A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration

Michael Leuschel, Danny De Schreye and André de Waal
Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{michael,dannyd,andred}@cs.kuleuven.ac.be

Abstract

The relation between partial deduction and the unfold/fold approach has been a matter of intense discussion. In this paper we consolidate the advantages of the two approaches and provide an extended partial deduction framework in which most of the tupling and deforestation transformations of the fold/unfold approach, as well the current partial deduction transformations, can be achieved. Moreover, most of the advantages of partial deduction, e.g. lower complexity and a more detailed understanding of control issues, are preserved. We build on well-defined concepts in partial deduction and present a conceptual embedding of folding into partial deduction, called conjunctive partial deduction. Two minimal extensions to partial deduction are proposed: using conjunctions of atoms instead of atoms as the principle specialisation entity and also renaming conjunctions of atoms instead of individual atoms. Correctness results for the extended framework (with respect to computed answer semantics and finite failure semantics) are given. Experiments with a prototype implementation are presented, showing that, somewhat to our surprise, conjunctive partial deduction not only handles the removal of unnecessary variables, but also leads to substantial improvements in specialisation for standard partial deduction examples.

1 Introduction

Two approaches to program transformation have received considerable attention over the last few decades: the unfold/fold approach (see e.g. [27], [25, 26], [21, 24, 23] and partial deduction, also referred to — in slightly different contexts — as partial evaluation or program specialisation (see e.g. [8], [5, 4], [18]). The relation between these two streams of work has been a matter of research, discussion and controversy over the years. Some illuminating discussions, in the context of logic programming, can be found in [20], [24, 21], [26] and [1].

From a technical perspective and in the context of definite logic programs, their relation is clear: 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 transformation rule. Other basic unfold/fold rules, such as "folding", "definition", "lemma application" or "goal replacement", or — in more general unfold/fold contexts — "clause replacement" and others, are not supported.

This is only a rough representation of the relationship between the two approaches. To refine it, first, it should be noted that any partial deduction algorithm imposing the Lloyd-Shepherdson closedness condition ([18]) involves a (weak) implicit folding step. Intuitively, this closedness condition requires that partial deduction should not be performed on the basis of a single (atomic) goal, but on a set of atoms, say S, and that each atom which occurs in a body of a clause in the transformed program, and which has a predicate occurring in S, should be an instance of one of the atoms in S. If the condition holds, each such atom in a body of a transformed clause refers back to one of the heads of the transformed clauses, and (a limited form of) implicit folding is obtained.

Moreover, most partial deduction methods make use of renaming transformations. Again this relates to the Lloyd-Shepherdson framework: in this case to the independence condition. Atoms in the set S, together with their corresponding occurrences in the transformed clauses, are renamed apart to avoid duplication of code for those pairs of atoms which share instances. This avoids the generation of redundant solutions — and, in the presence of negation, of incorrect solutions — while preserving polyvariance. Again, renaming is closely related to unfold/fold. Roughly stated, it can be formalised as a two-step basic 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 additional 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 [21, 23]). Two types of redundant variables are often distinguished in the literature. The first refer to those cases in which the same input datastructure is consumed twice. As an example, consider the predicate

$$max\_length(x, m, l) \leftarrow max(x, m), length(x, l)$$

which is true if m is the maximum of the list of numbers x, and if l is the length of the list. By unfold/fold, the definitions for $max/2$ and $length/2$ can be merged, producing a definition for $max\_length/3$, which only traverses x once. In the functional community, such a transformation is referred to as *tupling*.

A second type of redundant variables turns up in cases where a datastructure is first constructed by some procedure, and, in a next part of the computation, decomposed again. As an example, consider the predicate

$$double\_app(x,y,z,r) \leftarrow app(x,y,i), app(i,z,r)$$

which holds if the list r can be obtained by concatenating the lists x, y and z. Again, unfold/fold allows to merge the two calls to $app/3$ and to eliminate the construction of the intermediate datastructure i. In this case, in the functional community, the transformation is referred to as *deforestation*. Neither of these transformations are achievable through partial deduction alone.

On the other hand, partial deduction has some 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. These 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. Formal frameworks have been developed, analysing issues of termination, code- and search-explosion and obtained efficiency gains ([2, 19], [5, 4], [11, 14]). Several fully automated systems have been developed (SP, SAGE, PADDY, MIXTUS, CHTREE/ECCE) and have been successfully applied to at least medium-size applications ([12], [15], [3], [9]). As a result, partial deduction has reached a degree of maturity that brings it to the edge of wide-scale industrial applicability, which is beyond what any other transformation technology for logic programs has achieved today.

The aim of this paper is to provide a basic starting point to bring the advantages of these two transformation methods together. In order to do so, we only rely and build on concepts which are already well understood in partial deduction: the Lloyd-Shepherdson framework and renaming transformations. No explicit new basic transformation rules, such as folding or definition, are introduced. Nevertheless, we provide a framework in which most tupling and deforestation transformations, in addition to the current partial deduction transformations, can be achieved.

More precisely, we propose two minimal extensions to partial deduction methods, prove their correctness and illustrate how they achieve removal of unnecessary variables within a framework of *conjunctive partial deduction*.

One of these minimal extensions is on the level of the Lloyd-Shepherdson framework itself: we will consider sets S of conjunctions of atoms instead of the usual sets of atoms. A second extension is on the level of the renaming transformations, where we will use renamings of conjunctions of atoms, instead of renamings of single atoms. Together, they provide for a setting of conjunctive partial deduction 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 already pointed out that, at least on the technical level, most of these ideas have already been raised in the context of unfold/fold (e.g. [24, 21], [26]). In these papers too, the step from partial deduction to (certain strategies in) unfold/fold has been characterised as essentially moving from sets of atoms to sets of conjunctions. Therefore, technically, the current paper is strongly related to these works, and for a number of our proofs we actually apply the results of [21]. The objectives, however, differ. Where [24, 21] aim to clarify the relation between the two approaches, by characterising partial deduction within the unfold/fold framework, we will provide minimal extensions to partial deduction with the aim of including a large part of the unfold/fold power. We show how correctness results for partial deduction can be reformulated in the extended context. The ultimate goal being to provide a generalised framework that allows to easily extend current results on control of partial deduction and current partial deduction systems. Particular instances of this framework, along with control issues, are dealt with in [6].

2 Conjunctive partial deduction

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. Throughout the paper, we restrict the attention to definite programs and goals. In section 4 we discuss a further extension to the normal case.

We suppose familiarity with basic notions in logic programming ([17]). As usual in partial deduction, we assume that the standard notion of an SLD-tree is generalised ([18]) to allow it to be incomplete: at any point we may decide not to select any atom and terminate a derivation. Leaves of this kind will be called *dangling* ([19]). Also, we will call an SLD-tree *trivial* iff its root is a dangling leaf. The following basic notion is adapted from [18].

Definition 2.1 (resultant) *Let P be a program, $G = \leftarrow Q$ a goal, where Q is a conjunction of atoms, $G_0 = G, G_1, ..., G_n$ a finite derivation for $P \cup \{G\}$, with substitutions $\theta_1, ..., \theta_n$, and let G_n have the form $\leftarrow Q_n$. We say that $Q\theta_1...\theta_n \leftarrow Q_n$ is* the resultant *of the derivation $G_0, G_1, ..., G_n$.*

Note that, in general, resultants are not clauses: their left-hand sides may contain a conjunction of atoms. The notion can be generalised to SLD-trees. Given a finite SLD-tree τ for $P \cup \{G\}$, there is a corresponding set of resultants R_τ, including one resultant for each non-failed derivation of τ.

In the partial deduction notion introduced in [18] (there referred to as *partial evaluation*), the SLD-trees and resultants are restricted to *atomic* top-level goals. We omit this restriction here. In order to formulate this generalisation, we need the following concept.

Definition 2.2 (generalised program) *A* generalised program *is any set of Horn clauses and resultants.*

Definition 2.3 (pre-conjunctive partial deduction)
Let $S = \{Q_1, ..., Q_s\}$ be a set of conjunctions of atoms, and $T = \{\tau_1, ..., \tau_s\}$

a set of finite, non-trivial SLD-trees for $P \cup \{\leftarrow Q_1\}, ..., P \cup \{\leftarrow Q_s\}$, with associated sets of resultants $R_1, ..., R_s$, respectively. Let P_S be the generalised program obtained by removing all clauses from P which define predicates occurring in atomic elements of S, and by adding the resultants in $R_1, ..., R_s$. P_S is called a pre-conjunctive partial deduction *of P wrt S (and T).*[1]

Let us immediately illustrate this notion with a very simple example we already referred to in the introduction. The example, as well as the *double_app* Example 2.9, is fairly trivial. This does not relate to any limitations of the proposed framework, but to a deliberate choice of selecting minimally complex examples for illustrating the proposed concepts and method. Also, in the following, we introduce the connective $\wedge$ to avoid confusion between conjunction and the set punctuation symbol ",".

Example 2.4 (max_length) *Let P be the following program.*

(C_1) $max\_length(x,m,l) \leftarrow max(x,m) \wedge length(x,l)$
(C_2) $max(x,m) \leftarrow max\_1(x,0,m)$
(C_3) $max\_1(nil,m,m) \leftarrow$
(C_4) $max\_1(h.t,n,m) \leftarrow h \leq n \wedge max\_1(t,n,m)$
(C_5) $max\_1(h.t,n,m) \leftarrow h > n \wedge max\_1(t,h,m)$
(C_6) $length(nil,0) \leftarrow$
(C_7) $length(h.t,l) \leftarrow length(t,k) \wedge l \ is \ k+1$

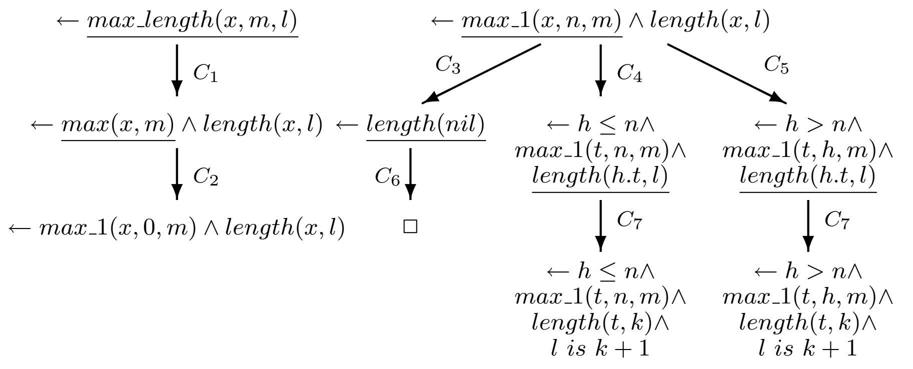
Let $S = \{max\_length(x,m,l), max\_1(x,n,m) \wedge length(x,l)\}$. Assume that we construct the finite SLD-trees T_1, T_2 depicted in Figure 1. The associated resultant sets $R_1 = \{R_{1,1}\}$ and $R_2 = \{R_{2,1}, R_{2,2}, R_{2,3}\}$ are:

$(R_{1,1})$ $max\_length(x,m,l) \leftarrow max\_1(x,0,m) \wedge length(x,l)$
$(R_{2,1})$ $max\_1(nil,n,n) \wedge length(nil,0) \leftarrow$
$(R_{2,2})$ $max\_1(h.t,n,m) \wedge length(h.t,l) \leftarrow$
$h \leq n \wedge max\_1(t,n,m) \wedge length(t,k) \wedge l \ is \ k+1$
$(R_{2,3})$ $max\_1(h.t,n,m) \wedge length(h.t,l) \leftarrow$
$h > n \wedge max\_1(t,h,m) \wedge length(t,k) \wedge l \ is \ k+1$

P_S then consists of the resultants from R_1 and R_2, in addition to the clauses for *max*, *max_1* and *length* from P. Clearly P_S is a generalised (non-standard) program. Apart from that, with the exception that the redundant variable still has multiple occurrences, P_S has the desired tupling structure. The two functionalities (*max*/3 and *length*/2) in the original program have been merged into single traversals.

In order to convert the generalised program into a standard one, we will rename conjunctions of atoms by new atoms, using fresh predicate symbols. Such renamings require some care. For one thing, given a generalised program P_S, obtained as a pre-conjunctive partial deduction of P wrt a set

[1]In the remainder of this paper we will often talk about a pre-conjunctive partial deduction wrt S without explicitly mentioning T.



$\leftarrow \underline{max\_length(x,m,l)}$
C_1
$\leftarrow \underline{max(x,m)} \wedge length(x,l)$
C_2
$\leftarrow max\_1(x,0,m) \wedge length(x,l)$
$\leftarrow \underline{max\_1(x,n,m)} \wedge length(x,l)$
C_3
C_4
C_5
$\leftarrow \underline{length(nil)}$
C_6
$\Box$
$\leftarrow h \leq n \wedge$ $max\_1(t,n,m) \wedge$ $\underline{length(h.t,l)}$
$\leftarrow h > n \wedge$ $max\_1(t,h,m) \wedge$ $\underline{length(h.t,l)}$
C_7
C_7
$\leftarrow h \leq n \wedge$ $max\_1(t,n,m) \wedge$ $length(t,k) \wedge$ $l\ is\ k+1$
$\leftarrow h > n \wedge$ $max\_1(t,h,m) \wedge$ $length(t,k) \wedge$ $l\ is\ k+1$

Figure 1: SLD-trees for Example 2.4

S, there may be ambiguity concerning which conjunctions in the bodies to rename. For instance, if P_S contains the clause $p(x,y) \leftarrow r(x) \wedge q(y) \wedge r(z)$ and S 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.*

Below, we use 2^A as a notation for the powerset of a set A. $=_r$ denotes syntactic identity, up to reordering.

Definition 2.5 (partitioning function) *Let C denote the set of all conjunctions of atoms over the given alphabet. A* partitioning function *is a mapping $p : C \rightarrow 2^C$, such that for any $Q \in C$: $Q =_r \wedge_{Q_i \in p(Q)} Q_i$.*

For the *max_length* example, let p be the partitioning function which, maps any conjunction $Q =_r max\_1(x,n,m) \wedge length(x,l) \wedge B_1 \wedge ... \wedge B_n$ to $\{max\_1(x,n,m) \wedge length(x,l), B_1, ..., B_n\}$, where $B_1, ..., B_n$, $n \geq 0$, are atoms with predicates different from max_1 and *length*. We leave p undefined on other conjunctions.

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, generalised programs into standard ones). 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 variables.

Below we introduce a class of generalised renaming functions, making abstraction of the actual functionalities supported by its members. This will allow us to state our results in full generality, not restricted to any particular selected renaming scheme.

Definition 2.6 (atomic renaming) *An* atomic renaming α *for a given pre-conjunctive partial deduction P_S (of P wrt S) is a mapping from S to atoms such that for each $s \in S$:*

- *$vars(\alpha(s)) \subseteq vars(s)$,*
- *$\alpha(s)$ has a predicate symbol which does not occur in P, and is distinct from the predicate symbols of any $\alpha(s')$ with $s' \in S \wedge s' \neq s$.*

We stress that this notion is to some extent overly general. It allows to remove any subset of the variables from the given conjunction. This may lead to incorrect transformations. A simple safe subclass of atomic renamings is obtained by imposing that $vars(\alpha(s)) = vars(s)$. More concretely, we could map any conjunction $s \in S$ to an atom $p_s(x_1, ..., x_n)$, where $x_1, ..., x_n$ is the list of all distinct variables occuring in s, and p_s the fresh predicate symbol. Such instances of the definition would support the first two functionalities stated above. More refined correct instances of the definition, supporting also the third functionality, will be presented later on. Also, note that with this definition, we are actually also renaming the atomic elements of S. This is not really essential for converting generalised programs into standard ones, but it will prove useful for various other aspects (e.g. dealing with "independence" and facilitating correctness proofs).

Definition 2.7 (renaming function) *Let α be an atomic renaming for a pre-conjunctive partial deduction P_S and p a partitioning function. A* renaming function $\rho_{\alpha,p}$ *(based on α and p)*[2] *for P_S is a mapping from conjunctions to atoms such that: $\rho_{\alpha,p}(B) = \bigwedge_{Q \in p(B)} Q'$ where*

- *if Q is an instance of an element in S, then $Q' = \alpha(s)\theta$, for some $s \in S$ with $Q = s\theta$,*
- *if Q is not an instance of an element in S, then $Q' = Q$.*

Observe that there is a degree of non-determinism here. If S contains elements s and s' which share common instances, then there are several renaming functions $\rho_{\alpha,p}$ associated with the same atomic renaming α and partitioning p. Also, in that case, we do not necessarily have $\alpha(s) = \rho_{\alpha,p}(s)$.

Definition 2.8 (conjunctive partial deduction) *Let $\rho_{\alpha,p}$ be a renaming function for a pre-conjunctive partial deduction P_S, with associated set S and resultant sets $R_s, s \in S$. The* conjunctive partial deduction *of P_S (under $\rho_{\alpha,p}$) is the program $P_{\rho_{\alpha,p}}$ containing:*

- *for each resultant $s\theta \leftarrow B$ in any set $R_s, s \in S$, the clause:*
  $\alpha(s)\theta \leftarrow \rho_{\alpha,p}(B)$
- *for each other clause $H \leftarrow B$ of P_S, the clause:*
  $H \leftarrow \rho_{\alpha,p}(B)$

[2]When writing $\rho_{\alpha,p}$ we will implicitly assume that the renaming $\rho_{\alpha,p}$ is based on an atomic renaming α and a partitioning function p.

Returning to our example, we introduce a fresh predicate for each of the two elements in S via the atomic renaming α: $\alpha(max\_length(x, m, l)) = p_1(x, m, l)$ and $\alpha(max\_1(x, n, m) \wedge length(x, l)) = p_2(x, n, m, l)$. As S does not contain elements with common instances, there exists only one renaming function $\rho_{\alpha,p}$ based on α and p. The conjunctive partial deduction is now obtained as follows. The head $max\_length(x, m, l)$ in the single clause of R_1 is replaced by $p_1(x, m, l)$. The head-occurrences, $max\_1(nil, n, n) \wedge length(nil, 0)$ and $max\_1(h.t, n, m) \wedge length(h.t, l)$ are replaced by $p_2(nil, n, n, 0)$ and $p_2(h.t, n, m, l)$. The body occurrences $max\_1(x, 0, m) \wedge length(x, l)$, $max\_1(t, n, m) \wedge length(t, k)$ and $max\_1(t, h, m) \wedge length(t, k)$ are replaced by $p_2(x, 0, m, l)$, $p_2(t, n, m, k)$ and $p_2(t, h, m, k)$ respectively. The resulting program is (in addition to the clauses for $max/2$, $max\_1/3$ and $length/2$):

$p_1(x, m, l) \leftarrow p_2(x, 0, m, l).$
$p_2(nil, n, n, 0) \leftarrow$
$p_2(h.t, n, m, l) \leftarrow h \leq n \wedge p_2(t, n, m, k) \wedge l\ is\ k + 1.$
$p_2(h.t, n, m, l) \leftarrow h > n \wedge p_2(t, h, m, k) \wedge l\ is\ k + 1.$

$$\begin{tikzcd}
 & \leftarrow \underline{app(x,y,i)} \wedge app(i,z,r) \arrow[dl, "C_1"'] \arrow[dr, "C_2"] & \\
\leftarrow app(y,z,r) & & \leftarrow app(x',y,i') \wedge \underline{app(h.i',z,r)} \arrow[d, "C_2"] \\
 & & \leftarrow app(x',y,i') \wedge app(i',z,r')
\end{tikzcd}$$

Figure 2: SLD-tree for Example 2.9

Example 2.9 (double append)

$(C_1) \quad app(nil, l, l) \leftarrow$
$(C_2) \quad app(h.x, y, h.z) \leftarrow app(x, y, z)$

Let $P = \{C_1, C_2\}$ be the (all too) well known *append* program. Let $S = \{app(x, y, i) \wedge app(i, z, r)\}$ and assume that we construct the finite SLD-tree T_1 depicted in Figure 2. The pre-conjunctive partial deduction P_S of P wrt S then contains the clauses of P as well as the following resultants:

$(R_1) \quad app(nil, y, y) \wedge app(y, z, r) \leftarrow app(y, z, r)$
$(R_2) \quad app(h.x', y, h.i') \wedge app(h.i', z, h.r') \leftarrow app(x', y, i') \wedge app(i', z, r')$

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 P_S such that $\alpha'(app(x, y, i) \wedge app(i, z, r)) = da(x, y, i, z, r)$ (i.e. the distinct variables have been collected and have been ordered according to their first appearance), the conjunctive partial deduction of P_S under $\rho_{\alpha',p}$ will contain the clauses C_1, C_2 of P as well as:

(C_3') $\quad da(nil, y, y, z, r) \leftarrow app(y, z, r)$
(C_4') $\quad da(h.x', y, h.i', z, h.r') \leftarrow da(x', y, i', z, r')$

Executing $Q = app(x, y, i) \wedge app(i, z, r)$ in the original program leads to the construction of an intermediate list i by $app(x, y, i)$, which is then traversed again (consumed) by $app(i, z, r)$. In the conjunctive partial deduction the inefficiency caused by this unnecessary traversal of i has been completely removed. However the intermediate list i is still constructed, and if we are not interested in its value, then this is an unnecessary overhead. We can remedy this by using an atomic renaming α such that $\alpha(app(x, y, i) \wedge app(i, z, r)) = da(x, y, z, r)$. We refer the reader to the next sections on why α is correct. For this more sophisticated renaming, the conjunctive partial deduction of P_S under $\rho_{\alpha,p}$ contains the clauses C_1, C_2 as well as:

(C_3) $\quad da(nil, y, z, r) \leftarrow app(y, z, r)$
(C_4) $\quad da(h.x', y, z, h.r') \leftarrow da(x', y, z, r')$

The unnecessary variable i, as well as the inefficiencies caused by it, have now been removed.

3 Correctness

In this section we will state correctness results of conjunctive partial deduction. All proofs and lemmas have been omitted and can be found in the extended version of this paper [13].

As already mentioned in the introduction, partial deduction is a strict subset of the (full) unfold/fold transformation technique as defined for instance in [21]. 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 [21].

Basically a conjunctive partial deduction $P_{\rho_{\alpha,p}}$ of P_S under $\rho_{\alpha,p}$ can be obtained from P using 4 transformation phases. In the first phase one introduces definitions for every conjunction in S, 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 S: exactly one unfolding for each corresponding unfolding in the trees. In the third phase conjunctions in the body of clauses are folded using the definitions introduced in phase 1. Finally, in the fourth phase, the definitions which define predicates occuring in atomic elements of S are removed. The first three phases can be mapped to the unfold/fold transformation framework of [21] 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 [21]).

In [13] it is shown that the definition steps are *T&S-Definition* steps ([27, 21]). Furthermore, given the following correctness criterion for renaming functions, the folding steps are *T&S-Folding* steps [27, 21]. Basically the criterion states that variables removed by the renaming should be existential variables.

Definition 3.1 (correct renaming) *Let $\rho_{\alpha,p}$ be a renaming function for P_S. We say that $\rho_{\alpha,p}$ is* correct *(for P_S) iff for every clause $H' \leftarrow \bigwedge_{Q \in p(B)} Q'$ in $P_{\rho_{\alpha,p}}$ derived from $H \leftarrow B \in P_S$, and every $Q \in p(B)$ with $Q' = \alpha(s)\theta$, $s \in S$, $Q = s\theta$, $vars(s) \setminus vars(\alpha(s)) = \{x_1, \ldots, x_n\}$, $V = \langle x_1, \ldots, x_n \rangle$ we have that $V\theta$ is a (possibly empty) sequence of distinct variables which occur neither in H', Q', nor in $p(B) \setminus \{Q\}$.*

Example 3.2 Take $P_S = \{C_1, C_2, C_3, C_4\}$, $\rho_{\alpha,p}$ and $\rho_{\alpha',p}$ of Example 2.9. $\rho_{\alpha',p}$ does not remove any variables and the above requirements are thus trivially met ($V\theta = \langle\rangle$). $\rho_{\alpha,p}$ however removes variables and we have to examine each clause of P_S. For C_1, C_2, C_3 no conjunction in a body gets renamed and $V\theta = \langle\rangle$. For C_4 we have that $Q = app(x', y, i') \wedge app(i', z, r')$ gets renamed into $Q' = da(x', y, z, r')$ and we have $V = \langle i \rangle$, $V\theta = \langle i' \rangle$. $V\theta$ is a sequence of distinct variables which occur neither in $H' = da(h.x', y, z, h.r')$, Q' nor in the (empty) remainder $p(B) \setminus \{Q\}$. Hence $\rho_{\alpha,p}$ is correct.

In Definition 2.3 (as well as in the standard definition of partial deduction in [18]) we imposed that the SLD-trees are non-trivial. In the context of standard partial deduction of atoms, this condition avoids problematic resultants of the form $A \leftarrow A$ and is fully sufficient for total correctness (given independence and closedness). In the context of conjunctive partial deductions of conjunctions we need (for some results) an extension of this condition:

Definition 3.3 (non-trivial SLD-tree wrt p) *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 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 $A_1\theta, \ldots, A_{i-1}\theta, A_{i+1}\theta, \ldots, A_n\theta$ are* inherited from G in G'. *We extend this notion to derivations by taking the transitive and reflexive closure.*

Let τ be an SLD-tree for $P \cup \{G\}$ and let the goals in the dangling leaves of τ be $\{\leftarrow L_1, \ldots, \leftarrow L_n\}$. Also, let p be a partitioning function and S a set of conjunctions. τ is said to be non-trivial wrt p and S *iff for every conjunction Q' in $p(L_i)$, $1 \leq i \leq n$, which is an instance of a conjunction in S, none of its atoms are inherited from G in $\leftarrow L_i$. A pre-conjunctive partial deduction P_S is* non-trivial wrt p *iff the SLD-trees for the elements $s \in S$ are non-trivial wrt p and S.*

Example 3.4 Let τ 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 are $\{\leftarrow L_1, \leftarrow L_2\}$, with $L_1 = app(y, z, r)$ and $L_2 = app(x', y, i') \wedge app(i', z, r')$. For $S = \{\ app(x, y, i) \wedge app(i, z, r)\ \}$ and p, such that $p(B) = \{B\}$, we have that τ is non-trivial wrt p and S:

- $app(x, y, i)$ and $app(x, y, i)$ are not inherited from G in $\leftarrow L_2$.
- $app(y, z, r)$ is inherited from G, but L_1 is not an instance of a conjunction in S.

Note however that, for $S' = S \cup \{app(x, y, z)\}$, τ is *not* non-trivial wrt p and S', because now $app(y, z, r)$ is an instance of a conjunction in S'.

The above means that an atom can only be renamed (on its own or because it is inside a conjunction which gets renamed) if it is not inherited from the top-level conjunction. This is a stronger but simpler condition than the one of *fold-allowing* in [21] or *inherited* in [25].[3] 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 discussions see e.g. [25]).

We are now in position to state a correctness result similar to the one in [18]. In contrast to [18], we do not need an independence condition (because of the renaming), but we still need an adapted closedness condition:

Definition 3.5 (S-closed wrt p) *Let p be a partitioning function and S a set of conjunctions. We say that a conjunction Q is S-*closed wrt *p iff every conjunction $Q' \in p(Q)$ is either an instance of an element in S or it is an atomic conjunction whose predicate symbol is different from those of all atomic conjunctions in S. Furthermore a generalised program P is S-*closed wrt *p iff every body of every clause or resultant in P is S-closed wrt p.*

Example 3.6 Let $S = \{q(x) \wedge r, q(a)\}$, $Q = q(a) \wedge q(b) \wedge r$. Then, for a partitioning function p such that $p(Q) = \{q(b) \wedge r, q(a)\}$, Q is S-closed wrt p. However, for p' with $p'(Q) = \{q(a) \wedge r, q(b)\}$, Q is not S-closed wrt p'.

Theorem 3.7 *Let $P_{\rho_{\alpha,p}}$ be a conjunctive partial deduction of P_S under $\rho_{\alpha,p}$ and let $\rho_{\alpha,p}$ be correct for $P_S \cup \{G\}$. If $P_S \cup \{G\}$ is S-closed wrt p then*

- *$P \cup \{Q\}$ has an SLD-refutation with c.a.s. θ, with $\theta' = \theta \mid_{vars(\rho_{\alpha,p}(Q))}$, iff $P_{\rho_{\alpha,p}} \cup \{\rho_{\alpha,p}(Q)\}$ has an SLD refutation with c.a.s. θ'.*

If in addition P_S is non-trivial wrt p then

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

Example 3.8 (double-append) Let P and $P_{\rho_{\alpha,p}}$ be the programs from Example 2.9. Let $G = \leftarrow app(1.2.nil, 3.nil, i) \wedge app(i, 4.nil, r)$. We have that $\rho_{\alpha,p}(G) = da(1.2.nil, 3.nil, 4.nil, r)$. It can be seen that $\rho_{\alpha,p}$ is correct for $P_S \cup \{G\}$[4] and indeed $P \cup \{G\}$ and $P_{\rho_{\alpha,p}} \cup \{\rho_{\alpha,p}(G)\}$ have the same set of computed answers (restricted to r): $\{\{r/1.2.3.4.nil\}\}$. Note that the SLD-trees of Example 2.9 are non-trivial wrt p and therefore finite failure is also preserved. However, for $G' = \leftarrow app(1.2.nil, 3.nil, i) \wedge app(i, 4.nil, i)$, $\rho_{\alpha,p}$ is *not* correct for $P_S \cup \{G'\}$ and indeed $P \cup \{G'\}$ fails infinitely, while

[3]Theorem 3.7 also holds for these weaker conditions, which might be useful when using e.g. determinate unfolding rules, which do not always produce non-trivial trees.

[4]This also ensures that the existential (according to $\rho_{\alpha,p}$) term i in G is a free variable.

$P_{\rho_{\alpha,p}} \cup \{\leftarrow da(1.2.nil, 3.nil, 4.nil, i)\}$ succeeds with the computed answer $\{i/1.2.3.4.nil\}$.

4 Discussion, Experiments and Conclusion

Negation and Normal Programs

When extending conjunctive partial deduction for negation two issues come up: one is correctly handling normal logic programs instead of definite programs and the second one is to build SLDNF-trees instead of SLD-trees (i.e. extending unfolding to allow the selection of ground negative literals).

The former is not so difficult, as a lot of results from the literature can be reused. For instance we get preservation of the perfect model semantics for stratified programs [25] and preservation of the well-founded semantics for general programs [26]. If in addition we have non-trivial SLD-trees wrt p the conditions of modified (T&S) folding of [25] hold and we can reuse its results for the SLDNF success and finite failure set for stratified programs.

Allowing SLDNF-trees instead of SLD-trees is more difficult. Note that [25, 26, 21] do not allow the unfolding of negative literals. For further details we have to refer to [13]. For the moment there seems to be no equivalent to the correctness theorem of [18] for normal logic programs and further work will be needed to extend the correctness results of the previous section.

Prototype Implementation and Experiments

The system based on characteristic trees and atoms described in [11, 14] has been augmented to perform conjunctive partial deduction. The adaptation was pretty straightforward, further underlining our earlier claim that conjunctive partial deduction is just a simple, but powerful, extension of partial deduction.

Although in this paper, we have not addressed the control issues involved in conjunctive partial deduction, for our experimentation certain choices needed to be fixed. For a thorough discussion of the control (and termination) issues we refer to [6]. We briefly indicate some of the choices we made. The generalisation of conjunctions was implemented in a straightforward, but naive way, meaning that the prototype cannot handle some of the harder unfold/fold examples. We used a determinate unfolding rule with a flexible lookahead and for global control we used the techniques of [11, 14].

Table 1 contains absolute timings and speedups for some benchmarks (which can be found in [13]; more recent and extensive experiments can also be found in [7]). The timings were obtained under Sicstus Prolog 3 on a Sparc Classic. The *doubleapp* benchmark is the Example 2.9 of this paper. The *maxlength* is a slightly adapted version of Example 2.4 (not using built-ins). Both of these examples are standard "unfold/fold" examples illustrating the benefits of removing unnecessary variables — the speedups are satisfactory, given the fact that no partial input was provided.

The next three benchmarks are "standard" partial deduction benchmarks. The *depth* benchmark is the meta-interpreter also found in [10], but using a more sophisticated object program. The good results seem to indicate that folding combined with a rather simple unfolding rule can solve some of the traditional local control problems for meta-interpreters. Folding also solves a problem already raised in [20]. Take for example a meta-interpreter containing the clause $solve(X) \leftarrow exp(X) \wedge clause(X, B) \wedge solve(B)$, where $exp(X)$ is an expensive test which for some reason cannot be (fully) unfolded. Here standard partial deduction faces a dilemma when specialising $solve(\bar{d})$. Either it unfolds $clause(\bar{d}, B)$, thereby propagating partial input $\bar{d}$ over to $solve(B)$, but at the cost of duplicating $exp(\bar{d})$ and most probably leading to inefficient programs. Or standard partial deduction can stop the unfolding, but then the partial input $\bar{d}$ can no longer be exploited inside $solve(B)$. Using conjunctive partial deduction however, we can be efficient *and* propagate information at the same time, simply by stopping unfolding and specialising the conjunction $clause(\bar{d}, B) \wedge solve(B)$.[5] All this leads us to believe that conjunctive partial deduction can be highly beneficial for real-life applications, given that a satisfactory treatment of the control issues (based on [6]) can be achieved.

The *relative* and *contains* benchmarks are the same as in [10]. For the *contains* example, conjunctive partial deduction gives a considerable improvement. *contains* has long been a difficult benchmark, especially for determinate unfolding rules. This might indicate that conjunctive partial deduction diminishes the need for more sophisticated unfolding rules.

So, to our surprise, conjunctive partial deduction not only handles the removal of unnecessary variables, but leads to substantial improvements in specialisation for standard partial deduction examples as well.

Benchmark	Original		Standard PD		Conjunctive PD	
doubleapp	0.320 s	**1**	0.325 s	**0.98**	0.230 s	<u>**1.39**</u>
maxlength	1.985 s	**1**	1.170 s	**1.70**	1.020 s	<u>**1.95**</u>
depth	0.980 s	**1**	0.875 s	**1.12**	0.340 s	<u>**2.88**</u>
relative	1.042 s	**1**	1.039 s	**1.00**	1.005 s	<u>**1.04**</u>
contains	0.740 s	**1**	0.700 s	**1.06**	0.160 s	<u>**4.63**</u>

Table 1: Some Experiments — Absolute Timings and Speedups

[5]The Paddy system [22] uses non-atomic, non-recursive folding to avoid backpropagation of bindings (to preserve Prolog semantics). In general this is sufficient to solve the above problem. Apart from that, Paddy only performs folding of atoms, i.e. the same implicit folding used in *standard* partial deduction.

General Remarks and Conclusion

In the above implementation we still use a simple, but safe renaming (with $vars(\alpha(s)) = vars(s)$). Details about how to obtain an optimal, correct renaming can be found in [16]. Also note that the framework of conjunctive partial deduction, is mono-variant wrt existential variables: i.e. if $p(x, i)$ is used once, with i existential, and somewhere else with i not being existential, then we cannot remove the existential variable for the first call. This is also dealt with in [16].

Remark that in Definition 2.5 a partitioning function p is a mapping from conjunctions to sets of conjunctions. To be fully general we would have to extend this so that p maps to multisets of conjunctions (making notations more burdensome, but the same results apply).

In conclusion, we have presented a simple but powerful extension of partial deduction, showed that it can perform tupling and deforestation and proved a correctness result similar to the one of standard partial deduction. Experiments conducted with a prototype confirm that techniques developed for standard partial deduction carry over and that the added power actually pays off in practice.

Acknowledgements

Michael Leuschel is supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation". Danny De Schreye is senior research associate of the Belgian National Fund for Scientific Research. André de Waal is supported by HCM Project Compulog Group—Cooperation Group in Computational Logic. We thank Robert Glück, Jesper Jørgensen, Bern Martens, Alberto Pettorossi, Maurizio Proietti and Morten Heine Sørensen for interesting discussions and comments. Jesper Jørgensen and Bern Martens also provided valuable feedback on several drafts of this paper.

References

[1] A. Bossi, N. Cocco, and S. Dulli. A method for specialising logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.

[2] M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.

[3] D. de Waal and J. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In A. Bundy, editor, *Automated Deduction—CADE-12*, pages 207–221. Springer-Verlag, 1994.

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

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

[6] R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. Technical Report CW 226, Departement Computerwetenschappen, K.U. Leuven, Belgium, February 1996.

[7] J. Jørgensen, M. Leuschel and B. Martens. Conjunctive Partial Deduction in Practice. Submitted.

[8] J. Komorowksi. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation.* PhD thesis, Linköping University, Sweden, 1981. Linköping Studies in Science and Technology Dissertations 69.

[9] A. Lakhotia and L. Sterling. How to control unfolding when specializing interpreters. *New Generation Computing*, 8:61–70, 1990.

[10] J. Lam and A. Kusalik. A comparative analysis of partial deductors for pure Prolog. Technical report, Department of Computational Science, University of Saskatchewan, Canada, May 1990. Revised April 1991.

[11] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, *Proceedings of LOPSTR'95*, LNCS 1048, pages 1–16, 1996. Springer-Verlag.

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

[13] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. Technical Report CW 225, Departement Computerwetenschappen, K.U. Leuven, Belgium, February 1996. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[14] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, P. Thiemann, editors, *Proceedings 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS, Springer Verlag. To Appear.

[15] M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 495–509, Portland, USA, December 1995. MIT Press.

[16] M. Leuschel and M. H. Sørensen. Redundant Argument Filtering of Logic Programs. May 1996. Submitted for Publication.

[17] J. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.

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

[19] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 1996. To Appear.

[20] S. Owen. Issues in the partial evaluation of meta-interpreters. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 319–339. MIT Press, 1989.

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

[22] S. Prestwich. The PADDY partial deduction system. Technical Report ECRC-92-6, ECRC, Munich, Germany, 1992.

[23] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. In J. Małuszyński and M. Wirsing, editors, *Proceedings of PLILP'91*, LNCS 528, Springer Verlag, pages 347–358, 1991.

[24] M. Proietti and 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.

[25] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.

[26] H. Seki. Unfold/fold transformation of general programs for the well-founded semantics. *Journal of Logic Programming*, 16:5–23, 1993.

[27] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984.