Controlling Generalisation and Polyvariance in Partial Deduction of Normal Logic Programs

Michael Leuschel *Bern Martens*
Danny De Schreye

Report CW 248, February 1997

Department of Computing Science, K.U.Leuven

Abstract

In this paper, we further elaborate global control for partial deduction: For which atoms, among possibly infinitely many, should partial deductions be produced, meanwhile guaranteeing correctness as well as termination, and providing ample opportunities for fine-grained polyvariance?

Our solution is based on two ingredients. First, we use the well-known concept of a characteristic tree to guide abstraction (or generalisation) and polyvariance, and aim for producing one specialised procedure per characteristic tree generated. Previous work along this line failed to provide abstraction correctly dealing with characteristic trees. We show how this can be rectified in an elegant way. Secondly, we structure combinations of atoms and associated characteristic trees in global trees registering "causal" relationships among such pairs. This will allow us to spot looming non-termination and consequently perform proper generalisation in order to avert the danger, without having to impose a depth bound on characteristic trees. Leaving unspecified the specific local control one may wish to plug in, the resulting global control strategy enables partial deduction that always terminates in an elegant, non ad hoc way, while providing excellent specialisation as well as fine-grained (but reasonable) polyvariance.

Controlling Generalisation and Polyvariance in Partial Deduction of Normal Logic Programs

Michael Leuschel* Bern Martens† Danny De Schreye‡

Departement Computerwetenschappen
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail : {michael,bern,dannyd}@cs.kuleuven.ac.be

Abstract

Given a program and some input data (i.e. a partially instantiated query), partial deduction computes a specialised program that should handle remaining input (i.e. further instantiated queries) more efficiently than the original program. With the partial data, a computation is executed, and the resulting program is synthesised from this. Ideally, this process can be as precise as concrete, full evaluation and then leads to a maximally specialised output program. However, more often than not, such computations will not terminate. Hence, the need for special provisions arises, and this matter has been the subject of intensive research on controlling partial deduction.

In this setting, a clear conceptual distinction has recently emerged between local and global control, and on both levels concrete strategies as well as general frameworks have been proposed. In this paper, we further elaborate the crucial issue of global control: For which atoms, among possibly infinitely many, should partial deductions be produced, meanwhile guaranteeing correctness as well as termination, and providing ample opportunities for fine-grained polyvariance?

Our solution is based on two ingredients. First, we use the well-known concept of a characteristic tree to guide abstraction (or generalisation) and polyvariance, and aim for producing one specialised procedure per characteristic tree generated. Previous work along this line failed to provide abstraction correctly dealing with characteristic trees. We show how this can be rectified in an elegant way. Secondly, we structure combinations of atoms and associated characteristic trees in global trees registering "causal" relationships among such pairs. This will allow us to spot looming non-termination and consequently perform proper generalisation in order to avert the danger, without having to impose a depth bound on characteristic trees. Leaving unspecified the specific local control one may wish to plug in, the resulting global control strategy enables partial deduction that always terminates in an elegant, non ad hoc way, while providing excellent specialisation as well as fine-grained (but reasonable) polyvariance.

A similar approach may contribute to improve upon current (on-line) control strategies for functional program transformation methods such as (positive) supercompilation. It might also prove valuable in the context of abstract interpretation to handle infinite domains of infinite height with more precision.

*Supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation".

†Postdoctoral Fellow of the K.U.Leuven Research Council, Belgium; Also partially supported by the HCM Network "Logic Program Synthesis and Transformation", contract nr. CHRX-CT-93-00414, University of Padova, Italy.

‡Senior Research Associate of the Belgian National Fund for Scientific Research.

Finally, the practical relevance and benefits of this approach are illustrated through extensive experiments.

Keywords: Partial Deduction, Partial Evaluation, Program Transformation, Logic Programming, Flow analysis, Supercompilation.
CR Subject Classification: I.2.2 [**Artificial Intelligence**]: Automatic Programming, D.1.2 [**Programming Techniques**]: Automatic Programming, I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving — *logic programming*, F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic — *logic programming*, D.1.6 [**Programming Techniques**]: Logic Programming.

1 Introduction

A major concern in the specialisation of functional (see e.g. [8, 31, 67]) as well as logic programs (see e.g [49, 35, 19, 57, 10]) has been the issue of control: How can the transformation process be guided in such a way that termination is guaranteed and results are satisfactory?

This problem has been tackled from two (until now) largely separate angles: the so-called *off-line* versus *on-line* approaches. Partial evaluation of functional programs [8, 31] has mainly stressed the former, while supercompilation of functional [67, 68, 65] and partial deduction of logic programs [21, 64, 4, 6, 52, 54] have concentrated on on-line control. (Some exceptions are [69, 56, 38, 33].)

Basically, within the off-line approach, an analysis phase (by hand and/or automatically), prior to the specialisation proper, provides annotations that guide the specialiser when it has to decide on control issues such as structure propagation, termination and the right amount of polyvariance. Separating out this part of control simplifies considerably the core specialisation program, thus often reducing the overall complexity of the transformation process. It also renders specialisation more readily amenable to self-application [32, 26] and the realisation of the well-known Futamura projections.

Most specialisation research in logic programming has not been concerned with anything beyond the first Futamura (or specialiser [24]) projection. Self-application consequently being less of an issue, most specialisers for logic programs focus on (fully automatic) on-line control.[1]

It is within this well established on-line control tradition that the present work provides a novel and important contribution.

In partial deduction of logic programs, one distinguishes two levels of control [19, 54]: the local and the global level. In a nutshell, the local level decides on how SLD(NF)-trees for individual atoms should be built. The (branches of the) resulting trees allow to construct specialised clauses for the given atoms [49, 3]. At the global level on the other hand, one typically attends to the overall correctness of the resulting program (satisfying the closedness condition in [49]) and strives to achieve the "right" amount of polyvariance, producing sufficiently many (but not too much) specialised versions for each predicate definition in the original program. At both levels, in the context of providing a fully

[1] There may be other reasons involved, such as the occurrence of partially instantiated data structures both at partial deduction and at run time, and the unification based data propagation (as opposed to constant based in partial evaluation of functional programs), but a detailed analysis of these and related aspects is beyond the scope of this paper.

automatic tool, termination is obviously also of prime importance.

Gallagher in [19] writes that providing adequate global control seems much harder than handling the local level. So, in this paper, it is to the latter, global, level that we turn our attention. The concept of *characteristic trees* has been advocated as a suitable and refined basis for the global control of partial deduction by Gallagher and Bruynooghe in [21, 18]. The main idea is, instead of using the *syntactic* structure to decide upon polyvariance, to examine the specialisation *behaviour* of the atoms to be specialised: only if this behaviour is sufficiently different from one atom to the other should different specialised versions be generated. However, the approaches based on characteristic trees have been, up to now, ridden by two major problems. First the approaches failed to preserve the characteristic trees upon generalisation (which is needed to ensure termination). In other words two atoms with identical specialisation behaviour may be replaced in the generalisation process by an atom with a completely different specialisation behaviour. This may lead to significant losses in specialisation, as well as to problems for the termination of the partial deduction process. Secondly, the characteristic trees have always been limited by some ad hoc depth bound, possibly leading to very undesirable specialisation results (as we will show later in the paper).

In the present paper, we endeavour to solve these two problems. After presenting some required background about partial deduction and characteristic trees in Section 2, we solve the first problem in Section 3 by developing the ecological partial deduction principle, ensuring the preservation of characteristic trees. In Section 4 we then solve the second problem, blending (a slightly adapted version of) the general framework in [54] with the framework developed in Section 3. We thus obtain an elegant, sophisticated and precise apparatus for on-line global control of partial deduction. Extensive benchmarks, illustrating the practical benefits of this approach, as well as some discussions, can be found in Section 5, while . Section 6 subsequently concludes the paper.

Finally, we would like to mention that part of the material in this paper was presented in a somewhat preliminary form in two earlier workshop papers:

- [39] describes how to impose characteristic trees and perform set-based partial deduction with characteristic atoms,
- while [45] elaborates the former approach into one using global trees, and thus not requiring any depth bound.

2 Preliminaries and Motivations

Throughout this paper, we suppose familiarity with basic notions in logic programming [48] and partial deduction [49]. Notational conventions are standard and self-evident. In particular, in programs, we denote variables through strings starting with (or usually just consisting of) an upper-case symbol, while the notations of constants, functions and predicates begin with a lower-case character. Unless stated explicitly otherwise, the terms "(logic) program" and "goal" will refer to a *normal* logic program and goal, respectively. As common in partial deduction, the notion of SLDNF-trees is extended to also allow *incomplete* SLDNF-trees which, in addition to success and failure leaves, may also contain leaves where no literal has been selected for a further derivation step. Leaves of the latter kind will be called *dangling* [52].

2.1 Partial Deduction

In contrast to (full) evaluation, a *partial evaluator* is given a program P along with a *part* of its input, called the *static input*. The remaining part of the input, called the *dynamic input* will only be known or given at some later point in time. Given the static input S, the partial evaluator then produces a *specialised* version P_S of P which, when given the dynamic input D, produces the same output as the original program P.

In the context of logic programming, full input to a program P consists of a goal G and evaluation corresponds to constructing a complete SLDNF-tree for $P \cup \{G\}$. The partial input, that a partial evaluator is given, then takes the form of a *partially instantiated* goal G'. One often uses the term *partial deduction* to refer to partial evaluation of pure logic programs (see [35]), a convention we will adhere to in this paper. So, given this goal G' and the original program P, partial deduction then produces a new program P' which is P "specialised" to the goal G': P' can be called for all instances of G', producing the same output as P, but often much more efficiently.

In order to avoid constructing infinite SLDNF-trees for partially instantiated goals, partial deduction is based on constructing finite, but possibly *incomplete* SLDNF-trees for a set of atoms $\mathbf{A}$. The derivation steps in these SLDNF-trees correspond to the computation steps which have been performed beforehand by the *partial deducer* and the clauses of the specialised program are then extracted from these trees by constructing one specialised clause per branch. The incomplete SLDNF-trees are obtained by applying an unfolding rule, defined as follows:

Definition 2.1 An *unfolding rule* U is a function which, given a program P and a goal G, returns a finite and possibly incomplete SLDNF-tree for $P \cup \{G\}$.

For reasons to be clarified later, Definition 2.1 is so general as to even allow a *trivial* SLDNF-tree, i.e. one whose root is a dangling leaf.

The resulting specialised clauses are extracted from the incomplete SLDNF-trees in the following manner:

Definition 2.2 Let P be a program and A an atom. Let τ be a finite, incomplete and non-trivial[2] SLDNF-tree for $P \cup \{\leftarrow A\}$. Let $\leftarrow G_1, \ldots, \leftarrow G_n$ be the goals in the (non-root) leaves of the non-failing branches of τ. Let $\theta_1, \ldots, \theta_n$ be the computed answers of the derivations from $\leftarrow A$ to $\leftarrow G_1, \ldots, \leftarrow G_n$ respectively. Then the set of resultants $resultants(\tau)$ is defined to be $\{A\theta_1 \leftarrow G_1, \ldots, A\theta_n \leftarrow G_n\}$.

Partial deduction, as defined e.g. in [49, 3], uses the resultants for a given set of atoms $\mathbf{A}$ to construct the specialised program and for each atom in $\mathbf{A}$ a different specialised predicate definition is generated, replacing the original definition in P. Under the conditions stated in [49], namely closedness and independence, correctness of the specialised program is guaranteed. *Independence* requires that no two atoms in $\mathbf{A}$ have a common instance. This ensures that no two specialised predicate definitions match the same (run-time) call. Usually this condition is satisfied by performing a renaming of the atoms in $\mathbf{A}$, see [20, 2]. *Closedness* (as well as the related notion of *coveredness* in [3]) basically requires that every atom in the body of a resultant is matched by a specialised

[2] To avoid the problematic resultant $A \leftarrow A$.

predicate definition. This guarantees that $\mathbf{A}$ forms a *complete description* of all possible computations that can occur at run-time of the specialised program.

The main practical difficulty of partial deduction is the *control of polyvariance* problem, which boils down to finding a *terminating* procedure to produce a *finite* set of atoms $\mathbf{A}$ which satisfies the above *correctness* conditions while at the same time providing as much potential for *specialisation* as possible.

2.2 Abstraction

Termination is usually obtained by using a suitable abstraction operator, defined as follows:

Definition 2.3 Let $\mathbf{A}$ and $\mathbf{A}'$ be sets of atoms. Then $\mathbf{A}'$ is an *abstraction* of $\mathbf{A}$ iff every atom in $\mathbf{A}$ is an instance of an atom in $\mathbf{A}'$. An *abstraction operator* is an operator which maps every finite set of atoms to a finite abstraction of it.

The following generic scheme, based on a similar one in [18, 19], describes the basic layout of practically all algorithms for controlling partial deduction.

Algorithm 2.4
Input: A program P and a goal G
Output: A specialised program P'
Initialise: $i = 0$, $\mathbf{A}_i = \{A \mid A \text{ is an atom in } G\}$
repeat
  for each $A_k \in \mathbf{A}_i$, compute a finite SLDNF-tree τ_k for $P \cup \{\leftarrow A_k\}$ by applying an unfolding rule U;
  let $\mathbf{A}'_i := \mathbf{A}_i \cup \{B_l \mid B_l$ is an atom in a leaf of some tree τ_k, such that
    B_l is not an instance (respectively variant) of any $A_j \in \mathbf{A}_i\}$;
  let $\mathbf{A}_{i+1} :=$ *abstract*$(\mathbf{A}'_i)$ where *abstract* is an abstraction operator
until $\mathbf{A}_{i+1} = \mathbf{A}_i$
Apply a renaming transformation to $\mathbf{A}_i$ to ensure independence and then construct P' by taking resultants.

In itself the use of an abstraction operator does not yet guarantee global termination. But if the above algorithm terminates then independence and coveredness are ensured. With this observation we can reformulate the *control of polyvariance problem* as one of finding an *abstraction operator which maximises specialisation while ensuring termination.*

The abstraction operator examines the set of atoms to be partially deduced and then decides which of the atoms should be abstracted and which ones should be left unmodified. A very simple abstraction operator, which ensures termination, can be obtained by imposing a finite maximum number of atoms in $\mathbf{A}_i$ and using the *most specific generalisation (msg)*[3] to stick to that maximum. This approach is however quite unsatisfactory. First it involves an ad hoc maximum number, which might lead to either too much or too little polyvariance, depending on the context. Secondly, the *msg* is just based on the *syntactic structure* of the atoms to be specialised. This is generally not such a good idea. Indeed, two atoms can be unfolded and specialised in a very similar way in the context

[3] Also known as anti-unification or least general generalisation, see for instance [37].

of one program P_1, while in the context of another program P_2 their specialisation behaviour can be drastically different. The syntactic structure of the two atoms is of course unaffected by the particular context and an operator like the *msg* will perform exactly the same abstraction within P_1 and P_2, although vastly different generalisations might be called for.

A better candidate for an abstraction might be to examine the finite (possibly incomplete) SLDNF-tree 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 similar behaviour in the context of P_1 and their dissimilar behaviour within P_2, and can therefore take appropriate actions in the form of different generalisations. The following example illustrates these points.

Example 2.5 Let P be the append program:

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

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. To avoid clutter, we will drop the substitutions in such figures.

Let $\mathbf{A} = \{B, C\}$ be a (dependent) 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 $\mathbf{A}$ in the way depicted in Figure 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 for $resultants(\tau_B)$ the single fact:

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

while for $append(X, [a], Y)$ we obtain the following recursive set of clauses $resultants(\tau_C)$:

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

So, in this case, it is vital to keep separate specialised versions for B and C and not abstract them by e.g. their *msg*.
However, it is very easy to come up with another context in which 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 Figure 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 and that a refined abstraction operator should also take the associated SLD(NF)-trees into consideration.

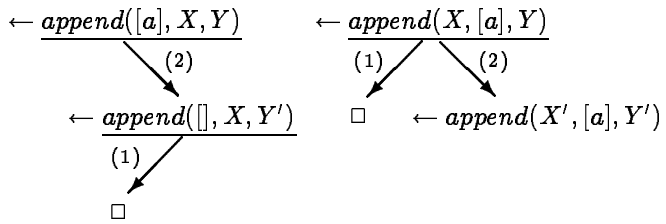


Figure 1: SLD-trees τ_B and τ_C for Example 2.5

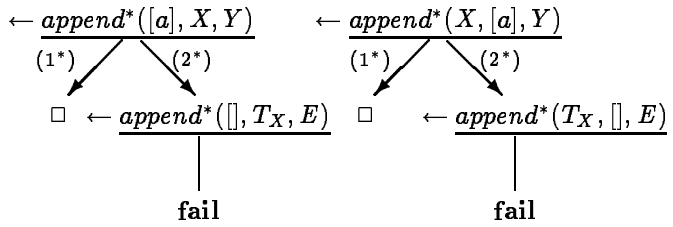


Figure 2: SLD-trees τ_B^* and τ_C^* for Example 2.5

2.3 Characteristic Paths and Trees

Above we have illustrated the interest of examining the (possibly incomplete) SLDNF-trees generated for the atoms to be partially deduced and e.g. only abstract two (or more) atoms if their associated trees are "similar enough". A crucial question is of course which part of these SLDNF-trees should be taken into account to decide upon similarity. If we take everything into account, i.e. only abstract two atoms if their associated trees are identical, this amounts to performing no abstraction at all. So an abstraction operator should focus on the "essential" structure of an SLDNF-tree and for instance disregard the particular substitutions and goals within the tree. The following two definitions, adapted from [18], do just that: they characterise the essential structure of SLDNF-derivations and trees.

Definition 2.6 Let G_0 be a goal and let P be a normal program whose clauses are numbered. Let $G_0, \ldots, G_n$ be the goals of a finite, possibly incomplete SLDNF-derivation D of $P \cup \{G_0\}$. The *characteristic path* of the derivation D is the sequence $\langle l_0 \circ c_0, \ldots, l_{n-1} \circ c_{n-1} \rangle$, where l_i is the position of the selected literal in G_i, and c_i is defined as:

- if the selected literal is an atom, then c_i is the number of the clause chosen to resolve with G_i.

- if the selected literal is $\neg p(\bar{t})$, then c_i is the predicate p.

The set containing the characteristic paths of all possible finite SLDNF-derivations for $P \cup \{G_0\}$ will be denoted by $chpaths(P, G_0)$.

For example, the characteristic path of the derivation associated with the only branch of the SLD-tree τ_B in Figure 1 is $\langle 1 \circ 2, 1 \circ 1 \rangle$.

Recall that an SLDNF-derivation D can be either failed, incomplete, successful or infinite. As we will see below, characteristic paths will only be used to characterise *finite* and *non-failing* derivations of *atomic* goals, corresponding to the atoms to be partially deduced. Still, one might wonder why a characteristic path does not contain information on whether the associated derivation is successful or incomplete. Fortunately, in the context of finite, non-failing derivations of atomic goals, the information about whether the derivation associated with a characteristic path is incomplete or successful is already implicitly present and no further precision would be gained by adding it. This is proven in [41]. Also, once the top-level goal is known, the characteristic path is sufficient to reconstruct all the intermediate goals as well as the final one.

Now that we have characterised derivations, we can characterise goals by characterising the derivations of their associated finite SLDNF-trees.

Definition 2.7 Let G be a goal and P a normal program and τ be a finite SLDNF-tree for $P \cup \{G\}$. Then the *characteristic tree* $\hat{\tau}$ *of* τ is the set containing the characteristic paths of the non-failing SLDNF-derivations associated with the branches of τ. $\hat{\tau}$ is called a characteristic tree iff it is the characteristic tree of some finite SLDNF-tree.

Let U be an unfolding rule such that $U(P, G) = \tau$. Then $\hat{\tau}$ is also called *the characteristic tree* of G (in P) via U. We introduce the notation $chtree(G, P, U) = \hat{\tau}$. We also say that $\hat{\tau}$ is *a characteristic tree of* G *(in* P*)* if it is *the* characteristic tree of G (in P) via some unfolding rule U.

Note that the characteristic path of an empty derivation is the empty path $\langle\rangle$, and the characteristic tree of a trivial SLDNF-tree is $\{\langle\rangle\}$.

Although a characteristic tree only contains a collection of characteristic paths, the actual tree structure can be reconstructed without ambiguity. The "glue" is provided by the clause numbers inside the characteristic paths (branching in the tree is indicated by differing clause numbers).

Example 2.8 The characteristic trees of the finite SLD-trees τ_B and τ_C in Figure 1 are $\{\langle 1 \circ 1, 1 \circ 2 \rangle\}$ 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 Figure 2 are both $\{\langle 1 \circ 1^* \rangle\}$.

The following observation underlines the interest of characteristic trees in the context of partial deduction. Indeed, 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).
- 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.

An aspect that is not explicitly captured by the characteristic tree τ_A is how the atoms in the leaves of the associated SLDNF-tree are further specialised.

By examining only the non-failing branches we do not capture *how* exactly some branches were pruned. The following example illustrates why this is adequate and even beneficial.

Example 2.9 Let P be the following program:

(1) $member(X, [X|T]) \leftarrow$
(2) $member(X, [Y|T]) \leftarrow member(X, T)$

Let $A_1 = member(a, [a, b])$ and $A_2 = member(a, [a])$. Suppose that A_1, A_2 are unfolded as depicted in Figure 3. Then both these atoms have the same characteristic tree $\tau = \{\langle 1 \circ 1 \rangle\}$ although the associated SLDNF-trees differ by the structure of their failing branches. However, this is of no relevance, because the failing branches do not materialise within the resultants (i.e. the specialised code generated for the atoms) and furthermore the single resultant $member(a, [a|T]) \leftarrow$ could be used for both A_1 and A_2 without loosing any specialisation.

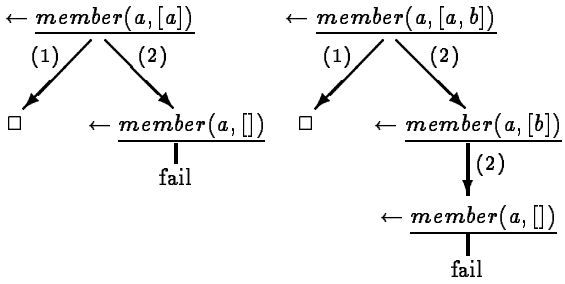


Figure 3: SLD-trees for Example 2.9

In summary, characteristic trees seem to be an almost ideal vehicle for a refined control of polyvariance [21, 18], a fact we will try to exploit in the following.

2.4 An Abstraction Operator using Characteristic Trees

The following definition captures a first attempt at using characteristic trees for the control of polyvariance.

Definition 2.10 Let P be a normal program, U an unfolding rule and S a set of atoms. For every characteristic tree τ, let S_τ be defined as $S_\tau = \{A \mid A \in S \wedge chtree(\leftarrow A, P, U) = \tau\}$. The abstraction operator $chabs_{P,U}$ is then defined as: $chabs_{P,U}(S) = \cup_\tau \{msg(S_\tau)\}$.

Unfortunately, although for a lot of practical cases $chabs_{P,U}$ performs quite well, it does not always preserve the characteristic trees, entailing a sometimes quite severe loss of precision and specialisation.

We first illustrate the possible specialisation losses in the following example.

Example 2.11

Let us specialise the program of Example 2.9 for $G = \leftarrow member(a,[a,b])$, $member(a,[a])$ using Algorithm 2.4. We start with the set $\mathbf{A}_0 = \{member(a,[a,b]), member(a,[a])\}$. Let us suppose that the unfolding rule U unfolds as depicted in Figure 3. For both atoms the associated characteristic tree is $\tau = \{\langle 1 \circ 1 \rangle\}$ and there are no atoms in the leaves. We now apply the abstraction operator: $\mathbf{A}_1 = chabs_{P,U}(A_0) = \{member(a,[a|T])\}$ (the *msg* has been calculated). Unfortunately, the general atom has a different characteristic tree τ', no matter how U unfolds $member(a,[a|T])$. Taking for instance a determinate unfolding rule U (with lookahead), we obtain $\tau' = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2 \rangle\}$.

This loss of precision leads to sub-optimal specialised programs. At the next step of the algorithm the atom $member(a,T)$ will be added to $\mathbf{A}_1$. This atom also has the characteristic tree τ' under U. Hence the final set $\mathbf{A}'$ equals $\{member(a,L)\}$ (containing the *msg* of $member(a,[a|T])$ and $member(a,T)\}$), and we obtain the following specialisation, sub-optimal for $\leftarrow member(a,[a,b]), member(a,[a])$:

(1') $member(a,[a|T]) \leftarrow$

(2') $member(a,[X|T]) \leftarrow member(a,T)$

So, although partial deduction was able to figure out that $member(a,[a,b])$ as well as $member(a,[a])$ have only one non-failing resolvent and are determinate, this information has been lost due to an imprecision of the abstraction operator, leading to a sub-optimal residual program in which the determinacy is not explicit (and redundant computation steps occur at run-time). Note that a "perfect" program for $\leftarrow member(a,[a,b]), member(a,[a])$ would just consist of clause (1').

As shown in [41], these losses of precision can also lead to non-termination of the partial deduction process, further illustrating the importance of preserving characteristic trees upon generalisation. Also note that the abstraction operators presented in [21] and [18] basically behave like the operator $chabs_{P,U}$ and encounter exactly the same problems (see [41] for more details). If however we manage to keep precision, we will obtain an abstraction operator for partial deduction with optimal local precision (in the sense that all the local specialisation achieved by the unfolding rule is preserved by the abstraction) and which guarantees termination. This quest is pursued in Section 3.

3 Partial Deduction with Characteristic Atoms

In the previous section we have dwelled upon the appeal of characteristic trees for controlling polyvariance, but we have also touched upon the difficulty of preserving characteristic trees in the abstraction process as well as the ensuing problems concerning termination and precision. In this section, we present an elegant solution to this entanglement. Its basic idea is to simply *impose* characteristic trees on the generalised atoms. This amounts to associating characteristic trees with the atoms to be specialised, allowing the preservation of characteristic trees in a straightforward way and circumventing the need for intricate generalisations.

3.1 Characteristic Atoms

We first introduce the crucial notion of a *characteristic atom*, which associates a characteristic tree with an atom.

Definition 3.1 A *characteristic atom* is a couple (A, τ) consisting of an atom A and a characteristic tree τ.

Note that τ is not required to be a characteristic tree of $\leftarrow A$ in the context of the particular program P under consideration.

Example 3.2 Let $\tau = \{\langle 1 \circ 1 \rangle\}$ be a characteristic tree. Then $CA_1 = (member(a, [a, b]), \tau)$ and $CA_2 = (member(a, [a]), \tau)$ are characteristic atoms. $CA_3 = (member(a, [a|T]), \tau)$ is also a characteristic atom, but e.g. in the context of the *member* program P from Example 2.9, τ is *not* a characteristic tree of its atom component $member(a, [a|T])$ (cf. Example 2.11). Intuitively such a situation corresponds to *imposing* the characteristic tree τ on the atom $member(a, [a|T])$. Indeed, as we will see later, CA_3 can be seen as a *precise* generalisation (in P) of the atoms $member(a, [a, b])$ and $member(a, [a])$, solving the problem of Example 2.11.

A characteristic atom will be used to represent a possibly infinite set of atoms, called its concretisations. This is nothing really new: in standard partial deduction, an atom A also represents a possibly infinite set of concrete atoms, namely its instances. The characteristic tree component of a characteristic atom will just act as a constraint on the instances, i.e. keeping only those instances which have a particular characteristic tree. This is captured by the following definition.

Definition 3.3 Let (A, τ_A) be a characteristic atom and P a program. An atom B is a *precise concretisation* of (A, τ_A) (in P) iff B is an instance of A and, for some unfolding rule U, $chtree(\leftarrow B, P, U) = \tau_A$. An atom is a *concretisation* of (A, τ_A) (in P) iff it is an instance of a precise concretisation of (A, τ_A) (in P).
We denote the set of concretisations of (A, τ_A) in P by $\gamma_P(A, \tau_A)$.

A characteristic atom thus represents a (possibly infinite) set of atoms, namely its concretisations according to the above definition. A characteristic atom with a non-empty set of concretisations in P will be called *well-formed* in P or a *P-characteristic atom*. We will from now on usually restrict our attention to P-characteristic atoms. In particular, the partial deduction algorithm presented later on in Section 3.4 will produce P-characteristic atoms (and the associated partial deduction), where P is the original program to be specialised.

Example 3.4 Take the characteristic atom $CA_3 = (member(a, [a|T]), \tau)$ with $\tau = \{\langle 1 \circ 1 \rangle\}$ from Example 3.2 and take the *member* program P from Example 2.9. The atoms $member(a, [a])$ and $member(a, [a, b])$ are precise concretisations of CA_3 in P (under the unfolding rule of Figure 3). Also, neither $member(a, [a|T])$ nor $member(a, [a, a])$ are concretisations of CA_3 in P.
Finally note that CA_3 is a P-characteristic atom while $(member(a, [a|T]), \{\langle 2 \circ 5 \rangle\}$ or even $(member(a, [a|T]), \{\langle 1 \circ 2 \rangle\}$ are not.

Example 3.5 Let P be the following simple program:

(1) $p(a, Y) \leftarrow$
(2) $p(X, Y) \leftarrow \neg q(Y), q(X)$
(3) $q(b) \leftarrow$

Let $\tau = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2, 1 \circ q, 1 \circ 3 \rangle\}$ and $CA = (p(X,Y), \tau)$. Then $p(X,a)$ and $p(X,c)$ are precise concretisations of CA in P and neither $p(b,b)$, $p(X,b)$, $p(a,Y)$ nor $p(X,Y)$ are concretisations of CA in P. Also $p(b,a)$ and $p(a,a)$ are both concretisations of CA in P, but they are not precise concretisations. Note that the selection of the negative literal $\neg q(Y)$, corresponding to $1 \circ q$ in τ, is unsafe for $\leftarrow p(X,Y)$, but that for any concretisation of CA the corresponding derivation step is safe and succeeds (i.e. the negative literal is ground and succeeds).

Note that by definition, the set of concretisations associated with a characteristic atom is *downwards closed* (or *closed under substitution*). This observation justifies the following definition.

Definition 3.6 We will call a P-characteristic atom (A, τ_A) which has A as one of its concretisations (in P) *unconstrained* (in P).

The concretisations of an unconstrained characteristic atom (A, τ_A) are identical to the instances of the ordinary atom A (because the concretisations are downwards closed and $\gamma_P(A, \tau)$ contains no atom strictly more general than A). In still other words, characteristic atoms along with Definition 3.3 provide a proper generalisation of the way atoms are used in the standard partial deduction approach.

3.2 Generating Resultants

We will now address the generation of resultants for characteristic atoms. The simplest approach is just to unfold the atom A of a characteristic atom (A, τ) precisely as indicated by τ. As illustrated in Example 3.5 above, the only complication is that this might lead to the selection of non-ground negative literals. This will not turn out to be a problem however, because, as we will prove later, the corresponding derivations for any concretisation of (A, τ) will be safe and the selected negative literals will succeed.

We need the following definition in order to formalise the resultants associated with a characteristic atom.

Definition 3.7 A *generalised SLDNF-derivation* is either composed of SLDNF-derivation steps or of derivation steps in which a non-ground negative literal is selected and removed. Generalised SLDNF-derivations containing steps of the latter kind will be called *unsafe*.

Most of the definitions for ordinary SLDNF-derivations, like the associated characteristic path and resultant, carry over to generalised derivations.

We first define a set of possibly unsafe generalised SLDNF-derivations associated with a characteristic atom:

Definition 3.8 Let P be a program and (A, τ) a P-characteristic atom. If $\tau \neq \{\langle\rangle\}$ then $\delta_P(A, \tau)$ is the set of all generalised SLDNF-derivations for $P \cup \{\leftarrow A\}$ such that their characteristic paths are in τ. If $\tau = \{\langle\rangle\}$ then $\delta_P(A, \tau)$ is the set of all non-failing SLD-derivations for $P \cup \{\leftarrow A\}$ of length 1.[4]

[4] Just like in standard partial deduction, we want to construct only non-trivial SLDNF-trees for $P \cup \{\leftarrow A\}$ to avoid the problematic resultant $A \leftarrow A$.

Note that the derivations in $\delta_P(A,\tau)$ are necessarily finite and non-failing. See also Lemma 3.22.

We will call a P-characteristic atom (A,τ) *safe* (in P) iff all derivations in $\delta_P(A,\tau)$ are safe. Note that an unconstrained characteristic atom in P is also inevitably safe in P (because A must be a precise concretisation of (A,τ) we have that τ is a characteristic tree of $\leftarrow A$ and thus all derivations in $\delta_P(A,\tau)$ are ordinary SLDNF-derivations and therefore safe).

Based on the definition of $\delta_P(A,\tau)$, we can now define the resultants, and hence the partial deduction, associated with characteristic atoms:

Definition 3.9 Let P be a program and (A,τ) a P-characteristic atom. Let $\{\delta_1,\ldots,\delta_n\}$ be the generalised SLDNF-derivations in $\delta_P(A,\tau)$ and let $\leftarrow G_1,\ldots,\leftarrow G_n$ be the goals in the leaves of these derivations. Let $\theta_1,\ldots,\theta_n$ be the computed answers of the derivations from $\leftarrow A$ to $\leftarrow G_1,\ldots,\leftarrow G_n$ respectively. Then the set of resultants $\{A\theta_1 \leftarrow G_1,\ \ldots,\ A\theta_n \leftarrow G_n\}$ is called the *partial deduction of* (A,τ) *in* P.
Every atom occurring in some of the G_i will be called a *body atom (in P)* of (A,τ). We will denote the set of such body atoms by $BA_P(A,\tau)$.

Example 3.10 The partial deduction of $(member(a,[a|T]),\{\langle 1\circ 1\rangle\})$ in the program P of Example 2.9 is $\{member(a,[a|T]) \leftarrow\}$. Note that it is different from any set of resultants that can be obtained for the ordinary atom $member(a,[a|T])$. However, as we will prove below, the partial deduction is correct for any concretisation (as defined in Definition 3.3) of $(member(a,[a|T]),\{\langle 1\circ 1\rangle\})$.

Example 3.11 The partial deduction P' of $(p(X,Y),\tau)$ with $\tau = \{\langle 1\circ 1\rangle, \langle 1\circ 2, 1\circ q, 1\circ 3\rangle\}$ of Example 3.5 is

(1') $p(a,Y) \leftarrow$
(2') $p(b,Y) \leftarrow$

Note that using P' instead of P is correct for e.g. the concretisations $p(b,a)$ or $p(X,a)$ of $(p(X,Y),\tau)$ but not for e.g. $p(b,b)$ or $p(X,b)$ which are not concretisations of $(p(X,Y),\tau)$.

We now generate partial deductions not for sets of atoms, but for sets of *characteristic* atoms. As such, a same atom A might occur in several characteristic atoms but with different associated characteristic trees. This means that renaming as a way to ensure independence imposes itself even more than in the standard partial deduction setting.

In addition to renaming, we will also incorporate argument filtering, leading to the following definition.

Definition 3.12 An *atomic renaming* α for a set $\mathcal{A}$ of characteristic atoms is a mapping from $\mathcal{A}$ to atoms such that

- for each $(A,\tau)\in\mathcal{A}$: $vars(\alpha((A,\tau))) = vars(A)$
- for $CA, CA' \in \mathcal{A}$ such that $CA \neq CA'$ the predicate symbols of $\alpha(CA)$ and $\alpha(CA')$ are distinct (but not necessarily fresh).

Let P be a program. A *renaming function* ρ_α for $\mathcal{A}$ in P based on α is a mapping from atoms to atoms such that:

- $\rho_\alpha(A) = \alpha((A',\tau'))\theta$ for some $(A',\tau')\in\mathcal{A}$ with $A = A'\theta \wedge A \in \gamma_P(A',\tau')$.

We leave $\rho_\alpha(A)$ undefined if A is not a concretisation in P of an element in $\mathcal{A}$. A renaming function ρ_α can also be applied to first-order formulas, by applying it individually to each atom of the formulas.

Note that if the set of concretisations of two or more elements in $\mathcal{A}$ overlap then ρ_α must make a choice for the atoms in the intersection of the concretisations and several renaming functions based on the same α exist.

Definition 3.13 Let P be a program, $\mathcal{A} = \{(A_1, \tau_1), \ldots, (A_n, \tau_n)\}$ be a finite set of P-characteristic atoms and let ρ_α be a renaming for $\mathcal{A}$ in P based on the atomic renaming α. For each $i \in \{1, \ldots, n\}$, let R_i be the partial deduction of (A_i, τ_i) in P. Then the program $\{\alpha((A_i, \tau_i))\theta \leftarrow \rho_\alpha(Bdy) \mid A_i\theta \leftarrow Bdy \in R_i \wedge 1 \leq i \leq n \wedge \rho_\alpha(Bdy) \text{ is defined } \}$ is called the *partial deduction of P wrt $\mathcal{A}$ and ρ_α*.

Example 3.14 Let P be the following program

(1) $member(X, [X|T]) \leftarrow$
(2) $member(X, [Y|T]) \leftarrow member(X, T)$
(3) $t \leftarrow member(a, [a]), member(a, [a, b])$

Let $\tau = \{\langle 1 \circ 1 \rangle\}$ and $\tau' = \{\langle 1 \circ 3 \rangle\}$ and let $\mathcal{A} = \{(member(a, [a|T]), \tau), (t, \tau')\}$. Also let $\alpha((member(a, [a|T]), \tau)) = m_1(T)$ and $\alpha((t, \tau')) = t$. Because the concretisations in P of the elements in $\mathcal{A}$ are disjoint, there exists only one renaming function ρ_α based on α. Notably $\rho_\alpha(\leftarrow member(a, [a]), member(a, [a, b])) = \leftarrow m_1([]), m_1([b])$ because both atoms are concretisations of $(member(a, [a|T]), \tau)$. Therefore the partial deduction of P wrt $\mathcal{A}$ and ρ_α is:[5]

(1') $m_1(X) \leftarrow$
(2') $t \leftarrow m_1([]), m_1([b])$

Note that in Definition 3.13 the original program P is completely "thrown away". This is actually what a lot of practical partial evaluators for functional or logic programming languages do, but is unlike e.g. the definitions in [49]. However, there is no fundamental difference between these two approaches: keeping part of the original program can always be "simulated" very easily in our approach by using unconstrained characteristic atoms of the form $(A, \langle\rangle)$ combined with a renaming α such that $\alpha((A, \langle\rangle)) = A$.

3.3 Correctness Results

Let us first rephrase the coveredness condition in the context of characteristic atoms. This definition will ensure that the renamings, applied for instance in Definition 3.13, are always defined.

Definition 3.15 Let P be a program and $\mathcal{A}$ a set of characteristic atoms. Then $\mathcal{A}$ is called *P-covered* iff for every characteristic atom in $\mathcal{A}$, each of its body atoms (in P) is a concretisation in P of a characteristic atom in $\mathcal{A}$.
Also, a goal G is *covered by $\mathcal{A}$ in P* iff every atom A occurring in G is a concretisation of a characteristic atom in $\mathcal{A}$.

[5] The FAR filtering algorithm of [47] can be used to further improve the specialised program by removing the redundant argument of m_1.

The main correctness result for partial deduction with characteristic atoms is as follows:

Theorem 3.16 Let P be a normal program, G a goal, $\mathcal{A}$ any finite set of P-characteristic atoms and P' the partial deduction of P wrt $\mathcal{A}$ and some ρ_α. If $\mathcal{A}$ is P-covered and if G is covered by $\mathcal{A}$ in P then the following hold:

1. $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ iff $P \cup \{G\}$ does.
2. $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree iff $P \cup \{G\}$ does.

In the remainder of Subsection 3.3 we will prove this theorem in three successive stages.

1. First, we will restrict ourselves to unconstrained characteristic atoms. This will allow us to reuse the correctness results for standard partial deduction with renaming in a rather straightforward manner.
2. We will then move on to safe characteristic atoms. Such partial deductions can basically be obtained from partial deductions for unconstrained characteristic atoms by removing certain clauses. We will show that these clauses can be safely removed without affecting the computed answers nor the finite failure.
3. In the final step we will allow any characteristic atom. The associated partial deductions can be obtained from partial deductions for safe characteristic atoms, basically by removing negative literals from the clauses. We will establish correctness by showing that, for all concrete executions, these negative literals will be ground and succeed.

The reader not interested in the details of the correctness proof can immediately continue with Subsection 3.4.

3.3.1 Correctness for Unconstrained Characteristic Atoms

Note that if $\mathcal{A}$ is a set of unconstrained characteristic atoms we simply have a standard partial deduction with renaming. We will use this observation as a starting point for proving correctness of partial deduction for characteristic atoms.

The following is an adaptation of the correctness of standard partial deduction with renaming and filtering:

Theorem 3.17 Let P' be a partial deduction of P wrt $\mathcal{A}$ and ρ_α such that $\mathcal{A}$ is a finite set of unconstrained P-characteristic atoms and such that $\mathcal{A}$ is P-covered and G is covered by $\mathcal{A}$ in P. Then:

1. $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ iff $P \cup \{G\}$ does.
2. $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree iff $P \cup \{G\}$ does.

Proof First note that the coveredness conditions on $\mathcal{A}$ and G ensure that the renamings performed to obtain P' (according to Definition 3.13), as well as the renaming $\rho_\alpha(G)$, are defined. The result then follows in a rather straightforward manner from the Theorems 3.5 and 4.11 in [2]. In [2] the filtering has been split into 2 phases: one which does just the renaming to ensure independence (called partial deduction with dynamic renaming; correctness of this phase is proven in Theorem 3.5 of [2]) and one which does the filtering (called post-processing renaming; the correctness of this phase is proven in Theorem 4.11 of [2]).

To apply these results we simply have to notice that:

- P' corresponds to partial deduction with dynamic renaming and post-processing renaming for the multiset of atoms $\mathbf{A} = \{A \mid (A, \tau) \in \mathcal{A}\}$ (indeed the same atom could in principle occur in several characteristic atoms, this is not a problem however as the results in [2] carry over to multisets of atoms — alternatively one could add an extra unused argument to P', $\rho_\alpha(G)$ and $\mathbf{A}$ and then place different variables in that new position to transform the multiset $\mathbf{A}$ into an ordinary set).

- $P' \cup \{\rho_\alpha(G)\}$ is $\mathbf{A}$-covered because $\mathcal{A}$ is P-covered and G is covered by $\mathcal{A}$ in P (and because the original program P is unreachable in the predicate dependency graph from within P' or within $\rho_\alpha(G)$).

Three minor technical issues have to be addressed in order to reuse the theorems from [2]:

- Theorem 3.5 of [2] requires that no renaming be performed on G, i.e. $\rho_\alpha(G)$ must be equal to G. However, without loss of generality, we can assume that the top-level query is the unrenamed atom $new(X_1, \ldots, X_k)$, where new is a fresh predicate symbol and $vars(G) = \{X_1, \ldots, X_k\}$. We just have to add the clause $new(X_1, \ldots, X_k) \leftarrow Q$, where $G = \leftarrow Q$, to the initial program. Trivially the query $\leftarrow new(X_1, \ldots, X_k)$ and G are equivalent wrt c.a.s. and finite failure (see also Lemma 2.2 in [20]).

- Theorem 4.11 of [2] requires that G contains no variables or predicates in $\mathbf{A}$. The requirement about the variables is not necessary in our case because we do not base our renaming on the *mgu*. The requirement about the predicates is another way of ensuring that $\rho_\alpha(G)$ must be equal to G, which can be circumvented in a similar way as for the first point above.

- Theorems 3.5 and 4.11 of [2] require that the predicates of the renaming do not occur in the original P. Our Definition 3.12 does not require this. This is of no importance as the original program is always "completely thrown away" in our approach. We can still apply these theorems by using an intermediate renaming ρ' which satisfies the requirements of Theorems 3.5 and 4.11 of [2] and then applying an additional one step post-processing renaming ρ'', with $\rho_\alpha = \rho'\rho''$, along with an extra application of Theorem 4.11 of [2].

□

3.3.2 Correctness for Safe Characteristic Atoms

To prove the correctness for safe characteristic atoms we need the following adaptation of Lemma 4.12 from [49] (we just formalise the use in [49] of "corresponding SLDNF-derivation" in terms of characteristic paths).

Lemma 3.18 Let R be the resultant of a finite (possibly incomplete) SLDNF-derivation for $P \cup \{\leftarrow A\}$ whose characteristic path is δ. If $\leftarrow A\theta$ resolves with R giving the resultant R_A then there exists a finite (possibly incomplete) SLDNF-derivation for $P \cup \{\leftarrow A\theta\}$ whose characteristic path is δ and whose resultant is R_A.

The following is an extension of the above lemma. It establishes a more precise link between derivations in the renamed (standard) partial deduction and derivations in the original program. For that, the following concept will prove to be useful. We call A' an *admissible renaming* of A under α for $\mathcal{A}$ in P iff there exists some renaming function ρ'_α based on α such that $A' = \rho'_\alpha(A)$. This concept can be extended to any formula F in a straightforward way.

Lemma 3.19 Let P' be a partial deduction of P wrt $\mathcal{A}$ and ρ_α such that $\mathcal{A}$ is a finite set of safe P-characteristic atoms and such that $\mathcal{A}$ is P-covered and G is covered by $\mathcal{A}$ in P.

Let G' be an admissible renaming of G under α for $\mathcal{A}$ in P. Let D' be a finite SLDNF-derivation for $P' \cup \{G'\}$ leading to the resolvent RG' via the c.a.s. θ. Then there exists a finite SLDNF-derivation for $P \cup \{G\}$ leading to the resolvent RG via c.a.s. θ such that RG' is an admissible renaming of RG under α for $\mathcal{A}$ in P and such that RG is covered by $\mathcal{A}$ in P.

Proof First note that, if RG' is an admissible renaming of RG, RG must by definition be covered by $\mathcal{A}$ in P.

Also note that not all resolvents of $P \cup \{G\}$ are covered — there can be some intermediate goals which have no correspondent goal in P' (because entire derivation sequences can be compressed into one resultant of P'). So it is only at some specific points that a resolvent for $P \cup \{G\}$ has a counterpart in $P' \cup \{G'\}$. We will however show in the following that every derivation for $P' \cup \{G'\}$ has a counterpart in $P \cup \{G\}$.

We first do the proof for SLDNF-derivations in which no negative literals are selected (i.e. SLD$^+$-derivations). Let P'' be the unrenamed version of P' (i.e. the union of the partial deductions of the elements of $\mathcal{A}$). We can repeatedly apply Lemma 3.18 (because $\mathcal{A}$ contains only safe P-characteristic atoms) on the unrenamed version P'' of P' to deduce that if a derivation for $P'' \cup \{G\}$ leading to the resolvent RG via c.a.s. θ exists then there also exists a derivation for $P \cup \{G\}$ with the same resolvent and c.a.s. (the resolvent is just the body of the resultant and if the head of two resultants for the same goal are identical then so are the c.a.s.).

Now, by construction of P' from P'' (Definition 3.13), every derivation for $P' \cup \{G'\}$ has an equivalent in $P'' \cup \{G\}$ (but not vice versa, because independence in P'' is not necessarily ensured) and the c.a.s. are not affected by correctness of the renaming (see e.g. Lemma 2.2 in [20]). We can also easily prove inductively that, if we start with a goal which is an admissible renaming of some goal under α, we will after each derivation step with a clause in P' still obtain a goal which is an admissible renaming under α for $\mathcal{A}$ in P because:

- all body atoms of P' are covered by $\mathcal{A}$ in P (because $\mathcal{A}$ is P-covered) and are therefore, by Definition 3.12 of a renaming, admissible renamings under α for $\mathcal{A}$ in P.

- the resolvent will contain instances of these body atoms which are still covered because the set of concretisations of characteristic atoms is downwards closed and which are therefore still admissible renamings under α for $\mathcal{A}$ in P.

So the lemma holds for SLDNF-derivations in which no negative literals are selected.

Let us now allow the selection of a negative literal in the derivation D' (for $P' \cup \{G'\}$ leading to the resolvent RG'). In that case we can apply the just established result for the derivation leading up to the selected negative literal. Then, by Theorem 3.17, we can deduce that the negative literal must also succeed in the original program. Because neither the c.a.s. nor the resultant are affected the result still holds after the selection of the negative literal. We can therefore apply the above result for derivations without negative literals again until the end of the derivation D'. Finally, we can repeat this same reasoning inductively for any number of selected negative literals. □

Usually one will call the specialised program with one specific renaming and not just with an admissible one. So one might wonder why we only prove in Lemma 3.19 above that RG' is an admissible renaming of RG and not that $RG' = \rho_\alpha(RG)$. The reason is that in the course of performing resolution steps atoms might become more instantiated and applying the renaming function ρ_α on the more instantiated atom might result in a different renaming. Take for example the set $\mathcal{A} = \{(p(X), \tau), (p(a), \tau')\}$ of unconstrained characteristic atoms, the goal $G = \leftarrow p(X), p(X)$ and take α such that $\alpha((p(X), \tau)) = p'(X)$, $\alpha((p(a), \tau')) = p_a$. Then $\rho_\alpha(G) = \leftarrow p'(X), p'(X)$. Also assume that $\rho_\alpha(p(a)) = p_a$. Now suppose that the clause $p'(a) \leftarrow$ is in the partial deduction P' wrt an original P and the set $\mathcal{A}$. Then after one resolution step for $P' \cup \rho_\alpha(G)$ we obtain the goal $\leftarrow p'(a)$. Similarly, after one resolution step for $P \cup \{G\}$ we obtain the goal $\leftarrow p(a)$. Now $\rho_\alpha(\leftarrow p(a)) = \leftarrow p_a \neq \leftarrow p'(a)$. However, $\leftarrow p'(a)$ is still an admissible renaming of $\leftarrow p(a)$ and Lemma 3.19 holds.

The following lemmas will prove useful later on.

Lemma 3.20 Let τ be a characteristic tree. Let $\delta_1 \in \tau$ and $\delta_2 \in \tau$. If δ_2 is a prefix of δ_1 then $\delta_1 = \delta_2$.

Proof The property follows immediately from the definitions of SLDNF-trees and characteristic trees. □

Lemma 3.21 Let A and B be atoms and τ_A a characteristic tree of $\leftarrow A$ in the program P. If B is an instance of A then there exists a characteristic tree τ_B of $\leftarrow B$ in P such that $\tau_B \subseteq \tau_A$.

Proof By Definition 2.7, for some unfolding rule U we have that $\tau_A = chtree(\leftarrow A, P, U)$. Because B is more instantiated than A, all resolution steps for $\leftarrow A$ are either also possible for $\leftarrow B$ or they fail. Therefore, for some unfolding rule U', we have that $\tau_B = chtree(\leftarrow B, P, U') \subseteq \tau_A$. □

Lemma 3.22 Let τ be a characteristic tree, P a program and (A,τ) a P-characteristic atom. If (A,τ) is safe then there exists an unfolding rule such that $\tau \subseteq chtree(\leftarrow A, P, U)$.

Proof As (A,τ) is a P-characteristic atom, we must have by definition at least one precise concretisation A' whose characteristic path is τ in P. As all the derivations in $\delta_P(A,\tau)$ are safe, we can unfold $\leftarrow A$ in a similar way as $\leftarrow A'$. This will result in a characteristic tree τ' which contains all the paths in τ as well as possibly some additional paths (due to additionally matching clauses). □

The above Lemma 3.22 (also) establishes that the partial deduction $P_{(A,\tau)}$ of a safe characteristic atom (A,τ) is a subset of a partial deduction P_A of the ordinary atom A. The following lemma shows that it is correct to remove the resultants $P_A \setminus P_{(A,\tau)}$ for the concretisations of (A,τ). In the proof of this lemma we need to combine characteristic paths. A characteristic path being a sequence, we can simply concatenate two characteristic paths δ_1, δ_2. For this we will use the standard notation $\delta_1\delta_2$ from formal language theory [1, 29].

Lemma 3.23 Let P be a program, (A,τ_A) and (A,τ_A^*) safe P-characteristic atoms with $\tau_A \subseteq \tau_A^*$. Let $A' \in \gamma_P(A,\tau_A)$ and $C \in resultants(\tau_A^*) \setminus resultants(\tau_A)$.

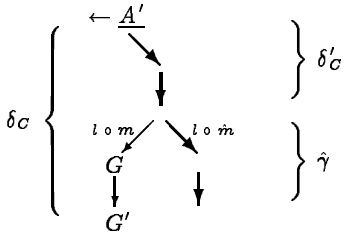
If $\leftarrow A'$ resolves with C to G' then G' fails finitely in P.

Proof Let δ_C be the characteristic path associated (Definition 3.8) with C (i.e. C is the resultant of the generalised SLDNF-derivation for $P \cup \{\leftarrow A\}$ whose characteristic path is δ_C). Because $C \in resultants(\tau_A^*) \setminus resultants(\tau_A)$ we know that $\delta_C \in \tau_A^*$ and $\delta_C \notin \tau_A$. Also, because (A,τ_A^*) is safe, C is the resultant of an SLDNF-derivation (and not of an unsafe generalised one). We can therefore apply Lemma 3.18 to deduce that there exists a finite SLDNF-derivation for $P \cup \{\leftarrow A'\}$ whose characteristic path is δ_C and whose resolvent is G' (if the resultant is identical then so is its body, namely the resolvent).

Furthermore $A' \in \gamma_P(A,\tau_A)$ implies by Lemma 3.21 that, for some unfolding rule U, $\hat{\tau} = chtree(\leftarrow A', P, U) \subseteq \tau_A$.

If $\hat{\tau} = \emptyset$ then $\leftarrow A'$ fails finitely. Therefore, because a finite SLDNF-derivation from $\leftarrow A'$ to G' exists, we can deduce by persistence of failure (Lemma 4.10 in [49]), that G' must also fail finitely.

If $\hat{\tau} \neq \emptyset$ then the largest prefix δ'_C of δ_C, such that $\exists\hat{\gamma}.\delta'_C\hat{\gamma} \in \hat{\tau}$, must exist (the smallest one being $\langle\rangle$). By Lemma 3.20 we know that no proper prefix of δ_C can be in τ_A^*, and therefore neither in τ_A nor $\hat{\tau}$ (because $\hat{\tau} \subseteq \tau_A \subseteq \tau_A^*$). This means that $\hat{\gamma}$ must be non-empty. We also know, again by Lemma 3.20, that δ'_C is a proper prefix of δ_C, i.e. $\delta_C = \delta'_C(l \circ m)\delta''_C$. As there is branching immediately after δ'_C (because $\delta'_C\hat{\gamma} \in \hat{\tau}$ and $\delta'_C(l \circ m)\delta''_C = \delta_C \in \tau_A^* \setminus \tau_A$ and no extension of δ'_C has this property) we even know that the selected literal at position l is a positive literal (the selection of a negative literal can never lead to branching), that m is therefore a clause number and also that $\hat{\gamma} = (l \circ \hat{m})\hat{\gamma}'$ with $\hat{m} \neq m$.



Let G be the goal obtained from the finite SLDNF-derivation for $P \cup \{\leftarrow A'\}$ whose characteristic path is $\delta'_C(l \circ m)$ (this must exist because an SLDNF-derivation for $P \cup \{\leftarrow A'\}$ with characteristic path δ_C exists). In order to arrive at the characteristic tree $\hat{\tau}$ for $\leftarrow A'$ the unfolding rule U has to also reach the goal G, because $\hat{\tau}$ contains the characteristic path $\delta'_C(l \circ \hat{m})\hat{\gamma}'$ and G is "reached" via $\delta'_C(l \circ m)$, a step which cannot be avoided by U if it wants to get as far as $\delta'_C(l \circ \hat{m})$. Furthermore, as neither $\delta'_C(l \circ m)$ nor any extension of it are in $\hat{\tau}$ (by definition of δ'_C) this means that $\leftarrow G$ finitely fails.

Finally, as δ_C is an extension of $\delta'_C(l \circ m)$, we know that a finite (possibly empty) SLDNF-derivation from G to G' exists and therefore, by persistence of failure (Lemma 4.10 in [49]), G' must also finitely fail.

□

We now present a correctness theorem for safe characteristic atoms.

Theorem 3.24 Let P be a normal program, G a goal, $\mathcal{A}$ a finite set of safe P-characteristic atoms and P' the partial deduction of P wrt $\mathcal{A}$ and some ρ_α. If $\mathcal{A}$ is P-covered and if G is covered by $\mathcal{A}$ in P then the following hold:

1. $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ iff $P \cup \{G\}$ does.
2. $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree iff $P \cup \{G\}$ does.

Proof The basic idea of the proof is to transform the characteristic atoms in $\mathcal{A}$ so as to make them unconstrained and thus being able to reuse Theorem 3.17 (the latter is possible by Lemma 3.22). The so obtained partial deduction P'' is therefore totally correct. P'' is also a superset of P', i.e. $P'' = P' \cup P_{New}$, and we will show, mainly using Lemma 3.23, that the clauses P_{New} can be safely removed without affecting the total correctness. The details of the proof are elaborated in the following.

In order to make a characteristic atom (A, τ) in $\mathcal{A}$ unconstrained we have to add some characteristic paths to τ. We have that for every $(A, \tau) \in \mathcal{A}$ the set of derivations $\delta_P(A, \tau)$ is safe. By Lemma 3.22 we know that for some unfolding rule U: $\tau \subseteq chtree(\leftarrow A, P, U)$. Let $\tau' = chtree(\leftarrow A, P, U) \setminus \tau$ and let R_{New} be the partial deduction of (A, τ') (i.e. the unrenamed clauses that have to be added to the partial deduction of (A, τ) in order to arrive at a standard partial deduction of the unconstrained characteristic atom $(A, chtree(\leftarrow A, P, U))$). We denote by $New_{(A,\tau)}$ the following set of clauses $\{\alpha((A, \tau))\theta \leftarrow \rho_\alpha(Bdy) \mid A\theta \leftarrow Bdy \in R_{New}\}$. By adding for each $(A, \tau) \in \mathcal{A}$ the clauses $New_{(A,\tau)}$ to P' we obtain a partial deduction of P wrt an unconstrained set $\mathcal{A}'$ and ρ_α.

Unfortunately, although G remains covered by $\mathcal{A}'$ in P, $\mathcal{A}'$ is not necessarily P-covered any longer. The reason is that new uncovered body atoms can arise inside $New_{(A,\tau)}$. Let UC be these uncovered atoms. To arrive at a covered partial deduction we simply have to add, for every predicate symbol p of arity n occurring in UC, the characteristic atom $(p(X_1, \ldots, X_n), \langle\rangle)$ to $\mathcal{A}'$, where $X_1, \ldots, X_n$ are distinct variables. This will give us the new set $\mathcal{A}'' \supseteq \mathcal{A}'$. Let P'' be the partial deduction of P wrt $\mathcal{A}''$ and ρ_α. Now $\mathcal{A}''$ is trivially P-covered and we can apply the correctness Theorem 3.17 to deduce that the computations for $P'' \cup \{\rho_\alpha(G)\}$ are totally correct wrt the computations in $P \cup \{G\}$.

Note that, by construction, we have that $P' \subseteq P''$. We will now show that by removing the clauses $P_{New} = P'' \setminus P'$ we do not lose any computed answer nor do we remove any infinite failure. In other words any complete SLDNF-derivation for $P'' \cup \{\rho_\alpha(G)\}$ which

uses a clause in $P'' \setminus P'$ fails finitely. This is sufficient to establish that P' is also totally correct.

Let us first prove it for SLDNF-derivations of rank 0, i.e. derivations without selected negative literals. Let D be an SLDNF-derivation for $P'' \cup \{\rho_\alpha(G)\}$ which uses at least one clause in $P'' \setminus P'$. Let D' be the largest prefix SLDNF-derivation of D such that D' uses only clauses within P'. Let RG' be the last goal of D'. We can apply Lemma 3.19 to deduce that there exists an SLDNF-derivation for $P \cup \{G\}$ leading to RG such that RG' is an admissible renaming of RG under α for $\mathcal{A}$ in P and such that RG is covered by $\mathcal{A}$ in P. Let $RG' = \rho'_\alpha(RG)$ and let $\rho'_\alpha(p(\bar{t}))$ be the literal selected in RG' by D (i.e. the next step after performing D'). Because RG' is an admissible renaming of RG, we know that $p(\bar{t}) \in \gamma_P(A,\tau)$ where $\rho'_\alpha(p(\bar{t})) = \alpha((A,\tau))\sigma'$ for some σ'. We can therefore apply Lemma 3.23 to deduce that $\leftarrow p(\bar{t})$, and therefore also RG, fails finitely in P. We can now apply the correctness Theorem 3.17 for the goal $\leftarrow p(\bar{t})$ (it is possible to apply this theorem because $\leftarrow p(\bar{t})$ is covered by $\mathcal{A}$ in P) to deduce that $\leftarrow \rho_\alpha(p(\bar{t}))$, and therefore the derivation D, also fails finitely in P''.

Let us proceed by induction and suppose that the theorem holds for SLDNF-derivations up to rank k. Let us now prove it for rank $k+1$. Let D be an SLDNF-derivation for $P'' \cup \{\rho_\alpha(G)\}$ of rank $k+1$. If D does not directly use any clause in $P'' \setminus P'$, correctness of removing the clauses $P'' \setminus P'$ follows from the induction hypothesis. If however D directly uses a clause in $P'' \setminus P'$ then we can show by a similar reasoning as for rank 0 (just using the induction hypothesis to ensure that nothing has been changed for the negative sub-trees) that D fails finitely.

□

3.3.3 Correctness for Unrestricted Characteristic Atoms

We are finally in the position to prove the general correctness Theorem 3.16 for partial deductions of safe and unsafe P-characteristic atoms.

Proof of Theorem 3.16 For every $(A,\tau) \in \mathcal{A}$ let us remove the unsafely selected negative literals by $\delta_P(A,\tau)$ from τ resulting in τ'. We thereby obtain safe characteristic atoms (A,τ') as well as a set $\mathcal{A}'$ of safe characteristic atoms. Just like in the proof for Theorem 3.24, we might have to add uncovered atoms to $\mathcal{A}'$, giving the new set $\mathcal{A}'' \supseteq \mathcal{A}'$.

Let P'' be the partial deduction of P wrt $\mathcal{A}'$ and ρ_α. We can apply Theorem 3.24 to deduce that the computations for $P'' \cup \{\leftarrow \rho_\alpha(G)\}$ are totally correct wrt $P \cup \{\leftarrow G\}$.

Every clause C in P'' can be obtained from a clause in P' by adding the negative literals we have removed above. For every SLD-derivation in P' there exists a corresponding derivation in P'', with however additional negative literals in the resolvent. We will show that these negative literals always succeed and thereby establish a one to one correspondence between derivations in P'' and derivations in P'. Let G'_1 be a goal which is an admissible renaming (under α for $\mathcal{A}$ in P) of a goal G_1 for P (i.e. $G'_1 = \rho'_\alpha(G_1)$ for some ρ'_α — only such goals can occur for derivations inside P') which resolves with a clause C_δ (constructed for the characteristic path δ) in P' and with selected literal $\rho'_\alpha(p(\bar{t}))$ leading to a resolvent RG'. Then G'_1 also resolves with a clause C' in P'', under the same selected literal, leading to a resolvent RG'' which has the same atoms as RG' plus possibly some additional negative literals N. Because G'_1 is an admissible renaming of G_1 we know that $p(\bar{t}) \in \gamma_P(A,\tau)$ where $\rho'_\alpha(p(\bar{t})) = \alpha((A,\tau))\sigma'$ for some σ'. Because $p(\bar{t})$ is a concretisation of an element in $\mathcal{A}$ we know that it must be the instance of a

precise concretisation A. For this concretisation A there exists a characteristic tree which contains the characteristic path δ and for which the negative literals corresponding to N are ground and succeed. Therefore, because $p(\bar{t})$ is an instance of A, the literals in N are identical to the ones selected for $P \cup \{\leftarrow A\}$ and are thus also ground and succeed. We can thus immediately (by Theorem 3.24) select them in P'' and can thus construct a corresponding derivation in P'. By inductively applying the above we can prove this result for SLDNF-derivations of any length and any rank.

Similarly, if for every derivation in P'' we impose the (fair) condition that the additional negative literals N are immediately selected, then we can establish a one to one correspondence between derivations in P'' and P'. Also the clauses that had to be added for coveredness are not reachable within P' and can therefore also be removed. So total correctness of P', given the coveredness conditions, holds. □

3.4 A Set Based Algorithm and its Termination

In this section, we present a first, simple (set based) algorithm for partial deduction through characteristic atoms. We will prove correctness as well as termination of this algorithm, given certain conditions.

As in [18, 19], we first define an abstraction operator. One can notice that, by definition, the abstraction operator preserves the characteristic trees.

Definition 3.25 Let $\mathcal{A}$ be a set of characteristic atoms. Also let, for every characteristic tree τ, $\mathcal{A}_\tau$ be defined as $\mathcal{A}_\tau = \{A \mid (A, \tau) \in \mathcal{A}\}$. The operator *abstract* is defined as: $abstract(\mathcal{A}) = \cup_\tau \{(msg(\mathcal{A}_\tau), \tau)\}$.

In other words, only one characteristic atom per characteristic tree is allowed in the resulting abstraction. For example, in the case that $\mathcal{A} = \{(p(a), \{\langle 1 \circ 1 \rangle\}), (p(b), \{\langle 1 \circ 1 \rangle\})\}$, we obtain $abstract(\mathcal{A}) = \{(p(X), \{\langle 1 \circ 1 \rangle\})\}$.

Definition 3.26 Let A be an ordinary atom, U an unfolding rule and P a program. Then $chatom(A, P, U) = (A, \tau)$ where $chtree(\leftarrow A, P, U) = \tau$. We extend *chatom* to sets of atoms: $chatoms(\mathcal{A}, P, U) = \{chatom(A, P, U) \mid A \in \mathcal{A}\}$.

Note that A is a precise concretisation of $chatom(A, P, U)$.

The following algorithm for partial deduction with characteristic atoms is parametrised by an unfolding rule U, thus leaving the particulars of local control unspecified. Recall that $BA_P(A, \tau_A)$ represents the body atoms of (A, τ) (see Definition 3.9).

Algorithm 3.27

Input
a program P and a goal G

Output
a specialised program P'

Initialisation
$k := 0$ and $\mathcal{A}_0 := chatoms(\{A \mid A \text{ is an atom in } G\}, P, U)$

Repeat
$\mathcal{A}_{k+1} := abstract(\mathcal{A}_k \cup chatoms(\{BA_P(A, \tau_A) \mid (A, \tau_A) \in \mathcal{A}_k\}, P, U))$
Until $\mathcal{A}_k = \mathcal{A}_{k+1}$ (modulo variable renaming)
$\mathcal{A}' := \mathcal{A}_k$
$P' :=$ a partial deduction of P wrt $\mathcal{A}'$ and some renaming function ρ_α

Let us illustrate the operation of Algorithm 3.27 on Example 2.9.

Example 3.28 Let $G = \leftarrow member(a,[a]), member(a,[a,b])$ and P be the *member* program from Example 2.9. Also let $\tau = chtree(\leftarrow member(a,[a]), P, U) = chtree(\leftarrow member(a,[a,b]), P, U) = \{\langle 1 \circ 1 \rangle\}$ (see Figure 3 for the corresponding SLD-trees). The algorithm operates as follows:

1. $\mathcal{A}_0 = \{(member(a,[a]),\tau), (member(a,[a,b]),\tau)\}$
2. $BA_P(member(a,[a]),\tau) = BA_P(member(a,[a,b]),\tau) = \emptyset$,
   $\mathcal{A}_1 = abstract(\mathcal{A}_0) = \{(member(a,[a|T]),\tau)\}$
3. $BA_P(member(a,[a|T]),\tau) = \emptyset$, $\mathcal{A}_2 = abstract(\mathcal{A}_1) = \mathcal{A}_1$ and we have reached the fixpoint $\mathcal{A}'$.

A partial deduction P' wrt $\mathcal{A}'$ and ρ_α with $\alpha((member(a,[a|T]),\tau)) = m_1(T)$ is:

$m_1(X) \leftarrow$

$\mathcal{A}'$ is covered and every atom in G is a concretisation of an atom in $\mathcal{A}'$. Hence Theorem 3.16 can be applied: we obtain the renamed goal $G' = \rho_\alpha(G) = \leftarrow m_1([]), m_1([b])$ and $P' \cup \{G'\}$ yields the correct result.

In the remainder of this subsection, we formally prove correctness of Algorithm 3.27, as well as its termination under a certain condition.

By an *expression* we mean either a term, an atom, a literal, a conjunction, a disjunction or a program clause. Expressions are constructed using the alphabet $\mathcal{A}_P$ which we implicitly assume underlying the program P under consideration. Note that $\mathcal{A}_P$ may contain additional symbols not occurring in P, but that, unless explicitly stated otherwise, $\mathcal{A}_P$ contains only finitely many constant, function and predicate symbols ! To simplify the presentation, we also assume that, when talking about expressions, predicate symbols are treated like new functors and for conjunctions, disjunctions and clauses, the required connectives are also represented as new functors (e.g. $\wedge$ and $\leftarrow$ are binary functors which cannot be confounded with the original functors).

The following well-founded measure function is taken from [21] (also in the extended version of [54]):

Definition 3.29 Let *Expr* denote the sets of expressions. We define the function $s : Expr \rightarrow \mathbb{N}$ counting symbols by:

- $s(t) = 1 + s(t_1) + \ldots + s(t_n)$ if $t = f(t_1, \ldots, t_n)$, $n > 0$
- $s(t) = 1$ otherwise

Let the number of distinct variables in an expression t be $v(t)$. We now define the function $h : Expr \rightarrow \mathbb{N}$ by $h(t) = s(t) - v(t)$.

The well-founded measure function h has the property that $h(t) \geq 0$ and $h(t) > 0$ for any non-variable t. The following important lemma is proven for $h(.)$ in [20] (see also [54]).

Lemma 3.30 If A and B are expressions such that B is strictly more general than A, then $h(A) > h(B)$.

It follows that, for every expression A, there are no infinite chains of strictly more general expressions.

Definition 3.31 Let $\mathcal{A}$ be a set of characteristic atoms and let $T = \langle \tau_1, \ldots, \tau_n \rangle$ be a finite vector of characteristic trees. We then define the *weight vector* of $\mathcal{A}$ wrt T by $hvec_T(\mathcal{A}) = \langle w_1, \ldots, w_n \rangle$ where

- $w_i = \infty$ if $\mathcal{A}_{\tau_i} = \emptyset$
- $w_i = \sum_{A \in \mathcal{A}_{\tau_i}} h(A)$ if $\mathcal{A}_{\tau_i} \neq \emptyset$

The set of weight vectors is partially ordered by the usual order relation among vectors: $\langle w_1, \ldots, w_n \rangle \leq \langle v_1, \ldots, v_n \rangle$ iff $w_1 \leq v_1, \ldots, w_n \leq v_n$. Also, given a non-strict partial order $\leq_S$ on a set S, we from now on assume that the associated equivalence relation $\equiv_S$ and the associated strict partial order $>_S$ are implicitly defined in the usual way:

- $s_1 \equiv_S s_2$ iff $s_1 \leq s_2 \wedge s_2 \leq s_1$ and $s_1 <_S s_2$ iff $s_1 \leq s_2 \wedge s_2 \not\leq s_1$.

The set of weight vectors is well-founded (no infinite strictly decreasing sequences exist) because the weights of the atoms are well-founded.

Proposition 3.32 Let P be a normal program, U an unfolding rule and let $T = \langle \tau_1, \ldots, \tau_n \rangle$ be a finite vector of characteristic trees. For every pair of finite sets of characteristic atoms $\mathcal{A}$ and $\mathcal{B}$, such that the characteristic trees of their elements are in T, we have that one of the following holds:

- $abstract(\mathcal{A} \cup \mathcal{B}) = \mathcal{A}$ (up to variable renaming) or
- $hvec_T(abstract(\mathcal{A} \cup \mathcal{B})) < hvec_T(\mathcal{A})$.

Proof Let $hvec_T(\mathcal{A}) = \langle w_1, \ldots, w_n \rangle$ and let $hvec_T(abstract(\mathcal{A} \cup \mathcal{B})) = \langle v_1, \ldots, v_n \rangle$. Then for every $\tau_i \in T$ we have two possible cases:

- $\{msg(\mathcal{A}_{\tau_i} \cup \mathcal{B}_{\tau_i})\} = \mathcal{A}_{\tau_i}$ (up to variable renaming). In this case the abstraction operator performs no modification for τ_i and $v_i = w_i$.
- $\{msg(\mathcal{A}_{\tau_i} \cup \mathcal{B}_{\tau_i})\} = \{M\} \neq \mathcal{A}_{\tau_i}$ (up to variable renaming). In this case $(M, \tau_i) \in abstract(\mathcal{A} \cup \mathcal{B})$, $v_i = h(M)$ and there are three possibilities:
  - $\mathcal{A}_{\tau_i} = \emptyset$. In this case $v_i < w_i = \infty$.
  - $\mathcal{A}_{\tau_i} = \{A\}$ for some atom A. In this case M is strictly more general than A (by definition of *msg* because $M \neq A$ up to variable renaming) and hence $v_i < w_i$.
  - The cardinality of $\mathcal{A}_{\tau_i}$ is greater than 1. In this case M is more general (but not necessarily strictly more general) than any atom in $\mathcal{A}_{\tau_i}$ and $v_i < w_i$ because at least one atom is removed by the abstraction.

We have that $\forall i \in \{1, \ldots, n\} : v_i \leq w_i$ and either the abstraction operator performs no modification (and $\vec{v} = \vec{w}$) or the well-founded measure $hvec_T$ strictly decreases. □

Theorem 3.33 If the number of distinct characteristic trees is finite then Algorithm 3.27 terminates and generates a partial deduction satisfying the requirements of Theorem 3.16 for any goal G' whose atoms are instances of atoms in G.

Proof Reaching the fixpoint guarantees that all predicates in the bodies of resultants are precise concretisations of at least one characteristic atom in $\mathcal{A}_k$, i.e. $\mathcal{A}_k$ is P-covered. Furthermore *abstract* always generates more general characteristic atoms (even in the sense that any precise concretisation of an atom in $\mathcal{A}_i$ is a precise concretisation of an

atom in $\mathcal{A}_{i+1}$ — this follows immediately from Definitions 3.3 and 3.25). Hence, because any instance of an atom in the goal G is a precise concretisation of a characteristic atom in $\mathcal{A}_0$, the conditions of Theorem 3.16 are satisfied for goals G' whose atoms are instances of atoms in G, i.e. G' is covered by $\mathcal{A}_k$ in P. Finally, termination is guaranteed by Proposition 3.32, given that the number of distinct characteristic trees is finite. □

The method for partial deduction as described in this section, using the framework of Section 3, has been called *ecological* partial deduction in [39] because it guarantees the preservation of characteristic trees. A prototype partial deduction system, using Algorithm 3.27, has been implemented and experiments with it can be found in [39] and [55]. We will however further improve the algorithm in the next section and present extensive benchmarks in Section 5.2.

Let us conclude this section with some comments related to [41], which also solves the problem of preserving characteristic trees upon generalisation. In fact, [41] achieves this by incorporating disequality constraints into the partial deduction process. Note that in this section and paper, the characteristic tree τ inside a characteristic atom (A, τ) can also be seen as an implicit representation of constraints on A. However, here these constraints are used only locally and are not propagated towards other characteristic atoms, while in [41] the constraints are propagated and thus used globally. Whether this has any significant influence in practice remains to be seen. On the other hand, the method in this section is conceptually simpler and can handle *any* unfolding rule as well as *normal* logic programs, while [41] is currently limited to purely determinate unfoldings (without a lookahead) and definite programs.

4 Removing Depth Bounds by Adding Global Trees

Having solved the first problem related to characteristic trees, their preservation upon generalisation, we now turn to the second problem: getting rid of the associated depth bound.

4.1 The Depth Bound Problem

The algorithm for ecological partial deduction presented in Section 3.4 only terminates when imposing a depth bound on characteristic trees. In this section we present 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 will return to the relation of neighbourhoods to characteristic trees in Section 5.4.)

When, for the given program, query and unfolding rule, the above sketched method generates a *finite number of different characteristic trees*, its global control regime guarantees termination and correctness of the specialised program as well as "perfect" polyvariance: *For every predicate, exactly one specialised version is produced for each of its different associated characteristic trees.* Now, the algorithm of the previous section, as well as all earlier approaches based on characteristic trees [21, 18, 41], achieves the mentioned finiteness condition at the cost of imposing an ad hoc (typically very large) depth bound on characteristic trees. However, for a fairly *large class of realistic programs* (and unfolding rules), *the number of different characteristic trees generated, is not naturally*

bounded. In those cases, the underlying depth bound will have to ensure termination, meanwhile propagating its ugly, ad hoc nature into the resulting specialised program.

We illustrate this problem through some examples, setting out with a slightly artificial, but very simple one.

Example 4.1 The following is the well known reverse with accumulating parameter where a list type check on the accumulator has been added.

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

As can be noticed in Figure 4, (determinate [21, 18, 41] and well-founded [6, 53, 52], among others) unfolding produces an infinite number of different characteristic atoms, all with a different characteristic tree. Imposing a depth bound of say 100, we obtain termination, but the algorithm produces 100 different *reverse* versions and the specialised program 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. To be fair, it must be admitted that 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. As illustrated by Example 4.1 above, this state of affairs will often already be changed when one adds type checking in the style of [22] to even the simplest logic programs.

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. Another example is the "mixed" meta-interpreter of [28, 42] (sometimes called *InstanceDemo*; part of it is depicted in Figure 5) for the ground representation in which the goals are "lifted" to the non-ground representation

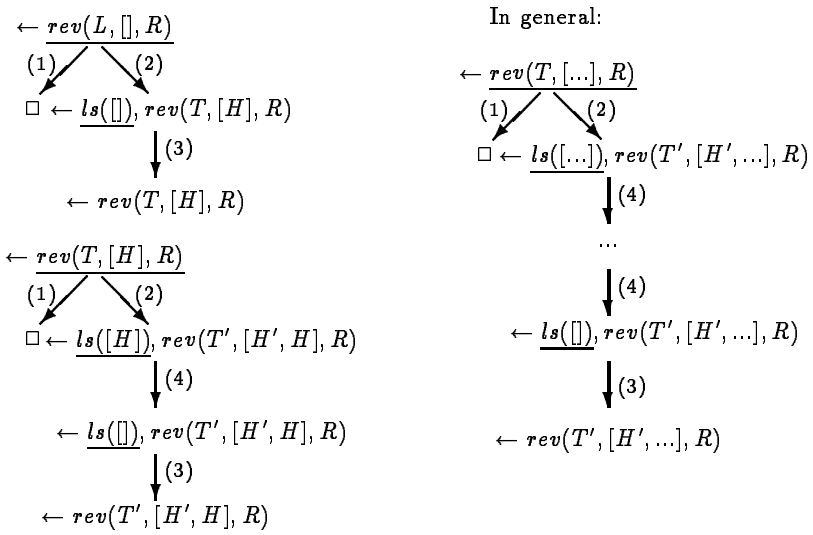


Figure 4: SLD-trees for Example 4.1

for resolution. To perform the lifting, an accumulating parameter is used to keep track of the variables that have already been encountered. This accumulator influences the computation: Upon encountering a new variable, the program inspects the accumulator.

Example 4.2 Let $A = l\_mng(Lg, Ln, [sub(N, X)], S)$ and P be the program of Figure 5 (this situation arose in an actual experiment). As can be seen in Figure 6, unfolding A (e.g. using well-founded measures), the atom $l\_mng(Tg, Tn, [sub(N, X), sub(J, Hn)], S)$ is added at the global (control) level. Notice that the third argument has grown (i.e. we have an accumulator).
So, when in turn unfolding $l\_mng(Tg, Tn, [sub(N, X), sub(J, Hn)], S)$, we will obtain a deeper characteristic tree (because *mng* traverses the third argument and thus needs one more step to reach the end) with

$$l\_mng(Tg', Tn', [sub(N, X), sub(J, Hn), sub(J', Hn')], S)$$

as one of its leaves. An infinite sequence of ever growing characteristic trees results and again, as in Example 4.1, we obtain non-termination without a depth bound, and very unsatisfactory (ad hoc) specialisations with it.

Summarising, computations influenced by one or more growing data structures are by no means rare and will, very often, lead to ad hoc behaviour of partial deduction, where the global control is founded on characteristic trees with a depth bound. In the next section, we show how this annoying depth bound can be removed without endangering termination.

Program:

(1) $make\_non\_ground(GrTerm, NgTerm) \leftarrow$
$mng(GrTerm, NgTerm, [], Sub)$

(2) $mng(var(N), X, [], [sub(N, X)]) \leftarrow$

(3) $mng(var(N), X, [sub(N, X)|T], [sub(N, X)|T]) \leftarrow$

(4) $mng(var(N), X, [sub(M, Y)|T], [sub(M, Y)|T1]) \leftarrow$
$not(N = M),\ mng(var(N), X, T, T1)$

(5) $mng(struct(F, GrArgs), struct(F, NgArgs), InSub, OutSub) \leftarrow$
$l\_mng(GrArgs, NgArgs, InSub, OutSub)$

(6) $l\_mng([], [], Sub, Sub) \leftarrow$

(7) $l\_mng([GrH|GrT], [NgH|NgT], InSub, OutSub) \leftarrow$
$mng(GrH, NgH, InSub, InSub1),$
$l\_mng(GrT, NgT, InSub1, OutSub)$

Example query:

$\leftarrow make\_non\_ground(struct(f, [var(1), var(2), var(1)]), F)$
$\leadsto$ c.a.s. $\{F/struct(f, [Z, V, Z])\}$

Figure 5: Lifting the ground representation

4.2 Partial Deduction using Global Trees

A general framework for global control, not relying on any depth bounds, is proposed in [54]. Marked trees (m-trees) are introduced to register descendency relationships among atoms at the global level. These trees are subdivided into classes of nodes and associated measure functions map nodes to well-founded sets. The overall tree is then kept finite through ensuring monotonicity of the measure functions and termination of the algorithm follows, provided the abstraction operator (on atoms) is similarly well-founded. It is to this framework that we now turn for inspiration on how to solve the depth bound problem uncovered in Section 4.1.

First, we have chosen to use the term "global tree" rather than "marked tree" in the present paper, because it better indicates its functionality. Moreover, global trees rely on a well-quasi-order between nodes, rather than a well-founded one, to ensure their finiteness. Apart from that, in essence, their structure is similar: They register which atoms derive from which at the global control level. The initial part of such a tree, showing the descendency relationship between the atom in the root and those in the (dangling) leaves of the SLDNF-tree in Figure 6, is depicted in Figure 7.

Now, the basic idea will be to have just a single class covering the whole global tree structure and to watch over the evolution of characteristic trees associated to atoms along its branches. Obviously, just measuring the depth of characteristic trees would be far too crude: Global branches would be cut off prematurely and entirely unrelated atoms could be mopped together through generalisation, resulting in unacceptable specialisation losses. No, as can be seen in Figure 4, 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

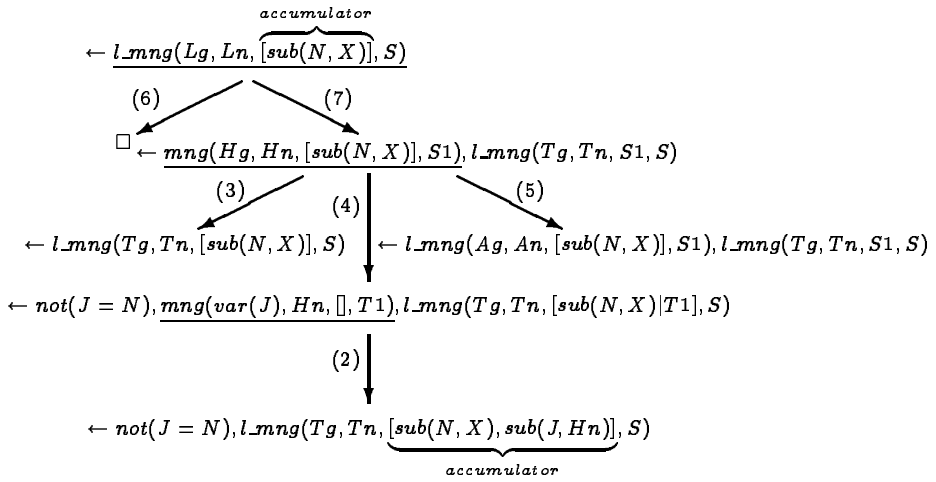


Figure 6: Accumulator growth in Example 4.2

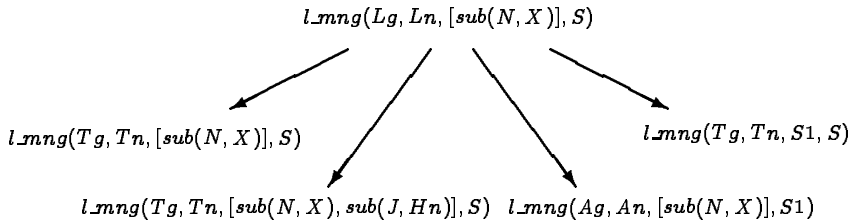


Figure 7: Initial Section of a Global Tree for Example 4.2 and the Unfolding of Figure 6

in Example 4.1), it seems reasonable to stop expanding the global tree, generalise the offending atoms and produce a specialised procedure for the generalisation instead.

However, a closer look at the following variation of Example 4.2 shows that also this approach would sometimes overgeneralise and consequently fall short of providing sufficiently detailed polyvariance.

Example 4.3 Reconsider the program in Figure 5, and suppose that determinate unfolding is used for the local control.
The atom $A = mng(G, struct(cl, [struct(f, [X, Y])|B]), [], S)$ will now be the starting point for partial deduction (also this situation arose in an actual experiment). When unfolding A (see Figure 8), we obtain an SLD-tree with $mng(H, struct(f, [X, Y]), [], S1)$ in one of its leaves. If we subsequently (determinately) unfold the latter atom, we obtain a tree that is "larger" than its predecessor, also in the more refined sense. Potential nontermination would therefore be detected and an abstraction operator executed. However, the atoms in the leaves of the second tree are more general than those already met, and simply continuing partial deduction without generalisation will lead to natural termination without any depth bound intervention.

Example 4.3 demonstrates that only measuring growth of characteristic trees, even in a refined way, does not always lead to satisfactory specialisation. In fact, whenever the (local) unfolding rule does not unfold "as deeply as possible" (for whatever reason, e.g.

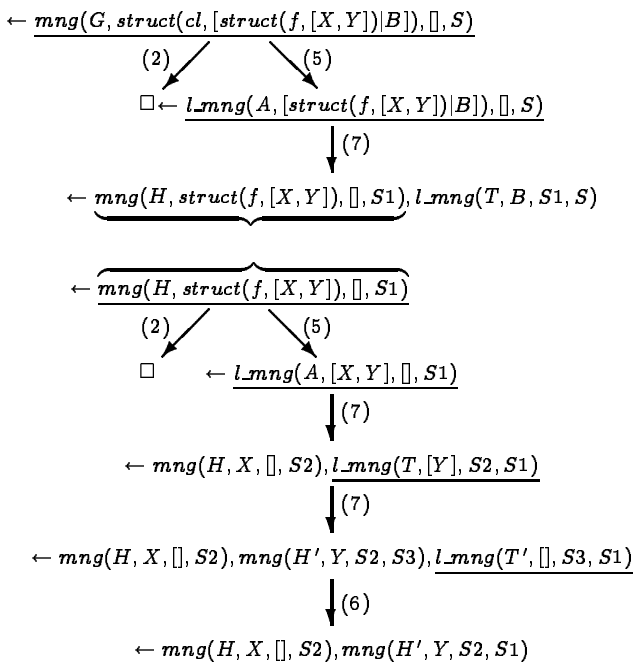


Figure 8: SLD-trees for Example 4.3

efficiency of the specialised program or because it is not refined enough), then a growing characteristic tree might simply be caused by splitting the "maximally deep tree" in such a way that the second part "contains" the first part. (In Example 4.3, an unfolding rule based on well-founded measures could have continued unfolding more deeply for the first atom, thus avoiding the fake growing problem in this case.)

Luckily, the same example also suggests a solution to this problem: Rather than measuring and comparing characteristic trees, we will *scrutinise entire characteristic atoms, comparing both the syntactic content of the ordinary atoms they contain and the associated characteristic trees.* Accordingly, the global tree nodes will not be labelled by plain atoms as in [54], but by entire characteristic atoms.

The rest of this section, then, contains the formal elaboration of this new approach. In Subsection 4.3.1, we first extend the familiar generalisation notion defined on atoms to characteristic atoms, and subsequently proceed to introduce the precise comparison operation to be used on the latter. Some important properties connecting both operations are also stated and proved. Next, Subsection 4.4 introduces global trees and a characterisation of their finiteness. Finally, our refined algorithm for partial deduction is presented and proved correct and terminating in Subsection 4.5.

4.3 More on Characteristic Atoms

4.3.1 Generalising Characteristic Atoms

In this subsection we extend the notions of variants, instances and generalisations, familiar for ordinary atoms, to characteristic trees and atoms.

Henceforth, we will use symbols like $\prec$, $\succ$ (possibly annotated by some subscript)

to refer to strict and $\preceq$, $\succeq$ to refer to non-strict partial orders. We will use either "directionality" as is convenient in the context. Remember that, given a non-strict partial order $\preceq$, we will use the associated equivalence relation $\equiv$ and strict partial order $\prec$, as defined in Section 3.4.

For ordinary atoms, $A_1 \preceq A_2$ will denote that A_1 is more general than A_2. We will now define a similar notion for characteristic atoms along with an operator to compute the most specific generalisation. In a first attempt one might use the concretisation function to that end, i.e. one could say that (A, τ_A) is more general than (B, τ_B) iff $\gamma_P(A, \tau_A) \supseteq \gamma_P(B, \tau_B)$. The problem with this definition, from a practical point of view, is that this notion is undecidable in general and a most specific generalisation is uncomputable. For instance $\gamma_P(A, \tau) = \gamma_P(A, \tau \cup \{\delta\})$ holds iff for every instance of A, the last goal associated with δ finitely fails. Deciding this is equivalent to the halting problem. We will therefore present a safe but computable approximation of the above notion, for which a most specific generalisation can be easily computed and which has some nice properties in the context of a partial deduction algorithm (see e.g. Definition 4.21 below).

We first define an ordering on characteristic trees. In that context, the following notation will prove to be useful: $prefix(\tau) = \{\delta \mid \exists \gamma \text{ such that } \delta\gamma \in \tau\}$.

Definition 4.4 Let τ_1, τ_2 be characteristic trees. We say that τ_1 is *more general* than τ_2, and denote this by $\tau_1 \preceq_\tau \tau_2$, iff

1. $\delta \in \tau_1 \Rightarrow \delta \in prefix(\tau_2)$ and
2. $\delta' \in \tau_2 \Rightarrow \exists \delta \in prefix(\{\delta'\})$ such that $\delta \in \tau_1$.

Note that $\preceq_\tau$ is a non-strict partial order on the set of characteristic trees and that $\tau_1 \preceq_\tau \tau_2$ is equivalent to saying that τ_2 can be obtained from τ_1 by attaching sub-trees to the leaves of τ_1.

Example 4.5 Given $\tau_1 = \{\langle 1 \circ 3 \rangle\}$, $\tau_2 = \{\langle 1 \circ 3, 2 \circ 4 \rangle\}$ and $\tau_3 = \{\langle 1 \circ 3 \rangle, \langle 1 \circ 4 \rangle\}$ we have that $\tau_1 \preceq_\tau \tau_2$ and $\tau_1 \preceq_\tau \tau_2$ but not that $\tau_1 \preceq_\tau \tau_3$ nor $\tau_2 \preceq_\tau \tau_3$. We also have that $\{\langle\rangle\} \prec_\tau \tau_1$ but not that $\emptyset \preceq_\tau \tau_1$. In fact, $\{\langle\rangle\} \preceq_\tau \tau$ holds for any $\tau \neq \emptyset$, while $\emptyset \preceq_\tau \tau$ only holds for $\tau = \emptyset$. Also $\tau \preceq_\tau \{\langle\rangle\}$ only holds for $\tau = \{\langle\rangle\}$ and $\tau \preceq_\tau \emptyset$ only holds for $\tau = \emptyset$.

The next two lemmas respectively establish a form of anti-symmetry as well as transitivity of the order relation on characteristic trees.

Lemma 4.6 Let τ_1, τ_2 be two characteristic trees. Then $\tau_1 \preceq_\tau \tau_2$ and $\tau_2 \preceq_\tau \tau_1$ iff $\tau_1 = \tau_2$.

Proof The if part is obvious because δ and δ' can be taken as prefixes of themselves for the two points of Definition 4.4.

For the only-if part, let us suppose that $\tau_1 \preceq_\tau \tau_2$ and $\tau_2 \preceq_\tau \tau_1$ but $\tau_1 \neq \tau_2$. This means that there must be a characteristic path δ in τ_1 which is not in τ_2 (otherwise we reverse the roles of τ_1 and τ_2). We know however by point 1 of Definition 4.4 that an extension $\delta_x = \delta\gamma$ of p must be in τ_2, as well as by point 2 of the same definition that a prefix δ_s of δ must be in τ_2. Therefore τ_2 contains the paths δ_x and δ_s, where $\delta_x \neq \delta_s$ and $\delta_x = \delta_s\gamma'\gamma$. But this is impossible by Lemma 3.20. □

Lemma 4.7 Let τ_1, τ_2 and τ_3 be characteristic trees. If $\tau_1 \preceq_\tau \tau_2$ and $\tau_2 \preceq_\tau \tau_3$ then $\tau_1 \preceq_\tau \tau_3$.

Proof Immediate from Definition 4.4 because a prefix of a prefix remains a prefix. □

Lemma 4.8 Let τ_1, τ_2 be two non-empty characteristic trees. Then there exists a unique most specific generalisation of τ_1 and τ_2, denoted by $msg(\tau_1, \tau_2)$, obtained by taking the largest common initial subtree of τ_1 and τ_2:

- $\delta \in msg(\tau_1, \tau_2)$ iff
  - $\delta \in prefix(\tau_1) \wedge \delta \in prefix(\tau_2)$
  - $\nexists \delta''$ such that $\delta\delta'' \in prefix(\tau_1) \wedge \delta\delta'' \in prefix(\tau_2)$.

Proof First, $\tau = msg(\tau_1, \tau_2)$ is a generalisation of both τ_1 and τ_2. Indeed, if $\delta \in \tau$ then, by definition of $msg(\tau_1, \tau_2)$, point 1 of Definition 4.4 is verified. For point 2 it suffices to notice that $\langle\rangle$ is both in $prefix(\tau_1)$ and in $prefix(\tau_2)$ because τ_1 and τ_2 are not empty. Therefore, given $\delta' \in \tau_1$ (respectively τ_2), the largest prefix of δ' which is both in $prefix(\tau_1)$ and in $prefix(\tau_2)$ must exist and trivially satisfies the last point of the definition of $msg(\tau_1, \tau_2)$. Hence point 2 of Definition 4.4 is also satisfied for τ_1 (respectively τ_2).

$\tau = msg(\tau_1, \tau_2)$ is also a most specific generalisation because any further extension of τ must, by definition, have a path which is not in $prefix(\tau_1)$ or $prefix(\tau_2)$ and can therefore not be more general than both τ_1 and τ_2.

Also, any characteristic tree τ' which is not a common initial subtree of τ_1 and τ_2 (i.e. is incomparable with τ according to $\preceq_\tau$) is not more general than τ_1 and τ_2, because both τ_1 and τ_2 must be obtained from τ' by attaching sub-trees to the leaves. Hence the most specific generalisation is unique. □

If both $\tau_1 = \emptyset$ and $\tau_2 = \emptyset$ then $\emptyset$ is the unique most specific generalisation and we therefore define $msg(\emptyset, \emptyset) = \emptyset$. Only in case one of the characteristic trees is empty while the other is not, do we leave the msg undefined.

Example 4.9 Given $\tau_1 = \{\langle 1 \circ 3\rangle\}$, $\tau_2 = \{\langle 1 \circ 3, 2 \circ 4\rangle\}$, $\tau_3 = \{\langle 1 \circ 3\rangle, \langle 1 \circ 4\rangle\}$, $\tau_4 = \{\langle 1 \circ 3, 2 \circ 4\rangle, \langle 1 \circ 3, 2 \circ 5\rangle\}$, we have that $msg(\tau_1, \tau_2) = \tau_1$, $msg(\tau_1, \tau_3) = msg(\tau_2, \tau_3) = \{\langle\rangle\}$ and $msg(\tau_2, \tau_4) = \tau_1$.

Definition 4.10 A characteristic atom (A_1, τ_1) is *more general* than another characteristic atom (A_2, τ_2), denoted by $(A_1, \tau_1) \preceq_{ca} (A_2, \tau_2)$, iff $A_1 \preceq A_2$ and $\tau_1 \preceq_\tau \tau_2$. Also (A_1, τ_1) is said to be a *variant* of (A_2, τ_2) iff $(A_1, \tau_1) \equiv_{ca} (A_2, \tau_2)$.

The following proposition shows that the above definition safely approximates the optimal but impractical "more general" definition based on the set of concretisations.

Proposition 4.11 Let $(A, \tau_A), (B, \tau_B)$ be two characteristic atoms. If $(A, \tau_A) \preceq_{ca} (B, \tau_B)$ then $\gamma_P(A, \tau_A) \supseteq \gamma_P(B, \tau_B)$.

Proof Let C be a precise concretisation of (B, τ_B). By Definition 3.3, there must be an unfolding rule U such that $chtree(\leftarrow C, P, U) = \tau_B$. By Definition 4.4, τ_B is obtained by attaching subtrees to the leaves of τ_A and therefore, for some unfolding rule U' which unfolds less than U, we must have that $chtree(\leftarrow C, P, U') = \tau_A$. Furthermore, by Definition 4.10, B is an instance of A and therefore C is also an instance of A. We can conclude that C is also a precise concretisation of (A, τ_A). In other words, any precise concretisation of (B, τ_B) is also a precise concretisation of (A, τ_A). The result for general concretisations follows immediately from this by Definition 3.3. □

The converse of the above proposition does of course not hold. Take the *member* program from Example 2.9 and let $A = member(a, [a])$, $\tau = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$ and $\tau' = \{\langle 1 \circ 1\rangle\}$. Then $\gamma_P(A, \tau) = \gamma_P(A, \tau') = \{member(a, [a])\}$ (because the resolvent $\leftarrow member(a, [])$ associated with $\langle 1 \circ 2\rangle$ fails finitely) but neither $(A, \tau) \preceq_{ca} (A, \tau')$ nor $(A, \tau') \preceq_{ca} (A, \tau)$ hold.

The following is an immediate corollary of the previous proposition.

Corollary 4.12 Let P be a program and CA, CB be two characteristic atoms such that $CA \preceq_{ca} CB$. If CB is a P-characteristic atom then so is CA.

Finally, we extend the notion of *most specific generalisation (msg)* to characteristic atoms:

Definition 4.13 Let (A_1, τ_1), (A_2, τ_2) be two characteristic atoms such that $msg(\tau_1, \tau_2)$ is defined. Then $msg((A_1, \tau_1), (A_2, \tau_2)) = (msg(A_1, A_2), msg(\tau_1, \tau_2))$.

Note that the above *msg* for characteristic atoms is still unique up to variable renaming. Its further extension to *sets* of characteristic atoms (rather than just pairs) is straightforward, and will not be included explicitly.

4.3.2 Well-Quasi Ordering of Characteristic Atoms.

We now proceed to introduce another order relation on characteristic atoms. It will be instrumental in guaranteing termination of the refined partial deduction method to be presented.

Definition 4.14 A poset $V, \leq_V$ is called *well-quasi-ordered (wqo)* iff for any infinite sequence of elements $e_1, e_2, \ldots$ in V there are $i < j$ such that $e_i \leq_V e_j$. We also say that $\leq_V$ is a *well-quasi order (wqo) on* V.

An interesting wqo is the homeomorphic embedding relation $\trianglelefteq$. It has been adapted from [16, 17], where it is used in the context of term rewriting systems, for use in supercompilation in [65]. Its usefulness as a stop criterion for partial evaluation is also discussed and advocated in [50]. Some complexity results can be found in [66] (also summarised in [50]).

Recall that expressions are formulated using the alphabet $\mathcal{A}_P$ which we implicitly assume underlying the programs and queries under consideration. Remember that it may contain symbols occurring in no program and query but that it contains only finitely many constant, function and predicate symbols. The latter property is of crucial importance for some of the propositions and proofs below. In Section 5.1 we will present a way to lift this restriction.

Definition 4.15 The *homeomorphic embedding* relation $\trianglelefteq$ on expressions is defined inductively as follows:

1. $X \trianglelefteq Y$ for all variables X, Y
2. $s \trianglelefteq f(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i
3. $f(s_1, \ldots, s_n) \trianglelefteq f(t_1, \ldots, t_n)$ if $\forall i \in \{1, \ldots, n\} : s_i \trianglelefteq t_i$.

Example 4.16 We have: $p(a) \trianglelefteq p(f(a))$, $X \trianglelefteq X$, $p(X) \trianglelefteq p(f(Y))$, $p(X, X) \trianglelefteq p(X, Y)$ and $p(X, Y) \trianglelefteq p(X, X)$.

Proposition 4.17 The relation $\trianglelefteq$ is a wqo on the set of expressions over a finite alphabet.

Proof (Proofs similar to this one are standard in the literature. We include it for completeness.) We first need the following concept from [17]. Let $\leq$ be a relation on a set S of functors (of arity ≥ 0). Then the *embedding extension* of $\leq$ is a relation $\leq_{emb}$ on terms constructed (only) from the functors in S which is inductively defined as follows:

1. $s \leq_{emb} f(t_1, \ldots, t_n)$ if $s \leq_{emb} t_i$ for some i
2. $f(s_1, \ldots, s_n) \leq_{emb} g(t_1, \ldots, t_n)$ if $f \leq g$ and $\forall i \in \{1, \ldots, n\} : s_i \leq_{emb} t_i$.

We define the relation $\leq$ on the set $\mathcal{S} = \mathcal{V} \cup \mathcal{F}$ of symbols, containing the (infinite) set of variables $\mathcal{V}$ and the (finite) set of functors and predicates $\mathcal{F}$, as the least relation satisfying:

- $x \leq y$ if $x \in \mathcal{V} \wedge y \in \mathcal{V}$
- $f \leq f$ if $f \in \mathcal{F}$

This relation is a wqo on $\mathcal{S}$ (because $\mathcal{F}$ is finite) and hence by Kruskal's theorem (see [17]), its embedding extension to terms, $\leq_{emb}$, which is by definition identical to $\trianglelefteq$, is a wqo on the set of expressions. □

The intuition behind Definition 4.15 is that when some structure re-appears within a larger one, it is homeomorphically embedded by the latter. As is argued in [50] and [65], this provides a good starting point for detecting growing structures created by possibly non-terminating processes.

However, as can be observed in Example 4.16, the homeomorphic embedding relation $\trianglelefteq$ as defined in Definition 4.15 is rather crude wrt variables. In fact, all variables are treated as if they were the same variable, a practice which is clearly undesirable in a logic programming context. Intuitively, in the above example, $p(X, Y) \trianglelefteq p(X, X)$ is acceptable, while $p(X, X) \trianglelefteq p(X, Y)$ is not.[6]

To remedy the problem, we refine the above introduced homeomorphic embedding as follows:

Definition 4.18 Let A, B be expressions. Then B *(strictly homeomorphically) embeds* A, written as $A \trianglelefteq^* B$, iff $A \trianglelefteq B$ and A is not a strict instance of B.

[6] $p(X, X)$ can be seen as standing for something like $and(eq(X, Y), p(X, Y))$ which clearly embeds $p(X, Y)$, but the reverse does not hold.

Example 4.19 We now have that $p(X,Y) \trianglelefteq^* p(X,X)$ but not $p(X,X) \trianglelefteq^* p(X,Y)$. Note that still $X \trianglelefteq^* Y$ and $X \trianglelefteq^* X$.

Theorem 4.20 The relation $\trianglelefteq^*$ is a wqo on the set of expressions over a finite alphabet.

Proof In Appendix B □

We now extend the embedding relation of Definition 4.18 to characteristic atoms. Notice that the relation $\preceq_\tau$ is not a wqo on characteristic trees, even in the context of a given fixed program P. For example, take a look at the infinite sequence of characteristic trees depicted in Figure 4. None of these trees is an instance of any other tree.

One way to obtain a wqo is to first define a term representation of characteristic trees and then use the embedding relation $\trianglelefteq^*$ with this term representation.

Definition 4.21 By $\lceil . \rceil$ we denote a total mapping from characteristic trees to terms (expressible in some finite alphabet) such that:

- $\tau_1 \prec_\tau \tau_2 \Rightarrow \lceil \tau_1 \rceil \prec \lceil \tau_2 \rceil$ (i.e. $\lceil . \rceil$ is strictly monotonic) and
- $\lceil \tau_1 \rceil \trianglelefteq^* \lceil \tau_2 \rceil \Rightarrow msg(\tau_1, \tau_2)$ is defined.

The following proposition establishes that such a mapping $\lceil . \rceil$ actually exists.

Proposition 4.22 A function $\lceil . \rceil$ satisfying Definition 4.21 exists.

Proof In Appendix C. The difficult part is to ensure strict monotonicity. The "trick" is to represent leaves of the characteristic trees by variables. □

From now on we will fix $\lceil . \rceil$ to be one particular mapping satisfying Definition 4.21.

Definition 4.23 Let $(A_1,\tau_1),(A_2,\tau_2)$ be characteristic atoms. We say that (A_2,τ_2) *embeds* (A_1,τ_1), denoted by $(A_1,\tau_1) \trianglelefteq^*_{ca} (A_2,\tau_2)$, iff $A_1 \trianglelefteq^* A_2$ and $\lceil \tau_1 \rceil \trianglelefteq^* \lceil \tau_2 \rceil$.

Proposition 4.24 Let $\mathcal{A}$ be a set of P-characteristic atoms. Then $\mathcal{A}, \trianglelefteq^*_{ca}$ is well-quasi-ordered.

Proof Let $\mathcal{E}$ be the set of expressions in the finite alphabet $\mathcal{A}_P$. By Theorem 4.20, $\trianglelefteq^*$ is a wqo on $\mathcal{E}$. Let $\mathcal{F}$ be the set of functors consisting of just one binary functor $ca/2$ as well as all the elements of $\mathcal{E}$ as constant symbols. Let us extend $\trianglelefteq^*$ to $\mathcal{F}$ by (only) adding that $ca \trianglelefteq^* ca$. $\trianglelefteq^*$ is still a wqo on $\mathcal{F}$, and hence, by Kruskal's theorem (see e.g. [17]), its embedding extension (see proof of Proposition 4.17) to terms constructed from $\mathcal{F}$ is also a wqo. Let us also restrict ourselves to terms $ca(A,T)$ constructed by using this functor exactly once such that A is an atom and T the representation of some characteristic tree. For this very special case, the embedding extension $\trianglelefteq^*_{emb}$ of $\trianglelefteq^*$ coincides with Definition 4.23 (i.e. $ca(A_1,\lceil\tau_1\rceil) \trianglelefteq^*_{emb} ca(A_2,\lceil\tau_2\rceil)$ iff $(A_1,\tau_1) \trianglelefteq^*_{ca} (A_2,\tau_2)$) and hence $\mathcal{A}, \trianglelefteq^*_{ca}$ is well-quasi-ordered. □

4.4 Global Trees and Characteristic Atoms

In this subsection, we adapt and instantiate the m-tree concept presented in [54] according to our particular needs in this paper.

Definition 4.25 A *global tree* γ_P for a program P is a (finitely branching) tree where nodes can be either *marked* or *unmarked* and each node carries a label which is a P-characteristic atom.

In other words, a node in a global tree γ_P will look like $(n, mark, CA)$, where n is the node identifier, $mark$ an indicator that can take the values m or u designating whether the node is marked or unmarked, and the P-characteristic atom CA is the node's label. Informally, a marked node corresponds to a characteristic atom which has already been treated by the partial deduction algorithm.

In the sequel, we consider a global tree γ partially ordered through the usual relationship between nodes: *ancestor_node* $>_\gamma$ *descendent_node*. Given a node $n \in \gamma$, we denote by $Anc_\gamma(n)$ the set of its γ ancestor nodes (including itself).

We now introduce the notion of a global tree being well-quasi-ordered, and subsequently prove that it provides a sufficient condition for finiteness. Let γ_P be a global tree. Then we will henceforth denote as L_{γ_P} the set of its labels. And for a given node n in a tree γ, we will refer to its label by l_n.

Definition 4.26 Let γ be a global tree. Then we define its *associated label mapping* f_γ as the injective (i.e. one-to-one) mapping $f_\gamma : (\gamma, >_\gamma) \rightarrow (L_\gamma, \trianglelefteq^*_{ca})$ such that $n \mapsto l_n$. f_γ will be called *quasi-monotonic* iff $\forall n_1, n_2\ n_1 >_\gamma n_2 \Rightarrow l_{n_1} \not\trianglelefteq^*_{ca} l_{n_2}$.

Definition 4.27 We call a global tree γ *well-quasi-ordered* if f_γ is quasi-monotonic.

Theorem 4.28 A global tree γ is finite if it is well-quasi-ordered.

Proof Assume that γ is not finite. Then it contains (König's Lemma) at least one infinite branch $n_1 >_\gamma n_2 >_\gamma \ldots$. Consider the corresponding infinite sequence of elements $l_{n_1}, l_{n_2}, \ldots \in L_\gamma, \trianglelefteq^*_{ca}$. From Proposition 4.24, we know that $L_\gamma, \trianglelefteq^*_{ca}$ is wqo and therefore, there must exist $l_{n_i}, l_{n_j}, i < j$ in the above mentioned sequence such that $l_{n_i} \trianglelefteq^*_{ca} l_{n_j}$. But this implies that f_γ is not quasi-monotonic. □

4.5 A Tree Based Algorithm

In this subsection, concluding Section 4, we present the actual refined partial deduction algorithm where global control is imposed through characteristic atoms in a global tree.

A formal description of the algorithm can be found in Figure 9. Please note that it is parametrised by an unfolding rule U, thus leaving the particulars of local control unspecified. As for Algorithm 3.27, we need the notation $chatom(A, P, U)$ (see Definition 3.26). Also, without loss of generality, we suppose the initial goal to consist of a single atom.

As in e.g. [19, 54] (but unlike Algorithm 3.27), Algorithm 4.29 does not output a specialised program, but rather a set of (characteristic) atoms from which the actual code can be generated in a straightforward way. Most of the algorithm is self-explanatory, except perhaps the **For**-loop. In $\mathcal{B}$, all the characteristic atoms are assembled, corresponding

Algorithm 4.29

Input
a normal program P and goal $\leftarrow A$
Output
a set of characteristic atoms $\mathcal{A}$
Initialisation
$\gamma := \{(1, u, (A, \tau_A))\}$
While γ contains an unmarked leaf **do**
let n be such an unmarked leaf in γ: $(n, u, (A_n, \tau_{A_n}))$
mark n
$\mathcal{B} := \{chatom(B, P, U) | B \in BA_P(A_n, \tau_{A_n})\}$
For all $CA_B \in \mathcal{B}$ **do**
remove CA_B from $\mathcal{B}$
If $\mathcal{H} = \{CA_C \in Anc_\gamma(n) | CA_C \trianglelefteq^*_{ca} CA_B\} = \emptyset$
Then add (n_B, u, CA_B) to γ as a child of n
Else If $\{CA_D \in Anc_\gamma(n) | CA_D \equiv_{ca} CA_B\} = \emptyset$
Then add $msg(\mathcal{H} \cup \{CA_B\})$ to $\mathcal{B}$
Endfor
Endwhile
$\mathcal{A} := L_\gamma$

Figure 9: Partial deduction with global trees.

to the atoms occurring in the leaves of the SLDNF-tree built (locally, of course) for A_L according to τ_{A_L}. Elements of $\mathcal{B}$ are subsequently inserted into γ as (unmarked) child nodes of L if they do not embed the label of n or any of its ancestor nodes. If one does, and it is a variant of n's label or that of an ancestor of n, then it is simply not added to γ. (Note that one can change to an instance test by simply replacing $\preceq_{ca}$ by $\equiv_{ca}$.) Finally, if a characteristic atom $CA_B \in \mathcal{B}$ does embed an ancestor label, but there is no variant to be found labelling any of the ancestor nodes, then the most specific generalisation M of CA_B and all embedded ancestor labels $\mathcal{H}$ is re-inserted into $\mathcal{B}$. The latter case is of course the most interesting: Simply adding a node labelled CA_B would violate the well-quasi ordering of the tree and thus endanger termination. Calculating the *msg* M (which always exists by the conditions of Definition 4.21) and trying to add it instead secures finiteness, as proven below, while trying to preserve as much information as seems possible (see however Sections 4.6 and 5). Note that, similarly to CA_B, we cannot add the *msg* M directly to the tree because this might produce a tree which is not well-quasi ordered (and termination would also be endangered). Indeed, although by generalising CA_B into M we are actually sure that M will not embed any characteristic atom in $\mathcal{H}$ (for a proof of this property see [45]) it might embed other characteristic atoms in the tree. Take for example a branch in a global tree which contains the characteristic atoms $(p(f(a)), \tau)$ and $(p(X), \tau)$. Then $CA_B = (p(f(f(a))), \tau)$ embeds $(p(f(a)), \tau)$ but not $(p(X), \tau)$. Thus $\mathcal{H} = \{(p(f(a)), \tau)\}$ and $msg(\{(p(f(a)), \tau), (p(f(f(a))), \tau)\}) = (p(f(X)), \tau)$, which no longer

embeds $(p(f(a)), \tau)$ but now embeds $(p(X), \tau)$! So, to ensure that the global tree remains well-quasi ordered it is important to re-check the *msg* for embeddings before adding it to the global tree.

We obtain the following theorems:

Theorem 4.30 Algorithm 4.29 always terminates.

Proof Upon each iteration of the while-loop in Algorithm 4.29, exactly one node in γ is marked, and zero or more (unmarked) nodes are added to γ. Moreover, Algorithm 4.29 never deletes a node from γ, neither does it ever "unmark" a marked node. Hence, since all branchings are finite, non-termination of the algorithm must result in the construction of an infinite branch. It is therefore sufficient to argue that after every iteration of the while-loop, γ is a wqo global tree.
First, this holds after initialisation. Also, obviously, a global tree will be converted into a new global tree through the while-loop. Now, a while-iteration adds zero or more, but finitely many, child nodes to a particular leaf n in the tree, thus creating a (finite) number of new branches that are extensions of the old branch leading to n. We prove that on all of the new branches, f_γ is quasi-monotonic. The branch extensions are actually constructed in the for-loop, (at most) one for every element of $\mathcal{B}$. So, let us take an arbitrary characteristic atom $CA_B \in \mathcal{B}$, then there are three cases to consider:

1. Either CA_B does not embed any label on the branch (up) to (and including) n. It is then added in a fresh leaf. Obviously, f_γ will be quasi-monotonic on the newly created branch.

2. Or some such label is embedded, but there is also a variant (or more general respectively, in case an instance test is used) label already in some node on the branch to n. Then, no corresponding leaf is inserted in the tree, and there is nothing left to prove.

3. Or, finally, some labels on the branch are embedded, but none are variants (or more general respectively). We then calculate the *msg* M of CA_B and all[7] the labels $\mathcal{H} = \{L_1, \ldots, L_k\}$ on the branch it embeds. In that case, M must be strictly more general than CA_B. Indeed, if M would be a variant of CA_B then CA_B must be more general than all the elements in $\mathcal{H}$ (by property of the *msg*), and even strictly more general because no ancestor was found of which it was a variant (or instance respectively). This is in contradiction with the definition of $\trianglelefteq^*$, which requires that each L_i is not a strict instance of CA_B for $L_i \trianglelefteq^* CA_B$ to hold. (More precisely, given $L_i = (A_i, \tau_i)$ and $CA_B = (A_B, \tau_B)$, $L_i \trianglelefteq^* CA_B$ implies that A_i is not a strict instance of A_B and that $\lceil \tau_i \rceil$ is not a strict instance of $\lceil \tau_B \rceil$; the latter implies, by strict monotonicity[8] of $\lceil . \rceil$ that τ_i is not a strict instance of τ_B and thus, by Definition 4.10, we have that L_i is not a strict instance of CA_B.) So in this step we have not modified the tree (and it remains wqo), but replaced an atom in $\mathcal{B}$ by a strictly more general one, which we can do only finitely many times (by Lemma 3.30) and thus termination of the for-loop is ensured.

[7] The algorithm would also terminate if we only pick one such label at every step.

[8] The proof actually also goes through if $\lceil . \rceil$ is not strictly monotonic but just satisfies that whenever τ_i is a strict instance of τ_B then $\tau_i \not\trianglelefteq^* \tau_B$.

Note that this is not true for the $\trianglelefteq$ relation, i.e. the algorithm could loop if we use it instead of $\trianglelefteq^*$. For example take a global tree having the single node $(p(X,X),\tau)$ and where we try to add $\mathcal{B} = \{(p(X,Y),\tau)\}$. Now we have that $p(X,X) \trianglelefteq p(X,Y)$ and $msg(\{(p(X,X),\tau),(p(X,Y),\tau)\}) = (p(X,Y),\tau)$ and we have a loop.

□

Theorem 4.31 Let P be a program, input to Algorithm 4.29, and $\mathcal{A}$ the corresponding set of characteristic atoms produced as output. Then $\mathcal{A}$ is P-covered.

Proof First, it is straightforward to prove that throughout the execution of Algorithm 4.29, any unmarked node in γ must be a leaf. It therefore suffices to show that after each while-loop iteration, only unmarked leaves in γ possibly carry a non-covered label.[9] Trivially, this property holds after initialisation. Now, in the while-loop, one unmarked leaf n is selected and marked. The for-loop then precisely proceeds to incorporate (unmarked leaf) nodes into γ such that all body atoms of n's label are concretisations of at least one label in the new, extended γ. □

The correctness of the specialisation now follows from Theorem 3.16 presented earlier.

4.6 Further Improvements

Unlike ecological partial deduction as presented in Section 3.4, Algorithm 4.29 will obviously often output several characteristic atoms with the same characteristic tree (each giving rise to a different specialised version of the same original predicate definition). Such "duplicated" polyvariance is however superfluous (in the context of simply running the resulting program) when it increases neither local nor global precision. As far as preserving local precision is concerned, matters are simple: One procedure per characteristic tree is what you want. The case of global precision is slightly more complicated: Generalising atoms with identical characteristic trees might lead to the occurrence of more general atoms in the leaves of the associated local tree. In other words, we might loose subsequent instantiation at the global level, possibly leading to a different and less precise set of characteristic atoms.

The polyvariance reducing post-processing that we propose in this section therefore avoids the latter phenomenon. In order to obtain the desired effect, it basically collapses and generalises several characteristic atoms with the same characteristic tree only if this does not modify the global specialisation. To that end we number the body atoms of each characteristic atom and then label the arcs of the global tree with the number of the body atom it refers to. We then try to collapse nodes with identical characteristic trees using the well-known algorithm for minimisation of finite state automata [1, 29]: we start by putting all characteristic atoms with the same characteristic tree into the same class, and subsequently split these classes if corresponding body atoms fall into different classes. As stated in [29], the complexity of this algorithm is $O(kn^2)$ where n is the maximum number of states (in our case the number of characteristic atoms) and k the number of symbols (in our case the maximum number of body atoms).

The following example illustrates the use of this minimisation algorithm for removing superfluous polyvariance.

[9] I.e. a label with at least one body atom (in P) that is not a concretisation of any label in γ.

Example 4.32 Let us return to the *member* program of Example 2.9:

(1) $member(X, [X|T]) \leftarrow$
(2) $member(X, [Y|T]) \leftarrow member(X, T)$
(3) $t(T) \leftarrow member(a, [a, b, c, d|T]), member(b, T)$

Suppose that after executing Algorithm 4.29 we obtain the following set of characteristic atoms $\mathcal{A} = \{(member(b, L), \tau), (member(a, [a, b, c, d|T]), \tau), (member(a, [b, c, d|T]), \tau'), (member(a, L), \tau), (t(T), \{\langle 1 \circ 3 \rangle\})\}$ where $\tau = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2 \rangle\}$ and $\tau' = \{\langle 1 \circ 2, 1 \circ 2, 1 \circ 2 \rangle\}$. Depending on the particular renaming function, a partial deduction of P wrt $\mathcal{A}$ will look like:

$mem_{a,[a,b,c,d]}(T) \leftarrow$
$mem_{a,[a,b,c,d]}(T) \leftarrow mem_{a,[b,c,d]}(T)$
$mem_{a,[b,c,d]}(T) \leftarrow mem_a(T)$
$mem_a([a|T]) \leftarrow$
$mem_a([Y|T]) \leftarrow mem_a(T)$
$mem_b([b|T]) \leftarrow$
$mem_b([Y|T]) \leftarrow mem_b(T)$
$t(T) \leftarrow mem_{a,[a,b,c,d]}(T), mem_b(T)$

The labelled version of the corresponding global tree can be found in Figure 10. Adapting the algorithm from [1, 29] to our needs, we start out by generating three classes (one for each characteristic tree) of states:

- $C_1 = \{(member(a, [a, b, c, d|T]), \tau), (member(a, L), \tau), (member(b, L), \tau)\}$,
- $C_2 = \{(member(a, [b, c, d|T]), \tau')\}$ and
- $C_3 = \{(t(T), \{\langle 1 \circ 3 \rangle\})\}$.

The class C_1 must be further split, as the state $(member(a, [a, b, c, d|T]), \tau)$ has a transition via the label #1 to a state of C_2 while the other states of C_1, $(member(a, L), \tau)$ and $(member(b, L), \tau)$, do not. This means that by actually collapsing all elements of C_1 we would lose subsequent specialisation at the global level (namely the pruning and pre-computation that is performed within $(member(a, [b, c, d|T]), \tau')$).
We now obtain the following 4 classes:

- $C_2 = \{(member(a, [b, c, d|T]), \tau')\}$,
- $C_3 = \{(t(T), \{\langle 1 \circ 3 \rangle\})\}$,
- $C_4 = \{(member(a, [a, b, c, d|T]), \tau)\}$ and
- $C_5 = \{(member(a, L), \tau), (member(b, L), \tau)\}$.

The only class that might be further split is C_5, but both states of C_5 have transitions with identical labels leading to the same classes, i.e. no specialisation will be lost by collapsing the class. We can thus generate one characteristic atom per class (by taking the *msg* of the atom parts and keeping the characteristic tree component). The resulting, minimised program is thus:

$mem_{a,[a,b,c,d]}(T) \leftarrow$
$mem_{a,[a,b,c,d]}(T) \leftarrow mem_{a,[b,c,d]}(T)$
$mem_{a,[b,c,d]}(T) \leftarrow mem_x(a, T)$
$mem_x(X, [X|T]) \leftarrow$
$mem_x(X, [Y|T]) \leftarrow mem_x(X, T)$
$t(T) \leftarrow mem_{a,[a,b,c,d]}(T), mem_x(b, T)$

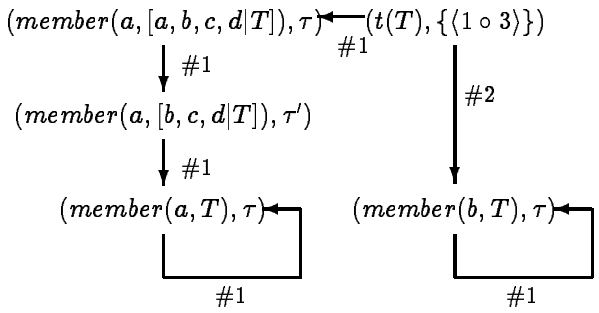


Figure 10: Labelled global tree of Example 4.32 before post-processing

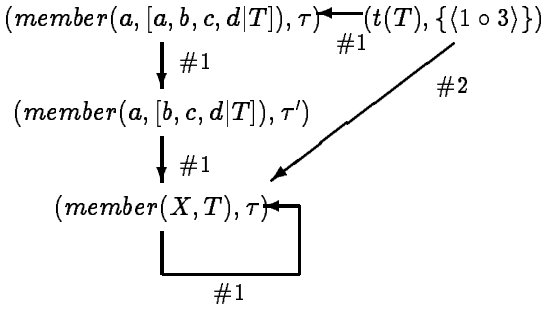


Figure 11: Labelled global tree of Example 4.32 after post-processing

A similar use of this minimisation algorithm is made in [70] and [62]. The former aims at minimising polyvariance in the context of multiple specialisation by optimising compilers. The latter, in a somewhat different context, studies polyvariant parallelisation and specialisation of logic programs based on abstract interpretation.

Upon close inspection, Algorithm 4.29 itself might be made more precise by choosing another, less general, label to insert upon generalisation. As it currently stands, the algorithm computes the *msg* of the characteristic atom to be added and the labels of all its embedded (future) ancestors. Now, there may be less general generalisations of (B, τ_B) that likewise do not embed any ancestor label, and can therefore be added safely and correctly instead of (B, τ_B). For example, given $\mathcal{H} = \{(p(a, b), \tau_B)\}$ and $B = p(f(a), f(b))$, the atom part of $msg(\mathcal{H} \cup \{(B, \tau_B)\})$ is $p(X, Y)$. However, both $p(f(X), f(b))$ and $p(f(a), f(Y))$ are more specific generalisations of B that nevertheless do not embed $p(a, b)$. A similar consideration applies to the case where no node is added since some ancestor carries a label, more general than (B, τ_B). To what extent such refinements would significantly influence the specialisation potential, and, if so, whether it is actually possible to incorporate them into a concrete algorithm, are topics for future research.

One further possibility for improvement lies in refining the ordering relation $\preceq_{ca}$ on characteristic atoms and the related *msg* operator, so that they more precisely capture the intuitive, but uncomputable order based on the set of concretisations. One could also try to use an altogether more accurate abstraction operator than taking an *msg* on

characteristic atoms. For instance, one can endeavour to extend the constraint based abstraction operator proposed in [41] to normal programs and arbitrary unfolding rules. This would probably result in a yet (slightly) more precise abstraction, causing a yet smaller global precision loss. Finally, one might also try to incorporate more detailed efficiency and cost estimations into the global control, e.g. based on [14, 15], in order to analyse the trade-off between improved specialisation and increased polyvariance and code size.

5 Experimental Results and Discussion

5.1 Systems

In this section we present an implementation of the ideas of the preceding sections, as well as an extensive set of experiments which highlight the practical benefits of the implementation over existing partial deduction systems.

The system which integrates the ideas of this paper, called ECCE, is publicly available in [40] and is actually an implementation of a generic version of Algorithm 4.29 which allows the advanced user to change and even implement e.g. the unfolding rule as well as the abstraction operator and non-termination detection method. For instance, by adapting the settings of ECCE, one can obtain exactly Algorithm 4.29. But one can also simulate a (global tree oriented) version of Algorithm 3.27 using depth bounds to ensure termination.

All unfolding rules of ECCE were complemented by simple *more specific resolution* steps in the style of SP [18]. Constructive negation (see [7],[26]) has not yet been incorporated, but the selection of ground negative literals is allowed. Post-processing removal of unnecessary polyvariance, using the algorithm outlined in Section 4.6, determinate post-unfolding as well as redundant argument filtering (see [47]) were enabled throughout the experiments discussed below.

The ECCE system also handles a lot of Prolog built-ins, like for instance =, is, <, =<, <, >=, *nonvar*, *ground*, *number*, *atomic*, *call*, \==, \=. All built-ins are supposed to be declarative (e.g. the built-in *ground* is supposed to be delayed until its argument is ground). The method presented in this paper is extended by also registering built-ins in the characteristic trees. The only problematic aspect is that, when generalising built-ins which generate bindings (like is/2 or =../2 but unlike >/2 or </2) and which are no longer executable after generalisation, these built-ins have to be removed from the generalised characteristic tree (i.e. they are no longer selected). With that, the concretisation definition for characteristic atoms scales up and the technique will ensure correct specialisation. It should also be possible to incorporate the `if-then-else` into characteristic trees (and then use a specialisation technique similar to [38]).

Also, the embedding relation $\trianglelefteq$ of Definition 4.15 (and the relations $\trianglelefteq^*$ and $\trianglelefteq^*_{ca}$ based on it) has to be adapted. Indeed, some built-ins (like = ../2 or *is*/2) can be used to dynamically construct infinitely many new constants and functors and thus $\trianglelefteq$ is no longer a wqo. To remedy this, the constants and functors are partitioned into the *static* ones, occurring in the original program and the partial deduction query, and the *dynamic* ones. (This approach is also used in [63, 64].) The set of dynamic constants and functors is possibly infinite, and we will therefore treat it like the infinite set of variables in Definition 4.15 by adding the following rule to the ECCE system:

$f(s_1, \ldots, s_m) \trianglelefteq g(t_1, \ldots, t_n)$ if both f and g are dynamic

We present benchmarks using 3 different settings of ECCE, hereafter called ECCE-D, ECCE-X-10 and ECCE-X. The settings ECCE-D and ECCE-X use Algorithm 4.29, with a different unfolding rule, while ECCE-X-10 uses a (global tree oriented) version of Algorithm 3.27 with a depth bound of 10 to ensure termination. We also compare with MIXTUS[64, 63], PADDY [59] and SP [18, 19], of which the following versions have been used: MIXTUS 0.3.3, the version of PADDY delivered with ECLIPSE 3.5.1 and a version of SP dating from September 25^{th}, 1995.

Basically, the above mentioned systems use two different unfolding rules :

- *"MIXTUS-like " unfolding*: This is the unfolding strategy explained in [64, 63] which in general unfolds deeply enough to solve the "fully unfoldable" problems[10] but also has safeguards against excessive unfolding and code explosion. It requires however a number of ad hoc settings. (In the future, we plan experiments with unfolding along the lines of [52] which is free of such elements.) For instance for ECCE-X and ECCE-X-10 we used the settings (see [64]) $max\_rec = 2$, $max\_depth = 2$, $maxfinite = 7$, $maxnondeterm = 10$ and only allowed non-determinate unfolding when no user predicates were to the left of the selected literal. The method ECCE-X-10 was also complemented by a level 10 depth bound. For MIXTUS and PADDY we used the respective default settings of the systems. Note that the "MIXTUS-like" unfolding of ECCE, MIXTUS and PADDY differ slightly from each other, probably due to some details not outlined or fully explained in [64, 59] as well as the fact that the different global control regimes influence the behaviour of the "MIXTUS-like" local control.

- *Determinate unfolding*: Only (except once) select atoms that match a single clause head. The strategy is refined with a so-called "look-ahead" to detect failure at a deeper level (see e.g. [19, 18]). This approach is used by ECCE-D and SP.[11]

5.2 Experiments

The benchmark programs are taken from [40], short descriptions can be found in Appendix A. In addition to the "Lam & Kusalik" benchmarks (originally in [36]) they contain a whole set of more involved and practical examples, like e.g. a model-elimination theorem prover and a meta-interpreter for an imperative language.

For the experimentation, we tried to be as realistic as possible and measure what a normal user sees. In particular, we did not count the number of inferences, the cost of which varies a lot, or some other abstract measure, but the actual execution time and size of compiled code. The results are summarised in Table 1 while the full details can be found in Tables 2 and 3. Further details and explanations about the benchmarks are listed below:

- Transformation times (TT):
  The transformation times of ECCE and MIXTUS also include time to write to file. Time for SP does not and for PADDY we do not know. We briefly explain the use of ∞ in the tables:

[10] I.e. those problems for which normal evaluation terminates.

[11] Note however that the unfolding used by SP does not seem to fully conform to determinate unfolding (look e.g. at the results for the benchmark *depth.lam* below).

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

In Tables 2 and 3, the transformation times (TT) are expressed in seconds while the total transformation time in Table 1 is expressed in minutes (on a Sparc Classic running under Solaris, except for PADDY which for technical reasons had to be run on a Sun 4).

It seems that the latest version 0.3.6 of MIXTUS does terminate for the missionaries example, but we did not yet have time to redo the experiments. PADDY and SP did not terminate for one other example (memo-solve and imperative.power respectively) when we accidentally used *not* instead of \+ (*not* is not defined in SICStus Prolog; PADDY and SP follow this convention and interpreted *not* as an undefined, failing predicate). After changing to \+, both systems terminated.

- Runtimes (RT) of the specialised code:
  The timings are not obtained via a loop with an overhead but via special Prolog files, generated automatically by ECCE. These files call the original and specialised programs directly (i.e. without overhead), at least 100 times for the respective runtime queries. The timings were obtained via the *time*/2 predicate of Prolog by BIM 4.0.12 on a Sparc Classic under Solaris. Runtimes in Tables 2 and 3 are given relative to the runtimes of the original programs. In computing averages and totals, the time and size of the original program were taken in case of non-termination (i.e. we did not punish MIXTUS, PADDY and SP for the non-termination). The total speedups are obtained by the formula

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

  where n is the number of benchmarks and $spec_i$ and $orig_i$ are the absolute execution times of the specialised and original programs respectively. In Table 1 the column for "FU" holds the total speedup for the fully unfoldable benchmarks (see Appendix A) while the column for "Not FU" holds the total speedup for the benchmarks which are not fully unfoldable.

  All timings were for renamed queries, except for the original and for SP (which does not rename the top-level query — this puts SP at a disadvantage of about 10% in average for speed but at an advantage for code size). Note that PADDY systematically included the original program and the specialised part could only be called in a renamed style. We removed the original program whenever possible and added 1 clause which allows calling the specialised program also in an unrenamed style (just like MIXTUS and ECCE). The latter was not actually used in the benchmarks but avoids distortions in the code size figures (wrt MIXTUS and ECCE).

- Size of the specialised code:
  The compiled code size was obtained via *statistics*/4 and is expressed in units, were 1 unit = 4.08 bytes (in the current implementation of Prolog by BIM).

System	Total Speedup	Worst Speedup	FU Speedup	Not FU Speedup	Total Code Size (in KB)	Total TT (in min)
ECCE-D	1.90	0.85	2.57	1.74	166.69	2.64
ECCE-X-10	2.13	0.79	7.07	1.71	224.35	112.72
ECCE-X	2.51	0.92	8.36	2.02	136.17	2.70
MIXTUS	2.08	0.65	8.13	1.65	152.26	∞+2.49
PADDY	2.08	0.68	8.12	1.65	196.19	∞+0.28
SP	1.46	0.86	2.08	1.32	182.02	3∞+1.92

Table 1: Short summary of the results (higher speedup and lower code size is better)

5.3 Analysing the Results

The ECCE based systems did terminate on all examples, as would be expected by the results presented earlier in the paper. To our surprise however, all the existing systems, MIXTUS, PADDY and SP, did not (properly) terminate for at least one benchmark each. (Apparently, a more recent version of MIXTUS does not exhibit this non-termination, but we have not been able to verify this.) Even ignoring this fact, the system ECCE-X clearly outperforms MIXTUS, PADDY and SP, as well for speed as for code size, while having the best worst case performance. Even though ECCE is still a prototype, the transformation times are reasonable and usually close to the ones of MIXTUS. ECCE can certainly be speeded up considerably, maybe even by using the ideas in [60] which help PADDY to be (except for one glitch) the fastest system overall.

Note that even the system ECCE-X-10 based on a depth bound of 10, outperforms the existing systems for speed of the specialised programs. Its transformation times as well as the size of the specialised code are not so good. Also note that for some benchmarks the (ad-hoc) depth bound of 10 was too shallow (e.g. relative) while for others it was too deep and resulted in excessive transformation times (e.g. model-elim.app). But by removing depth bound (ECCE-X) we increase the total speedup from 2.13 to 2.51 while decreasing the size of the specialised code from 235 KB down to 142 KB. Also the total transformation time drastically decreases by a factor of 42. This clearly illustrates that getting rid of the depth bound is a very good idea in practice.

The difference between ECCE-D and ECCE-X in the resulting speedups shows that determinate unfolding, at least in the context of standard partial deduction, is in general not sufficient for fully satisfactory specialisation (see also Section 6). The "MIXTUS-like" unfolding seems to be a good compromise for standard partial deduction. Also note that for the not fully unfoldable benchmarks (which for most applications will be the case: if a benchmark is fully unfoldable one does not need a partial evaluator, normal evaluation would be sufficient) ECCE-D actually outperforms MIXTUS, PADDY and SP (but not ECCE-X).

In conclusion, the ideas presented in this paper do not only make sense in theory on accounts of precision and termination, but they also pay off in practice, resulting in a specialiser which thoroughly beats some major existing systems on accounts of speed and size of the specialised programs.

	ECCE-D			ECCE-X-10			ECCE-X		
Benchmark	RT	Size	TT	RT	Size	TT	RT	Size	TT
advisor	0.47	412	0.79	0.32	809	1.01	0.31	809	0.78
contains.kmp	0.83	1363	2.90	0.80	1974	11.53	0.09	685	4.48
depth.lam	0.94	1955	1.53	0.06	1802	2.25	0.02	2085	1.91
doubleapp	0.98	277	0.61	0.95	216	0.59	0.95	216	0.53
ex_depth	0.76	1614	2.78	0.32	350	1.73	0.32	350	1.58
grammar.lam	0.17	309	1.92	0.14	218	2.27	0.14	218	1.90
groundunify.cplx	0.40	9502	25.04	0.53	4511	29.10	0.53	4800	0.75
groundunify.smpl	0.25	368	0.78	0.25	368	0.83	0.25	368	22.03
imperative.power	0.37	2401	61.28	0.56	1578	18.14	0.54	1578	27.42
liftsolve.app	0.06	1179	5.42	0.53	4544	16.70	0.06	1179	6.57
liftsolve.db1	0.01	1280	12.95	0.02	2767	22.15	0.02	1326	7.33
liftsolve.db2	0.17	4694	14.95	0.47	11303	154.94	0.61	4786	34.25
liftsolve.lmkng	1.07	1730	1.70	1.02	2385	3.44	1.02	2385	2.75
map.reduce	0.07	507	0.84	0.08	348	0.84	0.08	348	0.86
map.rev	0.11	427	0.88	0.13	427	1.02	0.11	427	0.89
match.kmp	0.73	639	1.17	0.70	669	1.24	0.70	669	1.23
memo-solve	1.17	2318	4.22	1.26	2033	10.56	1.09	2308	4.31
missionaries	0.81	2294	4.31	1.03	2927	26.67	0.72	2226	9.21
model_elim.app	0.63	2100	2.83	0.42	6092	5864.28	0.13	532	3.56
regexp.r1	0.50	594	1.29	0.29	435	6.77	0.29	435	0.98
regexp.r2	0.55	629	1.29	0.43	1373	8.63	0.51	1159	4.87
regexp.r3	0.50	828	1.74	0.48	2041	10.82	0.42	1684	14.92
relative.lam	0.82	1074	1.92	0.02	709	506.89	0.00	261	4.06
rev_acc_type	1.00	242	0.70	0.99	2188	22.95	1.00	242	0.83
rev_acc_type.inffail	0.60	527	0.71	0.55	1503	21.47	0.60	527	0.80
ssuply.lam	0.06	262	1.18	0.14	426	7.84	0.06	262	1.18
transpose.lam	0.17	2312	2.56	0.18	2312	8.75	0.17	2312	1.98
Average	0.40	1550	5.86	0.44	2085	250.50	0.30	1266	6.00
Total	14.19	41837	158.29	12.67	56308	6763.41	10.75	34177	161.96
Speedup	**1.90**			**2.13**			**2.51**		

Table 2: ECCE Standard Partial Deduction Methods

5.4 Further Discussion

Returning to the global control method as laid out in Section 4, one can observe that a possible drawback might be its considerable complexity. Indeed, first, ensuring termination through a well-quasi-ordering is structurally much more costly than the alternative of using a well-founded ordering. The latter only requires comparison with a single "ancestor" object and can be enforced without any search through "ancestor lists" (see [52]). Testing for well-quasi-ordering, however, unavoidably does entail such searching and repeated comparisons with several ancestors. Moreover, in our particular case, checking $\trianglelefteq^*_{ca}$ on characteristic atoms is in itself a quite costly operation, adding to the innate complexity of maintaining a well-quasi-ordering. But as the experiments of the previous subsection show, in practice, the complexity of the transformation does not seem to be all that bad, especially since the experiments were still conducted with a prototype which was not yet tuned for transformation speed.

As already mentioned earlier, the partial deduction method of [18] was extended in [11] by adorning characteristic trees with a depth-k abstraction of the corresponding

	MIXTUS			PADDY			SP		
Benchmark	RT	Size	TT	RT	Size	TT	RT	Size	TT
advisor	0.31	809	0.85	0.31	809	0.10	0.40	463	0.29
contains.kmp	0.16	533	2.48	0.11	651	0.55	0.75	985	1.13
depth.lam	0.04	1881	4.15	0.02	2085	0.32	0.53	928	0.99
doubleapp	1.00	295	0.30	0.98	191	0.08	1.02	160	0.11
ex_depth	0.40	643	2.40	0.29	1872	0.53	0.27	786	1.35
grammar.lam	0.17	841	2.73	0.43	636	0.22	0.15	280	0.71
groundunify.cplx	0.67	5227	11.68	0.60	4420	1.53	0.73	4050	2.46
groundunify.smpl	0.25	368	0.45	0.25	368	0.13	0.61	407	0.20
imperative.power	0.57	2842	5.35	0.58	3161	2.18	0.97	1706	6.97
liftsolve.app	0.06	1179	4.78	0.06	1454	0.80	0.23	1577	2.46
liftsolve.db1	0.01	1280	5.36	0.02	1280	1.20	0.82	4022	3.95
liftsolve.db2	0.31	8149	58.19	0.32	4543	1.60	0.82	3586	3.71
liftsolve.lmkng	1.16	2169	4.89	0.98	1967	0.32	1.16	1106	0.37
map.reduce	0.68	897	0.17	0.08	498	0.20	0.09	437	0.23
map.rev	0.11	897	0.16	0.26	2026	0.37	0.13	351	0.20
match.kmp	1.55	467	4.89	0.69	675	0.28	1.08	527	0.49
memo-solve	0.60	1493	12.72	1.48	3716	1.70	1.15	1688	3.65
missionaries	-	-	∞	-	-	∞	0.73	16864	82.59
model_elim.app	0.13	624	5.73	0.10	931	0.90	-	-	∞
regexp.r1	0.20	457	0.73	0.29	417	0.13	0.54	466	0.37
regexp.r2	0.82	1916	2.85	0.67	3605	0.63	1.08	1233	0.67
regexp.r3	0.60	2393	4.49	1.26	10399	1.35	1.03	1646	1.20
relative.lam	0.01	517	7.76	0.00	517	0.42	0.69	917	0.35
rev_acc_type	1.00	497	0.99	0.99	974	0.33	-	-	∞
rev_acc_type.inffail	0.97	276	0.77	0.94	480	0.28	-	-	∞
ssuply.lam	0.06	262	0.93	0.08	262	0.08	0.06	231	0.52
transpose.lam	0.18	1302	3.89	0.18	1302	0.43	0.26	1267	0.52
Average	0.35	1470	5.76	0.35	1894	0.64	0.50	1903	4.81
Total	13.00	38214	149.7	12.96	49239	16.7	18.48	45683	115.5
Speedup	**2.08**			**2.08**			**1.46**		

Table 3: Some Existing Systems

atom component. This was done in order to increase the amount of polyvariance so that a (mono-variant) post-processing abstract interpretation phase (detecting useless clauses) can obtain a more precise result. However, this k parameter is of course yet another ad hoc depth bound. Our method is free of any such bounds and, without the Section 4.6 post-processing, nevertheless obtains a similar effect.

The above corresponds to adding extra information to characteristic trees. Sometimes, however, characteristic trees carry too much information, in the sense that *different* characteristic trees can represent the *same* local specialisation behaviour. Indeed characteristic trees also encode the particular order in which literals are selected and thus do not take advantage of the independence of the computation rule. A quite simple solution to this problem exists: after having performed the local unfolding we just have to *normalise* the characteristic trees by imposing a fixed (e.g. left-to-right) computation rule and delaying the selection of all negative literals to the end. The results and discussions of this paper remain valid independently of whether this normalisation is applied or not. A similar effect can be obtained, in the context of definite programs, via the *trace terms* of [23].

Algorithm 4.29 can also be seen as performing an abstract interpretation on an *infinite domain* of *infinite height* (i.e. the ascending chain condition of [9] is not satisfied) and without a priori limitation of the precision (i.e., if possible, we do not perform any abstraction at all and obtain simply the concrete results). Very few abstract interpretations of logic programs use infinite domains of infinite height (some notable exceptions are [5, 30, 27]) and to our knowledge all of them have some a priori limitation of the precision, at least in practice. An adaptation of Algorithm 4.29, with its non ad hoc termination and precise generalisations, might provide a good starting point to introduce similar features into abstract interpretation methods, where they might prove equally beneficial.

Some additional discussions, further motivating the use of global trees and characteristic atoms, can be found in Appendix D.

We conclude this section with a brief discussion on the relation between our global control and what may be termed thus in supercompilation [67, 68, 65]. (A distinction between local and global control is not yet made in supercompilation, but we feel that this situation is likely to change in the near future.) We already pointed out that the inspiration for using $\trianglelefteq$ derives from [50] and [65]. In the latter, a generalisation strategy for positive supercompilation (no negative information propagation while driving) is proposed. It uses $\trianglelefteq$ to compare nodes in a marked partial process tree (a notion roughly corresponding to marked or global trees in partial deduction). These nodes, however, only contain syntactical information corresponding to ordinary atoms (or rather goals, see Section 6). It is our current understanding that both the addition of something similar to characteristic trees and the use of the refined $\trianglelefteq^*$ embedding can lead to improvements of the method proposed in [65]. Finally, it is interesting to return to an observation already made in Section 4 of [54]: Neighbourhoods of order "n", forming the basis for generalisation in (full) supercompilation [68], are essentially the same as classes of atoms (or goals) with an identical depth n characteristic tree. Adapting our technique to the supercompilation setting would therefore probably allow to remove the depth bound on neighbourhoods.

6 Conclusion and Future Work

In the first part of this paper we have presented a new framework and a new algorithm for partial deduction. The framework and the algorithm can handle *normal* logic programs and place *no* restrictions on the unfolding rule. We provided general correctness results for the framework as well as correctness and termination proofs for the algorithm. Also, the abstraction operator of the algorithm preserves the characteristic trees of the atoms to be specialised and ensures termination (when the number of distinct characteristic trees is bounded) while providing a fine grained control of polyvariance. All this is achieved without using constraints in the partial deduction process.

In the second part of the paper, we have first identified the problems of imposing a depth bound on characteristic trees (or neighbourhoods for that matter) using some practical and realistic examples. We have then developed an even more sophisticated online global control technique for partial deduction of normal logic programs. Importing and adapting m-trees from [54], we have overcome the need for a depth bound on characteristic trees to guarantee termination of partial deduction. Plugging in a depth bound

free local control strategy (see e.g. [6, 52]), we thus obtain a fully automatic, concrete partial deduction method that always terminates and produces precise and reasonable polyvariance, without resorting to any ad hoc techniques. To the best of our knowledge, this is the very first such method.

Along the way, we have defined generalisation and embedding on characteristic atoms, refining the homeomorphic embedding relation $\trianglelefteq$ from [16, 17, 50, 65] into $\trianglelefteq^*$, and showing that the latter is more suitable in a logic programming setting. We have also touched upon a post-processing intended to sift superfluous polyvariance, possibly produced by the main algorithm. Extensive experiments with an implementation of the method showed its practical value; outperforming existing partial deduction system for speedup as well as code size while guaranteing termination.

We believe that the global control proposed in this paper is a very good one, but the quality of the specialisation produced by any fully concrete instance of Algorithm 4.29 will obviously also heavily depend on the quality of the specific local control used. At the local control level, a number of issues are still open: fully automatic satisfactory unfolding of meta-interpreters and a good treatment of truly non-determinate programs are among the most pressing.

Recent work brought a closer integration of abstract interpretation and partial deduction [46], as well as an extension of partial deduction [43, 25] to incorporate more powerful unfold/fold-like transformations [57], allowing for example to eliminate unnecessary variables from programs [61]. The latter extension boils down to the lifting of entire goals (instead of separate atoms) to the global level, as for instance in supercompilation (where non-atomic goals translate into nested function calls). This opens up a whole range of challenging new control issues. It turned out that the global control techniques presented in the current paper can be extended and that they significantly contribute in that context too (see [43, 25, 34]).

Acknowledgements

We would like to thank John Gallagher and Maurice Bruynooghe for interesting remarks and for pointing out several improvements. We also thank the following persons for interesting comments and/or stimulating discussions on this work: Eddy Bevers, Bart Demoen, André de Waal, Robert Glück, Manuel Hermenegildo, Neil Jones, Jesper Jørgensen, Laura Lafave, Dominique Meulemans, Torben Mogensen, Kristian Nielsen, Alberto Pettorossi, Maurizio Proietti, Dan Sahlin, David Sands, Morten Heine Sørensen, Valentin Turchin, Wim Vanhoof and anonymous referees of LOPSTR'95 and of the 1996 Dagstuhl Seminar on Partial Evaluation. Finally, Alain Callebaut provided helpful information on benchmarking Prolog by BIM programs.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] 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.

[3] K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 343–358. MIT Press, 1990.

[4] R. Bol. Loop checking in partial deduction. *The Journal of Logic Programming*, 16(1&2):25–46, 1993.

[5] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *The Journal of Logic Programming*, 10:91–124, 1991.

[6] 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.

[7] D. Chan and M. Wallace. A treatment of negation during partial evaluation. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming, Proceedings of the Meta88 Workshop, June 1988*, pages 299–318. MIT Press, 1989.

[8] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of POPL'93*, Charleston, South Carolina, January 1993. ACM Press.

[9] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2 & 3):103–179, 1992.

[10] D. De Schreye, M. Leuschel, and B. Martens. Tutorial on program specialisation (abstract). In J. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, Portland, USA, December 1995. MIT Press.

[11] D. A. de Waal. *Analysis and Transformation of Proof Procedures.* PhD thesis, University of Bristol, October 1994.

[12] 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.

[13] 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.

[14] S. Debray and N.-W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[15] S. Debray, P. López García, M. Hermenegildo, and N.-W. Lin. Estimating the computational cost of logic programs. In B. Le Charlier, editor, *Proceedings of SAS'94*, LNCS 864, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[16] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[17] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.

[18] J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, University of Bristol, November 1991.

[19] 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.

[20] J. Gallagher and M. Bruynooghe. Some low-level transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of Meta90 Workshop on Meta Programming in Logic*, pages 229–244, Leuven, Belgium, 1990.

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

[22] 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.

[23] J. Gallagher and L. Lafave. Regular approximations of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 115–136, Schloß Dagstuhl, 1996.

[24] R. Glück. On the generation of specialisers. *Journal of Functional Programming*, 4(4):499–514, 1994.

[25] R. Glück, J. Jørgensen, B. Martens, and M. 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. Extended version as Technical Report CW 226, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[26] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel.* PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[27] N. Heintze. Practical aspects of set based analysis. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 765–779, Washington D.C., 1992. MIT Press.

[28] P. Hill and J. Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, 1994. To be published in *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5.* Oxford Science Publications, Oxford University Press.

[29] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[30] 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.

[31] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[32] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[33] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 238–262, Schloß Dagstuhl, 1996. Extended version as Technical Report CW 221, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[34] J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallager, editor, *Pre-Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, pages 46–62, Stockholm, Sweden, August 1996. Also in the Proceedings of BENELOG'96. Extended version as Technical Report CW 242, K.U. Leuven.

[35] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, Lecture Notes in Computer Science 649, pages 49–69. Springer-Verlag, 1992.

[36] 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.

[37] J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan-Kaufmann, 1988.

[38] M. Leuschel. Partial evaluation of the "real thing". In L. Fribourg and F. Turini, editors, Logic Program Synthesis and Transformation — Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, Lecture Notes in Computer Science 883, pages 122–137, Pisa, Italy, June 1994. Springer-Verlag.

[39] M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, Lecture Notes in Computer Science 1048, pages 1–16, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[40] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Obtainable via `http://www.cs.kuleuven.ac.be/~lpai`, 1996.

[41] M. Leuschel and D. De Schreye. An almost perfect abstraction operation for partial deduction using characteristic trees. Technical Report CW 215, Departement Computerwetenschappen, K.U. Leuven, Belgium, October 1995. Submitted for Publication. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[42] 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, La Jolla, California, June 1995. ACM Press.

[43] 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. Extended version as Technical Report CW 225, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[44] 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. Extended version as Technical Report CW 210, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[45] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, LNCS 1110, pages 263–283, Schloß Dagstuhl, 1996. Extended version as Technical Report CW 220, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[46] M. Leuschel and D. 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. Extended version as Technical Report CW 232, K.U. Leuven. Accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[47] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallager, editor, *Pre-Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, pages 63–77, Stockholm, Sweden, August 1996. Extended version as Technical Report CW 243, K.U. Leuven.

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

[49] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3& 4):217–242, 1991.

[50] R. Marlet. *Vers une Formalisation de l'Évaluation Partielle.* PhD thesis, Université de Nice - Sophia Antipolis, December 1994.

[51] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming.* PhD thesis, Departement Computerwetenschappen, K.U.Leuven, Belgium, February 1994.

[52] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *The Journal of Logic Programming*, 28(2):89–146, August 1996. Abridged and

revised version of Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, October 1993, accessible via `http://www.cs.kuleuven.ac.be/~lpai`.

[53] B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1–2):97–117, 1994.

[54] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–613, Kanagawa, Japan, June 1995. MIT Press. Extended version as Technical Report CSTR-94-16, University of Bristol.

[55] D. Meulemans. Partiële deductie: Een substantiële vergelijkende studie. Master's thesis, Departement Computerwetenschappen, K.U. Leuven, Belgium, 1995.

[56] T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 214–227. Springer-Verlag, 1992.

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

[58] D. Poole and R. Goebel. Gracefully adding negation and disjunction to Prolog. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 635–641. Springer Verlag, 1986.

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

[60] S. Prestwich. Online partial deduction of large programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 111–118. ACM Press, 1993.

[61] 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 the 3rd International Symposium on Programming Language Implementation and Logic Programming, PLILP'91*, LNCS 528, Springer Verlag, pages 347–358, 1991.

[62] G. Puebla and M. Hermenegildo. Implementation of multiple specialization in logic programs. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 77–87, La Jolla, California, June 1995. ACM Press.

[63] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, Swedish Institute of Computer Science, March 1991.

[64] D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.

[65] M. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. Lloyd, editor, *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479, Portland, USA, December 1995. MIT Press.

[66] J. Stillman. *Computational Problems in Equational Theorem Proving.* PhD thesis, State University of New York at Albany, 1988.

[67] V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[68] V. F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.

[69] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures*, LNCS 523, pages 165–191, Harvard University, 1991. Springer Verlag.

[70] W. Winsborough. Multiple specialization using minimal-function graph semantics. *The Journal of Logic Programming*, 13(2 & 3):259–290, 1992.

A Benchmark Programs

The benchmark programs were carefully selected and/or designed in such a way that they cover a wide range of different application areas, including: pattern matching, databases, expert systems, meta-interpreters (non-ground vanilla, mixed, ground), and more involved particular ones: a model-elimination theorem prover, the missionaries-cannibals problem, a meta-interpreter for a simple imperative language. The benchmarks marked with a star (*) can be fully unfolded. Full descriptions can be found in [40].

Benchmark	Description
advisor*	A very simple expert system - benchmark by Thomas Horvath.
contains.kmp	A benchmark based on the "contains" Lam & Kusalik benchmark [36] but with more sophisticated run-time queries.
depth.lam*	A Lam & Kusalik benchmark [36].
ex_depth	A variation of *depth.lam* with a more sophisticated object program.
grammar.lam	A Lam & Kusalik benchmark [36].
groundunify.complex	Specialise a ground unification algorithm calculating explicit substitutions (see e.g. [12, 44]).
groundunify.simple*	Specialise the same unification algorithm for a simpler query.
imperative.power	A solver for a simple imperative language. Specialise a power sub-program for a known power and base but unknown environment
liftsolve.app	The lifting meta-interpreter (see e.g. [18, 42, 28]) for the ground representation with append as object program.
liftsolve.db1*	The lifting meta-interpreter with a simple, fully unfoldable object program.
liftsolve.db2	The lifting meta-interpreter with a partially specified object program.
liftsolve.lmkng	Testing part of lifting meta-interpreter (generates an ∞ number of chtrees).
map.reduce	Specialising the higher-order map/3 (using call and =..) for the higher-order reduce/4 in turn applied to add/3.
map.rev	Specialising the higher-order map for the reverse program.
match.kmp	Try to obtain a KMP matcher. A benchmark based on the "match" Lam & Kusalik benchmark [36] but with more sophisticated run-time queries.
memo-solve	A variation of ex_depth with a simple loop prevention mechanism based on keeping a call stack.
missionaries	A program for the missionaries and cannibals problem.
model_elim.app	Specialise the Poole-Goebel [58] model elimination prover (also used by De Waal-Gallagher [13]) for the append program.
regexp.r1	A naive regular expression matcher. Regular expression: (a+b)*aab.
regexp.r2	Same program as regexp.r1 for ((a+b)(c+d)(e+f)(g+h))*.
regexp.r3	Same program as regexp.r1 for ((a+b)(a+b)(a+b)(a+b)(a+b)(a+b))*.
relative.lam*	A Lam & Kusalik benchmark [36].
rev_acc_type	A simple benchmark generating an infinite number of different characteristic trees.
rev_acc_type.inffail	A simple benchmark with infinite determinate failure at pe time.
ssuply.lam*	A Lam & Kusalik benchmark [36].
transpose.lam*	A Lam & Kusalik benchmark [36].

Table 4: Description of the benchmark programs

B Proof of Theorem 4.20

The following definitions and lemmas are needed to prove Theorem 4.20.

Definition B.1 An order $>_S$ is called a *well-founded order (wfo)* on S iff there is no infinite sequence of elements $s_1, s_2, \ldots$ in S such that $s_i > s_{i+1}$, for all $i \geq 1$.

Lemma B.2 Let $<_V$ be a well-founded order on V. Then $\preceq_V$, defined by $v_1 \preceq_V v_2$ iff $v_1 \not>_V v_2$, is a wqo on V.

Proof Suppose that there is an infinite sequence $v_1, v_2, \ldots$ of elements of V such that for all $i < j$ $v_i \not\preceq_V v_j$. By definition this means that for all $i < j$ $v_i >_V v_j$. In particular this means that we have an infinite sequence with $v_i >_V v_{i+1}$, for all $i \geq 1$. We thus have a contradiction with Definition B.1 of a well-founded order and $\preceq_V$ must be a wqo on V. □

Lemma B.3 Let $\preceq_V$ be a wqo on V and let $\sigma = v_1, v_2, \ldots$ be an infinite sequence of elements of V.

1. There exists an $i > 0$ such that the set $\{v_j \mid i < j \wedge v_i \preceq_V v_j\}$ is infinite.
2. There exists an infinite subsequence $\sigma^* = v_1^*, v_2^*, \ldots$ of σ such that for all $i < j$ we have $v_i^* \preceq_V v_j^*$.

Proof The proof will make use of the axiom of choice at several places. Given a sequence ρ, we denote by $\rho_{\preceq_V, v}$ the subsequence of ρ consisting of all elements v' which satisfy $v \preceq_V v'$. Similarly, we denote by $\rho_{\not\preceq_V, v}$ the subsequence of ρ consisting of all elements v' which satisfy $v \not\preceq_V v'$.
Let us now prove point 1. Assume that such an i does not exist. We can then construct the following infinite sequence $\sigma_0, \sigma_1, \ldots$ of sequences inductively as follows:

- $\sigma^0 = \sigma$
- if $\sigma^i = v_i'.\rho^i$ then $\sigma^{i+1} = \rho^i_{\not\preceq_V, v_i'}$

All σ_i are indeed properly defined because at each step only a finite number of elements are removed (by going from ρ^i to $\rho^i_{\not\preceq_V, v_i'}$; otherwise we would have found an index i satisfying point 1). Now the infinite sequence $v_1', v_2', \ldots$ has by construction the property that for $i < j$ $v_i' \not\preceq_V v_j'$. Hence $\preceq_V$ cannot be a wqo on V and we have a contradiction.
We can now prove point 2. Let us construct $\sigma^* = v_1^*, v_2^*, \ldots$ inductively as follows.

- $\sigma^0 = \sigma$
- $v_{i+1}^* = r_k$ and $\sigma^{i+1} = \rho^i_{\preceq_V, r_k}$ where k is the first index satisfying the requirements of point 1 for the sequence $\sigma^i = r_1, r_2, \ldots$ (i.e. $\{r_j \mid k < j \wedge r_k \preceq_V r_j\}$ is infinite) and where $\rho^i = r_{k+1}, r_{k+2}, \ldots$.

By point 1 we know that each $\rho^i_{\preceq_V, r_k}$ is infinite and σ^* is thus an infinite sequence which by construction satisfies that for all $i < j$ we have $v_i^* \preceq_V v_j^*$. □

Lemma B.4 Let $\preceq_V^1$ and $\preceq_V^2$ be wqo's on V. Then the quasi order $\preceq_V$ defined by $v_1 \preceq_V v_2$ iff $v_1 \preceq_V^1 v_2$ and $v_1 \preceq_V^2 v_2$, is also a wqo on V.

Proof Let σ be any infinite sequence of elements from V. We can apply point 2 of Lemma B.3 to obtain the infinite subsequence $\sigma^* = v_1^*, v_2^*, \ldots$ of σ such that for all $i < j$ we have $v_i^* \preceq_V^1 v_j^*$. Now, as $\preceq_V^2$ is also a wqo we know that for some $i < j$ $v_i^* \preceq_V^2 v_j^*$ holds as well. Hence for these particular indices $v_i^* \preceq_V v_j^*$ and $\preceq_V$ satisfies the requirements of a wqo on V. □

We can now actually prove Theorem 4.20.

Proof of Theorem 4.20. $\trianglelefteq^*$ can be expressed as a combination of two quasi orders on expressions: $\trianglelefteq$ and $\preceq_{NotStrictInst}$ where $A \preceq_{NotStrictInst} B$ iff $B \not\prec A$ (i.e. B is not strictly more general than A or equivalently A is not a strict instance of B). By Lemma 3.30 we know that $\prec$ is a well-founded order on expressions. Hence by Lemma B.2 $\preceq_{NotStrictInst}$ is a wqo on expressions. By Proposition 4.17 we also have that $\trianglelefteq$ is a wqo on expression (given a finite underlying alphabet). Hence we can apply Lemma B.4 to deduce that $\trianglelefteq^*$ is also a wqo on expressions over a finite alphabet. □

C Proof of Proposition 4.22

The following notations will be useful in defining a representation $\lceil . \rceil$ satisfying Definition 4.21: $\tau \downarrow \delta = \{\gamma \mid \delta\gamma \in \tau\}$ and $top(\tau) = \{(l \circ m) \mid (l \circ m) \in prefix(\tau)\}$. Recall that $prefix(\tau) = \{\delta \mid \exists\gamma \text{ such that } \delta\gamma \in \tau\}$. Note that, for a non-empty characteristic tree τ, $\Leftrightarrow \tau = \{\langle\rangle\}$. The following lemma will also prove useful.

Lemma C.1 Let τ_1 and τ_2 be two characteristic trees and let $\delta \in prefix(\tau_1)$ be a characteristic path. If $\tau_1 \preceq_\tau \tau_2$ then $\tau_1 \downarrow \delta \preceq_\tau \tau_2 \downarrow \delta$.

Proof If $\delta' \in \tau_1 \downarrow \delta$ then by definition $\delta\delta' \in \tau_1$. Therefore $\delta\delta' \in prefix(\tau_2)$ because $\tau_1 \preceq_\tau \tau_2$. Thus $\delta' \in prefix(\tau_2 \downarrow \delta)$ and point 1 of Definition 4.4 is verified for $\tau_1 \downarrow \delta$ and $\tau_2 \downarrow \delta$.

Secondly, if $\delta' \in \tau_2 \downarrow \delta$ then $\delta\delta' \in \tau_2$ and we have $\exists\hat{\delta} \in prefix(\{\delta\delta'\})$ such that $\hat{\delta} \in \tau_1$. Now, because $\delta \in prefix(\tau_1)$ we know that $\hat{\delta}$ must have the form $\hat{\delta} = \delta\gamma$ (otherwise we arrive at a contradiction with Lemma 3.20) where $\gamma \in prefix(\{\delta'\})$. By definition we have $\gamma \in \tau_2 \downarrow \delta$ and also point 2 of Definition 4.4 is verified for $\tau_1 \downarrow \delta$ and $\tau_2 \downarrow \delta$. □

Proof of Proposition 4.22. The strict monotonicity of Definition 4.21 is the more difficult one, but it can be satisfied by representing leaves of the characteristic tree by variables. Let us first define the representation $\lceil \tau \rceil$ of a non-empty characteristic tree τ inductively as follows (using the functor $m/2$ as well as the usual functors for representing lists):

- $\lceil \tau \rceil = X$ where X is a fresh variable **if** $top(\tau) = \emptyset$
- $\lceil \tau \rceil = [m(m_1, \lceil \tau \downarrow (l \circ m_1) \rceil), \ldots, m(m_k, \lceil \tau \downarrow (l \circ m_k) \rceil)]$ **if** $top(\tau) = \{(l \circ m_1), \ldots, (l \circ m_k)\}$ and where $m_1 < \ldots < m_k$.

For example, using the above definition, we have

$$\lceil \{\langle 1 \circ 3 \rangle\} \rceil = [m(3, X)]$$

and

$$\lceil \{\langle 1 \circ 3, 2 \circ 4 \rangle\} \rceil = [m(3, [m(4, X)])].$$

Note that $\{\langle 1 \circ 3 \rangle\} \prec_\tau \{\langle 1 \circ 3, 2 \circ 4 \rangle\}$ and indeed $\lceil \{\langle 1 \circ 3 \rangle\} \rceil \prec_\tau \lceil \{\langle 1 \circ 3, 2 \circ 4 \rangle\} \rceil$. Also note that, because there are only finitely many clause numbers, these terms can be expressed in a finite alphabet.[12]

[12] We can make $\lceil . \rceil$ injective (i.e. one-to-one) by adding the numbers of the selected literals. But because these numbers are not a priori bounded we then have to represent them differently than the clause numbers, e.g. in the form of $s(\ldots(0)\ldots)$, in order to stay within a finite alphabet.

Note that if $\tau_1 \prec_\tau \tau_2$ we immediately have by Definition 4.4 that $\tau_1 \neq \emptyset$ and $\tau_2 \neq \emptyset$. It is therefore sufficient to prove strict monotonicity for two non-empty characteristic trees $\tau_1 \prec_\tau \tau_2$, which can be done inductively as follows.

Base Case: First if $top(\tau_1) = \emptyset$ then $\lceil\tau_1\rceil$ is a fresh variable X. Furthermore, because $\tau_2 \neq \tau_1$, $top(\tau_2) \neq \emptyset$ and $\lceil\tau_2\rceil$ will be a strict instance of X, i.e. $\lceil\tau_1\rceil \prec \lceil\tau_2\rceil$. Note that, if we just have $\tau_1 \preceq_\tau \tau_2$ we still have $\lceil\tau_1\rceil \preceq \lceil\tau_2\rceil$.

Induction Step: If however $top(\tau_1) \neq \emptyset$ then, because $\tau_1 \prec_\tau \tau_2$, we trivially have that $top(\tau_1) = top(\tau_2)$. Let $top(\tau_1) = \{(l \circ m_1), \ldots, (l \circ m_k)\}$. Both $\lceil\tau_1\rceil$ and $\lceil\tau_2\rceil$ will by definition have the same top-level term structure — they might only differ in their respective subterms $\{\lceil\tau_1 \downarrow (l \circ m_i)\rceil \mid 1 \leq i \leq k\}$ and $\{\lceil\tau_2 \downarrow (l \circ m_i)\rceil \mid 1 \leq i \leq k\}$. We can now proceed by induction. First by Lemma C.1 we have that $\tau_1 \downarrow (l \circ m_i) \preceq_\tau \tau_2 \downarrow (l \circ m_i)$. Furthermore there must be at least one index j such that $\tau_1 \downarrow (l \circ m_j) \prec_\tau \tau_2 \downarrow (l \circ m_j)$, otherwise $\tau_1 \equiv_\tau \tau_2$. For this index j we can show by induction hypothesis that $\lceil\tau_1 \downarrow (l \circ m_j)\rceil \prec \lceil\tau_2 \downarrow (l \circ m_j)\rceil$. For the other indexes $i \neq j$ we can show that $\lceil\tau_1 \downarrow (l \circ m_i)\rceil \preceq \lceil\tau_2 \downarrow (l \circ m_i)\rceil$ by a subsidiary induction, very similar to the current one (showing that whenever $\tau_1 \preceq_\tau \tau_2$ then $\lceil\tau_1\rceil \preceq \lceil\tau_2\rceil$ — the base case of this induction has already been handled above). Finally, because all variables used are fresh and thus distinct, there can be no aliasing between the respective subterms and we can therefore conclude that $\lceil\tau_1\rceil \prec_\tau \lceil\tau_2\rceil$.

Remember that $msg(\tau_1, \tau_2)$ is defined unless one of the characteristic trees is empty while the other one is not. Therefore, to guarantee that if $\lceil\tau_1\rceil \trianglelefteq^* \lceil\tau_2\rceil$ holds then $msg(\tau_1, \tau_2)$ is defined, we simply have to ensure that

- $\lceil\emptyset\rceil \trianglelefteq^* \lceil\tau\rceil$ iff $\tau = \emptyset$ and
- $\lceil\tau\rceil \trianglelefteq^* \lceil\emptyset\rceil$ iff $\tau = \emptyset$.

This can be done by defining $\lceil\emptyset\rceil = empty$ where the predicate $empty$ is not used in the representation of characteristic trees different from $\emptyset$. □

The existence of such a mapping also implies, by Lemma 3.30, that there are no infinite chains of strictly more general characteristic trees. (This would not have been true for a more refined definition of "more general" based on just the set of concretisations.)

Also note that the inverse of the mapping $\lceil.\rceil$, introduced in the proof of Proposition 4.22, is not total but still strictly monotonic. We do not need this property in the paper however. But note that, because the inverse is not total, we cannot simply apply the *msg* on the term representation of characteristic trees in order to obtain a generalisation. In other words, the definition of $\lceil.\rceil$ does not make the notion of an *msg* for characteristic trees, defined in Lemma 4.8, superfluous.

D Some Additional Discussions

In this appendix we include some additional discussions, further motivating the use of global trees and characteristic atoms. These discussions can be moved into the paper or entirely dropped, depending on the referees' preference.

D.1 On the Importance of Global Trees

Note that Algorithm 4.29, as well as the ECCE system based on it, structure the characteristic atoms in a global tree, and do not just put them in a set as in Algorithm 3.27 of

Section 3.4. The following example shows that generalisation with non-ancestors may in general severely limit specialisation potential.

Example D.1 Let P be the usual *append* program where a type check on the second argument has been added:

(1) $app([], L, L) \leftarrow$
(2) $app([H|X], Y, [H|Z]) \leftarrow ls(Y), app(X, Y, Z)$
(3) $ls([]) \leftarrow$
(4) $ls([H|T]) \leftarrow ls(T)$

Let $A = app(X, [], Z)$ and $B = app(X, [Y], Z)$. When unfolding as depicted in Figure 12, we obtain $chtree(\leftarrow A, P, U) = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 3\rangle\} = \tau_A$ and $chtree(\leftarrow B, P, U) = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 4, 1 \circ 3\rangle\} = \tau_B$. The only body atom of (A, τ_A) is $app(X', [], Z')$ and the sole body atom of (B, τ_B) is $app(X', [Y], Z')$. In other words, the set $\{(A, \tau_A), (B, \tau_B)\}$ is covered and no potential for non-termination exists. However, if we collect characteristic atoms in a set rather than a tree, we do not notice that (B, τ_B) does not descend from (A, τ_A). Consequently, a growing of characteristic trees (and syntactic structure) will be detected, leading to an unnecessary generalisation of (B, τ_B), and an unacceptable loss of precision.

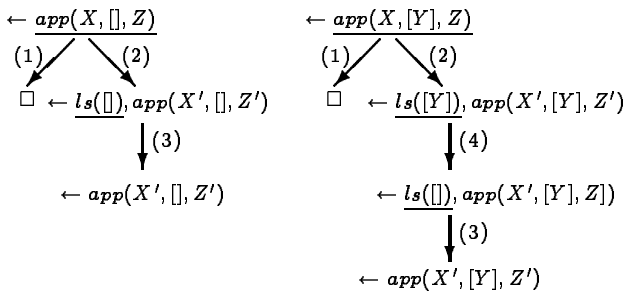


Figure 12: SLD-trees for Example D.1

D.2 On the Importance of Characteristic Atoms

One might also wonder whether, in a setting where the characteristic atoms are structured in a global tree, it would not be sufficient to just test for homeomorphic embedding on the atom part. The intuition behind this would be that a growth of the structure of an atom part would for reasonable programs and unfolding rules lead to a growth of the associated characteristic tree as well — so, using characteristic trees for deciding when to abstract would actually be superfluous. For instance, in Example 4.1 we observe that $rev(L, [], R) \trianglelefteq^* rev(T, [H], R)$ and, indeed, for the corresponding characteristic trees $\lceil\{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 3\rangle\}\rceil \trianglelefteq^* \lceil\{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 4, 1 \circ 3\rangle\}\rceil$ holds. Nonetheless, the intuition turns out to be incorrect. The following examples illustrate this point.

Example D.2 Let P be the following normal program searching for paths without loops:

(1) $path(X,Y,L) \leftarrow \neg member(X,L), arc(X,Y)$
(2) $path(X,Y,L) \leftarrow \neg member(X,L), arc(X,Z), path(Z,Y,[X|L])$
(3) $arc(a,b) \leftarrow$
(4) $arc(b,a) \leftarrow$
(5) $member(X,[X|T]) \leftarrow$
(6) $member(X,[Y|T]) \leftarrow member(X,T)$

Let $A = path(a,Y,[])$ and $B = path(a,Y,[b,a])$ and U an unfolding rule based on $\trianglelefteq^*$. The SLDNF-tree accordingly built for $\leftarrow A$ is depicted in Figure 13. B occurs in a leaf $(A \trianglelefteq^* B)$ and will hence descend from A in the global tree. But the (term representation of) the characteristic tree $\tau_A = \{\langle 1 \circ 1, 1 \circ member, 1 \circ 3\rangle, \langle 1 \circ 2, 1 \circ member, 1 \circ 3, 1 \circ 1, 1 \circ member, 1 \circ 4\rangle, \langle 1 \circ 2, 1 \circ member, 1 \circ 3, 1 \circ 2, 1 \circ member, 1 \circ 4\rangle\}$ is not embedded in (the representation of) $\tau_B = \emptyset$, and no danger for non-termination exists (more structure resulted in this case in failure and thus less unfolding). A method based on testing only $\trianglelefteq^*$ on the atom component would abstract B unnecessarily.

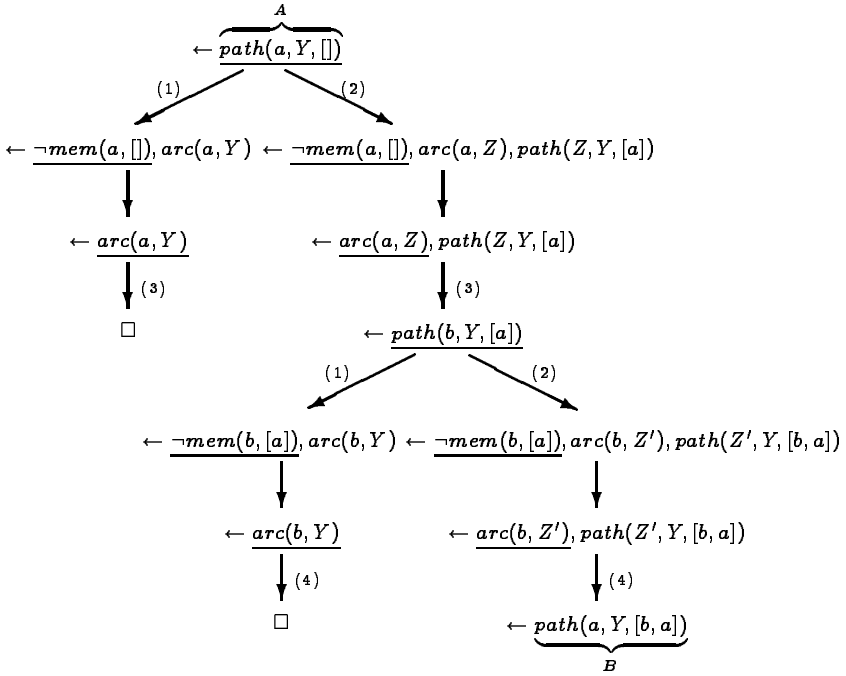


Figure 13: SLDNF-tree for Example D.2

Example D.3 Let P be the following *definite* program:

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

Let $A = path(L)$. Unfolding $\leftarrow A$ (using an unfolding rule U based on $\trianglelefteq^*$) will result in lifting $B = path([b|T])$ to the global level. The characteristic trees are:
$\tau_A = \{\langle 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 3\rangle\}$,
$\tau_B = \{\langle 1 \circ 1\rangle\}$. Again, $A \trianglelefteq^* B$ holds, but not $\lceil \tau_A \rceil \trianglelefteq^* \lceil \tau_B \rceil$.

Finally, in recent experiments, it also turned out that characteristic trees might be a vital asset when trying to solve the *parsing problem* [51], which appears when unfolding meta-interpreters with non-trivial object programs. In such a setting a growing of the syntactic structure also does not imply a growing of the characteristic tree.

Example D.4 Take the vanilla meta-interpreter with a simple family database at the object level:

$solve(empty) \leftarrow$
$solve(A\&B) \leftarrow solve(A), solve(B)$
$solve(A) \leftarrow clause(A, B), solve(B)$
$clause(anc(X, Y), parent(X, Y)) \leftarrow$
$clause(anc(X, Z), parent(X, Y)\&anc(Y, Z)) \leftarrow$
$clause(parent(peter, paul), empty) \leftarrow$
$clause(parent(paul, mary), empty) \leftarrow$

Let $A = solve(anc(X, Z))$ and $B = solve(parent(X, Y)\&anc(Y, Z))$. We have $A \trianglelefteq^* B$ and without characteristic trees these two atoms would be generalised (supposing that both these atoms occur at the global level) by their *msg* $solve(G)$. If however, we take characteristic trees into account, we will notice that the object level atom $anc(Z, X)$ within A has more solutions than the object level atom $anc(Y, Z)$ within B (because in the latter one Y will get instantiated through further unfolding). I.e. the characteristic tree of B does not embed the one of A and no unnecessary generalisation will occur.