Advanced Logic Program Specialisation

Michael Leuschel

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
mal@ecs.soton.ac.uk
www: http://www.ecs.soton.ac.uk/~mal

1 Introduction

In first part of this course [28] we have laid the theoretical foundations for logic program specialisation, notably introducing the technique of *partial deduction* along with some basic techniques to automatically control it. In this part of the course we first present in Section 2 an advanced way of controlling polyvariance based upon *characteristic trees*. We then show in Section 3 how partial deduction can be extended into *conjunctive partial deduction*, incorporating much of the power of unfold/fold program transformation techniques, such as tupling and deforestation, while keeping the automatic control of partial deduction. Finally, in Section 4 we elaborate on combining *abstract interpretation* with conjunctive partial deduction, showing how together they are more powerful than either method alone.

2 Characteristic Trees for Global Control

2.1 Structure and abstraction

In the first part of the course [28] we have presented the generic partial deduction Algorithm 3.1. This algorithm is parametrised by an unfolding rule for the local control and by an abstraction operator for the control of polyvariance. The abstraction operator examines a set of atoms and then decides which of the atoms should be abstracted and which ones should be left unmodified.

An abstraction operator like the *msg* is just based on the *syntactic structure* of the atoms to be specialised. However, two atoms can be specialised in a very similar way in the context of one program P_1, and in a very dissimilar fashion in the context of another program P_2. The syntactic structure of the two atoms being unaffected by the particular context, an operator like the *msg* will perform exactly the same abstraction within P_1 and P_2, even though very different generalisations might be called for. A much better idea might therefore be to examine the SLDNF-trees generated for these atoms. These trees capture (to some depth) how the atoms behave computationally in the context of the respective programs. They also capture (part of) the specialisation that has been performed on these atoms. An abstraction operator which takes these trees into account will notice their similarity in the context of P_1 and their dissimilarity in P_2, and can therefore take appropriate actions in the form of different generalisations. The following example illustrates these points.

Example 1. Let P be the *append* program:

(1) $append([], Z, Z) \leftarrow$
(2) $append([H|X], Y, [H|Z]) \leftarrow append(X, Y, Z)$

Note that we have added clause numbers, which we will henceforth take the liberty to incorporate into illustrations of SLD-trees in order to clarify which clauses have been resolved with.
Let $\mathcal{A} = \{B, C\}$ be a set of atoms, where $B = append([a], X, Y)$ and $C = append(X, [a], Y)$. Typically a partial deducer will unfold the two atoms of $\mathcal{A}$ in the way depicted in Fig. 1, returning the finite SLD-trees τ_B and τ_C. These two trees, as well as the associated resultants, have a very different structure. The atom $append([a], X, Y)$ has been fully unfolded and we obtain as only resultant the fact:

$$append([a], X, [a|X]) \leftarrow$$

while for $append(X, [a], Y)$ we obtain the following resultants:

$$append([], [a], [a]) \leftarrow$$
$$append([H|X], [a], [H|Z]) \leftarrow append(X, [a], Z)$$

In this case, it is thus vital to keep separate specialised versions for B and C. However, it is very easy to come up with another context in which the specialisation behaviour of B and C are almost indiscernible. Take for instance the following program P^* in which $append^*$ no longer appends two lists but finds common elements at common positions:

$$(1^*) \quad append^*([X|T_X], [X|T_Y], [X]) \leftarrow$$
$$(2^*) \quad append^*([X|T_X], [Y|T_Y], E) \leftarrow append^*(T_X, T_Y, E)$$

The associated finite SLD-trees τ_B^* and τ_C^*, depicted in Fig. 2, are now almost fully identical. In that case, it is not useful to keep different specialised versions for B and C because the following single set of specialised clauses could be used for B and C without specialisation loss:

$$append^*([a|T_1], [a|T_2], [a]) \leftarrow$$

This illustrates that the syntactic structures of B and C alone provide insufficient information for a satisfactory control of polyvariance.

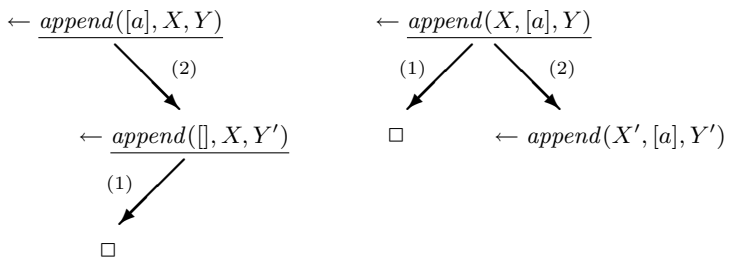

Fig. 1. SLD-trees τ_B and τ_C for Example 1

2.2 Characteristic trees

The above illustrates the interest of (also) examining the "essential structure" of the SLDNF-trees generated for the atoms to be partially deduced. This leads to the definition of *characteristic trees*, initially presented in [13, 12] and later exploited in [31, 33], which abstracts SLDNF-trees by only remembering, for the non-failing branches[1]:

[1] The failing branches do not materialise within the residual code and it is not interesting to know how a certain branch has failed; see [31, 33].

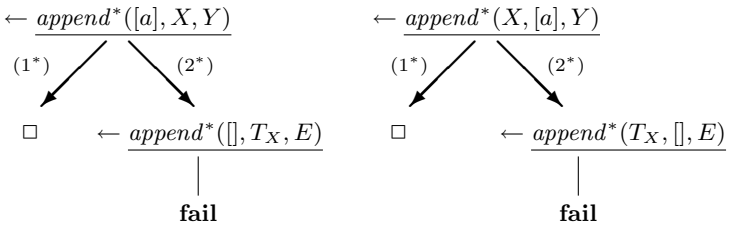

Fig. 2. SLD-trees τ_B^* and τ_C^* for Example 1

1. the position of the selected literals and
2. the (number of the) clauses that have been resolved with.

We will now use $pos \circ cl$ to denote a selection of a literal at position pos within a goal which is resolved with the clause numbered cl. We will represent trees by the set of their branches. For example, the characteristic trees of the finite SLD-trees τ_B and τ_C in Fig. 1 are then $\{\langle 1 \circ 2, 1 \circ 1\rangle\}$ and $\{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$ respectively. The characteristic trees of the finite SLD-trees τ_B^* and τ_C^* in Fig. 2 are both $\{\langle 1 \circ 1^*\rangle\}$.

The characteristic tree of an atom A explicitly or implicitly captures the following important aspects of specialisation:

- the branches that have been pruned through the unfolding process (namely those that are absent from the characteristic tree). For instance, by inspecting the characteristic trees of τ_B and τ_C from Ex. 1, we can see that two branches have been pruned for the atom B (thereby removing recursion) whereas no pruning could be performed for C.
- how deep $\leftarrow A$ has been unfolded and which literals and clauses have been resolved with each other in that process. This captures the computation steps that have already been performed at partial deduction time.
- the number of clauses in the resultants of A (namely one per characteristic path) and also (implicitly) which predicates are called in the bodies of the resultants. This means that a single predicate definition can (in principle) be used for two atoms which have the same characteristic tree.

Furthermore, Ex. 2 below (further examples can be found in [33]; similar situations also arise in the context of specialising metainterpreters) illustrate that sometimes a *growth* of syntactic structure (as spotted, e.g., by $\trianglelefteq$) is accompanied by a *shrinking* of the associated SLDNF-trees. In such situations there is, despite the growth of syntactic structure, actually no danger of non-termination. An abstraction operator solely focussing on the syntactic structure would unnecessarily force generalisation, thus often resulting in sub-optimal specialisation.

Example 2. Let P be the following definite program:

(1) $path([N]) \leftarrow$
(2) $path([X, Y|T]) \leftarrow arc(X, Y), path([Y|T])$
(3) $arc(a, b) \leftarrow$

Unfolding $\leftarrow path(L)$ (e.g., using an unfolding rule based on $\trianglelefteq$; see Fig. 3 for the SLD-trees constructed) will result in lifting $path([b|T])$ to the global level. Notice that we have a growth of syntactic structure ($path(L) \trianglelefteq path([b|T])$). However, one can see that further unfolding $path([b|T])$ results in a SLD-tree whose characteristic tree $\tau_B = \{\langle 1 \circ 1\rangle\}$ is strictly smaller than the one for $path(L)$ (which is $\tau_A = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 3\rangle\}$).

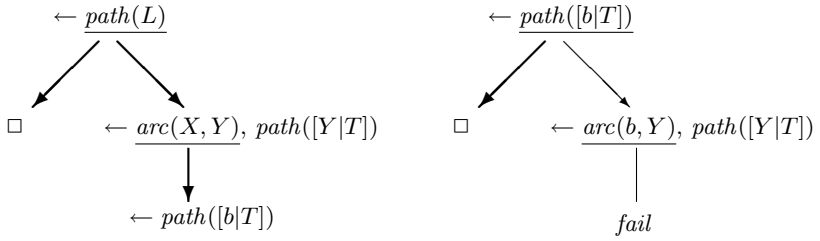

Fig. 3. SLD-trees for Example 2

In summary, characteristic trees seem to be an almost ideal vehicle for a refined control of polyvariance, a fact we will try to exploit in the following section.

2.3 An abstraction operator using characteristic trees

A first attempt at using characteristic might be as follows: classify atoms at the global control level by their associated characteristic tree and apply generalisation (*msg*) only on those atoms which have the same characteristic tree. The following example illustrates this approach.

Example 3. Let P be the program reversing a list using an accumulating parameter:

(1) $rev([], Acc, Acc) \leftarrow$
(2) $rev([H|T], Acc, Res) \leftarrow rev(T, [H|Acc], Res)$

We will use a (shower) determinate unfolding rule U inside the generic Algorithm 3.1 in [28].
When starting out with $\mathcal{A}_0 = \{rev([a|B], [], R)\}$ the following steps are performed by Algorithm 3.1:

- the only atom in $\mathcal{A}_0$ is unfolded (see Fig. 4) and the atoms in the leaves are added, yielding: $\mathcal{A}'_0 = \{rev([a|B], [], R), rev(B, [a], R)\}$.
- the atoms in $\mathcal{A}'_0$ all have different characteristic trees, and we obtain $\mathcal{A}_1 = \mathcal{A}'_0$.
- the atoms in $\mathcal{A}_1$ are unfolded (see Fig. 4) and the atoms in the leaves are added, yielding:
  $\mathcal{A}'_1 = \{rev([a|B], [], R), rev(B, [a], R), rev(T, [H, a], R)\}$.
- the atoms $rev(B, [a], R)$ and $rev(T, [H, a], R)$ have the same characteristic tree (see Fig. 4) and we thus apply the *msg* and obtain:
  $\mathcal{A}_2 = \{rev([a|B], [], R), rev(T, [A|B], R)\}$.
- the atoms in $\mathcal{A}_2$ are unfolded and the leaf atoms added:
  $\mathcal{A}'_2 = \{rev([a|B], [], R), rev(T, [A|B], R), rev(T', [H', A|B], R)\}$.
- the atoms $rev(T, [A|B], R)$ and $rev(T', [H', A|B], R)$ have the same characteristic tree and we thus apply the *msg* and obtain: $\mathcal{A}_3 = \mathcal{A}_2$. We have reached a fixpoint and thus obtain the following partial deduction satisfying the closedness condition (and which is also independent without renaming):

  $rev([a|B], [], R) \leftarrow rev(B, [a], R)$
  $rev([], [A|B], [A|B]) \leftarrow$
  $rev([H|T], [A|B], Res) \leftarrow rev(T, [H, A|B], Res)$

Because of the selective application of the *msg*, no loss of precision has been incurred, e.g., the pruning and pre-computation for $rev([a|B], [], R)$ has been preserved. An abstraction operator allowing just one version per predicate would have

lost this local specialisation, while a method with unlimited polyvariance (called dynamic renaming, in [1]) does not terminate.

For this example, our approach provides a terminating and fine grained control of polyvariance, conferring just as many polyvariant versions as necessary.

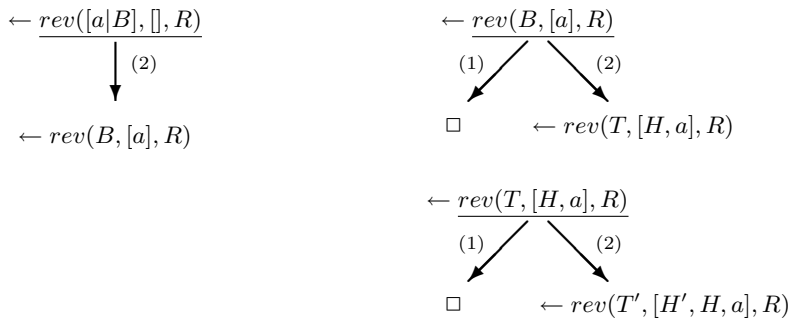

Fig. 4. SLD-trees for Example 3

The above example is thus very encouraging and one might hope that characteristic trees are always preserved upon generalisation and that we already have a refined solution to the control of polyvariance problem. Unfortunately, the approach still has two major problems:

1. it does not always preserve the characteristic trees, entailing a loss of precision and specialisation, and
2. it is not guaranteed to terminate (even if the number of distinct characteristic trees is finite).

We illustrate these problems and show how they can be overcame in the Sections 2.4 and 2.5 below.

2.4 Preserving characteristic trees upon generalisation

Let us show why the approach described in Section 2.3 does not preserve characteristic trees:

Example 4. Let P be the program:

(1) $p(X) \leftarrow q(X)$
(2) $p(c) \leftarrow$

Let $\mathcal{A} = \{p(a), p(b)\}$. Supposing that we do not unfold $q(X)$, $p(a)$ and $p(b)$ have the same characteristic tree $\tau = \{\langle 1 \circ 1 \rangle\}$. We thus calculate $msg(p(a), p(b)) = p(X)$ which has unfortunately the characteristic tree $\tau' = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2 \rangle\} \neq \tau$ and the pruning that was possible for the atoms $p(a)$ and $p(b)$ has been lost. More importantly, there exists *no* atom, more general than $p(a)$ and $p(b)$, which has τ as its characteristic tree.

The problem in the above example is that, through generalisation, a new non-failing derivation has been added, thereby modifying the characteristic tree. Another problem can occur when negative literals are selected by the unfolding rule. More involved and realistic examples can be found in [33, 31].

These losses of precision can have some regrettable consequences in practice:

– opportunities for specialisation can be lost and
– termination of Algorithm 3.1 in [28] can be undermined (even assuming a finite number of characteristic trees).

Two different solutions to this problem are:

1. *Ecological Partial Deduction* [25, 33]
   The basic idea is to simply *impose* characteristic trees on the generalised atoms. To solve Ex. 4 one would produce the generalisation $p(X)$ on which we impose $\tau = \{\langle 1 \circ 1 \rangle\}$ (this is denoted by $(p(X), \tau)$ in [25, 33], also called a *characteristic atom*). The residual code generated for $(p(X), \tau)$ is:
   $$p(X) \leftarrow q(X)$$
   In other words the pruning possible for $p(a)$ and $p(b)$ has now been preserved. However, the above residual code is not valid for all instances of $p(X)$; it is only valid for those instances (called concretisations) for which τ is a "proper" characteristic tree. For example, the code is valid for $p(a)$, $p(b)$, $p(d)$, but *not* for $p(c)$. The full details, along with correctness results, of this approach can be found in [25, 33].
2. *Constrained Partial Deduction* [31]
   The basic idea of constrained partial deductions is to, instead of producing a partial deduction for a set of atoms, to produce it for a set of *constrained atoms*. A constrained atom is a formula of the form $C \square A$, where A is an ordinary atom and C a constraint over some domain $\mathcal{D}$ (see also [21]). The set of "proper" instances (called concretisations) of a constrained atom $C \square A$ are then all the atoms $A\theta$ such that $C\theta$ holds in $\mathcal{D}$.
   [31] then achieves the preservation of characteristic trees by using *disequality constraints*, designed in such a way as to prune the possible computations into the right shape. To solve Ex. 4 one would produce the generalisation $X \neq c \square p(X)$. The residual code generated for this generalisation is the same as for ecological partial deduction, and again the pruning possible for $p(a)$ and $p(b)$ has been preserved:
   $$p(X) \leftarrow q(X)$$

The approach in [31] is more general and constraints are also propagated globally (i.e., in the terminology of supercompilation [44, 19, 43, 42], one can "drive negative information"). On the other hand, the method in [33] is conceptually simpler and can handle *any* unfolding rule as well as *normal* logic programs, while the concrete algorithm in [31] is currently limited to determinate unfoldings without a lookahead and definite programs.

2.5 Ensuring termination without depth-bounds

It turns out that for a fairly *large class of realistic programs* (and unfolding rules), the characteristic tree based approaches described above only terminate when imposing a depth bound on characteristic trees. [33] presents some natural examples which show that this leads to undesired results in cases where the depth bound is actually required. (These examples can also be adapted to prove a similar point about neighbourhoods in the context of supercompilation of functional programs.)

We illustrate the problem through a slightly artificial, but very simple example.

Example 5. The following is the reverse with accumulating parameter of Ex. 3 where a list type check (in the style of [14]) on the accumulator has been added.

(1) $rev([], Acc, Acc) \leftarrow$
(2) $rev([H|T], Acc, Res) \leftarrow ls(Acc), rev(T, [H|Acc], Res)$

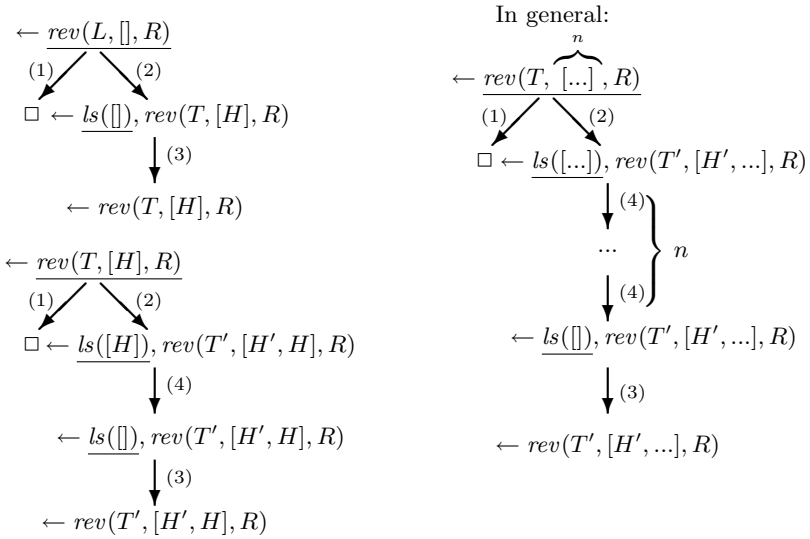

Fig. 5. SLD-trees for Example 5.

(3) $ls([]) \leftarrow$
(4) $ls([H|T]) \leftarrow ls(T)$

As can be noticed in Fig. 5, unfolding (determinate, $\trianglelefteq$-based, and well-founded, among others) produces an infinite number of different atoms, all with a different characteristic tree. Imposing a depth bound of say 100, we obtain termination; however, 100 different characteristic trees (and instantiations of the accumulator) arise, and 100 different versions of *rev* are generated: one for each characteristic tree. The specialised program thus looks like:

(1') $rev([], [], []) \leftarrow$
(2') $rev([H|T], [], Res) \leftarrow rev_2(T, [H], Res)$
(3') $rev_2([], [A], [A]) \leftarrow$
(4') $rev_2([H|T], [A], Res) \leftarrow rev_3(T, [H, A], Res)$
$\vdots$
(197') $rev_{99}([], [A_1, \ldots, A_{98}], [A_1, \ldots, A_{98}]) \leftarrow$
(198') $rev_{99}([H|T], [A_1, \ldots, A_{98}], Res) \leftarrow$
$rev_{100}(T, [H, A_1, \ldots, A_{98}], Res)$
(199') $rev_{100}([], [A_1, \ldots, A_{99}|AT], [A_1, \ldots, A_{99}|AT]) \leftarrow$
(200') $rev_{100}([H|T], [A_1, \ldots, A_{99}|AT], Res) \leftarrow$
$ls(AT), rev_{100}(T, [H, A_1, \ldots, A_{99}|AT], Res)$
(201') $ls([]) \leftarrow$
(202') $ls([H|T]) \leftarrow ls(T)$

This program is certainly far from optimal and clearly exhibits the ad hoc nature of the depth bound.

Situations like the above typically arise when an accumulating parameter influences the computation, because then the growing of the accumulator causes a corresponding growing of the characteristic trees. With most simple programs, this is not the case. For instance, in the standard reverse with accumulating parameter, the accumulator is only copied in the end, but never influences the computation.

For this reason it was generally felt that natural logic programs would give rise to only finitely many characteristic trees.

However, among larger and more sophisticated programs, cases like the above become more and more frequent, even in the absence of type-checking. For instance, in an explicit unification algorithm, one accumulating parameter is the substitution built so far. It heavily influences the computation because new bindings have to be added and checked for compatibility with the current substitution.

A solution to this problem is developed in [33], whose basic ingredients are as follows:

1. Register descendency relationships among atoms at the global level by putting them into a *global tree* (instead of a global set).
2. To watch over the evolution of the characteristic trees associated with atoms along the branches of the global tree in order to detect dangerous growths. Obviously, just measuring the depth of characteristic trees would be far too crude. As can be seen in Fig. 5, we need a more refined measure which would somehow spot when a characteristic tree (piecemeal) "contains" characteristic trees appearing earlier in the same branch of the global tree. If such a situation arises—as it indeed does in Ex. 5—it seems reasonable to stop expanding the global tree, generalise the offending atoms, and produce a specialised procedure for the generalisation instead. As shown in [33], this can be accomplished by extending the homeomorphic embedding relation $\trianglelefteq$ to work on characteristic trees.

The techniques formally elaborated in [33] have led to the implementation of the ECCE partial deduction system which is publicly available [26]. Extensive experiments are reported on in [33, 27]. The ECCE system also handles a lot of Prolog built-ins, like for instance =, is, <, =<, <, >=, *number*, *atomic*, *call*, \==, \=. All built-ins are supposed to be declarative and their selection delayed until they are sufficiently instantiated. These built-ins are then also registered within the characteristic trees (see [27]).

3 Conjunctive Partial Deduction

Partial deduction, based upon the Lloyd-Shepherdson framework [35], specialises a *set of atoms*. Even though conjunctions may appear within the SLDNF-trees constructed for these atoms, only atoms are allowed at the global level. In other words, when we stop unfolding, every conjunction at the leaf is automatically split into its atomic constituents which are then specialised (and possibly further abstracted) separately at the global level. As we show below, this restriction considerably restricts the potential power of partial deduction. The main goal of this section is to show how this limitation can be overcome, by going to the framework of *conjunctive partial deduction*.

3.1 Basics

Let us start by examining a very simple example. In the following, we introduce the connective $\wedge$ to avoid confusion between conjunction and the set punctuation symbol ",".

Example 6. Let P be the following program.

(1) $max\_length(X, M, L) \leftarrow max(X, M) \wedge length(X, L)$
(2) $max(X, M) \leftarrow max\_1(X, 0, M)$
(3) $max1([\,], M, M) \leftarrow$

(4) $max1([H|T], N, M) \leftarrow H \leq N \wedge max1(T, N, M)$
(5) $max1([H|T], N, M) \leftarrow H > N \wedge max1(T, H, M)$
(6) $length([\,], 0) \leftarrow$
(7) $length([H|T], L) \leftarrow length(T, K) \wedge L \; is \; K + 1$

Let us try to specialise this program for calls to $max\_length(x, m, l)$, which calculate the length and maximum element of a list. One can see that the original program above is needlessly inefficient at this: it traverses the list x twice, once to calculate the maximum and then again to compute the length. One might hope that by specialisation this inefficiency can be removed, i.e., that these two computations can be "tupled" together. Unfortunately this optimisation is out of the reach of partial deduction, due to its inability to handle conjunctions at the global level. For instance, assume that we construct the finite SLD-tree for $max\_length(x, m, l)$ depicted at the left in Fig. 6. Now the atoms $max\_1(x, 0, m)$ and $length(X, l)$ have to be specialised separately and the specialised program will contain two predicates each traversing x on its own. The situations remains the same, no matter how deeply we unfold $max\_length(x, m, l)$. In other words, partial deduction is incapable of translating multiple visits of the same data structure into a single visit (called *tupling*); something which can be achieved using unfold/fold program transformation methods [39].

To overcome this limitation, [32, 17, 27] present a minimal extension to partial deduction, called *conjunctive partial deduction*. This technique extends the standard partial deduction approach by simply considering a set $S = \{C_1, \ldots, C_n\}$ of *conjunctions of atoms* instead of just atoms.

Now, as the SLDNF-trees constructed for each C_i are no longer restricted to having *atomic* top-level goals, resultants (cf. Definition 25 in the first part of the course [28]) are not necessarily Horn clauses anymore: their left-hand side may contain a conjunction of atoms. To transform such resultants back into standard clauses, conjunctive partial deduction requires a *renaming* transformation, from conjunctions to atoms, in a post-processing step. We illustrate this below; the formal details are in [32, 17, 27].

Let us return to Ex. 6 to illustrate the basic workings of conjunctive partial deduction. Let $S = \{max\_length(x, m, l), max\_1(x, n, m) \wedge length(x, l)\}$ be the set of conjunctions we specialise. Assume that we construct the SLD-trees depicted in Fig. 6. The associated resultants are $\{R_{1,1}\}$ and $\{R_{2,1}, R_{2,2}, R_{2,3}\}$ with:

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

Clearly $R_{2,1}, R_{2,2}, R_{2,3}$ are not program clauses. Apart from that, with the exception that the redundant variable still has multiple occurrences, the above set of resultants 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 above into a standard logic program, we will rename conjunctions of atoms into atoms. Such renamings require some care. For one thing, there may be ambiguity concerning which conjunctions in the bodies to rename. For instance, if we have the resultant $p(x, y) \leftarrow r(x) \wedge q(y) \wedge r(z)$ and S contains $r(u) \wedge q(v)$, then either the first two, or the last two atoms in the body of this resultant are candidates for renaming. To formally fix such choices, we introduce the notion of a *partitioning function* p. Second, a particular conjunction in the body might be an instance of several elements in S (unless S is independent). Finally, we have to fix

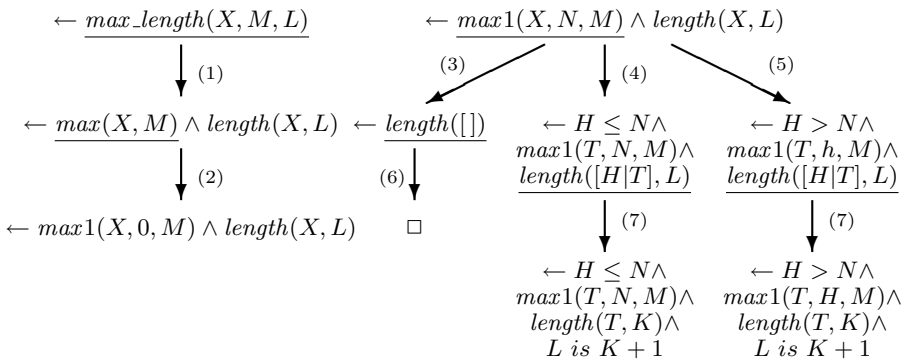

Fig. 6. SLD-trees τ_1 and τ_2 for Example 6

a mapping α (called an *atomic renaming* in [32]) from each element C_i of S to an atom A_i, having exactly the same variables as C_i, and such that each A_i uses a distinct predicate symbol.

For the *max_length* example, we simply use a partitioning function p which splits the conjunctions

$$H \leq N \wedge max1(T, N, M) \wedge length(T, K) \wedge L \; is \; K + 1$$
$$H > N \wedge max1(T, H, M) \wedge length(T, K) \wedge L \; is \; K + 1$$

into

$$\{H \leq N, \;\; max1(T, N, M) \wedge length(T, K), \;\; L \; is \; K + 1\}$$
$$\{H > N, \;\; max1(T, H, M) \wedge length(T, K), \;\; L \; is \; K + 1\}$$

respectively. Let us now map each element of S to an atom:
- $\alpha(max\_length(X, M, L)) = max\_length(X, M, L)$ and
- $\alpha(max1(X, N, M) \wedge length(X, L)) = ml(X, N, M, L)$.

S does not contain elements with common instances and for the resultants at hand there exists only one possible renaming based on α and p.

The conjunctive partial deduction wrt S is now obtained as follows. The head $max\_length(X, M, L)$ in $R_{1,1}$ is replaced by itself. The head-conjunctions $max1([\,], N, N) \wedge length([\,], 0)$ 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 three 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 the atoms $ml(X, 0, M, L)$, $ml(T, N, M, K)$ and $ml(T, H, M, K)$ respectively.
The resulting program is:

$max\_length(X, M, L) \leftarrow ml(X, 0, M, L)$
$ml([\,], N, N, 0) \leftarrow$
$ml([H|T], N, M, L) \leftarrow H \leq N \wedge ml(T, N, M, K) \wedge L \; is \; K + 1$
$ml([H|T], N, M, L) \leftarrow H > N \wedge ml(T, H, M, K) \wedge L \; is \; K + 1$

We were thus able to produce an ordinary logic program in which the two functionalities of *max* and *length* are accomplished in a single traversal.

3.2 Deforestation

In this section we show how conjunctive partial deduction can be used to get rid of intermediate data structures, something which is called *deforestation* [49].

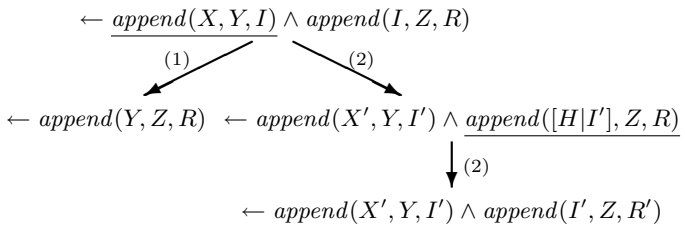

Fig. 7. SLD-tree for Example 7

Example 7. Let P be the *append* program from Ex. 1. One way to append *three* lists is to use the goal $append(Xs, Ys, I) \wedge append(I, Zs, R)$, which is simple and elegant, but inefficient to execute. Given Xs, Ys, Zs and assuming left-to-right execution, $append(Xs, Ys, I)$ constructs from Xs and Ys an intermediate list I which is then traversed to append Zs to it. We now show how conjunctive partial deduction offers salvation.

Let $S = \{append(X, Y, I) \wedge append(I, Z, R),\ append(X, Y, Z)\}$ and assume that we construct the finite SLD-tree τ_1 depicted in Fig. 7 for the query $\leftarrow append(X, Y, I) \wedge append(I, Z, R)$ as well as a simple tree τ_2 with a single unfolding step for $\leftarrow append(X, Y, Z)$, whose resultants are simply the original program P. For τ_1 we get the following resultants:

(R_1) $\quad append([\,], Y, Y) \wedge append(Y, Z, R) \leftarrow append(Y, Z, R)$
(R_2) $\quad append([H|X'], Y, [H|I']) \wedge append([H|I'], Z, [H|R']) \leftarrow append(X', Y, I') \wedge append(I', Z, R')$

Suppose that we use a partitioning function p which performs no partitioning, i.e., $p(B) = \{B\}$ for all conjunctions B. If we now take an atomic renaming α for S such that $\alpha(append(X, Y, I) \wedge append(I, Z, R)) = da(X, Y, I, Z, R)$ and $\alpha(append(X, Y, Z)) = append(X, Y, Z)$ (i.e. the distinct variables have been collected and have been ordered according to their first appearance), then the conjunctive partial deduction wrt S will contain, in addition to the original program P (re-created from the resultants of τ_2), the following:

(3) $\quad da([\,], Y, Y, Z, R) \leftarrow append(Y, Z, R)$
(4) $\quad da([H|X'], Y, [H|I'], Z, [H|R']) \leftarrow da(X', Y, I', Z, R')$

Executing $G = \leftarrow append(X, Y, I) \wedge append(I, Z, R)$ in the original program leads to the construction of an intermediate list I by $append(X, Y, I)$, which is then traversed again (consumed) by $append(I, Z, R)$. 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 called *redundant argument filtering* (RAF) presented in [34, 27]. The resulting specialised program then contains the original append program P as well as:

$(3')$ $\quad da([\,], Y, Z, R) \leftarrow append(Y, Z, R)$
$(4')$ $\quad da([H|X'], Y, Z, [H|R']) \leftarrow da(X', Y, Z, R')$

The unnecessary variable I, as well as the inefficiencies caused by it, have now been completely removed; i.e., we have achieved *deforestation*.

3.3 Diminished Need for Aggressive Local Control

In addition to enabling tupling- and deforestation-like optimisations, conjunctive partial deduction also solves a problem already raised in [38]. Take for example a metainterpreter 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 "classical" partial deduction faces an unsolvable dilemma, e.g., when specialising $solve(\bar{s})$, where $\bar{s}$ is some static data. Either it unfolds $clause(\bar{s}, B)$, thereby propagating the static data $\bar{s}$ over to $solve(B)$, but at the cost of duplicating $exp(\bar{s})$ and most probably leading to inefficient programs (cf. Ex. 6 in [28]). Or "classical" partial deduction can stop the unfolding, but then the partial input $\bar{s}$ can no longer be exploited inside $solve(B)$ as it will be specialised in isolation. Using conjunctive partial deduction however, we can be efficient *and* propagate information at the same time, simply by stopping unfolding and specialising the conjunction $clause(\bar{s}, B) \wedge solve(B)$.

In other words, the local control no longer has to be clever about propagating partial input ($\bar{s}$) from one atom ($clause(\bar{s}, B)$) to the other ($solve(B)$); it can concentrate solely on efficiency concerns (not duplicating the expensive $exp(\bar{s})$). Conjunctive partial deduction therefore diminishes the need for aggressive unfolding rules (a claim empirically verified in [24, 27]) and allows to reconcile precision and efficiency.

3.4 Global Control and Implementation

A termination problem specific to conjunctive partial deduction lies in the possible appearance of ever growing conjunctions at the global level. To cope with this, abstraction in the context of conjunctive partial deduction must include the ability to *split* a conjunction into several parts, thus producing *subconjunctions* of the original one (cf. Ex. 6). A method to deal with this problem has been developed in [17, 9].

Apart from this aspect, the conventional control notions described earlier also apply in a conjunctive setting. Notably, the concept of characteristic trees can be generalised to handle conjunctions. The ECCE system [26], discussed earlier, has been extended to handle conjunctive partial deduction and the extensive experiments conducted in [24, 27] suggest that it was possible to consolidate partial deduction and unfold/fold program transformation, incorporating most of the power of the latter while keeping the automatic control and efficiency of the former. There are, however, still some practical limitations of the ECCE system concerning tupling and deforestation (getting rid of these limitations is a topic of further research, see [24, 27]).

3.5 Conjunctive Partial Deduction and Supercompilation

Partial deduction and related techniques in functional programming are often very similar [18] (and cross-fertilisation has taken place). Actually, conjunctive partial deduction has in part been inspired by supercompilation of functional programming (and by unfold/fold transformation techniques [39]) and the techniques have a lot in common. However, there are still some subtle differences. Notably, while conjunctive partial deduction can perform deforestation *and* tupling, supercompilation [19, 43] is incapable of achieving tupling. On the other hand, the techniques developed for tupling of functional programs [5, 6] are incapable of performing deforestation.

The reason for this extra power conferred by conjunctive partial deduction, is that conjunctions with shared variables can be used both to elegantly represent *nested function calls*

$$f(g(X)) \quad \mapsto \quad g(X, \underline{ResG}) \wedge f(\underline{ResG}, Res)$$

as well as *tuples*

$$\langle f(X), g(X) \rangle \quad \mapsto \quad g(\underline{X}, ResG) \wedge f(\underline{X}, ResF)$$

or any mixture thereof. The former enables deforestation while the latter is vital for tupling, explaining why conjunctive partial deduction can achieve both.

Let us, however, also note that actually achieving the tupling or deforestation in a logic programming context can be harder. For instance, in functional programming we know that for the same function call we always get the same, unique output. This is often important to achieve tupling, as it allows one to replace multiple function calls by a single call. For example we can safely transform $fib(N) + fib(N)$ into `let` $X = fib(N)$ `in` $X + X$. In the context of logic programming it is, however, generally unsafe to transform the corresponding conjunction $fib(N, R1) \wedge fib(N, R2) \wedge Res\ is\ R1 + R2$ into $fib(N, R) \wedge Res\ is\ R + R$. If, e.g., fib is defined by the facts $fib(N, 1)$ and $fib(N, 2)$ then the first conjunction allows for three results $Res = 2, 3, 4$ while the second conjunction only allows $Res = 2, 4$. The above transformation is thus generally unsafe, unless we are certain that fib behaves like a function. Tupling in logic programming thus often requires one to establish *functionality* of the involved predicates. This can for instance be done via the approach presented in the next section or via (correct) user declarations (e.g., "`:- mode fib(i,o) is determinate.`").

Furthermore, in functional programming function calls cannot *fail* while predicate calls in logic programming can. This means that *reordering* calls in logic programming can induce a change in the termination behaviour; something which is not a problem in (pure) functional programming. For instance reordering the conjunction $fail \wedge loop$ (where *loop* is a predicate that does not terminate) into $loop \wedge fail$ will change the Prolog termination behaviour: the former conjunction fails finitely while the latter does not terminate. Unfortunately, reordering is often required to achieve deforestation or tupling (although it was not required for the examples treated earlier). Let for example r and p be two predicates taking a binary tree as input and producing a modified tree as output. If we apply conjunctive partial deduction to $r(In, T) \wedge p(T, Out)$ we will typically get in the leaf of one of the branches of the SLD-tree the conjunction $r(InL, TL) \wedge r(InR, TR) \wedge p(TL, OutL) \wedge p(TR, OutR)$ where $InL, TL, OutL$ ($InR, TR, OutR$ respectively) are the left (right respectively) subtrees of L, T and Out. To achieve deforestation we need to reorder this conjunction into $r(InL, TL) \wedge p(TL, OutL) \wedge r(InR, TR) \wedge p(TR, OutR)$ so as to be able to eventually produce a residual conjunction such as $rp(InL, OutL) \wedge rp(InR, OutR)$. This means that to actually achieve deforestation or tupling in logic programming one often needs an additional analysis to ensure that termination is preserved [3, 2].

4 Incorporating Abstract Interpretation

The main idea of *abstract interpretation* [8, 4, 20] is to analyse programs by executing them over some *abstract domain*. This is done in such a way as to ensure termination of the abstract interpretation and to ensure that the so derived results are a *safe approximation* of the programs' concrete runtime behaviour(s).

Abstract interpretation has already been used successfully as a post-processing optimisation [10, 14, 11] and it is often felt that there is a close relationship between abstract interpretation and program specialisation. Recently, there has been a lot of interest in the integration of these two techniques [30, 23, 41, 40, 47, 29, 48]. In this section we illustrate, on a simple example in the context of logic programming, why this integration is a worthwhile goal.

Example 8. Take the following simple program:

$app\_last(L, X) \leftarrow append(L, [a], R) \wedge last(R, X)$
$append([], L, L) \leftarrow$
$append([H|X], Y, [H|Z]) \leftarrow append(X, Y, Z)$

$last([X], X) \leftarrow$
$last([H|T], X) \leftarrow last(T, X)$

One would hope that some of the specialisation techniques we have seen so far are sufficiently strong to infer that a query $\leftarrow app\_last(L, X)$ only produces answers where $X = a$. Unfortunately, this is not the case. Even more surprisingly, most abstract interpretation techniques proposed in the literature, such as [22, 15, 37], on their own are incapable of deriving this result.

This is a very simple example where a statically known value (a) is stored (using *append*) in an unknown datastructure (L) and then later consulted (using *last*). More involved and realistic examples occur in, e.g., interpreters for imperative languages (where variable bindings are stored in some environment and then later consulted again) or explicit unification algorithms. Not being able to solve Ex. 8 means that any value stored in some (partially) unknown datastructure is lost for successive specialisation. This limitation is thus very regrettable for a lot of practical applications.

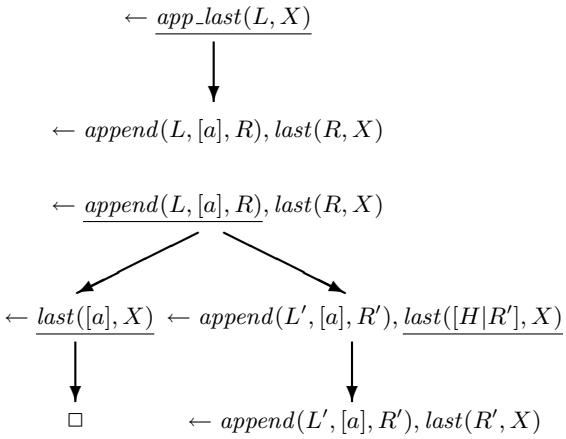

Fig. 8. SLD-trees for Ex. 8

The reason why most abstract interpretation techniques are incapable of solving Ex. 8 is that they analyse atoms *separately*. In this program (and many, much more relevant others), we are interested in analysing the conjunction $append(L, [a], R) \wedge last(R, X)$ with a linking intermediate variable (whose structure is too complex for the particular abstract domain). If we could consider this conjunction as a *basic unit*, and therefore not perform abstraction on the separate atoms, but only on conjunctions of the involved atoms, we would retain a precise side-ways information passing analysis. This is exactly what can be achieved by combining the abstract interpretation with conjunctive partial deduction (the latter can also propagate goal dependent information).

Let us illustrate how conjunctive partial deduction combined with a simple abstract interpretation technique from [36] *does* solve Ex. 8. Starting from the atom $app\_last(X)$ and using straightforward control for conjunctive partial deduction [24], we can obtain $S = \{\ app\_last(X),\ append(L, [a], R) \wedge last(R, X)\ \}$ and the corresponding SLD-trees in Fig. 8.

Using a renaming transformation based on $\alpha(append(x, y, z) \wedge last(z, u)) = al(x, y, z, u)$ the resulting transformed program is:

$app\_last(L, X) \leftarrow al(L, [a], R, X)$
$al([], [a], [a], a) \leftarrow$
$al([H|L'], [a], [H|R'], X) \leftarrow al(L', [a], R', X)$

Notice that conjunctive partial deduction *alone* does not yet produce the desired result (X is not instantiated to a in the first clause). This is due to a lack of inference of global success information, i.e., being unable to extract information from an infinite number of different computation paths. If the length of the list L were given, then (conjunctive) partial deduction could obtain the desired result simply by unfolding. Here, however, the length of L is unknown, and an infinite number of unfoldings would be required. It is here that abstract interpretation comes to the rescue, as it can infer information about an infinite number of different computation paths. One way [36] this can be done, is by examining the above program bottom-up:

1. Start by assuming that every call fails: we approximate the *success set* (i.e., those atoms that succeed) by $S_1 = \emptyset$ and set $i = 1$.
2. Unify every atom in the body of a clause with an element of S_i and instantiate the head accordingly. Put the so obtained heads into S_{i+1}.
3. Apply the *msg* predicatewise to S_{i+1} and stop if $S_{i+1} = S_i$; otherwise increment i and goto step 2.

Performing these steps on our residual program, we obtain the following scenario. First we unify the body atoms with elements of $S_1 = \emptyset$:

$app\_last(L, X) \leftarrow fail$
$al([], [a], [a], a) \leftarrow$
$al([H|L'], [a], [H|R'], X) \leftarrow fail$

Examining each clause we get as second approximation of the success set $S_2 = \{al([], [a], [a], a)\}$. Unifying the body atoms with elements of S_2 gives:

$app\_last([], a) \leftarrow al([], [a], [a], a)$
$al([], [a], [a], a) \leftarrow$
$al([H], [a], [H, a], a) \leftarrow al([], [a], [a], a)$

We thus get as third approximation of the success set $S_3 = \{al([], [a], [a], a), al([H], [a], [H, a], a), app\_last([], a)\}$. To ensure termination we apply the *msg* predicate-wise and obtain $S_3 = \{al(X, [a], Y, a),\ app\_last([], a)\}$. Unifying the body atoms with elements of S_3 gives:

$app\_last([], a) \leftarrow al([], [a], R, a)$
$al([], [a], [a], a) \leftarrow$
$al([H|L'], [a], [H|R'], a) \leftarrow al(L', [a], R', a)$

At the next step we obtain $S_4 = \{al(X, [a], Y, a),\ app\_last(L, a)\}$ as well as the instantiated program P_4:

$app\_last(L, a) \leftarrow al(L, [a], R, a)$
$al([], [a], [a], a) \leftarrow$
$al([H|L'], [a], [H|R'], a) \leftarrow al(L', [a], R', a)$

We now obtain $S_5 = S_4$ and our abstract interpretation is complete. This residual program P_4 now explicitly contains the information that $X = a$ in $app\_last(L, X)$. According to the results in [36] we can actually use this program in place of the original residual program delivered by conjunctive partial deduction (computed answers and finite failure is preserved; however, infinite failure might be replaced by finite one).

Note, that the above technique *fails* to deliver this information when applied to the original program, even after a magic-set transformation (see [30, 27]). Via

filtering and redundant argument filtering (cf. Section 3.2) we can even further simplify this into the "optimal"[2] program:

$app\_last(L, a) \leftarrow al(L)$
$al([]) \leftarrow$
$al([H|L']) \leftarrow al(L')$

In addition to the already stated applications, the combination of conjunctive partial deduction and abstract interpretation is an elegant way to infer functionality of predicates [30, 27], perform some basic inductive theorem proving tasks[3] [30, 27], sophisticated program inversion tasks, as well as software verification tasks such as model checking. For further details, comprising algorithms and technical results, we refer to [30, 27, 29].

5 Conclusion and Outlook

In Section 2 we have investigated the problematic question of when is it sensible to generate different specialised versions for a particular predicate and when is it sensible to perform abstraction instead. For this recurring, difficult problem, termed the control of polyvariance problem, we presented the advantages of characteristic trees over a purely syntactic approach. We thereafter illustrated some of the technical difficulties of using characteristic trees in practice, related to precision and termination, and have shown how they can be overcome.

The control of polyvariance problem occurs in different disguises in many areas of program analysis, manipulation and optimisation. We therefore believe that the presented techniques can be adapted for other (declarative) programming paradigms and that they might prove equally useful in the context of, e.g., abstract interpretation systems or optimising compilers.

Section 3 was aimed at augmenting the power of partial deduction. Indeed, partial deduction was heretofore incapable of performing certain useful unfold/fold transformations, like tupling or deforestation. We presented the framework of conjunctive partial deduction which, by specialising conjunctions instead of individual atoms, is able to accommodate these optimisations. Deforestation and tupling like transformation are useful even in the absence of partial input. This warrants the integration of the presented techniques into a compiler, as their systematic use might prove to be highly beneficial and allow users to more easily decompose and combine procedures and programs without having to worry about the ensuing inefficiencies.

In Section 4 we illustrated that abstract interpretation and (conjunctive) partial deduction have limitations on their own and that a combination of these techniques might therefore be extremely useful in practice. We have illustrated how this can be accomplished on a simple example and have hinted at the benefits of such an integration for practical applications.

Acknowledgements

The author greatly benefited from discussions and joint work with Maurice Bruynooghe, Danny De Schreye, Robert Glück, Jesper Jørgensen, Neil Jones, Bern Martens, and Morten Heine Sørensen.

[2] In the absence of type information we have to keep the call to $al(L)$, ensuring that we deliver $X = a$ only if L is actually a list (see also [29]).

[3] The relation between program specialisation and *(inductive) theorem proving* has already been raised several times in the literature [45, 16, 46].

References

1. K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, October 1993.
2. A. Bossi and N. Cocco. Replacement can Preserve Termination. In J. Gallagher, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'96*, LNCS 1207, pages 104–129, Stockholm, Sweden, August 1996. Springer-Verlag.
3. A. Bossi, N. Cocco and S. Etalle. Transformation of Left Terminating Programs: The Reordering Problem. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, pages 33–45, Utrecht, Netherlands, September 1995. Springer-Verlag.
4. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10:91–124, 1991.
5. W.-N. Chin. Towards an automated tupling strategy. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132. ACM Press, 1993.
6. W.-N. Chin and S.-C. Khoo. Tupling functions with multiple recursion parameters. In *Proceedings of the Third International Workshop on Static Analysis*, number 724 in LNCS 724, pages 124–140, Padova, Italy, Sept. 1993. Springer-Verlag.
7. C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.
8. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
9. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms and experiments. *The Journal of Logic Programming*, 1999. To appear.
10. D. A. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'91*, pages 205–220, Manchester, UK, 1991.
11. D. A. 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.
12. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.
13. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3 & 4):305–333, 1991.
14. J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 151–167, Manchester, UK, 1992.
15. J. Gallagher and D. A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613. The MIT Press, 1994.
16. R. Glück and J. Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Proceedings of SAS'94*, LNCS 864, pages 432–448, Namur, Belgium, September 1994. Springer-Verlag.
17. R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag.
18. R. Glück and M. H. Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings, Proceedings of PLILP'94*, LNCS 844, pages 165–181, Madrid, Spain, 1994. Springer-Verlag.
19. R. Glück and M. H. Sørensen. A roadmap to supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 137–160, Schloß Dagstuhl, 1996. Springer-Verlag.
20. M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *The Journal of Logic Programming*, 13(4):349–366, 1992.

21. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–581, 1994.
22. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *The Journal of Logic Programming*, 13(2 & 3):205–258, 1992.
23. N. D. Jones. Combining abstract interpretation and partial evaluation. In P. Van Hentenryck, editor, *Static Analysis, Proceedings of SAS'97*, LNCS 1302, pages 396–405, Paris, 1997. Springer-Verlag.
24. J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag.
25. M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, pages 1–16, Utrecht, The Netherlands, September 1995. Springer-Verlag.
26. M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~dtai`, 1996.
27. M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, K.U. Leuven, May 1997.
Accessible via `http://www.cs.kuleuven.ac.be/~michael`.
28. M. Leuschel. *Logic Program Specialisation*. In *this volume*.
29. M. Leuschel. Program specialisation and abstract interpretation reconciled. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'98*, pages 220–234, Manchester, UK, June 1998. MIT Press.
30. M. Leuschel and D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 137–151, Aachen, Germany, September 1996. Springer-Verlag.
31. M. Leuschel and D. De Schreye. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing*, 16(3):283–342, 1998.
32. M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.
33. M. Leuschel, B. Martens, and 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, January 1998.
34. M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 83–103, Stockholm, Sweden, August 1996. Springer-Verlag.
35. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.
36. K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990.
37. A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-structure data-flow analysis for prolog. *ACM Transactions on Programming Languages and Systems*, 16(2):205–258, 1994.
38. 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.
39. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, 19& 20:261–320, May 1994.
40. G. Puebla, J. Gallagher, and M. Hermenegildo. Towards integrating partial evaluation in a specialization framework based on generic abstract interpretation. In M. Leuschel, editor, *Proceedings of the ILPS'97 Workshop on Specialisation of Declarative Programs and its Application*, K.U. Leuven, Tech. Rep. CW 255, pages 29–38, Port Jefferson, USA, October 1997.

41. G. Puebla and M. Hermenegildo. Abstract specialization and its application to program parallelization. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 169–186, Stockholm, Sweden, August 1996.
42. M. H. Sørensen. *Introduction to Supercompilation.* In *this volume.*
43. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
44. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
45. V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.
46. V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 482–509, Schloß Dagstuhl, 1996. Springer-Verlag.
47. W. Vanhoof. Bottom-up information propagation for partial deduction. In M. Leuschel, editor, *Proceedings of the ILPS'97 Workshop on Specialisation of Declarative Programs and its Application*, K.U. Leuven, Tech. Rep. CW 255, pages 73–82, Port Jefferson, USA, October 1997.
48. W. Vanhoof, B. Martens, D. De Schreye, and K. De Vlaminck. Specialising the other way around. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'98*, Manchester, UK, June 1998. MIT Press.
49. P. Wadler. Deforestation: Transforming programs to eliminate intermediate trees. *Theoretical Computer Science*, 73:231–248, 1990. Preliminary version in ESOP'88, LNCS 300.