding on selected atoms has recently gained popularity as the basis for such an order.

Global Control

At the global control level, closedness [55] is ensured and the degree of polyvariance is decided: For which atoms should partial deductions be produced? Obviously, again, termination is an important issue, as well as obtaining a good overall specialisation. The following ingredients are important in recent approaches:

- *characteristic trees* [23, 22, 45, 40]
  A characteristic tree is an abstraction of an SLD-tree. It registers which atoms have been selected and which clauses were used for resolution. As such, it provides a good characterisation of the computation and specialisation connected with a certain atom (or goal). Its use in partial deduction lies in the control of polyvariance: Produce one specialised definition per characteristic tree encountered.

- *global trees* [59, 50, 51]
  Partially deduced atoms (or characteristic atoms, see below) can be registered in a tree structure that is kept well-founded or well-quasi-ordered to ensure (global) termination. In general, doing so, while maintaining closedness, requires abstraction (generalisation).

- *characteristic atoms* [40, 50, 51]
  Recent work has shown that the best control of polyvariance can be obtained not on the basis of either syntactical structure (atoms) or specialisation behaviour (characteristic trees) separately, but rather through a combination of both. Such pairs consisting of an atom and an associated (imposed) characteristic tree are called *characteristic atoms*.

Finally, subsidiary transformations, applicable in a post-processing phase, have been proposed, e.g. to remove certain superfluous structures [21, 3] or to reduce unnecessary polyvariance [51].

The essential aspect of conjunctive partial deduction lies in the joint treatment of entire conjunctions of atoms, connected through shared variables, at the global level.

A termination problem specific to conjunctive partial deduction therefore lies in the possible appearance of ever growing conjunctions at the global level (see Section 3 of [58] for a comparable phenomenon in the context of local control). To cope with this, abstraction [24, 50, 26] must allow *splitting* a conjunction into several parts, thus producing *subconjunctions* of the original one. (See also e.g. [65] for a related generalisation operation in the context of an unfold/fold transformation technique.) This can be seen as a refinement of abstraction wrt the standard case, where any conjunction is always split (i.e. abstracted) into its constituent atoms before lifting the latter to the global level.

Apart from this aspect, the conventional control notions described above also apply in a conjunctive setting. Notably, the concept of characteristic atoms can

be generalised to *characteristic conjunctions*, which are just pairs consisting of a conjunction and an associated characteristic tree.

In fact, in one sense the control problem for conjunctive partial deduction seems to be *easier* than in the conventional setting. Since conventional partial deduction splits atoms at the global level, it is crucial to have an aggressive local unfolding mechanism, so as to accommodate communication between different atoms in a conjunction at *this* level. In contrast, this splitting is not done in conjunctive partial deduction, and therefore the local level does not seem equally crucial.[5]

3.2. A Basic Conjunctive Partial Deduction Algorithm

We first present a basic algorithm which computes conjunctive partial deductions satisfying the conditions of Theorem 2.19. The algorithm uses an *unfolding rule* for controlling local unfolding and an *abstraction operator* for controlling global termination, respectively.

Definition 3.1. An *unfolding rule* U maps a program P and a conjunction Q to a non-trivial SLD-tree for $P \cup \{\leftarrow Q\}$.

Definition 3.2. An *abstraction operator* Abs maps any finite set of conjunctions $\mathcal{Q}$ to a finite set of conjunctions $Abs(\mathcal{Q})$ such that if $Q \in \mathcal{Q}$, there exist $Q_i \in Abs(\mathcal{Q})$ and θ_i $(i = 1 \ldots n)$ with $Q =_r Q_1\theta_1 \wedge \ldots \wedge Q_n\theta_n$. For a single conjunction Q we write $Abs(Q)$ for $Abs(\{Q\})$.

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

Algorithm 3.3. (*basic algorithm*)

Input: a program P and a goal $\leftarrow Q$
Output: a set of conjunctions $\mathcal{Q}$
Initialisation: $\mathcal{Q}_{new} := \{Q\}$
repeat
  $\mathcal{Q}_{old} := \mathcal{Q}_{new}$;
  for all $Q \in \mathcal{Q}_{old}$ **do**
    for all $B \in bodies(U(P, Q))$ **do**
      $\mathcal{Q}_{new} := Abs(\mathcal{Q}_{new} \cup \{B\})$
until $\mathcal{Q}_{old} = \mathcal{Q}_{new}$ (modulo variable renaming)
output $\mathcal{Q} := \mathcal{Q}_{new}$

When Q in goal $\leftarrow Q$ is an atom, and the abstraction operator Abs splits every conjunction into atoms, subsequently performing some generalisation on the resulting set, Algorithm 3.3 is essentially Gallagher's Basic Algorithm [24] restricted to definite programs.

From a program P and a goal $\leftarrow Q$, using some unfolding rule U and abstraction operator Abs, Algorithm 3.3 constructs a set of conjunctions $\mathcal{Q}$, which determines a conjunctive partial deduction of each $Q_i \in \mathcal{Q}$. 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 renaming, and

[5]This perhaps explains why analogous transformers for functional programs, e.g., supercompilation, have not found it necessary to operate (explicitly) with a local unfolding level.

use this to construct a conjunctive partial deduction of P wrt $\mathcal{Q}$, satisfying the conditions of Theorem 2.19.

3.3. A Concrete Conjunctive Partial Deduction Algorithm

We now refine the above basic algorithm for conjunctive partial deduction into a concrete one. Following [59, 50] for the conventional case, we introduce a *tree structure* to record dependencies among conjunctions in the successive $\mathcal{Q}_{new}$ and choose specific unfolding and abstraction operators. Throughout, we adhere to a conceptual *separation between local and global control* [24, 59, 50].

Trees for Global Control

As in conventional 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. We start by giving a definition of global trees:

Definition 3.4. A *global tree* γ is a labelled tree, where every node N is labelled with a conjunction Q_N. $Nodes_\gamma$ denotes the set of its labels, and $Leaves_\gamma \subseteq Nodes_\gamma$ the set of its leaf labels. For a branch β in γ, Seq_β is the *sequence* of these labels, in the order they appear in β. For a leaf node $L \in \gamma$, β_L denotes the (unique) branch in γ containing L.

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 conventional context (cf. Section 3.5).

Algorithm 3.3 is then refined as follows where each iteration no longer considers all conjunctions in $\mathcal{Q}_{old}$, but only those labelling leaves of γ_{old} (all not yet partially deduced conjunctions in the global tree are indeed leaf labels).

Algorithm 3.5. (*concrete algorithm*)

Input: a program P and a goal $\leftarrow Q$
Output: a set of conjunctions $\mathcal{Q}$
Initialisation: $\gamma_{new} :=$ the global tree with a single node, labelled Q
repeat
  $\gamma_{old} := \gamma_{new}$;
  for all $Q_L \in Leaves_{\gamma_{old}}$ **do**
    for all $B \in bodies(U(P, Q_L))$ **do**
      $\{Q_1, \ldots, Q_n\} := \{Q \in Abs_{\gamma_{old},L}(B) \mid \nexists Q' \in Nodes_{\gamma_{new}} : Q \equiv Q'\}$;
      γ_{new}:=add n children to L with labels $Q_1, \ldots, Q_n$ in γ_{new}
until $\gamma_{old} = \gamma_{new}$
output $\mathcal{Q} := Nodes_{\gamma_{new}}$

The abstraction operators $Abs_{\gamma,L}$ are applied to a single conjunction at a time and, when abstracting the body of a new resultant, they may e.g. only take the conjunctions in the branch β_L in γ into account, which the new child nodes are potentially going to extend. Note, however, that we do not add variants of labels already present anywhere else in the tree.

What remains is to choose an unfolding rule U and an abstraction operator $Abs_{\gamma,L}$, and to discuss termination and correctness for the corresponding conjunctive partial deduction algorithm.

Unfolding Rules

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. As mentioned above, there exists an extensive literature on this topic in conventional partial deduction. We propose a method which ensures non ad hoc local termination and provides a good basis for performing the kind of transformations we have in mind.

The following homeomorphic embedding relation $\trianglelefteq$ is taken from [50, 51] where in turn it was adapted from [72]. The power of $\trianglelefteq$ is discussed in [43] and other ways to improve $\trianglelefteq$ in a logic programming context can be found in [44].

Definition 3.6. (strict homeomorphic embedding) Let X, Y range over variables, f over functors, and p over predicates. Define $\trianglelefteq$ on terms and atoms:

$$\begin{array}{ll} X \trianglelefteq Y & \\ s \trianglelefteq f(t_1,\ldots,t_n) & \Leftarrow s \trianglelefteq t_i \text{ for some } i \\ f(s_1,\ldots,s_n) \trianglelefteq f(t_1,\ldots,t_n) & \Leftarrow s_i \trianglelefteq t_i \text{ for all } i \\ p(s_1,\ldots,s_n) \trianglelefteq p(t_1,\ldots,t_n) & \Leftarrow s_i \trianglelefteq t_i \text{ for all } i \text{ and } p(t_1,\ldots,t_n) \not\leq p(s_1,\ldots,s_n) \end{array}$$

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

Definition 3.7. (*descends*) Let the goal $G' = \leftarrow (A_1 \wedge \ldots A_{i-1} \wedge B_1 \wedge \ldots B_k \wedge A_{i+1} \wedge \ldots A_n)\theta$ be derived via an SLD-resolution step from the goal $G = \leftarrow A_1 \wedge \ldots A_i \wedge \ldots A_n$, and the clause $H \leftarrow B_1 \wedge \ldots B_k$, with selected atom A_i. We say that the atoms $B_1\theta, \ldots B_k\theta$ in G' *descend* from A_i in G as well as that $A_j\theta$ in G', for $j \neq i$, *descends* from A_j in G. We extend this notion to derivations by taking the transitive and reflexive closure.

Definition 3.8. (*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 3.9. (*concrete unfolding rule*) Repeatedly unfold the left-most selectable atom in each leaf of the SLD-tree under construction until no atom is selectable.

The following theorem is an extension of Higman-Kruskal's theorem ([28, 35], see also [17]) proven in [51].

Theorem 3.10. For any infinite sequence $A_0, A_1, \ldots$ of atoms, for some $0 \leq i < j$: $A_i \trianglelefteq A_j$.

And we obtain the following corollary from Definitions 3.6, 3.8 and 3.9, and Theorem 3.10.

Corollary 3.11. Let P be a program, G a goal, and U the unfolding rule in Definition 3.9. Then $U(P,G)$ is a finite, fair SLD-tree for $P \cup \{G\}$. We say the unfolding rule U is terminating.

Abstraction Operators

We now specify the abstraction operators $Abs_{\gamma,L}$, deciding which conjunctions are added to the global tree in order to ensure coveredness for bodies of newly derived resultants.

Ensuring coveredness is basically 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 $bodies(U(P,Q_L))$, the abstraction operator $Abs_{\gamma,L}$ should consider whether adding B to the global level may endanger termination. To this end, $Abs_{\gamma,L}$ should detect whether B is (in some sense) bigger than another label already occurring in Seq_{β_L}, since adding B might then lead to some systematic growing behaviour resulting in non-termination.

According to Definition 3.2, 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.

Since we aim at removing 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 (conjunctions)*[6] 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 3.12. (*maximal connected subconjunctions*) Given a conjunction $Q = A_1 \wedge \ldots \wedge A_n$, where $A_1, \ldots, A_n$ are atoms, we define the binary relation $\downarrow_Q$ over the atoms in Q as follows: $A_i \downarrow_Q A_j$ iff $vars(A_i) \cap vars(A_j) \neq \emptyset$. By $\Downarrow_Q$ we denote the reflexive and transitive closure of $\downarrow_Q$. The *maximal connected subconjunctions* of Q, denoted by $mcs(Q)$, are defined to be the multiset of conjunctions $\{Q_1, \ldots, Q_m\}$ such that

1. $Q =_r Q_1 \wedge \ldots \wedge Q_m$,
2. $A_i \Downarrow_Q A_j$ iff A_i and A_j occur in the same Q_k and
3. for every Q_k there exists a sequence of indices $j_1 < j_2 < \ldots < j_l$ such that $Q_k = A_{j_1} \wedge \ldots \wedge A_{j_l}$.

For two conjunctions $Q = A_1 \wedge \ldots \wedge A_n$ and $Q' = A'_1 \wedge \ldots \wedge A'_n$ where A_i and A'_i have the same predicate symbols for all i, a most specific generalisation $msg(Q, Q')$ exists, which is unique modulo variable renaming.

Given a conjunction $Q = A_1 \wedge \ldots \wedge A_k$ any conjunction $Q' = A_{i_1} \wedge \ldots \wedge A_{i_j}$ such that $1 \leq i_1 < \ldots < i_j \leq k$ is called an *ordered subconjunction* of Q.

We now extend the definition of homeomorphic embedding to conjunctions.

[6] This notion is closely related to those of "variable-chained sequence" and "block" of atoms used in [64, 65].

Definition 3.13. (*homeomorphic embedding*) Let $Q = A_1 \wedge \ldots \wedge A_n$ and Q' be conjunctions. We say that Q is embedded in Q', denoted by $Q \trianglelefteq Q'$, iff $Q' \not\preceq Q$ and there exists an ordered subconjunction $A'_1 \wedge \ldots \wedge A'_n$ of Q' such that $A_i \trianglelefteq A'_i$ for all i.

This relation $\trianglelefteq$ still satisfies Theorem 3.10 (for sequences of conjunctions). This can be proven easily using the results of [51] combined with Higman-Kruskal's theorem ([28, 35], see also [17]) by considering $\wedge$ as a functor of variable arity (i.e. an associative operator).

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

Assume that $Q = A_1 \wedge \ldots \wedge A_n$ is embedded in Q'. An obvious way is to split Q' into $A'_1 \wedge \ldots \wedge A'_n$ and R, where each 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 Seq_{β_L}. Thus, in order to obtain a set of conjunctions not embedding any label in Seq_{β_L}, we *recursively repeat* splitting on R.

There can be several conjunctions in Seq_{β_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 connecting variables. Consider two conjunctions $Q = p(X,Y) \wedge q(Y,Z)$ and $Q' = p(X,T) \wedge p(T,Y) \wedge q(Y,Z)$. 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 3.14. (*best matching conjunction*) Let Q be a conjunction and $\mathcal{Q}$ be a set of conjunctions. A *best matching conjunction for Q in $\mathcal{Q}$* is a minimally general element of the set

$$bmc(Q, \mathcal{Q}) = \{msg(Q, Q') \mid Q' \in \mathcal{Q} \text{ and } msg(Q, Q') \text{ exists}\}$$

The set $bmc(Q, \mathcal{Q})$ may be empty, but when it is non-empty, we denote by $bmc(Q, \mathcal{Q})$ one particular best matching conjunction for Q in $\mathcal{Q}$. It might for example be chosen 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' with a maximal number of edges in the graph for $msg(Q, Q')$.

Definition 3.15. (*splitting*) Let $Q = A_1 \wedge \ldots \wedge A_n$, Q' be conjunctions such that $Q \trianglelefteq Q'$. Let $\mathcal{Q}$ be the set of all ordered subconjunctions Q'' of Q' consisting of n atoms such that $Q \trianglelefteq Q''$. Then $split_Q(Q')$ is the pair (B, R) where $B = bmc(Q, \mathcal{Q})$ and R is the ordered subconjunction of Q' such that $Q' =_r B \wedge R$.

Before presenting a fully concrete abstraction operation (Definition 3.19), we define the $Abs_{\gamma,L}$-operators in Algorithm 3.5 on an intermediate generic level. This will be useful in Section 5 of the paper.

Algorithm 3.16. (*generic abstraction algorithm*) For a global tree γ and a node L in γ define $Abs_{\gamma,L}$ by:

Input: a conjunction Q

Output: a set of conjunctions $\mathcal{Q}_{out}$
Initialisation: $\mathcal{Q}_{out} := \emptyset$; $\mathcal{Q} = partition(Q)$;
while $\mathcal{Q} \neq \emptyset$ **do**
select $M \in \mathcal{Q}$;
$\mathcal{Q} := \mathcal{Q} \backslash \{M\}$;
if $whistle(\gamma, L, M)$ **then**
$\mathcal{Q} := \mathcal{Q} \cup generalise(\gamma, L, M)$
else $\mathcal{Q}_{out} := \mathcal{Q}_{out} \cup \{M\}$;
output $\mathcal{Q}_{out}$

The function *partition* does the initial splitting of the bodies into connected subconjunctions (or mcs's or plain atoms for standard partial deduction). Then for each of the subconjunctions it is checked if there is a risk of non-termination. This is done by the function *whistle*. The whistle will look at the labels (conjunctions) on the branch in the global tree to which the new conjunction M is going to be added as a leaf and if M is "larger" than one of these, it returns true. Finally, if the "whistle blows" for some subconjunction M, then M is generalised by using the function *generalise*.

To obtain a concrete abstraction algorithm, we first choose a concrete whistle, a concrete generalisation, and use $mcs(Q)$ for $partition(Q)$.

Definition 3.17. (*concrete whistle*) For a global tree γ, a node L, and a conjunction M define

$$whistle(\gamma, L, M) = \exists B \in Seq_{\beta_L} : B \trianglelefteq M \wedge B \not\equiv M$$

Definition 3.18. (*concrete generalise*) For a global tree γ, a node L, and a conjunction M define

$$\begin{aligned} &generalise(\gamma, L, M) = \\ &\quad \textbf{let } B \in Seq_{\beta_L} \text{ such that } B \trianglelefteq M \wedge B \not\equiv M \\ &\qquad (M_1, M_2) = split_B(M) \\ &\quad \textbf{in } mcs(msg(M_1, B)) \cup mcs(M_2); \end{aligned}$$

Algorithm 3.19. (*concrete abstraction algorithm*) A concrete abstraction algorithm is defined by Algorithm 3.16 together with the concrete whistle 3.17, the concrete generalisation 3.18 and $mcs(Q)$ for partitioning.

Let us now prove *termination* of Algorithm 3.19 and 3.5.

Proposition 3.20. Algorithm 3.19 terminates. A conjunction $Q \in \mathcal{Q}_{out}$ either does not embed any $B \in Seq_{\beta_L}$, or it is a variant of some such B.

Proof. Upon every iteration, a conjunction is removed from $\mathcal{Q}$, and either replaced by finitely many strictly smaller conjunctions (i.e. with fewer atoms) or is replaced by a conjunction which is strictly more general (i.e. the result of *generalise*).

Let $W = msg(M_1, B)$ in *generalise*. Indeed, if M_2 is not empty or if $mcs(W) \neq W$ then the conjunctions added to $\mathcal{Q}$ will be strictly smaller than the removed M. Alternatively, if M_2 is empty and $mcs(W) = W$ then W must be strictly more general than M. In fact, if W is a variant of M then M must be more general than B (by a property of the *msg*), and even strictly more general because no B is a

variant of M ($B \not\equiv M$). This is in contradiction with the definition of $\trianglelefteq$, which requires that B is not a strict instance of M for $B \trianglelefteq M$ to hold.

As there are no infinite chains of strictly more general expressions (see e.g. [21]), termination follows.

The second part of the proposition follows from the fact that what we want to prove is implied by the negation of *whistle*.

□

Each operator $Abs_{\gamma,L}$ is an abstraction operator in the sense of Definition 3.2, abstracting a singleton $\{Q\}$. It is this property which ensures the existence of a partitioning (and a renaming) such that the output of Algorithm 3.5 leads to a conjunctive partial deduction satisfying the conditions of Theorem 2.19.

So, abstraction according to Algorithm 3.19 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 Higman-Kruskal's Theorem.

Theorem 3.21 (termination). Algorithm 3.5 terminates if U is a terminating unfolding rule and the Abs's are defined as in Algorithm 3.19.

3.4. Refinements of the Algorithm

There are several ways in which the above algorithm can be refined further. At this point, we briefly mention two. Both techniques can (and should) be tuned in such a way as to ensure that the resulting partial deductions are weakly fair.

The simplest technique is as follows: If a conjunction Q' at a leaf in an SLD-tree is a variant (or instance respectively) of a conjunction $Q \in Nodes_{\gamma_i}$, 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.

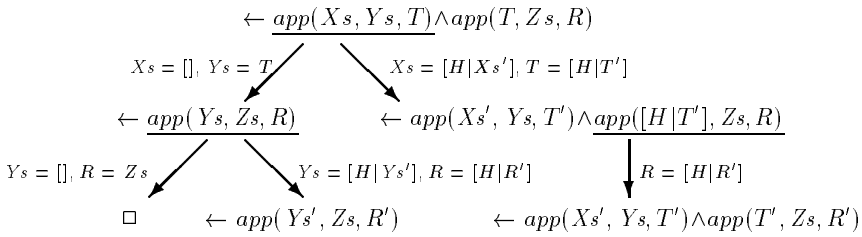
Another technique applies variant (instance) checking in a post-processing phase. At the end of specialisation, it inspects the SLD-trees connected to the conjunctions in $Nodes_\gamma$, and removes from them all subtrees rooted in nodes whose goal body is a variant (instance) of a conjunction in $Nodes_\gamma$. 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*.

3.5. Examples

In this section, we (re)consider examples illustrating optimisations that can be achieved by conjunctive partial deduction. We will, unless explicitly stated otherwise, use Algorithm 3.5 with the concrete strategy formulated in Section 3.3, as well as the *variant check rule* and the *post variant check rule* described in Subsection 3.4.

Double Append

Initially, the global tree contains a single node labelled $app(Xs, Ys, T), app(T, Zs, R)$. Unfolding produces the fair SLD-tree shown below.



Here $app(Ys', Zs, R')$ and $app(Xs', Ys, T') \wedge app(T', Zs, R')$ are the next conjunctions to be considered. 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 the above SLD-tree. 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 conjunctive partial deduction containing clauses C_1, C_2, C_3' and C_4' shown in Example 2.9.

This example also illustrates a point mentioned in Section 3.3: It is even more crucial to use global trees for conjunctive partial deduction than in a conventional context. Indeed, if we run an algorithm based on sets of conjunctions, then $app(Xs', Ys, T') \wedge app(T', Zs, R')$ embeds $app(Ys, Zs, R)$ and abstraction splits $app(Xs', Ys, T') \wedge app(T', Zs, R')$ into two separate atoms. Consequently, no optimisation is obtained.

Rotate-Prune

Consider the rotate-prune program, adopted from [64]:

$$
\begin{array}{l}
rotate(l(N), l(N)). \\
rotate(t(L, N, R), t(L', N, R')) \leftarrow rotate(L, L'), rotate(R, R'). \\
rotate(t(L, N, R), t(R', N, L')) \leftarrow rotate(L, L'), rotate(R, R'). \\
\\
prune(l(N), l(N)). \\
prune(t(L, 0, R), l(0)). \\
prune(t(L, s(N), R), t(L', s(N), R')) \leftarrow prune(L, L'), prune(R, R').
\end{array}
$$

The goal $rotate(T1, T2)$ is true if the trees $T1$ and $T2$ are equal apart from interchanged left and right subtrees in zero or more nodes; $prune(T1, T2)$ holds for a pair of trees where $T2$ can be obtained from $T1$ by replacing each subtree of the latter with label 0 by a leaf labelled 0. Given $T1$, the goal $rotate(T1, U), prune(U, T2)$ first rotates and then prunes $T1$ by means of an intermediate variable U.

Conjunctive partial deduction produces the program below, to be run with the goal $rp(T1, U, T2)$.

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

This result is not entirely satisfactory. Indeed, though no longer used for pruning, the intermediate rotated tree is still constructed. We develop a remedy to this problem in the next section.

4. REDUNDANT ARGUMENT FILTERING

As already noted, conjunctive partial deduction as presented so far, produces in some cases a program that constructs useless intermediate data structures. In this section we describe a simple post-processing phase which in many cases removes such structures.

Reconsider Example 2.9. Conjunctive partial deduction of the *append* program with respect to the goal $\leftarrow app(X, Y, I) \wedge app(I, Z, R)$ yielded the new goal $\leftarrow da(X, Y, I, Z, R)$ and the program $\{C_1, C_2, C_3', C_4'\}$, where

(C_1) $app([\,], L, L) \leftarrow$
(C_2) $app([H|X], Y, [H|Z]) \leftarrow app(X, Y, Z)$
(C_3') $da([\,], Y, Y, Z, R) \leftarrow app(Y, Z, R)$
(C_4') $da([H|X'], Y, [H|I'], Z, [H|R']) \leftarrow da(X', Y, I', Z, R')$

As mentioned in the example, although the result list R now is constructed without reference to the intermediate list I, the latter is still computed. What we want is the goal $\leftarrow da'(X, Y, Z, R)$ and the program $\{C_1, C_2, C_3, C_4\}$, where

(C_3) $da'([\,], Y, Z, R) \leftarrow app(Y, Z, R)$
(C_4) $da'([H|X'], Y, Z, [H|R']) \leftarrow da'(X', Y, Z, R')$

Until now, the step from $da/5$ to $da'/4$ has been left open. It cannot be obtained by the *renaming operation* in [21, 3] which only improves programs in which some atom in some body contains functors or multiple occurrences of the same variable. In fact, this operation has *already* been employed by conjunctive partial deduction to arrive at the program with $da/5$. The step also cannot be obtained by other transformation techniques, such as partial deduction itself, or the *more specific program construction* in [56, 57] which calculates more specific versions of programs. Indeed, any method which preserves the least Herbrand model or the computed answer semantics for all predicates is incapable of transforming $da/5$ to $da'/4$. The point is that although the list I is redundant in some sense—which is made precise below—the change of arity also changes the semantics.

Redundant arguments appear in a variety of other situations, e.g., in programs generated by standard partial deduction with conservative unfolding, and in programs obtained by re-use of general predicates for more specific purposes—see [52].

In this section we rigorously define the notion of a redundant argument. It turns out to be undecidable whether a given argument is redundant, so we present an efficient algorithm which computes a safe approximation of the set of redundant arguments and removes these. Correctness of the technique is also established. The resulting algorithm should then be combined with conjunctive partial deduction proper, removing redundant arguments in a post-processing phase.

4.1. Correct Erasures

Let $Pred(P)$ denote the set of predicates occurring in a logic program P, $arity(p)$ denote the arity of a predicate p, and $Clauses(P)$ denote the set of clauses in P.

Also, for a substitution θ and a set of variables V, $\theta\mid_V$ denotes the restriction of θ to V.

First, we formalise *redundant arguments* in terms of *correct erasures*.

Definition 4.1. (erasure, full erasure) Let P be a program.

1. An *erasure* of P is a set of pairs (p,k) with $p \in Pred(P)$, $1 \leq k \leq arity(p)$.
2. The *full erasure* for P is $\top_P = \{(p,k) \mid p \in Pred(P) \wedge 1 \leq k \leq arity(p)\}$.

The effect of applying an erasure to a program is to erase a number of arguments in every atom in the program. For simplicity we assume that, for every program P and goal G of interest, each predicate symbol occurs only with one particular arity; this prevents unintended name clashes after erasing certain argument positions.

Definition 4.2. Let G be a goal, P a program, and E an erasure of P.

1. For an atom $A = p(t_1,\ldots,t_n)$ in P, let $1 \leq j_1 < \ldots < j_k \leq n$ be all the indexes such that $(p,j_i) \notin E$. We then define $A|E = p(t_{j_1},\ldots,t_{j_k})$.
2. $P|E$ and $G|E$ arise by replacing all atoms A by $A|E$ in P and G, respectively.

How are the semantics of P and $P|E$ of Definition 4.2 related? Since the predicates in P may have more arguments than the corresponding predicates in $P|E$, the two programs have incomparable semantics. Nevertheless, the two programs may have the same semantics for some of their arguments.

Example 4.3. Consider the programs P and $P|E$, where $E = \{(p,3)\}$:

$$\begin{array}{ll} p(0,0,0) & \leftarrow \\ p(s(X),f(Y),g(Z)) & \leftarrow p(X,Y,Z) \end{array} \qquad \begin{array}{ll} p(0,0) & \leftarrow \\ p(s(X),f(Y)) & \leftarrow p(X,Y) \end{array}$$

The goal $G = \leftarrow p(s(s(0)),B,C)$ has exactly one SLD-refutation, with computed answer $\{B/f(f(0)), C/g(g(0))\}$. The goal $G|E = \leftarrow p(s(s(0)),B)$ has exactly one SLD-refutation, with computed answer $\{B/f(f(0))\}$. Thus, although we have erased the third argument of p, the computed answer for the variables in the remaining two arguments is not affected. Taking finite failures into account too, this suggests the following notion of equivalence.

Definition 4.4. (correct erasure) Erasure E is *correct* for program P and goal G iff

1. $P \cup \{G\}$ has an SLD-refutation with computed answer θ with $\theta' = \theta\mid_{vars(G|E)}$ iff $P|E \cup \{G|E\}$ has an SLD-refutation with computed answer θ'.
2. $P \cup \{G\}$ has a finitely failed SLD-tree iff $P|E \cup \{G|E\}$ has.

Given a goal G and a program P, we may now say that the i'th argument of a predicate p is *redundant* if there is an erasure E which is correct for P and G and which contains (p,i). However, we will continue to use the terminology with correct erasures, rather than redundant arguments.

Usually there is a certain set of argument positions I which we do not want to erase. For instance, for $G = app([a],[b],R)$ and the append program, the erasure $E = \{(app,3)\}$ is correct, but applying the erasure will also make the result of the computation invisible. In other words, we wish to retain some arguments because

we are interested in their values. Therefore we only consider subsets of $\top_P \setminus I$ for some I. Not all erasures included in $\top_P \setminus I$ are of course correct, but among the correct ones we will prefer those that remove more arguments. This motivates the following definition.

Definition 4.5. (set of erasures) Let G be a goal, P a program, $\mathcal{E}$ a set of erasures of P, and $E, E' \in \mathcal{E}$.

1. E is *better than* E' iff $E \supseteq E'$.
2. E is *strictly better than* E' iff E is better than E' and $E \neq E'$.
3. E is *best* iff no other $E' \in \mathcal{E}$ is strictly better than E.

Proposition 4.6. Let G be a goal, P a program and $\mathcal{E}$ a collection of erasures of P. Among the correct erasures for P and G in $\mathcal{E}$ there is a best one.

PROOF. There are only finitely many erasures in $\mathcal{E}$ that are correct for P and G. Just choose one which is not contained in any other. □

Best correct erasures are not always unique. For $G = \leftarrow p(1,2)$ and P:

$$\begin{array}{ll} p(3,4) & \leftarrow q \\ q & \leftarrow \end{array}$$

both $\{(p,1)\}$ and $\{(p,2)\}$ are best correct erasures, but $\{(p,1),(p,2)\}$ is incorrect.

4.2. Computing Correct Erasures

Unfortunately, best correct erasures are, in general, uncomputable—for a proof, see [52]. We therefore now introduce the computable approximate notion of a *safe* erasure, which captures many interesting cases. To provide some intuition for this quest, the following examples illustrate some aspects of correctness.

The first example shows what may happen if we try to erase a variable that occurs several times in the body of a clause.

Example 4.7. Consider the programs P and $P|E$, where $E = \{(r,2)\}$:

$$\begin{array}{ll} p(X) & \leftarrow r(X,Y), q(Y) \\ r(X,1) & \leftarrow \\ q(0) & \leftarrow \end{array} \qquad \begin{array}{ll} p(X) & \leftarrow r(X), q(Y) \\ r(X) & \leftarrow \\ q(0) & \leftarrow \end{array}$$

In P the goal $G = \leftarrow p(X)$ fails finitely, while in $P|E$ the goal $G|E = \leftarrow p(X)$ succeeds. Thus E is not correct for P and G. The source of the problem is that the existential variable Y links the calls to r and q with each other. By erasing Y in $\leftarrow r(X,Y)$, we also erase the synchronisation between r and q.

In a similar vein, erasing a variable that occurs several times within the same call, but is not linked to other atoms, can also be problematic.

Example 4.8. Consider the programs P and $P|E$, where $E = \{(p,2)\}$:

$$\begin{array}{ll} p(a,b) & \leftarrow \\ p(f(X),g(X)) & \leftarrow p(Y,Y) \end{array} \qquad \begin{array}{ll} p(a) & \leftarrow \\ p(f(X)) & \leftarrow p(Y) \end{array}$$

Here $G = \leftarrow p(f(X), Z)$ fails finitely in P, while $G|E = \leftarrow p(f(X))$ succeeds (with the empty computed answer) in $P|E$.

Note that, for $E = \{(p,1),(p,2)\}$, $P|E$ is the program:

$$p.$$
$$p \leftarrow p.$$

Again $G|E = \leftarrow p$ succeeds in $P|E$ and the problem arises independently of whether the second occurrence of Y is erased or not.

Still another problem is illustrated in the next example.

Example 4.9. Consider the programs P and $P|E$, where $E = \{(p,2)\}$:

$$\begin{array}{ll} p([],[]) & \leftarrow \\ p([X|Xs],[X|Ys]) & \leftarrow p(Xs,[0|Ys]) \end{array} \qquad \begin{array}{ll} p([]) & \leftarrow \\ p([X|Xs]) & \leftarrow p(Xs) \end{array}$$

In P, the goal $G = \leftarrow p([1,1],Y)$ fails finitely, while in $P|E$ the goal $G|E = \leftarrow p([1,1])$ succeeds. This phenomenon can occur when erased arguments of predicate calls contain non-variable terms.

Finally, problems may arise when erasing in the body of a clause a variable which also occurs in a non-erased position of the head of a clause:

Example 4.10. Consider the programs P and $P|E$, where $E = \{(p,2)\}$:

$$\begin{array}{ll} p(a,b) & \leftarrow \\ p(X,Y) & \leftarrow p(Y,X) \end{array} \qquad \begin{array}{ll} p(a) & \leftarrow \\ p(X) & \leftarrow p(Y) \end{array}$$

Here $G = \leftarrow p(c,Y)$ fails (infinitely) in P while $G|E = \leftarrow p(c)$ succeeds in $P|E$. The synchronisation of the alternating arguments X and Y is lost by the erasure.

The above criteria lead to the following definition, where (1) rules out Example 4.9, (2) rules out Examples 4.7 and 4.8, and (3) rules out Example 4.10.

Definition 4.11. (safe erasure) Let P be a program and E an erasure of P. E is *safe* for P iff for all $(p,k) \in E$ and all $H \leftarrow C, p(t_1,\ldots,t_n), C' \in \mathit{Clauses}(P)$:

1. t_k is a variable X.
2. X occurs only once in $C, p(t_1,\ldots,t_n), C'$.
3. X does not occur in $H|E$.

This in particular applies to goals:

Definition 4.12. (safe goal) Let P be a program and E an erasure of P. E is *safe* for a goal G iff for all $(p,k) \in E$ where $G = \leftarrow C, p(t_1,\ldots,t_n), C'$ it holds that:

1. t_k is a variable X.
2. X occurs only once in in $C, p(t_1,\ldots,t_n), C'$.

The conditions in Definitions 4.11 and 4.12 occur, in a less obvious formulation, within the formalisation of T&S-folding (see e.g. [62]). This will allow us to reuse correctness results from the unfold/fold literature in the proof below. Indeed, the method of this section can be seen as a novel application of T&S-folding using a particular control strategy.

Proposition 4.13. Let G be a goal, P a program, and E an erasure of P. If E is safe for P and for G then E is correct for P and G.

PROOF. We will show that $P|E$ can be obtained from P by a sequence of T&S-definition, unfolding and folding steps (see also Section 2.4).

Let $P_0 = P \cup query(\bar{X}) \leftarrow Q$ be the initial program of our transformation sequence, where $G = \leftarrow Q$ and $\bar{X}$ is the sequence of distinct variables occurring in $G|E$. First, for each predicate defined in P such that $A \neq A|E$, where $A = p(X_1, \ldots, X_n)$ is a maximally general atom, we introduce the definition $Def_p = A|E \leftarrow A$. The predicate of A occurs in P_0, and is therefore old according to the definitions in [74] (if one wants to use the definitions in [62] one can use exactly the same "trick" as explained in the proof of Lemma 1). By the conditions we imposed earlier on P we also know that the predicate of $P|E$ does not occur in P_0. Thus these definition steps are T&S-definition introduction steps.

We now unfold every definition $A|E \leftarrow A$ wrt A using the clauses defining A in P_0, giving us the program P_k (where k is the number of definitions that have been unfolded).

For every atom $p(t_1, \ldots, t_n)$ in the body of a clause C of P_k, for which a definition Def_p has been introduced earlier, we perform a folding step of C wrt $p(t_1, \ldots, t_n)$ using Def_p. Note that every such atom $p(t_1, \ldots, t_n)$ is fold-allowing (because either it has been obtained by unfolding a definition $A|E \leftarrow A$ and is not inherited from A or it stems from the original program). The result of the folding step is that of replacing $p(t_1, \ldots, t_n)$ by $p(t_1, \ldots, t_n)|E$. This means that after having performed all the resolution steps we obtain a program $P' = P|E \cup query(\bar{X}) \leftarrow Q|E \cup P''$ where P'' are the original definitions of those predicates for which we have introduced a definition Def_p.

Now, as already mentioned earlier, the conditions in Definition 4.11 and Definition 4.12 are equivalent to the conditions of T&S-folding and therefore P' can be obtained from P_k by a sequence of T&S-unfolding and then T&S-folding steps on fold-allowable atoms. Note that here it is vital to define $\bar{X}$ to be the variables of $G|E$ in the clause $query(\bar{X}) \leftarrow Q$ of P_0, otherwise the folding steps performed on the atoms of Q would not be T&S-folding steps. We can thus apply Theorems 10 and 12 in [62] to deduce preservation of the computed answers and of finite failure.

Finally, as P'' is unreachable from $\leftarrow query(\bar{X})$ we can remove P'' and because $G|E = \leftarrow Q|E$, the conditions of Definition 4.4 are verified for P and G. □

The following algorithm constructs a safe erasure for a given program.

Algorithm 4.14. (*RAF*)

Input: a program P, an initial erasure E_0.
Output: an erasure E with $E \subseteq E_0$.
Initialisation: $i := 0$;
while there exists a $(p, k) \in E_i$ and a $H \leftarrow C, p(t_1, \ldots, t_n), C' \in Clauses(P)$ s.t.:

1. t_k is not a variable; **or**
2. t_k is a variable that occurs more than once in $C, p(t_1, \ldots, t_n), C'$; **or**
3. t_k is a variable that occurs in $H|E_i$

do $E_{i+1} := E_i \setminus \{(p, k)\}$; $i := i + 1$;
return E_i

The above algorithm starts out from an initial erasure E_0 contained in $\top_P \setminus I$, where I are positions of interest (i.e. we are interested in the computed answers

they yield). Furthermore E_0 should be safe for any goal of interest. Concretely, E_0 is usually taken equal to $\top_P \setminus I$ with I the argument positions of the top=level goal with respect to which the program is specialised (cf.Examples 4.16 and 4.17 below).

Proposition 4.15. With input E_0, RAF terminates, and output E is a unique erasure, which is the best safe erasure for P contained in E_0.

PROOF. The proof consists of four parts: *termination* of RAF, *safety* of E for P, *uniqueness* of E, and *optimality* of E. The two first parts are obvious; termination follows from the fact that each iteration of the while loop decreases the size of E_i, and safety is immediate from the definition.

To prove uniqueness, note that the non-determinism in the algorithm is the choice of which (p,k) to erase in the while loop. Given a logic program P, let the *reduction* $E(p,k)F$ denote the fact that E is not safe for P and that an iteration of the while loop may choose to erase (p,k) from E yielding $F = E\setminus\{(p,k)\}$.

Now suppose $E(p,k)F$ and $E(q,j)G$. Then by analysis of all the combinations of reasons that (p,k) and (q,j) could be removed from E it follows that $F(q,j)H$ and $G(p,k)H$ with $H = E\setminus\{(p,k),(q,j)\}$. This property implies that for any two sequences

$$E(p_1,k_1)F_1\ldots F_{n-1}(p_n,k_n)F_n \quad \text{and} \quad E(q_1,j_1)G_1\ldots G_{m-1}(q_m,j_m)G_m$$

there are sequences:

$$F_n(q_1,j_1)G'_1\ldots G'_{m-1}(q_m,j_m)H \quad \text{and} \quad G_m(p_1,k_1)F'_1\ldots F'_{n-1}(p_n,k_n)H$$

with $H = F_n \cap G_m$. In particular, if F_n and G_m are safe, so that no reductions apply, it follows that $F_n = G_m$. Hence the output is a unique erasure.

To see that this is the best one among the safe erasures contained in E_0, note that $E(p,k)F$ implies that no safe erasure contained in E contains (p,k). □

Example 4.16. In the *append* example, we augment the original program with the clause $dapp(X,Y,Z,R) \leftarrow app(X,Y,I), app(I,Z,R)$ and subsequently run conjunctive partial deduction as before. We obtain $dapp(X,Y,Z,R) \leftarrow da(X,Y,I,Z,R)$ and the program $\{C_1, C_2, C'_3, C'_4\}$ as on page 260. Application of RAF, starting from $E_0 = \top_P \setminus \{(dapp,i) \mid 1 \leq i \leq 4\}$ (stating the fact that we are only interested in queries to *dapp*), now yields $E = \{(da,3)\}$, showing that the third argument of *da* can be safely removed.

Example 4.17. For the rotate-prune example in Section 3.5, with a similar top-level clause containing the rotate-prune query, RAF generates $E = \{(rp,2),(p,2)\}$ and hence the program obtained before will be further transformed into:

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

to be run with the goal $rp(T_1,T_2)$. This program completely avoids construction of the intermediate rotated tree. It is equivalent to what unfold/fold transformations can obtain [64].

More about redundant argument filtering can be found in [52], including a polyvariant version of RAF allowing different erasures to be applied in different contexts, and a variant of RAF (named FAR), detecting further superfluous arguments. The paper also contains a series of benchmark results exhibiting an average speed increase of 18% and an average code size reduction of 21%, when RAF is run after conjunctive partial deduction (where the initial erasure E_0 contained all argument positions except those of the top-level query to be specialised), as compared to running conjunctive partial deduction alone.

Work very much related to RAF is [15], which provides some pragmatics for removing unnecessary variables in the context of optimising binarised Prolog programs. Another related technique is truncation of Prolog programs derived by extended execution [19]. In some cases truncation is more powerful than RAF, but in order to apply it (soundly), one has to prove termination of the runtime goal and functionality of the goal to be truncated. Techniques similar to RAF have also appeared in functional programming, e.g. Chin [13] describes a technique to remove *useless variables* using abstract interpretation. Compared to these techniques the algorithm for redundant argument filtering (RAF) is strikingly simple and very efficient. The obvious drawback of our technique is that it is less precise. Nevertheless, the mentioned benchmarks show that it performs well on a range of mechanically generated programs, indicating a good trade-off between complexity and precision.

It would seem that our algorithm RAF for removal of redundant arguments is related to Proietti and Pettorossi's work on unfold/fold transformations for removal of *unnecessary variables* (see e.g.[64]). However, the two should not be directly compared. RAF is intended as a simple, efficient post-processing phase for program transformers, in particular for conjunctive partial deduction, whereas the unfold/fold approach is less efficient, but far more powerful and able to remove intermediate data structures from programs. For instance, it can produce the desired versions of the double-append rotate-prune programs in Examples 4.16 and 4.17. Very roughly, whereas unfold/fold eliminates the production and subsequent consumption of intermediate data structures, conjunctive partial deduction only eliminates the consumption and RAF then removes their production. Thus, one should rather compare unfold/fold to the composition of conjunctive partial deduction with RAF. We discuss this further in Section 6.

5. CONJUNCTIVE PARTIAL DEDUCTION FOR PURE PROLOG

In this section, we show how conjunctive partial deduction can be used to transform pure Prolog programs. We describe different options for the design and the implementation of a conjunctive partial deduction system and discuss several control problems that have to be solved in practice. After discussing control in the Prolog context (Section 5.1), we survey the different methods we tested (Section 5.2), and discuss experimental results (Section 5.3).

5.1. Control in Pure Prolog

We will be concerned with conjunctive partial deduction for a declarative subset of Prolog. This means, beside omitting non-pure language features, that we suppose a fixed (unfair) computation rule. Moreover, we will demand preservation of program

termination under the given computation rule (in the sequel assumed "left-to-right", unless explicitly stated otherwise).

Unfolding Rules

In the given context, *determinate unfolding* has been proposed as a way to ensure that partial deduction will never actually worsen the behaviour of the program [23, 22]. Indeed, even fairly simple examples suffice to show that *non-leftmost, non-determinate unfolding* may duplicate computations in the resulting programs. *Leftmost, non-determinate unfolding*, usually allowed to compensate for the all too cautious nature of purely determinate unfolding, avoids the more drastic deterioration pitfalls, but can still lead to multiplying unifications.

Splitting and Abstraction

Contiguous splitting. Abstraction, as presented in Section 3, splits a conjunction into subconjunctions. However, these subconjunctions are not necessarily *contiguous*. Let us present a simple example. Consider the two conjunctions Q_1 and Q_2:

$$Q_1 = p(X,Y) \wedge q(Y,Z)$$

$$Q_2 = p(f(X),Y) \wedge r(Z,R) \wedge q(Y,Z)$$

If specialisation of Q_1 leads to specialisation of Q_2, there is a danger of non-termination ($Q_1 \trianglelefteq Q_2$). The method presented in Section 3 will remedy this by first splitting Q_2 into $Q = p(f(X),Y) \wedge q(Y,Z)$ and $r(Z,R)$ and subsequently taking the msg of Q_1 and Q. As a result, only $r(Z,R)$ will be considered for further specialisation.

Now, given a left-to-right computation rule, the above operation alters the sequence in which goals are executed. Indeed, the p- and q-subgoals will henceforth be treated jointly (they will probably be renamed to a single atom). Consequently, there is no way an r-call can be interposed.

From a purely declarative point of view, there is, of course, no reason why goals should not be interchanged, but under a fixed (unfair) computation rule, non-contiguous splitting can degrade program performance, and even change the termination behaviour of a program.

In fact, the latter point has already been addressed in the context of unfold/fold transformations (e.g. [7, 6, 9, 10]). To the best of our knowledge, however, no satisfactory solution exists, suitable to be incorporated in a fully automatic transformation system. Thus, below we have in all but two methods *limited splitting to be contiguous*, that is, we split into contiguous subconjunctions only. (This can be compared with the outruling of goal switching in [6].) On the one hand, compared to the basic (declarative) method in Section 3, some opportunities for program improvements are not exploited, on the other hand, Prolog programs are significantly less prone to actual deterioration rather than optimisation.

Static conjunctions. Even though abstraction (splitting) ensures that the length of conjunctions (the number of atoms) remains finite, there are (realistic) examples

where conjunctions can get very large. This, combined with the use of homeomorphic embeddings (or lexicographical orderings for that matter), can lead to very large global trees, large residual programs and degrade transformation time complexity.

A simple way to avoid this practical problem, is to use another, less explosive strategy on conjunctions, e.g. requiring a decrease in the total term size. Another way is to limit the size of conjunctions at the global level using static conjunctions. A *static conjunction* is any conjunction or a generalisation of any conjunction that can be obtained by non-recursive unfolding of the goal to be partially evaluated. A static analysis can be used to compute a set of static conjunctions $\mathcal{S}$ from the program and the goal. During partial deduction only those conjunctions will be allowed (at the global level) that are instances of an element of $\mathcal{S}$; any disallowed conjunctions that may occur are split further. (A related technique is used in [65].)

In our implementation, we use a simple-minded method of approximating the set of static conjunctions, based on counting the maximum number of occurrences of each predicate symbol in a conjunction in the program or in the goal to be partially deduced, and disallowing conjunctions surpassing these numbers.

5.2. The System and the Implemented Methods

The partial deduction system which we used to investigate the effects of conjunctive partial deduction is the ECCE partial deduction system (developed by Leuschel [53]). The system consists of an implementation of the concrete algorithm 3.5 to which one may add one's own methods for unfolding, partitioning, generalisation, etc. All built-ins handled by the system are supposed to be declarative (e.g. `ground` is supposed to be delayed until its argument is ground,...). Some of the built-ins that are handled are: =, `is`, <, =<, <, >=, `nonvar`, `ground`, `number`, `atomic`, `call`, \==, \=. In the following we will give a short description of the different methods that we used in the experiments.

The system implements a variant of the concrete algorithm described in Section 3.3. The algorithm uses a global tree γ with nodes labelled with (characteristic) conjunctions. When a conjunction Q gets unfolded, then the conjunctions in the bodies of the resultants of Q (maybe further split by the abstraction) are added as child nodes (leaves) of Q in the global tree.

After the algorithm terminates the residual program is obtained from the output by unfolding and renaming (details can be found in [48, 26, 52]).

The Concrete Settings

We have concentrated on three local unfolding rules for U. All unfolding rules were complemented by a simple more specific transformation in the style of SP [22] and allow the selection of ground negative literals.

1. *Safe determinate* (t-det.): do determinate unfolding allowing one left-most non-determinate step using homeomorphic embedding with covering ancestors of selected atoms to ensure finiteness.

2. *Homeomorphic embedding and reduction of search space* (h-rs): non-left-most unfolding is allowed if the search space is reduced by the unfolding. In

other words, an atom $p(\bar{t})$ can be selected if it does not match all the clauses defining p. Again, homeomorphic embeddings are used to ensure finiteness. Note that, in contrast to 2 and 3, this method might worsen the backtracking behaviour.

3. *"MIXTUS"-like unfolding* (x): See [67] for further details (we used $max_rec = 2$, $max_depth = 2$, $maxfinite = 7$, $maxnondeterm = 10$ and only allowed non-determinate unfolding when no user predicates were to the left of the selected literal).

The measures that we have used in whistles are the following:

1. homeomorphic embedding (homeo.) on the conjunctions
2. termsize (i.e. the number of function symbols in the terms) on the conjunctions
3. homeomorphic embedding (homeo.) on the conjunctions and homeomorphic embedding on the associated characteristic trees
4. termsize on the conjunctions and homeomorphic embedding on the characteristic trees

The methods for partitioning are based either on splitting into mcs's (non-contiguous) or into maximal contiguous connected subconjunctions. Additionally we may limit the size of conjunctions by using static conjunctions.

An extension wrt [40, 50] relates to built-ins which are also registered in the characteristic tree. The only problematic aspect is that, when generalising built-ins which generate bindings (like is/2, =../2) and which are no longer executable after generalisation, these built-ins have to be removed from the generalised characteristic tree (i.e. they are no longer selected).

5.3. Results and Discussion

We incorporated the methods into the ECCE partial deduction system [53, 40, 50] and ran an extensive set of benchmarks. We will now discuss the resulting speedups, the transformation time and the code size. Also, we shall compare our results to standard partial deduction with ECCE and to three other partial deduction systems.

Systems

Table 1 gives an overview of the tested partial deduction systems. They fall into three categories:

- *Conjunctive partial deduction.* All systems use safe, contiguous splitting (contig), except two systems that use an unsafe, non-contiguous variant (mcs). The first two systems, marked (dynamic), do not use static conjunctions (static) to limit the size of conjunctions at the global level. On the global control level, we investigate the effect of homeomorphic embeddings (homeo) and termsize measures (termsize). Optionally, we employ characteristic trees (chtree) with homeomorphic embedding, marked (none) if unused. Local unfolding is always determinate (t-det), except in one case (h-rs).

- *Standard partial deduction.* Disallowing conjunctions on the global level gives us conventional partial deduction. We tested standard partial deduction with three different unfolding rules. One system (SE-hh-x) uses the ecological partial deduction principle [40] to ensure preservation of characteristic trees upon generalisation.
- *Existing systems.* We compare our results with those produced by three existing systems based on standard partial deduction: MIXTUS [67], PADDY [63], and SP [22, 24]. The following versions of these systems have been used: version 0.3.3 of MIXTUS, the version of PADDY delivered with ECLIPSE 3.5.1, and a version of SP dating from September 25th, 1995.

All ECCE-based systems use the same post-processor which performs redundant argument filtering, determinate post-unfolding, and removal of unnecessary polyvariance [50].

Benchmarks

We used a set of small and medium sized benchmark programs taken from [53]. The benchmark programs were carefully selected and/or designed in such a way that they cover a wide range of different applications, including: pattern matching, databases, expert systems, meta-interpreters (non-ground vanilla, mixed, ground), as well as more involved ones: a model-elimination theorem prover, the missionaries-cannibals problem, a meta-interpreter for a simple imperative language. A few benchmarks can be fully unfolded. Detailed descriptions can be found in [53, 41].

Together, we claim, the benchmarks give a good impression of the specialisations and transformations obtained by the different systems.

The entry TT in Table 1 is the total transformation time in minutes to transform all the benchmarks. The entry $> 12h$ means that the specialisation was interrupted after 12 hours (though, theoretically, it should have terminated by itself when granted sufficient time to do so).

We briefly explain the use of ∞ in the tables:

∞, SP: this means real non-termination
∞, MIXTUS: heap overflow after 20 minutes
∞, PADDY: thorough system crash after 2 minutes

Results

The results are summarised in Table 2. We adopted a practical approach and measured the execution time and the size of compiled code of the specialised programs.

The timings were obtained via special Prolog files which call the original and specialised programs directly and at least 100 times for the respective run-time queries, using the $time/2$ predicate of Prolog by BIM 4.0.12 on a Sparc Classic under Solaris.

The second column contains the total speedup for all benchmarks:

$$\frac{n}{\sum_{i=1}^{n} \frac{spec_i}{orig_i}}$$

where n is the number of benchmarks and $spec_i$ and $orig_i$ are the absolute execution times of the specialized and original programs respectively. The weighted speedups

System	Partition		Whistle		Unf	Total
	contig	s/d	conj	chtree		TT (min)
Conjunctive Partial Deduction						
Cdc-hh-t	contig	dyn	homeo	homeo	t-det	62.46
Cdc-th-t	contig	dyn	termsize	homeo	t-det	31.18
Csc-hh-t	contig	static	homeo	homeo	t-det	29.72
Csc-th-t	contig	static	termsize	homeo	t-det	5.95
Csc-hn-t	contig	static	homeo	none	t-det	35.49
Csc-tn-t	contig	static	termsize	none	t-det	2.67
Cdm-hh-t	mcs	dyn	homeo	homeo	t-det	$> 12h + 110.49$
Csm-hh-h	mcs	static	homeo	homeo	h-rs	$> 12h + 73.55$
Standard Partial Deduction						
S-hh-t	-	-	homeo	homeo	t-det	3.00
SE-hh-x	-	-	homeo	homeo	mixtus	2.96
Existing Systems						
MIXTUS	-	-	mixtus	none	mixtus	$\infty + 2.71$
PADDY	-	-	mixtus	none	mixtus	$\infty + 0.31$
SP	-	-	pred =	=	det ?	$3^{*}\infty + 1.99$

TABLE 1. Overview: systems and transformation times

are obtained by using the code sizes $size_i$ of the original programs as a weight for computing the average.

The fourth column contains the total speedup for those benchmarks which are "fully unfoldable" (i.e. those for which normal evaluation terminates) while the fifth column contains the total speedup for those benchmarks which are not "fully unfoldable".

The last column of Table 2, finally, contains the average of the relative code size $specsize_i/size_i$, where $specsize_i$ are the code sizes of the specialised programs.

5.4. Discussion of Results

The experiments show that conjunctive partial deduction (using determinate unfolding and contiguous splitting) pays off compared to standard partial deduction and existing systems.

On the fully unfoldable benchmarks, standard partial deduction S-hh-t gave a speedup of 2.57 while conjunctive partial deduction Csc-hh-t achieved a speedup of 5.90, which shows that conjunctive partial deduction diminishes the need for aggressive unfolding. Notice that Mixtus and Paddy have very aggressive unfolding rules and fare well on the fully unfoldable benchmarks. However, on the non-fully unfoldable ones, even standard partial deduction S-hh-t, based on determinate unfolding, is already better. The best standard partial deduction method, for both runtime and (apart from SP) code size, is standard partial deduction SE-hh-x. Still, compared to any of the standard partial deduction methods, our conjunctive methods (except for Csm-hh-h and Csc-tn-t, which are not meant to be competitors anyway) have a significantly better average speedup.

The experiments also show that conjunctive partial deduction can be made efficient, especially if one uses determinate unfolding combined with a termsize measure on conjunctions (Csc-th-t and Csc-tn-t) in which case the average transformation time is comparable with that of standard partial deduction. Of course only further

System	Total Speedup	Weighted Speedup	Fully Unfoldable Speedup	Not Fully Unfoldable Speedup	Average Relative Size (orig = 1)
Conjunctive Partial Deduction					
Cdc-hh-t	1.93	2.44	5.90	1.66	2.39
Cdc-th-t	1.96	2.49	5.90	1.69	2.27
Csc-hh-t	1.89	2.38	5.90	1.62	2.02
Csc-th-t	1.92	2.44	5.90	1.65	1.68
Csc-hn-t	1.89	2.40	5.90	1.62	1.67
Csc-tn-t	1.76	2.18	4.48	1.54	1.53
Cdm-hh-t	2.00	2.39	5.90	1.72	3.17
Csm-hh-h	0.77	0.52	6.16	0.63	3.91
Standard Partial Deduction					
S-hh-t	1.56	1.86	2.57	1.42	1.60
SE-hh-x	1.76	2.24	8.36	1.48	1.46
Existing Systems					
MIXTUS	1.65	2.11	8.13	1.38	1.67
PADDY	1.65	2.00	8.12	1.38	2.49
SP	1.34	1.54	2.08	1.23	1.18

TABLE 2. Summary of benchmarks (higher speedup and lower code size is better)

experiments may show how the transformation times grow with the size of programs. In fact, the system was not written with efficiency as a first concern and there is a lot of room for improvement on this point.

Static Conjunctions. Comparing Csc-hh-t and Cdc-hh-t, one can see that using static conjunctions pays off in terms of shorter transformation time without much loss of specialisation. Examining the detailed results for static/dynamic conjunctions [31] shows that the speedup and the transformation times are almost identical except for a few cases where static conjunctions were needed.

Termsize. The experiments demonstrate that using the termsize measure instead of homeomorphic embedding on conjunctions clearly improves the average transformation time without loosing too much specialisation. But they also show that *if* one uses the termsize measure *then* the use of characteristic trees becomes vital (compare Csc-th-t and Csc-tn-t). However, methods with homeomorphic embedding on conjunctions (e.g. Csc-hn-t), do not seem to benefit from adding homeomorphic embedding on characteristic trees as well (e.g. Csc-hh-t).

This, at first sight somewhat surprising phenomenon, can be explained by the fact that, for the benchmarks at hand, the homeomorphic embedding on conjunctions, in a global tree setting, is already a very generous whistle, and, in the absence of negation (see the discussions in [50]), a growing of the conjunction will often result in a growing of the characteristic tree as well.

"MIXTUS"-like Unfolding. For standard partial deduction "MIXTUS"-like unfolding leads to definite improvement over determinate unfolding. Note that the "MIXTUS"-like unfolding used by SE-hh-x does not seem to pay off for conjunctive partial deduction at all. In a preliminary experiment, the method Csc-th-x only produced a total speedup of 1.69, i.e. only slightly better than MIXTUS or PADDY and worse

than SE-hh-x. In future work we will examine how more aggressive unfolding rules can be more successfully used for conjunctive partial deduction.

Non-Determinate Unfolding. For some benchmarks, the best speedup is obtained by the non-safe methods Cdm-hh-t or Csm-hh-h based on non-contiguous mcs splitting. But in some cases, these methods indeed lead to a considerable slowdown for reasons explained earlier. This shows that methods based on non-contiguous splitting can lead to better specialisation due to *tupling* and *deforestation*, but that we need some method to control the splitting and unfolding to ensure that no slowdown, or change in termination can occur.

Conclusion

From the results, we can conclude that conjunctive partial deduction indeed pays off for a wide range of applications, but there are still a number of open problems that need to be addressed in practice. Indeed, the speedups compared to standard partial deduction are significant but less dramatic than initially expected. This is due to the fact that non-contiguous conjunctive partial deduction on the one hand often leads to substantial slowdowns and is not really practical for most applications, while contiguous conjunctive partial deduction on the other hand is in general too weak to deforest or tuple data structures.

Therefore it is vital, if one wants to more heavily exploit the advantages of conjunctive partial deduction, to add non-contiguous splitting (i.e. reordering) in a safe way which guarantees no serious slowdown. A first step towards a solution is presented in [9], but it remains quite restrictive and considers only ground queries. Another, more pragmatic approach might be based on making use of some mode system to allow reordering of literals as long as the resulting conjunction remains well-moded. This would be very similar to the way in which the compiler for Mercury [71] reorders literals to create different modes for the same predicate. For the semantics of Mercury any well-moded re-ordering of the literals is allowed. Although this approach does not ensure the preservation of termination, it is then simply considered a programming error if one well-moded query terminates while the other does not.

6. RELATED WORK AND DISCUSSION

Conjunctive partial deduction is strongly related to both its conventional precursor and unfold/fold. In the previous section, we compared an implementation of our approach to existing, standard partial deduction systems. We now discuss the relation with unfold/fold.

First, it should be noted that an experimental comparison, of the type presented in Section 5, is not possible. To our knowledge, no automatic unfold/fold systems are available for experimentation.

However, some explicit strategies for unfold/fold transformation have been proposed. Let us consider some of the most well-known of these: loop absorption and generalisation (LAG) [65] and unfold-definition-fold (UDF) [64] (see also [62]). We take the liberty of distinguishing between local and global components of the strategies in both unfold/fold and partial deduction. No such division is explicitly

present in the former, but as they have become standard in the latter, examining their counterparts in unfold/fold provides us with a useful angle for comparison.

On the level of local control, both LAG and UDF use a class of computation rules, called *synchronised descent rules*. This class formalises a heuristic tuned towards foldability (and therefore, indirectly, termination) and the generation of optimal transformed programs. However, no specific instance of this class has been fixed, so that no specific algorithm can be subjected to experimentation. Also, synchronised descent rules do not guarantee termination in general. Instead, classes of programs are identified for which termination is ensured. In the context of partial deduction, a much broader range of local control techniques has been examined. Many of these are based on formal mathematical notions, such as well-founded and well quasi orders and homeomorphic embedding, guaranteeing termination for *all* programs. Concrete systems have made explicit choices, thus allowing experimental comparisons and optimisation.

On the level of global control, one finds to some extent a similar situation. In [65], a class of different generalisation heuristics is presented. The global control component in UDF is not easily isolated, but the overall picture corresponds to the one for LAG. None of them guarantees termination in general, but again classes of programs are identified for which they do. In partial deduction, generalisation methods — based on well-founded orders and homeomorphic embeddings — have been proved to secure termination for *all* programs. Moreover, notions capturing the specialisation behaviour, such as characteristic trees, have been shown instrumental in providing maximally precise generalisation. This level of technical detail has allowed implementation, experimental evaluation and further improvements.

Concluding, we can discern a clear *methodological distinction* between work in unfold/fold and (conjunctive) partial deduction.

Let us now turn to the issue of *transformational power*, concentrating on comparing unfold/fold with conjunctive partial deduction (enhanced with RAF post-processing). First, observe that our approach does not include goal replacement. So, in general one cannot expect that it can handle transformations requiring goal replacement. The non-linear optimisation of the Fibonacci program through factoring on functional predicates in [64] provides such an example.[7] Automation of goal replacement, however, is notoriously difficult (correctness of goal replacement is undecidable in general).

We conjecture that conjunctive partial deduction with polyvariant RAF is comparable in power to unfold/fold without goal replacement. In fact, for all practical examples examined so far, the possibility to unfold or fold wrt *any* prior program in a transformation sequence does not seem to add any power over the more restricted approach employed by conjunctive partial deduction. But, obviously, as our conjunctive partial deduction *algorithms* terminate for *any* program they will sometimes produce sub-optimal results. A task for future work will be to identify, as in unfold/fold, specific classes of programs for which our correctness and termination conditions can be relaxed and optimal results guaranteed.

[7] Although the ECCE system *can* achieve this optimisation automatically, when it is given mode and determinism declarations. Also see [47, 42], which combines conjunctive partial deduction with abstract interpretation, resulting in a method which is then able to infer (and exploit) functionality.

7. CONCLUSION

We perceive standard partial deduction as a stream of work within the overall unfold/fold area. It has restricted its attention to a much less powerful, but more easily manageable subset of transformations and, as a result, has produced fully automatic, practical, terminating unfold/fold based systems. No systems for full unfold/fold were obtained featuring a comparable level of automation.

The main contribution of this paper therefore lies in showing how fairly small enhancements to standard partial deduction technology substantially boost its transformational power so as to cover a much larger class of unfold/fold transformations. These extensions are: transforming conjunctions instead of atoms and supporting more general renaming schemes. In doing so, we have been able to rely on the extensive work in automatic control of standard partial deduction. This constitutes our main success: We have shown how existing techniques can be made much more powerful, with fairly little effort.

Of course, we could have presented our approach in an unfold/fold style rather than a partial deduction one. To some extent, the choice is just a matter of individual taste and preference. However, since we aimed for fully automatic control guaranteeing termination on all programs, the partial deduction setting, with its traditional focus on these issues, offers considerable advantages.

Let us finally summarise the main achievements of the article. Conjunctive partial deduction was designed with the aim of overcoming some limitations inherent in conventional partial deduction. The main contribution of our work therefore lies in showing how minimal enhancements to standard partial deduction technology substantially boost its transformational power so as to cover a much larger class of unfold/fold transformations. We presented powerful extensions, showed that they can perform tupling and deforestation, and proved a correctness result similar to the one of standard partial deduction. We provided a basis for the design of concrete algorithms within this extended framework by introducing a basic algorithm for conjunctive partial deduction and refining this algorithm into a fully automatic one. Conjunctive partial deduction was put on trial, and extensive experiments conducted with a prototype confirmed that many techniques developed for standard partial deduction carry over and that the additional power actually pays off in practice. They give a good impression of specialisation and transformation obtained by various methods on a declarative subset of Prolog.

Acknowledgements

We would like to thank Annalisa Bossi, André de Waal, John Gallagher, Fergus Henderson, Jan Hric, Robert Kowalski, Torben Mogensen, Alberto Pettorossi, Maurizio Proietti, Thomas Reps, Dan Sahlin, and Zoltan Somogyi for valuable discussions on different aspects of this work. We also thank anonymous referees, of the present paper and of earlier work presented at JICSLP'96, PLILP'96 and LOPSTR'96, for their comments.

Danny De Schreye is a Senior Research Associate of the Belgian National Fund for Scientific Research. Michael Leuschel and Bern Martens were supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation" and Michael Leuschel is now a lecturer at the Department of Electronics and Computer Science, University of Southampton. Support was also provided by the project "Design,

Analysis and Reasoning about Tools" funded by the Danish Natural Sciences Research Council.

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. C. Aravindan, P.M. Dung. On the correctness of unfold/fold transformation of normal and extended logic programs. *Journal of Logic Programming*, 24(3):201–217, 1995.
3. K. Benkerimi, P.M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, 1993.
4. R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.
5. A. Bossi, N. Cocco, S. Dulli. A method for specialising logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.
6. A. Bossi, N. Cocco. Preserving Universal Termination through Unfold/Fold. In G. Levi, M. Rodriguez-Artalejo (eds.), *Proc. 4th International Conference on Algebraic and Logic Programming*, LNCS 850, 269–286, Springer-Verlag, 1994.
7. A. Bossi, S. Etalle. Transforming Acyclic Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1081–1096,1994.
8. A. Bossi, S. Etalle. More on unfold/fold transformations of normal programs: Preservation of Fitting's semantics. In L. Fribourg, F. Turini (eds.), Logic Program Synthesis and Transformation—Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, LNCS 883, 311–331, Springer-Verlag, 1994.
9. A. Bossi, N. Cocco, S. Etalle. Transformation of Left Terminating Programs: The Reordering Problem. In M. Proietti (ed.), Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, 33–45, Springer-Verlag, 1995.
10. A. Bossi, N. Cocco. Replacement can Preserve Termination. In J. Gallagher (ed.), *Pre-Proceedings of LOPSTR'96*, 78–91, Stockholm, Sweden, August 1996.
11. 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.
12. R.M. Burstall, J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
13. W.-N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, 1990.
14. O. Danvy, R. Glück, P. Thiemann (eds.). *Partial Evaluation*. LNCS 1110, Springer-Verlag, 1996.
15. B. Demoen. On the transformation of a Prolog program to a more efficient binary program. In [38], 242–252, 1993.
16. D. De Schreye, M. Leuschel, B. Martens. Program specialisation for logic programs. Tutorial. Abstract in J. Lloyd (ed.), *Proceedings ILPS'95*, 615–616, MIT Press, 1995.
17. N. Dershowitz, J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, 244–320, Elsevier, 1992.
18. D.A. de Waal, J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy (ed.), *Automated Deduction—CADE-12*, 207–221, Springer-Verlag, 1994.
19. L. Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. *Proceedings ICLP'90*, 685–699, MIT Press, 1990.
20. Y. Futamura. Partial evaluation of a computation process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

21. J. Gallagher, M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe (ed.), *Proceedings Meta'90*, 229–244, Leuven, Belgium, 1990.
22. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, 1991.
23. J. Gallagher, M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
24. J. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 88–98, ACM Press, 1993.
25. P. A. Gardner, J. C. Shepherdson. Unfold/fold transformations in logic programs. In J.-L. Lassez, G. Plotkin (eds.), *Computational Logic, Essays in Honor of Alan Robinson*, 565–583, MIT Press, 1991.
26. R. Glück, J. Jørgensen, B. Martens, M.H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen, S.D. Swierstra (eds.), *Programming Language Implementation and Logic Programming. Proceedings*, LNCS 1140, 152–166, Springer-Verlag, 1996.
1996.
27. C.A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, Department of Computer Science, University of Bristol, 1994.
28. G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 2:326–336, 1952.
29. N.D. Jones, C.K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
30. J. Jørgensen, M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In [14], 238–262, 1996.
31. J. Jørgensen, M. Leuschel, B. Martens. Conjunctive partial deduction in practice. In John Gallager (ed.), *Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96*, LNCS 1207, 59–82, Stockholm, Sweden, August 1996.
32. T. Kanamori, K. Horiuichi. Construction of logic programs based on generalized unfold/fold rules. In J.-L. Lassez (ed.), *Proceedings of the 4th International Conference on Logic Programming*, 744–768, MIT Press, 1987.
33. 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.
34. J. Komorowski. An introduction to partial deduction. In A. Pettorossi (ed.), *Proceedings Meta'92*, LNCS 649, 49–69, Springer-Verlag, 1992.
35. J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.
36. A. Lakhotia. To PE or not to PE. In M. Bruynooghe (ed.), *Proceedings of Meta90 Workshop on Meta Programming in Logic*, 218–228, Leuven, Belgium, 1990.
37. A. Lakhotia, L. Sterling. How to control unfolding when specializing interpreters. *New Generation Computing*, 8:61–70, 1990.
38. K.-K. Lau, T. Clement (eds.). *Logic Program Synthesis and Transformation. (Proceedings of LOPSTR'92)*. Workshops in Computing, Springer-Verlag, 1993.
39. M. Leuschel. Partial evaluation of the "real thing". In L. Fribourg, F. Turini (eds.), Logic Program Synthesis and Transformation — Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, LNCS 883, 122–137, Springer-Verlag, 1994.
40. M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti (ed.), Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, 1–16, Springer-Verlag, 1995.

41. M. Leuschel. Advanced Techniques for Logic Program Specialisation. PhD thesis, K.U. Leuven, May 1997. Available via `http://www.cs.kuleuven.ac.be/~lpai`.
42. M. Leuschel. Program Specialisation and Abstract Interpretation Reconciled. In J. Jaffar (ed.), *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'98*, 220–234, MIT Press, 1998.
43. M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In G. Levi (ed.), *Static Analysis. Proceedings of SAS'98*, LNCS 1503, 230–245, 1998.
44. M. Leuschel. Improving Homeomorphic Embedding for Online Termination. In P. Flener (ed.), Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'98*, LNCS 1559. To appear.
45. M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing*, 16(3):283–342, 1998.
46. M. Leuschel, D. De Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *The Journal of Logic Programming*, 36:149–193, 1998. Earlier version in *Proceedings of PEPM'95*, 253–263, ACM Press, 1995.
47. M. Leuschel, D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen, S.D. Swierstra (eds.), *Programming Languages: Implementations, Logics and Programs*, LNCS 1140, 137–151, 1996.
48. M. Leuschel, D. De Schreye, A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher (ed.), *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, 319–332, MIT Press, 1996.
49. M. Leuschel, B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. Lloyd (ed.), *Proceedings of ILPS'95, the International Logic Programming Symposium*, 495–509, MIT Press, 1995.
50. M. Leuschel, B. Martens. Global control for partial deduction through characteristic atoms and global trees. In [14], 263–283, 1996.
51. M. Leuschel, B. Martens, D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.
52. M. Leuschel, M.H. Sørensen. Redundant argument filtering of logic programs. In John Gallager (ed.), *Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96*, LNCS 1207, 83–103, Stockholm, Sweden, August 1996.
53. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Available via `http://www.cs.kuleuven.ac.be/~lpai`.
54. J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1987.
55. J.W. Lloyd, J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
56. K. Marriott, L. Naish, J.-L. Lassez. Most specific logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, IEEE, MIT Press, 1988.
57. K. Marriott, L. Naish, J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.
58. B. Martens, D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 28:89–146, 1996.
59. B. Martens, J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling (ed.), *Proceedings ICLP'95*, 597–613, MIT Press, 1995. Extended version as Technical Report CSTR-94-16, University of Bristol.
60. T. Mogensen, A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In [38], 214–227, 1993.

61. S. Owen. Issues in the partial evaluation of meta-interpreters. In H. Abramson, M. Rogers (eds.), *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, 319–339, MIT Press, 1989.
62. A. Pettorossi, M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19&20:261–320, 1994.
63. S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.
64. M. Proietti, A. Pettorossi. Unfolding – definition – folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.
65. M. Proietti, A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *The Journal of Logic Programming*, 16(1&2):123–162, May 1993.
66. M. Proietti, A. Pettorossi. Completeness of some transformation strategies for avoiding unnecessary logical variables. In P. Van Hentenryck (ed.), *Proceedings ICLP'94*, 714–729, MIT Press, 1994.
67. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
68. T. Sato. Equivalence-preserving first-order unfold/fold transformation systems. *Theoretical Computer Science*, 105:57–84, 1992.
69. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
70. H. Seki. Unfold/fold transformation of general programs for the well-founded semantics. *Journal of Logic Programming*, 16:5–23, 1993.
71. Z. Somogyi, F. Henderson, T. Conway. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29:17-64, 1996.
72. M.H. Sørensen, R. Glück. An algorithm of generalization in positive supercompilation. In J. Lloyd (ed.), *Proceedings ILPS'95*, 465–479, MIT Press, 1995.
73. M.H. Sørensen, R. Glück, N.D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In D. Sannella (ed.), *Programming Languages and Systems*, LNCS 788, 485–500, Springer-Verlag, 1994.
74. H. Tamaki, T. Sato. Unfold/fold transformations of logic programs. In S.-Å. Tärnlund (ed.), *Proceedings of the Second International Conference on Logic Programming*, 127–138, Uppsala, Sweden, 1984.
75. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
76. F. van Harmelen. The limitations of partial evaluation. In P. Jackson, H. Reichgelt, F. van Harmelen (eds.), *Logic-Based Knowledge Representation*, 87–111, MIT Press, 1989.
77. P.L. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990.