Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs

MICHAEL LEUSCHEL, BERN MARTENS, and DANNY DE SCHREYE
Katholieke Universiteit Leuven

Given a program and some input data, partial deduction computes a specialized program handling any remaining input more efficiently. However, controlling the process well is a rather difficult problem. In this article, we elaborate global control for partial deduction: for which atoms, among possibly infinitely many, should specialized relations be produced, meanwhile guaranteeing correctness as well as termination? Our work is based on two ingredients. First, we use the concept of a characteristic tree, encapsulating specialization behavior rather than syntactic structure, to guide generalization and polyvariance, and we show how this can be done in a correct and elegant way. Second, we structure combinations of atoms and associated characteristic trees in global trees registering "causal" relationships among such pairs. This allows us to spot looming nontermination and perform proper generalization in order to avert the danger, without having to impose a depth bound on characteristic trees. The practical relevance and benefits of the work are illustrated through extensive experiments. Finally, a similar approach may improve upon current (on-line) control strategies for program transformation in general such as (positive) supercompilation of functional programs. It also seems valuable in the context of abstract interpretation to handle infinite domains of infinite height with more precision.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*logic programming*; D.1.6 [**Programming Techniques**]: Logic Programming; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*logic programming*; I.2.2 [**Artificial Intelligence**]: Automatic Programming

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Flow analysis, partial deduction, partial evaluation, program transformation, supercompilation

Michael Leuschel is Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.), and was also supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation." Bern Martens is a Postdoctoral Fellow of the K.U.Leuven Research Council, Belgium, and was also partially supported by the HCM Network "Logic Program Synthesis and Transformation," contract no. CHRX-CT-93-00414, University of Padova, Italy, and the Belgian GOA "Non-Standard Applications of Abstract Interpretation." Danny De Schreye is Senior Research Associate of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.).

Authors' addresses: Departement Computerwetenschappen, K.U. Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium; email: {michael; bern; dannyd}@cs.kuleuven.ac.be

1. INTRODUCTION

A major concern in the specialization of functional (e.g., see Consel and Danvy [1993], Jones et al. [1993], and Turchin [1986]) as well as logic programs (e.g., see Lloyd and Shepherdson [1991], Komorowski [1992], Gallagher [1993], Pettorosii and Prooietti [1994], and De Schreye et al. [1995]) 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 [Consel and Danvy 1993; Jones et al. 1993] has mainly stressed the former, while supercompilation of functional programs [Turchin 1986; 1988; Søensen and Glück 1995] and partial deduction of logic programs [Bol 1993; Bruynooghe et al. 1992; Gallagher and Bruynooghe 1991; Martens and De Schreye 1996 Martens and Gallagher 1995; Sahlin 1993] have concentrated on on-line control. (Some exceptions are the works of Weise et al. [1991], Mogensen and Bondorf [1992], Leuschel [1994], Jørgensen and Leuschel [1996], and Bruynooghe et al. [1998].)

The concrete setting of the present article concerns logic program partial deduction. We are, however, convinced that its contribution is not limited to that particular field. Indeed, its techniques and ideas are also relevant to the control of supercompilation and on-line partial evaluation of functional (and perhaps also imperative) languages. Moreover, even abstract interpretation can benefit from them. We return to these points in Section 4.7.

In partial deduction of logic programs, one distinguishes two levels of control [Gallagher 1993; Martens and Gallagher 1995]: 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 us to construct specialized clauses for the given atoms [Benkerimi and Lloyd 1990; Lloyd and Shepherdson 1991]. At the global level on the other hand, one typically attends to the overall correctness of the resulting program (satisfying the closedness condition in Lloyd and Shepherdson [1991]) and strives to achieve the "right" amount of polyvariance, producing sufficiently many (but not more) specialized versions for each predicate definition in the original program. At both levels, in the context of providing a fully automatic tool, termination is obviously of prime importance.

In this article, it is to the global level that we turn our attention. The concept of *characteristic trees* has been proposed as a basis for the global control of partial deduction in Gallagher and Bruynooghe [1991] and Gallagher [1991]. The main idea is, instead of using the *syntactic* structure to decide upon polyvariance, to examine the specialization *behavior* of the atoms to be specialized: only if this behavior is sufficiently different from one atom to the other should different specialized versions be generated. However, the approaches based on characteristic trees have been, up to now, ridden by two major problems. First, they failed to preserve the characteristic trees upon generalization: two atoms with identical specialization behavior may be generalized by one with a completely different specialization behavior. This may lead to significant losses in specialization, as well as to problems for the termination of the partial deduction process. Second, the characteristic trees have always been limited by some ad hoc depth bound, possibly leading to very undesirable specialization results (as we will show later in the article).

In the present article, we solve these two problems. After presenting some required background about partial deduction and characteristic trees in Section 2, we address 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 the general framework in Martens and Gallagher [1995] with the framework developed in Section 3. We thus obtain an elegant, sophisticated, and precise apparatus for on-line global control of partial deduction. Benchmark results can be found in Section 5. Section 6 subsequently concludes the article.

Finally, we would like to mention that two workshop papers address part of the material in a preliminary form:

—Leuschel [1995] describes how to impose characteristic trees and perform set-based partial deduction with characteristic atoms,

—while Leuschel and Martens [1996] elaborates the former approach into one using global trees, and thus not requiring any depth bound.

2. PRELIMINARIES AND MOTIVATIONS

Throughout this article, we suppose familiarity with basic notions in logic programming [Lloyd 1987] and partial deduction [Lloyd and Shepherdson 1991]. Notational conventions are standard and self-evident. In particular, in programs, we denote variables through (strings starting with) an uppercase symbol, while the notations of constants, functions, and predicates begin with a lowercase character. Unless stated explicitly otherwise, the terms "(logic) program" and "goal" will refer to a *normal* logic program and goal, respectively. 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 language $\mathcal{L}_P$ which we implicitly assume underlying the program P under consideration. $\mathcal{L}_P$ may contain additional symbols not occurring in P, but, unless explicitly stated otherwise, $\mathcal{L}_P$ contains only finitely many constant, function, and predicate symbols. To simplify the presentation we also assume that, when talking about expressions, predicate symbols and connectives are treated like functors which cannot be confounded with the original functors and constants (e.g., $\wedge$ and $\leftarrow$ are binary functors distinct from the other binary functors). As common in partial deduction, the notion of SLDNF-trees is extended to allow *incomplete* SLDNF-trees which may contain leaves where no literal has been selected for a further derivation step. Leaves of the latter kind will be called *dangling* [Martens and De Schreye 1996].

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 *specialized* 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 $P \cup \{G\}$.

Partial input 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 Komorowski [1992]), a convention we will adhere to in this article. So, given this goal G' and the original program P, partial deduction produces a new program P' which is P "specialized" 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 $\mathcal{A}$. The derivation steps in these SLDNF-trees correspond to the computation steps performed (beforehand) by the *partial deducer*, and the specialized program is extracted from these trees by constructing one specialized clause per branch. The incomplete SLDNF-trees are obtained by applying an unfolding rule, defined as follows:

Definition 2.1.1. 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 2.1.1 is so general as to even allow a *trivial* SLDNF-tree, i.e., one whose root is a dangling leaf.

Specialized clauses are extracted from the SLDNF-trees as follows:

Definition 2.1.2. Let P be a program and A an atom. Let τ be a finite, incomplete, and nontrivial[1] SLDNF-tree for $P \cup \{\leftarrow A\}$. Let $\leftarrow G_1, \ldots, \leftarrow G_n$ be the goals in the (nonroot) leaves of the nonfailing 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 for example by Lloyd and Shepherdson [1991] and Benkerimi and Lloyd [1990], uses the resultants for a given set of atoms $\mathcal{A}$ to construct the specialized program and for each atom in $\mathcal{A}$ a different specialized predicate definition is generated, replacing the original definition in P. Under the conditions stated in Lloyd and Shepherdson [1991], namely closedness and independence, correctness of the specialized program is guaranteed. *Independence* requires that no two atoms in $\mathcal{A}$ have a common instance. This ensures that no two specialized predicate definitions match the same (run-time) call. Usually this condition is satisfied by performing a renaming of the atoms in $\mathcal{A}$ (see Gallagher and Bruynooghe [1990] and Benkerimi and Hill [1993]). *Closedness* (as well as the related notion of *coveredness* in Benkerimi and Lloyd [1990]) basically requires that every atom in the body of a resultant is matched by a specialized predicate definition. This guarantees that $\mathcal{A}$ forms a *complete description* of all possible computations that can occur at run-time of the specialized program.

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

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

2.2 Abstraction

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

Definition 2.2.1. Let $\mathcal{A}$ and $\mathcal{A}'$ be sets of atoms. Then $\mathcal{A}'$ is an *abstraction* of $\mathcal{A}$ if and only if every atom in $\mathcal{A}$ is an instance of an atom in $\mathcal{A}'$. 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 Gallagher [1991] and Gallagher [1993], describes the basic layout of practically all algorithms for controlling partial deduction.

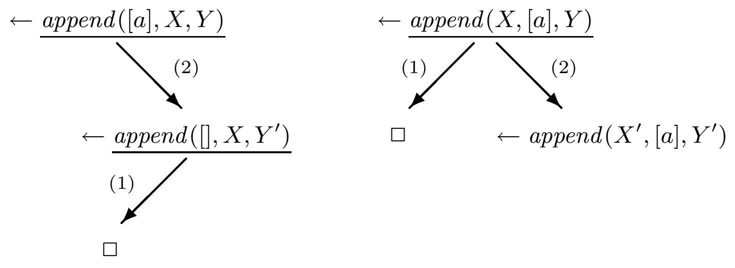
Algorithm 2.2.2 (*Standard Partial Deduction*).

```
Input: A program $P$ and a goal $G$
Output: A specialized program $P'$
Initialization: $i = 0$ , $\mathcal{A}_i = \{A \mid A$ is an atom in $G\}$
  repeat
    for each $A_k \in \mathcal{A}_i$, compute a finite SLDNF-tree $\tau_k$ for $P \cup \{\leftarrow A_k\}$
        by applying unfolding rule $U$;
    let $\mathcal{A}_i' := \mathcal{A}_i \cup \{B_l | B_l$ is an atom in a leaf of some tree $\tau_k$, such that
                            $B_l$ is not an instance [2] of any $A_j \in \mathcal{A}_i\}$ ;
    let $\mathcal{A}_{i+1} :=$ abstract$(\mathcal{A}_i')$ where abstract is an abstraction operator
    let $i := i + 1$;
  until $\mathcal{A}_{i+1} = \mathcal{A}_i$
  Apply a renaming transformation to $\mathcal{A}_i$ 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 maximizes specialization while ensuring termination*.

The abstraction operator examines the set of atoms to be partially deduced and then decides which 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 $\mathcal{A}_i$ and using the *most specific generalization* (*msg*)[3] to stick to that maximum. This approach is however unsatisfactory. First, it involves an ad hoc maximum number, which might lead to either too much or too little polyvariance, depending on the context. Second, the *msg* is just based on the *syntactic structure* of the atoms to be specialized. This is generally not such a good idea. Indeed, two atoms can be unfolded and specialized in a very similar way in the context of one program P_1, while in the context of another program P_2 their specialization behavior can be drastically different. The

[2]One can also use the variant test to make the algorithm more precise.
[3]Also known as antiunification or least general generalization; see for instance Lassez et al. [1988].



$\leftarrow \underline{append([a],X,Y)}$
(2)
$\leftarrow \underline{append([],X,Y')}$
(1)
$\square$
$\leftarrow \underline{append(X,[a],Y)}$
(1)
(2)
$\square$
$\leftarrow append(X',[a],Y')$

Fig. 1. SLD-trees τ_B and τ_C for Example 2.2.3.

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 different generalizations might be called for.

A better candidate for an abstraction might be to examine the SLDNF-trees generated for these atoms. These trees capture (to some depth) how the atoms behave computationally in the context of the respective programs. They also capture (part of) the specialization that has been performed on these atoms. An abstraction operator which takes these trees into account will notice their similar behavior in the context of P_1 and their dissimilar behavior within P_2, and can therefore take appropriate actions in the form of different generalizations. The following example illustrates these points.

Example 2.2.3. Let P be the *append* program:

(1) $append([],Z,Z) \leftarrow$

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

Note that we have added clause numbers, which we will henceforth take the liberty to incorporate into illustrations of SLD-trees in order to clarify which clauses have been resolved with. To avoid cluttering the figures we will also sometimes drop the substitutions in such figures.

Let $\mathcal{A} = \{B, C\}$ 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 $\mathcal{A}$ 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

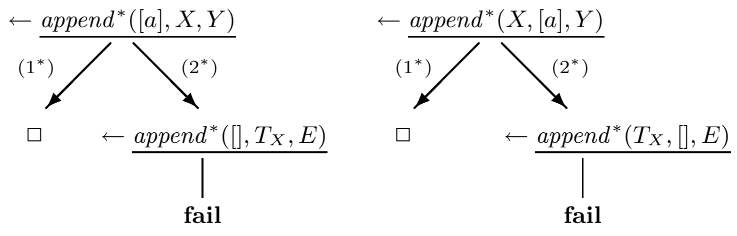
$$append([a],X,[a|X]) \leftarrow$$

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

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

So, in this case, it is vital to keep separate specialized versions for B and C and not abstract them for example by their *msg*.

However, it is very easy to come up with another context in which the specialization behaviors of B and C are almost indiscernible. Take for instance the following



$\leftarrow \underline{append^*([a], X, Y)}$
(1*)
(2*)
$\square$
$\leftarrow \underline{append^*([], T_X, E)}$
fail
$\leftarrow \underline{append^*(X, [a], Y)}$
(1*)
(2*)
$\square$
$\leftarrow \underline{append^*(T_X, [], E)}$
fail

Fig. 2. SLD-trees τ_B^* and τ_C^* for Example 2.2.3.

program P^* in which *append** no longer appends two lists, but finds common elements at common positions:

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

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 specialized versions for B and C because the following single set of specialized clauses could be used for B and C without specialization 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 SLDNF-trees into consideration.

2.3 Characteristic Paths and Trees

Above we have illustrated the interest of examining the SLDNF-trees generated for the atoms to be partially deduced and for example only abstract 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 Gallagher [1991], do just that: they characterize the essential structure of SLDNF-derivations and trees.

Definition 2.3.1 (*Characteristic Path*). 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 $\langle l_0 : c_0, \ldots, l_{n-1} : c_{n-1}\rangle$, where l_i is the position of the selected literal in G_i, and c_i is defined as follows:

—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 $\langle 1:2, 1:1 \rangle$.

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 characterize *finite* and *nonfailing* derivations of *atomic* goals. Once the top-level goal is known, the characteristic path is sufficient to reconstruct all the intermediate goals as well as the final one. So, using p in the second point of Definition 2.3.1 instead of a unique symbol to signal the selection of a negative literal is a matter of convention rather than necessity.

Now that we have characterized derivations, we can characterize goals through the derivations in their associated SLDNF-trees.

Definition 2.3.2 (*Characteristic Tree*). Let G be a goal, P a normal program, and τ 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 nonfailing SLDNF-derivations associated with the branches of τ. $\hat{\tau}$ is called a characteristic tree if and only if 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 is not lost and 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).

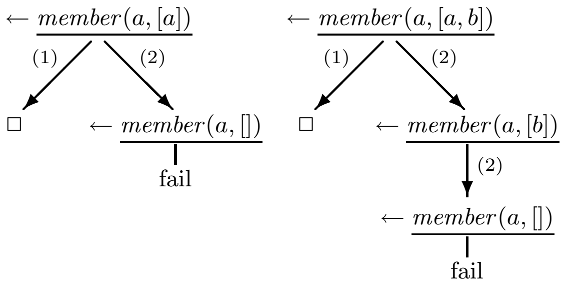
Example 2.3.3. The characteristic trees of the finite SLD-trees τ_B and τ_C in Figure 1 are $\{\langle 1:2, 1:1 \rangle\}$ and $\{\langle 1:1 \rangle, \langle 1:2 \rangle\}$ respectively. The characteristic trees of the finite SLD-trees τ_B^* and τ_C^* in Figure 2 are both $\{\langle 1:1^* \rangle\}$.

The characteristic tree captures all the relevant aspects of specialization, attained by the local control for a particular atom:

—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 (implicitly) which predicates are called in the bodies of the resultants. As we will see later, this means that a single predicate definition can (in principle) be used for two atoms which have the same characteristic tree.



$\leftarrow \underline{member(a,[a])}$
(1)
(2)
$\square$
$\leftarrow \underline{member(a,[])}$
fail
$\leftarrow \underline{member(a,[a,b])}$
(1)
(2)
$\square$
$\leftarrow \underline{member(a,[b])}$
(2)
$\leftarrow \underline{member(a,[])}$
fail

Fig. 3. SLD-trees for Example 2.3.4.

A specialization aspect that does not materialize within a characteristic tree is how the atoms in the leaves of the associated SLDNF-tree are further specialized, i.e., the global control and precision are not captured.

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

Example 2.3.4. Let P be the following program:

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

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

Let $A = member(a,[a,b])$ and $B = member(a,[a])$. Suppose that A and B are unfolded as depicted in Figure 3. Then both these atoms have the same characteristic tree $\tau = \{\langle 1:1 \rangle\}$ 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 materialize within the resultants (i.e., the specialized code generated for the atoms), and furthermore the single resultant $member(a,[a|T]) \leftarrow$ could be used for both A and B without loosing any specialization.

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

2.4 An Abstraction Operator Using Characteristic Trees

The following abstraction operator represents a first attempt at using characteristic trees for the control of polyvariance. Basically it classifies atoms according to their associated characteristic tree. Generalization, in this case the *msg*, is then only applied on those atoms which have the same characteristic tree.

Definition 2.4.1 ($chabs_{P,U}$). Let P be a normal program, U an unfolding rule and $\mathcal{A}$ a set of atoms. For every characteristic tree τ, let $\mathcal{A}_\tau$ be defined as $\mathcal{A}_\tau = \{A \mid A \in \mathcal{A} \wedge chtree(\leftarrow A, P, U) = \tau\}$. The abstraction operator $chabs_{P,U}$ is then defined as $chabs_{P,U}(\mathcal{A}) = \{msg(\mathcal{A}_\tau) \mid \tau \text{ is a characteristic tree}\}$.

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 specialization.

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

Example 2.4.2. Let us return to Example 2.3.4 and specialize the *member* program for $A = member(a, [a, b])$ and $B = member(a, [a])$. First we put the atoms into $\mathcal{A}_0$: $\mathcal{A}_0 = \{member(a, [a, b]), member(a, [a])\}$. For both atoms the associated characteristic tree is $\tau = \{\langle 1 : 1\rangle\}$, given for example a determinate unfolding rule with lookahead (see Figure 3). Thus $chabs_{P,U}$, as well as the method of Gallagher [1991], abstracts the two atoms and produces the generalized set $\mathcal{A}_1 = \{member(a, [a|T])\}$. Unfortunately, the generalized atom $member(a, [a|T])$ has a different characteristic tree τ', independently of the particular unfolding rule. For a determinate unfolding rule with lookahead we obtain $\tau' = \{\langle 1 : 1\rangle, \langle 1 : 2\rangle\}$.

This loss of precision leads to suboptimal specialized programs. At the next step of the algorithm the atom $member(a, T)$ will be added to $\mathcal{A}_1$. This atom also has the characteristic tree τ' under U. Hence the final set $\mathcal{A}_2$ equals $\{member(a, L)\}$ (containing the *msg* of $member(a, [a|T])$ and $member(a, T)\}$), and we obtain the following specialization, suboptimal 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 nonfailing resolvent, this information has been lost due to an imprecision of the abstraction operator, thereby leading to a suboptimal 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 (a filtered version of) clause (1').

As explained by Leuschel and De Schreye [1998], these losses of precision can also lead to nontermination of the partial deduction process, further illustrating the importance of preserving characteristic trees upon generalization.

3. PARTIAL DEDUCTION WITH CHARACTERISTIC ATOMS

We now present a way to preserve characteristic trees during abstraction. The basic idea is to simply *impose* characteristic trees on the generalized atoms.

3.1 Characteristic Atoms

We first introduce the crucial notion of a *characteristic atom*.

Definition 3.1.1. 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 A in the context of the particular program P under consideration.

Example 3.1.2. Let $\tau = \{\langle 1 : 1\rangle\}$ be a characteristic tree. Then the couples $CA_1 = (member(a, [a, b]), \tau)$ and $CA_2 = (member(a, [a]), \tau)$ are characteristic atoms. $CA_3 = (member(a, [a|T]), \tau)$ is also a characteristic atom, but for example in the context of the *member* program P from Example 2.3.4, τ is *not* a characteristic tree of its atom component $member(a, [a|T])$ (cf. Example 2.4.2). 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* generalization (in P) of the atoms $member(a, [a, b])$ and $member(a, [a])$, solving the problem of Example 2.4.2.

A characteristic atom will be used to represent a possibly infinite set of atoms, called its concretizations. 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 3.1.3 (*Concretization*). Let (A, τ_A) be a characteristic atom and P a program. An atom B is a *precise concretization* of (A, τ_A) (in P)[4] if and only if B is an instance of A and, for some unfolding rule U, $chtree(\leftarrow B, P, U) = \tau_A$. An atom is a *concretization* of (A, τ_A) (in P) if and only if it is an instance of a precise concretization of (A, τ_A) (in P). We denote the set of concretizations of (A, τ_A) in P by $\gamma_P(A, \tau_A)$.

A characteristic atom with a nonempty set of concretizations 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 only produce P-characteristic atoms, where P is the original program to be specialized.

Example 3.1.4. Take the characteristic atom $CA_3 = (member(a, [a|T]), \tau)$ with $\tau = \{\langle 1 : 1 \rangle\}$ from Example 3.1.2 and the *member* program P from Example 2.3.4. The atoms $member(a, [a])$ and $member(a, [a, b])$ are precise concretizations of CA_3 in P (see Figure 3). Also, neither $member(a, [a|T])$ nor $member(a, [a, a])$ are concretizations of CA_3 in P. Finally, observe that CA_3 is a P-characteristic atom while for instance $(member(a, [a|T]), \{\langle 2 : 5 \rangle\}$ or even $(member(a, [a|T]), \{\langle 1 : 2 \rangle\}$ are not.

Example 3.1.5. 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 $\tau = \{\langle 1 : 1 \rangle, \langle 1 : 2, 1 : q, 1 : 3 \rangle\}$ and $CA = (p(X, Y), \tau)$. Then $p(X, a)$ and $p(X, c)$ are precise concretizations of CA in P and neither $p(b, b)$, $p(X, b)$, $p(a, Y)$, nor $p(X, Y)$ are concretizations of CA in P. Also $p(b, a)$ and $p(a, a)$ are both concretizations of CA in P, but they are not precise concretizations. Observe that the selection of the negative literal $\neg q(Y)$, corresponding to $1 : q$ in τ, is unsafe for $\leftarrow p(X, Y)$, but that for any concretization of CA in P the corresponding derivation step is safe and succeeds (i.e., the negated atom is ground and fails finitely).

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

[4] If P is clear from the context, we will not explicitly mention it.

Definition 3.1.6 (*Unconstrained Characteristic Atom*). A P-characteristic atom (A, τ_A) with $A \in \gamma_P(A, \tau_A)$ is called *unconstrained* (in P).

The concretizations of an unconstrained characteristic atom (A, τ_A) are identical to the instances of the ordinary atom A (because the concretizations are downward closed, and $\gamma_P(A, \tau)$ contains no atom strictly more general than A). So, characteristic atoms along with Definition 3.1.3 provide a proper generalization of the way atoms are used in the standard partial deduction approach.

3.2 Generating Resultants

We now address the generation of resultants for characteristic atoms. We need the following definition in order to formalize the resultants associated with a characteristic atom.

Definition 3.2.1. A *generalized SLDNF-derivation* is either composed of ordinary SLDNF-derivation steps or of derivation steps in which a nonground negative literal is selected and removed. Generalized SLDNF-derivations will be called *unsafe* if they contain steps of the latter kind and *safe* if they do not.

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

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

Definition 3.2.2 ($D_P(A, \tau)$). Let P be a program and (A, τ) a P-characteristic atom. If $\tau \neq \{\langle\rangle\}$ then $D_P(A, \tau)$ is the set of all generalized SLDNF-derivations of $P \cup \{\leftarrow A\}$ such that their characteristic paths are in τ. If $\tau = \{\langle\rangle\}$ then $D_P(A, \tau)$ is the set of all nonfailing SLD-derivations of $P \cup \{\leftarrow A\}$ of length 1.[5]

Note that the derivations in $D_P(A, \tau)$ are necessarily finite and nonfailing (because (A, τ) is a P-characteristic atom; see also Lemma B.2.9, in the electronic appendix to this article).

We will call a P-characteristic atom (A, τ) *safe* (in P) if and only if all derivations in $D_P(A, \tau)$ are safe. An unconstrained characteristic atom in P is safe in P.[6]

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

Definition 3.2.3 (*Partial Deduction of* (A, τ)). Let P be a program and (A, τ) a P-characteristic atom. Let $\{D_1, \dots, D_n\}$ be the generalized SLDNF-derivations in $D_P(A, \tau)$, and let $\leftarrow G_1, \dots, \leftarrow G_n$ be the goals in the leaves of these derivations. Let $\theta_1, \dots, \theta_n$ be the computed answers of the derivations from $\leftarrow A$ to $\leftarrow G_1, \dots, \leftarrow G_n$ respectively. Then the set of resultants $\{A\theta_1 \leftarrow G_1, \dots, A\theta_n \leftarrow G_n\}$ is called the *partial deduction of* (A, τ) *in* P. Every atom occurring in some of the G_i will be called a *leaf atom* (in P) of (A, τ). We will denote the set of such leaf atoms by $leaves_P(A, \tau)$.

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

[6] Because A must be a precise concretization of (A, τ), we have that τ is a characteristic tree of A, and thus all derivations in $D_P(A, \tau)$ are ordinary SLDNF-derivations and therefore safe.

Example 3.2.4. The partial deduction of $(member(a,[a|T]),\{\langle 1:1\rangle\})$ in the program P of Example 2.3.4 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 concretization of $(member(a,[a|T]),\{\langle 1:1\rangle\})$.

Example 3.2.5. The partial deduction P' of $(p(X,Y),\tau)$ with $\tau = \{\langle 1:1\rangle, \langle 1:2, 1:q, 1:3\rangle\}$ of Example 3.1.5 is

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

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

We can now generate partial deductions not for sets of atoms, but for sets of *characteristic* atoms. As such, the 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, becomes even more compelling than in the standard partial deduction setting.

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

Definition 3.2.6 (*Renaming*). An *atomic renaming* α for a set $\tilde{\mathcal{A}}$ of characteristic atoms is a mapping from $\tilde{\mathcal{A}}$ to atoms such that

—for each $(A,\tau) \in \tilde{\mathcal{A}}$, $vars(\alpha((A,\tau))) = vars(A)$;
—for $CA, CA' \in \tilde{\mathcal{A}}$, such that $CA \neq CA'$, the predicate symbols of $\alpha(CA)$ and $\alpha(CA')$ are distinct (but not necessarily fresh, in the sense that they can occur in $\tilde{\mathcal{A}}$).

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

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

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

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

Definition 3.2.7 (*Partial Deduction with respect to* $\tilde{\mathcal{A}}$). Let P be a program and let $\tilde{\mathcal{A}} = \{(A_1,\tau_1),\ldots,(A_n,\tau_n)\}$ be a finite set of P-characteristic atoms. Also, let ρ_α be a renaming function for $\tilde{\mathcal{A}}$ 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 *with respect to* $\tilde{\mathcal{A}}$ *and* ρ_α.

Example 3.2.8. 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 $\tau = \{\langle 1 : 1\rangle\}$, $\tau' = \{\langle 1 : 3\rangle\}$, and let $\tilde{\mathcal{A}} = \{(member(a, [a|T]), \tau), (t, \tau')\}$. Also let $\alpha((member(a, [a|T]), \tau)) = m_1(T)$ and $\alpha((t, \tau')) = t$. Because the concretizations in P of the elements in $\tilde{\mathcal{A}}$ 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 concretizations of $(member(a, [a|T]), \tau)$. Therefore the partial deduction of P with respect to $\tilde{\mathcal{A}}$ and ρ_α is[7]

(1') $m_1(X) \leftarrow$

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

Note that in Definition 3.2.7 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 dissimilar to the Lloyd and Shepherdson [1991] framework. However, there is no fundamental difference between these two approaches: keeping part of the original program can be simulated 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 of ordinary partial deduction in the context of characteristic atoms. This definition will ensure that the renamings, applied for instance in Definition 3.2.7, are always defined.

Definition 3.3.1 (*P-Covered*). Let P be a program and $\tilde{\mathcal{A}}$ a set of characteristic atoms. Then $\tilde{\mathcal{A}}$ is called *P-covered* if and only if for every characteristic atom in $\tilde{\mathcal{A}}$, each of its leaf atoms (in P) is a concretization in P of a characteristic atom in $\tilde{\mathcal{A}}$. Also, a goal G is *P-covered by* $\tilde{\mathcal{A}}$ if and only if every atom A occurring in G is a concretization in P of a characteristic atom in $\tilde{\mathcal{A}}$.

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

THEOREM 3.3.2. *Let P be a normal program, G a goal, $\tilde{\mathcal{A}}$ any finite set of P-characteristic atoms, and P' the partial deduction of P with respect to $\tilde{\mathcal{A}}$ and some ρ_α. If $\tilde{\mathcal{A}}$ is P-covered and if G is P-covered by $\tilde{\mathcal{A}}$ then the following hold:*

(1) $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ if and only if $P \cup \{G\}$ does.

(2) $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree if and only if $P \cup \{G\}$ does.

The proof can be found in the (electronic) Appendix B and proceeds in three successive stages.

[7]The FAR filtering algorithm of Leuschel and Sørensen [1996] can be used to further improve the specialized program by removing the redundant argument of m_1.

(1) First, we restrict ourselves to unconstrained characteristic atoms. This allows us to straightforwardly reuse the correctness results for standard partial deduction with renaming.

(2) We then move on to safe characteristic atoms. Their partial deductions can basically be obtained from partial deductions for unconstrained characteristic atoms by removing certain clauses. We show that these clauses can be safely removed without affecting computed answers or finite failure.

(3) In the final step we 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 establish correctness by showing that, for all concrete executions, these negative literals will be ground and succeed.

3.4 A Set-Based Algorithm and Its Termination

In this section, we present a simple (set-based) algorithm for partial deduction through characteristic atoms.

We first define an abstraction operator which, by definition, preserves the characteristic trees.

Definition 3.4.1 ($chmsg(,)\tilde{\mathcal{A}}|_\tau$). Let $\tilde{\mathcal{A}}$ be a set of characteristic atoms. Also let, for every characteristic tree τ, $\tilde{\mathcal{A}}|_\tau$ be defined as $\tilde{\mathcal{A}}|_\tau = \{A \mid (A,\tau) \in \tilde{\mathcal{A}}\}$. The operator *chmsg*() is defined as

$$chmsg(\tilde{\mathcal{A}}) = \{(msg(\tilde{\mathcal{A}}|_\tau), \tau) \mid \tau \text{ is a characteristic tree }\}.$$

In other words, only one characteristic atom per characteristic tree is allowed in the resulting abstraction. Given for example, $\tilde{\mathcal{A}} = \{(p(a), \{\langle 1:1\rangle\}), (p(b), \{\langle 1:1\rangle\})\}$, we obtain the abstraction $chmsg(\tilde{\mathcal{A}}) = \{(p(X), \{\langle 1:1\rangle\})\}$.

Definition 3.4.2 (*chatom*, *chatoms*). Let A be an ordinary atom, U an unfolding rule, and P a program. We then define *chatom*:

$$chatom(A,P,U) = (A,\tau), \text{ where } \tau = chtree(\leftarrow A,P,U)$$

We extend *chatom* to sets of atoms:

$$chatoms(\tilde{\mathcal{A}},P,U) = \{chatom(A,P,U) \mid A \in \tilde{\mathcal{A}}\}$$

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

The following algorithm for partial deduction with characteristic atoms is parametrized by an unfolding rule U, thus leaving the particulars of local control unspecified. Recall that $leaves_P(A,\tau_A)$ represents the leaf atoms of (A,τ_A) (see Definition 3.2.3).

Algorithm 3.4.3 (*Ecological Partial Deduction*).

```
**Input:** a program $P$ and a goal $G$
**Output:** a specialized program $P'$
**Initialization:** $k := 0$; $\tilde{\mathcal{A}}_0 := chatoms(\{A \mid A \text{ is an atom in } G\}, P, U)$;
  **repeat**
    $\tilde{\mathcal{L}}_k := chatoms(\{leaves_P(A,\tau_A) \mid (A,\tau_A) \in \tilde{\mathcal{A}}_k\}, P, U)$;
    $\tilde{\mathcal{A}}_{k+1} := chmsg(\tilde{\mathcal{A}}_k \cup \tilde{\mathcal{L}}_k)$;   $k := k+1$;
    **until** $\tilde{\mathcal{A}}_k = \tilde{\mathcal{A}}_{k+1}$ (modulo variable renaming)
    $\tilde{\mathcal{A}} := \tilde{\mathcal{A}}_k$;
    $P' :=$ a partial deduction of $P$ with respect to $\tilde{\mathcal{A}}$ and some $\rho_\alpha$;
```

Let us illustrate the operation of Algorithm 3.4.3 on Example 2.3.4.

Example 3.4.4. Let $G = \leftarrow member(a,[a]), member(a,[a,b])$, and P be the program from Example 2.3.4. Also let $chtree(\leftarrow member(a,[a]),P,U) = chtree(\leftarrow member(a,[a,b]),P,U) = \{\langle 1:1\rangle\} = \tau$ (see Figure 3 for the corresponding SLD-trees). The algorithm operates as follows:

(1) $\tilde{\mathcal{A}}_0 = \{(member(a,[a]),\tau),(member(a,[a,b]),\tau)\}$
(2) $leaves_P(member(a,[a]),\tau) = leaves_P(member(a,[a,b]),\tau) = \emptyset$,
$\tilde{\mathcal{A}}_1 = chmsg(\tilde{\mathcal{A}}_0) = \{(member(a,[a|T]),\tau)\}$
(3) $leaves_P(member(a,[a|T]),\tau) = \emptyset$, $\tilde{\mathcal{A}}_2 = chmsg(\tilde{\mathcal{A}}_1) = \tilde{\mathcal{A}}_1$ and we have reached the fixpoint $\tilde{\mathcal{A}}$.

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

$m_1(X) \leftarrow$

$\tilde{\mathcal{A}}$ is P-covered, and every atom in G is a concretization of a characteristic atom in $\tilde{\mathcal{A}}$. Hence Theorem 3.3.2 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.

The following theorem establishes the correctness of Algorithm 3.4.3, as well as its termination under a certain condition.

THEOREM 3.4.5. *If Algorithm 3.4.3 generates a finite number of distinct characteristic trees then it terminates and produces a partial deduction satisfying the requirements of Theorem 3.3.2 for any goal G' whose atoms are instances of atoms in G.*

PROOF. See electronic Appendix C. □

The method for partial deduction as described in this section, using the framework of Section 3, has been called *ecological* partial deduction by Leuschel [1995] because it guarantees the preservation of characteristic trees. A prototype partial deduction system, using Algorithm 3.4.3, has been implemented, and experimental results have been reported by Leuschel [1995] and Meulemans [1995]. We will however further refine the algorithm in the next section and present extensive benchmarks in Section 5.2.

Let us conclude this section with some comments on the relation with the work of Leuschel and De Schreye [1998], which also solves the problem of preserving characteristic trees upon generalization. In fact, Leuschel and De Schreye [1998] achieve this by incorporating disequality constraints into the partial deduction process. Note that in this section and article, 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 to other characteristic atoms, while in the work of Leuschel and De Schreye [1998] 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 the work of Leuschel and De Schreye [1998] 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 generalization, we now turn to the second problem: getting rid of the depth bound, necessary to ensure termination of Algorithm 3.4.3.

4.1 The Depth Bound Problem

In some cases Algorithm 3.4.3 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 neighborhoods in the context of supercompilation of functional programs. We will return to the relation of neighborhoods to characteristic trees in Section 4.7.)

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 specialized program as well as "perfect" polyvariance: *for every predicate, exactly one specialized version is produced for each of its different associated characteristic trees.* However, it turns out that for a fairly *large class of realistic programs* (and unfolding rules), *the number of different characteristic trees generated is infinite.* In those cases Algorithm 3.4.3, as well as all earlier approaches based on characteristic trees [Gallagher 1991; Gallagher and Bruynooghe 1991; Leuschel and De Schreye 1998], terminates at the cost of imposing an ad hoc depth bound on characteristic trees.

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

Example 4.1.1. The following is the well-known reverse with accumulating parameter where a list type check (in the style of Gallagher and de Waal [1992]) 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, unfolding (determinate [Gallagher 1991; Gallagher Bruynooghe [1991]; Leuschel and De Schreye 1998] and well founded [Bruynooghe et al. 1992; Martens and De Schreye 1996; Martens et al. 1994], among others) produces an infinite number of different characteristic atoms, all with a different characteristic tree. Imposing a depth bound of say 100, we obtain termination; however, 100 different characteristic trees (and instantiations of the accumulator) arise, and the algorithm produces 100 different versions of *rev*: one for each characteristic tree. The specialized program thus looks like:

(1') $rev([], [], []) \leftarrow$

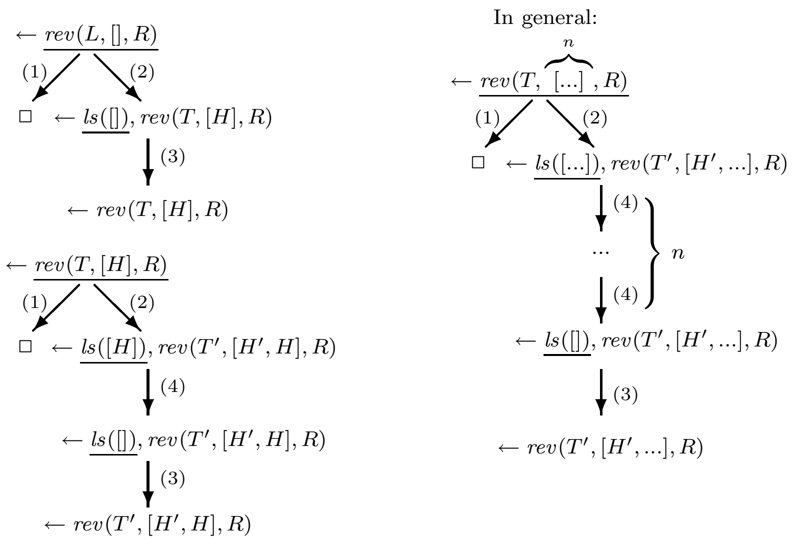


Fig. 4. SLD-trees for Example 4.1.1.

(2') $rev([H|T], [], Res) \leftarrow rev_2(T, [H], Res)$

(3') $rev_2([], [A], [A]) \leftarrow$

(4') $rev_2([H|T], [A], Res) \leftarrow rev_3(T, [H, A], Res)$

(1) $\vdots$

(197') $rev_{99}([], [A_1, \ldots, A_{98}], [A_1, \ldots, A_{98}]) \leftarrow$

(198') $rev_{99}([H|T], [A_1, \ldots, A_{98}], Res) \leftarrow$
$rev_{100}(T, [H, A_1, \ldots, A_{98}], Res)$

(199') $rev_{100}([], [A_1, \ldots, A_{99}|AT], [A_1, \ldots, A_{99}|AT]) \leftarrow$

(200') $rev_{100}([H|T], [A_1, \ldots, A_{99}|AT], Res) \leftarrow$
$ls(AT), rev_{100}(T, [H, A_1, \ldots, A_{99}|AT], Res)$

(201') $ls([]) \leftarrow$

(202') $ls([H|T]) \leftarrow ls(T)$

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

Situations like the above typically arise when an accumulating parameter influences the computation, because then the growing of the accumulator causes a corresponding growing of the characteristic trees. With most simple programs, this is not the case. For instance, in the standard reverse with accumulating parameter, the accumulator is only copied in the end, but never influences the computation. For this reason it was generally felt that natural logic programs would give rise to only finitely many characteristic trees.

```
Program:
 (1) make_non_ground(GrTerm, NgTerm) ←
         mng(GrTerm, NgTerm, [], Sub)
 (2) mng(var(N), X, [], [sub(N, X)]) ←
 (3) mng(var(N), X, [sub(N, X)|T], [sub(N, X)|T]) ←
 (4) mng(var(N), X, [sub(M, Y)|T], [sub(M, Y)|T1]) ←
         not(N = M), mng(var(N), X, T, T1)
 (5) mng(struct(F, GrArgs), struct(F, NgArgs), InSub, OutSub) ←
         l_mng(GrArgs, NgArgs, InSub, OutSub)
 (6) l_mng([], [], Sub, Sub) ←
 (7) l_mng([GrH|GrT], [NgH|NgT], InSub, OutSub) ←
         mng(GrH, NgH, InSub, InSub1),
         l_mng(GrT, NgT, InSub1, OutSub)

Example query:
         ← make_non_ground(struct(f, [var(1), var(2), var(1)]), F)
             ⇝ c.a.s. {F/struct(f, [Z, V, Z])}
```

Fig. 5. Lifting the ground representation.

However, among larger and more sophisticated programs, cases like the above become more and more frequent, even in the absence of type-checking. For instance, in an explicit unification algorithm, one accumulating parameter is the substitution built so far. It heavily influences the computation because new bindings have to be added and checked for compatibility with the current substitution. Another example is the "mixed" metainterpreter of Hill and Gallagher [1994] and Leuschel and De Schreye [1995] (called *InstanceDemo* in the former; part of it is depicted in Figure 5) for the ground representation in which (operationally) the goals are "lifted" to the nonground representation for resolution. To perform the lifting, an accumulating parameter is used to keep track of the ground representation of variables that have already been encountered. This accumulator influences the computation: upon encountering a new ground representation of a variable ($var(N)$ in clauses (2)–(4) of Figure 5), the program inspects the accumulator (the incoming accumulator is represented in the third argument of the predicate *mng*; the resulting accumulator is generated in the fourth argument).

Example 4.1.2. Let $A = l\_mng(Lg, Ln, [sub(N, X)], S)$ and P be the program of Figure 5 in which the predicate *l_mng* transforms a list of ground terms (its first argument) into a list of nonground terms (its second argument; the third and fourth arguments represent the incoming and outgoing accumulator respectively). 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. The third argument has grown, i.e., we have an accumulator. When in turn unfolding $l\_mng(Tg, Tn, [sub(N, X), sub(J, Hn)], S)$, we 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,

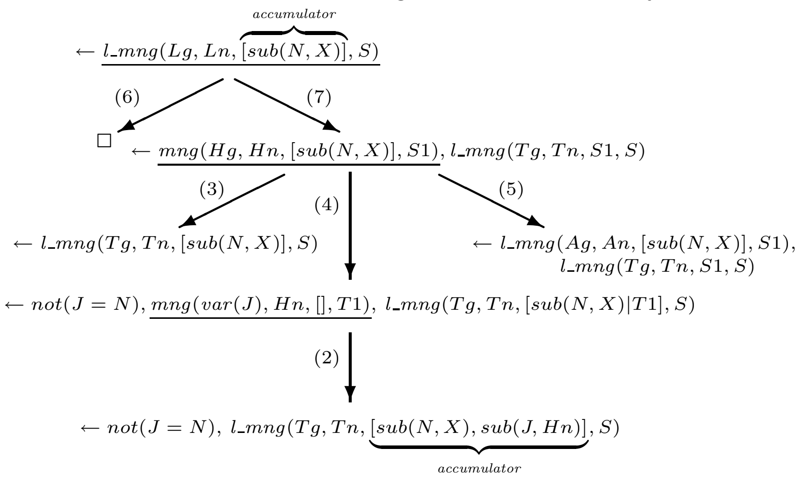

accumulator
← l_mng(Lg, Ln, [sub(N, X)], S)
(6)
(7)
□
← mng(Hg, Hn, [sub(N, X)], S1), l_mng(Tg, Tn, S1, S)
(3)
(4)
(5)
← l_mng(Tg, Tn, [sub(N, X)], S)
← l_mng(Ag, An, [sub(N, X)], S1), l_mng(Tg, Tn, S1, S)
← not(J = N), mng(var(J), Hn, [], T1), l_mng(Tg, Tn, [sub(N, X)|T1], S)
(2)
← not(J = N), l_mng(Tg, Tn, [sub(N, X), sub(J, Hn)], S)
accumulator

Fig. 6. Accumulator growth in Example 4.1.2.

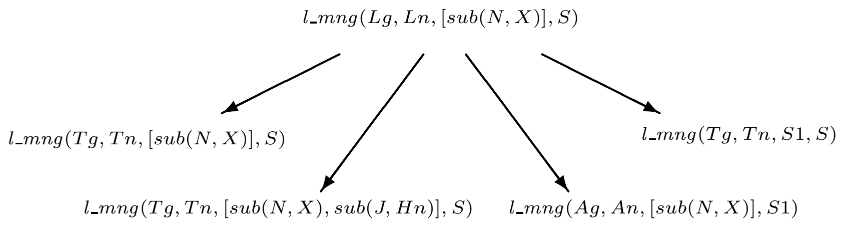

l_mng(Lg, Ln, [sub(N, X)], S)
l_mng(Tg, Tn, [sub(N, X)], S)
l_mng(Tg, Tn, S1, S)
l_mng(Tg, Tn, [sub(N, X), sub(J, Hn)], S)
l_mng(Ag, An, [sub(N, X)], S1)

Fig. 7. Initial section of a global tree for Example 4.1.2 and the unfolding of Figure 6.

and again, as in Example 4.1.1, we obtain nontermination without a depth bound, and very unsatisfactory ad hoc specializations with it.

Summarizing, computations influenced by one or more growing data structures are not so rare after all, and they cause ad hoc behavior of partial deduction, where the global control is founded on characteristic trees with a depth bound. In the next section, we show how this depth bound can be removed without endangering termination.

4.2 Partial Deduction Using Global Trees

A general framework for global control, not relying on any depth bounds, is proposed by Martens and Gallagher [1995]. 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 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 turn for inspiration on how to solve the depth bound problem described in Section 4.1.

First, we have chosen to use the term "global tree" rather than "marked tree" in the present article, because it better indicates its functionality. Moreover, global trees rely on a well-quasi-order (or well-quasi-relation) 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.[8]

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 generalization, resulting in unacceptable specialization losses. 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 in Example 4.1.1—it seems reasonable to stop expanding the global tree, generalize the offending atoms, and produce a specialized procedure for the generalization instead.

However, a closer look at the following variation of Example 4.1.2 shows that also this approach would sometimes overgeneralize and consequently fall short of providing sufficiently detailed polyvariance.

Example 4.2.1. Reconsider the program in Figure 5, and suppose that determinate unfolding is used for the local control. Take as starting point for partial deduction $A = mng(G, struct(cl, [struct(f, [X, Y])|B]), [], S)$. When unfolding A (see Figure 8), we obtain an SLD-tree containing the atom $mng(H, struct(f, [X, Y]), [], S1)$ 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 a generalization executed. However, the atoms in the leaves of the second tree are more general than those already met, and simply continuing partial deduction without generalization will lead to natural termination without any depth bound intervention.

Example 4.2.1 demonstrates that only measuring growth of characteristic trees, even in a refined way, does not always lead to satisfactory specialization. In fact, whenever the unfolding rule does not unfold "as deeply" as would be possible using a refined termination relation (for whatever reason, e.g., efficiency of the specialized program or because the unfolding rule is not refined enough), then a growing characteristic tree might simply be caused by splitting the "maximally deep tree" (i.e., the one constructed using a refined termination relation) in such a way that the second part "contains" the first part. Indeed, in Example 4.2.1, 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 *scrutinize entire characteris-*

[8]The global tree in Figure 6 also contains two nodes which are variants of their parent node. Usually, atoms which are variants of one of their ancestor nodes will not be added to the global tree, as they do not give rise to further specialization.

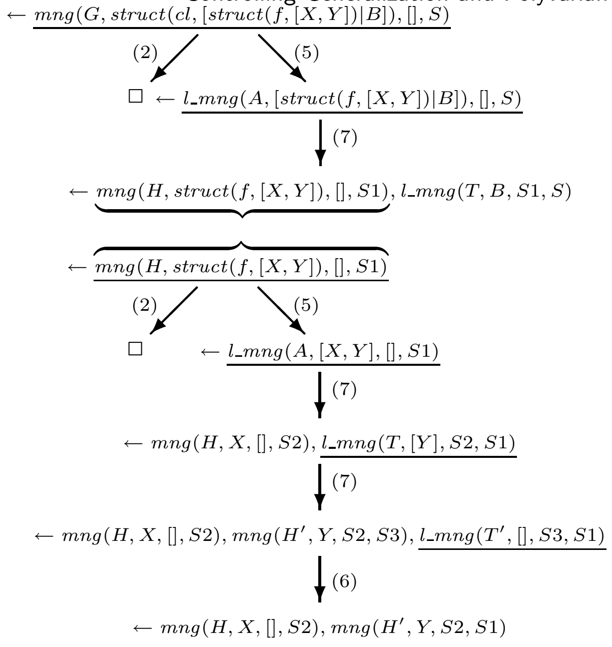

← mng(G, struct(cl, [struct(f, [X, Y])|B]), [], S)
(2)
(5)
□ ← l_mng(A, [struct(f, [X, Y])|B]), [], S)
(7)
← mng(H, struct(f, [X, Y]), [], S1), l_mng(T, B, S1, S)
← mng(H, struct(f, [X, Y]), [], S1)
(2)
(5)
□ ← l_mng(A, [X, Y], [], S1)
(7)
← mng(H, X, [], S2), l_mng(T, [Y], S2, S1)
(7)
← mng(H, X, [], S2), mng(H′, Y, S2, S3), l_mng(T′, [], S3, S1)
(6)
← mng(H, X, [], S2), mng(H′, Y, S2, S1)

Fig. 8. SLD-trees for Example 4.2.1.

tic 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 labeled by plain atoms as in the work of Martens and Gallagher [1995], but by entire characteristic atoms. A growing of a characteristic tree not coupled with a growing of the syntactic structure then indicates a fake growing caused by a conservative unfolding rule.

The rest of this section contains the formal elaboration of this new approach.

4.3 Generalizing Characteristic Atoms

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

Henceforth, we will use symbols like $\prec$ and $\succ$ (possibly annotated by some subscript) to refer to strict partial orders (antisymmetric, antireflexive, and transitive binary relations) and $\preceq$ and $\succeq$ to refer to quasi orders (reflexive and transitive binary relations). We will use either "directionality" as is convenient in the context. Given a quasi order $\preceq$, we will use the associated equivalence relation $\equiv$ and strict partial order $\prec$.

For ordinary atoms, $A_1 \preceq A_2$ will denote that A_1 is more general than A_2. We define a similar notion for characteristic atoms along with an operator to compute the most specific generalization. In a first attempt one might use the concretization function to that end, i.e., one could stipulate that (A, τ_A) is more general than (B, τ_B) if and only if $\gamma_P(A, \tau_A) \supseteq \gamma_P(B, \tau_B)$. The problem with this definition, from a practical point of view, is that this notion is undecidable in general, and a most specific generalization is uncomputable. For instance, $\gamma_P(A, \tau) = \gamma_P(A, \tau \cup \{\delta\})$ holds if and only if for every instance of A, the last goal associated with δ fails finitely. 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 generalization can be easily computed and which has some nice properties in the context of a partial deduction algorithm (e.g., see Definition 4.4.8).

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

Definition 4.3.1 ($\preceq_\tau$). Let τ_1, τ_2 be characteristic trees. We say that τ_1 is *more general* than τ_2, and we denote this by $\tau_1 \preceq_\tau \tau_2$, if and only if

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

Note that $\preceq_\tau$ is a quasi 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 subtrees to the leaves of τ_1.

Example 4.3.2. Given $\tau_1 = \{\langle 1:3\rangle\}$, $\tau_2 = \{\langle 1:3, 2:4\rangle\}$, and $\tau_3 = \{\langle 1:3\rangle, \langle 1:4\rangle\}$ we have that $\tau_1 \preceq_\tau \tau_2$ and even $\tau_1 \prec_\tau \tau_2$, 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 antisymmetry and transitivity of the order relation on characteristic trees.

LEMMA 4.3.3. *Let τ_1, τ_2 be two characteristic trees. Then $\tau_1 \equiv_\tau \tau_2$ if and only if $\tau_1 = \tau_2$.*

PROOF. See electronic Appendix D □

LEMMA 4.3.4. *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 4.3.1. □

We now present an algorithm to generalize two characteristic trees by computing the common initial subtree. We will later prove that this algorithm calculates the most specific generalization. The following notations will be useful in formalizing the algorithm.

Definition 4.3.5. Let τ be a characteristic tree and δ a characteristic path. We then define the notations $\tau \downarrow \delta = \{\gamma \mid \delta\gamma \in \tau\}$ and $\mathit{top}(\tau) = \{l:m \mid \langle l:m\rangle \in \mathit{prefix}(\tau)\}$.

Note that, for a nonempty characteristic tree τ, $\mathit{top}(\tau) = \emptyset \Leftrightarrow \tau = \{\langle\rangle\}$.

Algorithm 4.3.6 (*msg of Characteristic Trees*).

```
**Input:** two nonempty characteristic trees $\tau_A$ and $\tau_B$
**Output:** the *msg* $\tau$ of $\tau_A$ and $\tau_B$
**Initialization:** $i := 0$; $\tau_0 := \{\langle\rangle\}$;
**while** $\exists\delta \in \tau_i$ such that $top(\tau_A \downarrow \delta) = top(\tau_B \downarrow \delta) \neq \emptyset$ **do**
```

```
    $\tau_{i+1} := (\tau_i \setminus \{\delta\}) \cup \{\delta\langle l : m\rangle \mid l : m \in top(\tau_A \downarrow \delta)\}$;   $i := i + 1$;
**end while**
**return** $\tau = \tau_i$
```

Example 4.3.7. Take the characteristic trees $\tau_A = \{\langle 1:1, 1:2\rangle\}$ and $\tau_B = \{\langle 1:1, 1:2\rangle, \langle 1:1, 1:3\rangle\}$. Then Algorithm 4.3.6 proceeds as follows:

(1) $\tau_0 = \{\langle\rangle\}$,
(2) $\tau_1 = \{\langle 1:1\rangle\}$ as for $\langle\rangle \in \tau_0$ we have $top(\tau_A \downarrow \langle\rangle) = top(\tau_B \downarrow \langle\rangle) = \{1:1\}$.
(3) $\tau = \tau_1$ as $top(\tau_A \downarrow \langle 1:1\rangle) = \{1:2\}$ and $top(\tau_B \downarrow \langle 1:1\rangle) = \{1:2, 1:3\}$ and the while loop terminates.

LEMMA 4.3.8. *Algorithm 4.3.6 terminates and produces as output a characteristic tree τ such that if $chtree(G, P, U) = \tau_A$ (respectively τ_B), then for some U', $chtree(G, P, U') = \tau$. The same holds for any τ_i arising during the execution of Algorithm 4.3.6.*

PROOF. See electronic Appendix E. □

PROPOSITION 4.3.9. *Let τ_A, τ_B be two nonempty characteristic trees. Then the output τ of Algorithm 4.3.6 is the unique most specific generalization of τ_A and τ_B.*

For $\tau_A \neq \emptyset$ and $\tau_B \neq \emptyset$ we denote by $msg(\tau_A, \tau_B)$ the output of Algorithm 4.3.6. If both $\tau_A = \emptyset$ and $\tau_B = \emptyset$ then $\emptyset$ is the unique most specific generalization, 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 4.3.10. Given $\tau_1 = \{\langle 1:3\rangle\}$, $\tau_2 = \{\langle 1:3, 2:4\rangle\}$, $\tau_3 = \{\langle 1:3\rangle, \langle 1:4\rangle\}$, $\tau_4 = \{\langle 1:3, 2:4\rangle, \langle 1:3, 2:5\rangle\}$, we have that $msg(\tau_1, \tau_2) = \tau_1$, $msg(\tau_1, \tau_3) = msg(\tau_2, \tau_3) = \{\langle\rangle\}$, and $msg(\tau_2, \tau_4) = \tau_1$.

Lemma 4.3.8 and Proposition 4.3.9 can also be used to prove an interesting property about the $\preceq_\tau$ relation.

COROLLARY 4.3.11. *Let τ_1 and τ_2 be characteristic trees such that $\tau_1 \preceq_\tau \tau_2$. If $chtree(G, P, U) = \tau_2$ then for some U', $chtree(G, P, U') = \tau_1$.*

PROOF. First we have that Algorithm 4.3.6 will produce for τ_1 and τ_2 the output $\tau = \tau_1$ (because Algorithm 4.3.6 computes the most specific generalization by Proposition 4.3.9). Hence we have the desired property by Lemma 4.3.8. □

Definition 4.3.12 ($\preceq_{ca}$). A characteristic atom (A_1, τ_1) is *more general* than another characteristic atom (A_2, τ_2), denoted by $(A_1, \tau_1) \preceq_{ca} (A_2, \tau_2)$, if and only if $A_1 \preceq A_2$ and $\tau_1 \preceq_\tau \tau_2$. (A_1, τ_1) is said to be a *variant* of (A_2, τ_2) if and only if $(A_1, \tau_1) \equiv_{ca} (A_2, \tau_2)$.

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

PROPOSITION 4.3.13. *Let (A, τ_A) and (B, τ_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. See electronic Appendix D. □

The converse of the above proposition does of course not hold. Take the *member* program from Example 2.3.4, and let $A = member(a,[a])$, $\tau = \{\langle 1:1\rangle, \langle 1:2\rangle\}$ and $\tau' = \{\langle 1:1\rangle\}$. Then $\gamma_P(A,\tau) = \gamma_P(A,\tau') = \{member(a,[a])\}$ (because the resolvent $\leftarrow member(a,[])$ associated with $\langle 1:2\rangle$ 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 Proposition 4.3.13.

COROLLARY 4.3.14. *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 generalization* (*msg*) to characteristic atoms:

Definition 4.3.15. Let (A_1,τ_1), (A_2,τ_2) be two characteristic atoms such that their *msg* 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 indeed a most specific generalization (because $msg(A_1,A_2)$ and $msg(\tau_1,\tau_2)$ are most specific generalizations for the atom and characteristic tree parts respectively) and 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.

4.4 Well-Quasi Ordering of Characteristic Atoms.

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

Definition 4.4.1. A relation $\leq_V$ on V is called a *well-quasi relation* (WQR) *on V* if and only if for any infinite sequence of elements $e_1, e_2, \ldots$ in V there are $i < j$ such that $e_i \leq_V e_j$. A well-quasi relation $\leq_V$ is also called a *well-quasi order* (WQO) *on V* if it is a quasi order.

An interesting WQO is the homeomorphic embedding relation $\trianglelefteq$. It has been adapted from the work of Dershowitz [1987] and Dershowitz and Jouannaud [1990], where it is used in the context of term-rewriting systems, for use in supercompilation by Sørensen and Glück [1995]. Its usefulness as a stop criterion for partial evaluation is also discussed and advocated by Marlet [1994]. Some complexity results can be found in the works of Stillman [1988] and Gustedt [1992] (also summarized by Marlet [1994]).

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.

The following is the definition of Sørensen and Glück [1995], which adapts the pure homeomorphic embedding of Dershowitz and Jouannaud [1990] by adding a rudimentary treatment of variables.

Definition 4.4.2 ($\trianglelefteq$). The (*pure*) *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 4.4.3. We have that $p(a) \trianglelefteq p(f(a))$, $X \trianglelefteq X$, $p(X) \trianglelefteq p(f(Y))$, $p(X, X) \trianglelefteq p(X, Y)$, and $p(X, Y) \trianglelefteq p(X, X)$.

Proposition 4.4.4. *The relation $\trianglelefteq$ is a WQO on the set of expressions over a finite alphabet.*

Proof. See electronic Appendix D. □

The intuition behind Definition 4.4.2 is that when some structure reappears within a larger one, it is homeomorphically embedded by the latter. As is argued by Marlet [1994] and Sørensen and Glück [1995], this provides a good starting point for detecting growing structures created by possibly nonterminating processes.

However, as can be observed in Example 4.4.3, the homeomorphic embedding relation $\trianglelefteq$ as defined in Definition 4.4.2 is rather crude with respect to variables. In fact, all variables are treated as if they were the same variable, a practice which is 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. Indeed $p(X, X)$ 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.

To remedy the problem (as well as another one related to the *msg* which we discuss later), we refine the above introduced homeomorphic embedding as follows:

Definition 4.4.5 ($\trianglelefteq^*$). Let A, B be expressions. Then B (*strictly homeomorphically*) *embeds* A, written as $A \trianglelefteq^* B$, if and only if $A \trianglelefteq B$ and A is not a strict instance of B.

Example 4.4.6. We now still 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 $X \trianglelefteq^* X$.

An alternate approach might be based on numbering variables using some mapping $\#(.)$ and then stipulating that $X \trianglelefteq^{\#} Y$ if and only if $\#(X) \leq \#(Y)$. For instance Marlet [1994] proposes a de Bruijn numbering of the variables. Such an approach, however, has a somewhat ad hoc flavor to it. Take for instance the terms $p(X, Y, X)$, and $p(X, Y, Y)$. Neither term is an instance of the other and we thus have $p(X, Y, X) \trianglelefteq^* p(X, Y, Y)$ and $p(X, Y, Y) \trianglelefteq^* p(X, Y, X)$. Depending on the particular numbering we will have either $p(X, Y, X) \not\trianglelefteq^{\#} p(X, Y, Y)$ or $p(X, Y, Y) \not\trianglelefteq^{\#} p(X, Y, X)$, while there is no apparent reason why one expression should be considered smaller than the other.[9]

[9]Marlet [1994] also proposes to consider all possible numberings, but (leading to $n!$ complexity, where n is the number of variables in the terms to be compared). It is unclear how such a relation compares to $\trianglelefteq^*$ of Definition 4.4.5.

THEOREM 4.4.7. *The relation $\trianglelefteq^*$ is a well-quasi relation on the set of expressions over a finite alphabet.*

PROOF. See electronic Appendix F. □

Observe that $\trianglelefteq^*$ is not a WQO because it is not transitive. We now extend the embedding relation of Definition 4.4.5 to characteristic atoms. Notice that the relation $\preceq_\tau$ is not a WQR on characteristic trees, even in the context of a given fixed program P. Take for example 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 WQR is to first define a term representation of characteristic trees and then apply the embedding relation $\trianglelefteq^*$ to this term representation.

Definition 4.4.8 ($\lceil.\rceil$). 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 conditions of Definition 4.4.8 are essential for the termination of a Algorithm 4.6.1.

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

PROPOSITION 4.4.9. *A function $\lceil.\rceil$ satisfying Definition 4.4.8 exists.*

PROOF. See electronic Appendix G. □

From now on, we fix $\lceil.\rceil$ to be a particular mapping satisfying Definition 4.4.8. The mapping developed in Appendix G is actually a good candidate, as it has the desirable property that the structure of a characteristic tree τ is reflected in the treestructure of the term $\lceil\tau\rceil$, thus ensuring that the usefulness of $\trianglelefteq$ for spotting nonterminating processes (e.g., see Marlet [1994]) carries over to characteristic trees.

Definition 4.4.10 ($\trianglelefteq^*_{ca}$). 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 $(A_1, \tau_1) \trianglelefteq^*_{ca} (A_2, \tau_2)$, if and only if $A_1 \trianglelefteq^* A_2$ and $\lceil\tau_1\rceil \trianglelefteq^* \lceil\tau_2\rceil$.

PROPOSITION 4.4.11. *Let $\tilde{\mathcal{A}}$ be a set of P-characteristic atoms. Then $\trianglelefteq^*_{ca}$ is a well-quasi relation on $\tilde{\mathcal{A}}$.*

PROOF. See electronic Appendix D. □

4.5 Global Trees and Characteristic Atoms

In this section, we adapt and instantiate the m-tree concept presented by Martens and Gallagher [1995] according to our particular needs in this article.

Definition 4.5.1 (*Global Tree*). A *global tree* γ_P for a program P is a (finitely branching) tree where nodes can be either *marked* or *unmarked*, and where each node carries a label which is a P-characteristic atom.

In other words, a node in a global tree γ_P looks 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 we subsequently prove that it provides a sufficient condition for finiteness. Let γ_P be a global tree. Then we henceforth denote as Lbl_{γ_P} the set of its labels. And for a given node n in a tree γ, we refer to its label by lbl_n.

Definition 4.5.2 (*Label Mapping*). Let γ be a global tree. Then we define its *associated label mapping* f_γ as the mapping $f_\gamma : (\gamma, >_\gamma) \rightarrow (Lbl_\gamma, \trianglelefteq^*_{ca})$ such that $n \mapsto lbl_n$. f_γ will be called *quasi-monotonic* if and only if $\forall n_1, n_2 \; n_1 >_\gamma n_2 \Rightarrow lbl_{n_1} \not\trianglelefteq^*_{ca} lbl_{n_2}$.

Definition 4.5.3. A global tree γ is *well-quasi-ordered* if f_γ is quasi-monotonic.

THEOREM 4.5.4. *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 $lbl_{n_1}, lbl_{n_2}, \ldots \in Lbl_\gamma, \trianglelefteq^*_{ca}$. From Proposition 4.4.11, we know that $\trianglelefteq^*_{ca}$ is WQR on Lbl_γ, and therefore there must exist $lbl_{n_i}, lbl_{n_j}, i < j$ in the above-mentioned sequence such that $lbl_{n_i} \trianglelefteq^*_{ca} lbl_{n_j}$. But this implies that f_γ is not quasi-monotonic. □

4.6 A Tree-Based Algorithm

We now present the actual refined partial deduction algorithm where global control is imposed through characteristic atoms in a global tree.

Please note that the algorithm is parametrized by an unfolding rule U, thus leaving the particulars of the local control unspecified. As for Algorithm 3.4.3, we need the notation $chatom(A, P, U)$ (see Definition 3.4.2). Also, without loss of generality, we suppose that the initial goal contains just a single atom (otherwise we get a global forest instead of a global tree).

Algorithm 4.6.1 (*Partial Deduction with Global Trees*).

```
**Input:** a normal program $P$ and goal $\leftarrow A$
**Output:** a set of characteristic atoms $\tilde{\mathcal{A}}$
**Initialization:** $\gamma := \{(1, u, (A, \tau_A))\}$;
**while** $\gamma$ contains an unmarked leaf **do**
  let $n$ be such an unmarked leaf in $\gamma$: $(n, u, (A_n, \tau_{A_n}))$;
  mark $n$;
  $\tilde{\mathcal{B}} := \{chatom(B, P, U) \mid B \in leaves_P(A_n, \tau_{A_n})\}$;
  **while** $\tilde{\mathcal{B}} \neq \emptyset$ **do**
    select $CA_B \in \tilde{\mathcal{B}}$;  remove $CA_B$ from $\tilde{\mathcal{B}}$;
    **if** $\tilde{\mathcal{H}} = \{CA_C \in Anc_\gamma(n) \mid CA_C \trianglelefteq^*_{ca} CA_B\} = \emptyset$ **then**
```

```
            add $(n_B, u, CA_B)$ to $\gamma$ as a child of $n$;
        **else if** $\nexists CA_D \in Lbl_\gamma$ such that $CA_D \equiv_{ca} CA_B$ **then**
            add $msg(\tilde{\mathcal{H}} \cup \{CA_B\})$ to $\tilde{\mathcal{B}}$;
    **end while**
**end while**
**return** $\tilde{\mathcal{A}} := Lbl_\gamma$
```

As in, for example, Gallagher [1993] and Martens and Gallagher [1995] (but unlike Algorithm 3.4.3), Algorithm 4.6.1 does not output a specialized 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 inner **while**-loop. In $\tilde{\mathcal{B}}$, all the characteristic atoms are assembled, corresponding to the atoms occurring in the leaves of the SLDNF-tree built for A_n according to τ_{A_n}. Elements of $\tilde{\mathcal{B}}$ 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 another node in γ, then it is simply not added to γ. (Note that one can change to an instance test by simply replacing $\equiv_{ca}$ by $\preceq_{ca}$.) Finally, if a characteristic atom $CA_B \in \tilde{\mathcal{B}}$ does embed an ancestor label, but there is no variant to be found in γ, then the most specific generalization M of CA_B and of all embedded ancestor labels $\tilde{\mathcal{H}}$ is reinserted into $\tilde{\mathcal{B}}$. The latter case is of course the most interesting: simply adding a node labeled 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 4.4.8) and trying to add it instead secures finiteness, as proven below, while trying to preserve as much information as seems possible (see however Sections 4.8 and 5).

We obtain the following theorems:

THEOREM 4.6.2. *Algorithm 4.6.1 always terminates.*

PROOF. Upon each iteration of the outer **while**-loop in Algorithm 4.6.1, exactly one node in γ is marked, and zero or more (unmarked) nodes are added to γ. Moreover, Algorithm 4.6.1 never deletes a node from γ, neither does it ever "unmark" a marked node. Hence, since all branchings are finite, nontermination of the outer **while**-loop must result in the construction of an infinite branch. It is therefore sufficient to argue that the inner **while**-loop terminates and that after every iteration of the outer loop, γ is a WQO global tree.

First, this holds after initialization. Also, obviously, a global tree will be converted into a new global tree through the outer **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 inner **while**-loop, at most one for every element of $\tilde{\mathcal{B}}$. So, let us take an arbitrary characteristic atom $CA_B \in \tilde{\mathcal{B}}$; 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 of the tree γ. 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 no variants (or more general characteristic atoms respectively) are to be found in γ. We then calculate the *msg* M of CA_B and all[10] the labels $\tilde{\mathcal{H}} = \{L_1, \dots, L_k\}$ 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 $\tilde{\mathcal{H}}$ (by property of the *msg*), and even strictly more general because no label 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[11] of $\lceil . \rceil$, that τ_i is not a strict instance of τ_B, and thus by Definition 4.3.12 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 $\tilde{\mathcal{B}}$ by a strictly more general one, which we can do only finitely many times,[12] and thus termination of the inner **while**-loop is ensured (as in the other two cases above an element is removed from $\mathcal{B}$ and none are added). Note that, when using the $\trianglelefteq$ relation instead of $\trianglelefteq^*$, M would not necessarily be more general than CA_B, i.e., the algorithm could loop. For example, take a global tree having the single node $(p(X,X), \tau)$ and where we try to add $\tilde{\mathcal{B}} = \{(p(X,Y), \tau)\}$. Now we have that $p(X,X) \trianglelefteq p(X,Y)$, and we calculate $msg(\{(p(X,X),\tau), (p(X,Y),\tau)\}) = (p(X,Y), \tau)$.

We have thus established termination of Algorithm 4.6.1. □

THEOREM 4.6.3. *Let P be a program, input to Algorithm 4.6.1, and let $\tilde{\mathcal{A}}$ be the corresponding set of characteristic atoms produced as output. Then $\tilde{\mathcal{A}}$ is P-covered.*

PROOF. First, it is straightforward to prove that throughout the execution of Algorithm 4.6.1, any unmarked node in γ must be a leaf. It therefore suffices to show, because the output contains only marked nodes, that after each iteration of the outer **while**-loop, only unmarked leaves in γ possibly carry a noncovered label.[13] Trivially, this property holds after initialization. Now, in the outer **while**-loop, one unmarked leaf n is selected and marked. The inner **while**-loop then precisely proceeds to incorporate (unmarked leaf) nodes into γ such that all leaf atoms of n's label are concretizations of at least one label in the new, extended γ. □

The correctness of the specialization now follows from Theorem 3.3.2.

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

[11]Note that the proof 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 $\tau_i \not\trianglelefteq^* \tau_B$.

[12]See for example Lemma C.2 in electronic Appendix C.

[13]That is, a label with at least one leaf atom (in P) that is not a concretization of any label in γ.

4.7 Further Discussion

One might 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 4.1.1 we observe that $rev(L, [], R) \trianglelefteq^* rev(T, [H], R)$ and, indeed, for the corresponding characteristic trees $\lceil\{\langle 1:1\rangle, \langle 1:2, 1:3\rangle\}\rceil \trianglelefteq^* \lceil\{\langle 1:1\rangle, \langle 1:2, 1:4, 1:3\rangle\}\rceil$ holds. Nonetheless, the intuition turns out to be incorrect. The following examples illustrate this point.

Example 4.7.1. 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 let U be an unfolding rule based on $\trianglelefteq^*$ (i.e., only allow the selection of an atom if it does not embed a covering ancestor). The SLDNF-tree accordingly built for $\leftarrow A$ is depicted in Figure 9. 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 $\tau_A = \{\langle 1:1, 1:member, 1:3\rangle, \langle 1:2, 1:member, 1:3, 1:1, 1:member, 1:4\rangle, \langle 1:2, 1:member, 1:3, 1:2, 1:member, 1:4\rangle\}$ is not embedded in (the representation of) $\tau_B = \emptyset$, and no danger for nontermination 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.

Example 4.7.2. Let P be the following *definite* program:

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

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

—$\tau_A = \{\langle 1:1\rangle, \langle 1:2, 1:3\rangle\}$, and
—$\tau_B = \{\langle 1:1\rangle\}$.

Again, $A \trianglelefteq^* B$ holds, but not $\lceil\tau_A\rceil \trianglelefteq^* \lceil\tau_B\rceil$.

In recent experiments, it also turned out that characteristic trees might be a vital asset when trying to solve the *parsing problem* [Martens 1994], which appears when

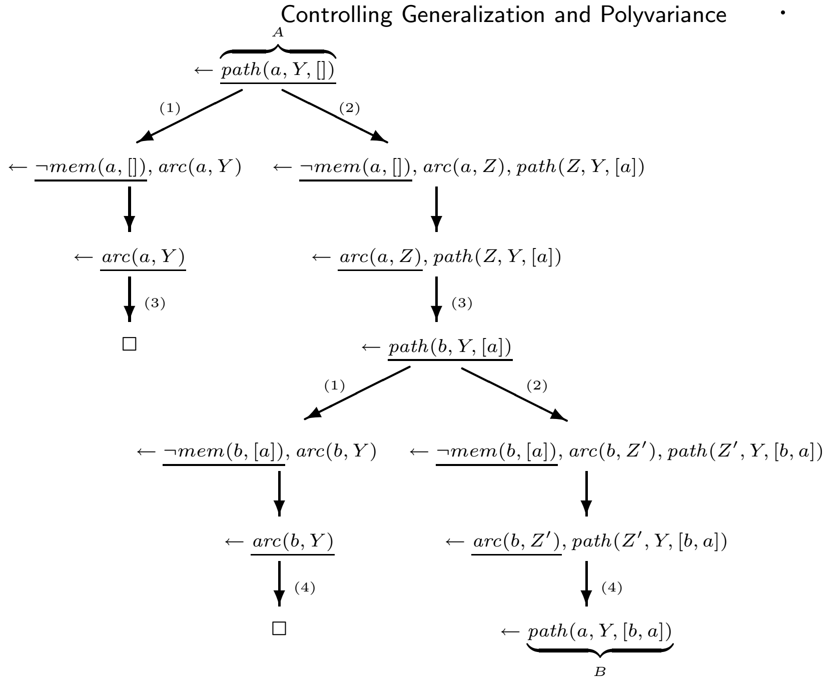

A
$\leftarrow \underline{path(a, Y, [])}$
(1)
(2)
$\leftarrow \underline{\neg mem(a, [])}, arc(a, Y)$
$\leftarrow \underline{\neg mem(a, [])}, arc(a, Z), path(Z, Y, [a])$
$\leftarrow \underline{arc(a, Y)}$
$\leftarrow \underline{arc(a, Z)}, path(Z, Y, [a])$
(3)
(3)
$\square$
$\leftarrow \underline{path(b, Y, [a])}$
(1)
(2)
$\leftarrow \underline{\neg mem(b, [a])}, arc(b, Y)$
$\leftarrow \underline{\neg mem(b, [a])}, arc(b, Z'), path(Z', Y, [b, a])$
$\leftarrow \underline{arc(b, Y)}$
$\leftarrow \underline{arc(b, Z')}, path(Z', Y, [b, a])$
(4)
(4)
$\square$
$\leftarrow \underline{path(a, Y, [b, a])}$
B

Fig. 9. SLDNF-tree for Example 4.7.1.

unfolding metainterpreters with nontrivial object programs. In such a setting a growing of the syntactic structure also does not imply a growing of the characteristic tree.

Example 4.7.3. Take the vanilla metainterpreter with a simple family database at the object level:

(1) $solve(empty) \leftarrow$
(2) $solve(A\&B) \leftarrow solve(A), solve(B)$
(3) $solve(A) \leftarrow clause(A, B), solve(B)$
(4) $clause(anc(X, Y), parent(X, Y)) \leftarrow$
(5) $clause(anc(X, Z), parent(X, Y)\&anc(Y, Z)) \leftarrow$
(6) $clause(parent(peter, paul), empty) \leftarrow$
(7) $clause(parent(paul, mary), empty) \leftarrow$

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 generalized (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 generalization will occur. See also Vanhoof and Martens [1997].

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 relation or order 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 Martens and De Schreye [1996]). 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 in Section 5 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 Gallagher [1991] was extended by de Waal [1994] by adorning characteristic trees with a depth-k abstraction of the corresponding atom component. This was done in order to increase the amount of polyvariance so that a (monovariant) postprocessing 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 4.8 postprocessing) 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 specialization behavior. 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 *normalize* 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 article remain valid, independently of whether this normalization is applied or not. A similar effect can be obtained, in the context of definite programs, via the *trace terms* of Gallagher and Lafave [1996].

Algorithm 4.6.1 can also be seen as performing an abstract interpretation on an *infinite domain* of *infinite height* (i.e., the ascending chain condition of Cousot and Cousot [1992] 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 Bruynooghe [1991], Janssens and Bruynooghe [1992], and Heintze [1992]), and to our knowledge all of them have some a priori limitation of the precision, at least in practice. An adaptation of Algorithm 4.6.1, with its non ad hoc termination and precise generalizations, may provide a good starting point to introduce similar features into abstract interpretation methods, where they can prove equally beneficial.

Next, there is an interesting relation between the control techniques presented in this article and current practice in supercompilation [Glück and Sørensen 1996; Sørensen and Glück 1995; Turchin 1986; Turchin 1988]. We already pointed out that the inspiration for using $\trianglelefteq$ derives from Marlet [1994] and Sørensen and Glück [1995]. In the latter, a generalization 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 originating from Glück and Klimov

[1993] and corresponding to 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 by Sørensen and Glück [1995]. It is also interesting to return to an observation already made in Section 4 of Martens and Gallagher [1995]: neighborhoods of order "n," forming the basis for generalization in full supercompilation [Turchin 1988], are essentially the same as classes of atoms or goals with an identical depth-n characteristic tree. Adapting our technique to the supercompilation setting will therefore allow us to remove the depth bound on neighborhoods.

Finally, we conjecture that the techniques presented in this article can fruitfully be adapted to deal with program specialization and transformation in various other settings. Indeed, partial evaluation of functional-logic programs [Alpuente et al. 1996; 1997; Lafave and Gallagher 1997] already features the use of homeomorphic embedding and m-trees [Martens and Gallagher 1995]. On the other hand, the generalization strategy as well as the generation of polyvariance remains entirely based on *syntactic* structure of the manipulated expressions. Taking into account *computational* behavior, as through characteristic trees or neighborhoods, will probably allow significant improvements.

Similarly, on-line[14] partial evaluation of full Prolog programs [Prestwich 1992b; Sahlin 1991; Sahlin 1993], constraint logic programs [Smith 1991; Smith and Hickey 1990], functional programs [Sperber 1996; Weise et al. 1991], and indeed program transformation in general [Bossi et al. 1990; Burstall and Darlington 1977; Futamura et al. 1991; Pettorossi and Proietti 1994; Wadler 1990] is likely to benefit from adapted versions of our techniques for ensuring termination, performing generalization, and determining polyvariance. A further investigation of these conjectures will be the subject of future work. We can, however, already point out that our methods were successfully adapted to automatically control the recently developed *conjunctive* partial deduction (see Section 6).

4.8 Further Improvements

Unlike ecological partial deduction as presented in Section 3.4, Algorithm 4.6.1 will obviously often output several characteristic atoms with the same characteristic tree (each giving rise to a different specialized 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: generalizing 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.

[14]Part of our techniques might also be useful in an off-line setting. For instance, one can imagine off-line partial evaluation which memoizes not (only) syntactic structure but (also) the computational behavior of expressions (see Gallagher and Lafave [1996] for a first step in that direction).

The polyvariance-reducing postprocessing that we propose in this section therefore avoids the latter phenomenon. In order to obtain the desired effect, it basically collapses and generalizes several characteristic atoms with the same characteristic tree only if this does not modify the global specialization. To that end we number the leaf atoms of each characteristic atom and then label the arcs of the global tree with the number of the leaf atom it refers to. We also add arcs in case a leaf atom is a variant of another characteristic atom in the tree and has therefore not been lifted to the global level. We thus obtain a *labeled global graph*. We then try to collapse nodes with identical characteristic trees using the well-known algorithm for minimization of finite-state automata [Aho et al. 1986; Hopcroft and Ullman 1979]: we start by putting all characteristic atoms with the same characteristic tree into the same class, and subsequently split these classes if corresponding leaf atoms fall into different classes. As stated by Hopcroft and Ullman [1979], 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 is the number of symbols (in our case the maximum number of leaf atoms).

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

Example 4.8.1. Let us return to the *member* program of Example 2.3.4, augmented with one additional clause:

(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 4.6.1 we obtain the following set of characteristic atoms:

$$\tilde{\mathcal{A}} = \{(member(b, L), \tau),\ (member(a, [a, b, c, d|T]), \tau),$$
$$(member(a, [b, c, d|T]), \tau'),\ (member(a, L), \tau),\ (t(T), \{\langle 1:3\rangle\})\}$$

where $\tau = \{\langle 1 : 1\rangle, \langle 1 : 2\rangle\}$ and $\tau' = \{\langle 1 : 2, 1 : 2, 1 : 2\rangle\}$. Depending on the particular renaming function, a partial deduction of P with respect to $\tilde{\mathcal{A}}$ 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 labeled graph version of the corresponding global tree can be found in Figure 10. Adapting the algorithm from Aho et al. [1986] and Hopcroft and Ullman [1979] to our needs, we start out by generating three classes (one for each characteristic tree) of states:

$$\begin{tikzcd}
(\mathit{member}(a,[a,b,c,d|T]),\tau) \arrow[d, "\#1"] & (t(T),\{\langle 1:3\rangle\}) \arrow[l, "\#1"] \arrow[dd, "\#2"] \\
(\mathit{member}(a,[b,c,d|T]),\tau') \arrow[d, "\#1"] & \\
(\mathit{member}(a,L),\tau) \arrow[loop below, "\#1"] & (\mathit{member}(b,L),\tau) \arrow[loop below, "\#1"]
\end{tikzcd}$$

Fig. 10. Labeled global graph of Example 4.8.1 before postprocessing.

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

The class C_1 must be further split, because the state $(\mathit{member}(a,[a,b,c,d|T]),\tau)$ has a transition via the label #1 to a state of C_2 while all the other states of C_1, $(\mathit{member}(a,L),\tau)$ and $(\mathit{member}(b,L),\tau)$, do not. This means that by actually collapsing all elements of C_1 we would lose subsequent specialization at the global level (namely, the pruning and precomputation that is performed within $(\mathit{member}(a,[b,c,d|T]),\tau')$).

We now obtain the following 4 classes:

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

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 specialization 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, minimized program is

$\mathit{mem}_{a,[a,b,c,d]}(T) \leftarrow$
$\mathit{mem}_{a,[a,b,c,d]}(T) \leftarrow \mathit{mem}_{a,[b,c,d]}(T)$
$\mathit{mem}_{a,[b,c,d]}(T) \leftarrow \mathit{mem}_x(a,T)$
$\mathit{mem}_x(X,[X|T]) \leftarrow$
$\mathit{mem}_x(X,[Y|T]) \leftarrow \mathit{mem}_x(X,T)$
$t(T) \leftarrow \mathit{mem}_{a,[a,b,c,d]}(T), \mathit{mem}_x(b,T)$

A similar use of this minimization algorithm is made by Winsborough [1992] and Puebla and Hermenegildo [1995]. The former aims at minimizing polyvariance in the context of multiple specialization by optimizing compilers. The latter, in a

$$\begin{tikzcd}
(member(a,[a,b,c,d|T]),\tau) \arrow[d, "\#1"] & (t(T),\{\langle 1:3\rangle\}) \arrow[l, "\#1"'] \arrow[ddl, "\#2"] \\
(member(a,[b,c,d|T]),\tau') \arrow[d, "\#1"] & \\
(member(X,L),\tau) \arrow[loop below, "\#1"] &
\end{tikzcd}$$

Fig. 11. Labeled global graph of Example 4.8.1 after postprocessing.

somewhat different context, studies polyvariant parallelization and specialization of logic programs based on abstract interpretation.

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 concretizations. Alternatively, one could try to use an altogether more accurate abstraction operator than taking an *msg* on characteristic atoms. For instance, one can endeavor to extend the constraint-based abstraction operator proposed by Leuschel and De Schreye [1998] 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 estimates into the global control, e.g., based on the works of Debray and Lin [1993], Debray et al. [1994], and Debray [1997], in order to analyze the trade-off between improved specialization and increased polyvariance and code size.

5. EXPERIMENTAL RESULTS

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.

The system which integrates the ideas of this article, called ECCE, is publicly available [Leuschel 1996] and is actually an implementation of a generic version of Algorithm 4.6.1, which allows the advanced user to change and even implement for example the unfolding rule as well as the abstraction operator and nontermination detection method. For instance, by adapting the settings of ECCE, one can obtain exactly Algorithm 4.6.1. But one can also simulate a (global tree-oriented) version of Algorithm 3.4.3 using depth bounds to ensure termination.

All unfolding rules of ECCE were complemented by simple *more specific resolution* steps in the style of SP [Gallagher 1991]. Constructive negation (see Chan and Wallace [1989] and Gurr [1994]) has not yet been incorporated, but the selection of ground negative literals is allowed. Postprocessing removal of unnecessary polyvariance, using the algorithm outlined in Section 4.8, determinate postunfolding, and redundant argument filtering (see Leuschel and Sørensen [1996]) were enabled throughout the experiments discussed below.

The ECCE system also handles a lot of Prolog built-ins, e.g., =, is, <, =<, <, >=, *number*, *atomic*, *call*, \==, \=. All built-ins are supposed to be declarative and their selection delayed until they are sufficiently instantiated. The method presented earlier is extended by also registering built-ins in the characteristic trees. One problematic aspect is that, when generalizing calls to built-ins which generate bindings (like is/2 or =../2 but unlike >/2 or </2) and which are no longer executable after generalization, these built-ins have to be removed from the generalized characteristic tree (i.e., they are no longer selected). With that, the concretization definition for characteristic atoms scales up, and the technique will ensure correct specialization. It should also be possible to incorporate the `if-then-else` into characteristic trees.

Also, the homeomorphic embedding relation $\trianglelefteq$ of Definition 4.4.2 (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 *static*, occurring in the original program and the partial deduction query, and *dynamic* ones. (This approach is also used by Sahlin [1991; Sahlin [1993].) The set of dynamic constants and functors is possibly infinite, and we will therefore treat it like the infinite set of variables in Definition 4.4.2 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

Some dynamic functors, which already have a natural WQR (or a well-founded order, which can be turned into a WQR; e.g., see Lemma F.1 in Appendix F) associated with them, might be treated in a more refined way. For instance for integers we can define

$i \trianglelefteq j$ if both i and j are integers and $i \leq j$.

An even more refined solution (not implemented for the current experiments) might be based on using the *general* homeomorphic embedding relation of Kruskal [1960], which can handle infinitely many function symbols provided that a WQO $\preceq$ on the function symbols is given.[15] Furthermore, as proven by Leuschel [1997b], it is possible to perform the "not strict instance test" for $\trianglelefteq^*$ of Definition 4.4.5 not just once at the top but *recursively* within the structure of the expressions, leading to the rule (as a replacement of rule (3) of Definition 4.4.2):

$f(s_1, \ldots, s_m) \trianglelefteq^* g(t_1, \ldots, t_n)$ if $f \preceq g$ and $\exists 1 \leq i_1 < \ldots i_m \leq n$ such that $\forall j \in \{1, \ldots, m\} : s_j \trianglelefteq^* t_{i_j}$ and $\langle s_1, \ldots, s_m \rangle$ is not a strict instance of $\langle t_1, \ldots, t_n \rangle$.

For further details we refer to Leuschel [1997b].

We present benchmarks using three different settings of ECCE, hereafter called ECCE-d, ECCE-x-10, and ECCE-x. The settings ECCE-d and ECCE-x use Algorithm 4.6.1, with a different unfolding rule, while ECCE-x-10 uses a (global

[15] A simple one might be $f \preceq g$ if both f and g are dynamic or if $f = g$.

tree-oriented) version of Algorithm 3.4.3 with a depth bound of 10 on characteristic trees to ensure termination. We also include results for MIXTUS [Sahlin 1991; Sahlin 1993], PADDY [Prestwich 1992a; Prestwich 1992b; Prestwich 1993] and SP [Gallagher 1991; Gallagher 1993], 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, 1995.

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

—*"MIXTUS-like" unfolding*: This is the unfolding strategy explained by Sahlin [1991; 1993], which in general unfolds deeply enough to solve the "fully unfoldable" problems,[16] 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 Martens and De Schreye [1996] which is free of such elements.) For instance, for ECCE-x and ECCE-x-10 we used the settings (see Sahlin [1993]) $max\_rec = 2$, $max\_depth = 2$, $maxfinite = 7$, $maxnondeterm = 10$ and only allowed nondeterminate unfolding when no user predicates were to the left of the selected literal. For MIXTUS and PADDY we used the respective default settings of the systems. Note that the "MIXTUS-like" unfolding strategies of ECCE, MIXTUS, and PADDY differ slightly from each other, probably due to some details not fully elaborated in Sahlin [1993] and Prestwich [1992a] as well as the fact that the different global control regimes influence the behavior 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 "lookahead" to detect failure at a deeper level. This approach is used by ECCE-d and SP. Note however that SP seems to employ a refined determinate unfolding rule (as indicated for example by the results for the benchmark *depth.lam* below).

Both of these unfolding rules actually ensure that neither the number nor the order of the solutions under Prolog execution are altered. Also, termination under Prolog will be preserved by these unfolding rules (termination behavior might however be improved, as for example $\leftarrow loop, fail$ can be transformed into $\leftarrow fail$). For more details related to the preservation of (Prolog) termination we refer to Proietti and Pettorossi [1991a], Bossi and Cocco [1994], and Bossi et al. [1995].

5.2 Experiments

The benchmark programs can be found in Leuschel [1996]; short descriptions are presented in electronic Appendix A. In addition to the benchmarks by Lam and Kusalik [1990] they contain a whole set of more involved and realistic examples, for example, such as a model-elimination theorem prover and a metainterpreter for an imperative language. For some of these new benchmark tasks it is impossible to get (big) speedups—the goal of these tasks consists in testing whether no pessimization, code explosion or nontermination occurs.

For the experimentation, we tried to be as realistic and practice-oriented as possible. In particular, we did not count the number of inferences, the cost of which

[16]i.e., those problems for which normal evaluation terminates.

varies a lot, or some other abstract measure, but the actual execution time and size of compiled code. The results are summarized in Tables I and II, while the full details can be found in Tables III, IV, and V. Further details and explanations about the benchmark results are listed below:

—*Transformation times* (*TT*): The transformation times of ECCE and MIXTUS include the time to write the specialized program to file. Time for SP does not, and for PADDY we do not know. We briefly explain the use of $\perp$ in the tables:

—$\perp$, SP: this means "real" nontermination (based upon the description of the algorithm in Gallagher [1991])

—$\perp$, MIXTUS: heap overflow after 20 minutes

—$\perp$, PADDY: thorough system crash after 2 minutes

In Tables III, IV, and V, the transformation times (TT) are expressed in seconds while the total transformation time in Table II 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). Each system was executed using the Prolog system it runs on: ProLog by BIM for ECCE, SICStus Prolog for MIXTUS and SP, and Eclipse for PADDY). So, except when comparing the different settings of ECCE, the transformation times should only be used for a rough comparison.

—*Relative run-times* (*RRT*) *of the specialized 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 specialized programs directly (i.e., without overhead), at least 100 times for the respective run-time queries, using the *time*/2 predicate of ProLog by BIM 4.0.12 on a Sparc Classic under Solaris. Sufficient memory was given to the Prolog system to prevent garbage collection. Run-times in Tables III, IV, and V are given *relative* to the run-times of the original programs. In computing averages and totals, the time and size of the original program were taken in case of nontermination (i.e., we did not punish MIXTUS, PADDY, and SP for the nontermination). 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 specialized and original programs respectively. The weighted speedups are obtained by using the code size $size_i$ of the original program as a weight for computing the average:

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

In Table I the column for "FU" holds the total speedup for the fully unfoldable benchmarks (see electronic 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 programs 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 specialized part

Table I. Short Summary of the Speedups

System	Total Speedup	Weighted Speedup	Worst Speedup	FU Speedup	Not FU Speedup
ECCE-d	1.90	2.16	0.85	2.57	1.74
ECCE-x-10	2.13	2.23	0.79	7.07	1.71
ECCE-x	2.51	2.75	0.92	8.36	2.02
MIXTUS	2.08	2.48	0.65	8.13	1.65
PADDY	2.08	2.31	0.68	8.12	1.65
SP	1.46	1.56	0.86	2.08	1.32

Table II. Short Summary of the Code Size and Transformation Times

System	Total Code Size in KB	Total TT in min
ECCE-d	166.69	2.64
ECCE-x-10	224.35	112.72
ECCE-x	135.91	2.70
MIXTUS	152.26	$\perp$+2.49
PADDY	196.19	$\perp$+0.28
SP	182.02	3$\perp$+1.92

could only be called in a renamed style. We removed the original program whenever possible and added one clause which allows calling the specialized program also in an unrenamed style (just like MIXTUS and ECCE). This possibility was not used in the benchmarks, but avoids distortions in the code size figures (with respect to MIXTUS and ECCE).

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

5.3 Analyzing the Results

The ECCE-based systems did terminate on all examples, as would be expected by the results presented earlier in the article. 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 nontermination, but we have not been able to verify this.) Even ignoring this fact, the system ECCE-x clearly outperforms the others, for speed as well 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 of Prestwich [1993] which help PADDY to be (except for one glitch) the fastest system overall.

Note that even the system ECCE-x-10 has a better total speedup than earlier systems. Its transformation times, weighted speedup, and size of the specialized code are not so good. Also note that for some benchmarks the 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 the depth bound (ECCE-x) we increase the total speedup from 2.13 to 2.51 (and the weighted speedup from 2.23 to 2.75) while decreasing the size of the specialized code from 235KB down to 142KB. Also the total transformation time drastically decreases by

Table III. Detailed Results for ECCE-x-10 and ECCE-x

	ECCE-x-10			ECCE-x		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.32	809	1.01	0.31	809	0.78
contains.kmp	0.80	1974	11.53	0.09	685	4.48
depth.lam	0.06	1802	2.25	0.02	2085	1.91
doubleapp	0.95	216	0.59	0.95	216	0.53
ex_depth	0.32	350	1.73	0.32	350	1.58
grammar.lam	0.14	218	2.27	0.14	218	1.90
groundunify.complex	0.53	4511	29.10	0.53	4800	0.75
groundunify.simple	0.25	368	0.83	0.25	368	22.03
imperative.power	0.56	1578	18.14	0.54	1578	27.42
liftsolve.app	0.53	4544	16.70	0.06	1179	6.57
liftsolve.db1	0.02	2767	22.15	0.02	1326	7.33
liftsolve.db2	0.47	11303	154.94	0.61	4786	34.25
liftsolve.lmkng	1.02	2385	3.44	1.02	2385	2.75
map.reduce	0.08	348	0.84	0.08	348	0.86
map.rev	0.13	427	1.02	0.11	427	0.89
match.kmp	0.70	669	1.24	0.70	669	1.23
memo-solve	1.26	2033	10.56	1.09	2241	4.31
missionaries	1.03	2927	26.67	0.72	2226	9.21
model_elim.app	0.42	6092	5864.28	0.13	532	3.56
regexp.r1	0.29	435	6.77	0.29	435	0.98
regexp.r2	0.43	1373	8.63	0.51	1159	4.87
regexp.r3	0.48	2041	10.82	0.42	1684	14.92
relative.lam	0.02	709	506.89	0.00	261	4.06
rev_acc_type	0.99	2188	22.95	1.00	242	0.83
rev_acc_type.inffail	0.55	1503	21.47	0.60	527	0.80
ssuply.lam	0.14	426	7.84	0.06	262	1.18
transpose.lam	0.18	2312	8.75	0.17	2312	1.98
Average	0.47	2085	250.50	0.40	1263	6.00
Total	12.67	56308	6763.41	10.75	34177	161.96
Total Speedup	**2.13**			**2.51**		
Weighted Speedup	**2.23**			**2.75**		

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 specialization (see also Section 6). The "MIXTUS-like" unfolding seems to be a good compromise for standard partial deduction.

The weighted speedup of ECCE-x is 2.75, and the speedup for the fully unfoldable benchmarks is 2.02, i.e., by partial deduction we were able to (more than) halve execution times. Compared to speedups that are usually obtained by low-level compiler optimizations, these figures are extremely satisfactory. Taken on its own, however, these figures might look a bit disappointing as to the potential of partial deduction. But, as already mentioned, for some benchmark tasks it is impossible to get significant speedups. Also, partial deduction is of course not equally

Table IV. Detailed Results for ECCE-d and SP

	ECCE-d			SP		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.47	412	0.79	0.40	463	0.29
contains.kmp	0.83	1363	2.90	0.75	985	1.13
depth.lam	0.94	1955	1.53	0.53	928	0.99
doubleapp	0.98	277	0.61	1.02	160	0.11
ex_depth	0.76	1614	2.78	0.27	786	1.35
grammar.lam	0.17	309	1.92	0.15	280	0.71
groundunify.complex	0.40	9502	25.04	0.73	4050	2.46
groundunify.simple	0.25	368	0.78	0.61	407	0.20
imperative.power	0.37	2401	61.28	0.97	1706	6.97
liftsolve.app	0.06	1179	5.42	0.23	1577	2.46
liftsolve.db1	0.01	1280	12.95	0.82	4022	3.95
liftsolve.db2	0.17	4694	14.95	0.82	3586	3.71
liftsolve.lmkng	1.07	1730	1.70	1.16	1106	0.37
map.reduce	0.07	507	0.84	0.09	437	0.23
map.rev	0.11	427	0.88	0.13	351	0.20
match.kmp	0.73	639	1.17	1.08	527	0.49
memo-solve	1.17	2318	4.22	1.15	1688	3.65
missionaries	0.81	2294	4.31	0.73	16864	82.59
model_elim.app	0.63	2100	2.83	–	–	$\perp$
regexp.r1	0.50	594	1.29	0.54	466	0.37
regexp.r2	0.55	629	1.29	1.08	1233	0.67
regexp.r3	0.50	828	1.74	1.03	1646	1.20
relative.lam	0.82	1074	1.92	0.69	917	0.35
rev_acc_type	1.00	242	0.70	–	–	$\perp$
rev_acc_type.inffail	0.60	527	0.71	–	–	$\perp$
ssuply.lam	0.06	262	1.18	0.06	231	0.52
transpose.lam	0.17	2312	2.56	0.26	1267	0.52
Average	0.53	1550	5.86	0.68	1903	4.81
Total	14.19	41837	158.29	18.48	45683	115.5
Total Speedup	**1.90**			**1.46**		
Weighted Speedup	**2.16**			**1.56**		

suited for all tasks; but for those tasks for which it is suited, partial deduction can be even much more worthwhile. For instance for the benchmarks model_elim.app, liftsolve.app, and liftsolve.db1—which exhibit interpretation overhead—ECCE-x obtains speedup factors of about 8, 17, and 50 respectively. Getting rid of higher-order overhead also seems highly beneficial, yielding about one order of magnitude speedup. Another area in which partial deduction might be very worthwhile is in handling overly general programs, e.g., getting rid of unnecessary intermediate variables or making use of the hidden part of abstract data types. However, such optimizations require a more powerful partial-deduction framework. Such a framework is provided in Leuschel et al. [1996], Glück et al. [1996], and Leuschel [1997a], and Jørgensen et al. [1996] show that the techniques elaborated in the present article carry over to that context, leading to further improvements over prior techniques. So, given the difficulty of the benchmarks (and the worst-case slowdown of only 8%), the speedup figures are actually very satisfactory, and we conjecture that it

Table V. Detailed Results for MIXTUS and PADDY

	MIXTUS			PADDY		
Benchmark	RRT	Size	TT	RRT	Size	TT
advisor	0.31	809	0.85	0.31	809	0.10
contains.kmp	0.16	533	2.48	0.11	651	0.55
depth.lam	0.04	1881	4.15	0.02	2085	0.32
doubleapp	1.00	295	0.30	0.98	191	0.08
ex_depth	0.40	643	2.40	0.29	1872	0.53
grammar.lam	0.17	841	2.73	0.43	636	0.22
groundunify.complex	0.67	5227	11.68	0.60	4420	1.53
groundunify.simple	0.25	368	0.45	0.25	368	0.13
imperative.power	0.57	2842	5.35	0.58	3161	2.18
liftsolve.app	0.06	1179	4.78	0.06	1454	0.80
liftsolve.db1	0.01	1280	5.36	0.02	1280	1.20
liftsolve.db2	0.31	8149	58.19	0.32	4543	1.60
liftsolve.lmkng	1.16	2169	4.89	0.98	1967	0.32
map.reduce	0.68	897	0.17	0.08	498	0.20
map.rev	0.11	897	0.16	0.26	2026	0.37
match.kmp	1.55	467	4.89	0.69	675	0.28
memo-solve	0.60	1493	12.72	1.48	3716	1.70
missionaries	–	–	$\perp$	–	–	$\perp$
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
regexp.r2	0.82	1916	2.85	0.67	3605	0.63
regexp.r3	0.60	2393	4.49	1.26	10399	1.35
relative.lam	0.01	517	7.76	0.00	517	0.42
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
transpose.lam	0.18	1302	3.89	0.18	1302	0.43
Average	0.48	1470	5.76	0.48	1894	0.64
Total	13.00	38214	149.7	12.96	49239	16.7
Total Speedup	**2.08**			**2.08**		
Weighted Speedup	**2.48**			**2.31**		

will be beneficial to integrate the techniques, developed above, into a compiler.

In conclusion, the ideas presented in this article do not only make sense in theory on accounts of elegance, generality, precision, and termination, but they also pay off in practice. The resulting specializer improves significantly upon earlier work with respect to speed and size of the specialized programs.

6. CONCLUSION AND FUTURE WORK

In the first part of this article 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 specialized 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 article, we have first identified the problems of imposing a depth bound on characteristic trees (or neighborhoods for that matter) using some practical and realistic examples. We have then developed an even more sophisticated on-line global control technique for partial deduction of normal logic programs. Importing and adapting m-trees of Martens and Gallagher [1995], 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 Bruynooghe et al. [1992] and Martens and De Schreye [1996]), 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. Moreover, we believe it to be useful for on-line partial evaluation and transformation in other programming paradigms as well (cf. Section 4.7).

Along the way, we have defined generalization and embedding on characteristic atoms, refining the homeomorphic embedding relation $\trianglelefteq$ of Dershowitz [1987], Dershowitz and Jouannaud [1990], Marlet [1994], and Sørensen and Glück [1995] into $\trianglelefteq^*$, and showing that the latter is more suitable in a logic programming setting. We have also touched upon a postprocessing 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 systems for speedup as well as code size while guaranteeing termination.

We believe that the global control proposed in this article is a very good one, but the quality of the specialization produced by any fully concrete instance of Algorithm 4.6.1 will obviously also depend heavily 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 metainterpreters and a good treatment of truly nondeterminate programs are among the most pressing.

Recent work brought a closer integration of abstract interpretation and partial deduction [Leuschel and De Schreye 1996b], as well as an extension of partial deduction [Glück et al. 1996; Leuschel et al. 1996] to incorporate more powerful unfold/fold-like transformations [Pettorossi and Proietti 1994], allowing for example to eliminate unnecessary variables from programs [Proietti and Pettorossi 1991b]. The latter extension basically reduces to the lifting of entire goals (instead of separate atoms) to the global level, as for instance in supercompilation (where nonatomic 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 article can be extended and that they significantly contribute in that context too (see Leuschel et al. [1996], Glück et al. [1996], Leuschel [1997a], and Jørgensen et al. [1996]).

ONLINE-ONLY APPENDICES

A. BENCHMARK PROGRAMS

B. PROOF OF THEOREM 3.3.2

C. PROOF OF THEOREM 3.4.5

D. ASSORTED SHORT PROOFS

E. PROOF OF PROPOSITION 4.3.9

F. PROOF OF THEOREM 4.4.7

G. PROOF OF PROPOSITION 4.4.9

Appendices A-G are available only online in a seperate document. You should be able to get the online-only appendices from the citation page for this article:

http://www.acm.org/pubs/citations/journals/toplas/1998-20-1/p208-leuschel/

Alternative instructions on how to obtain online-only appendices are given on the back inside cover of current issues of ACM TOPLAS or on the ACM TOPLAS web page:

http://www.acm.org/toplas

ACKNOWLEDGMENTS

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 the anonymous referees, also of LOPSTR'95 and the 1996 Dagstuhl Seminar on Partial Evaluation. Finally, Alain Callebaut provided helpful information on benchmarking ProLog by BIM programs.

REFERENCES

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

Alpuente, M., Falaschi, M., Julián, P., and Vidal, G. 1997. Spezialisation of lazy functional logic programs. In *Proceedings of PEPM'97, the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, New York, 151–162.

Alpuente, M., Falaschi, M., and Vidal, G. 1996. Narrowing-driven partial evaluation of functional logic programs. In *Proceedings of the 6th European Symposium on Programming, ESOP'96*, H. Riis Nielson, Ed. Lecture Notes in Computer Science, vol. 1058. Springer-Verlag, Berlin, 45–61.

Benkerimi, K. and Hill, P. M. 1993. Supporting transformations for the partial evaluation of logic programs. *J. Logic Comput. 3,* 5 (Oct.), 469–486.

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

Bol, R. 1993. Loop checking in partial deduction. *J. Logic Progam. 16,* 1–2, 25–46.

Bossi, A. and Cocco, N. 1994. Preserving universal termination through unfold/fold. In *Proceedings of the 4th International Conference on Algebraic and Logic Programming*, G. Levi and M. Rodriguez-Artalejo, Eds. Lecture Notes in Computer Science, vol. 850. Springer-Verlag, Berlin, 269–286.

Bossi, A., Cocco, N., and Dulli, S. 1990. A method for specialising logic programs. *ACM Trans. Program. Lang. Syst. 12,* 2, 253–302.

Bossi, A., Cocco, N., and Etalle, S. 1995. Transformation of left terminating programs: The reordering problem. In Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, M. Proietti, Ed. Lecture Notes in Computer Science, vol. 1048. Springer-Verlag, Berlin, 33–45.

Bruynooghe, M. 1991. A practical framework for the abstract interpretation of logic programs. *J. Logic Progam. 10*, 91–124.

Bruynooghe, M., De Schreye, D., and Martens, B. 1992. A general criterion for avoiding infinite unfolding during partial deduction. *New Gen. Comput. 11,* 1, 47–79.

Bruynooghe, M., Leuschel, M., and Sagonas, K. 1998. A polyvariant binding-time analysis for off-line partial deduction. In *Proceedings of the European Symposium on Programming (ESOP'98)*, C.Hankin, Ed. Lecture Notes in Computer Science. Springer-Verlag, Berlin. To appear.

Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *J. ACM 24,* 1, 44–67.

Chan, D. and Wallace, M. 1989. A treatment of negation during partial evaluation. In *Meta-Programming in Logic Programming, Proceedings of the META'88 Workshop*, H. Abramson and M. Rogers, Eds. MIT Press, Cambridge, Mass., 299–318.

Consel, C. and Danvy, O. 1993. Tutorial notes on partial evaluation. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL'93).* ACM Press, New York.

Cousot, P. and Cousot, R. 1992. Abstract interpretation and application to logic programs. *J. Logic Progam. 13,* 2–3, 103–179.

De Schreye, D., Leuschel, M., and Martens, B. 1995. Tutorial on program specialisation (abstract). In *Proceedings of ILPS'95, the International Logic Programming Symposium*, J. W. Lloyd, Ed. MIT Press, Cambridge, Mass., 615–616.

de Waal, D. A. 1994. Analysis and transformation of proof procedures. Ph.D. thesis, Dept. of Computer Science, Univ. of Bristol, Bristol, U.K.

de Waal, D. A. and Gallagher, J. 1991. Specialisation of a unification algorithm. In Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'91*, T. Clement and K.-K. Lau, Eds. Workshops in Computing. Springer-Verlag, Berlin, 205–220.

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

Debray, S. 1997. Resource-bounded partial evaluation. In *Proceedings of PEPM'97, the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, New York, 179–192.

Debray, S. and Lin, N.-W. 1993. Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst. 15,* 5, 826–875.

Debray, S., López García, P., Hermenegildo, M., and Lin, N.-W. 1994. Estimating the computational cost of logic programs. In *Proceedings of SAS'94*, B. Le Charlier, Ed. Lecture Notes in Computer Science, vol. 864. Springer-Verlag, Berlin, 255–265.

Dershowitz, N. 1987. Termination of rewriting. *J. Symbol. Comput. 3*, 69–116.

Dershowitz, N. and Jouannaud, J.-P. 1990. Rewrite systems. In *Handbook of Theoretical Computer Science.* Vol. B, *Formal Models and Semantics*, J. van Leeuwen, Ed. Elsevier, MIT Press, Cambridge, Mass., 243–320.

Futamura, Y., Nogi, K., and Takano, A. 1991. Essence of generalized partial computation. *Theor. Comput. Sci. 90,* 1, 61–79.

Gallagher, J. 1991. A system for specialising logic programs. Tech. Rep. TR-91-32, Univ. of Bristol, Bristol, U.K.

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

Gallagher, J. and Bruynooghe, M. 1990. Some low-level transformations for logic programs. In *Proceedings of the META'90 Workshop on Meta Programming in Logic*, M. Bruynooghe, Ed. K.U. Leuven, Belgium, 229–244.

Gallagher, J. and Bruynooghe, M. 1991. The derivation of an algorithm for program specialisation. *New Gen. Comput. 9,* 3–4, 305–333.

Gallagher, J. and de Waal, D. A. 1992. Deletion of redundant unary type predicates from logic programs. In Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, K.-K. Lau and T. Clement, Eds. Workshops in Computing. Springer-Verlag, Berlin, 151–167.

Gallagher, J. and Lafave, L. 1996. Regular approximations of computation paths in logic and functional languages. In *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, Berlin, 115–136.

Glück, R. and Klimov, A. V. 1993. Occam's razor in metacomputation: the notion of a perfect process tree. In *Proceedings of WSA'93*, G. Filé, P. Cousot, M. Falaschi, and A. Rauzy, Eds. Lecture Notes in Computer Science, vol. 724. Springer-Verlag, Berlin, 112–123.

Glück, R. and Sørensen, M. H. 1996. A roadmap to supercompilation. In *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, Berlin, 137–160.

Glück, R., Jørgensen, J., Martens, B., and Sørensen, M. H. 1996. Controlling conjunctive partial deduction of definite logic programs. In *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, H. Kuchen and S. Swierstra, Eds. Lecture Notes in Computer Science, vol. 1140. Springer-Verlag, Berlin, 152–166. Extended version as Tech. Rep. CW 226, K.U. Leuven.

Gurr, C. A. 1994. A self-applicable partial evaluator for the logic programming language Gödel. Ph.D. thesis, Dept. of Computer Science, Univ. of Bristol, Bristol, U.K.

Gustedt, J. 1992. Algorithmic aspects of ordered structures. Ph.D. thesis, Technische Universität Berlin, Berlin, Germany.

Heintze, N. 1992. Practical aspects of set based analysis. In *Proceedings of the Joint International Conference and Symposium on Logic Programming.* MIT Press, Cambridge, Mass., 765–779.

Higman, G. 1952. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc. 2*, 326–336.

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

Hopcroft, J. E. and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Mass.

Horváth, T. 1993. Experiments in partial deduction. M.S. thesis, Departement Computerwetenschappen, K.U. Leuven, Belgium.

Janssens, G. and Bruynooghe, M. 1992. Deriving descriptions of possible values of program variables by means of abstract interpretation. *J. Logic Progam. 13,* 2–3, 205–258.

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

Jørgensen, J. and Leuschel, M. 1996. Efficiently generating efficient generating extensions in Prolog. In *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, Berlin, 238–262. Extended version as Tech. Rep. CW 221, K.U. Leuven.

Jørgensen, J., Leuschel, M., and Martens, B. 1996. Conjunctive partial deduction in practice. In *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, J. Gallagher, Ed. Lecture Notes in Computer Science, vol. 1207. Springer-Verlag, Berlin, 59–82. Extended version as Tech. Rep. CW 242, K.U. Leuven.

Knuth, D. E., Morris, J. H., and Pratt, V. R. 1977. Fast pattern matching in strings. *SIAM J. Comput. 6,* 2, 323–350.

Komorowski, J. 1992. An introduction to partial deduction. In *Proceedings Meta'92*, A. Pettorossi, Ed. Lecture Notes in Computer Science, vol. 649. Springer-Verlag, Berlin, 49–69.

Kruskal, J. B. 1960. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Trans. Am. Math. Soc. 95*, 210–225.

Lafave, L. and Gallagher, J. 1997. Constraint-based partial evaluation of rewriting-based functional logic programs. In *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, N. Fuchs, Ed. Lecture Notes in Computer Science. Springer-Verlag, Berlin. To appear.

Lam, J. and Kusalik, A. 1990. A comparative analysis of partial deductors for pure Prolog. Tech. Rep. Dept. of Computational Science, Univ. of Saskatchewan, Canada. Revised April 1991.

Lassez, J.-L., Maher, M., and Marriott, K. 1988. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan-Kaufmann, San Mateo, Calif., 587–625.

Leuschel, M. 1994. Partial evaluation of the "real thing". In Logic Program Synthesis and T ransformation—Meta-Programming in Logic. *Proceedings of LOPSTR'94 and META'94*, L. Fribourg and F. Turini, Eds. Lecture Notes in Computer Science, vol. 883. Springer-Verlag, Berlin, 122–137.

Leuschel, M. 1995. Ecological partial deduction: Preserving characteristic trees without constraints. In Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, M. Proietti, Ed. Lecture Notes in Computer Science, vol. 1048. Springer-Verlag, Berlin, 1–16.

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

Leuschel, M. 1997a. Advanced techniques for logic program specialisation. Ph.D. thesis, K.U. Leuven. Accessible via http://www.cs.kuleuven.ac.be/~michael.

Leuschel, M. 1997b. Extending homeomorphic embedding in the context of logic programming. In *Proceedings of the 12th Workshop on Logic Programming (WLP'97)*, F.Bry, B. Freitag and D. Seipel, Eds. 92–103. Extended version as Tech. Rep. CW 252, K.U. Leuven, Belgium..

Leuschel, M. and De Schreye, D. 1995. 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.* ACM Press, New York, 253–263.

Leuschel, M. and De Schreye, D. 1996a. Creating specialised integrity checks through partial evaluation of meta-interpreters. Tech. Rep. CW 237, K.U. Leuven, Belgium. To appear in *J. Logic Progam.*

Leuschel, M. and De Schreye, D. 1996b. Logic program specialisation: How to be more specific. In *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, H. Kuchen and S. Swierstra, Eds. Lecture Notes in Computer Science, vol. 1140. Springer-Verlag, Berlin, 137–151. Extended version as Tech. Rep. CW 232, K.U. Leuven.

Leuschel, M. and De Schreye, D. 1998. Constrained partial deduction and the preservation of characteristic trees. *New Gen. Comput.* To Appear.

Leuschel, M. and Martens, B. 1996. Global control for partial deduction through characteristic atoms and global trees. In *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, Berlin, 263–283.

Leuschel, M. and Sørensen, M. H. 1996. Redundant argument filtering of logic programs. In *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, J. Gallagher, Ed. Lecture Notes in Computer Science, vol. 1207. Springer-Verlag, Berlin, 83–103. Extended version as Tech. Rep. CW 243, K.U. Leuven.

Leuschel, M., De Schreye, D., and de Waal, A. 1996. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, M. Maher, Ed. MIT Press, Cambridge, Mass., 319–332. Extended version as Tech. Rep. CW 225, K.U. Leuven.

Lloyd, J. W. 1987. *Foundations of Logic Programming.* Springer-Verlag, Berlin.

Lloyd, J. W. and Shepherdson, J. C. 1991. Partial evaluation in logic programming. *J. Logic Progam. 11,* 3–4, 217–242.

Marlet, R. 1994. Vers une formalisation de l'Évaluation partielle. Ph.D. thesis, Université de Nice-Sophia Antipolis.

Martens, B. 1994. On the semantics of meta-programming and the control of partial deduction in logic programming. Ph.D. thesis, K.U. Leuven.

Martens, B. and De Schreye, D. 1996. Automatic finite unfolding using well-founded measures. *J. Logic Progam. 28,* 2 (August), 89–146.

Martens, B. and Gallagher, J. 1995. Ensuring global termination of partial deduction while allowing flexible polyvariance. In *Proceedings ICLP'95*, L. Sterling, Ed. MIT Press, Cambridge, Mass., 597–613. Extended version as Tech. Rep. CSTR-94-16, Univ. of Bristol.

Martens, B., De Schreye, D., and Horváth, T. 1994. Sound and complete partial deduction with unfolding based on well-founded measures. *Theor. Comput. Sci. 122,* 1–2, 97–117.

Meulemans, D. 1995. Partiële deductie: Een substantiële vergelijkende studie. M.S. thesis, Departement Computerwetenschappen, K.U. Leuven, Belgium.

Mogensen, T. and Bondorf, A. 1992. Logimix: A self-applicable partial evaluator for Prolog. In Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, K.-K. Lau and T. Clement, Eds. Workshops in Computing. Springer-Verlag, Berlin, 214–227.

Pettorossi, A. and Proietti, M. 1994. Transformation of logic programs: Foundations and techniques. *J. Logic Progam. 19–20*, 261–320.

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

Prestwich, S. 1992a. The PADDY partial deduction system. Tech. Rep. ECRC-92-6, ECRC, Munich, Germany.

Prestwich, S. 1992b. An unfold rule for full Prolog. In Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, K.-K. Lau and T. Clement, Eds. Workshops in Computing. Springer-Verlag, Berlin, 199–213.

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

Proietti, M. and Pettorossi, A. 1991a. Semantics preserving transformation rules for Prolog. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics based Program Manipulation, PEPM'91*, N. D. Jones and P. Hudak, Eds. *SIGPLAN Not. 26*, 9, 274–284.

Proietti, M. and Pettorossi, A. 1991b. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming, PLILP'91*, J. Małuszyński and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 528. Springer-Verlag, Berlin, 347–358.

Puebla, G. and Hermenegildo, M. 1995. Implementation of multiple specialization in logic programs. In *Proceedings of PEPM'95, the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, New York, 77–87.

Sahlin, D. 1991. An automatic partial evaluator for full Prolog. Ph.D. thesis, Swedish Institute of Computer Science.

Sahlin, D. 1993. Mixtus: An automatic partial evaluator for full Prolog. *New Gen. Comput. 12,* 1, 7–51.

Smith, D. A. 1991. Partial evaluation of pattern matching in constraint logic programming languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, N. D. Jones and P. Hudak, Eds. *SIGPLAN Not. 26*, 9, 62–71.

Smith, D. A. and Hickey, T. 1990. Partial evaluation of a CLP language. In *Proceedings of the North American Conference on Logic Programming*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, Mass., 119–138.

Sørensen, M. H. and Glück, R. 1995. An algorithm of generalization in positive supercompilation. In *Proceedings of ILPS'95, the International Logic Programming Symposium*, J. W. Lloyd, Ed. MIT Press, Cambridge, Mass., 465–479.

Sperber, M. 1996. Self-applicable online partial evaluation. In *Proceedings of the 1996 Dagstuhl Seminar on Partial Evaluation*, O. Danvy, R. Glück, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1110. Springer-Verlag, Berlin, 465–480.

Stillman, J. 1988. Computational problems in equational theorem proving. Ph.D. thesis, State Univ. of New York at Albany, U.S.A.

Turchin, V. F. 1986. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst. 8,* 3, 292–325.

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

Vanhoof, W. and Martens, B. 1997. To parse or not to parse. In *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97)*, N. Fuchs, Ed. Lecture Notes in Computer Science. Springer-Verlag, Berlin. To appear. Also as Tech. Rep. CW 251, K.U.Leuven.

Wadler, P. 1990. Deforestation: Transforming programs to eliminate intermediate trees. *Theor. Comput. Sci. 73*, 231–248. Preliminary version in ESOP'88, Lecture Notes in Computer Science, vol. 300.

Weise, D., Conybeare, R., Ruf, E., and Seligman, S. 1991. Automatic online partial evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures.* Lecture Notes in Computer Science, vol. 523. Springer-Verlag, Berlin, 165–191.

Winsborough, W. 1992. Multiple specialization using minimal-function graph semantics. *J. Logic Progam. 13,* 2–3, 259–290.

Received December 1996; revised June 1997; accepted August 1997

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs

MICHAEL LEUSCHEL, BERN MARTENS, and DANNY DE SCHREYE
Katholieke Universiteit Leuven

ACM Transactions on Programming Languages and Systems, Vol. 20, No. 1, January 1998, Pages 208–258.

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, metainterpreters (nonground vanilla, mixed, ground), and more involved particular ones: a model-elimination theorem prover, the missionaries-cannibals problem, a metainterpreter for a simple imperative language. The benchmarks marked with a star (*) can be fully unfolded. The size of the compiled code (under ProLog by BIM 4.0.12) is given in parentheses. Full descriptions can be found in Leuschel [1996].

advisor.* (6810 bytes)
A very simple expert system which can be fully unfolded. A benchmark by Horváth [1993].

contains.kmp. (2570 bytes)
A benchmark based on the "contains" benchmark by Lam and Kusalik [1990], but with improved run-time queries. The program is a rather involved, but still inefficient (because highly nondeterministic), pattern matcher.

depth.lam.* (4415 bytes)
A simple metainterpreter which keeps track of the maximum length of refutations. It has to be specialized for a simple, fully unfoldable object program. A benchmark from Lam and Kusalik [1990].

doubleapp. (653 bytes)
The double append example (see Leuschel et al. [1996], Glück et al. [1996], and Leuschel [1997a]) in which three lists are appended by reusing the ordinary *append* program. Tests whether deforestation can be done.

ex_depth. (4741 bytes)
A variation of *depth.lam* with a more sophisticated object program (which cannot be fully unfolded).

grammar.lam. (9490 bytes)
A DCG (Definite Clause Grammar) parser which has to be specialized for a particular grammar. It is one of the benchmarks from Lam and Kusalik [1990].

groundunify.complex. (10106 bytes)
The task consists in specializing an explicit unification algorithm for the ground representation. The full code can be found in Leuschel [1997a], where it is adapted from de Waal and Gallagher [1991].

groundunify.simple.* (10106 bytes)
The same unification algorithm as for groundunify.complex, but with a simpler specialization query.

imperative.power. (9368 bytes)
An interpreter for a simple imperative language which stores values of variables in an environment. It has to be specialized for a *power* subprocedure, calculating $Base^{Exp}$, for a known exponent Exp and base $Base$ but an unknown environment.

liftsolve.app. (5194 bytes)
A metainterpreter for the ground representation which "lifts" the program to the nonground representation for resolution. In Leuschel and De Schreye [1996a] and Leuschel [1997a] this is called the mixed representation. A description along with the code can be found in Leuschel and De Schreye [1996a] and Leuschel [1997a] (see also Gallagher [1991]). The goal is to specialize this metainterpreter for *append* as the object program.

liftsolve.db1.* (5194 bytes)
The same metainterpreter as liftsolve.app with a simple, fully unfoldable object program.

liftsolve.db2. (5194 bytes)
Again the same metainterpreter as liftsolve.app, but this time with a *partially* specified object program.

liftsolve.lmkng. (5194 bytes)
The goal here consists in specializing part of the above "lifting" metainterpreter. The specialization task is such that it may give rise to an ∞ number of characteristic trees.

map.reduce. (2868 bytes)
Specializing the higher-order map/3 (using the built-ins `call/1` and `=../2`) for the higher-order reduce/4 in turn applied to add/3.

map.rev. (2868 bytes)
Specializing the higher-order map for the reverse program.

match.kmp. (975 bytes)
A semi-naive pattern matcher; the goal is to obtain a Knuth-Morris-Pratt (see Knuth et al. [1977]) pattern matcher by specialization for the pattern "*aab*." The benchmark is based on the "match" Lam and Kusalik [1990] benchmark, but uses improved run-time queries (in order to detect whether a Knuth-Morris-Pratt matcher has been obtained).

memo-solve. (5251 bytes)
A variation of the ex_depth benchmark with a simple loop prevention mechanism based on keeping a call stack.

missionaries. (9221 bytes)
A program for the missionaries and cannibals problem.

model_elim.app. (7948 bytes)
Specialize the model elimination prover of Poole and Goebel [1986] (also used by de Waal and Gallagher [1994]) for the *append* program as the object-level theory.

regexp.r1. (1489 bytes)
A naive regular expression matcher which has to be specialized for the regular expression (a+b)*aab.

regexp.r2. (1489 bytes)
Same program as regexp.r1 for ((a+b)(c+d)(e+f)(g+h))*.

regexp.r3. (1489 bytes)
Same program as regexp.r1 and regexp.r2 for the regular expression
((a+b)(a+b)(a+b)(a+b)(a+b)(a+b))*.

relative.lam*. (3056 bytes)
A benchmark by Lam and Kusalik [1990] consisting of a fully unfoldable family database.

rev_acc_type. (828 bytes)
The benchmark program consists of the "reverse with accumulating parameter" program to which type-checking on the accumulator has been added. Without abstraction, the benchmark will give rise to an ∞ number of different characteristic trees. See Section 4.1 for details (and the code).

rev_acc_type.inffail. (828 bytes)
The same benchmark program as rev_acc_type, but this time the specialization task will give rise to infinite determinate failure at partial deduction time.

ssuply.lam*. (8335 bytes)
Another benchmark by Lam and Kusalik [1990].

transpose.lam*. (1599 bytes)
A benchmark program by Lam and Kusalik [1990] for transposing matrices. Also used in Gallagher [1991].

B. PROOF OF THEOREM 3.3.2

B.1 Correctness for Unconstrained Characteristic Atoms

If $\tilde{\mathcal{A}}$ is a set of unconstrained characteristic atoms, a partial deduction with respect to $\tilde{\mathcal{A}}$ can be seen as a standard partial deduction with renaming. We will make use of this observation to prove the following theorem.

Theorem B.1.1. *Let P' be a partial deduction of P with respect to $\tilde{\mathcal{A}}$ and ρ_α such that $\tilde{\mathcal{A}}$ is a finite set of* unconstrained *P-characteristic atoms and such that $\tilde{\mathcal{A}}$ is P-covered and G is P-covered by $\tilde{\mathcal{A}}$. Then:*

(1) *$P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ if and only if $P \cup \{G\}$ does.*

(2) *$P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree if and only if $P \cup \{G\}$ does.*

Proof. First note that the P-coveredness conditions on $\tilde{\mathcal{A}}$ and G ensure that the renamings performed to obtain P' (according to Definition 3.2.7), as well as the renaming $\rho_\alpha(G)$, are defined (because all the atoms in G, as well as all the leaf atoms of $\tilde{\mathcal{A}}$, are concretizations of elements in $\tilde{\mathcal{A}}$). The result then follows in a rather straightforward manner from the Theorems 3.5 and 4.11 of Benkerimi and Hill [1993]. Benkerimi and Hill [1993] splits the renaming 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 Benkerimi and Hill [1993]) and one which does the filtering (called postprocessing renaming; the correctness of this phase is proven in Theorem 4.11 of Benkerimi and Hill [1993]).

To apply these results we simply have to notice that:

—P' corresponds to partial deduction with dynamic renaming and postprocessing renaming for the multiset of atoms $\mathcal{A} = \{A \mid (A, \tau) \in \tilde{\mathcal{A}}\}$ (indeed the same atom could in principle occur in several characteristic atoms; this is not a problem however, as the results of Benkerimi and Hill [1993] carry over to multisets of atoms—alternatively one could add an extra unused argument to P', $\rho_\alpha(G)$ and $\mathcal{A}$ and then place different variables in that new position to transform the multiset $\mathcal{A}$ into an ordinary set).

—$P' \cup \{\rho_\alpha(G)\}$ is $\mathcal{A}$-covered because $\tilde{\mathcal{A}}$ is P-covered and G is P-covered by $\tilde{\mathcal{A}}$ (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 of Benkerimi and Hill [1993]:

—Theorem 3.5 of Benkerimi and Hill [1993] 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 where $vars(G) = \{X_1, \ldots, X_k\}$. We then 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 with respect to c.a.s. and finite failure (see also Lemma 2.2 of Gallagher and Bruynooghe [1990]).

—Theorem 4.11 of Benkerimi and Hill [1993] requires that G contains no variables or predicates in $\mathcal{A}$. 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 point above.

—Theorems 3.5 and 4.11 of Benkerimi and Hill [1993] require that the predicates of the renaming do not occur in the original P. Our Definition 3.2.6 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 Benkerimi and Hill [1993] and then applying an additional one step postprocessing renaming ρ'', with $\rho_\alpha = \rho'\rho''$, along with an extra application of Theorem 4.11 of Benkerimi and Hill [1993]. □

B.2 Correctness for Safe Characteristic Atoms

We first need the following adaptation of Lemma 4.12 from Lloyd and Shepherdson [1991] (we just formalize the use of "corresponding SLDNF-derivation" from Lloyd and Shepherdson [1991] in terms of characteristic paths).

LEMMA B.2.1. *Let R be the resultant of a finite (possibly incomplete) SLDNF-derivation of $P \cup \{\leftarrow A\}$ 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 of $P \cup \{\leftarrow A\theta\}$ whose characteristic path is δ and whose resultant is R_A.*

The following lemma is based upon Lemma B.2.1 and will prove useful later on. It establishes a link between SLD$^+$-derivations in the unrenamed specialized program and the original one.

LEMMA B.2.2. *Let P be a program, G a goal, $\tilde{\mathcal{A}}$ a set of* safe *characteristic atoms and P'' the union of partial deductions $R_{(A,\tau)}$ (in P), one for every element (A,τ) of $\tilde{\mathcal{A}}$. Let D be a finite SLD$^+$-derivation of $P'' \cup \{G\}$ with computed answer θ and resultant R and such that every selected atom A', which is resolved with a clause in $R_{(A,\tau)}$, is a concretization of (A,τ). Then there exists a finite SLD$^+$-derivation of $P \cup \{G\}$ with computed answer θ and resultant R.*

PROOF. We do the proof by induction on the length of the SLD$^+$-derivation D of $P'' \cup \{G\}$.

Induction Hypothesis: Lemma B.2.2 holds for SLD$^+$-derivations of $P'' \cup \{G\}$ with length $\leq k$.

Base Case: Let D have length 0. Then, trivially, the empty derivation of $P \cup \{G\}$ has the same resultant $G \leftarrow G$ and the same computed answer $\emptyset$.

Induction Step: Let D have length $k+1$. Let D''_k be the SLD$^+$-derivation of $P'' \cup \{G\}$ consisting of the first k steps of D. Let θ_k be the computed answer of D''_k and R_k its resultant. We can then apply the induction hypothesis to conclude that there is a SLD$^+$-derivation D_k of $P \cup \{G\}$ with the same resultant and computed answer. This also means that the resolvent—which is just the body of the resultant—of D''_k and D_k are the same. We denote this resolvent by RG. Let A' be the atom selected at the last step of D in the resolvent RG. Let $C \in P''$ be the clause with which A' is resolved. We know that C is the resultant of a finite SLDNF-derivation of an atom $P \cup \{A\}$, where $(A,\tau) \in \tilde{\mathcal{A}}$, because $\tilde{\mathcal{A}}$ contains only safe characteristic atoms. We also know, by assumption, that A' is a concretization of (A,τ) and therefore an instance of A. We can thus apply Lemma B.2.1 for the last derivation step of D to conclude that we can extend D_k in a similar way, obtaining the same resolvent (the resolvent is just the body of the resultant), the same overall computed answer θ (if the head of two resultants for the same goal RG are identical then so are the c.a.s., and by composing with θ_k we obtain the same overall computed answer) and thus also the same overall resultant (because, conversely, if the c.a.s. and resolvent, for a derivation starting from the same goal G, are the same then so are the resultants). $\square$

We will now extend Lemmas B.2.1 and B.2.2 and establish 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.

Definition B.2.3 (*Admissible Renaming*). Let F and F' be first-order formulas. Let P be a program and α an atomic renaming for a set $\tilde{\mathcal{A}}$ of characteristic atoms.

We call F' an *admissible renaming of* F *under* α *in* P if and only if there exists some renaming function ρ'_α for $\tilde{\mathcal{A}}$ in P based on α such that $F' = \rho'_\alpha(F)$.

Example B.2.4. Let P be the *member* program from Example 2.3.4 and let $\tilde{\mathcal{A}} = \{CA_1, CA_2\}$ with $CA_1 = (member(a, L), \{\langle 1 : 1\rangle\})$, $CA_2 = (member(a, L), \{\langle 1 : 1\rangle, \langle 1 : 2\rangle\})$. Let α be an atomic renaming such that $\alpha(CA_1) = mem_1(L)$ and $\alpha(CA_2) = mem_2(L)$. Then both $mem_1([a])$ and $mem_2([a])$ are admissible renamings of $member(a, [a])$ under α in P. Now $mem_2([a, a])$ is an admissible renaming of $member(a, [a, a])$ under α in P, while $mem_1([a, a])$ is not (because $member(a, [a, a])$ is not a concretization of CA_1). Also, $member(b, [a])$ has no admissible renamings under α in P.

The following lemma establishes a link between SLD$^+$-derivation steps in the (renamed) specialized program and the unrenamed specialized program.

Lemma B.2.5. *Let P' be a partial deduction of P with respect to $\tilde{\mathcal{A}}$ and ρ_α such that $\tilde{\mathcal{A}}$ is a finite set of* safe *P-characteristic atoms and such that $\tilde{\mathcal{A}}$ is P-covered and G is P-covered by $\tilde{\mathcal{A}}$.*

Let $C' = \alpha((A, \tau))\theta \leftarrow \rho_\alpha(Body)$, with $(A, \tau) \in \tilde{\mathcal{A}}$, be a clause in P' and let $C = A\theta \leftarrow Body$ be the unrenamed version of C'. Let G' be an admissible renaming of G under α in P and let γ be a substitution (which e.g., standardises C' apart with respect to G'). If RG' is derived from G' and $C'\gamma$ using θ' then there exists a goal RG such that:

(1) *RG is derived from G and $C\gamma$ using θ' and*

(2) *RG' is an admissible renaming of RG under α in P.*

Proof. Let L' be the selected atom in G' and let L be the corresponding atom in G. Because G' is an admissible renaming of G under α in P, we know that for some renaming function ρ'_α, we have $\rho'_\alpha(G) = G'$ and thus also $\rho'_\alpha(L) = L'$. Furthermore, as L' unifies with $\alpha((A, \tau))\theta\gamma$, we know that $L \in \gamma_P(A, \tau)$ and thus also $L = A\sigma$ and $L' = \alpha((A, \tau))\sigma$ for some substitution σ.
Now, we know that θ' is an idempotent and relevant *mgu* of $L' = \alpha((A, \tau))\sigma$ and $\alpha((A, \tau))\theta\gamma$. Because $vars(A) = vars(\alpha(A, \tau))$ (cf. Definition 3.2.6), we have that θ' is also an idempotent and relevant *mgu* of $L = A\sigma$ and $A\theta\gamma$. Hence, by selecting L in G for resolution with $C\gamma$, we obtain a goal RG which is derived from G and $C\gamma$ using the same θ'.
Finally, RG' is an admissible renaming of RG under α in P because:

—all atoms in *Body* are P-covered by $\tilde{\mathcal{A}}$ (because $\tilde{\mathcal{A}}$ is P-covered) and are therefore, by Definition 3.2.6 of a renaming, admissible renamings under α in P.

—the resolvent RG' will contain, in addition to instances of atoms in G', instances of the atoms in *Body*, all of which are still P-covered by $\tilde{\mathcal{A}}$, because the set of concretizations of characteristic atoms is downwards closed. RG' is therefore still an admissible renaming of RG under α in P. $\square$

Observe that if the substitution γ in the above lemma standardises C' apart with respect to G', then it also standardises C apart with respect to G (because, by Definition 3.2.6, $vars(G) = vars(G')$ as well as $vars(C) = vars(C')$).

$$\begin{tikzcd}
G' \arrow[d, "\theta", "P'"'] & G \arrow[l, dotted, "\rho'_\alpha"'] \arrow[dd, "\theta"', "P"] \\
RG' & \\
& RG \arrow[ul, dotted, "\rho''_\alpha"]
\end{tikzcd}$$

Fig. 12. Illustrating Lemma B.2.6.

Usually one will call the specialized program with one specific renaming and not just with an admissible one. So one might wonder why we only prove in Lemma B.2.5 above that RG' is an admissible renaming of RG under α in P 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 $\tilde{\mathcal{A}} = \{(p(X), \tau), (p(a), \tau')\}$ of unconstrained characteristic atoms, the goal $G = \leftarrow p(X), p(X)$ and take α such that:

—$\alpha((p(X), \tau)) = p'(X)$ and
—$\alpha((p(a), \tau')) = p_a$.

Then $\rho_\alpha(G) = \leftarrow p'(X), p'(X)$. Also assume that $\rho_\alpha(p(a)) = p_a$. Now suppose that the clause $C' = p'(a) \leftarrow$ is in the partial deduction P' with respect to an original P and the set $\tilde{\mathcal{A}}$. The clause $C = p(a) \leftarrow$ is then the unrenamed version of C'. Then $RG' = \leftarrow p'(a)$ is derived from $\rho_\alpha(G)$ and C' using $\{X/a\}$. Similarly, $RG = \leftarrow p(a)$ is derived from G and C using $\{X/a\}$ (no matter which literal we select). Now $\rho_\alpha(\leftarrow p(a)) = \leftarrow p_a \neq \leftarrow p'(a)$, i.e., $RG' \neq \rho_\alpha(RG)$! However, $\leftarrow p'(a)$ is still an admissible renaming of $\leftarrow p(a)$ under α in P and Lemma B.2.5 holds.

We now combine Lemmas B.2.2 and B.2.5 to establish a link between entire SLDNF-derivations in the renamed specialized program and original one. However, for the time being, we have restricted ourselves to unconstrained characteristic atoms in order to establish the result. An illustration of the following lemma can be found in Figure 12.

LEMMA B.2.6. *Let P' be a partial deduction of P with respect to $\tilde{\mathcal{A}}$ and ρ_α such that $\tilde{\mathcal{A}}$ is a finite set of* unconstrained *P-characteristic atoms and such that $\tilde{\mathcal{A}}$ is P-covered and G is P-covered by $\tilde{\mathcal{A}}$.*

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

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

Also note that not all resolvents of $P \cup \{G\}$ are P-covered by $\tilde{\mathcal{A}}$—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 of $P \cup \{G\}$ has a counterpart in $P' \cup \{G'\}$. We will however show in the following that, as asserted by the lemma, every derivation of $P' \cup \{G'\}$ has a counterpart in $P \cup \{G\}$.

We first do the proof for SLD$^+$-derivations, i.e., SLDNF-derivations in which no negative literals are selected. Let P'' be the unrenamed version of P' (i.e., the union of the partial deductions of the elements of $\tilde{\mathcal{A}}$). We can now inductively apply Lemma B.2.5 on every step of the SLD$^+$-derivation D' of $P' \cup \{G'\}$ and thus obtain a SLD$^+$-derivation of $P'' \cup \{G\}$ with the same computed answer θ and with a resolvent RG, such that RG' is an admissible renaming of RG. Furthermore, the so obtained derivation of $P'' \cup \{G\}$ will satisfy the conditions of Lemma B.2.2 (because every intermediate goal of this derivation can be renamed into an intermediate goal of D' and is thus P-covered by $\tilde{\mathcal{A}}$ in P, i.e., every selected atom is a concretization of an element of $\tilde{\mathcal{A}}$). We can thus apply Lemma B.2.2 to conclude that an SLD$^+$-derivation of $P \cup \{G\}$ with the same c.a.s. θ and the same resolvent RG (which is just the body of the resultant)—of which RG' is an admissible renaming—exists.

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 SLD$^+$-derivation D' (of $P' \cup \{G'\}$ leading to the resolvent RG'). In that case we can apply the just established result for the derivation leading up to the goal NG', in which the selected negative literal is selected and succeeds (because D' is not a failed derivation as it leads to a goal RG'). Then, because $NG' = \rho'_\alpha(NG)$ for some P-covered goal NG and some renaming function ρ'_α, and because $\tilde{\mathcal{A}}$ contains only unconstrained characteristic atoms, we can apply Theorem B.1.1, to deduce that the negative literal must also succeed in the original program (with the same computed answer, namely the identity substitution $\emptyset$). The next resolvents, $\widetilde{NG}$ and $\widetilde{NG}'$, are obtained from NG and NG' respectively, by simply removing a negative literal at the same position. Therefore, $\widetilde{NG}'$ is still an admissible renaming of $\widetilde{NG}$. We can thus re-apply the above result for SLD$^+$-derivations until the end of the derivation D'. Finally, we can repeat this same reasoning inductively for any number of selected negative literals. $\square$

In the proof of Lemma B.2.6, we used the fact that $\tilde{\mathcal{A}}$ contained only unconstrained characteristic atoms (to show that the behavior of the selected negative literals was preserved). We will now move to safe characteristic atoms and show that their partial deductions can be obtained from partial deductions of unconstrained characteristic atoms by removing certain clauses. This means that Lemma B.2.6 can actually be used to show soundness of SLD$^+$-derivations for partial deductions of safe characteristic atoms. To be able to show completeness, as well as allowing the selection of negative literals, we then show that these additional clauses can be removed without affecting the finite failure behavior.

The following lemmas will prove useful later on.

LEMMA B.2.7. *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 B.2.8. *Let A and B be atoms and τ_A a characteristic tree of A in the program P. If B is an instance of A then there exists a characteristic tree τ_B of B in P such that $\tau_B \subseteq \tau_A$.*

Proof. By Definition 2.3.2, for some unfolding rule U we have that $\tau_A = \mathit{chtree}(\leftarrow A, P, U)$. 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 $\tau_B = \mathit{chtree}(\leftarrow B, P, U') \subseteq \tau_A$. □

Lemma B.2.9. *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 $\tau \subseteq \mathit{chtree}(\leftarrow A, P, U)$.*

Proof. As (A,τ) is a P-characteristic atom, we must have by definition at least one precise concretization A' whose characteristic path is τ in P. As all the derivations in $D_P(A,\tau)$ 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 (which failed for $\leftarrow A'$). □

The above Lemma B.2.9 (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 Lemma B.2.11 below shows that it is correct to remove the resultants $P_A \setminus P_{(A,\tau)}$ for the concretizations 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 (cf. Aho et al. [1986] and Hopcroft and Ullman [1979]).

We also need the following lemma from Lloyd and Shepherdson [1991], where it is Lemma 4.10.

Lemma B.2.10(Persistence of Failure). *Let P be a normal program and G a normal goal. If $P \cup \{G\}$ has a finitely failed SLDNF-tree of height h and there is an SLDNF-derivation from G to G_1, then $P \cup \{G_1\}$ has a finitely failed SLDNF-tree of height $\leq h$.*

Lemma B.2.11. *Let P be a program and (A,τ_A), (A,τ_A^*) be safe P-characteristic atoms with $\tau_A \subseteq \tau_A^*$. Let $C \in \mathit{resultants}(\tau_A^*) \setminus \mathit{resultants}(\tau_A)$ and let $A' \in \gamma_P(A,\tau_A)$.*

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

Proof. Let δ_C be the characteristic path associated in Definition 3.2.2 (defining $D_P(A,\tau)$) with C, i.e., C is the resultant of the generalized SLDNF-derivation for $P \cup \{\leftarrow A\}$ whose characteristic path is δ_C. Because (A,τ_A^*) is safe, C is even the resultant of an SLDNF-derivation (and not of an unsafe generalized one). We can therefore apply Lemma B.2.1 to deduce that:

(1) there exists a finite SLDNF-derivation of $P \cup \{\leftarrow A'\}$ whose characteristic path is δ_C and whose resolvent is G'.

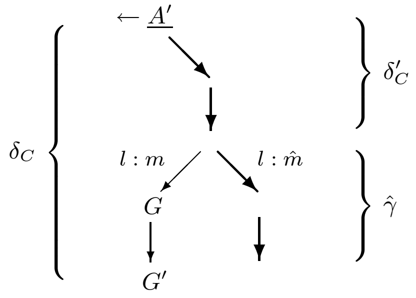


Fig. 13. Illustrating the proof of Lemma B.2.11.

Because $C \in \mathit{resultants}(\tau_A^*) \setminus \mathit{resultants}(\tau_A)$ we know that $\delta_C \in \tau_A^*$ and $\delta_C \notin \tau_A$. Furthermore $A' \in \gamma_P(A, \tau_A)$ implies by Lemma B.2.8 that, for some unfolding rule U, $\hat{\tau} = \mathit{chtree}(\leftarrow A', P, U) \subseteq \tau_A$.

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 B.2.10), that G' must also fail finitely.

If $\hat{\tau} \neq \emptyset$ then the largest prefix δ'_C of δ_C, such that for some $\hat{\gamma}$ we have $\delta'_C\hat{\gamma} \in \hat{\tau}$, must exist (the smallest one being $\langle\rangle$). By Lemma B.2.7 we know that no proper prefix of δ_C can be in τ_A^* (because $\delta_C \in \tau_A^*$), 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 nonempty. We also know, again by Lemma B.2.7, that δ'_C is a proper prefix of δ_C (because $\delta_C \in \tau_A^*$ and $\delta'_C\hat{\gamma} \in \tau_A^*$), i.e., $\delta_C = \delta'_C\langle l : m\rangle\delta''_C$. We can also see that $\hat{\gamma} \neq \langle l : m\rangle\delta''_C$ because $\delta'_C\hat{\gamma} \in \hat{\tau}$ while $\delta_C \notin \hat{\tau}$. This means that there is branching immediately after δ'_C (otherwise δ'_C is not the largest prefix of δ_C such that an extension of it is in $\hat{\tau}$). We even know that in $\delta_C = \delta'_C\langle l : m\rangle\delta''_C$ 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 $\hat{\gamma} = \langle l : \hat{m}\rangle\hat{\gamma}'$ with $\hat{m} \neq m$. The situation is summarized in Figure 13.

Let G be the goal obtained from the finite SLDNF-derivation of $P \cup \{\leftarrow A'\}$ whose characteristic path is $\delta'_C\langle l : m\rangle$ (this must exist because an SLDNF-derivation of $P \cup \{\leftarrow A'\}$ with characteristic path δ_C exists by point (1) above). In order to arrive at the characteristic tree $\hat{\tau}$ for $\leftarrow A'$ the unfolding rule U also had to reach the goal G, because $\hat{\tau}$ contains the characteristic path $\delta'_C\langle l : \hat{m}\rangle\hat{\gamma}'$ and G is "reached" via $\delta'_C\langle l : m\rangle$, a step which cannot be avoided by U if it wants to get as far as $\delta'_C\langle l : \hat{m}\rangle$. Furthermore, as neither $\delta'_C\langle l : m\rangle$ 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 $\delta'_C\langle l : m\rangle$, we know that a finite (possibly empty) SLDNF-derivation from G to G' exists and therefore, by persistence of failure (Lemma B.2.10), G' must also fail finitely. $\square$

We now present a correctness theorem for safe characteristic atoms.

Theorem B.2.12. *Let P be a normal program, G a goal, $\tilde{\mathcal{A}}$ a finite set of safe P-characteristic atoms and P' the partial deduction of P with respect to $\tilde{\mathcal{A}}$ and* *some ρ_α. If $\tilde{\mathcal{A}}$ is P-covered and if G is P-covered by $\tilde{\mathcal{A}}$ then the following hold:*

(1) $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ if and only if $P \cup \{G\}$ does.

(2) $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree if and only if $P \cup \{G\}$ does.

PROOF. The basic idea of the proof is as follows.

(1) First, we transform the characteristic atoms in $\tilde{\mathcal{A}}$ so as to make them unconstrained (which is possible by Lemma B.2.9). For the so obtained partial deduction P'' we can reuse Theorem B.1.1 to prove that P'' is totally correct.

(2) By construction, P'' will be a superset of P', i.e., $P'' = P' \cup P_{New}$, and we will show, mainly using Lemma B.2.11, that the clauses P_{New} can be safely removed without affecting the total correctness.

The details of the proof are elaborated in the following.

1. In order to make a characteristic atom (A, τ) in $\tilde{\mathcal{A}}$ unconstrained we have to add some characteristic paths to τ. We have that, for every $(A, \tau) \in \tilde{\mathcal{A}}$, the set of derivations $D_P(A, \tau)$ is safe. By Lemma B.2.9 we then know that for some unfolding rule U: $\tau \subseteq \mathit{chtree}(\leftarrow A, P, U)$. Let $\tau' = \mathit{chtree}(\leftarrow A, P, U) \setminus \tau$ 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 $(A, \mathit{chtree}(\leftarrow A, P, U))$). 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 $(A, \tau) \in \tilde{\mathcal{A}}$ the clauses $New_{(A,\tau)}$ to P' we obtain a partial deduction of P with respect to an unconstrained set $\tilde{\mathcal{A}}'$[17] and the renaming function ρ_α (where we extend α so that $\alpha(A, \tau \cup \tau') = \alpha(A, \tau)$). Note that, every concretization of (A, τ) is also a concretization of $(A, \tau \cup \tau')$ (because $(A, \tau \cup \tau')$ is unconstrained, and thus *any* instance of A is a concretization).

Unfortunately, although G remains P-covered by $\tilde{\mathcal{A}}'$, $\tilde{\mathcal{A}}'$ is not necessarily P-covered any longer. The reason is that new uncovered leaf atoms can arise inside $New_{(A,\tau)}$. Let UC be these uncovered atoms. To arrive at a P-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 $\tilde{\mathcal{A}}'$, where $X_1, \ldots, X_n$ are distinct variables. This will give us the new set $\tilde{\mathcal{A}}'' \supseteq \tilde{\mathcal{A}}'$ (and we extend α and ρ_α in an arbitrary manner for the elements in $\tilde{\mathcal{A}}'' \setminus \tilde{\mathcal{A}}'$). Let P'' be the partial deduction of P with respect to $\tilde{\mathcal{A}}''$ and ρ_α. Now $\tilde{\mathcal{A}}''$ is trivially P-covered and we can apply the correctness Theorem B.1.1 to deduce that the computations of $P'' \cup \{\rho_\alpha(G)\}$ are totally correct with respect to the computations in $P \cup \{G\}$.

2. 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, i.e.:

—$P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ if and only if $P' \cup \{\rho_\alpha(G)\}$ does.

[17]As in the proof of Theorem B.1.1, $\tilde{\mathcal{A}}'$ might actually be a multiset. However, this poses no problems, as all results so far also hold for multisets of characteristic atoms. Alternatively, one could add an extra unused argument to all predicates and ensure that all elements in $\tilde{\mathcal{A}}$ have a different variable in that position, thus guaranteeing that $\tilde{\mathcal{A}}'$ is an ordinary set.

—$P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree if and only if $P' \cup \{\rho_\alpha(G)\}$ does.

Combined with point 1., this is sufficient to establish that P' is also totally correct with respect to P. We do the proof by induction of the rank of the SLDNF-derivations and trees.

Induction Hypothesis:

—if $P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation of rank k with computed answer θ then $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ.

—if $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation of rank k with computed answer θ then $P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation (of rank k) with computed answer θ.

—if $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree of rank k then $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree (of rank k).

—if $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree of rank k then $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree.

Base Case: Because $P' \subseteq P''$, we have that whenever $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLD$^+$-tree so does $P' \cup \{\rho_\alpha(G)\}$, and whenever $P' \cup \{\rho_\alpha(G)\}$ has a SLD$^+$-refutation with computed answer θ so does $P'' \cup \{\rho_\alpha(G)\}$. We now show that every SLD$^+$-derivation of $P'' \cup \{\rho_\alpha(G)\}$, which uses at least a clause in $P'' \setminus P'$, fails finitely. This will ensure that $P'' \cup \{\rho_\alpha(G)\}$ cannot have any additional computed answer and that it fails finitely if and only if $P' \cup \{\rho_\alpha(G)\}$ does.
Let D be an SLD$^+$-derivation of $P'' \cup \{\rho_\alpha(G)\}$ which uses at least one clause in $P'' \setminus P'$. Let D' be the largest prefix SLD$^+$-derivation of D such that D' uses only clauses within P'. Let RG' be the last goal of D'. We can apply Lemma B.2.6 to deduce that there exists an SLD$^+$-derivation of $P \cup \{G\}$ leading to RG such that RG' is an admissible renaming of RG under α in P and such that RG is P-covered by $\tilde{\mathcal{A}}''$. 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').

$$\begin{tikzcd}
\rho_\alpha(G) \arrow[d, "P'"'] & G \arrow[l, dotted, "\rho_\alpha"'] \arrow[d, "P"] \\
RG' & RG \arrow[l, dotted, "\rho'_\alpha"]
\end{tikzcd}$$

Because RG' is an admissible renaming of RG, we have $p(\bar{t}) \in \gamma_P(A, \tau)$ where $\rho'_\alpha(p(\bar{t})) = \alpha((A, \tau))\sigma'$ for some σ', ρ'_α and (A, τ). Furthermore, we now that $p(\bar{t}) \in \gamma_P(A, \tau')$ for some $(A, \tau') \in \tilde{\mathcal{A}}$ with $\tau' \subseteq \tau$, because the elements in $\tilde{\mathcal{A}}'' \setminus \tilde{\mathcal{A}}'$ cannot be reached from the clauses in P'. We can therefore apply Lemma B.2.11 to deduce that $\leftarrow p(\bar{t})$, and therefore also RG, fails finitely in P. We can now apply the correctness Theorem B.1.1 for the goal $\leftarrow p(\bar{t})$ (it is possible to apply this theorem because $\leftarrow p(\bar{t})$ is P-covered by $\tilde{\mathcal{A}}$) to deduce that $\leftarrow \rho_\alpha(p(\bar{t}))$ also fails finitely in P''.

Induction Step: Let us now prove the hypothesis for SLDNF-derivations and trees of rank $k+1$. Because $P' \subseteq P''$, and by the induction hypothesis (because all subderivations and subtrees for negative literals have rank $\leq k$), we have that whenever $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree of rank $k+1$ so does $P' \cup \{\rho_\alpha(G)\}$, and whenever $P' \cup \{\rho_\alpha(G)\}$ has a SLDNF-refutation or rank $k+1$ with computed answer θ so does $P'' \cup \{\rho_\alpha(G)\}$. Similar to the base case, if we show that every SLDNF-derivation D of rank $k+1$ of $P'' \cup \{\rho_\alpha(G)\}$ which uses at least a clause in $P'' \setminus P'$ fails finitely, then $P'' \cup \{\rho_\alpha(G)\}$ cannot have any additional computed answer (over P) and it fails finitely if and only if $P' \cup \{\rho_\alpha(G)\}$ does. This can be done in exactly the same manner as for the base case, because Lemmas B.2.6, B.2.11 and Theorem B.1.1 hold for SLDNF-derivations and trees of any rank. □

B.3 Correctness for Unrestricted Characteristic Atoms

We are finally in the position to prove the general correctness Theorem 3.3.2 for partial deductions of safe and unsafe P-characteristic atoms.

Proof of Theorem 3.3.2. For every $(A, \tau) \in \tilde{\mathcal{A}}$ let us remove the derivation steps from τ which correspond to the selection of a nonground negative literal in $D_P(A, \tau)$, resulting in a modified characteristic tree τ'. Note that, trivially, any concretization of (A, τ) is also a concretization of (A, τ') (if we can build and SLDNF-tree for $P \cup \{\leftarrow A\theta\}$ with characteristic tree τ we can also construct an SLDNF-tree with characteristic tree τ' by simply not selecting the offending negative literals). By substituting (A, τ') for (A, τ) in $\tilde{\mathcal{A}}$ we obtain a set $\tilde{\mathcal{A}}'$ of safe characteristic atoms.[18] Every clause in the partial deduction with respect to $\tilde{\mathcal{A}}'$ and ρ_α can be obtained from a clause of P' by adding the negative literals which are no longer selected. So, just like in the proof of Theorem B.2.12, we might have to add characteristic atoms to $\tilde{\mathcal{A}}'$ (to cover all these negative literals), in order to arrive at a P-covered set, giving the new set of characteristic atoms $\tilde{\mathcal{A}}'' \supseteq \tilde{\mathcal{A}}'$. As in the proof of Theorem B.2.12 we also extend α so that $\alpha(A, \tau') = \alpha(A, \tau)$ (and we extend α and ρ_α in an arbitrary manner for the elements in $\tilde{\mathcal{A}}'' \setminus \tilde{\mathcal{A}}'$).

Let P'' be the partial deduction of P with respect to $\tilde{\mathcal{A}}''$ and ρ_α. We can apply Theorem B.2.12 to deduce that the computations of $P'' \cup \{\leftarrow \rho_\alpha(G)\}$ are totally correct with respect to $P \cup \{\leftarrow G\}$.

We will now establish total correctness of P' (with respect to P) by proving that:

—$P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ if and only if $P' \cup \{\rho_\alpha(G)\}$ does.

—$P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree if and only if $P' \cup \{\rho_\alpha(G)\}$ does.

We will do this proof by induction on the rank of the SLDNF-refutations and trees.

Induction Hypothesis:

—if $P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation of rank k with computed answer θ then $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation (of rank k) with computed answer θ.

[18]As in the proofs of Theorems B.1.1 and Theorem B.2.12, $\tilde{\mathcal{A}}'$ might actually be a multiset. Again, this poses no problems, as all results so far also hold for multisets of characteristic atoms. Alternatively, one could add an extra unused argument to all predicates and ensure that all elements in $\tilde{\mathcal{A}}$ have a different variable in that position, thus guaranteeing that $\tilde{\mathcal{A}}'$ is an ordinary set.

—if $P' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation of rank k with computed answer θ then $P'' \cup \{\rho_\alpha(G)\}$ has an SLDNF-refutation with computed answer θ.

—if $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree of rank k then $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree.

—if $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree of rank k then $P'' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLDNF-tree (of rank k).

Base Case: For every SLD$^+$-derivation in P' there exists a corresponding SLD$^+$-derivation in P'', with however many additional negative literals in the resolvent. Thus, every computed answer of $P'' \cup \{\rho_\alpha(G)\}$ (via an SLD$^+$-refutation) is also a computed answer of $P' \cup \{\rho_\alpha(G)\}$ (via an SLD$^+$-refutation). Also, if $P' \cup \{\rho_\alpha(G)\}$ has a finitely failed SLD$^+$-tree then $P'' \cup \{\rho_\alpha(G)\}$ also has a finitely failed SLD$^+$-tree. We will now show that these additional negative literals always succeed. Thus, if for every derivation in P'' we impose the (fair) condition that the additional negative literals are immediately selected, 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 by showing that the additional negative literals always succeed, we establish the base case.

Let G_1' be a goal which is an admissible renaming (under α 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 concretization of an element in $\tilde{\mathcal{A}}$ we know that it must be the instance of a precise concretization A. For this concretization 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 B.2.12) select them in P'' and can thus construct a corresponding derivation in P'.

Induction Step: The induction step is very similar to the base case and can basically be obtained by replacing SLD$^+$ by SLDNF of rank $k+1$. □

C. PROOF OF THEOREM 3.4.5

We recall the following well-founded measure function defined by Gallagher and Bruynooghe [1991] (also in the extended version of Martens and Gallagher [1995]):

Definition C.1 ($s(.)$, $h(.)$). 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 $h(t) \geq 0$ for any expression t and $h(t) > 0$ for any non-variable expression t. The following important lemma is proven for $h(.)$ by Gallagher and Bruynooghe [1990] (see also Martens and Gallagher [1995]).

LEMMA C.2. *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 C.3 (*Weight Vector*). Let $\tilde{\mathcal{A}}$ 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 $\tilde{\mathcal{A}}$ with respect to T by $hvec_T(\tilde{\mathcal{A}}) = \langle w_1, \ldots, w_n \rangle$ where

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

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

—$s_1 \equiv_S s_2$ if and only if $s_1 \leq s_2 \wedge s_2 \leq s_1$ and $s_1 <_S s_2$ if and only if $s_1 \leq s_2 \wedge s_2 \not\leq s_1$.

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

PROPOSITION C.4. *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 $\tilde{\mathcal{A}}$ and $\tilde{\mathcal{B}}$, such that the characteristic trees of their elements are in T, we have that one of the following holds:*

—$chmsg(\tilde{\mathcal{A}} \cup \tilde{\mathcal{B}}) = \tilde{\mathcal{A}}$ (up to variable renaming) or
—$hvec_T(chmsg(\tilde{\mathcal{A}} \cup \tilde{\mathcal{B}})) < hvec_T(\tilde{\mathcal{A}})$.

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

—$\{msg(\tilde{\mathcal{A}}|_{\tau_i} \cup \tilde{\mathcal{B}}|_{\tau_i})\} = \tilde{\mathcal{A}}|_{\tau_i}$ (up to variable renaming). In this case the abstraction operator performs no modification for τ_i and $v_i = w_i$.
—$\{msg(\tilde{\mathcal{A}}|_{\tau_i} \cup \tilde{\mathcal{B}}|_{\tau_i})\} = \{M\} \neq \tilde{\mathcal{A}}|_{\tau_i}$ (up to variable renaming). In this case $(M, \tau_i) \in chmsg(\tilde{\mathcal{A}} \cup \tilde{\mathcal{B}})$, $v_i = h(M)$ and there are three possibilities:
  —$\tilde{\mathcal{A}}|_{\tau_i} = \emptyset$. In this case $v_i < w_i = \infty$.
  —$\tilde{\mathcal{A}}|_{\tau_i} = \{A\}$ 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 $v_i < w_i$.
  —$\#(\tilde{\mathcal{A}}|_{\tau_i}) > 1$. In this case M is more general (but not necessarily strictly more general) than any atom in $\tilde{\mathcal{A}}|_{\tau_i}$ 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 3.4.5. *If the number of distinct characteristic trees is finite then Algorithm 3.4.3 terminates and generates a partial deduction satisfying the requirements of Theorem 3.3.2 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 concretizations of at least one characteristic atom in $\tilde{\mathcal{A}}_k$, i.e., $\tilde{\mathcal{A}}_k$ is P-covered. Furthermore *chmsg*() always generates more general characteristic atoms (even in the sense that any precise concretization of an atom in $\tilde{\mathcal{A}}_i$ is a precise concretization of an atom in $\tilde{\mathcal{A}}_{i+1}$—this follows immediately from Definitions 3.1.3 and 3.4.1). Hence, because any instance of an atom in the goal G is a precise concretization of a characteristic atom in $\tilde{\mathcal{A}}_0$, the conditions of Theorem 3.3.2 are satisfied for goals G' whose atoms are instances of atoms in G, i.e., G' is P-covered by $\tilde{\mathcal{A}}_k$. Finally, termination is guaranteed by Proposition C.4, given that the number of distinct characteristic trees is finite. □

D. ASSORTED SHORT PROOFS

Lemma 4.3.3. *Let τ_1, τ_2 be two characteristic trees. Then $\tau_1 \equiv_\tau \tau_2$ if and only if $\tau_1 = \tau_2$.*

Proof. The if part is obvious because δ and δ' can be taken as prefixes of themselves for the two points of Definition 4.3.1.

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 4.3.1, that an extension $\delta_x = \delta\gamma$ of δ 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 $\delta_x = \delta_s\gamma'\gamma$ and $\delta_x \neq \delta_s$ (because $\delta \notin \tau_2$). But this is impossible by Lemma B.2.7. □

Proposition 4.3.13. *Let (A, τ_A) and (B, τ_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 concretization of (B, τ_B). By Definition 3.1.3, there must be an unfolding rule U such that *chtree*$(\leftarrow C, P, U) = \tau_B$. By Corollary 4.3.11 we can find an unfolding rule U', such that *chtree*$(\leftarrow C, P, U') = \tau_A$. Furthermore, by Definition 4.3.12, B is an instance of A and therefore C is also an instance of A. We can conclude that C is also a precise concretization of (A, τ_A). In other words, any precise concretization of (B, τ_B) is also a precise concretization of (A, τ_A). The result for general concretizations follows immediately from this by Definition 3.1.3. □

Proposition 4.4.4. *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 Dershowitz and Jouannaud [1990]. Let $\leq$ be a relation on a set S of functors (of arity ≥ 0). Then the *homeomorphic embedding over* $\leq$ 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 $\forall i \in \{1, \ldots, n\} : s_i \leq_{emb} t_i$.

The definition by Dershowitz and Jouannaud [1990] actually also allows functors of variable arity, but we will not need this in the following. 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 the results of Higman [1952] and Kruskal [1960] (see also Dershowitz and Jouannaud [1990]), the homeomorphic embedding $\leq_{emb}$ over $\leq$, which is by definition identical to $\trianglelefteq$, is a WQO on the set of expressions. □

Proposition 4.4.11. *Let $\tilde{\mathcal{A}}$ be a set of P-characteristic atoms. Then $\trianglelefteq^*_{ca}$ is a well-quasi relation on $\tilde{\mathcal{A}}$.*

Proof. Let $\mathcal{E}$ be the set of expressions over the finite alphabet $\mathcal{A}_P$. By Theorem 4.4.7, $\trianglelefteq^*$ is a WQR on $\mathcal{E}$. Let $\mathcal{F}$ be the alphabet containing just one binary functor $ca/2$ as well as all the elements of $\mathcal{E}$ as constant symbols. Let us extend $\trianglelefteq^*$ from $\mathcal{E}$ to $\mathcal{F}$ by (only) adding that $ca \trianglelefteq^* ca$. $\trianglelefteq^*$ is still a WQR on $\mathcal{F}$, and hence, by the results of Higman [1952; Kruskal [1960] (see also Dershowitz and Jouannaud [1990]), the homeomorphic embedding $\trianglelefteq^*_{emb}$ over $\trianglelefteq^*$ (see proof of Proposition 4.4.4) on terms constructed from $\mathcal{F}$ is also a WQR. 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, $\trianglelefteq^*_{emb}$ coincides with Definition 4.4.10 (i.e., $ca(A_1, \lceil\tau_1\rceil) \trianglelefteq^*_{emb} ca(A_2, \lceil\tau_2\rceil)$ if and only if $(A_1, \tau_1) \trianglelefteq^*_{ca} (A_2, \tau_2)$) and hence $\tilde{\mathcal{A}}, \trianglelefteq^*_{ca}$ is a well-quasi relation. □

E. PROOF OF PROPOSITION 4.3.9

We first establish that Algorithm 4.3.6 indeed produces a proper characteristic tree as output.

Lemma E.1. *Any τ_i arising during the execution of Algorithm 4.3.6 satisfies the property that $\delta \in \tau_i \Rightarrow \delta \in prefix(\tau_A) \cap prefix(\tau_B)$.*

Proof. We prove this by induction on i.

Induction Hypothesis: For $i \leq k \leq max$ (where max is the maximum value that i takes during the execution of Algorithm 4.3.6) we have that $\delta \in \tau_i \Rightarrow \delta \in prefix(\tau_A) \cap prefix(\tau_B)$.

Base Case: For $i = 0$ we have $\tau_i = \{\langle\rangle\}$ and trivially $\langle\rangle \in prefix(\tau_A) \cap prefix(\tau_B)$ because $\tau_A \neq \emptyset$ and $\tau_B \neq \emptyset$.

Induction Step: Let $i = k+1 \leq max$. For $\delta \in \tau_{k+1}$ we either have $\delta \in \tau_k$, and we can apply the induction hypothesis to prove the induction step, or we have $\delta = \delta'\langle l : m\rangle$ with $l : m \in top(\tau_A \downarrow \delta')\}$ and $l : m \in top(\tau_B \downarrow \delta')\}$. By definition this implies that $\delta'\langle l : m\rangle\gamma \in \tau_A$ for some γ, i.e., $\delta'\langle l : m\rangle \in prefix(\tau_A)$. The same holds for τ_B, i.e., $\delta'\langle l : m\rangle \in prefix(\tau_B)$, and we have proven the induction step. □

LEMMA 4.3.8. *Algorithm 4.3.6 terminates and produces as output a characteristic tree τ such that if $chtree(G, P, U) = \tau_A$ (respectively τ_B), then for some U', $chtree(G, P, U') = \tau$. The same holds for any τ_i arising during the execution of Algorithm 4.3.6.*

PROOF. Termination of Algorithm 4.3.6 is obvious as τ_A and τ_B are finite, τ cannot grow larger (e.g., in terms of the sum of the lengths of the characteristic paths) than τ_A or τ_B and τ_{i+1} is strictly larger than τ_i. For the remaining part of the lemma we proceed by induction. Note that Algorithm 4.3.6 is symmetrical with respect to τ_A and τ_B and it suffices to show the result for τ_A.

Induction Hypothesis: For $i \leq k$ we have that if $chtree(G, P, U) = \tau_A$ then for some U' $chtree(G, P, U') = \tau_i$.

Base Case: $\tau_0 = \{\langle\rangle\}$ is a characteristic tree of every goal and the property trivially holds.

Induction Step: Let $i = k+1$ and let $\tau_{k+1} = (\tau_k \setminus \{\delta\}) \cup \{\delta\langle l_1 : m_1\rangle, \ldots, \delta\langle l_n : m_n\rangle\}$ where $top(\tau_A \downarrow \delta)\} = \{l_1 : m_1, \ldots, l_n : m_n\}$. By Lemma E.1 we know that $\delta\langle l_j : m_j\rangle \in prefix(\tau_A)$. Because τ_A is a characteristic tree we therefore know that $l_1 = \ldots = l_n = l$. By the induction hypothesis we know that for some U' $chtree(G, P, U') = \tau_k$. Let G' be the goal of the SLDNF-derivation of $P \cup \{G\}$ whose characteristic path is δ (which must exist because $chtree(G, P, U') = \tau_k$) and let T be the SLDNF-tree for $P \cup \{G\}$ whose characteristic tree is τ_k. If we expand T by selecting the literal at position l in G we obtain a SLDNF-tree T' whose characteristic tree is $\tau' = (\tau_k \setminus \{\delta\}) \cup \{\delta\langle l : m_1\rangle, \ldots, \delta\langle l : m_n\rangle\} \cup \{\delta\langle l : m'_1\rangle, \ldots, \delta\langle l : m'_q\rangle\}$ (because $\delta\langle l_j : m_j\rangle \in prefix(\tau_A)$ and because additional clauses might match). If a negative literal is selected we have that $n = 1$, $q = 0$, $\tau_{k+1} = \tau'$ and the induction step holds. If a positive literal is selected then we can further unfold the goals associated with derivations of $P \cup \{G\}$ whose characteristic paths are $\delta\langle l : m'_j\rangle$ and make them fail finitely (because $\delta\langle l : m'_j\rangle \notin \tau_A$). In both cases we can come up with an unfolding rule U'' such that $chtree(G, P, U'') = \tau_{k+1}$ □

PROPOSITION 4.3.9. *Let τ_A, τ_B be two nonempty characteristic trees. Then the output τ of Algorithm 4.3.6 is the (unique) most specific generalization of τ_A and τ_B.*

PROOF. We first prove that each τ_i arising while executing Algorithm 4.3.6 is a generalization of both τ_A and τ_B. For this we have to show that the two points of Definition 4.3.1 are satisfied; the fact that τ and all τ_i are proper characteristic trees (a fact required by Definition 4.3.1) is already established in Lemma 4.3.8.

(1) We have to show that $\delta \in \tau_i \Rightarrow \delta \in prefix(\tau_A) \cap prefix(\tau_B)$. But this is already proven in Lemma E.1.

(2) We have to show that $\delta' \in \tau_A \cup \tau_B \Rightarrow \exists \delta \in \mathit{prefix}(\{\delta'\})$ such that $\delta \in \tau_i$. Again we prove this by induction on i.

Induction Hypothesis: For $i \leq k \leq max$ (where max is the maximum value that i takes during the execution of Algorithm 4.3.6) we have that $\delta' \in \tau_A \cup \tau_B \Rightarrow \exists \delta \in \mathit{prefix}(\{\delta'\})$ such that $\delta \in \tau_i$.

Base Case: For all δ' we have that $\langle\rangle \in \mathit{prefix}(\{\delta'\})$ and $\langle\rangle \in \tau_0$.

Induction Step: Let $i = k + 1 \leq max$ and take a $\delta' \in \tau_A \cup \tau_B$. Let $\delta \in \mathit{prefix}(\{\delta'\})$ be such that $\delta \in \tau_k$, which must exist by the induction hypothesis. Either we have that $\delta \in \tau_{k+1}$, and the induction step holds for δ, or $\delta\langle l : m\rangle \in \tau_{k+1}$ for every $l : m \in \mathit{top}(\tau_A \downarrow \delta)$ where also $\mathit{top}(\tau_A \downarrow \delta) = \mathit{top}(\tau_B \downarrow \delta) \neq \emptyset$. Let γ be such that $\delta' = \delta\gamma$ (which must exist because $\delta \in \mathit{prefix}(\{\delta'\})$). By Lemma B.2.7 and because $\mathit{top}(\tau_A \downarrow \delta)$ $\mathit{top}(\tau_B \downarrow \delta) \neq \emptyset$ we know that γ cannot be empty and thus $\gamma = \langle l : m\rangle\gamma'$ where $l : m \in \mathit{top}(\tau_A \downarrow \delta)$. Hence, we have found a prefix $\delta\langle l : m\rangle$ of δ' such that $\delta\langle l : m\rangle \in \tau_{k+1}$.

We have thus proven that every τ_i, and thus also the output τ, is a generalization of both τ_A and τ_B. We will now prove that τ is indeed the most specific generalization. For this we will prove that, whenever $\tau^* \preceq_\tau \tau_A$ and $\tau^* \preceq_\tau \tau_B$ then $\tau^* \preceq_\tau \tau$. This is sufficient to show that τ is a most specific generalization; uniqueness then follows from Lemma 4.3.3. To establish $\tau^* \preceq_\tau \tau$ we now show that the two points of Definition 4.3.1 are satisfied.

(1) Let $\delta \in \tau^*$. We have to prove that $\delta \in \mathit{prefix}(\tau)$. As $\tau^* \preceq_\tau \tau_A$ and $\tau^* \preceq_\tau \tau_B$, we have by Definition 4.3.1 that $\delta \in \mathit{prefix}(\tau_A) \cap \mathit{prefix}(\tau_B)$. In order to prove that $\delta \in \mathit{prefix}(\tau)$ it is sufficient to prove that if $\delta = \delta'\langle l : m\rangle\delta''$ then $\mathit{top}(\tau_A \downarrow \delta') = \mathit{top}(\tau_B \downarrow \delta') \neq \emptyset$. Indeed, once we have proven this statement, we can proceed by induction (starting out with $\delta' = \langle\rangle$) to show that for some i we have $\delta \in \tau_i$ (because $\mathit{top}(\tau_A \downarrow \delta') = \mathit{top}(\tau_B \downarrow \delta') \neq \emptyset$ and $\delta \in \mathit{prefix}(\tau_A) \cap \mathit{prefix}(\tau_B)$ ensure that if $\delta' \in \tau_j$ then $\delta'\langle l : m\rangle \in \tau_{j+k}$ for some $k > 0$). From then on, the algorithm will either leave δ unchanged or extend it, and thus we will have established that $\delta \in \mathit{prefix}(\tau)$.

   (a) Let us first assume that $\mathit{top}(\tau_A \downarrow \delta') = \mathit{top}(\tau_B \downarrow \delta') = \emptyset$ and show that it leads to a contradiction. By this assumption we know that no extension of δ' can be in τ_A or τ_B, which is in contradiction with $\delta = \delta'\langle l : m\rangle\delta''$ $\in \mathit{prefix}(\tau_A) \cap \mathit{prefix}(\tau_B)$.

   (b) Now let us assume that $\mathit{top}(\tau_A \downarrow \delta') \neq \mathit{top}(\tau_B \downarrow \delta')$. This means that, for some $l : m'$, $\delta'\langle l : m'\rangle \in \mathit{prefix}(\tau_A)$ and $\delta'\langle l : m'\rangle \notin \mathit{prefix}(\tau_B)$ (otherwise we reverse the roles of τ_A and τ_B). We can thus deduce that, for any γ (even $\gamma = \langle\rangle$), $\delta'\langle l : m'\rangle\gamma \notin \tau^*$ (because otherwise, by point 1 of Definition 4.3.1, $\tau^* \not\preceq_\tau \tau_B$). Thus, in order to satisfy point 2 of Definition 4.3.1 for $\tau^* \preceq_\tau \tau_A$, we know that some prefix of δ' must be in τ^* (because $\delta'\langle l : m'\rangle\gamma' \in \tau_A$ for some γ'). But this is impossible by Lemma 4.3.3, because $\delta = \delta'\langle l : m\rangle\delta'' \in \tau^*$.

   So, assuming either $\mathit{top}(\tau_A \downarrow \delta') = \mathit{top}(\tau_B \downarrow \delta') = \emptyset$ or $\mathit{top}(\tau_A \downarrow \delta') \neq \mathit{top}(\tau_B \downarrow \delta')$ leads to a contradiction and we have established the desired result.

(2) Let $\delta' \in \tau$. We have to prove point 2 of Definition 4.3.1 (for $\tau^* \preceq_\tau \tau$), namely that $\exists \delta'' \in \mathit{prefix}(\{\delta'\})$ such that $\delta'' \in \tau^*$. We have already proven that τ is

a generalization of both τ_A and τ_B, and thus $\delta' \in \mathit{prefix}(\tau_A) \cap \mathit{prefix}(\tau_B)$. We also know, by the while-condition in Algorithm 4.3.6, that either

(a) $\mathit{top}(\tau_A \downarrow \delta') = \mathit{top}(\tau_B \downarrow \delta') = \emptyset$ or
(b) $\mathit{top}(\tau_A \downarrow \delta') \neq \mathit{top}(\tau_B \downarrow \delta')$.

Let us examine each of these cases.

(a) In that case we have $\delta' \in \tau_A \cap \tau_B$ which implies, by point 2 of Definition 4.3.1, that $\exists \delta'' \in \mathit{prefix}(\{\delta'\})$ such that $\delta'' \in \tau^*$ because $\tau^* \preceq_\tau \tau_A$ and $\tau^* \preceq_\tau \tau_B$.
(b) In that case we can find $\delta'\langle l : m\rangle\gamma_A \in \tau_A$ and $l : m \notin \mathit{top}(\tau_B \downarrow \delta')$ (otherwise we reverse the roles of τ_A and τ_B). As $\tau^* \preceq_\tau \tau_A$, we know by point 2 of Definition 4.3.1, that a prefix δ'' of $\delta'\langle l : m\rangle\gamma_A$ is in τ^*. Finally, δ'' must be a prefix of δ' (otherwise $\delta'' = \delta'\langle l : m\rangle\gamma'_A \in \tau^*$ and, as $\delta'\langle l : m\rangle\gamma'_A \notin \mathit{prefix}(\tau_B)$ because $l : m \notin \mathit{top}(\tau_B \downarrow \delta')$, we cannot have $\tau^* \preceq_\tau \tau_B$ by point 1 of Definition 4.3.1). □

F. PROOF OF THEOREM 4.4.7

The following definitions and lemmas are needed to prove Theorem 4.4.7.

An order $>_S$ is called a *well-founded order (WFO)* on S if and only if 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 F.1 (*WQR from WFO*). *Let $<_V$ be a well-founded order on V. Then $\preceq_V$, defined by $v_1 \preceq_V v_2$ if and only if $v_1 \not>_V v_2$, is a WQR 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 the definition of a well-founded order and $\preceq_V$ must be a WQR on V. □

Lemma F.2. *Let $\preceq_V$ be a WQR 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 $\rho_{v' \preceq_V \circ}$ the subsequence of ρ consisting of all elements v'' which satisfy $v' \preceq_V v''$. Similarly, we denote by $\rho_{v' \not\preceq_V \circ}$ 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 $\sigma^{i+1} = \rho^i_{v'_i \not\preceq_V \circ}$

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 $\rho^i_{v'_i \not\preceq_V \circ}$; 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 WQR on V and we have a contradiction.
We can now prove point 2. (This point is actually almost a corollary of point iv of Theorem 2.1 in Higman [1952]. But as no formal proof is provided by Higman [1952], we include one for completeness.) Let us construct $\sigma^* = v^*_1, v^*_2, \ldots$ inductively as follows:

—$\sigma^0 = \sigma$

—if $\sigma^i = r_1, r_2, \ldots$ then $v^*_{i+1} = r_k$ and $\sigma^{i+1} = \rho^i_{r_k \preceq_V \circ}$ where k is the first index satisfying the requirements of point 1 for the sequence σ^i (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 $\rho^i_{r_k \preceq_V \circ}$ is infinite and σ^* is thus an infinite sequence which, by construction, satisfies $v^*_i \preceq_V v^*_j$ for all $i < j$. □

Lemma F.3 (Combination of WQR). *Let $\preceq^1_V$ and $\preceq^2_V$ be WQR's on V. Then the relation $\preceq_V$ defined by $v_1 \preceq_V v_2$ if and only if $v_1 \preceq^1_V v_2$ and $v_1 \preceq^2_V v_2$, is also a WQR on V.*

Proof. (This lemma is actually almost a corollary of Theorem 2.3 in Higman [1952]. But as no formal proof is given in Higman [1952], we include one for completeness.) Let σ be any infinite sequence of elements from V. We can apply point 2 of Lemma F.2 to obtain the infinite subsequence $\sigma^* = v^*_1, v^*_2, \ldots$ of σ such that for all $i < j$ we have $v^*_i \preceq^1_V v^*_j$. Now, as $\preceq^2_V$ is also a WQR we know that, for some $i < j$, $v^*_i \preceq^2_V v^*_j$ holds as well. Hence, for these particular indices, $v^*_i \preceq_V v^*_j$ and $\preceq_V$ satisfies the requirements of a WQR on V. □

We can now actually prove Theorem 4.4.7.

Proof of Theorem 4.4.7. $\trianglelefteq^*$ can be expressed as a combination of two relations on expressions: $\trianglelefteq$ and $\preceq_{NotStrictInst}$ where $A \preceq_{NotStrictInst} B$ if and only if $B \not\prec A$ (i.e., B is not strictly more general than A or equivalently A is not a strict instance of B). We know that $\prec$ is a well-founded order on expressions (see e.g., Lemma C.2 in Appendix C). Hence by Lemma F.1 $\preceq_{NotStrictInst}$ is a WQR on expressions. By Proposition 4.4.4 we also have that $\trianglelefteq$ is a WQO on expressions (given a finite underlying alphabet). Hence we can apply Lemma F.3 to deduce that $\trianglelefteq^*$ is also a WQR on expressions over a finite alphabet. □

G. PROOF OF PROPOSITION 4.4.9

Recall that $\tau \downarrow \delta = \{\gamma \mid \delta\gamma \in \tau\}$ and *prefix*$(\tau) = \{\delta \mid \exists\gamma$ such that $\delta\gamma \in \tau\}$. The following lemma will prove useful to establish the existence of a mapping $\lceil . \rceil$.

Lemma G.1. *Let τ_1 and τ_2 be two characteristic trees and let $\delta \in$ prefix(τ_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, by point 1 of Definition 4.3.1, $\delta\delta' \in$ *prefix*(τ_2) because $\tau_1 \preceq_\tau \tau_2$. Thus $\delta' \in$ *prefix*$(\tau_2 \downarrow \delta)$ and point 1 of Definition 4.3.1 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, by point 2 of Definition 4.3.1, $\exists \hat{\delta} \in \mathit{prefix}(\{\delta\delta'\})$ such that $\hat{\delta} \in \tau_1$. Now, because $\delta \in \mathit{prefix}(\tau_1)$ we know that $\hat{\delta}$ must have the form $\hat{\delta} = \delta\gamma$ (otherwise we arrive at a contradiction with Lemma B.2.7) where $\gamma \in \mathit{prefix}(\{\delta'\})$. Thus $\gamma \in \tau_1 \downarrow \delta$ (because $\delta\gamma \in \tau_1$) and also point 2 of Definition 4.3.1 is verified for $\tau_1 \downarrow \delta$ and $\tau_2 \downarrow \delta$. □

Proof of Proposition 4.4.9. Within Definition 4.4.8, the strict monotonicity condition is slightly tricky, but it can be satisfied by representing leaves of the characteristic tree by variables. First, recall that $\mathit{top}(\tau) = \{l : m \mid \langle l : m\rangle \in \mathit{prefix}(\tau)\}$. Let us now define the representation $\lceil \tau \rceil$ of a nonempty characteristic tree τ inductively as follows (using a binary functor m to represent clause matches as well as the usual functors for representing lists):

—$\lceil \tau \rceil = X$ where X is a fresh variable **if** $\mathit{top}(\tau) = \emptyset$

—$\lceil \tau \rceil = [m(m_1, \lceil \tau \downarrow \langle l : m_1\rangle \rceil), \ldots, m(m_k, \lceil \tau \downarrow \langle l : m_k\rangle \rceil)]$ **if** $\mathit{top}(\tau) = \{l : m_1, \ldots, l : m_k\}$ and where $m_1 < \ldots < m_k$.

For example, using the above definition, we have

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

and

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

Note that $\{\langle 1:3\rangle\} \prec_\tau \{\langle 1:3, 2:4\rangle\}$ and indeed $\lceil \{\langle 1:3\rangle\} \rceil \prec_\tau \lceil \{\langle 1:3, 2:4\rangle\} \rceil$. Also note that, because there are only finitely many clause numbers, these terms can be expressed using a finite alphabet.[19]

Note that if $\tau_1 \prec_\tau \tau_2$ we immediately have by Definition 4.3.1 that $\tau_1 \neq \emptyset$ and $\tau_2 \neq \emptyset$. It is therefore sufficient to prove strict monotonicity for two nonempty characteristic trees $\tau_1 \prec_\tau \tau_2$. We will prove this by induction on the depth of τ_1, where the depth is the length of the longest characteristic path in τ_1. We also have to perform an auxiliary induction, showing that $\lceil \tau_1 \rceil \preceq_\tau \lceil \tau_2 \rceil$ whenever $\tau_1 \preceq_\tau \tau_2$.

Induction Hypothesis: For all characteristic trees τ_1 of depth $\leq d$ we have that $\lceil \tau_1 \rceil \prec_\tau \lceil \tau_2 \rceil$ whenever $\tau_1 \prec_\tau \tau_2$ and $\lceil \tau_1 \rceil \preceq_\tau \lceil \tau_2 \rceil$ whenever $\tau_1 \preceq_\tau \tau_2$

Base Case: τ_1 has a depth of 0, i.e., $\mathit{top}(\tau_1) = \emptyset$. This implies that $\lceil \tau_1 \rceil$ is a fresh variable X. If we have $\tau_1 \prec_\tau \tau_2$ then $\mathit{top}(\tau_2) \neq \emptyset$ and $\lceil \tau_2 \rceil$ will be a strict instance of X, i.e., $\lceil \tau_1 \rceil \prec \lceil \tau_2 \rceil$. If we just have $\tau_1 \preceq_\tau \tau_2$ we still have $\lceil \tau_1 \rceil \preceq \lceil \tau_2 \rceil$.

Induction Step: Let τ_1 have depth $d+1$. This implies that $\mathit{top}(\tau_1) \neq \emptyset$ and, because $\tau_1 \prec_\tau \tau_2$ or $\tau_1 \preceq_\tau \tau_2$, we have by Definition 4.3.1 and Lemma B.2.7 that $\mathit{top}(\tau_1) = \mathit{top}(\tau_2)$ (more precisely, by point 1 of Definition 4.3.1 we have $\mathit{top}(\tau_1) \subseteq \mathit{top}(\tau_2)$ and by point 2 of Definition 4.3.1 combined with Lemma B.2.7—the latter affirming that $\langle\rangle \notin \tau_1$—we get $\mathit{top}(\tau_1) \supseteq \mathit{top}(\tau_2)$). Let $\mathit{top}(\tau_1) = \{l : m_1, \ldots, l : m_k\}$. 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 $\{\lceil \tau_1 \downarrow \langle l : m_i\rangle \rceil \mid 1 \leq i \leq k\}$

[19]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.

and $\{\lceil \tau_2 \downarrow \langle l : m_i \rangle \rceil \mid 1 \leq i \leq k\}$. We can now proceed by induction. First by Lemma G.1 we have that $\tau_1 \downarrow \langle l : m_i \rangle \preceq_\tau \tau_2 \downarrow \langle l : m_i \rangle$. Furthermore, in case $\tau_1 \prec_\tau \tau_2$, there must be at least one index j such that $\tau_1 \downarrow \langle l : m_j \rangle \prec_\tau \tau_2 \downarrow \langle l : m_j \rangle$, otherwise $\tau_1 \equiv_\tau \tau_2$. For this index j we can apply the first part of the induction hypothesis (because the depth of the respective subtrees is strictly smaller) to show that $\lceil \tau_1 \downarrow \langle l : m_j \rangle \rceil \prec \lceil \tau_2 \downarrow \langle l : m_j \rangle \rceil$. For the other indexes $i \neq j$ we can apply the second part of the induction hypothesis to show that $\lceil \tau_1 \downarrow \langle l : m_i \rangle \rceil \preceq \lceil \tau_2 \downarrow \langle l : m_i \rangle \rceil$. 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 $\lceil \tau_1 \rceil \preceq \lceil \tau_2 \rceil$ if $\tau_1 \preceq_\tau \tau_2$ as well as $\lceil \tau_1 \rceil \prec \lceil \tau_2 \rceil$ if $\tau_1 \prec_\tau \tau_2$.

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

—$\lceil \emptyset \rceil \trianglelefteq^* \lceil \tau \rceil$ if and only if $\tau = \emptyset$ and

—$\lceil \tau \rceil \trianglelefteq^* \lceil \emptyset \rceil$ if and only if $\tau = \emptyset$.

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

The existence of such a mapping also implies, by Lemma C.2, 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 concretizations.)

Also note that the inverse of the mapping $\lceil . \rceil$, introduced in the proof of Proposition 4.4.9, is not total but still strictly monotonic. We do not need this property in the article 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 generalization. In other words, the definition of $\lceil . \rceil$ does not make the notion of an *msg* for characteristic trees, as well as Algorithm 4.3.6, superfluous.