Global Control for Partial Deduction through Characteristic Atoms and Global Trees

Michael Leuschel and Bern Martens

Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium

e-mail: {michael,bern}@cs.kuleuven.ac.be

Tel.: ++32(0)16-327555 ++32(0)16-327560

Fax: ++32(0)16-327996

Abstract

Recently, considerable advances have been made in the (on-line) control of logic program specialisation. A clear conceptual distinction has been established between local and global control and on both levels concrete strategies as well as general frameworks have been proposed. For global control in particular, recent work has developed concrete techniques based on the preservation of characteristic trees (limited, however, by a given, arbitrary depth bound) to obtain a very precise control of polyvariance. On the other hand, the concept of an m-tree has been introduced as a refined way to trace "relationships" of partially deduced atoms, thus serving as the basis for a general framework within which global termination of partial deduction can be ensured in a non ad hoc way.

Blending both, formerly separate, contributions, in this paper, we present an elegant and sophisticated technique to globally control partial deduction of normal logic programs. Leaving unspecified the specific local control one may wish to plug in, we develop a concrete global control strategy combining the use of characteristic atoms and trees with global (m-)trees. We thus obtain partial deduction that always terminates in an elegant, non ad hoc way, while providing excellent specialisation as well as fine-grained (but reasonable) polyvariance.

We conjecture that a similar approach may contribute to improve upon current (on-line) control strategies for functional program transformation methods such as (positive) supercompilation.

Keywords: Partial Deduction, Online Control, Global Control, Characteristic Trees, Termination

Global Control for Partial Deduction through Characteristic Atoms and Global Trees

Michael Leuschel* and Bern Martens**

Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail : {michael,bern}@cs.kuleuven.ac.be

Abstract. Recently, considerable advances have been made in the (on-line) control of logic program specialisation. A clear conceptual distinction has been established between local and global control and on both levels concrete strategies as well as general frameworks have been proposed. For global control in particular, recent work has developed concrete techniques based on the preservation of characteristic trees (limited, however, by a given, arbitrary depth bound) to obtain a very precise control of polyvariance. On the other hand, the concept of an m-tree has been introduced as a refined way to trace "relationships" of partially deduced atoms, thus serving as the basis for a general framework within which global termination of partial deduction can be ensured in a non ad hoc way.
Blending both, formerly separate, contributions, in this paper, we present an elegant and sophisticated technique to globally control partial deduction of normal logic programs. Leaving unspecified the specific local control one may wish to plug in, we develop a concrete global control strategy combining the use of characteristic atoms and trees with global (m-)trees. We thus obtain partial deduction that always terminates in an elegant, non ad hoc way, while providing excellent specialisation as well as fine-grained (but reasonable) polyvariance.
We conjecture that a similar approach may contribute to improve upon current (on-line) control strategies for functional program transformation methods such as (positive) supercompilation.

1 Introduction

A major concern in the specialisation of functional ([4, 13, 30]) as well as logic programs ([21, 15, 8, 27, 5]) 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

* Supported by the Belgian GOA "Non-Standard Applications of Abstract Interpretation"
** Senior Research Assistant of the K.U.Leuven Research Council

programs ([4, 13]) has mainly stressed the former, while supercompilation of functional ([30, 31, 29]) and partial deduction of logic programs ([10, 28, 2, 3, 22, 25, 17]) have concentrated on on-line control. (Some exceptions are [32, 26, 16, 14].) It is within this on-line control tradition that the present work provides a novel and important contribution.

In partial deduction of logic programs, one distinguishes two levels of control ([8, 25]): 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 (leaves of the) resulting trees allow to construct specialised clauses for the given atoms ([21, 1]). At the global level on the other hand, one typically attends to the overall correctness of the resulting program (satisfying the closedness condition in [21]) and strives to achieve the "right" amount of polyvariance, producing sufficiently many (but not too much) specialised versions for each predicate definition in the original program.

Gallagher in [8] writes that providing adequate global control seems much harder than handling the local level. So, in this paper, it is to the latter, global, level that we (again) turn our attention. In recent work, both authors of the present paper have already, separately, investigated issues in global control of partial deduction. [25] and its extended and slightly revised version [24] focus on termination and provide a quite general, refined framework for global control, different instances of which can be taken as the core of practical systems. [17] on the other hand starts from the central role of characteristic trees in preserving specialisation and determining polyvariance (as first proposed in [10]). It shows how partial deduction can be governed through the use of *characteristic atoms*, combining the usual atoms to be partially deduced with the characteristic trees to be enforced upon them. Global termination, however, is guaranteed at the cost of imposing an arbitrary, ad hoc depth bound on characteristic trees (as in all earlier partial deduction based on characteristic trees, see e.g. [10, 7, 18]).

In the present paper, we endeavour to obtain the best of both worlds, blending (a slightly adapted version of) the general framework in [25, 24] with the use of characteristic atoms and trees as in [17]. We thus obtain a very elegant, sophisticated and precise apparatus for on-line global control of partial deduction.

Below, we first recapitulate some necessary background material and provide motivating examples for our approach in Sect. 2. Sect. 3, subsequently, contains the formal elaboration of our method, leading to the presentation of a partial deduction algorithm, parameterised by its local control, the particular choice of which is left open in this paper. Next, in Sect. 4, we return to the examples in Sect. 2, showing how indeed the method developed in Sect. 3 deals properly with them and leads to greatly improved practical specialisation results. Finally, a possible drawback of the presented approach as well as some connections with related work in supercompilation are briefly discussed. We conclude the paper in Sect. 5.

Proofs are assembled in App. A.

2 Ecological Partial Deduction and the Depth Bound Problem

In what follows, we assume the reader to be familiar with the basic concepts of logic programming and partial deduction, as they are presented in e.g. [20, 21]. Throughout, unless stated explicitly otherwise, the terms "(logic) program" and "goal" will refer to a *normal* logic program and goal, respectively.

Given a program P and a goal G, partial deduction produces a new program P' which is P "specialised" to the goal G. The underlying technique is to construct "incomplete" SLDNF-trees for a set of atoms $\mathcal{A}$ to be specialised and extract the program P' from these incomplete search trees by taking resultants (for each atom in $\mathcal{A}$ a different specialised predicate definition will thus be generated). An *incomplete* SLDNF-tree is an SLDNF-tree which, in addition to success and failure leaves, may also contain leaves where no literal has been selected for a further derivation step. Leaves of the latter kind will be called *dangling*. Under the conditions stated in [21], namely closedness and independence, correctness of the specialised program is guaranteed.

In the context of partial deduction, incomplete SLDNF-trees are obtained by applying an unfolding rule, defined as follows:

Definition 1. An *unfolding rule* U is a function which given a program P and a goal G returns a finite (possibly incomplete) SLDNF-tree[3] for $P \cup \{G\}$.

2.1 Ecological Partial Deduction

The problem of *controlling polyvariance* in partial deduction boils down to finding a *terminating* procedure to produce a *finite* set of atoms $\mathcal{A}$ which satisfies the *correctness* conditions of [21] while at the same time being as *precise* as possible (usually the more fine-grained and instantiated the set $\mathcal{A}$ is, the better the potential for specialisation is). Most approaches in the literature so far are based on the syntactic structure of the atoms to be specialised, but it can be shown (see e.g. [18, 17]) that this provides insufficient detail.

[10, 7] therefore introduced the notion of a *characteristic tree*, capturing how atoms are specialised and as such constituting a more refined basis for polyvariance. The following definitions are adapted from [7, 17, 18].

Definition 2. Let G_1 be a goal and let P be a program the clauses of which are numbered. Let $G_1, \dots, G_n$ be a finite, incomplete SLDNF-derivation of $P \cup \{G_1\}$. The *characteristic path* of the derivation is the sequence $(l_1, c_1), \dots, (l_{n-1}, c_{n-1})$, where l_i is the position of the selected literal in G_i, and c_i is defined as:
- if the selected literal is an atom, then c_i is the number of the clause chosen to resolve with G_i.
- if the selected literal is $\neg p(\bar{t})$, then c_i is the predicate p.

The set of all characteristic paths for a given goal G and program P will be denoted by *chpaths*(G, P).

[3] We even allow a trivial SLDNF-tree, i.e. one whose root is a dangling leaf.

Definition 3. Let G be a goal, P a program and U an unfolding rule. Then *the characteristic tree τ of G (in P) via U* is the set of characteristic paths of the non-failing derivations of the incomplete SLDNF-tree obtained by applying U to G (in P). We introduce the notation $chtree(G, P, U) = \tau$. We also say that τ is *a characteristic tree of G (in P)* if it is *the* characteristic tree for some unfolding rule U. Also τ is *a characteristic tree* if it is *a* characteristic tree of some G in some P.

Note that the characteristic path of an empty derivation is the empty path $()$, and the characteristic tree of a trivial SLDNF-tree is $\{()\}$. Characteristic trees are an interesting abstraction of SLDNF-trees because they capture the specialisation performed locally by partial deduction. If two atomic goals have the same characteristic tree then the same branches have been pruned by partial deduction and the atoms have been unfolded in the same way (i.e. to the same depth and the same clauses have been resolved with literals in the same position). Also, in principle a single predicate definition can be used. For more details about the interest and relevance of characteristic trees, we refer to [10, 7, 18, 17].

When using characteristic trees to control polyvariance, the atoms to be specialised are classified according to their characteristic tree. The basic idea is to *have only one specialised version for each characteristic tree.* If several atoms have the same characteristic tree, then they are *abstracted by a single more general atom.* Earlier approaches to partial deduction using characteristic trees [7, 10] have been limited by not being able to *preserve characteristic trees* in that abstraction process (implying that the generalisation has a different local specialisation behaviour). As shown in [17, 18], this can lead to important precision losses as well as non-termination.

The problem has been solved in [17] by simply *imposing* characteristic trees on the generalised atoms. [4] This amounts to associating characteristic trees with the atoms to be specialised, thus allowing the preservation of characteristic trees without having to construct intricate generalisations. The rest of this subsection recapitulates and adapts the necessary material from [17].

From now on, throughout the rest of this paper, we implicitly assume the existence of some given finite underlying language $\mathcal{L}$ in which atoms and terms, goals and programs are expressed.

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

Definition 4. A *P-characteristic atom*, for a given program P, is a couple (A, τ) consisting of an atom A and a characteristic tree τ with $\tau \subseteq chpaths(\leftarrow A, P)$.

Often, when the context allows it, we will drop the P annotation and simply refer to *characteristic* atoms. Also note that τ is not necessarily a characteristic tree of $\leftarrow A$ in P. The following definition associates a set of concretisations with each characteristic atom.

[4] Another solution is presented in [18], which is however limited to definite programs and certain unfolding rules, but enjoys a better overall precision. The core ideas of the present paper can also be used to enhance the [18] method in order to eliminate a similar depth bound problem.

Definition 5. An atom A is a *precise concretisation* of a P-characteristic atom (A', τ') iff A is an instance of A' and for some unfolding rule U we have that $chtree(\leftarrow A, P, U) = \tau'$. An atom B is a *concretisation* of (A', τ') iff it is an instance of a precise concretisation of (A', τ').

A P-characteristic atom can thus be seen as standing for a (possibly infinite) set of atoms, namely the concretisations according to the above definition. A characteristic atom (A, τ) also uniquely determines a set of resultants:

Definition 6. Let (A, τ) be a P-characteristic atom. If $\tau \neq \{()\}$ then $\delta(P, (A, \tau))$ is the set of all (necessarily non-failing) SLDNF-derivations for $P \cup \{\leftarrow A\}$ such that their characteristic paths are in τ. If $\tau = \{()\}$ then $\delta(P, (A, \tau))$ is the set of all non-failing SLD-derivations for $P \cup \{\leftarrow A\}$ of length 1.[5]

Definition 7. Let (A, τ) be a P-characteristic atom. Let $\{\delta_1, \dots, \delta_n\}$ be the SLDNF-derivations in $\delta(P, (A, \tau))$ 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 *body atom (in P)* of (A, τ). We will denote the set of such body atoms by $BA_P(A, \tau)$.

The approach in [17] generates a partial deduction not for a set of atoms but for a set of *characteristic* atoms. As such, the same atom A might occur in several characteristic atoms with entirely different characteristic trees. In order to guarantee correctness of the specialised program, renaming (as well as filtering, see also [9]) is added in [17]. Then, given the following coveredness condition, correctness of the specialised program is established in [17].

Definition 8. Let P be a program and $\mathcal{A}$ a set of characteristic atoms. Then $\mathcal{A}$ is called *P-covered* iff for every characteristic atom in $\mathcal{A}$ each of its body atoms in P is a concretisation of a characteristic atom in $\mathcal{A}$.

2.2 The Depth Bound Problem

When, for the given program, query and unfolding rule, the above sketched method generates a *finite number of different characteristic trees*, its global control regime guarantees termination and correctness of the specialised program as well as "perfect"[6] polyvariance: *For every predicate, exactly one specialised version is produced for each of its different associated*[7] *characteristic trees.* Now, [17], as well as all earlier approaches based on characteristic trees ([10, 7, 18]), achieves the mentioned finiteness condition at the cost of imposing an ad hoc (typically very large) depth bound on characteristic trees. However, for a fairly

[5] The reason behind the special treatment of the case $\tau = \{()\}$ is that at least one unfolding step is needed to avoid the problematic resultant $A \leftarrow A$ in Definition 7.

[6] W.r.t. local precision, see [18, 17].

[7] I.e. a characteristic tree associated to an atom with this predicate symbol.

large class of realistic programs (and unfolding rules), *the number of different characteristic trees generated, is not naturally bounded.* In those cases, the underlying depth bound will have to ensure termination, meanwhile propagating its ugly, ad hoc nature into the resulting specialised program. We illustrate this problem through some examples, setting out with a slightly artificial, but very simple one.

Example 1. The following is the well known reverse with accumulating parameter where a list type check on the accumulator has been added.

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

As can be noticed in Fig. 1, (determinate ([10, 7, 18]) and well-founded ([3, 23, 22]), among others) unfolding produces an infinite number of different characteristic atoms, all with a different characteristic tree. Imposing a depth bound of say 100, we obtain termination, but the algorithm produces 100 different *reverse* versions and the specialised program looks like:

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

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

Situations like the above typically arise when an accumulating parameter influences the computation, because then the growing of the accumulator causes a corresponding growing of the characteristic trees. To be fair, it must be admitted that with most simple programs, this is not the case. For instance, in the standard reverse with accumulating parameter, the accumulator is only copied in the end, but never influences the computation. As illustrated by Example 1 above, this state of affairs will often already be changed when one adds type checking in the style of [11] to even the simplest logic programs.

Among larger and more sophisticated programs, cases like the above become more and more frequent, even in the absence of type checking.[8] For instance,

[8] Especially since efficiently written programs often use accumulating parameters.

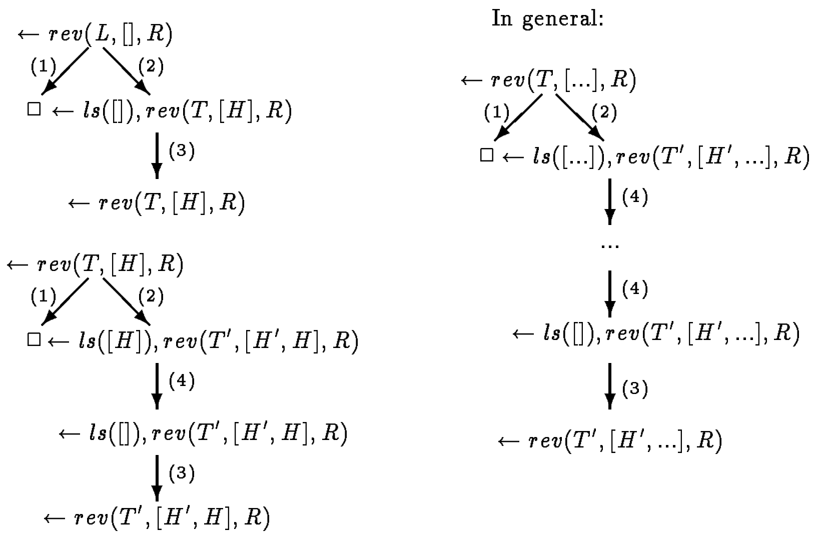

$\leftarrow rev(L, [], R)$
(1)
(2)
$\Box \leftarrow ls([]), rev(T, [H], R)$
(3)
$\leftarrow rev(T, [H], R)$
$\leftarrow rev(T, [H], R)$
(1)
(2)
$\Box \leftarrow ls([H]), rev(T', [H', H], R)$
(4)
$\leftarrow ls([]), rev(T', [H', H], R)$
(3)
$\leftarrow rev(T', [H', H], R)$
In general:
$\leftarrow rev(T, [...], R)$
(1)
(2)
$\Box \leftarrow ls([...]), rev(T', [H', ...], R)$
(4)
...
(4)
$\leftarrow ls([]), rev(T', [H', ...], R)$
(3)
$\leftarrow rev(T', [H', ...], R)$

Fig. 1. SLD-trees for Example 1

in an explicit unification algorithm, one accumulating parameter is the substitution built so far. It heavily influences the computation because new bindings have to be added and checked for compatibility with the current substitution. Another example is the "mixed" meta-interpreter of [12, 19] (sometimes called *InstanceDemo*; part of it is depicted in Fig. 2) for the ground representation in which the goals are "lifted" to the non-ground representation for resolution. To perform the lifting, an accumulating parameter is used to keep track of the variables that have already been encountered. This accumulator influences the computation: Upon encountering a new variable, the program inspects the accumulator.

Example 2. Let $A = l\_mng(Lg, Ln, [sub(N, X)], S)$ and P be the program of Fig. 2 (this situation arose in a real-life experiment). As can be seen in Fig. 3, unfolding A (e.g. using well-founded measures) causes the addition of the atom $l\_mng(Tg, Tn, [sub(N, X), sub(J, Hn)], S)$ at the global (control) level. Notice that the third argument has grown (i.e. we have an accumulator).
So, when in turn unfolding $l\_mng(Tg, Tn, [sub(N, X), sub(J, Hn)], S)$, we will obtain a deeper characteristic tree (because mng traverses the third argument and thus needs one more step to reach the end) which will have as one of its leaves the atom $l\_mng(Tg', Tn', [sub(N, X), sub(J, Hn), sub(J', Hn')], S)$. An infinite sequence of ever growing characteristic trees results and again, as in Example 1, we obtain non-termination without a depth bound, and very unsatisfactory (ad hoc) specialisations with it.

Program:

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

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

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

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

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

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

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

Example query:

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

Fig. 2. Lifting the ground representation

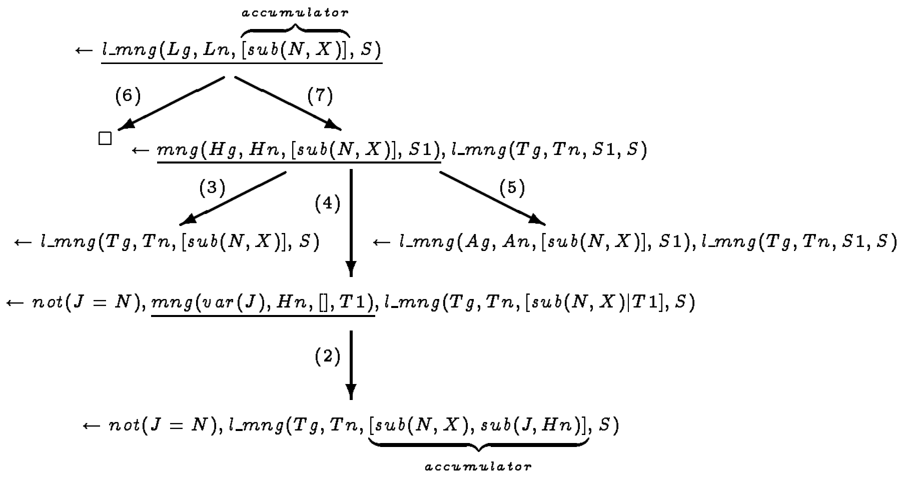

accumulator
$\leftarrow l\_mng(Lg, Ln, [sub(N, X)], S)$
(6)
(7)
□
$\leftarrow mng(Hg, Hn, [sub(N, X)], S1), l\_mng(Tg, Tn, S1, S)$
(3)
(4)
(5)
$\leftarrow l\_mng(Tg, Tn, [sub(N, X)], S)$
$\leftarrow l\_mng(Ag, An, [sub(N, X)], S1), l\_mng(Tg, Tn, S1, S)$
$\leftarrow not(J = N), mng(var(J), Hn, [], T1),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

Fig. 3. Accumulator growth in Example 2

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

3 Partial Deduction using Global Trees

3.1 Introduction

A general framework for global control, not relying on any depth bounds, is proposed in [25, 24]. Marked trees (m-trees) are introduced to register descendency relationships among atoms at the global level. The overall tree is then kept finite through ensuring monotonicity of well-founded measure functions and termination of the algorithm follows, provided the generalisation operation (on atoms) is similarly well-founded. It is to this framework that we now turn for inspiration on how to solve the depth bound problem uncovered in Subsect. 2.2.

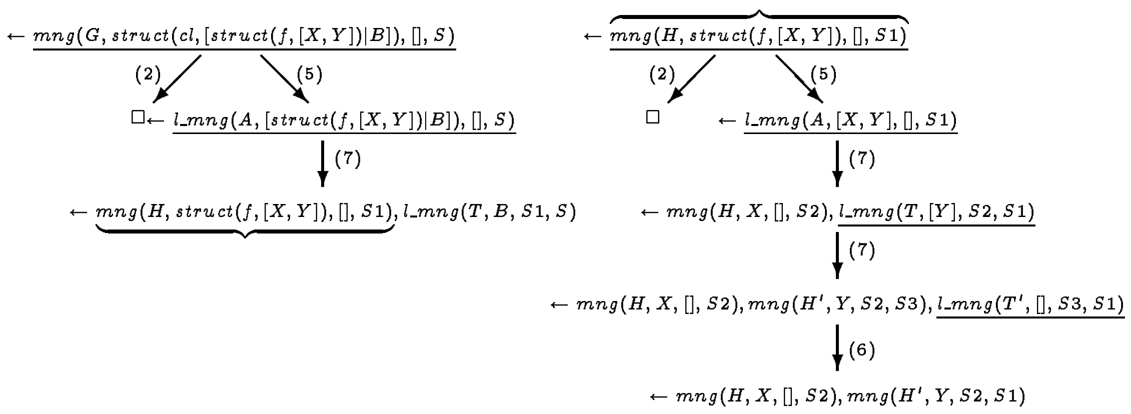
The basic idea will be to watch over the evolution of characteristic trees associated to atoms along the branches of the global tree. Obviously, just measuring the depth of characteristic trees would be far too crude: Global branches would be cut off prematurely and entirely unrelated atoms could be mopped together through generalisation, resulting in completely unacceptable specialisation losses. No, as can be seen in Fig. 1, 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 1), it seems reasonable to stop expanding the global tree, generalise the offending atoms and produce a specialised procedure for the generalisation instead.

However, a closer look at the following variation of Example 2 shows that also this approach would sometimes overgeneralise and consequently fall short of providing sufficiently detailed polyvariance.

Example 3. Reconsider the program in Fig. 2, and suppose that local control uses determinate unfolding. Let us now start partial deduction for the atom $A = mng(G, struct(cl, [struct(f, [X, Y])|B]), [], S)$ (also this situation arose in a real-life experiment). When unfolding A (see Fig. 4), we obtain an SLDNF-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 non-termination would therefore be detected and a generalisation operation executed. However, the atoms in the leaves of the second tree are more general than those already met, and simply continuing partial deduction without generalisation will lead to natural termination without any depth bound intervention.

Example 3 demonstrates that only measuring growth of characteristic trees, even in a refined way, does not always lead to satisfactory specialisation.[9] Luckily, the same example also suggests a solution to this problem: Rather than measuring and comparing characteristic trees, we will *scrutinise entire characteristic*

[9] In fact, whenever the (local) unfolding rule does not unfold "as deeply as possible" (for whatever reason), then a growing characteristic tree might simply be caused by splitting the "maximally deep tree" in such a way that the second part "contains" the first part.



← 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. 4. SLD-trees for Example 3

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 [25, 24], but by entire characteristic atoms.

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

3.2 More on Characteristic Atoms

Generalising Characteristic Atoms. We extend the notions of variants, instances and generalisations, familiar for ordinary atoms,[10] to characteristic trees and atoms:

Definition 9. A characteristic tree τ_1 is *more general* than another characteristic tree τ_2, denoted by $\tau_1 \preceq \tau_2$, iff τ_2 can be obtained by attaching subtrees to the leaves of τ_1. A characteristic atom (A_1, τ_1) is *more general* than another characteristic atom (A_2, τ_2), denoted by $(A_1, \tau_1) \preceq (A_2, \tau_2)$, iff $A_1 \preceq A_2$ and $\tau_1 \preceq \tau_2$. Also (A_1, τ_1) is a *variant* of (A_2, τ_2), denoted by $(A_1, \tau_1) \equiv (A_2, \tau_2)$, iff $(A_1, \tau_1) \preceq (A_2, \tau_2)$ and $(A_2, \tau_2) \preceq (A_1, \tau_1)$. Finally, $CA_1 \prec CA_2$ holds when $CA_1 \preceq CA_2$ but not $CA_1 \equiv CA_2$.

Note that $\{()\}$, a characteristic tree containing a single, empty path, can be extended into a more specific characteristic tree, while the empty characteristic tree $\emptyset = \{\}$ cannot.

Example 4. Given $\tau_1 = \{((1,3))\}, \tau_2 = \{((1,3),(2,4))\}$ and $\tau_3 = \{((1,3)),((1,4))\}$ we have that $\tau_1 \preceq \tau_2$ and $\tau_1 \prec \tau_2$ but not that $\tau_1 \preceq \tau_3$ nor $\tau_2 \preceq \tau_3$.

The following proposition shows that the above definition makes sense wrt the set of ordinary atoms represented by characteristic atoms.

[10] For ordinary atoms, $A_1 \preceq A_2$ will denote that A_1 is more general than A_2.

Proposition 10. *Let CA_1, CA_2 be two characteristic atoms such that $CA_1 \preceq CA_2$. Then every atom A which is a concretisation of CA_2 is also a concretisation of CA_1.*

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

Definition 11. Let τ_1, τ_2 be two P-characteristic trees. Then $msg(\tau_1, \tau_2)$ is the most specific characteristic tree which is more general than both τ_1 and τ_2.
Let $(A_1, \tau_1), (A_2, \tau_2)$ be two characteristic atoms. Then $msg((A_1, \tau_1), (A_2, \tau_2)) = (msg(A_1, A_2), msg(\tau_1, \tau_2))$.

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

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

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

Definition 12. A set $V, \leq_V$ is called *well-quasi-ordered (wqo)* iff for any infinite sequence of elements $e_1, e_2, \ldots$ in V there are $i < j$ such that $e_i \leq_V e_j$. We also say that $\leq_V$ is a *well-quasi order (wqo) on V.*

An interesting wqo is the homeomorphic embedding relation $\trianglelefteq$ of [29] (where it is adapted from [6]).

Definition 13. The *homeomorphic embedding* relation $\trianglelefteq$ on atoms and terms[11] 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 6. We have: $p(a) \trianglelefteq p(f(a))$, $X \trianglelefteq X$, $p(X) \trianglelefteq p(f(Y))$, $p(X,X) \trianglelefteq p(X,Y)$ and $p(X,Y) \trianglelefteq p(X,X)$.

Proposition 14. *The relation $\trianglelefteq$ is a wqo on the set of atoms and terms.*

[11] Expressed in the language $\mathcal{L}$ which we implicitly assume underlying the programs and queries under consideration. Remember that it contains only finitely many constant, function and predicate symbols ! This property is of crucial importance for the proofs in App. A.

The intuition behind Definition 13 is that when some structure re-appears within a larger one, it is homeomorphically embedded by the latter. As is argued in [29], this provides a good starting point for detecting growing structures created by (hence) possibly non-terminating processes.

However, as can be observed in Example 6, the homeomorphic embedding relation $\trianglelefteq$ as defined in Definition 13 is rather crude wrt variables. In fact, all variables are treated as if they were the same variable, a practice which is clearly undesirable in a logic programming context. Intuitively, in the above example, $p(X,Y) \trianglelefteq p(X,X)$ is acceptable, while $p(X,X) \trianglelefteq p(X,Y)$ is not.[12]

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

Definition 15. Let A, B be atoms or terms. Then B *(strictly homeomorphically) embeds* A, written as $A \trianglelefteq^* B$, iff $A \trianglelefteq B$ and A is not a strict instance of B.

Example 7. We now have that $p(X,Y) \trianglelefteq^* p(X,X)$ but not $p(X,X) \trianglelefteq^* p(X,Y)$. Note that still $X \trianglelefteq^* Y$ and $X \trianglelefteq^* X$.

Proposition 16. *The relation $\trianglelefteq^*$ is a wqo on the set of atoms and terms.*

We now extend the embedding relation of Definition 15 to characteristic atoms. One way to obtain a wqo is to first define a term representation of characteristic trees and then use the embedding relation $\trianglelefteq^*$ with this term representation.

Definition 17. By $\lceil . \rceil$ we denote an injective, monotonic[13] mapping from characteristic trees to (ordinary) atoms.

Such a mapping can be easily constructed by representing leaves of the tree by variables. For example we could have $\lceil \{((1,3))\} \rceil = \mathit{select}(1, [\mathit{match}(3, X)])$ and $\lceil \{((1,3),(2,4))\} \rceil = \mathit{select}(1, [\mathit{match}(3, \mathit{select}(2, [\mathit{match}(4, X)]))])$.[14]

Definition 18. 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 $(A_1, \tau_1) \trianglelefteq^* (A_2, \tau_2)$, iff $A_1 \trianglelefteq^* A_2$ and $\lceil \tau_1 \rceil \trianglelefteq^* \lceil \tau_2 \rceil$.

Proposition 19. *Let $\mathcal{A}$ be a set of P-characteristic atoms. Then $\mathcal{A}, \trianglelefteq^*$ is well-quasi-ordered.*

Finally, we consider the relationship between $\trianglelefteq^*$ and $\preceq$ on characteristic atoms.

Proposition 20. *Let CA_1, CA_2, CA_3 be characteristic atoms such that $CA_3 \preceq CA_2$. Then $CA_1 \trianglelefteq^* CA_3 \Rightarrow CA_1 \trianglelefteq^* CA_2$.*

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

[13] I.e. if $\tau_1 \prec \tau_2$ then $\lceil \tau_1 \rceil \prec \lceil \tau_2 \rceil$.

[14] Note that $\{((1,3))\} \prec \{((1,3),(2,4))\}$ and indeed $\lceil \{((1,3))\} \rceil \prec \lceil \{((1,3),(2,4))\} \rceil$.

So, a generalisation of a given characteristic atom will only embed characteristic atoms already embedded by the given one.

Proposition 21. *Let CA_1, CA_2 be characteristic atoms such that $CA_2 \preceq CA_1$. Then $CA_1 \trianglelefteq^* CA_2$ iff $CA_1 \equiv CA_2$.*

Proposition 22. *Let $CA_1, \ldots, CA_n$ be characteristic atoms and assume that $M = msg(CA_1, \ldots, CA_n)$.*[15] *Then the following three statements are equivalent (where $1 \leq i \leq n$):*

1. *$CA_i \trianglelefteq^* M$*
2. *$CA_i \equiv M$*
3. *$CA_i \preceq CA_1, \ldots, CA_i \preceq CA_n$*

Properties 20 and 22 will be used to prove termination of the partial deduction algorithm in Sect. 3.4. Note that Propositions 21 and 22 do not hold for the $\trianglelefteq$ relation (which makes $\trianglelefteq$ less suitable for ensuring termination of partial deduction).

3.3 Global Trees

In this subsection, we adapt and instantiate the m-tree concept presented in [25, 24] according to our particular needs in this paper.

Definition 23. A *global tree* γ_P for a program P is a (finitely branching) tree where nodes can be either *marked* or *unmarked* and each node carries a label which is a P-characteristic atom.

In other words, a node in a global tree γ_P will look as follows: $(n, mark, (A, \tau_A))$, where n is the node identifier, $mark$ an indicator that can take the values u or m designating whether the node is marked or not, and the P-characteristic atom (A, τ_A) is the node's label. Informally, a marked node corresponds to a characteristic atom which has already been treated by the partial deduction algorithm. We will often omit the P subscript when it is either clear from or not relevant in the particular context we are considering.

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

Let γ_P be a global tree. Then we will henceforth denote as L_{γ_P} the set of its labels. And for a given node n in a tree γ, we will refer to its label by l_n.

Definition 24. Let γ be a global tree. Then we define its *associated label mapping* f_γ as the one-to-one mapping $f_\gamma : \gamma, >_\gamma \rightarrow L_\gamma, \trianglelefteq^*$ such that $n \mapsto l_n$. f_γ will be called *non-monotonic* iff $\exists n_1, n_2$ such that $n_1 >_\gamma n_2$ and $l_{n_1} \trianglelefteq^* l_{n_2}$.

Definition 25. We call a global tree γ *well-quasi-ordered* if f_γ is not non-monotonic.

Theorem 26. *A global tree γ is finite if it is well-quasi-ordered.*

[15] As we in fact already did in the pair-wise case, we slightly abuse notation by not writing $msg(\{CA_1, \ldots, CA_n\})$, while that is of course actually what is intended.

3.4 An Algorithm for Partial Deduction

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

We first introduce the following definition:

Definition 27. Let A be an ordinary atom, U an unfolding rule and P a program. Then $chatom(A, P, U) = (A, \tau)$ where $chtree(\leftarrow A, P, U) = \tau$.

A formal description of the algorithm can be found in Fig. 5. Please note that it is parameterised by an unfolding rule U, thus leaving the particulars of local control unspecified. Without loss of generality, we suppose the initial goal to consist of a single atom.

Algorithm 28.

```
Input
  a normal program $P$ and goal $\leftarrow A$
Output
  a set of characteristic atoms $\mathcal{A}$
Initialisation
  $\gamma := \{(1, u, (A, \tau_A))\}$
While $\gamma$ contains an unmarked leaf do
  let $n$ be such an unmarked leaf in $\gamma$: $(n, u, (A_n, \tau_{A_n}))$
  mark $n$
  $\mathcal{B} := \{chatom(B, P, U) | B \in BA_P(A_n, \tau_{A_n})\}$
  For each $(B, \tau_B) \in \mathcal{B}$ do
    If $\mathcal{H} = \{(C, \tau_C) \in Anc_\gamma(n) | (C, \tau_C) \trianglelefteq^* (B, \tau_B)\} = \emptyset$
      Then add $(n_B, u, (B, \tau_B))$ to $\gamma$ as a child of $n$
      Else If $\{(D, \tau_D) \in Anc_\gamma(n) | (D, \tau_D) \preceq (B, \tau_B)\} = \emptyset$
            Then add $(n_B, u, msg(\mathcal{H} \cup \{(B, \tau_B)\}))$ to $\gamma$ as a child of $n$
  Endfor
Endwhile
$\mathcal{A} := L_\gamma$
```

Fig. 5. Partial deduction with global trees.

As in e.g. [8, 25, 17], Algorithm 28 does not output a specialised program, but rather a set of (characteristic) atoms from which the actual code can be generated in a straightforward way. Most of the algorithm is self-explanatory, except perhaps the **For**-loop. In $\mathcal{B}$, all the characteristic atoms are assembled, corresponding to the atoms occurring in the leaves of the SLDNF-tree built (locally, of course) for A_L according to τ_{A_L}. Elements of $\mathcal{B}$ are subsequently inserted into γ as (unmarked) child nodes of L if they do not embed the label of n or any of its ancestor nodes. If one does, and it is an instance of n's label or that of an ancestor of n, then it is simply not added to γ. Finally, if a characteristic atom $(B, \tau_B) \in \mathcal{B}$ does embed an ancestor label, but there is no more general characteristic atom to be found labelling any of the ancestor nodes, then n receives a child node carrying as label the most specific generalisation of (B, τ_B) and all embedded ancestor labels. The latter case is of course the most interesting: Simply adding a node labelled (B, τ_B) would endanger termination. Adding the *msg* label instead secures finiteness, while trying to preserve as much information as seems possible.[16] We obtain the following theorems:

Theorem 29. *Algorithm 28 always terminates.*

Theorem 30. *Let P be a program, input to Algorithm 28, and $\mathcal{A}$ the corresponding set of characteristic atoms produced as output. Then $\mathcal{A}$ is P-covered.*

From Theorem 30, correctness of the specialisation follows as in [17].

Finally, we have developed a post-processing phase reducing the polyvariance to some minimal level without removing any of the specialisation performed by the partial deduction described in Algorithm 28. Due to space limitations a description thereof could not be included.

4 Experimental Results and Discussion

Creating a fully fledged implementation of Algorithm 28 and the above described post-processing is the subject of ongoing work. However, to preliminarily check whether the developed ideas actually lead to improved practical results, we enhanced the system used for ecological partial deduction in [17] with global control through embedding and generalisation on characteristic atoms. We experimented with the result (denoted by *eco−embed*), comparing its performance with ecological partial deduction as in [17]. In the latter case, various depth bounds were imposed on the (local) SLDNF-trees in order to ensure termination (hence the notation *eco−db*__). Finally, in the experiments described below, local control is always based on the embedding relation on the atoms in the proof tree structure (i.e. it checks whether selected literals in covering ancestors, see [3], are embedded), possibly cut off by the imposed depth bound.

We report on three experiments. In the first one, we specialised the *reverse* (with type checking) program of Example 1 for the atom $rev(L, [], R)$. The second and third experiment involved the "lifting" *solve* meta-interpreter ([12, 19]). The program depicted in Fig. 2 is actually an excerpt from its code. Experiment 2 consisted in specialising this program for $solve(["c1", "c2", U], [struct(fa, [X, Y])])$, where $"e"$ denotes the ground representation of an expression e, $c1$ denotes the clause $fa(X, Y) \leftarrow p(X, Y), m(X)$ and $c2$ the clause $mo(X, Y) \leftarrow p(X, Y), f(X)$. Finally, in Experiment 3, we specialised the same *solve* for the atom $solve(["c1", "c2", "p(a, b)\leftarrow", "m(a)\leftarrow"], [struct(fa, [X, Y])])$.

[16] Further enhancing precision through even more cautious generalisation will be a topic of future research.

The first two experiments illustrate the problems encountered in Examples 1 and 2, and have to rely on a depth-bound for termination when using ecological partial deduction as in [17]. The third experiment does not require a depth bound, but illustrates another (well-known) adverse effect of using depth bounds.

The results are summarised in Tables 1, 2 and 3. The experiments were performed using Prolog by BIM on a Sparc Classic running Solaris. The size of the compiled code is expressed in units, 1 unit corresponding to 4.08 bytes on a Sparc Classic. The (compiled code) run times were obtained by using the *time*/2 predicate of Prolog by BIM on an extensive number of queries.

Experiment 1 shows that the additional polyvariance produced by using a depth bound to ensure termination does not pay off in efficiency but increases the code size unnecessarily. Experiment 2 illustrates this even more poignantly. The amount of polyvariance produced by the method *eco*−*db*50 was even so big that we could not complete the partial deduction.[17]

In Experiment 3, it is interesting to note that, even with a depth bound of 50 and for this very simple object program, we do not yet get the optimal result ! So, for Experiment 2, the depth bound of 50 generates way too much polyvariance, while for Experiment 3, the depth bound of 50 is not sufficient to guarantee optimal specialisation of the same program ! Concluding, it seems very likely that the global control improvements proposed in this paper will indeed pay off in partial deduction practice.

Note that the current system (*eco*−*embed*) does not yet structure the characteristic atoms in a global tree, but still just puts them in a set as in [17]. For the above described experiments, this had no influence, but generalisation with non-ancestors may in general severely limit specialisation potential.

A possible drawback of the global control method as laid out in Sect. 3, might be its considerable complexity. Indeed, first, ensuring termination through a well-quasi-ordering is structurally much more costly than the alternative of using a well-founded ordering. The latter only requires comparison with a single "ancestor" object and can be enforced without any search through "ancestor lists" (see [22]). Testing for well-quasi-ordering, however, unavoidably does entail such searching and repeated comparisons with several ancestors. Moreover, in our particular case, checking $\trianglelefteq^*$ on characteristic atoms is in itself a quite costly operation, adding considerably to the innate complexity of maintaining a well-quasi-ordering. It remains therefore to be seen whether a global control such as the one above can be used (or approximated in an efficient way) in circumstances where speed (or complexity) of the transformation is an important factor.

We conclude this section with a brief discussion on the relation between our global control and what may be termed thus in supercompilation ([30, 31, 29]). We already pointed out that the inspiration for using $\trianglelefteq$ derives from [29]. In that paper, a generalisation strategy for positive supercompilation (no negative information propagation while driving) is proposed. It uses $\trianglelefteq$ to compare nodes in a marked partial process tree (a notion roughly corresponding to marked or

[17] After running for over several hours, producing 134 different predicates and generalising 80 times, *eco*−*db*50 overflowed the heap limit of 4.5 Megabytes.

Method	Run Time	#Clauses/#Predicates	Compiled Code Size
original	6.7 s	4/2	203 u
eco-embed	6.7 s	6/3	331 u
eco-db10	6.7 s	23/12	1 955 u
eco-db50	7.1 s	103/52	29 573 u
eco-db100	7.5 s	203/102	122 391 u

Table 1. Experiment 1, *rev* program

Method	Run Time	#Clauses/#Predicates	Compiled Code Size
original	3.07 s	16/8	1 328 u
eco-embed	1.42 s	138/26	26 390 u
eco-db10	1.84 s	328/68	65 863 u
eco-db50	- s	-/ $\geq$134	- u

Table 2. Experiment 2, *solve* program

Method	Run Time	#Clauses/#Predicates	Compiled Code Size
original	2.44 s	16/8	1 328 u
eco-embed	0.04 s	1/1	77 u
eco-db10	1.63 s	34/19	4 164 u
eco-db50	0.06 s	8/4	1 062 u

Table 3. Experiment 3, *solve* program for another query

global trees in partial deduction). These nodes, however, only contain syntactical information. It is our current understanding that both the addition of something similar to characteristic trees and the use of the refined $\trianglelefteq^*$ embedding can lead to improvements of the method proposed in [29]. Finally, we return to an observation made in [24]: Neighbourhoods of order "n" in (full) supercompilation ([31]), are essentially the same as classes of atoms (or goals) with an identical depth n characteristic tree. Adapting our technique for supercompilation would therefore probably allow to remove the depth bound on neighbourhoods.

5 Conclusion

In this paper, we have developed a sophisticated on-line global control technique for partial deduction of normal logic programs. Importing and adapting m-trees from [25, 24], we have overcome the need for a depth bound on characteristic trees to guarantee termination of partial deduction as proposed in [17]. Plugging in a depth bound free local control strategy (see e.g. [3, 22]), we thus obtain a fully automatic, concrete partial deduction method that always terminates and produces precise and reasonable polyvariance, without resorting to any ad hoc techniques. To the best of our knowledge, this is the very first such method.

Along the way, we have defined generalisation and embedding on characteristic atoms, refining the homeomorphic embedding relation $\trianglelefteq$ from [29] into $\trianglelefteq^*$, and showing that the latter is more suitable in a logic programming setting. Initial experiments with a partial implementation of the method showed its great practical value; A fully fledged implementation and further experimentation are the subjects of ongoing work.

Acknowledgements

We thank the following persons for interesting comments and/or stimulating discussions on this work: Eddy Bevers, Danny De Schreye, André de Waal, Robert Glück, Neil Jones, Jesper Jørgensen, Torben Mogensen, Kristian Nielsen, David Sands and Morten Heine Sørensen. André de Waal and Jesper Jørgensen also gave useful feedback on a draft version of this paper. Alain Callebaut provided helpful information on benchmarking Prolog by BIM programs. Finally, we gratefully acknowledge Danny De Schreye's (almost) never failing enthusiasm and support.

References

1. K. Benkerimi and J. W. Lloyd. A partial evaluation procedure for logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings NACLP'90*, pages 343–358, Austin, Texas, October 1990. MIT Press.
2. R. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.
3. M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing*, 11(1):47–79, 1992.
4. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings POPL'93*, Charleston, South Carolina, January 1993. ACM.
5. D. De Schreye, M. Leuschel, and B. Martens. Tutorial on program specialisation (abstract). In J.W. Lloyd, editor, *Proceedings ILPS'95*, pages 615–616, Portland, Oregon, December 1995. MIT Press.
6. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, MIT Press, 1990.
7. J. Gallagher. A system for specialising logic programs. Technical Report TR-91-32, Computer Science Department, University of Bristol, U.K., November 1991.
8. J. Gallagher. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.
9. J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings Meta'90*, pages 229–244, Leuven, April 1990.
10. J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(3&4):305–333, 1991.

11. J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, *Proceedings LOPSTR'92*, pages 151–167. Springer-Verlag, Workshops in Computing Series, 1993.
12. P. M. Hill and J. Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, U.K., August 1994. To appear in Volume V of the Handbook of Logic in Artificial Intelligence and Logic Programming, Oxford University Press.
13. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
14. J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. Technical Report CW221, Departement Computerwetenschappen, K.U.Leuven, Belgium, December 1995. Abridged version submitted for publication.
15. J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, pages 49–69. Springer-Verlag, LNCS 649, 1992.
16. M. Leuschel. Partial evaluation of the "real thing". In F. Turini, editor, *Proceedings LOPSTR'94 and META'94*, pages 122–137. Springer-Verlag, Workshops in Computing Series, 1995.
17. M. Leuschel. Ecological partial deduction: Preserving characteristic trees without constraints. In M. Proietti, editor, *Proceedings of LOPSTR'95*. Springer-Verlag, 1996. To appear. Extended version as Technical Report CW 216, K.U.Leuven, October 1995. Accessible via **http://www.cs.kuleuven.ac.be/~lpai**.
18. M. Leuschel and D. De Schreye. An almost perfect abstraction operation for partial deduction using characteristic trees. Technical Report CW215, Departement Computerwetenschappen, K.U.Leuven, Belgium, October 1995. Submitted for Publication. Accessible via **http://www.cs.kuleuven.ac.be/~lpai**.
19. M. Leuschel and D. De Schreye. Towards creating specialised integrity checks through partial evaluation of meta-interpreters. In *Proceedings of PEPM'95, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 253–263, La Jolla, California, June 1995. ACM Press.
20. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
21. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3&4):217–242, 1991.
22. B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 1996. To Appear, abridged and revised version of Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, October 1993, accessible via **http://www.cs.kuleuven.ac.be/~lpai**.
23. B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122(1-2):97–117, 1994.
24. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. Technical Report CSTR-94-16, Computer Science Department, University of Bristol, U.K., December 1994. Accessible via **http://www.cs.kuleuven.ac.be/~lpai**.
25. B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. In L. Sterling, editor, *Proceedings ICLP'95*, pages 597–611, Shonan Village Center, Kanagawa, Japan, June 1995. MIT Press.
26. T. Mogensen and A. Bondorf. Logimix: A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *Proceedings LOPSTR'92*, pages 214–227. Springer-Verlag, Workshops in Computing Series, 1993.

27. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19/20:261–320, 1994.
28. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12(1):7–51, 1993.
29. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Proceedings ILPS'95*, Portland, Oregon, December 1995. MIT Press. To Appear.
30. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
31. V. F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.
32. D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Proceedings of the 3rd ACM Conference on Functional Programming Languages and Computer Architecture*, pages 165–191, Cambridge, Massachusets, USA, August 1991. Springer-Verlag, LNCS 523.

A Appendix: Proofs

A.1 Proposition 10

Proposition10. *Let CA_1, CA_2 be two characteristic atoms such that $CA_1 \preceq CA_2$. Then every atom A which is a concretisation of CA_2 is also a concretisation of CA_1.*

Proof. Let $CA_1 = (A_1, \tau_1)$ and $CA_2 = (A_2, \tau_2)$. Let A be a concretisation of CA_2. Then A must be an instance of some precise concretisation B of CA_2, i.e. $chtree(\leftarrow B, P, U) = \tau_2$ for some U. We know by definition 9 that τ_2 can be obtained from τ_1 by extending leaves. This means that we can find an unfolding rule U' such that $chtree(\leftarrow B, P, U') = \tau_1$. Hence A is also a concretisation of CA_1. □

A.2 Proposition 14

Proposition14. *The relation $\trianglelefteq$ is a wqo on the set of atoms and terms.*

Proof. (Proofs similar to this one, are standard in the literature. We include it for completeness.) We define the relation $\preceq$ on the set $\mathcal{S} = \mathcal{V} \cup \mathcal{F}$ of symbols, containing the set of variables $\mathcal{V}$ and the set of functors and predicates $\mathcal{F}$, as the least relation satisfying:

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

This relation is a wqo on $\mathcal{S}$ and hence by Kruskal's theorem (see [6]) its embedding extension to terms, $\preceq_{emb}$, which is identical to $\trianglelefteq$, is a wqo on the set of atoms and terms. □

A.3 Proposition 16

The following definitions and lemmas are needed to prove Proposition 16.

Definition31. Let *Term* and *Atom* denote the sets of terms and atoms, respectively, in a language $\mathcal{L}$. We define the function $s : Term \cup Atom \to I\!N$, counting the number of symbols in a term or an atom, by:

If $t = f(t_1, \ldots, t_n), n > 0$
then $s(t) = 1 + s(t_1) + \cdots + s(t_n)$
else $s(t) = 1$

Let the number of distinct variables in a term or atom t be $v(t)$. Define the function $h : Term \cup Atom \to I\!N, h(t) = s(t) - v(t)$.

This definition is taken from [9, 24]. Note that $h(t) > 0$ for any non-variable t. The following lemma is proven in [9].

Lemma 32. *If A and B are atoms or terms such that $B \prec A$, then $h(A) > h(B)$.*

It follows that, for every atom or term A, there are no infinite chains of strictly more general atoms or terms. In order to prove Proposition 16 we however need an even stronger property, namely that there are only a finite number of atoms (up to variable renaming) more general than any atom or term A. At this point, we repeat that we pre-suppose a finite underlying language (i.e. it contains only a finite number of constant, function and predicate symbols — the number of variables might still be infinite). This assumption is crucial in the remainder of this appendix.

Lemma 33. *For every natural number $n \geq 0$ there are only a finite number of distinct (up to variable renaming) atoms A such that $h(A) = n$.*

Proof. (Sketch) We define the function $c : \mathit{Term} \cup \mathit{Atom} \to \mathbb{N}$, counting the number of function and constant symbols in a term or an atom, by:

If $t = f(t_1, \ldots, t_n), n \geq 0$
then $c(t) = 1 + c(t_1) + \cdots + c(t_n)$
else $c(t) = 0$

Trivially, we have that $\forall t : s(t) \geq h(t) \geq c(t)$. Also, because the underlying language contains only a finite number of predicate, function and constant symbols, we have that for every $n \geq 0$ there are only a finite number of atoms or terms t (up to variable renaming) such that $c(t) = n$ (first prove this by induction for atoms and terms without repeated variables — the full result then follows because there is only a finite number of ways to substitute these variables for each other). This implies that for every $n \geq 0$ there are only a finite number of atoms or terms t (up to variable renaming) such that $c(t) \leq n$. Finally, we can conclude that the set of atoms and terms such that $h(t) = n$ is finite (up to variable renaming) since this set is contained in the finite set $\{t \mid c(t) \leq n\}$ (because $n = h(t) \geq c(t)$). □

Lemma 34. *For each atom A there are only a finite number of distinct (up to variable renaming) atoms more general than A.*

Proof. The result follows from Lemmas 32 and 33. □

Definition 35. Let σ be a sequence of atoms (or terms). Then σ^* is defined recursively as follows:

- $\epsilon^* = \epsilon$
- $(A.\rho)^* = A.(\rho_A)^*$ where ρ_A is the subsequence of ρ containing all atoms which are not strictly more general than A.

Lemma 36. *Let σ be a sequence of atoms (or terms). If σ is infinite and does not contain an infinite number of variants then σ^* is also infinite.*

Proof. Because by Lemma 34, for each atom A there is only a finite number of atoms M (up to variable renaming) more general than A, and because for each such M we only have a finite number of variants in σ, we can deduce that at each application of step 2 of Definition 35, only a finite number of atoms are removed. This means that we can indefinitely keep on applying step 2 and thus σ^* will be an infinite sequence. □

We can now actually prove Proposition 16.

Proposition16. *The relation $\trianglelefteq^*$ is a wqo on the set of atoms and terms.*

Proof. Suppose that we have an infinite sequence σ. If σ contains an infinite number of variants, then we take the subsequence containing these variants. For this subsequence, $\trianglelefteq$ is identical to $\trianglelefteq^*$, and therefore we can find $i < j$ such that $\sigma_i \trianglelefteq^* \sigma_j$ (actually by definition of $\trianglelefteq^*$ and $\trianglelefteq$, this will hold for any $i < j$ in the subsequence). Otherwise, we consider the subsequence σ^* which by the above Lemma 36 is still infinite. For this subsequence, $\trianglelefteq$ is again identical to $\trianglelefteq^*$, and therefore we can find $i < j$ such that $\sigma_i \trianglelefteq^* \sigma_j$. □

A.4 Proposition 19

Proposition19. *Let $\mathcal{A}$ be a set of P-characteristic atoms. Then $\mathcal{A}, \trianglelefteq^*$ is well-quasi-ordered.*

Proof. Let us take as our vocabulary $\mathcal{F}$ the set of terms and atoms. By proposition 16, $\trianglelefteq^*$ is a wqo on $\mathcal{F}$. Hence by Kruskal's theorem (see e.g. [6]), its embedding extension to terms constructed from $\mathcal{F}$ is also a wqo. We restrict ourselves to one binary functor $c/2$ and to terms $c(A, T)$ constructed by using this functor exactly once such that A is an atom and T the representation of some characteristic tree. For this special case, this embedding coincides with Definition 18 and hence $\mathcal{A}, \trianglelefteq^*$ is well-quasi-ordered. □

A.5 Propositions 20, 21 and 22

Proposition20. *Let CA_1, CA_2, CA_3 be characteristic atoms such that $CA_3 \preceq CA_2$. Then $CA_1 \trianglelefteq^* CA_3 \Rightarrow CA_1 \trianglelefteq^* CA_2$.*

Proof. We first prove the property for arbitrary terms. From that, the result follows immediately for the (ordinary) atom component.
So, let t_1, t_2, t_3 be terms (in the underlying language) such that $t_3 \preceq t_2$ and $t_1 \trianglelefteq^* t_3$. We have to argue that $t_1 \trianglelefteq^* t_2$. First, clearly, t_1 cannot be a strict instance of t_2. The proof that also $t_1 \trianglelefteq t_2$ is through induction on the structure of t_3. If t_3 is a variable, then t_1 and t_2 also are, and the result follows. Finally, if $t_3 = f(t_{31}, \ldots, t_{3n})$, then also $t_2 = f(t_{21}, \ldots, t_{2n})$ and $t_{31} \preceq t_{21}, \ldots, t_{3n} \preceq t_{2n}$. Irrespective of whether $t_1 = f(t_{11}, \ldots, t_{1n})$ or is some other term, we obtain the desired result through the induction hypothesis.
Next, by monotonicity of $\lceil . \rceil$ $\tau_3 \preceq \tau_2$ implies that $\lceil \tau_3 \rceil \preceq \lceil \tau_2 \rceil$. Hence the property for the characteristic tree component follows from the result for arbitrary terms. Finally, as $(A_3, \tau_3) \preceq (A_2, \tau_2)$ implies by definition both $A_3 \preceq A_2$ and $\tau_3 \preceq \tau_2$, we have proven the result for characteristic atoms. □

Proposition21. *Let CA_1, CA_2 be characteristic atoms such that $CA_2 \preceq CA_1$. Then $CA_1 \trianglelefteq^* CA_2$ iff $CA_1 \equiv CA_2$.*

Proof. Let $CA_1 = (A_1, \tau_1)$ and $CA_2 = (A_2, \tau_2)$.
($\Rightarrow$): $CA_1 \trianglelefteq^* CA_2$ implies that $A_1 \trianglelefteq A_2$ and that A_1 is not a strict instance of A_2. Hence A_1 must be a non strict instance, i.e. a variant of A_2. For the characteristic tree part we know that $\tau_1 \trianglelefteq \tau_2$ and that $\lceil A_1 \rceil$ is not a strict instance of $\lceil A_2 \rceil$. By monotonicity of $\lceil . \rceil$ we know that τ_1 must be identical to τ_2.
($\Leftarrow$): Trivial because CA_2 is not strictly more general than CA_1 and because two variants V_1, V_2 always homeomorphically embed each other (i.e. $V_1 \trianglelefteq V_2 \wedge V_2 \trianglelefteq V_1$).
□

Proposition22. *Let $CA_1, \ldots, CA_n$ be characteristic atoms and assume that $M = msg(CA_1, \ldots, CA_n)$. Then the following three statements are equivalent (where $1 \leq i \leq n$):*

1. $CA_i \trianglelefteq^* M$
2. $CA_i \equiv M$
3. $CA_i \preceq CA_1, \ldots, CA_i \preceq CA_n$

Proof. First observe that, by definition, $M \preceq CA_i$. We then obtain:

$1 \Rightarrow 2$ Immediate from the only-if part in Proposition 21.
$2 \Rightarrow 3$ Trivial by definition of msg.
$3 \Rightarrow 1$ Statement 3 implies that $CA_i \preceq msg(CA_1, \ldots, CA_{i-1}, CA_{i+1}, \ldots, CA_n)$. It follows that $CA_i \equiv M$ and we obtain statement 1 through the if part of Proposition 21.

□

A.6 Theorem 26

Theorem26. *A global tree γ is finite if it is well-quasi-ordered.*

Proof. Assume that γ is not finite. Then it contains (König's Lemma) at least one infinite branch $n_1 >_\gamma n_2 >_\gamma \ldots$. Consider the corresponding infinite sequence of elements $l_{n_1}, l_{n_2}, \ldots \in L_\gamma, \trianglelefteq^*$. From Proposition 19, we know that $L_\gamma, \trianglelefteq^*$ is wqo and therefore, there must exist $l_{n_i}, l_{n_j}, i < j$ in the above mentioned sequence such that $l_{n_i} \trianglelefteq^* l_{n_j}$. But this implies that f_γ is non-monotonic. □

A.7 Theorem 29

Theorem29. *Algorithm 28 always terminates.*

Proof. Upon each iteration of the while-loop in Algorithm 28, exactly one node in γ is marked, and zero or more (unmarked) nodes are added to γ. Moreover, Algorithm 28 never deletes a node from γ, neither does it ever "unmark" a marked node. Hence, since all branchings are finite, non-termination of the algorithm must result in the construction of an infinite branch. It is therefore sufficient to argue that after every iteration of the while-loop, γ is a wqo global tree.

First, this holds after initialisation. Also, obviously, a global tree will be converted into a new global tree through the while-loop. Now, a while-iteration adds zero or more, but finitely many, child nodes to a particular leaf n in the tree, thus creating a (finite) number of new branches that are extensions of the old branch leading to n. We prove that on none of the new branches, f_γ is non-monotonic. The branch extensions are actually constructed in the for-loop (which terminates trivially), (at most) one for every element of $\mathcal{B}$. So, let us take an arbitrary characteristic atom $(B, \tau_B) \in \mathcal{B}$, then there are three cases to consider:

- Either (B, τ_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 not be non-monotonic on the newly created branch.
- Or some such label is embedded, but there is also a more (or equally) general label already in some node on the branch to n. Then, no corresponding leaf is inserted in the tree, and there is nothing left to prove.
- Or, finally, some labels on the branch are embedded, but none are more general. The fresh leaf will then be labelled by the *msg* M of (B, τ_B) and all the labels on the branch it embeds. In that case, Propositions 20 and 22 together imply that M embeds none of the labels on the branch to n.

$\square$

A.8 Theorem 30

Theorem30. *Let P be a program, input to Algorithm 28, and $\mathcal{A}$ the corresponding set of characteristic atoms produced as output. Then $\mathcal{A}$ is P-covered.*

Proof. First, it is straightforward to prove that throughout the execution of Algorithm 28, any unmarked node in γ must be a leaf. It therefore suffices to show that after each while-loop iteration, only unmarked leaves in γ possibly carry a non-covered label.[18] Trivially, this property holds after initialisation. Now, in the while-loop, one unmarked leaf n is selected and marked. The for-loop then precisely proceeds to incorporate (unmarked leaf) nodes into γ such that all body atoms of n's label are concretisations of at least one label in the new, extended γ. $\square$

This article was processed using the LaTeX macro package with LLNCS style

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