Constrained Partial Deduction and the Preservation of Characteristic Trees

Michael Leuschel and Danny De Schreye

Report CW 250, June 1997

Department of Computing Science, K.U.Leuven

Abstract

Partial deduction strategies for logic programs often use an abstraction operator to guarantee the finiteness of the set of goals for which partial deductions are produced. Finding an abstraction operator which guarantees finiteness and does not lose relevant information is a difficult problem. In earlier work Gallagher and Bruynooghe proposed to base the abstraction operator on *characteristic paths* and *trees*, which capture the structure of the generated incomplete SLDNF-tree for a given goal.

In this paper we exhibit the advantages of characteristic trees over purely syntactical measures: if characteristic trees can be preserved upon generalisation, then we obtain an almost perfect abstraction operator, providing just enough polyvariance to avoid any loss of local specialisation. Unfortunately, the abstraction operators proposed in earlier work do not always preserve the characteristic trees upon generalisation. We show that this can lead to important specialisation losses as well as to non-termination of the partial deduction algorithm. Furthermore, this problem cannot be adequately solved in the ordinary partial deduction setting.

We therefore extend the expressivity and precision of the Lloyd and Shepherdson partial deduction framework by integrating constraints. We provide formal correctness results for the so obtained generic framework of *constrained partial deduction*. Within this new framework we are, among others, able to overcome the above mentioned problems by introducing an alternative abstraction operator, based on so called *pruning constraints*. We thus present a terminating partial deduction strategy which, for purely determinate unfolding rules, induces no loss of local specialisation due to the abstraction while ensuring correctness of the specialised programs.

This is a revised version of CW 199 and CW 215.

Keywords : Logic Programming, Program Specialisation, Partial Deduction, Constraints.

CR Subject Classification : I.2.2, D.1.2, I.2.3, F.4.1, D.1.6.

Constrained Partial Deduction and the Preservation of Characteristic Trees

Michael Leuschel and Danny De Schreye
K.U. Leuven, Department of Computer Science
Celestijnenlaan 200 A, B-3001 Heverlee, Belgium
e-mail: {michael,dannyd}@cs.kuleuven.ac.be

June 16, 1997

Abstract

Partial deduction strategies for logic programs often use an abstraction operator to guarantee the finiteness of the set of goals for which partial deductions are produced. Finding an abstraction operator which guarantees finiteness and does not lose relevant information is a difficult problem. In earlier work Gallagher and Bruynooghe proposed to base the abstraction operator on *characteristic paths* and *trees*, which capture the structure of the generated incomplete SLDNF-tree for a given goal.

In this paper we exhibit the advantages of characteristic trees over purely syntactical measures: if characteristic trees can be preserved upon generalisation, then we obtain an almost perfect abstraction operator, providing just enough polyvariance to avoid any loss of local specialisation. Unfortunately, the abstraction operators proposed in earlier work do not always preserve the characteristic trees upon generalisation. We show that this can lead to important specialisation losses as well as to non-termination of the partial deduction algorithm. Furthermore, this problem cannot be adequately solved in the ordinary partial deduction setting.

We therefore extend the expressivity and precision of the Lloyd and Shepherdson partial deduction framework by integrating constraints. We provide formal correctness results for the so obtained generic framework of *constrained partial deduction*. Within this new framework we are, among others, able to overcome the above mentioned problems by introducing an alternative abstraction operator, based on so called *pruning constraints*. We thus present a terminating partial deduction strategy which, for purely determinate unfolding rules, induces no loss of local specialisation due to the abstraction while ensuring correctness of the specialised programs.

Keywords: Logic Programming, Program Specialisation, Partial Deduction, Constraints

1 Introduction

Partial evaluation has received considerable attention in logic programming [18, 29, 53] and functional programming (see e.g. [25] and references therein). In [28] Komorowski introduced the topic in the logic programming setting and later, for pure logic programs, first refers to it as *partial deduction*. Another milestone is [43], where firm theoretical foundations for partial deduction are established. It introduces the notions of *independence* and *closedness*, which are properties of the set of atoms for which the partial deduction is performed. Under these conditions, soundness and completeness of the transformed program are guaranteed.

In the light of these conditions, a key problem in partial deduction is: given a set of atoms of interest, $\mathbf{A}$, provide a terminating procedure that computes a new set of atoms, $\mathbf{A}'$, and a partial deduction for the atoms in $\mathbf{A}'$, such that:

- every atom in $\mathbf{A}$ is an instance of an atom in $\mathbf{A}'$, and
- the closedness and independence conditions are satisfied.

Moving from the initial set $\mathbf{A}$ to the new set $\mathbf{A}'$ requires an abstraction operator. In addition to the conditions stated above, this abstraction operator should preserve as much of the specialisation that was (in principle) possible for the atoms in $\mathbf{A}$.

An approach which tries to achieve all these goals in an elegant and refined way is that of Gallagher and Bruynooghe [20, 17]. Its abstraction operator is based on the notions of *characteristic path*, *characteristic tree* and *most specific generalisation*. Intuitively, two atoms of $\mathbf{A}$ are replaced by their most specific generalisation in $\mathbf{A}'$, if their (incomplete) SLDNF-trees under the given unfolding rule have an identical structure (this structure is referred to as the characteristic tree). So, the main idea is, instead of using the *syntactic* structure of the atoms in $\mathbf{A}$, the abstraction operator examines their specialisation *behaviour*. Furthermore, if the characteristic trees are preserved by the generalisation then a lot of the specialisation that was possible within $\mathbf{A}$ will still be possible within $\mathbf{A}'$.

Unfortunately, although the approach is conceptually appealing, several errors turn up in the arguments provided in [20] and [17]. In the current paper we show that these errors can lead to relevant precision losses and even to non-termination of the partial deduction process. We will also show that these problems *cannot* be solved within the standard partial deduction approach based on [43]. We therefore extend the standard partial deduction framework by integrating ideas from constraint logic programming (CLP) so as to be able to place constraints on the atoms in $\mathbf{A}$. Within this new generic framework of *constrained partial deduction* we will be able to (significantly) adapt the approaches of [20, 17] to overcome the above mentioned problems. This is achieved by introducing an alternative abstraction operator, which is based on so called *pruning constraints* expressed using Clark's equality theory (CET). For definite programs and purely determinate unfolding rules[1], this adapted approach allows to solve all problems with the original formulations in [20] and [17], thus ensuring the claimed termination and precision properties.

The paper is structured as follows. In Section 2 we introduce partial deduction from a theoretical viewpoint, expose some of the practical difficulties, introduce the concepts of local and global precision and define the "control of polyvariance" problem. We also outline an algorithm for partial deduction and show the interest of using an abstraction operator. In Section 3 we introduce the concepts of characteristic paths and trees and exhibit their significance for partial deduction. This is the first time that, to our knowledge, the interest and motivations of characteristic trees (or neighbourhoods in supercompilation of functional programs [67, 68] for that matter) are made explicit. We also make a first attempt at defining a proper abstraction operator and show its (substantial) difficulties. We also illustrate the problem with the approaches in [20] and [17]. In Section 4 we introduce the framework for constrained partial deduction along with a fundamental correctness result. In Section 5 we present a particular instance of the framework, based on Clark's equality theory, along with an algorithm and an associated abstraction operator. We show that, for definite programs and certain unfolding rules, this approach ensures termination while providing a very precise and fine grained control of polyvariance. In Section 6 we present some results of an implementation

[1] This same limitation is also present in [20]. We will, however, show how this limitation can be lifted in a rather straightforward manner.

of this approach. In the discussion in Section 7 we point out several ways to extend the method to normal programs and more powerful unfolding rules. We also discuss related work and other potential applications of the constrained partial deduction framework of Section 4. The conclusion can be found in Section 8.

2 Preliminaries and Motivations

Throughout this paper, we suppose familiarity with basic notions in logic programming (see e.g. [1, 42]). Notational conventions are standard and self-evident. In particular, in programs, we denote variables through strings starting with (or usually just consisting of) an upper-case symbol, while the names of constants, functions and predicates begin with a lower-case character.

As common in partial deduction, the notion of SLDNF-trees is extended to also allow *incomplete* SLDNF-trees which, in addition to success and failure leaves, may also contain leaves where no literal has been selected for a further derivation step. Leaves of the latter kind will be called *dangling* [49]. Also, a *trivial* SLDNF-tree is one whose root is a dangling leaf.

2.1 Partial Deduction

Given a logic program P and a goal G, *partial deduction* produces a new program P' which is P "specialised" to the goal G; the aim being that the specialised program P' is more efficient than the original program P for all goals which are instances of G.

The technique of 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 which have been performed beforehand by the *partial deducer* and the clauses of the specialised program are then extracted from these trees by constructing one specialised clause per branch. The incomplete SLDNF-trees are obtained by applying an unfolding rule, defined as follows:

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

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

Definition 2.2 *($resultants(\tau)$) Let P be a normal program and A an atom. Let τ be a finite SLDNF-tree for $P \cup \{\leftarrow A\}$. Let $\leftarrow G_1, \ldots, \leftarrow G_n$ be the goals in the (non-root) leaves of the non-failed 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 $\{A\theta_1 \leftarrow G_1, \ldots, A\theta_n \leftarrow G_n\}$.*

As the goal in the root of τ is atomic, the resultants $resultants(\tau)$ are all clauses. We can thus formalise partial deduction in the following way.

Definition 2.3 (partial deduction) *Let P be a normal program and A an atom. Let τ be a finite, non-trivial SLDNF-tree for $P \cup \{\leftarrow A\}$. Then the set of clauses $resultants(\tau)$ is called a* partial deduction of A in P.

If $\mathbf{A}$ *is a finite set of atoms, then a* partial deduction of $\mathbf{A}$ in P *is the union of one partial deduction for each element of* $\mathbf{A}$. *A* partial deduction of P wrt $\mathbf{A}$ *is a normal program obtained from* P *by replacing the set of clauses in* P*, whose head contains one of the predicate symbols appearing in* $\mathbf{A}$ *(called the partially deduced predicates), with a partial deduction of* $\mathbf{A}$ *in* P.

Note that if τ is a trivial SLDNF-tree for $P \cup \{\leftarrow A\}$ then $resultants(\tau)$ consists of the problematic clause $A \leftarrow A$ and the specialised program of Definition 2.3 would contain a loop. That is why trivial trees are not allowed in Definitions 2.1 and 2.3. This is, however, not a sufficient condition for correctness of the specialised programs. In [43], Lloyd and Shepherdson presented a fundamental correctness theorem for partial deduction. The two (additional) basic requirements for correctness of a partial deduction of P wrt $\mathbf{A}$ are the *independence* and *closedness* conditions. The independence condition guarantees that the specialised program does not produce additional answers and the closedness condition guarantees that all calls, which might occur during the execution of the specialised program, are covered by some definition. The following summarises the correctness result of [43]:

Definition 2.4 (A-closed, independence) *Let* S *be a set of first order formulas and* $\mathbf{A}$ *a finite set of atoms. Then* S *is* $\mathbf{A}$-closed *iff each atom in* S *containing a predicate symbol occurring in an atom in* $\mathbf{A}$ *is an instance of an atom in* $\mathbf{A}$. *Furthermore we say that* $\mathbf{A}$ *is* independent *iff no pair of atoms in* $\mathbf{A}$ *have a common instance.*

Theorem 2.5 (correctness of partial deduction [43]) *Let* P *be a normal program,* G *a normal goal,* $\mathbf{A}$ *a finite, independent set of atoms, and* P' *a partial deduction of* P *wrt* $\mathbf{A}$ *such that* $P' \cup \{G\}$ *is* $\mathbf{A}$*-closed. Then the following hold:*

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

[3] also proposes an extension of Theorem 2.5 which uses a notion of *coveredness* instead of closedness. The basic idea is to restrict the attention to those parts of the specialised program P' which can be reached from G.

Example 2.6 Let P be the following program:

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

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

(3) $inboth(X, L1, L2) \leftarrow member(X, L1), member(X, L2)$

Then the following is a partial deduction wrt $\mathbf{A} = \{inboth(X, [a, b, c], [c, d, e])\}$ such that the conditions of Theorem 2.5 are verified for the goal $G = \leftarrow inboth(X, [a, b, c], [c, d, e])$.

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

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

(3') $inboth(c, [a, b, c], [c, d, e]) \leftarrow$

Note that the original unspecialised program P is also a partial deduction wrt $\mathbf{A} = \{member(X, L), inboth(X, L1, L2)\}$ which furthermore satisfies the correctness conditions of Theorem 2.5 for any goal G. In other words, neither Definition 2.3 nor the conditions of Theorem 2.5 ensure that any specialisation has actually been performed. Nor do they give any indication on how to construct a suitable set $\mathbf{A}$ and a suitable partial deduction wrt $\mathbf{A}$ satisfying the correctness criteria for a given goal G of interest. These are all considerations generally delegated to the *control* of partial deduction, which we discuss next.

2.2 Control of Partial Deduction

In partial deduction one usually distinguishes two levels of control [18, 51]:

- the *global control*, in which one chooses the set $\mathbf{A}$, i.e. one decides *which* atoms will be partially deduced, and
- the *local control*, in which one constructs the finite (possibly incomplete) SLDNF-trees for each individual atom in $\mathbf{A}$ and thus determines *what* the definitions for the partially deduced atoms look like.

In the following we examine how these two levels of control interact. In fact, when controlling partial deduction the three following, often conflicting, aspects have to be reconciled:

1. *Correctness*, i.e. ensuring that Theorem 2.5 or its extension can be applied. This can be divided into a local condition, requiring the constructing of non-trivial trees, and into a global one related to the independence and coveredness (or closedness) conditions.
2. *Termination.* This aspect can again be divided into a local and a global one. First, the problem of keeping each SLDNF-tree finite is referred to as the *local* termination problem. Secondly keeping the set $\mathbf{A}$ finite is referred to as the *global* termination problem.
3. *Precision.* For precision we can again discern two aspects. One which we might call *local* precision and which is related to the unfolding rule and to the fact that (potential for) specialisation can be lost if we stop unfolding an atom in $\mathbf{A}$ prematurely. Indeed, when we stop the unfolding process at a given goal Q, then all the atoms in Q are treated separately (partial deductions are defined for sets of *atoms* and not for sets of *goals*). For instance if we stop the unfolding process in Example 2.6 for $G = \leftarrow inboth(X, [a, b, c], [c, d, e])$ at the goal $Q = \leftarrow member(X, [a, b, c]), member(X, [c, d, e])$, partial deduction will not be able to infer that the only possible answer for Q and G is $\{X/c\}$. Another important issue in the context of local precision and specialisation is the choice of the particular selected literal.

   The second aspect could be called the *global* precision related to the set $\mathbf{A}$. In general having a more precise and fine grained set $\mathbf{A}$ (with more *instantiated* atoms) will lead to better specialisation. For instance given the set $\mathbf{A} = \{member(a, [a, b]), member(c, [d])\}$ partial deduction can perform much more specialisation (i.e. detecting that the goal $\leftarrow member(a, [a, b])$ always succeeds exactly once and that $\leftarrow member(c, [d])$ fails) than given the less instantiated set $\mathbf{A}' = \{member(X, [Y|T])\}$.

A good partial deduction algorithm will ensure correctness and termination while maximising the precision of point 3.

Let us now examine a bit closer how those three conflicting aspects can be reconciled and combined.

On the side of correctness there are two ways to ensure the independence condition. One is to apply a generalisation operator like the *msg*[2] on all the atoms in $\mathbf{A}$ which are not independent (first proposed in [3]). Applying this technique e.g. to the dependent set $\mathbf{A} = \{member(a, L), member(X, [b])\}$ yields the independent set $\{member(X, L)\}$. This approach also alleviates to some extent the global termination problem. However, it also diminishes the global precision and, as can be guessed from the above example, can seriously diminish the potential for specialisation.

This loss of precision can be completely avoided by using a *renaming* transformation to ensure independence. Renaming will map dependent atoms to new predicate symbols and thus always generate an independent set without precision loss. For instance the dependent set $\mathbf{A}$ above can be transformed into the independent set $\mathbf{A}' = \{member(a, L), member'(X, [b])\}$. The renaming transformation also has to map the atoms inside the residual program as well as the partial deduction goal to the correct versions of $\mathbf{A}'$ (e.g. rename the goal $G = \leftarrow member(a, [a, c]), member(b, [b])$ into $\leftarrow member(a, [a, c]), member'(b, [b])$). Renaming can often be combined with argument filtering to improve the efficiency of the specialised program. For instance, instead of renaming $\mathbf{A}$ into the set $\mathbf{A}'$ above, $\mathbf{A}$ would be renamed into $\{mem_a(L), mem_b(X)\}$ and the goal G would be renamed into $\leftarrow mem_a([a, c]), mem_b(b)$. For further details about filtering see e.g. [20] or [2] where the filtering phase is performed as a *one-step post-processing renaming*. See also [54], where filtering is obtained automatically when using folding to simulate partial evaluation. Filtering has also been referred to as "pushing down meta-arguments" in [64] or "PDMA" in [52]. In functional programming the term of "arity raising" has also been used.

Renaming and filtering are used in a lot of practical approaches (e.g. [17, 18, 20, 38, 33, 34]) and adapted correctness results can be found in [2]. See also the more powerful filtering techniques in [41].

The local control component is encapsulated in the unfolding rule, defined above. In addition to local correctness, termination and precision, the requirements on unfolding rules also include avoiding search space explosion as well as work duplication. One particular class of unfolding rules, addressing the two latter points, are based on *determinacy* [20, 18, 17]. Basically these rules stop unfolding as soon as a choice-point is encountered. We will define determinate unfolding rules as follows:

Definition 2.7 (determinate unfolding) *A tree is* determinate *if the root node is not a leaf node and if each node has either at most 1 child or has only leaves as its children. An unfolding rule is* (purely) determinate *if for every program P and every goal G it returns a determinate SLDNF-tree. An unfolding rule is* lookahead determinate *if for every program P and every goal G it returns an SLDNF-tree τ such that the subtree τ^- of τ, obtained by removing the failed branches, is determinate.*

Methods solely based on determinacy, avoid search space explosion and limit work duplication[3], but can be somewhat too conservative. Also, in itself, determinate unfolding

[2] Most specific generalisation, also known as anti-unification or least general generalisation, see for instance [32].

[3] Under the condition that non-determinate unfolding steps follow the computation rule of the underlying system.

does not guarantee termination, as there can be infinitely failing determinate computations. Termination can be ensured by imposing a depth bound, but much more refined approaches to ensure local termination exist. The methods in [6, 50, 49, 46] are based on well-founded orders, inspired by their usefulness in the context of static termination analysis (see e.g. [13, 10]). Instead of well-founded ones, *well-quasi orders* can be used [4, 58]. Homeomorphic embedding [63, 40] on selected atoms has recently gained popularity as the basis for such an order. These techniques ensure termination, while at the same time allowing unfolding related to the structural aspect of the program and goal to be partially deduced, by for instance allowing the consumption of relevant partial input inside the atoms of $\mathbf{A}$.

So if we use renaming to ensure independence and suppose that the local termination and precision problems have been solved, e.g. by [6, 50, 49, 46], we are still left with the problem of ensuring *closedness* and *global termination* while minimising the *global precision loss*. We call this combination of problems the *control of polyvariance problem* because it is very closely related to how many different specialised version of some given predicate should be put into $\mathbf{A}$.[4] It is this problem we address in this paper.

Let us examine how the 3 subproblems of the control of polyvariance problem are related.

- *Closedness vs. Global Termination*
  Closedness can be simply ensured by repeatedly adding the uncovered (i.e not satisfying Definition 2.4 for $\mathbf{A}$-closedness) atoms to $\mathbf{A}$ and unfolding them. Unfortunately this process generally leads to non-termination (even when using the *msg* to ensure independence). The classical example illustrating this non-termination is the "reverse with accumulating parameter" program (see Example 3.7 below or e.g. [46, 50]).
- *Global Termination vs. Global Precision*
  To ensure finiteness of $\mathbf{A}$ we can repeatedly apply an "abstraction" operator on $\mathbf{A}$ which generates a set of more general atoms. Unfortunately this induces a loss of global precision.

By using the two ideas above to (try to) ensure coveredness and global termination, we can formulate a generic partial deduction algorithm. First, the concept of an abstraction has to be defined.

Definition 2.8 (abstraction) *Let* $\mathbf{A}$ *and* $\mathbf{A}'$ *be sets of atoms. Then* $\mathbf{A}'$ *is an* abstraction *of* $\mathbf{A}$ *iff every atom in* $\mathbf{A}$ *is an instance of an atom in* $\mathbf{A}'$. *An* abstraction operator *is an operator which maps every finite set of atoms to a finite abstraction of it.*

The above definition of *abstract* guarantees that any partial deduction wrt $\mathbf{A}'$ is also correct wrt any atom in $\mathbf{A}$. Note that sometimes an abstraction operator is also referred to as a *generalisation operator*.

The following generic scheme, based on a similar one in [17, 18], describes the basic layout of practically all algorithms for controlling partial deduction.

Algorithm 2.9 (standard partial deduction)
Input: *A program* P *and a goal* G
Output: *A specialised program* P'
Initialise: $i = 0$, $\mathbf{A}_0 = \{A \mid A \text{ is an atom in } G\}$
repeat

[4] A method is called *monovariant* if it allows only one specialised version per predicate.

for *each* $A_k \in \mathbf{A}_i$, *compute a finite SLDNF-tree* τ_k *for* $P \cup \{\leftarrow A_k\}$ *by applying an unfolding rule* U;
let $\mathbf{A}'_i := \mathbf{A}_i \cup \{B_l | B_l$ *is an atom in a leaf of some tree* τ_k, *such that*
B_l *is not an instance*[5] *of any* $A_j \in \mathbf{A}_i\}$;
let $\mathbf{A}_{i+1} := abstract(\mathbf{A}'_i)$; *where* abstract *is an abstraction operator*
$i := i + 1$
until $\mathbf{A}_{i+1} = \mathbf{A}_i$
Apply a renaming transformation to $\mathbf{A}_i$ *to ensure independence and then construct* P' *by taking resultants.*

In itself the use of an abstraction operator does not yet guarantee global termination. But, if the above algorithm terminates then closedness (modulo renaming) is ensured. With this observation we can reformulate the *control of polyvariance problem* as one of finding an *abstraction operator which minimises loss of precision and ensures termination.*

A very simple abstraction operator which ensures termination can be obtained by imposing a finite maximum number of atoms in $\mathbf{A}_i$ and using the *msg* to stick to that finite number. For example, in [50] one atom per predicate is enforced by using the *msg*. However, using the *msg* in this way can induce an even bigger *loss of precision* (compared to using the *msg* to ensure independence) because it will now also be applied on *independent* atoms. For instance, calculating the *msg* for the set of atoms $\{solve(p(a)), solve(q(f(b)))\}$ yields the atom $solve(X)$ and all potential for specialisation is probably lost.

In [50] this problem has been remedied to some extent by using a static pre-processing renaming phase (as defined in [3]) which will generate one extra (renamed) version for the top-level atom to be specialised. However, this technique only works well if all relevant input can be consumed in one go (i.e. one unfolding) of this top-most atom. Apart from the fact that this huge unfolding is not always a good idea from a point of view of efficiency (e.g. it can considerably slow down the program due to search space explosion), in a lot of cases this simply cannot be accomplished (for instance if partial input is not consumed but carried along, like the representation of an object-program inside a meta-interpreter).

The basic goal pursued in the remainder of this paper is to define a flexible abstraction operator which does not exhibit this dramatic loss of precision and provides a fine-grained control of polyvariance, while still guaranteeing termination of the partial deduction process.

For a recent approach (orthogonal to ours), which tackles this problem from another perspective, see [51]. In this approach structure is added to the set of atoms $\mathbf{A}$ allowing the abstraction operator to be applied more selectively. We will discuss how these two approaches can be reconciled in Section 7.

3 Abstraction Using Characteristic Trees

In the previous section we have presented the generic partial deduction Algorithm 2.9. This algorithm is parametrised by an unfolding rule for the local control and by an abstraction operator for the control of polyvariance. The abstraction operator examines a set of atoms and then decides which of the atoms should be abstracted and which ones should be left unmodified. An abstraction operator like the *msg* is just based on the *syntactic structure* of

[5] Instead of an instance check one can also use a variant check. This gives more precision, at the cost of an increased danger for non-termination.

the atoms to be specialised. This is generally not such a good idea. Indeed, two atoms can be unfolded and specialised in a very similar way in the context of one program P_1, while in the context of another program P_2 their specialisation behaviour can be drastically different. The syntactic structure of the two atoms is of course unaffected by the particular context and an operator like the *msg* will perform exactly the same abstraction within P_1 and P_2, although vastly different generalisations might be called for.

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

Example 3.1 Let P be the append program:

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

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

Let $\mathbf{A} = \{B, C\}$ be a set of atoms, with $B = append([a], X, Y)$ and $C = append(X, [a], Y)$. Note that A and B have common instances. Typically a partial deducer will unfold the two atoms of $\mathbf{A}$ in the way depicted in Figure 1, returning the finite SLD-trees τ_B and τ_C. These two trees, as well as the associated resultants, have a very different structure. The atom $append([a], X, Y)$ has been fully unfolded and we obtain for $resultants(\tau_B)$ the single fact:

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

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

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

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

However, it is very easy to come up with another context in which the difference between atoms with identical structure to B and C is almost indiscernible. Take for instance the following program P^* in which the predicate *compos* no longer appends two lists but finds common elements at common positions:

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

The associated finite SLD-trees τ_B^* and τ_C^*, depicted in Figure 2, are now almost fully identical. In that case, it is not useful to keep different specialised versions for $B^* = compos([a], X, Y)$ and $C^* = compos(X, [a], Y)$ (which, apart from the predicate symbol, are identical to B and C respectively) because the following single set of specialised clauses could be used for B^* and C^* without specialisation loss:

$compos([a|T_1], [a|T_2], [a]) \leftarrow$

This illustrates that the syntactic structures of B, C and B^*, C^* alone provide insufficient information for a satisfactory control of polyvariance and that a refined abstraction operator should also take the associated SLD(NF)-trees into consideration.

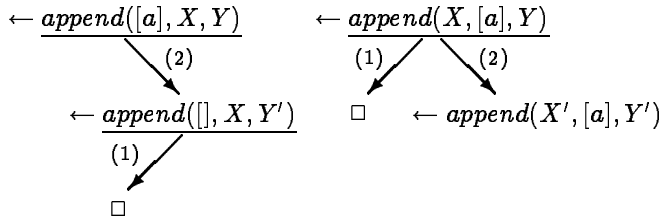

Figure 1: SLD-trees τ_B and τ_C for Example 3.1

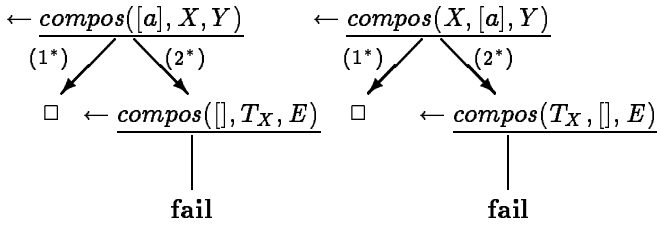

Figure 2: SLD-trees τ_B^* and τ_C^* for Example 3.1

3.1 Characteristic Paths and Trees

As motivated above, a refined abstraction operator should only generalise two (or more) atoms if their associated finite SLDNF-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 everything is taken into account, i.e. two atoms are abstracted only 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 [17], do just that: they characterise the essential structure of SLDNF-derivations and trees.

Definition 3.2 (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 δ of $P \cup \{G_0\}$. The* characteristic path *of the derivation δ is the sequence $\langle l_0 \circ c_0, \ldots, l_{n-1} \circ c_{n-1} \rangle$, where l_i is the position of the selected literal in G_i, and c_i is defined as:*

- *if the selected literal is an atom, then c_i is the number of the clause chosen to resolve with G_i.*
- *if the selected literal is $\neg p(\bar{t})$, then c_i is the predicate p.*

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

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

Recall that an SLDNF-derivation D can be either failed, incomplete, successful or infinite. As we will see below, characteristic paths will only be used to characterise *finite* and *non-failed* derivations of *atomic* goals, corresponding to the atoms to be partially deduced. Still, one might wonder why a characteristic path does not contain information on whether the associated derivation is successful or incomplete. The following proposition gives an answer to that question.

Proposition 3.3 *Let P be a normal program and let G_1, G_2 be two goals with the same number of literals. Let δ_1, δ_2 be two non-failed, finite derivations for $P \cup \{G_1\}$ and $P \cup \{G_2\}$ respectively. Also let δ_1 and δ_2 have the same characteristic path p. Then*
(1) δ_1 is successful iff δ_2 is and
(2) δ_1 is incomplete iff δ_2 is.

Proof As δ_1 and δ_2 can only be successful or incomplete, points (1) and (2) are equivalent and it is sufficient to prove point (1). Also, as δ_1 and δ_2 have the same characteristic path they must have the same length (i.e. same number of derivation steps) and we will prove the lemma by induction on the length of δ_1 and δ_2.
Induction Hypothesis: Proposition 3.3 holds for derivations δ_1, δ_2 with length $\leq n$.
Base Case: δ_1, δ_2 have the length 0.
This means that G_1 is the final goal of δ_1 and G_2 the final goal of δ_2. As G_1 and G_2 have the same number of literals it is impossible to have that $G_1 = \Box$ while $G_2 \neq \Box$ or $G_1 \neq \Box$ while $G_2 = \Box$, where $\Box$ denotes the empty goal.
Induction Step: δ_1, δ_2 have length $n+1$.
Let $R_0, \ldots, R_{n+1}$ be the sequence of goals of δ_1 (with $R_0 = G_1$) and let $Q_0, \ldots, Q_{n+1}$ be the sequence of goals of δ_2 (with $Q_0 = G_2$). Let δ_1' be the suffix of δ_1 whose sequence of goals is $R_1, \ldots, R_{n+1}$. Similarly, let δ_2' be the suffix of δ_2 whose sequence of goals is $Q_1, \ldots, Q_{n+1}$. Let $p = \langle l_0 \circ c_0, \ldots, l_n \circ c_n \rangle$ be the characteristic path of δ_1 and δ_2. There are two possibilities for $l_0 \circ c_0$, corresponding to whether a positive or negative literal has been selected. If a negative literal has been selected then (for both R_0 and Q_0) one literal has been removed and R_1 and Q_1 have the same number of literals. Similarly if a positive literal has been selected then trivially R_1 and Q_1 have the same number of literals (because the same clause c_1 in the same program P has been used). In both cases R_1 and Q_1 have the same number of literals and we can therefore apply the induction hypothesis on δ_1' and δ_2' to prove that δ_1' is successful iff δ_2' is. Finally, because δ_1 (respectively δ_2) is successful iff δ_1' (respectively δ_2') is, the induction step holds. $\Box$

As a corollary of the above lemma we have that, in the context of finite, non-failed derivations of atomic goals, the information about whether the derivation associated with a characteristic path is incomplete or successful is already implicitly present and no further precision would be gained by adding it.

Also, once the top-level goal is known, the characteristic path is sufficient to reconstruct all the intermediate goals as well as the final one.

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

Definition 3.4 (characteristic tree) *Let G be a goal and P a normal program and τ be a finite SLDNF-tree for $P \cup \{G\}$. Then the* characteristic tree $\hat{\tau}$ of τ *is the set containing*

the characteristic paths of the non-failed SLDNF-derivations associated with the branches of τ. $\hat{\tau}$ is called a characteristic tree iff it is the characteristic tree of some finite SLDNF-tree.

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

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

Example 3.5 The characteristic trees of the finite SLD-trees τ_B and τ_C in Figure 1 are $\{\langle 1 \circ 2, 1 \circ 1\rangle\}$ and $\{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$ respectively. The characteristic trees of the finite SLD-trees τ_B^* and τ_C^* in Figure 2 are both $\{\langle 1 \circ 1^*\rangle\}$.

The following observation underlines the interest of characteristic trees in the context of partial deduction. Indeed, the characteristic tree of an atom A explicitly or implicitly captures the following important aspects of specialisation:

- the branches pruned through the unfolding process (namely those that are absent from the characteristic tree). For instance by looking at the characteristic trees of τ_B, τ_C of Examples 3.1 and 3.5, we can see that two branches have been pruned for the atom B (thereby removing recursion) whereas no pruning could be performed for C.
- how deep $\leftarrow A$ has been unfolded and which literals and clauses have been resolved with each other in that process. This captures the computation steps that have already been performed at partial deduction time.
- the number of clauses in the resultants of A (namely one per characteristic path) and also (implicitly) which predicates are called in the bodies of the resultants. 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.

In other words, the characteristic tree τ_A captures all the relevant local specialisation aspects of A. An aspect that is not explicitly captured by the characteristic tree τ_A is how the atoms in the leaves of the associated SLDNF-tree are further specialised. These call patterns influence the set of atoms to be partially deduced, i.e. they influence the global control and precision.

Finally, note that characteristic trees only contains paths for the non-failed branches and therefore do not capture *how* exactly some branches were pruned. However, this is of no relevance, because the failing branches do not materialise within the resultants (i.e. the specialised code generated for the atoms).

In summary, characteristic trees seem to be an almost ideal vehicle for a refined control of polyvariance [20, 17], a fact we will try to exploit in the following section.

3.2 An Abstraction Operator using Characteristic Trees

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

Definition 3.6 ($chabs_{P,U}$) *Let P be a normal program, U an unfolding rule and $\mathbf{A}$ a set of atoms. For every characteristic tree τ, let $\mathbf{A}_\tau$ be defined as $\mathbf{A}_\tau = \{A \mid A \in S \wedge chtree(\leftarrow A, P, U) = \tau\}$. The abstraction operator $chabs_{P,U}$ is then defined as:*
$chabs_{P,U}(\mathbf{A}) = \{msg(\mathbf{A}_\tau) \mid \tau$ *is a characteristic tree*$\}$.

The following example illustrates the above definition.

Example 3.7 Let P be the program reversing a list using an accumulating parameter:

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

We will use $chabs_{P,U}$ with a purely determinate unfolding rule U (allowing non-determinate steps only in the root) inside the generic Algorithm 2.9. When starting out with the set $\mathbf{A}_0 = \{rev([a|B], [], R)\}$ the following steps are performed by Algorithm 2.9:

- unfold the atom in $\mathbf{A}_0$ (see Figure 3) and add the atoms in the leaves yielding $\mathbf{A}'_0 = \{rev([a|B], [], R), rev(B, [a], R)\}$.
- apply the abstraction operator:
  $\mathbf{A}_1 = chabs_{P,U}(\mathbf{A}'_0) = \{rev([a|B], [], R), rev(B, [a], R)\}$ because the atoms in $\mathbf{A}'_0$ have different characteristic trees.
- unfold the atoms in $\mathbf{A}_1$ (see Figure 3) and add the atoms in the leaves yielding
  $\mathbf{A}'_1 = \{rev([a|B], [], R), rev(B, [a], R), rev(T, [H, a], R)\}$.
- apply the abstraction operator: $\mathbf{A}_2 = chabs_{P,U}(\mathbf{A}'_1) =$
  $\{rev([a|B], [], R), rev(T, [A|B], R)\}$, because $rev(B, [a], R)$ and $rev(T, [H, a], R)$ have the same characteristic tree (see Figure 3).
- unfold the atoms in $\mathbf{A}_2$ and add the atoms in the leaves yielding:
  $\mathbf{A}'_2 = \{rev([a|B], [], R), rev(T, [A|B], R), rev(T', [H', A|B], R)\}$.
- apply the abstraction operator: $\mathbf{A}_3 = chabs_{P,U}(\mathbf{A}'_2) = \mathbf{A}_2$ and we have reached a fix-point and thus obtain the following partial deduction satisfying the coveredness condition (and which is also independent without renaming):

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

Because of the selective application of the *msg*, no loss of precision has been incurred by $chabs_{P,U}$, i.e. the pruning and pre-computation for e.g. the atom $rev([a|B], [], R)$ has been preserved. An abstraction operator allowing just one version per predicate would have lost this local specialisation, while a method with unlimited polyvariance (also called dynamic renaming, in e.g. [2]) does not terminate.

For this example, $chabs_{P,U}$ provides a terminating and fine grained control of polyvariance, conferring just as many versions as necessary. The abstraction operator $chabs_{P,U}$ is thus much more flexible than e.g. the static pre-processing renaming of [3, 50]).

The above example is thus very encouraging, and one might hope that $chabs_{P,U}$ always preserves the characteristic trees upon generalisation and that it might already provide a refined solution to the control of polyvariance problem. Unfortunately, although for a lot of practical cases $chabs_{P,U}$ performs quite well, it does not always preserve the characteristic trees, entailing a sometimes quite severe loss of precision and specialisation. Let us examine an example:

Example 3.8 Let P be the program:

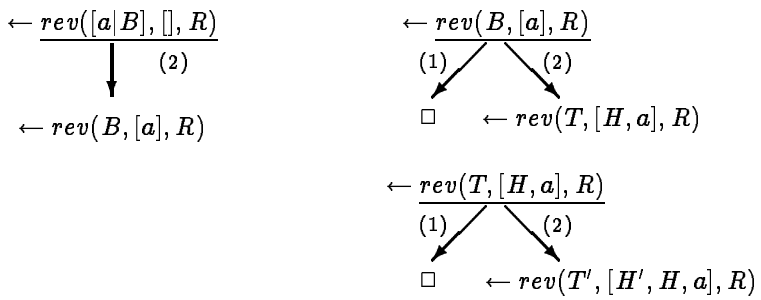

Figure 3: SLD-trees for Example 3.7

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

Take $\mathbf{A} = \{p(a), p(b)\}$. Using any non-trivial unfolding, the goals $\leftarrow p(a)$ and $\leftarrow p(b)$ have the same characteristic tree $\tau = \{\langle 1 \circ 1 \rangle\}$. Thus $chabs_{P,U}(S) = \{p(X)\}$ and unfortunately $\leftarrow p(X)$ has the characteristic tree $\tau' = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2 \rangle\}$ and the pruning that was possible for the atoms $p(a)$ and $p(b)$ has been lost. More importantly there exists *no* atom, more general than $p(a)$ and $p(b)$, which has τ as its characteristic tree.

The problem in the above example is that, through generalisation, a new non-failed derivation has been added (thereby modifying the characteristic tree). Starting in the next section we will present a solution to this problem by adding constraints to the generalisation in order to ensure that such new non-failed derivations cannot arise. For this example, we might produce as generalisation the atom $p(X)$ with the added constraint that X is different from c.

Another problem occurs when negative literals are selected by the unfolding rule.

Example 3.9 Let us examine the following program P:

(1) $p(X) \leftarrow \neg q(X)$
(2) $q(f(X)) \leftarrow$

For this program the goals $\leftarrow p(a)$ and $\leftarrow p(b)$ have the same characteristic tree $\{\langle 1 \circ 1, 1 \circ q \rangle\}$. The abstraction operator $chabs_{P,U}$ will therefore produce $\{p(X)\}$ as a generalisation of $\{p(a), p(b)\}$. Again however, $\leftarrow p(X)$ has the different characteristic tree $\{\langle 1 \circ 1 \rangle\}$, because the non-ground literal $\neg q(X)$ cannot be selected in the resolvent of $\leftarrow p(X)$. The problem is that, by generalisation, a previously selectable ground negative literal in a resolvent can become non-ground and thus no longer selectable by SLDNF.

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

- important opportunities for specialisation can be lost and
- termination of Algorithm 2.9 can be undermined.

Let us illustrate the possible precision losses through two simple, but more realistic examples.

Example 3.10 Let P be the following program, checking two lists for equality.

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

Given a purely determinate unfolding rule, the atoms $A = eqlist([1,2],L), B = eqlist(L,[3,4])$ have the same characteristic tree $\tau = \{\langle 1 \circ 2, 1 \circ 2, 1 \circ 1\rangle\}$. Unfortunately the abstraction operator $chabs_{P,U}$ is unable to preserve τ. Indeed, $chabs_{P,U}(\{A,B\}) = \{eqlist(X,Y)\}$ whose characteristic tree is $\{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$ and the precomputation and pruning performed on A and B has been lost.

The previous example is taken from [17], whose abstraction mechanism can solve the example. The following example can, however, not be solved by [17].

Example 3.11 Let P be the well known *member* program, already encountered in Example 2.6.

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

Then both $A = member(a,[b,c|T])$ and $B = member(a,[c,d|T])$ have the same characteristic tree $\tau = \{\langle 1\circ 2, 1\circ 2, 1\circ 1\rangle, \langle 1\circ 2, 1\circ 2, 1\circ 2\}$, using a purely determinate unfolding rule. However, $chabs_{P,U}(\{A,B\}) = \{member(a,[X,Y|T])\}$ whose characteristic tree is unfortunately $\{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$. The precomputation and pruning that was possible for A and B has again not been preserved by $chabs_{P,U}$. Applying e.g. Algorithm 2.9, we obtain at the next iteration the set $chabs_{P,U}(\{member(a,[X,Y|T]), member(a,[Y|T])\}) = \{member(a,[Y|T])\}$ and then the final set $chabs_{P,U}(\{member(a,[Y|T]), member(a,T)\}) = \{member(a,T)\}$. We thus obtain the following suboptimal, unpruned program P', performing redundant computations for both A and B:

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

Let us discuss the termination aspects next. One might hope that $chabs_{P,U}$ ensures termination of partial deduction Algorithm 2.9 if the number of characteristic trees is finite (which can be ensured by using a depth-bound for characteristic trees[6] or by the more sophisticated technique of [40] — we will return to this issue in Section 7).

Actually if the characteristic trees are preserved, then the abstraction operator $chabs_{P,U}$ does ensure termination of Algorithm 2.9. To prove this we have to show that when we add a set of atoms L to $\mathbf{A}_i$, then either $chabs_{P,U}(\mathbf{A}_i \cup L) = \mathbf{A}_i$ (i.e. we have reached a fixpoint in our algorithm) or $\|chabs_{P,U}(\mathbf{A}_i \cup L)\| < \|\mathbf{A}_i\|$ for some well-founded measure function $\|.\|$. Such a measure function is established in Appendix B and the above property is proven.

So, if characteristic trees are preserved by the abstraction operator, then termination of partial deduction is guaranteed. However, if characteristic trees are not preserved by the abstraction operator, then the proof of Appendix B no longer holds and indeed termination is no longer guaranteed (even assuming a finite number of characteristic trees)! An example illustrating this, can be found in [37]. The example exploits the non-monotonic nature of Algorithm 2.9. Indeed, termination of partial deduction based on $chabs_{P,U}$ and given a finite number of characteristic trees can also be ensured by making Algorithm 2.9 monotonic, i.e. instead of executing $\mathbf{A}_{i+1} := abstract(\mathbf{A}'_i)$ we would perform $\mathbf{A}_{i+1} := A_i \cup abstract(\mathbf{A}'_i)$. From a practical point of view, this solution is, however, not very satisfactory as it might unnecessarily increase the polyvariance, possibly leading to a code explosion of the specialised program as well as an increase in the transformation complexity. The former can be solved by a post-processing phase removing unnecessary polyvariance. However, by using an altogether

[6] The unfolding rule can still unfold as deep as it wants to ! See the discussion in Section 7.

more precise abstraction operator, preserving characteristic trees, these two problems will disappear automatically. We will then obtain an abstraction operator for partial deduction with optimal local precision (in the sense that all the local specialisation achieved by the unfolding rule is preserved by the abstraction) and which guarantees termination. This quest is pursued in the remainder of this paper.

3.3 Characteristic Trees in the Literature

Characteristic trees have been introduced in the context of definite programs and determinate unfolding rules by Gallagher and Bruynooghe in [20] and were later refined by Gallagher in [17] leading to the definitions that we have presented in this paper. Both [20] and [17] use a refined version of the abstraction operator $chabs_{P,U}$ and [17] uses a partial deduction algorithm very similar to Algorithm 2.9. In both [20] and [17] termination properties are claimed. No claim as to the preservation of characteristic trees is made in [17]. However, the authors of [20] actually claim in Lemma 4.11 to have found an operator (namely *chcall*) which, in the case of definite programs and purely determinate unfolding rules without lookahead (cf. Definition 2.7), preserves a structure quite similar to characteristic trees as of Definition 3.4.

Unfortunately this Lemma 4.11 is false and cannot be easily rectified. In Appendix A we provide a detailed description of a counterexample to this Lemma 4.11. Furthermore, in a lot of cases, the abstraction operators of [20] and [17] behave exactly like $chabs_{P,U}$, and the examples in this paper and in [37] actually provide counterexamples not only for the precision claim of [20] but also for the termination claims of both [20] and [17]. There are in fact some further problems with the abstraction operator of [17]. For instance the Example 3.9 with negation poses problems to [17] ([20] is restricted to definite programs, so the problem does not appear there) and unfolding rules which are not purely determinate can also cause problems. More detailed descriptions can be found in [37] as well as in [36], where the counterexample to Lemma 4.11 of [20] was first presented. The problems of negative literals and non-purely determinate unfolding rules will be touched again later in this paper.

4 Constrained Partial Deduction

In the previous chapter we have dwelled upon the appeal of characteristic trees for controlling polyvariance, but we have also highlighted the difficulty of preserving characteristic trees in the abstraction process as well as the ensuing problems concerning termination and precision. We have hinted briefly at the possibility of using constraints to solve this entanglement. In this section we present the framework of *constrained partial deduction*, which will allow us to incorporate constraints inside partial deduction. In Subsection 4.1 we first present some background on constraint logic programming. In Subsection 4.2 we present the framework of constrained partial deduction, whose correctness we then prove in Subsection 4.3.

Also, from now on we will restrict ourselves to definite programs and goals. We will return to the problem of negative literals in Section 7.

4.1 Constraint Logic Programming

To formalise constraints and their effect, we need some basic terminology from *constraint logic programming (CLP)* [24].

First, the predicate symbols are partitioned into two disjoint sets Π_c (the predicate symbols to be used for constraints, notably including "=") and Π_b (the predicate symbols for user-defined predicates). The signature Σ contains all predicate and function symbols with their associated arity. A *constraint* is a first-order formula whose predicate symbols are all contained in Π_c. A constraint is called *primitive* iff it contains no connectives or quantifiers (i.e. it is of the form $p(\bar{t})$ where $p \in \Pi_c$). A formula, atom or literal whose predicate symbols are all contained in Π_b will be called *ordinary*. We will often use the connective " $\square$ " (and as usual in standard logic programming ",") in the place of "$\wedge$". A *CLP-goal* is denoted by $\leftarrow c \square B_1, \ldots, B_n$, where c is a constraint and $B_1, \ldots, B_n$ are ordinary atoms. A *CLP-clause* is denoted by $H \leftarrow c \square B_1, \ldots, B_n$, where c is a constraint and $H, B_1, \ldots, B_n$ are ordinary atoms. Note that, although we do not allow negation within $H, B_1, \ldots, B_n$, negation can still be used within the constraint c. A *CLP-program* is a set of CLP-clauses. Note that CLP-programs will only be required as an intermediary step, the initial and the final specialised programs will be ordinary programs.

The semantics of constraints is given by a Σ-*structure* $\mathcal{D}$, consisting of a domain D and an assignment of functions and relations on D to function symbols in Σ and to predicate symbols in Π_c. Given a constraint c, we will denote by $\mathcal{D} \models c$ the fact that c is true under the interpretation provided by $\mathcal{D}$. Also, c will be called $\mathcal{D}$-*satisfiable* iff $\mathcal{D} \models \tilde{\exists}(c)$, where $\tilde{\exists}(F)$ denotes the existential closure of a formula F. We will also use the standard notation $\tilde{\forall}(F)$ for the universal closure of a first-order formula F and $\tilde{\exists}_{\mathcal{V}}(F)$ (respectively $\tilde{\forall}_{\mathcal{V}}(F)$) for the existential (respectively universal) closure of F except for the variables in the set $\mathcal{V}$.

Applying a substitution on a constraint is defined inductively as follows:

- $p(\bar{t})\theta = p(\bar{t}\theta)$ for $p \in \Pi_c$
- $(F \circ G)\theta = F\theta \circ G\theta$ for $\circ \in \{\wedge, \vee, \leftarrow, \rightarrow, \leftrightarrow\}$.
- $(\neg F)\theta = \neg(F\theta)$
- $(\Pi X.F)\theta = \Pi X'.(F\theta')$ where X' is a new fresh variable not occurring in F and θ, and where $\theta' = \{X/X'\} \cup \{x/t \mid x/t \in \theta \wedge x \neq X\}$ for $\Pi \in \{\forall, \exists\}$.

Applying a substitution on a constraint is used to make explicit the fact that certain variables are determined. For example, $(\forall X.\neg(Y = f(X)))\{Y/g(X)\} = \forall Z.\neg(g(X) = f(Z))$. The following notations for constraints and CLP-goals will also prove useful:

- $holds_{\mathcal{D}}(c) =_{\text{Def}} \mathcal{D} \models \tilde{\forall}(c)$,
- $\theta \ sat_{\mathcal{D}} \ c =_{\text{Def}} holds_{\mathcal{D}}(c\theta)$.
- $vars(c) =_{\text{Def}}$ the free variables in c.
- $vars(\leftarrow c \square Q) =_{\text{Def}} vars(c) \cup vars(Q)$.

Note that for CLP-goals $\leftarrow c \square Q$ we will in fact require that $vars(c) \subseteq vars(Q)$[7] (meaning that actually $vars(\leftarrow c \square Q) = vars(Q)$). This will be ensured by applying the existential closure $\tilde{\exists}_{vars(Q)}(.)$ during derivation steps below (this existential closure makes no difference wrt $\mathcal{D}$-satisfiability, but it makes a difference wrt $holds_{\mathcal{D}}$).

We will now define a counterpart to SLD-derivations for CLP-goals. In our context of partial deduction, the initial and final programs are just ordinary logic programs (i.e. they can be seen as CLP-programs using just equality constraints over the structure of feature terms $\mathcal{FT}$, see [44]). In order for our constraint manipulations to be correct wrt the initial *ordinary* logic program, we have to ensure that equality is not handled in an unsound manner in the intermediate CLP-program. For instance, something like $a = b$ should not succeed in the CLP-program. In other words, if there is no SLD-refutation for $P \cup \{\leftarrow Q\}$ then there

[7] As the conjunction Q contains no quantifiers, $vars(Q)$ are the free variables of Q.

should be no CLP-refutation for any $P \cup \{\leftarrow c \square Q\}$ either. To ensure this property we use the following definition of a derivation, adapted from [16], in which substitutions are made explicit. This will also enable us to construct resultants in a straightforward manner.

Definition 4.1 *Let $CG = \leftarrow c \square L_1, \ldots, L_k$ a CLP-goal and $C = A \leftarrow B_1, \ldots, B_n$ a program clause such that $k \geq 1$ and $n \geq 0$. Then CG'* is derived from *CG and C in $\mathcal{D}$ using θ iff the following conditions hold:*

- *L_m is an atom, called the* selected *atom (at position m), in CG.*
- *θ is a relevant and idempotent mgu of L_m and A.*
- *CG' is the goal $\leftarrow c' \square Q$, where $Q = (L_1, \ldots, L_{m-1}, B_1, \ldots, B_n, L_{m+1}, \ldots, L_k)\theta$ and $c' = \tilde{\exists}_{vars(Q)}(c\theta)$.*
- *c' is $\mathcal{D}$-satisfiable.*

CG' is called a resolvent *of CG and C in $\mathcal{D}$.*

Definition 4.2 (complete CLP$_=(\mathcal{D})$-derivation) *Let P be a definite program and CG_0 a CLP-goal. A* complete CLP$_=(\mathcal{D})$-derivation *of $P \cup \{CG_0\}$ is a tuple $(\mathcal{G}, \mathcal{C}, \mathcal{S})$ consisting of a (finite or infinite) sequence of CLP-goals $\mathcal{G} = \langle CG_0, CG_1, \ldots \rangle$, a sequence $\mathcal{C} = \langle C_1, C_2, \ldots \rangle$ of variants of program clauses of P and a sequence $\mathcal{S} = \langle \theta_1, \theta_2, \ldots \rangle$ of mgu's such that:*

- *for $i > 0$, $vars(C_i) \cap vars(CG_0) = \emptyset$;*
- *for $i > j$, $vars(C_i) \cap vars(C_j) = \emptyset$;*
- *for $i \geq 0$, CG_{i+1} is derived from CG_i and C_{i+1} in $\mathcal{D}$ using θ_{i+1} and*
- *the sequences $\mathcal{G}, \mathcal{C}, \mathcal{S}$ are maximal (given the choice of the selected atoms).*

A *CLP$_=(\mathcal{D})$-refutation* is just a complete CLP$_=(\mathcal{D})$-derivation whose last goal contains no atoms, i.e. it can be written as $\leftarrow c \square \epsilon$ where ϵ denotes the empty sequence of atoms. A *finitely failed CLP$_=(\mathcal{D})$-derivation* is a finite, complete CLP$_=(\mathcal{D})$-derivation whose last goal is *not* of the form $\leftarrow c \square \epsilon$. There are thus 3 forms of complete CLP$_=(\mathcal{D})$-derivations: refutations, finitely failed ones and infinite derivations.

In the context of partial deduction we also allow incomplete derivations. A CLP$_=(\mathcal{D})$-derivation is defined like a complete CLP$_=(\mathcal{D})$-derivation but may, in addition to leading to success or failure, also lead to a last goal where no atom has been selected for a further derivation step. Derivations of the latter kind will be called *incomplete*.

We can also extend the notion of characteristic paths of SLD-derivations for ordinary goals to *$\mathcal{D}$-characteristic paths* of CLP$_=(\mathcal{D})$-derivations for CLP-goals, simply by replacing in Definition 3.2 the SLDNF-derivation δ by a CLP$_=(\mathcal{D})$-derivation. We will denote by $chpaths_{\mathcal{D}}(P, CG)$ the $\mathcal{D}$-characteristic paths of all CLP$_=(\mathcal{D})$-derivations for $P \cup \{CG\}$.

In order to construct resultants we also need the following, where $\theta|_{\mathcal{V}}$ denotes the restriction of the substitution θ to the set of variables $\mathcal{V}$:

Definition 4.3 *The* computed answer *of a finite, non-failed CLP$_=(\mathcal{D})$-derivation δ for $P \cup \{\leftarrow c \square G\}$ with the sequence $\theta_1, \ldots, \theta_n$ of mgu's, is the substitution $cas(\delta) = (\theta_1 \ldots \theta_n)|_{vars(G)}$. Also, the last goal of δ will be called the* resolvent *of δ.*

The following lemma will prove useful later on.

Lemma 4.4 *Let P be a definite program and $\leftarrow c \square Q$ be a CLP-goal. If there exists a CLP$_=(\mathcal{D})$-derivation for $P \cup \{\leftarrow c \square Q\}$ with computed answer θ and $\mathcal{D}$-characteristic path p then there exists an SLD-derivation for $P \cup \{\leftarrow Q\}$ with the same computed answer and characteristic path.*

Proof Straightforward, by definition of a CLP$_=(\mathcal{D})$-derivation. □

The concept of CLP$_=(\mathcal{D})$-trees can be defined just like the concept of SLD-trees: its branches are just CLP$_=(\mathcal{D})$-derivations instead of SLD-derivations. An unfolding rule is now one which, given a definite program P and a CLP-goal CG returns a finite CLP$_=(\mathcal{D})$-tree for $P \cup \{CG\}$. Finally, the *$\mathcal{D}$-characteristic tree* of a finite CLP$_=(\mathcal{D})$-tree T is simply obtained by taking the union of the $\mathcal{D}$-characteristic paths of the non-failed CLP$_=(\mathcal{D})$-derivations in T. We will use the notation $chtree_{\mathcal{D}}(CG, P, U)$ to refer to the $\mathcal{D}$-characteristic tree obtained for the CLP-goal CG via U in P.

4.2 A Framework for Constrained Partial Deduction

We will now present a generic partial deduction scheme which, instead of working on sets of ordinary atoms, will work on sets of *constrained atoms*. The richer possibilities conferred by the use of the constraints will notably allows us to present an abstraction operator which preserves characteristic trees in Section 5. However, the generic framework is not restricted to this particular application nor the corresponding constraint structure. Amongst others, it can also be used to "drive negative information" (using the terminology of supercompilation [67, 68]), handle built-ins (like $< /2, \backslash == /2$) much more precisely and even make use of type information or argument size relations. We will briefly return to this issue in Section 7.

Definition 4.5 *A* constrained atom *is formula of the form $c \square A$ where c is a constraint and A an ordinary atom such that the free variables of c are contained in the variables of A.*

Definition 4.6 ($valid_{\mathcal{D}}$) *Let $c \square A$ be a constrained atom. The set of* valid $\mathcal{D}$-instances *of $c \square A$ is defined as: $valid_{\mathcal{D}}(c \square A) = \{A\theta \mid \theta \; sat_{\mathcal{D}} \; c\}$.*

By definition of $sat_{\mathcal{D}}$, the set of valid $\mathcal{D}$-instances is *downwards-closed* (or closed under substitution, i.e. if $A \in valid_{\mathcal{D}}(c \square A)$ then so is any instance of A). The constraint within a constrained atom thus specifies a property that holds for all valid instances, which in our case correspond to the possible runtime instances.

We also need an instance notion on constrained atoms.

Definition 4.7 ($\mathcal{D}$**-instance**) *Let $c \square A$, $c' \square A'$ be constrained atoms. Then $c' \square A'$ is a* $\mathcal{D}$-instance *of $c \square A$, denoted by $c' \square A' \preceq_{\mathcal{D}} c \square A$, iff $A' = A\gamma$ and $valid_{\mathcal{D}}(c' \square A') \subseteq valid_{\mathcal{D}}(c \square A)$.*

For example, independently of $\mathcal{D}$, $\neg(X = c) \square p(X)$ is a $\mathcal{D}$-instance of $true \square p(X)$ because every substitution satisfies $true$. In turn, if $\mathcal{D}$ contains e.g. Clark's equality theory (CET, see e.g. [9, 42]) then $true \square p(b)$ is a $\mathcal{D}$-instance of $\neg(X = c) \square p(X)$ because $\{X/b\}$ $sat_{\mathcal{D}}$ $\neg(X = c)$ given CET (i.e. $CET \models \tilde{\forall}(\neg(b = c))$).

Definition 4.8 (**partial deduction of** $c \square A$) *Let P be a program and $c \square A$ a constrained atom. Let τ be a finite, non-trivial and possibly incomplete CLP$_=(\mathcal{D})$-tree for $P \cup \{\leftarrow c \square A\}$ generated via the unfolding rule U and let $\leftarrow c_1 \square G_1, \ldots, \leftarrow c_n \square G_n$ be the CLP-goals in the leaves of this tree. Let $\theta_1, \ldots, \theta_n$ be the computed answers of the CLP$_=(\mathcal{D})$-derivations from $\leftarrow c \square A$ to $\leftarrow c_1 \square G_1, \ldots, \leftarrow c_n \square G_n$ respectively. Then the set of CLP-resultants $\{A\theta_1 \leftarrow c_1 \square G_1, \ldots, A\theta_n \leftarrow c_n \square G_n\}$ is called the* partial deduction of $c \square A$ in P (using $\mathcal{D}$ via U).

Example 4.9 Let us return to the program P from Example 3.8:

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

When using a constraint structure $\mathcal{D}$ containing CET (or any other structure in which $\neg(c = c)$ is unsatisfiable), a partial deduction of $\neg(X = c) \square p(X)$ in P (using $\mathcal{D}$[8]) is:

(1') $p(X) \leftarrow \neg(X = c) \square \epsilon$

We now generate partial deductions not for sets of atoms, but for sets of *constrained* atoms. As such, the same atom A might occur in several constrained atoms but with different associated constraints. This means that renaming as a way to ensure independence imposes itself even more than in the standard partial deduction setting. In addition to renaming, we will also allow argument filtering, leading to the following definition.[9]

First, given a CLP-clause $C = H \leftarrow c \square B_1, \ldots, B_n$, each constrained atom of the form $\tilde{\exists}_{vars(B_i)}(c) \square B_i$ will be called a *constrained body atom of* C. This notion extends to CLP-programs by taking the union of the constrained body atoms of the clauses.

Definition 4.10 (atomic renaming, renaming function) *An* atomic renaming α *for a set* $\mathcal{A}$ *of constrained atoms maps each constrained atom in* $\mathcal{A}$ *to an atom such that*

- *for each* $c \square A \in \mathcal{A}$: $vars(\alpha(c \square A)) = vars(A)$
- *for* $CA, CA' \in \mathcal{A}$ *such that* $CA \neq CA'$: *the predicate symbols of* $\alpha(CA)$ *and* $\alpha(CA')$ *are distinct (but may occur in* $\mathcal{A}$*).*

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

$$\rho_\alpha(c \square A) = \alpha(c' \square A')\theta \text{ for some } c' \square A' \in \mathcal{A} \text{ with } A = A'\theta \wedge \ c \square A \preceq_{\mathcal{D}} c' \square A'.$$

We leave $\rho_\alpha(A)$ *undefined if* $c \square A$ *is not a* $\mathcal{D}$*-instance of an element in* $\mathcal{A}$*.*
A renaming function ρ_α *can also be applied to constrained goals* $c \square B_1, \ldots, B_n$*, by applying it individually to each constrained body atom* $c_i \square B_i$*. Finally, we can apply a renaming function also to ordinary goals by defining* $\rho_\alpha(G) = \rho_\alpha(true \square G)$*.*

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

Definition 4.11 (partial deduction wrt $\mathcal{A}$) *Let* P *be a program,* $\mathcal{A} = \{c_1 \square A_1, \ldots, c_n \square A_n\}$ *be a finite set of constrained atoms and let* ρ_α *be a renaming for* $\mathcal{A}$ *based on the atomic renaming* α*. For each* $i \in \{1, \ldots, n\}$*, let* R_i *be the partial deduction of* $c_i \square A_i$ *in* P *and let* $\hat{P} = \{R_i \mid i \in \{1, \ldots, n\}\}$*. Then the program* $\{\alpha(c_i \square A_i)\theta \leftarrow \rho_\alpha(c \square Bdy) \mid A_i\theta \leftarrow c \square Bdy \in R_i \wedge 1 \leq i \leq n \wedge \rho_\alpha(c \square Bdy)$ *is defined* $\}$ *is called the* partial deduction of P wrt $\mathcal{A}$, $\hat{P}$ and ρ_α.

We showed in Example 3.8 that without constraints it is in general impossible to abstract atoms while still preserving their characteristic trees. Let us revisit Example 3.8 and see how we can achieve preservation of characteristic trees using partial deduction of constrained atoms.

[8] We will often take the liberty to not always explicitly mention the constraint domain $\mathcal{D}$ which was used to construct partial deductions and assume that $\mathcal{D}$ is fixed and known.

[9] The more powerful optimisations in [41], which remove redundant arguments, are not incorporated in this paper. They can easily be added as a post-processing phase.

Example 4.12 Let P be the program of Examples 3.8 and 4.9. Also let us use the same constraint structure $\mathcal{D}$ as in Example 4.9, containing Clark's equality theory. In the context of P, we can abstract the constrained atoms $true \square p(a)$ and $true \square p(b)$ of Example 3.8 by the more general constrained atom $\neg(X = c) \square p(X)$, having the same $\mathcal{D}$-characteristic tree $\tau = \{\langle 1 \circ 1 \rangle\}$. As illustrated in Example 4.9, the additional match with clause (2) is pruned for $\neg(X = c) \square p(X)$, because $\neg(X = c)\{X/c\}$ is unsatisfiable in $\mathcal{D}$. The partial deduction of $\neg(X = c) \square p(X)$ based on $\alpha(\neg(X = c) \square p(X)) = p'(X)$ is thus

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

Note that $\rho_\alpha(\leftarrow \neg(X = c) \square \epsilon) = \epsilon$, i.e. the empty goal. The renaming of the run-time goal $\leftarrow p(a), p(b)$ is $\leftarrow p'(a), p'(b)$.

Note that in Definition 4.11 the original program P is completely "thrown away". This is actually what a lot of practical partial evaluators for functional or logic programming languages do, but is unlike e.g. the definitions in [43] (cf. Definition 2.3). However, there is no fundamental difference between these two approaches: keeping part of the original program can always be "simulated" very easily in our approach by using (un)constrained atoms of the form $true \square A$ combined with an atomic renaming α such that $\alpha(true \square A) = A$.

Also, note that the partial deduction wrt $\mathcal{A}$ is an ordinary logic program *without constraints*. The coveredness criterion presented in the next subsection, will ensure that the constraint manipulations have already been incorporated (by pruning certain resultants) and no additional constraint processing at run-time is needed.

4.3 Correctness of Constrained Partial Deduction

Let us first rephrase the coveredness condition of standard partial deduction in the context of constrained atoms. This definition will also ensure that the renamings, applied for instance in Definition 4.11, are always defined.

Definition 4.13 *Let $\hat{P}$ be a CLP-program and $\mathcal{A}$ a set of constrained atoms. Then $\hat{P}$ is called* $\mathcal{A}, \mathcal{D}$-covered *iff each of its constrained body atoms is a $\mathcal{D}$-instance of a constrained atom in $\mathcal{A}$.*

We can extend the above notion also to ordinary programs and goals by inserting the constraint *true* (e.g. $H \leftarrow Bdy$ is $\mathcal{A}, \mathcal{D}$-covered iff $H \leftarrow true \square Bdy$ is).

The main correctness result for constrained partial deduction is as follows.

Theorem 4.14 *Let P be a definite program, G a definite goal, $\mathcal{A}$ a finite set of constrained atoms, ρ_α a renaming function for $\mathcal{A}$ based on α and P' the partial deduction of P wrt $\mathcal{A}$, $\hat{P}$ and ρ_α. If $\hat{P} \cup \{G\}$ is $\mathcal{A}, \mathcal{D}$-covered then the following hold:*

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

In the remainder of Subsection 4.3 we will prove this theorem in two successive stages.

1. First we will restrict ourselves to *unconstrained atoms*, i.e. constrained atoms of the form $true \square A$. This will allow us to reuse the correctness results for standard partial deduction with renaming in a rather straightforward manner.

2. We will then move on to general constrained atoms. Partial deductions of such constrained atoms can basically be obtained from partial deductions of unconstrained atoms by removing certain clauses (this a direct corollary of Lemma 4.4). We will show that these clauses can be safely removed without affecting the computed answers nor the finite failure.

The reader not interested in the details of the proof can immediately jump to Section 5.

4.3.1 Correctness for Unconstrained Atoms

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

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

Theorem 4.15 *Let P be a definite program, G a definite goal, $\mathcal{A}$ a finite set of unconstrained atoms, ρ_α a renaming function for $\mathcal{A}$ based on α and P' the partial deduction of P wrt $\mathcal{A}$, $\hat{P}$ and ρ_α. If $\hat{P} \cup \{G\}$ is $\mathcal{A},\mathcal{D}$-covered then the following hold:*

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

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

- P' corresponds to partial deduction with dynamic renaming and post-processing renaming for the set of atoms $\mathbf{A} = \{A \mid \mathit{true} \square A \in \mathcal{A}\}$.
- $P' \cup \{\rho_\alpha(G)\}$ is $\mathbf{A}$-covered because $\hat{P} \cup \{G\}$ is $\mathcal{A},\mathcal{D}$-covered (and because the original program P is unreachable in the predicate dependency graph from within P' or within $\rho_\alpha(G)$).

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

- Theorem 3.5 of [2] requires that no renaming be performed on G, i.e. $\rho_\alpha(G)$ must be equal to G. However, without loss of generality, we can assume that the top-level query is the unrenamed atom $new(X_1,\ldots,X_k)$, where new is a fresh predicate symbol and $vars(G) = \{X_1,\ldots,X_k\}$. We just have to add the clause $new(X_1,\ldots,X_k) \leftarrow Q$, where $G = \leftarrow Q$, to the initial program. Trivially the query $\leftarrow new(X_1,\ldots,X_k)$ and G are equivalent wrt c.a.s. and finite failure (see also Lemma 2.2 in [19]).
- Theorem 4.11 of [2] requires that G contains no variables or predicates in $\mathbf{A}$. The requirement about the variables is not necessary in our case because we do not base our renaming on the *mgu*. The requirement about the predicates is another way of ensuring that $\rho_\alpha(G)$ must be equal to G, which can be circumvented in a similar way as for the first point above.

- Theorems 3.5 and 4.11 of [2] require that the predicates of the renaming do not occur in the original P. Our Definition 4.10 does not require this. This is of no importance as the original program is always "completely thrown away" in our approach. We can still apply these theorems by using an intermediate renaming ρ' which satisfies the requirements of Theorems 3.5 and 4.11 of [2] and then applying an additional one step post-processing renaming ρ'', with $\rho_\alpha = \rho'\rho''$, along with an extra application of Theorem 4.11 of [2].

□

4.3.2 Correctness for Constrained Atoms

Lemma 4.16 *Let $c \square A$ be a constrained atom. Let θ $sat_{\mathcal{D}}$ c and let $c\sigma$ be unsatisfiable. Then $A\theta$ and $A\sigma$ have no common instance.*

Proof Suppose that $A\theta$ and $A\sigma$ have a common instance $A\theta\gamma = A\sigma\rho$. But this means that $\theta\gamma$ $sat_{\mathcal{D}}$ c, i.e. $holds_{\mathcal{D}}(c\theta\gamma)$ while $c\sigma\rho$ is unsatisfiable. Hence we have a contradiction because $c\sigma\rho$ is identical (up to renaming of the variables used for the quantifiers) to $c\theta\gamma$ (because $vars(c) \subseteq vars(A)$, i.e. $\theta\gamma\,|_{dom(c)} = \sigma\rho\,|_{dom(c)}$). □

Lemma 4.17 *Let P be a definite program and $c \square A$ and $true \square A$ be constrained atoms and let $A' \in valid_{\mathcal{D}}(c \square A)$ be an ordinary atom. Also let δ be a $CLP_{=}(\mathcal{D})$-derivation of $P \cup \{\leftarrow true \square A\}$ with characteristic path $p \notin chpaths_{\mathcal{D}}(P, c \square A)$ and whose computed answer is θ (and whose CLP-resultant is $A\theta \leftarrow true \square Bdy$). Then A' and $A\theta$ have no common instance.*

Proof First, $A' \in valid_{\mathcal{D}}(c \square A)$ is by definition equivalent to $A' = A\gamma$ and γ $sat_{\mathcal{D}}$ c, i.e. $holds_{\mathcal{D}}(c\gamma)$. Because $p \notin chpaths_{\mathcal{D}}(P, c \square A)$ we know that $c\theta$ is not $\mathcal{D}$-satisfiable. By Lemma 4.16, this means that $A\theta$ and $A\gamma$ have no common instance. □

The above shows that it is correct, for valid instances, to remove the resultants pruned by the constraints. Now, we just need to establish that every selected literal in the partial deduction is a valid instance of an element in $\mathcal{A}$.

For this we first need the following lemma.

Lemma 4.18 *Let $c \square A$ be a constrained atom. Then $true \square A \preceq_{\mathcal{D}} c \square A$ iff $A \in valid_{\mathcal{D}}(c \square A)$.*

Proof As anything satisfies *true* we have that $valid_{\mathcal{D}}(true \square A)$ consists of all instances of A, and notably $A \in valid_{\mathcal{D}}(true \square A)$. Therefore if $true \square A$ is an instance of $c \square A$ we have, by Definition 4.7, that $A \in valid_{\mathcal{D}}(c \square A)$. In the other direction, if $A \in valid_{\mathcal{D}}(c \square A)$ we have, by downwards-closedness, that all instances of A are also in $valid_{\mathcal{D}}(c \square A)$, and therefore Definition 4.7 is satisfied because $valid_{\mathcal{D}}(true \square A) \subseteq valid_{\mathcal{D}}(c \square A)$. □

The main correctness result for partial deduction with constrained atoms can now be proven as follows:

Proof of Theorem 4.14

1. In a first part of the proof we will reuse Theorem 4.15. To that end we have to relate P' to a covered partial deduction of unconstrained atoms.

First, as a direct corollary of Lemma 4.4, we know that P' is a subset of a partial deduction wrt unconstrained atoms, namely wrt the multiset[10] $\mathcal{A}' = \{true \ \square \ A \mid c \ \square \ A \in \mathcal{A}\}$, and using an atomic renaming α' such that $\alpha'(true \ \square \ A) = \alpha(c \ \square \ A)$ and a renaming function $\rho_{\alpha'}$ such that $\rho_{\alpha'}(G) = \rho_\alpha(G)$ for any ordinary or CLP-goal G for which $\rho_\alpha(G)$ is defined. (As $\mathcal{A}'$ contains more general constrained atoms than $\mathcal{A}$, whenever ρ_α is defined $\rho_{\alpha'}$ can also be defined.)
Let $PrCl$ denote the clauses pruned by the constraints, i.e. $P' \cup PrCl$ is the above mentioned partial deduction wrt $\mathcal{A}'$. Unfortunately, although G remains $\mathcal{A}', \mathcal{D}$-covered (as $\mathcal{A}'$ contains more general constrained atoms than $\mathcal{A}$), P is not necessarily $\mathcal{A}', \mathcal{D}$-covered. The reason is that new uncovered body atoms can arise inside $PrCl$. Let $\mathcal{U}$ be these uncovered atoms. To arrive at a covered partial deduction we simply have to add, for every predicate symbol p of arity n occurring in $\mathcal{U}$, the unconstrained atom $true \ \square \ p(X_1, \ldots, X_n)$ to $\mathcal{A}'$, where $X_1, \ldots, X_n$ are distinct variables. This will give us the new set $\mathcal{A}'' \supseteq \mathcal{A}'$. We will unfold the new unconstrained atoms $true \ \square \ p(X_1, \ldots, X_n)$ once (and keep the same unfolding for the elements in $\mathcal{A}'$) in order to obtain the set of resultants $\hat{P}'$. Let P'' be the partial deduction of P wrt $\mathcal{A}''$, $\hat{P}'$ and $\rho_{\alpha''}$, where $\rho_{\alpha''}$ is extended from $\rho_{\alpha'}$ in an arbitrary way for the new unconstrained atoms. Now $\hat{P}' \cup \{G\}$ is trivially $\mathcal{A}', \mathcal{D}$-covered. We can therefore apply the correctness Theorem 4.15 to deduce that the computations for $P'' \cup \{\rho_\alpha(G)\}$ (as $\rho_\alpha(G) = \rho_{\alpha'}(G) = \rho_{\alpha''}(G)$) are totally correct wrt the computations in $P \cup \{G\}$.
Note that, by construction, we have that $P' \subseteq P''$, and thus soundness of the computed answers (point 1, only-if part) and completeness of finite failure within P' (point 2, if part) are already established.

2. In the second part of the proof we will show that by removing the clauses $P_{New} = P'' \setminus P'$ we do not lose any computed answer nor do we remove any infinite failure. In other words any complete SLD-derivation for $P'' \cup \{\rho_\alpha(G)\}$ which uses a clause in $P'' \setminus P'$ fails finitely. This is sufficient to establish that P' is also totally correct.
Let D be an SLD-derivation for $P'' \cup \{\rho_\alpha(G)\}$ which uses at least one clause in $P'' \setminus P'$. Let D' be the largest prefix SLD-derivation of D which uses only clauses within P'. Let RG' be the last goal of D' and let $C'' \in P'' \setminus P'$ be the clause used in the last resolution step. First note that this clause C'' must be a clause for an unconstrained atom in $\mathcal{A}'$ (and not in $\mathcal{A}'' \setminus \mathcal{A}'$) because in D' we only used clauses from P' and because the predicates in $\mathcal{A}'' \setminus \mathcal{A}'$ are not reachable in the predicate dependency graph from within $P' \cup \{\rho_\alpha(G)\}$.

$$
\begin{array}{lll}
 & \rho_\alpha(G) & \\
 & \downarrow & C_1 \in P' \\
D' & \ldots & \\
 & \downarrow & C_n \in P' \\
RG' = & \leftarrow \ldots \underline{B'} \ldots & \\
 & \downarrow & C'' \in P'' \setminus P'
\end{array}
$$

Now let B' be the selected literal within RG'. We will show that resolution of B' with the clause C'' fails. To that end we will use Lemma 4.17. However, this lemma talks about

[10] Indeed, the same atom could in principle occur in several constrained atoms. This is not a problem, however, as the results in [2] carry over to multisets of atoms. Alternatively one could add an extra unused argument to P', $\rho_\alpha(G)$ and $\mathcal{A}'$ and then place different variables in that new position to transform the multiset $\mathcal{A}'$ into an ordinary set.

unification of an *unrenamed* atom B with the head of an unrenamed clause $\hat{C}'' \in \hat{P}$. Also, to be able to apply the lemma we need to have that $B \in valid_{\mathcal{D}}(c \square A)$ for some $c \square A \in \mathcal{A}$.

- For the top-level goal $\rho_\alpha(G)$ we know that it is a renaming of G such that $\rho_\alpha(G)$ is *defined*. Therefore, by definition of a renaming, we can deduce that each atom in G, notably the selected one, is a valid $\mathcal{D}$-instance of some $c \square A \in \mathcal{A}$. So if D' is the **empty** derivation, then we can directly apply Lemma 4.17 to deduce that resolution of B with the unrenamed version of C'' fails and, because renamings preserve non-unifiability, we can establish that resolution also fails for B' and C''.
- If RG' is reached after a **non-empty** derivation D', Lemma 4.17 can be applied if we are able to prove that RG' is a renaming of some goal RG using the atomic renaming α.
  An obvious candidate is the renaming function ρ_α. In general, however, RG' will only be obtainable from a goal RG through some renaming ρ'_α, not necessarily equal to ρ_α (but based on the same α). In Appendix C we illustrate this point with an example. We also prove in Lemma C.2 of Appendix C that such a renaming ρ'_α can always be constructed (given that $\hat{P} \cup \{G\}$ is $\mathcal{A}, \mathcal{D}$-covered). Indeed, Lemma C.2 states the following:

  > Let D' be a finite SLD-derivation for $P' \cup \{G'\}$ leading to the resolvent RG'. If $\hat{P} \cup \{G\}$ is $\mathcal{A}, \mathcal{D}$-covered and $G' = \rho_\alpha(G)$ then there exists an ordinary goal RG and a renaming function ρ'_α (also based on α) such that $RG' = \rho'_\alpha(RG)$ and such that RG is $\mathcal{A}, \mathcal{D}$-covered.

  We now round up the proof for the case that D' is not empty. By Definition 4.11 we know that C'' is of the form $\alpha''(true \square A)\theta \leftarrow \rho_{\alpha''}(Bdy)$ for $true \square A \in \mathcal{A}'$. By definition of α'', we can rewrite this into $\alpha(c \square A)\theta \leftarrow \rho_{\alpha''}(Bdy)$ for $c \square A \in \mathcal{A}$. Hence the selected literal B' in RG' must have the same predicate as $\alpha(c \square A)$. We can also apply Lemma C.2 (for $P' \cup \{\rho_\alpha(G)\}$ leading to RG') to deduce that $RG' = \rho'_\alpha(RG)$ for some ordinary goal RG and renaming function ρ'_α based on α (for $\mathcal{A}$) . Hence $B' = \rho'_\alpha(B)$ and also $B \in valid_{\mathcal{D}}(c \square A)$ (as B' has the same predicate as $\alpha(c \square A)$). Let $\hat{C}''$ be the unrenamed version of C'', i.e. $\hat{C}'' = H \leftarrow Bdy$. We can now apply Lemma 4.17 (because $B \in valid_{\mathcal{D}}(c \square A)$) to deduce that resolution of $\leftarrow B$ with $\hat{C}''$ fails (immediately). Finally, because renaming preserves non-unifiability (i.e. if t_1 and t_2 do not unify then neither do their renamings) we can deduce that resolution also fails for $\leftarrow B'$ with C'', and therefore the derivation D fails finitely in P''.

  □

We will illustrate this theorem through several examples later in Section 5.3.

5 Preserving Characteristic Trees

Based on the more expressive and powerful framework for constrained partial deduction, we now present a precise abstraction operator, preserving characteristic trees upon generalisation, as well as a terminating algorithm for constrained partial deductions satisfying the criteria of Theorem 4.14.

In order to formulate our approach we have to fix the particular constraint structure to be used — this was still left generic in the previous section. In fact, all we need in order to be able to preserve characteristic trees upon generalisation, is Clark's equality theory (CET). More precisely, we will use the structure $\mathcal{D} = \mathcal{FT}$ consisting of CET over the domain

of *finite trees* (with infinitely many functors of every arity[11]) including all functors in the programs and queries under consideration. So it is basically the same structure as the one used for CLP($\mathcal{FT}$), as defined e.g. in [62].[12] The same theory has also been used, for different purposes, in the constructive negation techniques (e.g. [7, 8, 15, 59, 66, 65]). Note that CET is a complete theory [30] and we suppose that we have the required algorithms for satisfiability checking, simplification and projection at our disposal (see e.g. [66, 65, 60]).

5.1 Pruning Constraints

As we have already seen in Section 3, when taking the *msg* of two atoms A and B with the same characteristic tree τ, we do not necessarily obtain an atom C which has the same characteristic tree. The basic idea is now quite simple. Instead of C, we will generate $c \square C$ as the generalisation of A and B, where the constraint c is designed in such a way as to prune the possible computations of C into the right shape, namely τ. Indeed, all the derivations that were possible for A and B are also possible for C (because we only consider definite programs and goals) and c only has to ensure that the additional matches wrt τ are pruned off at some point (cf. Figure 4).

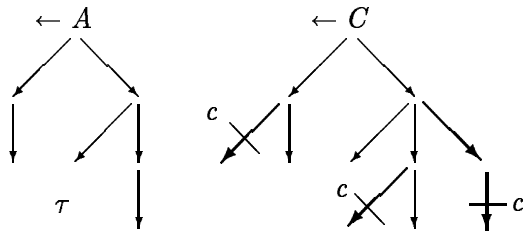

Figure 4: Pruning Constraints

Now, there are possibly infinitely many ways in which these additional matches can be pruned via c for C; one can e.g. vary the depth at which pruning occurs. Also some of these matches might have also be possible for A or B, but the unfolding rule has then constructed a finitely failed subtree for the corresponding resolvent. Again there are possibly infinitely many ways in which this can happen. However, for our constrained partial deduction algorithm, to be presented later, it is important to come up, for any given atom C and characteristic tree τ, with a *finite* constraint covering *all* instances of C which have τ as their characteristic tree. If we allow any unfolding rule then this most specific constraint will often be an infinite disjunction, and as such is not expressible in CET. In order to remedy this problem we will first restrict ourselves to a certain class of unfolding rules in which failure occurs only in a special way (we will show how this restriction can be lifted by e.g. incorporating failed branches into the characteristic trees later on). Because there is only one way in which failure can occur, it is possible to calculate a finite constraint c satisfying the above.

[11] For a detailed study of the relation between the underlying language and equality theory we refer the reader to [59].

[12] We will actually restrict ourselves to a subset of CLP($\mathcal{FT}$) in which satisfiability can be decided by simple matching. See [37] for further details.

In fact, purely determinate unfolding rules have the property that, if there is a failing branch, then the goal fails completely and the goal has an empty characteristic tree. So either there are no failed branches or the characteristic tree is empty. It turns out that this is exactly the property that we need. Indeed, goals with empty characteristic trees do not pose any problem for termination of any partial deduction algorithm, because the partial deductions of the goals are empty and no atoms in the bodies have to be added to the set of atoms to be partially deduced. An abstraction operator can thus leave atoms with empty characteristic trees untouched and for the others it knows that there are no failing branches at all.

Definition 5.1 (failure preserving unfolding rule) *An unfolding rule is said to be* failure preserving *iff for every CLP-goal it returns an incomplete $CLP_{=}(\mathcal{D})$-tree τ such that τ^{-} is either equal to τ or equal to $\emptyset$, where τ^{-} is obtained from τ by removing the failed branches (so either all the branches are failed*[13] *or none are).*

Proposition 5.2 *Any purely determinate unfolding rule is failure preserving.*

Proof Straightforward, by induction on the length of the generated incomplete SLDNF-tree. □

The class of failure preserving unfolding rules is larger than the one of purely determinate unfolding rules, albeit only slightly so (e.g. a determinate unfolding rule *with* a lookahead is not failure preserving).

We will now formalise a general method to calculate constraints ensuring the preservation of characteristic trees. For that it will be useful to denote by $mgu^*(A, B)$ a particular idempotent and relevant mgu of $\{A, B'\}$, where B' is obtained from B by renaming apart (wrt A). The mgu^* has the following interesting property:

Proposition 5.3 *Let A, B be two terms. Then $mgu^*(A, B) = fail$ iff A and B have no common instance.*

Proof $\Leftarrow$: Suppose $mgu^*(A, B) = \theta \neq fail$. This means that $A\theta = B\gamma\theta$ for some γ and A and B have a common instance and we have a contradiction.
$\Rightarrow$: Suppose that A and B have the common instance $A\theta = B\sigma$ and let γ be the renaming substitution for B used by mgu^*. This means that for some γ^{-1} we have $B\gamma\gamma^{-1} = B$ and $B\gamma\gamma^{-1}\sigma = A\theta$. Now as the variables of $B\gamma$ and A are disjoint the set of bindings $\theta^* = \theta \mid_{vars(A)} \cup (\gamma^{-1}\sigma) \mid_{vars(B\gamma)}$ is a well defined substitution and a unifier of A and $B\gamma$, i.e. $mgu^*(A, B) \neq fail$ and we have a contradiction. □

The following proposition characterises gives a condition which ensures that a particular characteristic path is pruned. We will later transform this condition into a constraint expressed using CET.

Proposition 5.4 *Let G be a definite goal, γ a substitution, P a definite program and let δ be an SLD-derivation for $P \cup \{G\}$ with computed answer θ and characteristic path p. Then $mgu^*(G\gamma, G\theta) = fail$ iff $p \notin chpaths(P, G\gamma)$.*

[13] Note that when a selected literal does not unify with a particular clause then this does *not* correspond to a failed branch.

Proof $\Leftarrow$: Is a direct consequence of Lemma 4.11 a (for atomic goals) and Lemma 4.11 b (for general goals) in [43] ($G\theta$ can be seen as the head of a resultant which is constructed from a derivation whose characteristic path is p).
$\Rightarrow$: Suppose that $p \in chpaths(P, G\gamma)$. Let δ' be a derivation for $P \cup \{G\gamma\}$ with computed answer θ' and whose characteristic path is p. We have that $G\theta$ is the head of the resultant R of the derivation δ for $P \cup \{G\}$. By Lemma 4.9 of [43] we can deduce that, because $G\gamma$ is an instance of G, the resultant R' of δ' is in turn an instance of R. Hence we know that the head $G\gamma\theta'$ of R' is also an instance of $G\theta$. Hence $G\gamma$ and $G\theta$ have a common instance and by Proposition 5.3 we can finally conclude that $mgu^*(G\gamma, G\theta) \neq fail$. □

Below, we denote finite sequences of elements (in particular, characteristic paths or subsequences thereof) by p, q and r, possibly adorned with subscripts. For two such sequences, p and q, we denote by pq the concatenation of p and q. In such a concatenation, we will allow p or q or both to denote an empty sequence of elements, in which case pq denotes p (or q or ϵ).

The following definition will turn out to be useful (and is illustrated in Figure 5).

Definition 5.5 (simple extension path) *Let P be a normal program. Let τ be a non-empty $\mathcal{D}$-characteristic tree for some CLP-goal CG in P and let p be a characteristic path. Then p is a* simple extension path *of τ iff*

1. *$\forall q$ we have that $pq \notin \tau$ and*
2. *$\exists r = r_1(lit, cl)r_2 \in \tau$, such that $p = r_1(lit, ncl)$ (where cl, ncl are numbers of clauses belonging to a same predicate definition in P).*[14]

We denote the set of simple extension paths of τ in P by $extpaths_P(\tau)$.

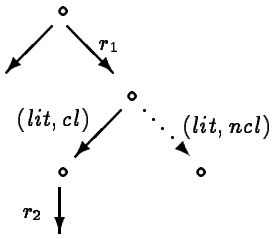

Figure 5: Illustrating Definition 5.5 (Simple Extension Paths)

Each simple extension path of a characteristic tree captures a potential new clause match. The following proposition captures the interesting aspect of (pruning) simple extension paths in the context of preserving characteristic trees.

Proposition 5.6 *Let U be an unfolding rule, P a definite program, CG a CLP-goal and τ a characteristic tree such that $\tau \subseteq chpaths_{\mathcal{D}}(P, CG)$. If $extpaths_P(\tau) \cap chpaths(P, CG) = \emptyset$ then τ is a $\mathcal{D}$-characteristic tree of CG in P.*

[14] Note that necessarily $ncl \neq cl$.

Proof Straightforward, by induction on the depth of τ. □

Example 5.7 We recall the program P from Example 3.8:

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

For $\tau = \{\langle 1 \circ 2 \rangle\}$ we have $extpaths_P(\tau) = \{\langle 1 \circ 1 \rangle\}$. We also have $\tau \subseteq chpaths_{\mathcal{FT}}(P, true \square \leftarrow p(X))$ and $\tau \subseteq chpaths_{\mathcal{FT}}(P, \leftarrow \neg(X = c) \square p(X))$. However, τ is not a $\mathcal{FT}$-characteristic tree for $\leftarrow true \square p(X)$ in P but is a $\mathcal{FT}$-characteristic tree for $\leftarrow \neg(X = c) \square p(X)$ in P. And indeed $extpaths_P(\tau) \cap chpaths_{\mathcal{FT}}(P, \leftarrow true \square p(X)) \neq \emptyset$ while $extpaths_P(\tau) \cap chpaths_{\mathcal{FT}}(P, \leftarrow \neg(X = c) \square p(X)) = \emptyset$.

In the following definition we calculate constraints which prune simple extension paths and hence ensure that the condition $extpaths_P(\tau) \cap chpaths_{\mathcal{D}}(P, CG) = \emptyset$ of Proposition 5.6 holds. This is a big step towards preserving characteristic trees. In order to simplify the presentation, will take the liberty to treat a conjunction of constraints like a set of constraints and introduce the following notation $cas_P(G, p)$ to be the computed answer of an SLD-derivation for $P \cup \{G\}$ with characteristic path p (if such a derivation exists, otherwise we leave $cas_P(G, p)$ undefined). Also, given two atoms A and B, we denote by $A = B$ the constraint $false$ if A and B have a different arity or predicate symbol, and $a_1 = b_1 \wedge \ldots \wedge a_k = b_k$ otherwise, where $A = p(a_1, \ldots, a_k)$ and $B = p(b_1, \ldots, b_k)$.

Definition 5.8 (pruning constraint) *Let P be a definite program, τ a non-empty characteristic tree and let A be an atom. For two atoms A_1, A_2 with the same predicate symbol p, the expression $A_1 \not\sim A_2$ denotes the constraint $\tilde{\forall}_{vars(A_1)}(\neg(A_1 = A_2'))$, where A_2' has been obtained from A_2 by standardising apart (wrt A_1).*
Then we define the pruning constraint *for A wrt τ (and P) by:*
$prune_P(A, \tau) = \{A \not\sim A\theta \mid p \in chpaths(P, \leftarrow A) \cap extpaths_P(\tau)$ *and* $\theta = cas_P(G, p)\}$.

A constrained atom of the form $prune_P(A, \tau) \square A$ will be called *normalised.*

Suppose that we have two constrained atoms $c_a \square A$, $c_b \square B$ both with $\mathcal{FT}$-characteristic tree τ. Using the newly introduced concepts, the obvious generalisation is the normalised constrained atom $c \square C$, where $C = msg(\{A, B\})$ and $c = prune_P(C, \tau)$. If $\tau \subseteq chpaths_{\mathcal{FT}}(P, \leftarrow c \square C)$, then we can apply Proposition 5.6 and we have achieved preservation of characteristic trees. All that remains is then to prove that $c \square C$ is indeed more general than both $c_a \square A$ and $c_b \square B$.

However, neither of these conditions is satisfied in general if we use an arbitrary unfolding rule (to construct τ for $c_a \square A$ and $c_b \square B$). In fact, the pruning constraints $prune_P(C, \tau)$ prune off each simple extension path immediately "at the source". But a simple extension path p of τ can in general be a valid path for $\leftarrow c_a \square A$ (i.e. $p \in chpaths_{\mathcal{FT}}(P, \leftarrow c_a \square A)$) and only lead to failure after further unfolding. In that case $c_a \square A$ is most likely not an $\mathcal{FT}$-instance of $c \square C$. The following example illustrates this.

Example 5.9 Let P be this program:

(1) $p(X) \leftarrow q(X)$
(2) $p(X) \leftarrow r(X)$
(3) $q(s(X)) \leftarrow q(X)$
(4) $r(X) \leftarrow$

Let the negative constraint $c_a = c_b = \emptyset$ and let $A = p(0)$, $B = p(s(0))$, $C = p(Z)$. Let the unfolding rule U be such that $chtree_{\mathcal{FT}}(\leftarrow c_a \square A, P, U) = chtree_{\mathcal{FT}}(\leftarrow c_b \square B, P, U) = \tau = \{\langle 1 \circ 2, 1 \circ 4 \rangle\}$. Then $c = prune_P(p(Z), \tau) = \forall X.\neg(p(Z) = p(X))$.
This constraint is unsatisfiable and hence $c \square C$ is not more general than either $true \square p(0)$ or $true \square p(s(0))$. Furthermore, $\tau \not\subseteq chpaths_{\mathcal{FT}}(P, \leftarrow c \square C)$ and τ is not a $\mathcal{FT}$-characteristic tree of $\leftarrow c \square C$.

Instead of restricting ourselves to failure preserving unfolding rules, we could also make the constraint $c = prune_P(C, \tau)$ more general in order to cover each possible failing behaviour. In general, however, there are infinitely many different ways in which a branch of an SLD-tree can fail. In that case, the most specific constraint ensuring that $c \square C$ covers *all* constrained atoms $c_a \square C\gamma$ with $\mathcal{FT}$-characteristic tree τ would have to consist of an infinite disjunction. For instance in the Example 5.9 above, the constraint would have to look like $\hat{c} = \forall X.\neg(p(Z) = p(s(X))) \vee \forall X.\neg(p(Z) = p(s(s(X)))) \vee \ldots$. This idea is very related to the work in [45] which attempts to construct maximally general fail substitutions for negation as failure. Indeed for every resolvent goal G of a simple extension path we can attempt to construct a maximally general fail constraint ensuring that G fails. This would allow us to handle any unfolding rule and preserve characteristic trees in the framework of constrained partial deduction of Section 4. However, $\hat{c} \square C$ does not necessarily have τ as its $\mathcal{FT}$-characteristic tree (according to Definition 4.1 of a CLP$_=(\mathcal{D})$-derivation step), although all the atoms in $valid_{\mathcal{FT}}(\hat{c} \square C)$ do. So one would have to extend Definition 4.1 to allow more powerful pruning possibilities (allowing, in some cases, to detect an infinitely failed subtree). Whether this can be done in a practical way is matter for future research.

Another solution is to consider the failed branches to be part of a characteristic tree and then prune off simple extension paths of this more detailed structure. The method presented in the remainder of this section can in fact be easily adapted in that direction, thus lifting the restriction on unfolding rules. A post-processing phase could be devised, e.g. based on techniques in [56, 40], to then remove the unnecessary (cf. Section 3) polyvariance generated by such an approach.

For failure preserving unfolding rules we can always find, for any given atom C and characteristic tree τ, a *finite*, *most general*[15] constraint (namely the pruning constraint) c such that $c \square C$ covers all constrained atoms $c_a \square C\gamma$ with $\mathcal{FT}$-characteristic tree τ (this is a corollary of point 1 of Theorem 5.11 proven below). This property in turn, will ensure that the condition $\tau \subseteq chpaths_{\mathcal{FT}}(P, \leftarrow prune_P(C, \tau) \square C)$ always holds (cf. point 2 of Theorem 5.11 below) and guarantee that characteristic trees are preserved (cf. point 3 of Theorem 5.11 below).

Before proving the preservation of characteristic trees for failure preserving unfolding rules we need one further lemma.

Lemma 5.10 *Let A_1, A_2 be two atoms. If $mgu^*(A_1\gamma, A_2) = fail$ then γ $sat_{\mathcal{FT}}$ $A_1 \not\sim A_2$.*

Proof By definition $A_1 \not\sim A_2$ stands for $\tilde{\forall}_{vars(A_1)}(\neg(A_1 = A_2'))$ where A_2' has been obtained by standardising apart (wrt A_1). It is well known (see e.g. [9] or Lemma 15.2 in [42]) that if B and C are not unifiable then CET$\models \tilde{\forall}(\neg(B = C))$. Now by definition of applying substitutions we have $(A_1 \not\sim A_2)\gamma = \tilde{\forall}_{vars(A_1\gamma)}(\neg(A_1\gamma = A_2'))$. Finally, $mgu^*(A_1\gamma, A_2) = fail$ means that

[15] This property is useful to show that the abstraction operator cannot generate an infinite sequence of generalisations.

$A_1\gamma$ and A'_2 are not unifiable, hence CET$\models \tilde{\forall}(\neg(A_1\gamma = A'_2))$ and therefore $\gamma\ sat_{\mathcal{FT}}\ A_1 \not\sim A_2$ (because $\tilde{\forall}(\tilde{\forall}_{vars(A_1\gamma)}(\neg(A_1\gamma = A'_2)))$ is equivalent to $\tilde{\forall}(\neg(A_1\gamma = A'_2))$ and CET is part of our constraint structure $\mathcal{FT}$). □

Theorem 5.11 (Preservation of characteristic trees) *Let P be a definite program, $c \square A$ a constrained atom and let B be an ordinary atom more general than A. Also let $\tau = chtree_{\mathcal{FT}}(\leftarrow c \square A, P, U)$ be a non-empty characteristic tree for a failure preserving unfolding rule U. Then:*

1. *$c \square A$ is a $\mathcal{FT}$-instance of $prune_P(B,\tau) \square B$.*
2. *$\tau \subseteq chpaths_{\mathcal{FT}}(P, \leftarrow prune_P(B,\tau) \square B)$.*
3. *τ is a $\mathcal{FT}$-characteristic tree of $\leftarrow prune_P(B,\tau) \square B$*

Proof

1. Because B is more general than A we have for some substitution γ: $A = B\gamma$. We have to prove that whenever $\theta\ sat_{\mathcal{FT}}\ c$, then also $\gamma\theta\ sat_{\mathcal{FT}}\ prune_P(B,\tau)$.
Let us examine every constraint $n = B \not\sim B\rho \in prune_P(B,\tau)$ (cf. Definition 5.8) and let $p \in chpaths(P, \leftarrow B)$ be the corresponding simple extension path in τ with $\rho = cas_P(\leftarrow B, p)$. Either $p \notin chpaths(P, \leftarrow A)$. In that case we can apply Proposition 5.4 (with $G = \leftarrow B$) and deduce that $mgu^*(B\gamma, B\rho) = fail$. Therefore, we have by Lemma 5.10 that $\gamma\ sat_{\mathcal{FT}}\ n$ which is by definition equivalent to $\mathcal{FT} \models \tilde{\forall}(n\gamma)$, and hence we also have that $\gamma\theta\ sat_{\mathcal{FT}}\ n$.
Or, $p \in chpaths(P, \leftarrow A)$. In that case, because τ is not empty and U is failure preserving (and hence no failing branches are possible) and because no extension of p is in τ, $c\sigma$ must be unsatisfiable, where $\sigma = cas_P(\leftarrow A, p)$. We have by Lemma 4.16 that $A\theta = B\gamma\theta$ has no common instance with $A\sigma = B\gamma\sigma$. By Proposition 5.3 this is equivalent to saying that $mgu^*(B\gamma\theta, B\gamma\sigma) = fail$. We can now use Proposition 5.4 (for $G = \leftarrow B\gamma$) to deduce that $p \notin chpaths(P, \leftarrow B\gamma\theta)$. Finally, by reusing Proposition 5.4 in the other direction (for $G = \leftarrow B$) we can deduce that $mgu^*(B\gamma\theta, B\rho) = fail$ and we can conclude by Lemma 5.10 that $\gamma\theta\ sat_{\mathcal{FT}}\ n$.
Hence, as $\gamma\theta$ satisfies every $n \in prune_P(B,\tau)$, we can deduce that $\gamma\theta\ sat_{\mathcal{FT}}\ prune_P(B,\tau)$.

2. By the (correct version of the) lifting lemma [27, 43, 14, 1] we can deduce that $\tau \subseteq chpaths(P, \leftarrow B)$. Let us examine every $p \in \tau$. When we take $\theta = cas_P(\leftarrow A, p)$ we know that $c\theta$ is satisfiable in $\mathcal{FT}$ (because $p \in chtree_{\mathcal{FT}}(\leftarrow c \square A, P, U)$). So, for some substitution γ we have that $\theta\gamma\ sat_{\mathcal{FT}}\ c$ and by point 1 of this proposition we can find a substitution σ such $\sigma\ sat_{\mathcal{FT}}\ prune_P(B,\tau)$ and $A\theta\gamma = B\sigma$. Again by the lifting lemma we can deduce that there exists a ρ such that $\rho = cas_P(\leftarrow B, p)$ and such that $A\theta$ is an instance of $B\rho$. This means that $prune_P(B,\tau)\rho$ must be satisfiable (for γ' such that $B\rho\gamma' = A\theta\gamma = B\sigma$ because $vars(prune_P(B,\tau)) \subseteq vars(B)$ and $holds_{\mathcal{FT}}(prune_P(B,\tau)\sigma)$) and hence we can conclude that $prune_P(B,\tau)$ has not pruned p, i.e. $p \in chpaths_{\mathcal{FT}}(P, \leftarrow prune_P(B,\tau) \square B)$. So we can conclude that $\tau \subseteq chpaths_{\mathcal{FT}}(P, \leftarrow prune_P(B,\tau) \square B)$.

3. By Proposition 5.6 and point 2 we only have to prove that:
$extpaths_P(\tau) \cap chpaths_{\mathcal{FT}}(P, \leftarrow prune_P(B,\tau) \square B) = \emptyset$. This is, however, a direct consequence of Definition 5.8 and the fact that $B\theta \not\sim B\theta$ is unsatisfiable. □

Note that if we take the Example 5.9, with its unfolding rule which is not failure preserving, then none of the points of Theorem 5.11 hold.

5.2 A Precise Abstraction Operator

We are now in a position to formally define our abstraction operator.

Definition 5.12 *($chabsc_{P,U}$) Let P be a definite program, U a failure preserving unfolding rule and $\mathcal{A}$ a set of constrained atoms.*
For any characteristic tree τ, let $\mathcal{A}_\tau = \{A \mid c \square A \in \mathcal{A} \wedge chtree_{\mathcal{FT}}(\leftarrow c \square A, P, U) = \tau\}$.
Then the operator $chabsc_{P,U}$ is defined as follows:
$chabsc_{P,U}(S) = \mathcal{A}_\emptyset \cup \{prune_P(A,\tau) \square msg(\mathcal{A}_\tau) \mid, \tau \neq \emptyset$ *is a characteristic tree* $\}$.

The following proposition establishes that the operator $chabsc_{P,U}$ is an abstraction operator (in the spirit of Definition 2.8).

Proposition 5.13 *Let $\mathcal{A}$ be a finite set of constrained atoms. Then $\mathcal{A}' = chabsc_{P,U}(\mathcal{A})$ is a finite set of constrained atoms such that every constrained atom in $\mathcal{A}$ is an $\mathcal{FT}$-instance of a constrained atom in $\mathcal{A}'$.*

Proof Immediate corollary of Definition 5.12 and point 1 of Theorem 5.11. □

By point 3 of Theorem 5.11 we can deduce that this abstraction operator preserves the characteristic trees, i.e. after abstraction, τ remains a $\mathcal{FT}$-characteristic tree for the CLP-goal $\leftarrow prune_P(A,\tau) \square A$. However, we cannot conclude that τ is *the* $\mathcal{FT}$-characteristic tree of $\leftarrow prune_P(A,\tau) \square A$ for U, because nothing prevents U from treating that goal completely differently (i.e. selecting different atoms) from the goals in $\mathcal{A}_\tau$. Such an arbitrary behaviour does not cause problems for the constrained partial deduction method as such, except when it comes to termination of $chabsc_{P,U}$ (which will be proven below in Proposition 5.18). To avoid this kind of arbitrary behaviour we need a feature of "stability" of the unfolding rule for normalised constrained atoms (or use a monotonic partial deduction algorithm, cf. Section 3.2).

Definition 5.14 (stable unfolding rule) *An unfolding rule U is called* stable *iff for each atom A' with $chtree(\leftarrow A', P, U) = \tau \neq \emptyset$, and for each atom A more general than A', we have that $chtree_{\mathcal{FT}}(\leftarrow prune_P(A,\tau) \square A, P, U) = \tau$.*

For instance a (purely) determinate unfolding rule U with a static (e.g. left-to-right) selection of the determinate atoms will not arbitrarily change the unfolding behaviour and in that case we are able to conclude that $chtree_{\mathcal{FT}}(\leftarrow prune_P(A,\tau) \square A, P, U) = \tau$.

Proposition 5.15 *Any purely determinate unfolding rule with a static selection of the determinate atoms is stable.*

Proof Straightforward, by induction on the depth of τ, because the pruning constraints preserve determinacy as well as non-determinacy inside τ and because the unfolding rule (due to its staticness) will then select the same literal as in τ. □

Also note that in the case that an unfolding rule does not exhibit this stability we can easily ensure it by simply *imposing* τ as the $\mathcal{FT}$-characteristic tree of the generalisation $\leftarrow prune_P(A,\tau) \square A$. In a practical algorithm (e.g. the one implemented for the experiments in Section 6) this amounts to storing the characteristic tree τ with the normalised atoms.

Stability also has the added benefit that unfolding does not have to be re-done for the generalised atom, because the resulting characteristic tree is already known. Also see [34] which pushes the idea of imposing characteristic trees on the generalisation one step further.

We can now adapt Algorithm 2.9 by incorporating constrained atoms into the partial deduction process and by using the abstraction operator of Definition 5.12.

Algorithm 5.16 (constrained partial deduction) *Let P be a definite program, let U be a failure preserving unfolding rule. Given a constrained atom CA, we denote by $resultants_{P,U}(CA)$ the partial deduction of CA in P when using the unfolding rule U. Also, by $CBA(\hat{P})$ we denote the set of constrained body atoms of a set of CLP-clauses $\hat{P}$. The following defines a partial deduction algorithm which preserves characteristic trees by using pruning constraints.*

Input: *A program P and a goal G*

Output: *A specialised program P'*

Initialise: $i = 0$, $\mathcal{A}_0 = chabsc_{P,U}(\{true \square A \mid A \text{ is an atom in } G\})$

repeat

  let $\hat{P}_i = \bigcup_{CA \in \mathcal{A}_i} resultants_{P,U}(CA)$;

  let $\mathcal{A}_{i+1} = chabsc_{P,U}(\mathcal{A}_i \cup CBA(\hat{P}_i))$;

  $i := i + 1$;

until $\mathcal{A}_{i+1} = \mathcal{A}_i$

Construct a partial deduction P' of P wrt $\mathcal{A}_i$, $\hat{P}_i$ and some ρ_α.

The following establishes the correctness of the above algorithm.

Proposition 5.17 *If Algorithm 5.16 terminates, starting from the original program P and the goal G, it generates a partial deduction P' of P wrt $\mathcal{A}_i$, $\hat{P}_i$ and some ρ_α satisfying the requirements of Theorem 4.14 for any goal G' whose atoms are instances of atoms in G.*

Proof We just have to show that $\hat{P}_i \cup \{G'\}$ is $\mathcal{A}_i, \mathcal{D}$-covered, i.e. each constrained body atom of $\hat{P}$ or G' should be a $\mathcal{D}-$instance of an element in $\mathcal{A}_i$. This condition is clearly satisfied when reaching the fixpoint of Algorithm 5.16. Indeed, by Proposition 5.13, all constrained body atoms of $\hat{P}_i$ are $\mathcal{D}$-instances of elements in $\mathcal{A}_{i+1} = \mathcal{A}_i$. By the same proposition, all atoms in G are also $\mathcal{D}$-instances of elements in $\mathcal{A}_i$, because they were $\mathcal{D}$-instances of elements in $\mathcal{A}_0$. Finally, as the atoms in G' are instances of the atoms in G, we can conclude (by downwards-closedness) that $\hat{P}_i \cup \{G'\}$ is $\mathcal{A}_i, \mathcal{D}$-covered. □

We will now prove termination of the above algorithm for stable unfolding rules. In fact, given a (albeit unnatural) unstable unfolding rule, we can basically reconstruct the pattern of an example in [37] to obtain an oscillating behaviour of the partial deduction Algorithm 5.16.

Proposition 5.18 *If the set of different characteristic trees is finite and the unfolding rule U is stable, then Algorithm 5.16 terminates.*

Proof In Appendix D. □

To ensure a finite number of different characteristic trees, we can simply enforce a depth bound on the unfolding rule used during partial deduction, thereby also ensuring local termination. It is, however, also possible not to impose any ad-hoc depth-bound on the unfolding rule and to impose the depth-bound only on characteristic paths and trees. A third alternative is presented in [40], which gets rid of the depth bound altogether (see also the discussion in Section 7).

5.3 Some Examples

In this section we illustrate the workings and the interest of the abstraction operator $chabsc_{P,U}$ along with Algorithm 5.16 on some practical examples. First note that Algorithm 5.16 solves all the problematic examples in [36] as well as the problematic non-termination example in [37].

Example 5.19 Let us return to the *member* program and the problematic Example 3.11.

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

Let $G = \leftarrow A, B$ be the goal of interest, where $A = member(a,[b,c|T])$ and where $B = member(a,[c,d|T])$. We start the algorithm with $\mathcal{A}_0 = chabsc_{P,U}(\{true \square A, true \square B\})$. Both $true \square A$ and $true \square B$ have the same $\mathcal{FT}$-characteristic tree $\tau = \{\langle 1 \circ 2, 1 \circ 2, 1 \circ 1\rangle, \langle 1 \circ 2, 1 \circ 2, 1 \circ 2\rangle\}$ when using a purely determinate unfolding rule. Hence we calculate $C = msg(\{A,B\}) = member(a,[X,Y|T])$ as well as the pruning constraint $c = prune_P(C,\tau) =$
$\forall Y' \forall T'. \neg(member(a,[X,Y|T]) = member(a,[a,Y'|T'])) \wedge$
$\forall X' \forall T'. \neg(member(a,[X,Y|T]) = member(a,[X,a|T]))$ (calculated for the simple extension paths $\langle 1 \circ 1\rangle$ and $\langle 1 \circ 2, 1 \circ 1\rangle$ respectively). (Given a simplification procedure we could rewrite c into the equivalent constraint $\neg(X = a) \wedge \neg(Y = a)$.) We now get $\mathcal{A}_0 = \{c \square C\}$. Unfolding $c \square C$ using the same unfolding rule still results in the $\mathcal{FT}$-characteristic tree τ, and the precomputation and pruning that was possible for $true \square A$ and $true \square B$ has been preserved by $chabsc_{P,U}$! The only constrained body atom in the next step of the algorithm is $\tilde{\exists}_{\{T\}}(c) \square member(a,T)$ which can be simplified to $true \square member(a,T)$. The $\mathcal{FT}$-characteristic tree of $true \square member(a,T)$ is $\{\langle 1 \circ 1\rangle, \langle 1 \circ 2\rangle\}$. Thus $chabsc_{P,U}$ performs no generalisation and at the next step of the algorithm we reach the fixpoint $\mathcal{A}_2 = \mathcal{A}_1 = \{c \square member(a,[X,Y|T]), true \square member(a,T)\}$. We thus obtain the following partial deduction P' wrt $\mathcal{A}_1$ (using an appropriate atomic renaming α):

(1') $mem_a([X,Y|T]) \leftarrow mem_a(T)$
(2') $member_a([a|T]) \leftarrow$
(3') $member_a([Y|T]) \leftarrow member_a(T)$

Now, for example $G_1 = \leftarrow member(a,[b,c]), member(a,[c,d,e])$ is $\mathcal{A}_1, \mathcal{D}$-covered and, by Theorem 4.14, P' is correct for the renaming $G_1' = \leftarrow mem_a([b,c]), mem_a([c,d,e])$. However, although $G_2 = \leftarrow member(a,[b,a])$ is also $\mathcal{A}_1, \mathcal{D}$-covered, $G_2' = \leftarrow mem_a([b,a])$ is not a correct renaming of G_2 (because $member(a,[b,a]) \notin valid_{\mathcal{D}}(c \square member(a,[X,Y|T]))$ and we cannot apply Theorem 4.14. And indeed, $P' \cup \{G_2'\}$ fails while $P \cup \{G_2\}$ succeeds. We can, however, still rename G_2 into $G_2'' = \leftarrow member_a([b,a])$. Theorem 4.14 can then applied to deduce that using $P' \cup \{G_2''\}$ is correct.

Example 5.20 Let P be the well known "vanilla" solve meta-interpreter (see e.g. [23, 47, 48]).

(1) $solve(empty) \leftarrow$
(2) $solve(X\&Y) \leftarrow solve(X), solve(Y)$
(3) $solve(X) \leftarrow clause(X,B), solve(B)$
(4) $clause(p(a)) \leftarrow$
(5) $clause(p(b)) \leftarrow$
(6) $clause(q(a)) \leftarrow$
(7) $clause(q(b)) \leftarrow$

Let us suppose we use a purely determinate unfolding rule U which allows non-determinate steps only at the top. Also suppose that we want to specialise P for the goal $G = \leftarrow solve(p(X)), solve(q(X))$. The characteristic trees of both these atoms will be $\tau = \{\langle 1 \circ 3 \rangle\}$.

Applying the abstraction operator $chabs_{P,U}$ without constraints of Definition 3.6 (as well as the abstraction operators of [20, 17]) will give us as generalisation $chabs_{P,U}(S) = \{solve(X)\}$ where $solve(X)$ has the characteristic tree $\tau' = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2 \rangle, \langle 1 \circ 3 \rangle\}$ and local precision and specialisation has been lost due to the abstraction.

When using the abstraction operator $chabsc_{P,U}$ with constraints we obtain $chabsc_{P,U}(S) = \neg(solve(X) = solve(empty)) \wedge \forall Y \forall Z. \neg(solve(X) = solve(Y \& Z)) \square solve(X)$. The abstraction still has the $\mathcal{FT}$-characteristic tree $\tau = \{\langle 1 \circ 3 \rangle\}$ and all the specialisation has been preserved. Using Algorithm 5.16 we obtain the following partial deduction P' (using a suitable atomic renaming α; determinate post-unfolding can be used to get rid of $solve\_empty$).

(1') $solve(X) \leftarrow clause(X), solve\_empty$
(2') $clause(p(a)) \leftarrow$
(3') $clause(p(b)) \leftarrow$
(4') $clause(q(a)) \leftarrow$
(5') $clause(q(b)) \leftarrow$
(6') $solve\_empty \leftarrow$

Example 5.21 The following is the well known reverse with accumulating parameter which we intend to use on lists of 0's and 1's and where a simple type check has been incorporated (to make it really declarative one would have to add an extra argument representing the output — to make it really effective one would have to add an `if-then-else`).

(1) $rev([], Acc, Acc) \leftarrow$
(2) $rev([H|T], Acc, Res) \leftarrow check\_list(Acc), rev(T, [H|Acc], Res)$
(3) $check\_list(0) \leftarrow print(\text{“type error, not list: 0”})$
(4) $check\_list(1) \leftarrow print(\text{“type error, not list: 1”})$
(5) $check\_list(X) \leftarrow$

For the initial goal $G = \leftarrow rev(L, [], R)$ and using a purely determinate unfolding rule the Algorithm 5.16 will generate the following sequence of constrained atoms (the corresponding SLD-trees can be found in Figure 6):

1. $\mathcal{A}_0 = \{rev(L, [], R)\}$
2. $CBA(\hat{P}_0) = \{rev(T, [H], R)\}$
3. $\mathcal{A}_1 = chabsc_{P,U}(\mathcal{A}_0 \cup CBA(\hat{P}_0)) = \{c \square rev(L, A, R)\}$, where $c = \forall L' \forall R'. \neg(rev(L, A, R) = rev(L', 0, R') \wedge \forall L' \forall R'. \neg(rev(L, A, R) = rev(L', 1, R'))$, because $chtree(\leftarrow rev(L, [], R), P, U) = chtree(\leftarrow rev(T, [H], R), P, U) = \tau_1 = \{\langle 1 \circ 1 \rangle, \langle 1 \circ 2, 1 \circ 5 \rangle\}$ and $extpaths_P(\tau_1) = \{\langle 1 \circ 2, 1 \circ 3 \rangle, \langle 1 \circ 2, 1 \circ 4 \rangle\}$.
4. $CBA(\hat{P}_1) = \{c' \square rev(T, [H|A], R)\}$, where $c' = \forall L \forall R'. \neg(rev([H|T], A, R) = rev(L, 0, R')) \wedge \forall L \forall R'. \neg(rev([H|T], A, R) = rev(L, 1, R'))$.
5. $\mathcal{A}_2 = chabsc_{P,U}(\mathcal{A}_1 \cup CBA(\hat{P}_1)) = \mathcal{A}_1$ as $chtree_{\mathcal{FT}}(\leftarrow c' \square rev(T, [H|A], R), P, U) = \tau_1$.

Given an atomic renaming α such that $\alpha(c \square rev(L, A, R)) = rev(L, A, R)$ we obtain the following partial deduction wrt $\mathcal{A}_1$ in which the (albeit simple) type checking has been completely removed.

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

Note that, if we use a dynamic renaming strategy, then, just like for Example 3.7, we run into non-termination. If we use a strategy without renaming and an abstraction operator which allows only one msg per predicate, then partial deduction will not be able to remove the type checking.

If we use the static renaming strategy of [3] then partial deduction is in this case able to remove the type checking while guaranteeing termination. However, this comes at the cost of a larger program (because of unnecessary polyvariance due to the static guidance). Furthermore the program P can be slightly adapted such that $3, 4, 5, ...$ renamed versions are required to remove the type checking.

Also, the abstraction operators in [20, 17] or the abstraction operator $chabs_{P,U}$ without constraints of Definition 3.6 cannot adequately handle the above example and are not able to remove the type checking. In fact $chabs_{P,U}(\mathcal{A}_0) = \{rev(L, A, R)\}$ and local precision has been lost and partial deduction is no longer able to remove the type checking.

In summary, for some more elaborate specialisation examples, it is vital that the abstraction operator preserves characteristic trees and the augmented precision of the new partial deduction method pays off in improved specialisation.

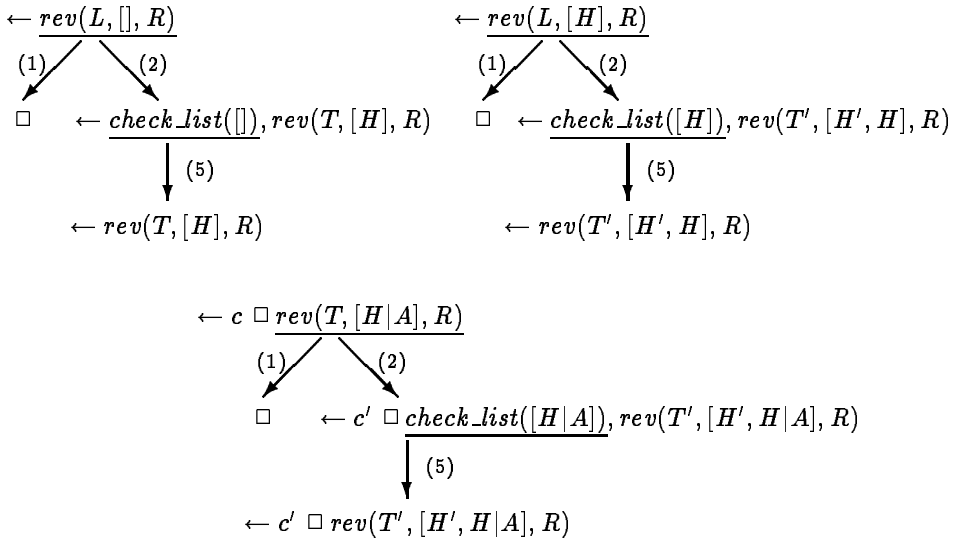

Figure 6: SLD-trees for Example 5.21

6 Some Experimental Results

An automatic partial deduction system, based on Algorithm 5.16, has been developed in order to check feasibility as well as practical potential of our approach. However, the extension to any unfolding rule and negation (by incorporating failed branches and sub-trees into the

characteristic tree with an adequate post-processing phase) has not been implemented yet. So the benchmarks and experiments were only conducted with purely determinate unfolding rules and for definite programs (which limits the amount of speedup one can expect). The particular unfolding rule used in the experiments allows non-determinate unfolding only at the top (thus guaranteeing that the backtracking behaviour will never be modified, because the top-level goal is atomic) and selects determinate literals in a left-to-right fashion.

To provide a fair comparison, we ran experiments for the following three abstraction operators, each time using exactly the same unfolding rule just described:

1. *onemsg*: This is an abstraction operator (already described in Section 2) which allows just one version per predicate and uses the *msg* to ensure this.

2. *chabs*: This is the abstraction operator defined in Definition 3.6 and which for the examples at hand basically behaves like the abstraction described in [20].

3. *chabsc*: the characteristic tree preserving abstraction operator of Definition 5.12 used inside the constrained partial deduction algorithm described in this paper.

Note that only *onemsg* and *chabsc* guarantee termination.

We compared the three approaches for the Lam & Kusalik benchmarks (see [31], they can also be found in [46, 57]) without negation and built-in's: *ancestor*, *depth*, *transpose*. We also experimented with the *rev_checklist* program from Example 5.21, which we specialised for the $S = \{rev(L, [], R)\}$. Another experiment, *member*, consisted in a slight adaptation of Example 3.11.

The timing results are summarised in Table 1. The first line contains the absolute timings, the second line contains the speedup as compared with the original program. The *Total* row contains the normalised total time (and the total speedup in the second line) of all the tests (each test was given the same weight by dividing by the execution time of the original unspecialised program). The timings were obtained by using the *time*/2 predicate of Prolog by BIM on a Sparc Classic under Solaris. Sufficient memory was given to the Prolog system to prevent garbage collections. The number of clauses and predicates was also measured and can be found in Table 2.

Test	Original	*chabsc*	*chabs*	*onemsg*
rev_checklist	0.67 s	0.32 s	0.90 s	0.67 s
	1	2.09	0.74	1
member	0.46 s	0.18 s	0.24 s	0.30 s
	1	2.56	1.92	1.53
ancestor	6.37 s	5.85 s	5.85 s	5.85 s
	1	1.09	1.09	1.09
depth	2.38 s	2.15 s	2.15 s	2.38 s
	1	1.11	1.11	1
transpose	2.00 s	0.43 s	0.43 s	0.41 s
	1	4.65	4.65	4.87
Total	5	2.90	3.90	3.77
	1	1.72	1.28	1.33

Table 1: Speedup Figures

Test	Original	*chabsc*	*chabs*	*onemsg*
rev_checklist	5-2	2-1	8-3	5-2
member	4-2	5-3	5-3	4-2
ancestor	15-5	15-5	15-5	5-2
depth	8-3	11-5	11-5	8-3
transpose	6-3	4-2	4-2	2-1

Table 2: Program Sizes: Number of clauses and predicates

Note that in the transpose example the extra version produced by *chabsc* (and *chabs*) was not beneficial which might have been caused by some Sparc caching behaviour. Also note that Lam & Kusalik benchmarks are not very sophisticated and the *chabs* operator had no problem with termination and precision.

In summary we can say that, even when using a simple unfolding rule, the abstraction operator *chabsc* looks very promising and seems to be a good basis for a flexible polyvariance providing just as many versions as necessary (i.e. only one version for the *rev_checklist* example but 5 for the *depth* example).

7 Further Improvements, Discussion and Related Work

In previous sections we have already hinted at two possibilities to lift the restriction to failure preserving unfolding rules. If we want to expand the method to encompass normal logic programs some further difficulties arise (cf. Example 3.9).

First, the abstraction operator will often have to ensure that a selected negative literal $\neg A$ succeeds. In the context of SLDNF, this amounts to ensuring that A is ground and fails finitely. For the former, some form of groundness constraint seems to be required (this problem can be avoided if we use the SLS semantics of [55]). The latter is very similar to the difficulty encountered for non-failure preserving unfolding rules (cf. Example 5.9) because there can also be an infinite number of possibilities in which the subtree for $\leftarrow A$ can be made to fail. So, a first possibility to solve this problem is to not only incorporate the failed branches into the characteristic trees, but the sub-trees for negative literals as well. This will lead to an even bigger polyvariance (which might be removed by a post-processing phase, but this might be impractical due to the large amount of polyvariance). A second possibility would be to extend the expressivity of the constraints. A promising approach in that direction is to extend the approach of computing fail substitutions [45].

There is, however, still a third possibility discussed in [34]. This method follows the same basic principle laid down in this paper, namely to use and preserve characteristic trees in order to obtain a fine-grained control of polyvariance, but achieves this without explicitly incorporating constraints into the partial deduction process. The central idea of [34] is actually rather simple (and is a further development of the idea which we used in the previous section to transform any unfolding rule into a stable one): the method just *imposes* a characteristic tree on the generalisation. This characteristic tree acts as a sort of implicit local constraint. As such the method does not have to impose any restriction on the unfolding rule, can handle negation (and some built-in's as well) while still ensuring termination.

However, the simplicity comes at the price of some loss of precision because the implicit

constraints in [34] are only used locally (the method here, based on negative constraints, uses the constraints explicitly and propagates them globally via constrained atoms to be partially deduced). Also, the full instance relation now becomes undecidable, and a computable approximation has to be used.

Algorithm 5.16 based on $chabsc_{P,U}$ (as well as [34]) still requires an ad-hoc depth bound on characteristic trees to ensure termination. As a partial remedy we can easily extend the algorithm so that the precision of the characteristic trees is limited to a certain depth but the unfolding rule has no a priori depth bound. Indeed, our abstraction operator $chabsc_{P,U}$ ensures that characteristic trees can be preserved, but does not *force* the unfolding rule to actually perform the same unfolding (and the unfolding rule can thus unfold deeper than the characteristic tree if it wants to).

However, even with that improvement, the precision of characteristic trees is still limited and the depth bound can result in unsatisfactory, ad-hoc specialisation (see [40, 35]). Fortunately, by combining our approach with [51], it is possible to get rid of this ad-hoc depth bound. The basic idea is to use a refined well-quasi order on characteristic trees which spots potential sequences of ever growing characteristic trees. The details of this approach have been elaborated in [40, 35] (applied to [34], but the approach can be applied in exactly the same manner to the method of this paper).

At first sight, the post-processing abstract interpretation phase of [12, 21], detecting useless clauses, might seem like a viable alternative to using pruning constraints and the framework of constrained partial deduction. However, such an approach can not bring back the precomputation that has been lost by an imprecise abstraction operator — it might only be able to bring back part of the pruning. But, when running the method of [12, 21] e.g. on the residual program P' of Example 3.11, no useless clauses are detected. Indeed to be able to do so, one needs an analysis which can do some form of unfolding and in that process preserve characteristic trees — in other words exactly the method that we have developed in this paper. So neither of the two approaches subsumes the other, they are complementary. Another related work is [11], which uses abstract substitutions to prune resultants while unfolding. These abstract substitutions play a role very similar to the constraints in the current paper. However, no formal correctness or termination result is given in [11] (and the issue of preserving characteristic trees is not addressed). Indeed, as abstract substitutions of [11] are not necessarily downwards-closed, this seems to be a much harder task and a normal coveredness condition will not suffice to ensure correctness (for instance the atoms in the bodies of clauses might be further instantiated at run-time and thus, in the absence of downwards-closedness, no longer covered). Our paper actually provides a framework within which correctness of [11] could be established for abstract substitutions which are downwards-closed. Another, more technical difference is that neither the method of [12, 21] nor the method of [11] preserve the finite failure semantics (i.e. infinite failure might be replaced by finite failure), while our approach, just like ordinary partial deduction, does.

Another method that might look like a viable alternative to our approach is the one of [5], situated within the context of unfold/fold transformations. In particular, [5] contains many transformation rules and allows first-order logic formulas to be used to constrain the specialisation. It is thus a very powerful framework. But also, because of that power, controlling it in an automatic way, as well as ensuring actual efficiency gains, is much more difficult. A prototype for [5] exists, but the control heuristics as well as the correctness proofs are still left to the user.

Let us also briefly discuss some further applications of constrained partial deduction,

beyond preserving characteristic trees. For example, a constraint structure over integers or reals could handle Prolog built-ins like $<, >, \leq, \geq$ in a much more sophisticated manner than ordinary partial evaluators. Also, one can provide a very refined treatment of the \== Prolog built-in using the $\mathcal{FT}$ structure (this feature has actually been incorporated in the prototype of the previous section, but has not been used in the experiments). The following example illustrates this, where a form of "driving of negative information" (using the terminology of supercompilation [67, 68]) is achieved by constrained partial deduction.

Example 7.1 Take the following adaptation of the *member* program which only succeeds once.

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

Let us start specialisation with the goal $\leftarrow member(X, [a, Y, a])$. Using a determinate unfolding rule (even with a lookahead) we would unfold this goal once and get the resolvent $\leftarrow X \backslash== a, member(X, [Y, a])$ in the second branch. Ordinary partial deduction would ignore $X \backslash== a$ and unfold $member(X, [Y, a])$, thus producing an extra superfluous resultant with the impossible (given the context) computed answer $\{X/a\}$. In the constrained partial deduction setting, we can incorporate $X \backslash== a$ as a constraint and unfold $\neg(X = a) \square member(X, [Y, a])$ instead of just $member(X, [Y, a])$ and thereby prune the superfluous resultant.

The program in the above example is actually almost a CLP-program, and we could go one step further and also specialise CLP-programs. As Algorithm 5.16 is based on the structure $\mathcal{FT}$ we conjecture that an adaptation of our technique might yield a refined specialisation technique for CLP($\mathcal{FT}$) [60, 61, 62]. Also, it is actually not very difficult to adapt the framework of Section 4.2 to work on CLP-programs instead of ordinary logic programs — we just have to require that equality is handled in the same manner as in logic programming. However, establishing the correctness will become much more difficult because one cannot reuse the correctness results of standard partial deduction. In that context, we would like to mention [66], which extends constructive negation for CLP-programs, as well as recent work on the transformation of CLP-programs [16]. Note, however, that [16] is situated within the unfold/fold transformation paradigm and also that no concrete algorithms are presented.

Finally, let us mention a recent extension of partial deduction, called conjunctive partial deduction [39, 22]. Conjunctive partial deduction handles conjunctions of atoms instead of just atoms. This means that when the unfolding rule stops, the atoms in the leaves of the SLD(NF)-tree are not automatically separated and treated in isolation. As such, the local precision problem disappears almost entirely and approaches based on determinate unfolding become much more viable (recent experiments in [26] confirm this, where determinate unfolding outperforms more eager unfolding rules based on well-founded or well-quasi measures). The method of this paper can be easily adapted to work in that setting, and there might even be no need to extend it to allow non-failure preserving unfolding rules.

8 Conclusion

We have shown that characteristic trees are very useful to obtain a fine grained control of polyvariance for partial deduction. We have shown that, for precision and termination, it is crucial that characteristic trees are preserved by the abstraction operator of a partial

deduction algorithm. If this is the case we can obtain a partial deduction method giving us the right amount of global precision which avoids any loss of local precision. However, the preservation of characteristic trees turns out to be a substantial problem, and the approaches in the literature so far do not exhibit this desirable property.

To overcome this difficulty we have developed the framework of *constrained partial deduction*, based on introducing constraints into the partial deduction process. We have provided formal correctness results for this framework and have shown that it offers potential beyond the preservation of characteristic trees.

Because of the added expressivity and precision of the constraints we were able to devise an abstraction operator for constrained partial deduction which preserves characteristic trees for definite logic programs and failure preserving unfolding rules (and which can be extended to any unfolding rule by incorporating the failing branches into the characteristic trees — other possibilities to extend the method were also outlined) while at the same time guaranteeing correctness and termination. The method has been shown to be useful on some examples leading to enhanced precision and specialisation and some promising experiments were conducted.

We were thus able to devise a partial deduction algorithm with a very fine grained *control of polyvariance*, no loss of local *precision* due to the abstraction while ensuring *termination* and *correctness*.

Acknowledgements

Michael Leuschel is supported by Esprit BR-project Compulog II. Danny De Schreye is senior research associate of the Belgian National Fund for Scientific Research. We would like to thank Bern Martens for proof-reading (several versions) of this paper, for his subtle comments and for the stimulating discussions. We would also like to thank John Gallagher and Maurice Bruynooghe for their helpful remarks. We appreciated interesting discussions with Włodek Drabent and André De Waal. We are also grateful to Marc Denecker for providing us with relevant insights into semantical issues and equality theory. Finally we thank anonymous referees for their comments and challenging criticism, which helped to substantially improve the paper.

References

[1] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 10, pages 495–574. North-Holland Amsterdam, 1990.

[2] K. Benkerimi and P. M. Hill. Supporting transformations for the partial evaluation of logic programs. *Journal of Logic and Computation*, 3(5):469–486, October 1993.

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

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

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

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

[7] D. Chan. Constructive negation based on the completed database. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 111–125, Seattle, 1988. IEEE, MIT Press.

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

[9] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[10] D. De Schreye and S. Decorte. Termination of logic programs: The never ending story. *The Journal of Logic Programming*, 19 & 20:199–260, May 1994.

[11] D. A. de Waal and J. Gallagher. Specialisation of a unification algorithm. In T. Clement and K.-K. Lau, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'91*, pages 205–220, Manchester, UK, 1991.

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

[13] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[14] K. Doets. Levationis laus. *Journal of Logic and Computation*, 3(5):487–516, 1993.

[15] W. Drabent. What is failure ? An apporach to constructive negation. *Acta Informatica*, 32:27–59, 1995.

[16] S. Etalle and M. Gabbrielli. A transformation system for modular CLP programs. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 681–695. The MIT Press, 1995.

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

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

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

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

[21] J. Gallagher and D. A. de Waal. Deletion of redundant unary type predicates from logic programs. In K.-K. Lau and T. Clement, editors, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'92*, pages 151–167, Manchester, UK, 1992.

[22] R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction of definite logic programs. In H. Kuchen and S. Swierstra, editors, *Proceedings of the International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'96)*, LNCS 1140, pages 152–166, Aachen, Germany, September 1996. Springer-Verlag. Extended version as Technical Report CW 226, K.U. Leuven. Available at `http://www.cs.kuleuven.ac.be/~lpai`.

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

[24] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–581, 1994.

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

[26] J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive partial deduction in practice. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 59–82, Stockholm, Sweden, August 1996. Springer-Verlag. Also in the Proceedings of BENELOG'96. Extended version as Technical Report CW 242, K.U. Leuven.

[27] H.-P. Ko and M. E. Nadel. Substitution and refutation revisited. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 679–692. MIT Press, 1991.

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

[29] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Proceedings Meta'92*, LNCS 649, pages 49–69. Springer-Verlag, 1992.

[30] K. Kunen. Answer sets and negation as failure. In J.-L. Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming*, pages 219–228. The MIT Press, 1987.

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

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

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

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

[35] M. Leuschel. *Advanced Techniques for Logic Program Specialisation.* PhD thesis, K.U. Leuven, May 1997. Available at `http://www.cs.kuleuven.ac.be/~michael`.

[36] M. Leuschel and D. De Schreye. An almost perfect abstraction operator for partial deduction. Technical Report CW 199, Departement Computerwetenschappen, K.U. Leuven, Belgium, December 1994.

[37] M. Leuschel and D. De Schreye. An almost perfect abstraction operation for partial deduction using characteristic trees. Technical Report CW 215, Departement Computerwetenschappen, K.U. Leuven, Belgium, October 1995.

[38] 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.

[39] M. Leuschel, D. De Schreye, and A. de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press. Extended version as Technical Report CW 225, K.U. Leuven. Available at `http://www.cs.kuleuven.ac.be/~lpai`.

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

[41] M. Leuschel and M. H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 83–103, Stockholm, Sweden, August 1996. Springer-Verlag. Extended version as Technical Report CW 243, K.U. Leuven.

[42] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

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

[44] M. Maher. A logic programming view of CLP. In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 737–753. The MIT Press, 1993.

[45] J. Małuszyński and T. Näslund. Fail substitutions for negation as failure. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference*, pages 461–476. MIT Press, 1989.

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

[47] B. Martens and D. De Schreye. Two semantics for definite meta-programs, using the non-ground representation. In K. R. Apt and F. Turini, editors, *Meta-logics and Logic Programming*, pages 57–82. MIT Press, 1995.

[48] B. Martens and D. De Schreye. Why untyped non-ground meta-programming is not (much of) a problem. *The Journal of Logic Programming*, 22(1):47–99, 1995.

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

[50] 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.

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

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

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

[54] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *The Journal of Logic Programming*, 16(1 & 2):123–162, May 1993.

[55] T. C. Przymusinksi. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.

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

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

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

[59] J. C. Shepherdson. Language and equality theory in logic programming. Technical Report PM-91-02, University of Bristol, 1991.

[60] D. A. Smith. Constraint operations for CLP($\mathcal{FT}$). In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 760–774. MIT Press, 1991.

[61] D. A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. In N. D. Jones and P. Hudak, editors, *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–71. ACM Press Sigplan Notices 26(9), 1991.

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

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

[64] L. Sterling and R. D. Beer. Metainterpreters for expert system construction. *The Journal of Logic Programming*, 6(1 & 2):163–178, 1989.

[65] P. J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 328–339, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[66] P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, April 1995.

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

[68] V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

A Counter Example

In this appendix we present a counter example to Lemma 4.11 on page 326 of [20]. Note that the definitions differ from the ones in [17] and from the ones adopted in our paper (for instance what is called a *chpath* in [20] corresponds more closely to the concept of a characteristic tree in our paper than to the notion of a characteristic path).

We take the following program P (similar to Example 3.8, the actual definitions of $r(X)$ and $s(X)$ are of no importance):

(c_1) $p(X) \leftarrow q(X)$
(c_2) $p(c) \leftarrow$
(c_3) $q(X) \leftarrow r(X)$
(c_4) $q(X) \leftarrow s(X)$
(c_5) $r(X) \leftarrow \ldots$
(c_6) $s(X) \leftarrow \ldots$

Now let the atom A be $p(b)$. Then according to definition 4.5 of [20] we have that $chpath(A) = (\langle c_1 \rangle, \{c_3, c_4\})$. According to definition 4.10 of [20] we obtain: $chpaths(A) = \{\langle c_1, c_3 \rangle, \langle c_1, c_4 \rangle\}$.

The most general resultants (definition 4.6 of [20]) of the paths in $chpaths(A)$ is the set $\{p(Z) \leftarrow r(Z), p(Z) \leftarrow s(Z)\}$.

By definition 4.10 of [20] we obtain the *characteristic call* of A:
$chcall(A) = msg\{p(Z), p(Z)\} = p(Z)$.

In Lemma 4.11 of [20] it is claimed that $chpath(chcall(A)) = chpath(A)$ and that $chpath(msg\{A, chcall(A)\}) = chpath(A)$, i.e. it is claimed that $chpath(msg\{A, chcall(A)\})$ "abstracts" A (finds a more general atom) while preserving the characteristic path structure. However, in our example we have that:
$chpath(chcall(A)) = chpath(msg\{A, chcall(A)\}) = chpath(p(Z)) = (\langle\rangle, \{c_1, c_2\}) \neq chpath(A)$
and thus Lemma 4.11 is false.

B Termination property of $chabs_{P,U}$

In this appendix we prove a termination property of the abstraction operator $chabs_{P,U}$ defined in Definition 3.6.

The following well-founded measure function is taken from [19] and can also be found in the extended version of [51]:

Definition B.1 *Let* Term *and* Atom *denote the sets of terms and atoms, respectively. We define the function* $scount : Term \cup Atom \rightarrow \mathbb{N}$ *counting symbols by:*

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

Let the number of distinct variables in a term or atom t *be* $vcount(t)$.

We now define the function $hcount : Term \cup Atom \rightarrow \mathbb{N}$ *by* $hcount(t) = scount(t) - vcount(t)$.

The well-founded measure function $hcount$ has the property that $hcount(t) > 0$ for any non-variable t. Also if A is an atom strictly more general than B we have that $hcount(A) < hcount(B)$ (see [51]).

Definition B.2 ($hvec_{T,P,U}$) *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. Also let* S *be a set of atoms. For every characteristic tree* τ_i, *let* $\mathcal{A}_{\tau_i}$ *be defined as* $\mathcal{A}_{\tau_i} = \{A \mid A \in S \wedge chtree(\leftarrow A, P, U) = \tau_i\}$.

We then define the weight vector *of* S *wrt* T, P *and* U, *denoted by* $hvec_{T,P,U}(S)$, *as:* $hvec_{T,P,U}(S) = \langle w_1, \ldots, w_n \rangle$ *where*

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

Weight vectors are partially ordered by the usual order relation among vectors (i.e. $\langle w_1, \ldots, w_n \rangle \leq \langle v_1, \ldots, v_n \rangle$ iff $w_1 \leq v_1, \ldots, w_n \leq v_n$ and $\vec{w} < \vec{v}$ iff $\vec{w} \leq \vec{v}$ and $\vec{v} \not\leq \vec{w}$. The set of weight vectors is well founded (no infinitely decreasing sequences exist) because the weights of the atoms are well founded.

Proposition B.3 (Termination using $chabs_{P,U}$) *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 finite set of atoms $\mathbf{A}$ and S such that the characteristic trees of their atoms are in T and such that the abstraction operator $chabs_{P,U}$ preserves the characteristic trees (in the sense that, for each $\mathbf{A}_\tau$ in Definition 3.6, the characteristic tree of $msg(\mathbf{A}_\tau)$ is exactly τ) we have that one of the following holds:

- $chabs_{P,U}(\mathbf{A} \cup S) = \mathbf{A}$ *(up to variable renaming) or*
- $hvec_{T,P,U}(chabs_{P,U}(\mathbf{A} \cup S)) < hvec_{T,P,U}(\mathbf{A})$.

Proof Again, let (for any finite set of atoms S and any characteristic tree τ) S_τ be defined as $S_\tau = \{A \mid A \in S \wedge chtree(\leftarrow A, P, U) = \tau\}$. Also let $hvec_{T,P,U}(\mathbf{A}) = \langle w_1, \ldots, w_n \rangle$ and let $hvec_{T,P,U}(chabs_{P,U}(\mathbf{A} \cup S)) = \langle v_1, \ldots, v_n \rangle$. Then for every $\tau_i \in T$ we have two cases:

- $\{msg(\mathbf{A}_{\tau_i} \cup S_{\tau_i})\} = \mathbf{A}_{\tau_i}$ (up to variable renaming). In this case the abstraction operator performs no modification for τ_i and $v_i = w_i$.
- $\{M\} = \{msg(\mathbf{A}_{\tau_i} \cup S_{\tau_i})\} \neq \mathbf{A}_{\tau_i}$ (up to variable renaming). In this case there are three possibilities:
  - $\mathbf{A}_{\tau_i} = \emptyset$. In this case $v_i < w_i = \infty$ because the characteristic tree of M is still τ_i.
  - $\mathbf{A}_{\tau_i} = \{A\}$ for some atom A. In this case M is strictly more general than A (by definition of *msg* because $M \neq A$) and hence $v_i < w_i$ because M has τ_i as its characteristic tree.
  - $\#(\mathbf{A}_{\tau_i}) > 1$. In this case M is more general (but not necessarily strictly more general) than any atom in $\mathbf{A}_{\tau_i}$) and $v_i < w_i$ because at least one atom is removed by the abstraction operator and because M has τ_i as its characteristic tree.

  Note that for three points above it was vital that the abstraction operator preserves the characteristic trees.

Note that $\forall i \in \{1, \ldots, n\}$ we have that $v_i \leq w_i$ and the new weight vector $\langle v_1, \ldots, v_n \rangle$ will be comparable to the old vector $\langle w_1, \ldots, w_n \rangle$. So either the abstraction operator performs no modification at all (and the weight vectors are identical) or the well-founded measure function $hvec_{T,P,U}$ strictly decreases. □

So, if characteristic trees are preserved by the abstraction operator then termination of the general partial deduction Algorithm 2.9 is guaranteed. However, if characteristic trees are not preserved by the abstraction operator then the above proof no longer holds and termination is no longer guaranteed (even assuming a finite number of characteristic trees, see [37]) !

C Lemmas for Proving Correctness of Constrained Partial Deduction

We extend the concept of valid $\mathcal{D}$-instances to goals by stating that $\leftarrow Q' \in valid_{\mathcal{D}}(\leftarrow c \,\square\, Q)$ iff there exists a substitution γ such that $Q' = Q\gamma$ and γ *sat* c.

Lemma C.1 (persistence of validity) *Let G be an ordinary goal and CG a CLP-goal. Let $G \in valid_{\mathcal{D}}(CG)$. Let CD be a $CLP_{=}(\mathcal{D})$-derivation for $P \cup \{CG\}$ with $\mathcal{D}$-characteristic path p and resolvent CG' and let D be an SLD-derivation for $P \cup \{G\}$ with characteristic path p and resolvent G'. Then $G' \in valid_{\mathcal{D}}(CG')$.*

Proof First, note that if θ *sat* c then for any set of variables $\mathcal{V}$ we have that θ *sat* $\tilde{\exists}_{\mathcal{V}}(c)$ (and even that $\theta\,|_{\mathcal{V}}$ *sat* $\tilde{\exists}_{\mathcal{V}}(c)$).
Let us do the proof by induction on the length of D and CD (as they have the same characteristic path they must be of the same length).
Induction Hypothesis: Lemma C.1 holds for all derivations D with length $\leq n$.
Base Case: (D and CD have length 0). Trivial, as $G = G'$ and $RG = RG'$.
Induction Step: (D and CD have length $n+1$). Let $G = \leftarrow A_1, \ldots, A_k$ and $CG = \leftarrow c \,\square\, C_1, \ldots, C_k$. We know by definition that $A_i = C_i\gamma$ where γ $sat_{\mathcal{D}}$ c. Let A_i be the selected literal and $C = H \leftarrow B_1, \ldots, B_q$ be the clause chosen for the first resolution step of D and CD. Let $G_1 = \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_q, A_{i+1}, \ldots, A_k)\theta$ be the goal after the first resolution step in D where θ be the first *mgu* in D (i.e. θ is an idempotent and relevant *mgu* of A_i and H). Let $CG_1 = \leftarrow c' \,\square\, Q$ be the CLP-goal after the first resolution step in CD, where $Q = (C_1, \ldots, C_{i-1}, B_1, \ldots, B_q, C_{i+1}, \ldots, C_k)\theta_C$ and where θ_C is the first *mgu* in CD (i.e. θ_c is an idempotent and relevant *mgu* of C_i and H) and $c' = \tilde{\exists}_{vars(Q)}(c\theta_c)$. By the (correct version) of the lifting lemma[16] we know that there exists a substitution β such that $C_j\theta_c\beta = A_j\theta$ for $j \in \{1, \ldots, k\}$ [17] and $B_j\theta_c\beta = B_j\theta$ for $j \in \{1, \ldots, q\}$.
Now, we know that γ $sat_{\mathcal{D}}$ c. Hence also $\gamma\theta$ $sat_{\mathcal{D}}$ c. Now as $C_j\gamma\theta = C_j\theta_c\beta$ for all j we know that $\gamma\theta$ and $\theta_c\beta$ have the same effect on the variables in CG and thus in c. Hence we also have that $\theta_c\beta$ $sat_{\mathcal{D}}$ c. By definition of satisfaction this is equivalent to saying that β $sat_{\mathcal{D}}$ $c\theta_c$. This also implies that β $sat_{\mathcal{D}}$ $\tilde{\exists}_{vars(Q)}(c\theta_c)$. As $G_1 = \leftarrow Q\beta$ we have established that $G_1 \in valid_{\mathcal{D}}(CG_1)$. We can now use the induction hypothesis for the remaining n steps of D and CD. □

The previous lemma talks about unrenamed goals and derivations in the original (unrenamed) program. For the general correctness theorem we have to reason on derivations of *renamed* goals in the (renamed) specialised program. The following lemma affirms, under certain conditions, that for every renamed goal we might possibly obtain in the specialised program we can always find some unrenamed goal of which it is a valid renaming (i.e. satisfying Definition 4.10).

Lemma C.2 *Let P' be a partial deduction of P wrt $\mathcal{A}$, $\hat{P}$ and ρ_α such that $\mathcal{A}$ is a finite set of constrained atoms and let G be an ordinary goal such that $\hat{P} \cup \{G\}$ is $\mathcal{A}, \mathcal{D}$-covered. Also let $G' = \rho'_\alpha(G)$, where ρ'_α is a renaming function based on α. Let D' be a finite SLD-derivation for $P' \cup \{G'\}$ leading to the resolvent RG'. Then there exists an ordinary goal RG and a renaming function ρ''_α based on α such that $RG' = \rho''_\alpha(RG)$ and such that RG is $\mathcal{A}, \mathcal{D}$-covered.*

Proof First note that, if $\rho'_\alpha(RG)$ is defined, RG must by definition be $\mathcal{A}, \mathcal{D}$-covered.
We can prove the lemma by induction on the length of D'.

[16] See e.g. the lifting lemmas in [27, 14, 1], but also [43] whose lifting Lemma 4.1 is different from the incorrect one in [42].

[17] The lifting lemma only affirms this for $j \neq i$, but it is easy to see that we can always find a β which also satisfies the above for $i = j$ (by simply applying the lifting lemma to clause C' in which we add H as a body atom to the clause C) because $C_i\theta_c = H\theta_c$.

Induction Hypothesis: Lemma C.2 holds for all derivations D with length $\leq n$.
Base Case: (D' has length 0). Trivial, we simply take $RG = G$ and $\rho''_\alpha = \rho'_\alpha$.
Induction Step: (D' has length $n+1$). Let $G = \leftarrow A_1, \ldots, A_i, \ldots, A_k$ and let $\rho'_\alpha(A_i)$ be the literal in $G' = \rho'_\alpha(G)$ which is selected by D'. Let $C' = \alpha(c \square A)\theta \leftarrow \rho_\alpha(c' \square B_1, \ldots, B_q)$ with $c \square A \in \mathcal{A}$ be the clause in P' used in the first resolution step of D'. Let $G'_1 = \leftarrow (\rho'_\alpha(A_1), \ldots, \rho'_\alpha(A_{i-1}), \rho_\alpha(B_1), \ldots, \rho_\alpha(B_q), \rho'_\alpha(A_{i+1}), \ldots, \rho'_\alpha(A_k))\sigma$ be the goal obtained after the first resolution step for G' in D' and let $G_1 = \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_q, A_{i+1}, \ldots, A_k)\sigma$ where σ is the first *mgu* in D'. We will show that it is possible to rename G_1 into G'_1.
Because $\rho'_\alpha(G)$ is defined we know that $true \square A_i$ is a $\mathcal{D}$-instance of $c \square A$. Furthermore, by Lemma 4.18 this means that $A_i \in valid_\mathcal{D}(c \square A)$. Let $\hat{C} = A\theta \leftarrow c' \square B_1, \ldots, B_q$ be the unrenamed version of C' in $\hat{P}$. By construction of $\hat{C}$, we know that there is a CLP$_=(\mathcal{D})$-derivation for $P \cup \{\leftarrow c \square A\}$ with computed answer θ and resolvent $c' \square B_1, \ldots, B_q$. We can now apply Lemma C.1 (persistence of validity) to deduce that the atoms $\leftarrow B_1\sigma, \ldots, B_q\sigma \in valid_\mathcal{D}(\leftarrow c' \square B_1, \ldots, B_q)$. This implies that each $B_i\sigma$ is a valid $\mathcal{D}$-instance of the corresponding constrained body atom of $\hat{C}$, i.e. $B_i\sigma \in valid_\mathcal{D}(\tilde{\exists}_{vars(B_i)}(c') \square B_i)$. Now as each constrained body atom $\tilde{\exists}_{vars(B_i)}(c') \square B_i$ of $\hat{C}$ is in turn an instance of a constrained atom CA_i in $\mathcal{A}$ with $\rho_\alpha(B_i) = \alpha(CA_i)\beta_i$ (because $\hat{P}$ is $\mathcal{A},\mathcal{D}$-covered and by definition of a renaming function), we simply construct a renaming ρ''_α such that $\rho''_\alpha(B_i\sigma) = \rho_\alpha(B_i)\sigma$ and such that $\rho''_\alpha(A_i\sigma) = \rho'_\alpha(A_i)\sigma$. We thus have constructed a renaming ρ''_α and a goal G_1 such that $\rho''_\alpha(G_1) = G'_1$ and we can apply the induction hypothesis for the remaining n steps of D'. □

One might wonder why three different renaming functions $(\rho_\alpha, \rho'_\alpha, \rho''_\alpha)$ are needed in the above lemma. Usually the top-level goal G' will be renamed using ρ_α and one might think that it is possible to prove that RG' is the renaming of some goal RG under ρ_α, i.e. $RG' = \rho_\alpha(RG)$. Unfortunately, in general, no such goal RG exists ! The reason is that in the course of performing resolution steps atoms might become more instantiated, meaning that the renaming function ρ_α would, based on this instantiation, rename differently. Take for example the set $\mathcal{A} = \{true \square p(X), true \square p(a)\}$ of unconstrained atoms, the goal $G = \leftarrow p(X), p(X)$ and take α such that $\alpha(true \square p(X)) = p'(X)$, $\alpha(true \square p(a)) = p_a$. Then $\rho_\alpha(G) = \leftarrow p'(X), p'(X)$. Also assume that $\rho_\alpha(p(a)) = p_a$. Now suppose that the clause $p'(a) \leftarrow$ is in the partial deduction P' wrt an original P and the set $\mathcal{A}$. Then after one resolution step for $P' \cup \rho_\alpha(G)$ we obtain the goal $\leftarrow p'(a)$ and for no goal RG we have that $\rho_\alpha(RG) = \leftarrow p'(a)$. Indeed $\rho_\alpha(\leftarrow p(a)) = \leftarrow p_a \neq \leftarrow p'(a)$. However, we can construct *another* renaming function ρ''_α such that $\rho''_\alpha(\leftarrow p(a)) = \leftarrow p'(a)$. So Lemma C.2 holds (and three, possibly distinct renaming functions are needed if we want to repeatedly apply the lemma).

D Termination of Constrained Partial Deduction

In this appendix we prove Proposition 5.18.
Proposition 5.18 If the set of different characteristic trees is finite and the unfolding rule U is stable, then Algorithm 5.16 terminates.
Proof The proof is very similar to the one in Appendix B (and stability of the unfolding rule ensures that the characteristic tree of the generalisation of A_τ is exactly τ). First we have to extend the definition of *hcount* in Appendix B to constrained atoms: $hcount(c \square A) = hcount(A)$. Then we extend the definition of $hvec_{T,P,U}$ to sets of constrained atoms by taking, for each characteristic tree τ, *hcount* of the normalised constrained atoms only (and ∞ if there are none). Note that after the first step of the algorithm, $\mathcal{A}_i$ will only contain

normalised constrained atoms. We now prove in a very similar way to Proposition B.3 that for every finite set of constrained atoms New_i and $\mathcal{A}_i$, where we define $New_i' = \{CA \in New_i \mid chtree(CA, P, U) \neq \emptyset\}$, we have the following:

- either $chabsc_{P,U}(\mathcal{A}_i \cup New_i') = \mathcal{A}_i$
- or $hvec_{T,P,U}(chabsc_{P,U}(\mathcal{A}_i \cup New_i')) < hvec_{T,P,U}(\mathcal{A}_i)$.

This is sufficient to prove termination, as constrained atoms with empty characteristic trees are kept unchanged by $chabsc_{P,U}$ and do not lead to further constrained body atoms that have to be added. In other words, if we reach a point where $chabsc_{P,U}(\mathcal{A}_i \cup New_i') = \mathcal{A}_i$, then at the next step of the Algorithm 5.16 we reach a point where $\mathcal{A}_{i+1} = chabsc_{P,U}(\mathcal{A}_{i+1} \cup New_{i+1})$.
□