An Almost Perfect Abstraction Operator for Partial Deduction

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

January 18, 1995

Abstract

A partial deduction strategy for logic programs usually uses 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 still does not loose relevant information (with respect to the partial deduction) is a difficult problem. In [4] and [7] Gallagher and Bruynooghe proposed to base the abstraction operator on characteristic paths and trees. A characteristic tree captures the structure of the generated partial SLDNF-tree for a given goal, i.e. it captures the relevant information for partial deduction. The generation of more general atoms having the same characteristic tree would lead to an almost perfect abstraction operator. Unfortunately the abstraction operators proposed in [4] and [7] do not always produce more general atoms and do not always preserve the characteristic trees. In this paper we propose to solve this problem through the use of constraints in the partial deduction process. We show that satisfiability of these constraints is decidable and that they do not introduce a termination problem of their own. We will thus present a partial deduction strategy which

- has an almost perfect abstraction operator.
- almost perfect control of polyvariance.
- always terminates while ensuring the closedness and independence conditions (slightly adapted to allow constraints) of [12].

1 Introduction

Partial evaluation has received considerable in logic programming. In [9] 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 [12], 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, **A**, provide a terminating procedure that computes a new set of atoms, **A**', and a partial deduction for the atoms in **A**', such that:

- every atom in **A** is an instance of an atom in **A**', and
- the closedness and independence conditions are satisfied.

Moving from the initial set **A** to the new set **A**' requires an abstraction operator. In addition to the conditions stated above, this abstraction operator should be such that the specialisation that the partial deduction achieves for the atoms in **A**' does not loose too much precision wrt the specialisation that a partial deduction could in principle obtain for the atoms in **A**.

An approach which achieves all these goals in an elegant and refined way is that of Gallagher and Bruynooghe ([7], [4]). Its abstraction operator is based on the notions of *characteristic path*, *characteristic tree* and *most specific generalisation*. Intuitively, two atoms of **A** are replaced by their most specific generalisation in **A**', if their (incomplete) SLDNF-trees under the given unfolding rule have an identical structure (this structure is referred to as the characteristic tree). By carefully tuning the various operations involved in the partial deduction algorithm, this abstraction operator allows to achieve all above mentioned goals.

Unfortunately, although the approach is conceptually appealing, several errors turn up in arguments provided in [7] and [4]. These errors invalidate the termination proofs as well as the arguments regarding preservation of specialisation under the abstraction operator. This also puts serious limitations on the practical results that can be obtained (especially wrt the control of polyvariance).

In the current paper, we significantly adapt the approach to overcome these problems. We introduce an alternative abstraction operator, which is based on so called *negative binding constraints*. The partial deduction procedure is then formulated in terms of a special purpose constraint logic programming language, which computes on the basis of such constraints. The adapted approach allows to solve all problems with the original formulations in [7] and [4], without loosing the claimed termination and precision properties. As such, we provide evidence that (a reformulation of) [7] and [4] is correct and provides all claimed benefits.

The paper is structured as follows. In the next section we introduce some preliminary notions and properties. In section 3 we present a general algorithm for partial deduction. Section 4 recalls the notion of a characteristic tree and provides examples of the errors and shortcomings in [7] and [4]. In section 5 we introduce negative binding constraints, which form the basis of the new abstraction operator. The next section presents how SLD and SLDNF can be adapted to incorporate computation on such constraints. Section 7 then reformulates the partial deduction procedure, using the constraint resolution of the previous section. We end with a brief discussion on how the approach can be extended to preserve characteristic trees for unfolding rules that select negative literals or are not determinate and then summarise our results. All proofs and auxiliary lemmas have been omitted due to space restrictions.

2 Preliminaries and Motivations

In [12] the concept of partial deduction in logic programming was put on a firm theoretical basis. The following general description of partial deduction was given. Given a program P and a goal G, partial deduction produces a new program P' which is P "specialised" to the goal G. The underlying technique is to construct "incomplete" search trees and extract the specialised program P' from these incomplete search trees. The following definition formalises this and is adapted from [12].

Definition 2.1 (partial deduction)
Let P be a normal program and A an atom. Let τ be an incomplete[1] SLDNF tree for $P \cup \{\leftarrow A\}$,

[1]An *incomplete* SLDNF tree can have arbitrary goal statements as leaves (and not only success and failure nodes).

let $\leftarrow G_1, \ldots, \leftarrow G_n$ be the goals in the (non-root) leaves of the non-failing branches of τ. Let $\theta_1, \ldots, \theta_n$ be the computed answers of the derivations from $\leftarrow A$ to $\leftarrow G_1, \ldots, \leftarrow G_n$ respectively. Then the set of resultants $\{A\theta_1 \leftarrow G_1, \ldots, A\theta_n \leftarrow G_n\}$ is called a *partial deduction of A in P*.

We also introduce the notation $resultants(\tau)$ to stand for the set $\{A\theta_1 \leftarrow G_1, \ldots, A\theta_n \leftarrow G_n\}$. $resultants(\tau)$ is undefined if τ is an incomplete SLDNF tree consisting only of a single node.

If $\mathbf{A}$ is a finite set of atoms, then the *partial deduction of* $\mathbf{A}$ *in* P is the union of the partial deductions of the elements 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.

In [12] a fundamental theorem for correctness of partial deduction is presented. The two basic requirements for the 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 that might occur during the execution of the specialised program are covered by some definition. The following summarises the correctness result from [12]:

Definition 2.2 (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* if 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* if no pair of atoms in $\mathbf{A}$ have a common instance.

Theorem 2.3 (correctness of partial deduction)
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. $P' \cup \{\leftarrow G\}$ has an SLDNF-refutation with computed answer θ iff $P \cup \{\leftarrow G\}$ does.
2. $P' \cup \{\leftarrow G\}$ has a finitely failed SLDNF tree iff $P \cup \{\leftarrow G\}$ does.

Note that, if we want our specialised program to be correct for all goals containing instances of an independent subset of $\mathbf{A}$, the independence condition can always be guaranteed by a renaming transformation (see for instance [4,5,7], see also [1] for a dynamic renaming strategy). A small example is treated in the appendix A. Also note that renaming is not the only way to ensure independence. Another possibility is to use a generalisation operator like the *msg*[2]. For instance applying the *msg* to the dependent set $\{p(a, Y), p(X, b)\}$ yields the independent set $\{p(X, Y)\}$. This is not always a good idea because a lot of potential for specialisation might be lost due to the application of the *msg*. By using a renaming transformation we can avoid this loss of precision.

While guaranteeing the independence condition can be rather easy, guaranteeing the closedness condition is more difficult. Usually the set $\mathbf{A}$ has to be augmented in some way to guarantee closedness. This brings along the related problem of termination. In fact when performing partial deduction there are two termination problems:

- Each incomplete SLDNF tree that is constructed for some $P \cup \{\leftarrow A\}$ has to be kept finite. For solutions (which don't just simply impose an ad-hoc depth bound) to this non-trivial problem see for instance [3] and [14].

[2]most specific generalisation, also known as anti-unification or least general generalisation, see for instance [10]. Below we will see that the *msg* can play another role in the partial deduction process.

- The set $\mathbf{A}$ has to be kept finite while still ensuring the closedness condition. This can be guaranteed by a proper (well-founded) abstraction operator.

In this paper we address the second termination problem. Quite a few approaches presented in the literature so far do not address this problem (i.e. non-termination can occur, even when using the *msg* to ensure closedness and independence[3]). Others impose a finite number of atoms in $\mathbf{A}$ and use the *msg* to stick to that finite number. An example of the second approach is [15]. For a recent approach presenting a refined framwork ensuring global termination see [16].

However, as we have seen above, using the *msg* can induce a loss of precision. In this case the loss of precision can even be more dramatic, as the *msg* will 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 $solve(X)$. The basic purpose of this paper is to present an abstraction operator which does not show this dramatic loss of precision while still guaranteeing termination of the partial deduction process.

3 The General Algorithm

In this section we present a general algorithm for partial deduction. The following definitions are based on ideas from [4,5,7].

Definition 3.1 (unfolding rule)
An *unfolding rule* U is a function which given a program P and a goal G returns an incomplete SLDNF tree for $P \cup \{G\}$. A *well-founded unfolding rule* U is an unfolding rule such that for every program P and and every atomic goal $\leftarrow A$ it returns a SLDNF tree τ such that $resultants(\tau)$ is a *finite* set of resultants that is a partial deduction of A in P.
If $\mathbf{A}$ is a finite set of atoms and P is a program, then the set of resultants obtained by applying a well-founded unfolding rule U to each atom in $\mathbf{A}$ and taking the union of all resultants of the generated SLDNF trees is called the *partial deduction of* $\mathbf{A}$ *in* P *using* U.

One particular unfolding rule (which is well-founded given a depth bound) presented in [7] is based on unfolding only *determinate* sections (variations involving for instance "lookaheads" can be found in [4,5]). Basically this rule stops unfolding after a choice-point has been encountered. We will define the class of determinate unfolding rules as follows:

Definition 3.2 (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 *determinate* if for every program P and every goal G it returns a determinate SLDNF tree.

Determinate unfolding does not modify the backtracking behaviour of the original program (under the condition that the last unfolding step, which can be non-determinate, follows the computation rule of the underlying system). Further motivations behind determinate unfolding can be found in [4,5,7]. The definition of determinate unfolding can be improved by using lookaheads to detect resolvents which are "dead" (see [4,5]). We will return to this later.

Definition 3.3 (abstract)
An operation $abstract(S)$ is any operation satisfying the following conditions. Let S be a finite

[3]The classical example exhibiting non-termination for these approaches is the "reverse with accumulating parameter" program (see for instance [14].

set of atoms; then *abstract*(S) is a finite set of atoms $\mathbf{A}$ with the same predicates as those in S, such that every atom in S is an instance of an atom in $\mathbf{A}$.

Usually we want the *abstract* operation to be defined such as to ensure termination of the following generic algorithm.

Algorithm 3.4
Let $\mathbf{A}$ be an independent set of atoms for which we want to generate a specialised version of the program P. We suppose that we have an operation *abstract*$(.)$ and a well-founded unfolding rule U at our disposal.

1. Put $k = 0$ and $S_0 = \mathbf{A}$.
2. Generate the partial deduction of S_k in P using U.
3. Let $S_{k+1} = \mathit{abstract}(S_k \cup L_k)$ where L_k are the atoms occurring in the bodies of the partial deductions of step 2.
4. If $S_k = S_{k+1}$ (modulo variable renaming) then continue with step 5.
   Otherwise increment k and go to step 2.
5. Rename apart those atoms in S_k (and the atoms covered[4] by S_k in the bodies of the partial deductions) which have common instances with each other. Add a renaming definition for the renamed atoms in $\mathbf{A}$.

The following proposition is a consequence of the fundamental partial deduction theorem 2.3 and of the correctness of the renaming transformation (see for instance [5–7] and [1] for some correct renaming transformations).

Proposition 3.5 (correctness)
If algorithm 3.4 terminates then, for any goal G which is $\mathbf{A}$-closed, the specialised program P' obtained by replacing the set of clauses in P whose head contains one of the predicate symbols appearing in A_k by the partial deduction of A_k in P using U is correct with respect to $\leftarrow G$ in the following way:

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

In algorithm 3.4 and in the remainder of this paper we consider each predicate to be selectable in the partial deduction process. The above algorithm (and our technique in general) can easily be extended to account for un-selectable predicates (like for instance open-predicates for which we have no definition yet).

[4]If an atom in a body is covered by more than one atom in S_k then there is no unique choice of a covering atom. But the actual choice does not matter from a theoretical point of view (although it is usually a good idea to try to choose the most specific covering atom in S_k).

4 Abstraction Using Characteristic Trees

In this section we present a first attempt at defining a sensible abstraction operator for partial deduction and show its shortcomings. First though we need the following two definitions which are adapted from [4]. We adopt the convention in this paper that any derivation is potentially incomplete (a derivation can thus be failed, incomplete, successful or infinite).

Definition 4.1 (characteristic path)
Let G_1 be a goal and let P be a normal program whose clauses are numbered. Let $G_1, \ldots, G_n$ be an SLDNF derivation of $P \cup \{G_1\}$. The *characteristic path* of the derivation is the sequence $(l_1, c_1), \ldots, (l_{n-1}, c_{n-1})$, where l_i is the position of the selected literal in G_i, and c_i is defined as:

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

The set of all characteristic paths for a given program P and a given goal G will be denoted by *chpaths*(P, G). Note that if two non-failing derivations for two goals with the same number of literals (and for the same program P) have the same characteristic path then they are either both successful or both incomplete. This justifies the fact that the leaves of derivations are not registered in characteristic paths.

Definition 4.2 (characteristic tree)
Let G be a goal and P a normal program. Let U be an unfolding rule. Then *the characteristic tree* τ of G (in P) via U is the set of characteristic paths of the non-failing derivations of the incomplete SLDNF tree obtained by applying U to G (in P). We introduce the notation *chtree*$(G, P, U) = \tau$. We also say that τ is *a* characteristic tree if it is *the* characteristic tree for some G, P and U. A characteristic tree τ is *determinate* if it is the characteristic tree for some G, P and some determinate unfolding rule U.

Based on the idea of characteristic trees, Gallagher and Bruynooghe have developed abstraction operators in [7] and [4]. The basic idea is to generalise atoms, all having the same characteristic tree (or some similarly defined concept for [7]), by another atom, hopefully having the same characteristic tree. If this can be achieved, then we obtain an *almost perfect abstraction operator*, in the sense that due to the abstraction we have lost nothing wrt to the pruning and unfolding work done at partial deduction for the initial set of atoms. In that case the characteristic trees also play the role of controlling polyvariance[5]: for a set of (potentially dependent) atoms we will generate as many different versions as there are distinct characteristic trees.

The authors of article [7] actually claim in lemma 4.11 to have found an operator (namely *chcall*) which preserves a structure quite similar to the characteristic tree (of definition 4.2 above) in the case of definite programs. Unfortunately this lemma 4.11 is false (example 4.4 below can be used to show this, the details are presented in appendix B) and cannot be easily rectified.

A similar problem holds for the more general (i.e. not restricted to definite programs) abstraction operator of [4] (note however that no claim of the preservation of characteristic trees is made in [4]). In fact there are three problems with the abstraction operator as defined in [4]:

[5]Controlling when and how different versions for atoms with the same predicate should be generated.

1. It is not guaranteed to generate more general atoms (sometimes it even generates more specific atoms) causing potential problems for termination of the flow analysis of [4].
2. It can change the characteristic tree because of additional matching clauses.
3. It can change the characteristic tree because groundness and finite failure of negated literals can be modified.

Apart from reducing the precision of the approach, points 2 and 3 cause further problems for the termination proof of the flow analysis of [4] and therefore raise worries about the overall usefulness of characteristic trees. We will however see in the rest of our paper that the first two problems can be solved by incorporating a simple form of constraint into the partial deduction procedure yielding perfect precision wrt the characteristic trees. We will discuss the third problem (and sketch a solution) in section 8.

In the following three examples we will use the determinate unfolding rule of [4] (which in example 4.4 behaves slightly differently than definition 3.2). We take an example from [4] to illustrate the first problem.

Example 4.3
Take the following program P:

(1) $Eqlist([],[]) \leftarrow$
(2) $Eqlist([X|Xs],[X|Ys]) \leftarrow Eqlist(Xs,Ys)$

Let A be $\leftarrow Eqlist([1,2],W)$ and let B be $\leftarrow Eqlist(W,[3,4])$. Then A and B have the same characteristic tree {((1,2),(1,2),(1,1))}. It is however impossible to find an atomic goal C more general than A and B having the same characteristic tree. For instance the $msg(\{A,B\}) = \leftarrow Eqlist(X,Y)$ has a different characteristic tree: {((1,1)),((1,2))}. The reason being (as pointed out in [4]) that different arguments independently force the same choice during a derivation. The compromise proposed in [4] is to abstract A and B by $\leftarrow Eqlist([X,Y],[X,Y])$ which is however not more general than either A or B. This can cause potential problems for the termination[6] of the flow analysis in [4]).

The following example illustrates the second problem of [4], which implies that characteristic trees are not preserved in general by the method in [4].

Example 4.4
Let P be the following program (the actual definitions of $r(X)$ and $s(X)$ do not matter):

(1) $p(X) \leftarrow q(X)$
(2) $p(c) \leftarrow$
(3) $q(X) \leftarrow r(X)$
(4) $q(X) \leftarrow s(X)$
(5) $r(X) \leftarrow \ldots$
(6) $s(X) \leftarrow \ldots$

[6]Although we don't believe that a non-terminating example can actually be constructed for [4] — but finding a termination proof is probably a non-trivial task.

For this program the goals $\leftarrow p(a)$ and $\leftarrow p(b)$ have the same characteristic tree[7] {((1,1))}. The "abstract" operation of [4] (and in a similar way [7]) produces $\leftarrow p(X)$ as generalisation. This goal has a different characteristic tree {((1,1)),((1,2))} because it additionally matches the clause (2). Again there is no way to create a more general atom while still preserving the characteristic tree in the "standard" setting. .

We now illustrate the third problem of [4], which appears when negative literals are selected ([7] is restricted to SLD, so the problem does not appear there).

Example 4.5
Let us examine the following program P:

(1) $p(X) \leftarrow not(q(X))$
(2) $q(f(X)) \leftarrow q(X)$

For this program the goals $\leftarrow p(a)$ and $\leftarrow p(b)$ have the same characteristic tree {((1,1),(1,q))}. The abstraction operator of [4] will produce $\leftarrow p(X)$ as a generalisation which has the characteristic tree {((1,1))}. The problem is that the generalisation can abstract subterms inside a selected negative literal by a variable. The generalisation of the negative literal is thus no longer selectable by SLDNF. Solving this problem requires a more complicated form of constraint which will be sketched in section 8.

Note that a characteristic tree does not incorporate the failing branches of the underlying SLDNF tree. This poses no problem for purely determinate unfolding (as defined in definition 3.2). In fact if there is a failing branch then the goal fails completely and as we will see in section 7, such goals do not have to be abstracted. However the failing branches causes a further problem for [4] (in which the unfolding rule employs a "lookahead" and a slightly different definition of "determinacy") because generalisation can transform "dead" resolvents into "live" ones. The characteristic tree can thus be changed in still another way.[8] In the remainder of this presentation we will thus restrict ourselves to purely determinate unfolding as defined in definition 3.2. In section 8 we will discuss how this restriction can be lifted.

5 Negative Constraints

In this section we present the constraints which we will use in our partial deduction method to guarantee the preservation of characteristic trees in the definite case. As we have seen in the previous section an abstraction operator like the *msg* will change the characteristic tree of a given set of atoms because the generated abstraction might match additional clauses. The constraints presented in this section will be used to prune such additional clause matches.

Definition 5.1 (negative binding)
A *negative binding* is an expression $T_1 \not\sim T_2$ where T_1 and T_2 are terms (or atoms).

Intuitively $T_1 \not\sim T_2$ signifies that T_1 should not unify with T_2. A negative binding will be used to make sure that a selected positive literal T_1 of a goal $G = \leftarrow \ldots, T_1, \ldots$ does not

[7]Using definition 3.2 we could also obtain $\{((1,1),(1,3)),((1,1),(1,4))\}$. For the other atoms in this example definition 3.2 produces the same results.
[8]This was first pointed out by Bern Martens.

unify with the head T_2 of a clause $T_2 \leftarrow Body$ (which has been standardised apart). Also a negative binding $T_1 \not\sim T_2$ is *standardised apart* by standardising apart T_2 (i.e. by replacing all the variables in T_2 by new fresh variables not used "anywhere else"). We also introduce the notation $dom(T_1 \not\sim T_2) = vars(T_1)$. Note that in first order logic we could write $T_1 \not\sim T_2$ as $\neg\exists(T_1 = T_2')$ where T_2' is obtained from T_2 by standardising apart.

Definition 5.2 (applying substitutions)
Applying a substitution θ on negative bindings is defined by the rule: $(G \not\sim H)\theta = G\theta \not\sim H$

The rationale behind this definition is that H is always standardised apart before being used, i.e. any variable that occurs in θ will never occur inside the standardised apart version of H.

Definition 5.3 ($holds(T_1 \not\sim T_2)$)
A negative binding $T_1 \not\sim T_2$ is satisfied, denoted as $holds(T_1 \not\sim T_2)$, iff $\{T_1, T_2'\}$ are not unifiable where T_2' is obtained from T_2 by standardising the negative binding apart.

Definition 5.4 ($\theta\ sat\ T_1 \not\sim T_2$)
A substitution θ satisfies a negative binding n, written as $\theta\ sat\ n$, iff $holds(n\theta)$.[9]

Example 5.5
Take for instance $n = p(X) \not\sim p(f(a))$ and let $\theta = \{X/f(b)\}$. Then $n\theta = p(f(b)) \not\sim p(f(a))$ and $\theta\ sat\ n$ because the set $\{p(f(b)), p(f(a))\}$ is not unifiable. In a similar way we have that $\{X/g(Y)\}\ sat\ n$. Also any substitution will satisfy the negative binding $m = p(X,a) \not\sim p(Z,b)$. In particular we have $holds(m)$. In fact it can be easily proven that $holds(n) \Rightarrow \forall\theta : \theta\ sat\ n$.

Relating the concept of negative bindings to example 4.4, the goals $\leftarrow p(a)$ and $\leftarrow p(b)$ could be generalised by an expression of the form $(\leftarrow p(X), X \not\sim c)$. Goals of the form $\leftarrow p(X)\theta$ such that $\theta\ sat\ X \not\sim c$ are precisely the ones with the same characteristic tree as $\leftarrow p(a)$ and $\leftarrow p(b)$. To be able to prune more than just one matching clause we need sets of negative bindings:

Definition 5.6 (negative constraint)
A *negative (binding) constraint* is either the special symbol *fail* or a finite set of negative bindings.

Definition 5.7 ($holds(c)$)
A negative constraint c is satisfied, denoted as $holds(c)$, iff $c \neq fail$ and $T_1 \not\sim T_2 \in c \Rightarrow holds(T_1 \not\sim T_2)$

Definition 5.8 (applying substitutions)
Applying a substitution θ to the negative constraint *fail* yields *fail*. Applying a substitution θ to a negative constraint $\{n_1, \ldots, n_k\}$ consists in applying θ to each negative binding n_i.

Definition 5.9 ($\theta\ sat\ c$)
A substitution θ satisfies a negative constraint c, written as $\theta\ sat\ c$, iff $holds(c\theta)$.
We also introduce the notation $sat(c) = \{\theta \mid \theta \text{ sat } c\}$. We say that a constraint is *unsatisfiable* iff $sat(c) = \phi$ and *satisfiable* iff $sat(c) \neq \phi$. Note that *fail* is unsatisfiable. Furthermore two constraints c_1 and c_2 are equivalent, written as $c_1 \equiv c_2$, iff $sat(c_1) = sat(c_2)$

[9]Standardising apart the variables of T_2 in definition 5.3 ensures that something like $\theta = \{X/f(Z)\}$ does not satisfy $\{p(X) \not\sim p(Z)\}$ (which should be unsatisfiable).

Example 5.10
Take the constraints: $c_1 = \{p(X,a) \not\sim p(b,a)\}$, $c_2 = \{X \not\sim b\}$, $c_3 = \{p(X,a) \not\sim p(V,a)\}$ and $c_4 = \{p(X,X) \not\sim p(b,a)\}$. Then $c_1 \equiv c_2$, $c_3 \equiv \mathit{fail}$ and $c_4 \equiv \phi$ (i.e. c_4 imposes no constraints and $sat(c_4)$ is the set of all substitutions).

6 Extending SLD and SLDNF with Negative Constraints

In this section we will extend SLD and SLDNF to allow goals to be annotated with negative constraints as defined in the previous section. The extension will only be used during partial deduction and the negative constraints will not be present in the specialised program.

Below, we frequently need to express that, given two atoms or terms A and B, there exists a substitution θ such that $A = B\theta$. We refer to this as A is *matched* by B, with matching substitution θ and denote it as $match(A, B, \theta)$. This of course implies that A is an instance of B. Also by $mgu(S)$ we denote a function which returns an idempotent and relevant most general unifier of the set of atoms or terms S if S is unifiable. Otherwise $mgu(S)$ returns *fail*. The following lemma captures an important property of matching and can be used to decide (using matching) the satisfiability of negative bindings.

Lemma 6.1
Let A, B be two terms or atoms such that $vars(A) \cap vars(B) = \phi$ and such that a functor f not used inside A, B exists, then:
there exists a θ such that $match(A, B, \theta)$ iff for every substitution θ', such that θ' has no variables in common with B, we have $mgu(\{A\theta', B\}) \neq \mathit{fail}$.
Note that the if-part also holds if there is no functor f not used inside A, B.

Based on the above lemma we introduce a normalisation operator $||c||$ for constraints which can be used to decide whether a given negative constraint c is satisfiable or not.

Definition 6.2 ($||c||$)
Let c be a negative constraint and let c' be obtained from c by standardising apart all its negative bindings. We define the following procedure which calculates the *solved form* $||c||$ of c.[10]

1. Remove any $T_1 \not\sim T_2 \in c'$ such that $mgu(T_1, T_2) = \mathit{fail}$.
2. If there exists a $T_1 \not\sim T_2 \in c'$ such that $match(T_1, T_2, \theta)$ then let $||c|| = \mathit{fail}$.

Proposition 6.3
Let c be a negative constraint. Then $||c||$ is a negative constraint and $c \equiv ||c||$.

Proposition 6.4
If c is a negative constraint and there exists some functor f not occurring inside c then
c is satisfiable $\Leftrightarrow$ $||c|| \neq \mathit{fail}$.

This means that satisfiability is a decidable concept if, for every constraint c under consideration, we can always find some functor f not used in c. This is always the case if the set of

[10]Note that there is in general no unique solved form.

functors is infinite. Also note that it is quite easy to show that the above proposition 6.4 also holds if the number of constants is infinite (or if there is a sufficient number of constants not occurring inside of c).

In the case that the number of functors and the number of constants cannot be assumed to be "sufficiently" large for proposition 6.4 to hold we can still use the concepts introduced in this section and the partial deduction method based on them. The only drawback is that unsatisfiability of constraints might not be detected at the earliest possible moment. This just adds unnecessary resultants to the specialised program, but poses no problems with respect to the correctness of the specialised program.

Example 6.5
This example illustrates why we need to have some functor f (or a sufficient number of constants) not occurring inside c in proposition 6.4. Take the constraint $c = \{p(X) \not\sim p(0), p(X) \not\sim p(s(Z))\}$. We have that $||c|| = c \neq \textit{fail}$, but c is nonetheless unsatisfiable if the Herbrand universe is just $\{0, s(0), s(s(0)), \ldots\}$.

Definition 6.6 (constrained atoms and goals)
A *constrained atom* (resp. *constrained goal*) is a couple consisting of an ordinary atom A (resp. goal G) and a negative constraint c.

Abusing notation we will sometimes denote (A, ϕ) by A. Intuitively the constraint c imposes restrictions on the values the variables in A (resp. G) can take. A constrained atom (resp. goal) can thus be seen as representing a (possibly infinite) set of valid atoms (resp. goals). For instance, given the Herbrand Universe $U_P = \{0, s(0), s(s(0)), \ldots\}$, the constrained atom $(p(X), \{p(X) \not\sim 0\})$ represents the set of atoms $\{p(s(X))\theta \mid \theta \text{ is any substitution}\}$. More generally:

Definition 6.7 (valid instances)
Let (A, c) be a constrained atom (resp. goal). The set of *valid instances* of (A, c) is defined as follows: $\textit{valid}((A, c)) = \{A\theta \mid \theta \textit{ sat } c\}$

Definition 6.8 (deriving constrained goals)
Let G be the ordinary goal $\leftarrow A_1, \ldots, A_m, \ldots, A_n$ and C be the renamed clause $A \leftarrow B_1, \ldots, B_q$. Also let c be a negative constraint. Then the constrained goal (G', c') is derived from (G, c) and C using mgu θ if the following conditions hold:

- A_m is an atom, called the *selected* atom, in G.
- θ is an mgu of A_m and A.
- G' is the goal $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_n)\theta$.
- $c' = ||c.\theta|| \neq \textit{fail}$.

What we have defined is a CLP language with a particular constraint domain, namely the set of negative constraints, and with a particular satisfiability operation, namely one based on calculating the solved form $||c||$ (see [8] for a survey on constraint logic programming). Note however that our constraints are never used to infer values of variables. This is a consequence of the fact that a negative constraint is either unsatisfiable or has an infinite number of substitutions satisfying it (assuming that some functor not used inside the constraint exists). In fact the constraints are just used to prune the search tree whenever it is guaranteed that no valid instance of the current goal exists.

Based on definition 6.8 we can define the concepts of SLDC-derivations, SLDC-refutation and SLDC-trees. Similarly we can easily extend SLDNF into SLDNFC to allow constraints[11]. We can also extend the notions of partial deduction, unfolding rules and determinate unfolding rules.

Definition 6.9 (instance)
A constrained goal (resp. atom) (G, c) is an instance of another constrained goal (resp. atom) (G', c'), denoted by $(G, c) \preceq (G', c')$, iff $\mathit{valid}((G, c)) \subseteq \mathit{valid}((G', c'))$.

Example 6.10
Suppose we have a constrained goal $C = (\leftarrow p(X), \{p(X) \not\sim p(f(g(Z)))\})$.
Then the goals $(\leftarrow p(a), \phi)$, $(\leftarrow p(f(h(Y))), \phi)$ and $(\leftarrow p(f(V)), \{p(f(V)) \not\sim p(f(g(Z)))\})$ are instances of C, while the goals $(\leftarrow p(f(V)), \phi)$ and $(\leftarrow p(f(g(W))), \phi)$ are not. Also it is always true that $(G, c) \preceq (G, \phi)$, or more generally $(G, c \cup c') \preceq (G, c)$.

7 Preserving Characteristic Trees

We can easily adapt definition 4.1 and definition 4.2, of characteristic paths and trees respectively, by just replacing SLDNF by SLDNFC. Note that the set of characteristic paths for a normal goal G under SLDNF is identical to the set of characteristic paths for the constrained goal (G, ϕ) under SLDNFC. For simplicity of the presentation we will restrict ourselves to SLDC and to determinate unfolding rules. In the discussion we will sketch a way to lift this restriction.

We showed in examples 4.3 and 4.4 that without constraints it is impossible to generalise atoms while still preserving their characteristic trees. Let us revisit examples 4.4 and 4.3 and see how we can achieve preservation of characteristic trees using constraints.

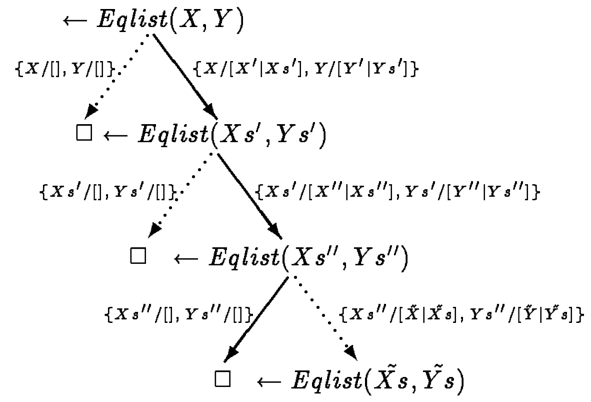
Example 7.1
We can abstract the goals $\leftarrow p(a)$ and $\leftarrow p(b)$ of example 4.4 by the (more general) constrained goal $(\leftarrow p(X), c')$ with $c' = \{p(X) \not\sim p(c)\}$ having the same characteristic tree $\tau = \{((1,1))\}$. Note that the substitution $\{X/c\}$ makes c' unsatisfiable and thereby the additional match with clause (2) is pruned.

Example 7.2
Again let A be $\leftarrow \mathit{Eqlist}([1,2], W)$ and let B be $\leftarrow \mathit{Eqlist}(W, [3,4])$ who both have the characteristic tree $\{((1,2),(1,2),(1,1))\}$. We can calculate the *msg* of A and B and choose constraints in such a way as to obtain the same characteristic tree. For this example we would get the abstraction $(\leftarrow \mathit{Eqlist}(X, Y), \{c_1, c_2, c_3\})$ with
$c_1 = \mathit{Eqlist}(X, Y) \not\sim \mathit{Eqlist}([], [])$, $c_2 = \mathit{Eqlist}(X, Y) \not\sim \mathit{Eqlist}([X'], [Y'])$,
$c_3 = \mathit{Eqlist}(X, Y) \not\sim \mathit{Eqlist}([X', X'', \tilde{X} | \tilde{Xs}], [Y', Y'', \tilde{Y} | \tilde{Ys}])$.
Note that $\{A, B\} \subset \mathit{valid}((\leftarrow \mathit{Eqlist}(X, Y), \{c_1, c_2, c_3\}))$. The constraints are constructed in such a way as to prune any extra branches in the characteristic tree. Figure 1 illustrates how this can be done: the constraints are chosen in such a way as to prune all the dotted resolution steps. For instance the constraint c_1 prunes the left-most dotted resolution step. The details of the calculation will be presented later in this section. We can now construct a specialised program for the constrained goal $(\leftarrow \mathit{Eqlist}(X, Y), \{c_1, c_2, c_3\})$, which covers both the call A and the call B (the clause (3) below replaces clauses (1) and (2) of example 4.3):

[11]Note that constraints do not have to be propagated for negative literals because negative literals have to be ground before they get executed.

(3) $Eqlist([V,W],[V,W]) \leftarrow$

This effectively solves one of the (potential) termination problems of [4], because we can always generate atoms which are more general and have the same characteristic tree. We will also show later that the constraints do not introduce a termination problem of their own.



$\leftarrow Eqlist(X,Y)$
$\{X/[], Y/[]\}$
$\{X/[X'|Xs'], Y/[Y'|Ys']\}$
$\square \leftarrow Eqlist(Xs',Ys')$
$\{Xs'/[], Ys'/[]\}$
$\{Xs'/[X''|Xs''], Ys'/[Y''|Ys'']\}$
$\square \quad \leftarrow Eqlist(Xs'',Ys'')$
$\{Xs''/[], Ys''/[]\}$
$\{Xs''/[\tilde{X}|\tilde{Xs}], Ys''/[\tilde{Y}|\tilde{Ys}]\}$
$\square \quad \leftarrow Eqlist(\tilde{Xs},\tilde{Ys})$

Figure 1: Calculating pruning constraints for example 7.2

Definition 7.3 (admissible)
Let τ be a characteristic tree. We say that τ is *admissible* for a constrained goal G (and for a program P) iff $\tau \subseteq chpaths(P,G)$.

Note that τ is admissible for G does not necessarily imply that τ is *the* characteristic tree for G via some U. However it signifies that τ can be obtained by pruning branches of some characteristic tree for G via some U. This is exactly what we will try to do below in definition 7.5 through the calculation of pruning constraints.

Proposition 7.4
Let the constrained goal (G,c) be an instance of (G',c') and let G be an instance of G'.
If τ is a characteristic tree admissible for (G,c) which involves no negative selected literals then τ is also admissible for (G',c').

Definition 7.5 (pruning constraint)
Let τ be a non-empty determinate characteristic tree admissible for the constrained goal $(\leftarrow A, \phi)$. Then we define the pruning constraint $prune(A,\tau)$ in the following way:
$prune(A,\tau) = \{A \not\sim A\theta \mid \exists p \in chpaths(P,(\leftarrow A,\phi))$ with $p \notin \tau$ such that $p = p'(l_n,c_n)$ and $p'p'' \in \tau$ where p', p'' are possibly empty, H is the head of the clause numbered with c_n, L is the selected literal at position l_n used in the last resolution of p and θ is the composition of the mgu's of the derivation associated with p (including the last step $(l_n,c_n))\}$.

Note that the above definition does not apply to empty characteristic trees. Indeed it is in general not possible, by just using conjunctions of negative bindings[12], to construct pruning

[12]It might be possible using disjunctions of negative bindings or maybe the constraints of the form $not_P(A)$ sketched in section 8, but the authors have not yet investigated this.

constraints valid for empty characteristic trees which at the same time satisfy propositions 7.6 and 7.7. Note however that atoms with empty characteristic trees do not pose any problem for termination of algorithm 3.4 because their partial deductions 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.

The constraints for $\leftarrow$Eqlist(X, Y) in example 7.2 and figure 1 were calculated by the above method. The next proposition establishes the correctness of this method.

Proposition 7.6 (Preservation of characteristic trees)
Let U be a determinate unfolding rule, let τ be *the* characteristic tree for some instance[13] of $(\leftarrow A, \phi)$ via U and let τ be non-empty and admissible for $(\leftarrow A, \phi)$ then:
τ is *the* characteristic tree of $(\leftarrow A, prune(A, \tau))$ for some determinate unfolding rule U'.

Note that, in the above proposition, we cannot conclude that τ is *the* characteristic tree of $(\leftarrow A, prune(A, \tau))$ for U because nothing prevents U from treating that goal completely differently (i.e. selecting different atoms) from the goal $(\leftarrow A, c)$. To avoid this kind of arbitrary behaviour we need a feature of "stability" of the unfolding rule. For instance a 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 $chtree((\leftarrow A, prune(A, \tau)), P, U) = \tau$. Also note that in the case that an unfolding rule does not exhibit this kind of "stability" we can still use the pruning constraints of definition 7.5 and simply impose τ as the characteristic tree of the generalisation. In both cases this 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.

Proposition 7.7
Let τ be the non-empty determinate characteristic tree of $(\leftarrow A\theta, c)$ for a determinate unfolding rule U and let τ be admissible for $(\leftarrow A, \phi)$ then $(\leftarrow A\theta, c)$ is an instance of $(\leftarrow A, prune(A, \tau))$.

This implies that any constrained atom more general than $(\leftarrow A, prune(A, \tau))$ has either a different characteristic tree or has a more general ordinary goal component. Note that the proposition does not hold for unfolding rules which are not determinate (the problem is due to failing branches, which are not represented in the characteristic tree). The proposition also implies that it can be quickly determined through characteristic trees whether something is surely an instance of a constrained atom $(\leftarrow A, prune(A, \tau))$. This leads to the following definition:

Definition 7.8 (cover)
Let P be a program, U a determinate unfolding rule, τ a characteristic tree admissible for $(\leftarrow A, \phi)$ and let (A, c) be a constrained atom where $c \equiv prune(A, \tau)$. We then define:
$cover((A, c)) = \{(A\theta, c') \mid chtree((\leftarrow A\theta, c'), P, U) = \tau\}$.
Under the conditions of proposition 7.6, this relation is a decidable and sound approximation (i.e. subset) of the set of constrained atoms $\{A' \mid A' \preceq (A, c)\}$ and will be used in our partial deduction algorithm to test whether atoms occurring in the bodies of partial deductions are covered or not.

We are now in a position to formally define our abstraction operator. This abstraction operator is only defined if no negative literals are selected during partial deduction. In the next section we will discuss how this restriction can be lifted.

[13] A small example, showing the importance of this requirement, can be found in appendix D.

Definition 7.9 (abstract)
Let P be a program, U a determinate and well-founded unfolding rule and let S be a set of constrained atoms. Then our abstraction operator is defined as follows:
$abstract(S) = \{(A, prune(A, \tau)) \mid A = msg(A_\tau) \text{ for some } \tau \neq \phi\} \cup S_\phi$ where
$A_\tau = \{A \mid \exists(A, c) \in S \text{ with } chtree((\leftarrow A, c), P, U) = \tau\}$ and
$S_\phi = \{(A, c) \mid (A, c) \in S \text{ and } chtree((\leftarrow A, c), P, U) = \phi\}$.

We can now define a variant (for SLDC instead of SLD) of algorithm 3.4 by using the abstract operation of definition 7.9 and by testing closedness (required for step 5 of algorithm 3.4) using the *cover* relation of definition 7.8 (which is applicable because any element of the set S_k with a non-empty characteristic tree will be of the form $(A, prune(A, \tau))$). Note that the specialised program, obtained by taking resultants and by renaming, is a SLD program *without constraints*. The constraint manipulation has already been incorporated by pruning branches and the closedness guarantees that no additional constraint processing is needed[14].

Proposition 7.10
If the set of different characteristic trees is finite then the *abstract* operation of definition 7.9 guarantees that (an adaption of) algorithm 3.4 terminates.

To make the previous proposition applicable we can simply enforce a depth bound on the unfolding rule used during partial deduction. Note that when performing determinate unfolding a depth bound is much less likely to have an (ad-hoc) effect. Also by using a depth bound we guarantee the well-foundedness of the unfolding rule. It is however also possible not to impose any (ad-hoc) depth-bound on the unfolding rule and to just change the definition of characteristic paths and trees such that it incorporates a depth bound.

The following simple example shows the kind of optimisations that are possible with our method.[15]

Example 7.11

(1) $rev(X, none, X) \leftarrow$
(2) $rev([], Acc, Acc) \leftarrow$
(3) $rev([X|T], Acc, Res) \leftarrow rev(T, [X|Acc], Res)$

For the query $\leftarrow rev(L, [], R)$ and using determinate unfolding our method will generate the following sequence of atoms to be partially deduced (see algorithm 3.4):
$S_0 = \{(rev(L, [], R), \phi)\}$
$S_1 = abstract(S_0 \cup \{(rev(T, [X], R), \phi)\}) = \{(rev(L, A, R), \{rev(L, A, R) \not\sim rev(X, none, X)\})\}$
$S_2 = abstract(S_1 \cup \{(rev(T, [X|A], R), \{rev([X|T], A, R) \not\sim rev(X, none, X)\})\}) = S_1$
All the atoms above have the characteristic tree $\tau = \{(1,2), (1,3)\}$. The following specialised program is produced:

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

[14]Note though that it is always correct to use a predicate definition generated for $p(X)$ for any call to a constrained goal $(\leftarrow p(X), c)$.
[15]This example looks artificial, but it is not easy to come up with a small example exhibiting at the same time non-termination problems and substantial loss of precision when using abstraction solely based on the *msg*.

If we treat the above example with a method based on dynamic renaming (see [1]) the set of partially deduced atoms will grow infinitely: $\{rev(L,[],R), rev(L,[X],R), rev(L,[X,X'],R),\ldots\}$. If we use a strategy without renaming, partial deduction (alone[16]) will not be able to remove the clause (1) from the specialised program. If we use the static renaming strategy of [2] it is in this case possible to remove the clause (1) while guaranteeing termination. But we obtain a larger program (because of unnecessary polyvariance due to the static guidance) and if we replace clause (1) by for instance "$rev(X,[Y|none],X) \leftarrow$" the strategy will no longer be able to remove clause (1) (unless the static renaming generates 3 versions of the predicate rev, for which we can again find another example which requires 4 versions,...). Similarly the abstraction operators in [4,7] cannot adequately handle the above example (or slight variations thereof). For instance the *chcall* operator of [7] modifies the characteristic tree: $chcall(rev(L,[],R)) = rev(L,A,R)$. The generalised atom now additionally matches clause (1).

8 Extending the Method

The abstraction operator we have presented in the previous section is only guaranteed to preserve the characteristic trees if the *msg* replaces no subterm inside a selected negative literal by a variable. For instance in example 4.5 the selected negative literals $\neg q(a)$ and $\neg q(b)$ get abstracted by $\neg q(X)$ thereby making it impossible to preserve the characteristic trees for the goals $\leftarrow p(a)$ and $\leftarrow p(b)$.

We can solve this problem by adding a basic constraint of the form $not_P(A)$ where A is an (ordinary) atom. We would then say that $not_P(A)$ is satisfied, written as $holds(not_P(A))$, iff the atom A is ground and $P \cup \{\leftarrow A\}$ has a finitely failed SLDNF tree.

The satisfiability of these constraints is no longer decidable. This means that we have to use a safe-approximation of satisfiability. We also have to extend SLDNFC such that, when a negative literal gets selected, it has to be checked whether a given negation constraint implies finite failure of the negated atom. Further modifications are also needed ot link variables of a constraint of the form $not_P(A)$ to variables in the top-level goal.

The solution for example 4.5 would then be to abstract $\leftarrow p(a)$ and $\leftarrow p(b)$ by the constrained goal $\leftarrow (p(X), \{not_P(q(X))\})$. The characteristic tree of this constrained goal would indeed be $\{((1,1),(1,q))\}$ and the following optimal specialisation, valid for the goals $\leftarrow p(a)$ and $\leftarrow p(b)$, gets constructed:

(3) $p(X) \leftarrow$

The method presented in the previous chapter was restricted to determinate unfolding rules. If we move to non-determinate unfolding rules we have a complication due to the appearance of failing branches. When generalising, these failing branches might become non-failing and make the pruning constraints method not directly applicable (the method will generate unsatisfiable constraints). A solution might be to incorporate failing branches into the characteristic tree structure and to use exactly the same definition of pruning constraints. But this is not fully satisfactory wrt the control of polyvariance — different versions of the same predicate, with just a different failing behaviour, might get generated. In ongoing work we attempt to use a disjunction of negation and negative constraints to solve this problem.

[16] An abstract interpretation phase could detect that clause (1) will never get used in this case, it may not work in general (for instance if we want to specialise the above program for the query $\leftarrow rev(L,[],R), rev(X,Y,Z)$).

9 Discussion and Conclusion

Note that our concepts of negative bindings and negative constraints are somewhat related to the concepts of sets of term inequalities in [13]. However, the constraints in [13] are used to specify an infinite set of substitutions. Another similar kind of constraints has been used in [11] to define a generalisation operator for machine learning which respects a set of counter examples. Our constraint programming language also resembles the CLP($\mathcal{FT}$) language with *dif* constraints in [17, 18]. An adaptation of our technique might yield a useful abstraction operator for CLP($\mathcal{FT}$).

In conclusion, we have presented an almost perfect abstraction operator for partial deduction. The operator ensures termination and correctness (through the closedness and independence conditions of [12]) of the partial deduction while preserving (almost) all of the pruning and unfolding performed on the initial set of atoms. The abstraction operator also provides (almost) perfect control of polyvariance, in the sense that different specialised versions for calls are only constructed when this is reasonable, i.e. only when the atoms involved have different characteristic trees.

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. His relevant remarks significantly helped to shape this report, eliminate errors and make it more readable. We would also like to thank John Gallagher for his helpful remarks.

References

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

[2] 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.

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

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] J.-L. Lassez and K. Marriott. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3:301–317, 1987.

[12] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[13] 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, 1989*, pages 461–476. MIT Press, 1989.

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

[15] 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.

[16] B. Martens and J. Gallagher. Ensuring global termination of partial deduction while allowing flexible polyvariance. Technical Report CSTR-94-16, Computer Science Department, University of Bristol, 1994.

[17] 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.

[18] 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.

A Small Renaming Example

Let $\mathbf{A}$=$\{q(X), p(a,X), p(X,a)\}$ and take the following program P:

$q(X) \leftarrow p(a,X)$
$q(f(X)) \leftarrow p(X,a)$
$p(a,X) \leftarrow$

We could have the following (not very sophisticated) partial deduction of **A** in P (the first row contains the atoms to be partially deduced, the second row contains their respective partial deductions):

$q(X)$	$p(a,X)$	$p(X,a)$
$q(X) \leftarrow p(a,X)$ $q(f(X)) \leftarrow p(X,a)$	$p(a,X) \leftarrow$	$p(a,a) \leftarrow$

We can see that the partial deduction of P wrt **A** has an SLDNF-refutation with computed answer $\{X/a\}$ for the query $\leftarrow q(X)$ whereas the original program does not have such an SLDNF-refutation. This is due to the fact that **A** is not independent (note that $P \cup \{\leftarrow q(X)\}$ is **A**-closed). Now suppose we want our specialised program to be correct for all goals which contain instances of the independent subset $A' = \{q(X), p(a,Y)\}$ of **A**. In that case we can rename all predicates in $\mathbf{A} \backslash A'$. In this case we would rename $p(Z,a)$ to $p'(Z,a)$ for instance, yielding the following specialised (and correct) program P':

$q(X) \leftarrow p(a,X)$
$q(f(X)) \leftarrow p'(X,a)$
$p(a,X) \leftarrow$
$p'(a,a) \leftarrow$

Note that P' does not have an (incorrect) SLDNF-refutation with computed answer $\{X/a\}$ for the query $\leftarrow q(X)$.

B Counter Example

In this appendix we present a counter example to lemma 4.11 on page 326 of [7]. Note that the definitions differ from the ones in [4] and from the ones adopted in our paper (for instance what is called a *chpath* in [7] corresponds more closely to the concept of a characteristic tree in our paper than to the notion of a characteristic path).

We take the program P from example 4.4 (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 [7] we have that $chpath(A) = (\langle c_1 \rangle, \{c_3, c_4\})$. According to definition 4.10 we obtain: $chpaths(A) = \{\langle c_1, c_3 \rangle, \langle c_1, c_4 \rangle\}$.

The most general resultants (definition 4.6 of [7]) of the paths in $chpaths(A)$ is the set $\{p(Z) \leftarrow r(Z), p(Z) \leftarrow s(Z)\}$.

By definition 4.10 of [7] we obtain the *characteristic call* of A:
$chcall(A) = msg\{p(Z), p(Z)\} = p(Z)$.

In lemma 4.11 of [7] 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.

C Specialisation Example

Let the original program P to be specialised be:

(1) $rev(X, none, X) \leftarrow$
(2) $rev([], Acc, Acc) \leftarrow$
(3) $rev([X|T], Acc, Res) \leftarrow rev(T, [X|Acc], Res)$

Let us follow step by step the execution of algorithm 3.4 for the query $\leftarrow rev(L, [], R)$:

$$S_0 = \{(rev(L, [], R), \phi)\}$$

$$\begin{tikzcd}
& (\leftarrow rev(L,[],R),\phi) \arrow[dl, "(2)"] \arrow[dr, "(3)"'] & \\
(\Box,\phi) & & (\leftarrow rev(T,[X],R),\phi)
\end{tikzcd}$$

The characteristic tree for $(\leftarrow rev(L, [], R), \phi)$ is $\{(1, 2), (1, 3)\}$ and the partial deduction is:

$rev([], [], []), \leftarrow$ (under the constraint ϕ)
$rev([X|T], [], R) \leftarrow rev(T, [X], R)$ (under the constraint ϕ)

The constrained atoms occurring in the bodies of the partial deductions are thus

$$L_0 = \{(rev(T, [X], R), \phi)\}$$

The constrained goal $(\leftarrow rev(T, [X], R), \phi)$ has the same characteristic tree $\tau = \{(1, 2), (1, 3)\}$ as $(\leftarrow rev(L, [], R), \phi)$. We thus calculate: $msg(\{rev(L, [], R), rev(T, [X], R)\}) = rev(L, A, R)$ and by calculating the pruning constraints $prune(rev(L, A, R), \tau)$ we obtain:

$$S_1 = abstract(S_0 \cup \{(rev(T, [X], R), \phi)\}) =$$
$$= \{(rev(L, A, R), \{rev(L, A, R) \not\sim rev(X, none, X)\})\}$$

$$\begin{tikzcd}
& (\leftarrow rev(L,A,R), \{rev(L,A,R) \not\sim rev(X,none,X)\}) \arrow[dl, "(2)"] \arrow[dr, "(3)"'] & \\
(\Box, \{rev([],A,A) \not\sim rev(X,none,X)\}) & & (\leftarrow rev(T,[X|A],R), \{rev([X|T],A,R) \not\sim rev(X,none,X)\}\})
\end{tikzcd}$$

The characteristic tree of $(\leftarrow rev(L,A,R), \{rev(L,A,R) \not\sim rev(X,none,X)\})$ is still $\{(1,2),(1,3)\}$ and the partial deduction is:

$rev([],A,A) \leftarrow$ (under the constraint $\{rev([],A,A) \not\sim rev(X,none,X)\} \equiv \phi$)
$rev([X|T],A,R) \leftarrow rev(T,[X|A],R)$ (under the constraint$\{rev([X|T],A,R) \not\sim rev(X,none,X)\})\})$

The atoms occurring in the bodies of the partial deductions are thus:

$$L_1 = \{(rev(T,[X|A],R), \{rev([X|T],A,R) \not\sim rev(X,none,X)\})\}$$

The constrained goal $(\leftarrow rev(T,[X|A],R), \{rev([X|T],A,R) \not\sim rev(X,none,X)\})$ also has the characteristic tree $\{(1,2),(1,3)\}$ and is an instance of $(\leftarrow rev(L,A,R), \{rev(L,A,R) \not\sim rev(X,none,X)\})$ (by proposition 7.7). We have thus reached the fixpoint in our algorithm:

$$S_2 = abstract(S_1 \cup \{(rev(T,[X|A],R), \{rev([X|T],A,R) \not\sim rev(X,none,X)\})\}) = S_1$$

The following specialised program is produced, which is (amongst others) correct for all atomic goals which are valid instances of $(\leftarrow rev(L,A,R), \{rev(L,A,R) \not\sim rev(X,none,X)\})$, e.g. it is correct for $\leftarrow rev(L,[],R)$:

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

D Example for Proposition 7.6

This example shows the importance of the requirement that "τ is *the* characteristic tree for some instance of $(\leftarrow A, \phi)$" in proposition 7.6. Let P be the following program:

(1) $p(X) \leftarrow q(Z)$
(2) $q(a) \leftarrow$
(3) $q(b) \leftarrow$

Although the characteristic tree $\tau = \{((1,1),(1,2))\}$ is admissible for the goal $(\leftarrow p(X), \phi)$ there exists no instance G of $(\leftarrow p(X), \phi)$ such that τ is a characteristic tree of G and indeed there exists no constraint c such that τ is the characteristic tree of $(\leftarrow p(X), c)$. In particular $prune(p(X), \tau)$ will yield the unsatisfiable constraint $\{p(X) \not\sim p(X)\}$. The problem is caused by the "existential" variable Z in clause (1) on which we can place no constraint. Note however that this is not a problem for our method because, by construction, τ of proposition 7.6 will always be a characteristic tree of some more instantiated goal.