A General Criterion for Avoiding Infinite Unfolding during Partial Deduction*

Maurice BRUYNOOGHE, Danny De SCHREYE and
Bern MARTENS**

*Department of Computer Science,
Katholieke Universiteit Leuven,
Celestijnenlaan 200A, 3001 Heverlee, Belgium.*

Received 15 July 1991

Revised manuscript received 6 April 1992

Abstract Well-founded orderings are a commonly used tool for proving the termination of programs. We introduce related concepts specialised to SLD-trees. Based on these concepts, we formulate formal and practical criteria for controlling the unfolding during the construction of SLD-trees that form the basis of a partial deduction. We provide algorithms that allow to use these criteria in a constructive way. In contrast to the many ad hoc techniques proposed in the literature, our technique provides both a formal and practically applicable framework.

Keywords: Logic Programming, Partial Deduction, Termination

§1 Introduction

Since its introduction in logic programming by Komorowski in 1981,²⁰⁾ partial evaluation has attracted the attention of many researchers in the field. Some, e.g. Refs. 36), 37) and 29) have addressed pragmatic issues related to the impurities of Prolog. Others, e.g. Refs. 17), 23), 28), 32) and 33) were attracted by the perspective of eliminating the overhead associated with meta interpreters. Finally, the subject was put on a firm theoretical basis in Ref. 24).

This paper takes the viewpoint of Ref. 24), that the specialized program should have the same answers as the original one. An approach which has more appropriately been called partial deduction in Ref. 21), leaving the term partial evaluation for works taking into account the non-logical features of Prolog and

* all authors are supported by the Belgian National Fund for Scientific Research.

** work partially supported by ESPRIT BRA COMPULOG (project 3012).

the order in which answers are produced, as e.g. Ref. 29).

One of the components in a partial deduction procedure for logic programs, e.g. Refs. 3) and 26), is a procedure computing a partial deduction of an atom A , given an SLD-tree for $\leftarrow A$ with a program P . Constructing the SLD-tree is the topic of this paper. One has to avoid constructing an infinite one and systems described in the literature use quite disparate criteria for assuring this, while often requiring some human assistance. We have developed a general criterion for avoiding the construction of an infinite SLD-tree. It is rooted in well-known techniques for proving termination of programs, which are based on well-founded sets, as developed e.g. by Refs. 15) and 25). We show that existing approaches can be seen as instances of our criterion and also that it can be integrated in an algorithm for the automatic construction of the SLD-tree.

In this paper, we take the approach of pushing the unfolding of the program to the limits, while still preserving finiteness of the SLD-tree. The limits are set by the constraint that some form of data-consumption must be made at each unfolding. Of course, this is not always desirable, as it can lead to an explosive growth of the program and unfolding not always improves performance. These other concerns are not pursued in the current paper.

The paper is organized as follows. In Section 2, we introduce some basic concepts on SLD-trees, well-foundedness and the relationship between them. Here, we define the notions of a well-founded SLD⁻-tree and —more generally—subset-wise founded SLD⁻-tree. In Section 3 we show how these formal notions can be used operationally to compute strongly expanded, but finite SLD-trees. Section 4 deals with further issues of applicability. On the one hand, we illustrate how the approach can be seen as a framework, in the sense that existing techniques to control unfolding in the context of partial deduction can easily be simulated with it. On the other hand, we show that the approach can be automated. We end with a discussion.

§2 Well-Founded and Subset-Wise Founded SLD-Trees

We assume familiarity with the basic concepts of logic programming and, for simplicity, we restrict ourselves to (pure) definite programs. Following Ref. 24), we extend the notion of SLD-tree by allowing it to be *incomplete*. This means that the leaves of the SLD-tree are not restricted to be either success or failure nodes, but that they can be any goal statement. A further extension that will be useful for several algorithms is the following:

Definition 2.1

An SLD⁻-tree is a subtree of an SLD-tree having the same root.

An SLD⁻-tree differs from an SLD-tree in the sense that in some branching points not all branches may be included. For any SLD⁻-tree, τ , we denote by τ^+ the unique minimal SLD-tree containing it. τ^+ is obtained from τ by adding to τ one single derivation step from each missing branch (originating

from a non-leaf) in τ .

We do recall two basic definitions on partial deduction from Ref. 24).

Definition 2.2 (adapted from Ref. 24))

Let P be a definite program, A an atom, and $\leftarrow A, G_1, \dots, G_n$, $n > 0$, an SLD-derivation for $P \cup \{\leftarrow A\}$. Let $\theta_1, \dots, \theta_n$ be the corresponding sequence of substitutions and let G_n be $\leftarrow A_1, \dots, A_m$.

We say that $A\theta_1\dots\theta_n \leftarrow A_1, \dots, A_m$ is the *resultant* of the derivation $\leftarrow A, G_1, \dots, G_n$.

We will call an SLD⁻-tree *finite* if all its derivations are finite. Observe that for a finite program P , a finite SLD⁻-tree is also a finite tree, in the sense that its set of nodes is finite. This is clear, since the tree is finitely branching and therefore Königs lemma (see e.g. Ref. 14)) can be applied.

Definition 2.3 (adapted from Ref. 24))

Let P be a definite program, A an atom and τ a finite SLD-tree for $P \cup \{\leftarrow A\}$. Let $\{G_i | i = 1, \dots, r\}$ be the (non-root) leaves of the non-failing branches of τ and $\{R_i | i = 1, \dots, r\}$ the resultants corresponding to the derivations $\{\leftarrow A, \dots, G_i | i = 1, \dots, r\}$. The set $\{R_i | i = 1, \dots, r\}$ is called a *partial deduction* for A in P .

It is clear that, given a program P , an atomic goal $\leftarrow A$ and a computation rule R for $P \cup \{\leftarrow A\}$, there can be (possibly infinitely) many different SLD-trees τ for $P \cup \{\leftarrow A\}$ under R . These are all subtrees of the complete SLD-tree τ_0 . The problem addressed in this paper is how to control the construction of an SLD-tree that forms a suitable basis for partial deduction. More precisely we provide general, but practically applicable criteria to ensure that the construction terminates.

In what follows, we frequently make use of strict order relations, denoted $>$. A strict order relation is an anti-reflexive, anti-symmetric and transitive binary relation. A strictly ordered set, $S, >$, will be called an s-poset, the corresponding order, $>$, an s-order.

Given an SLD⁻-tree τ , an s-poset can be associated to it in a natural way, by taking the goals in the tree as the set and the tree structure itself as a strict ordering on this set. For technical reasons, the actual definition of the associated s-poset $G_\tau, >_\tau$ will be slightly more complicated. Let τ_0 be the complete SLD-tree of which τ forms part. We assume a numbering on the nodes of τ_0 (e.g. left-to-right, top-down and breadth-first). $G_\tau = \{(G, i) | G \text{ is a goal of } \tau \text{ having } i \text{ as its associated number in } \tau_0\}$. The s-order is defined as $(G, i) >_\tau (G', j)$ if node i is an ancestor of node j in τ_0 .

Definitoin 2.4

An s-poset $S, >$ is said to be *well-founded* if there is no infinite sequence of elements $(s_1, s_2, \dots)$ in S , such that $s_i > s_{i+1}$, for all $i \geq 1$.

Definition 2.5

Let $S, >_s$ be an s-poset. A *well-founded measure*, f , on $S, >_s$ is a monotonic function from $S, >_s$ to some well-founded set $W, >_w$.

As discussed in e.g. Ref. 12), well-founded sets are a commonly used tool for proving termination of programs. Ref. 15) used them to prove termination of imperative programs, Ref. 25) for proving termination of production systems. We apply them in the context of SLD-tree construction.

Definition 2.6

An SLD⁻-tree τ is *well-founded* if there exists a well-founded measure f on $G_\tau, >_\tau$.

Paraphrasing a theorem of Ref. 25), we can formulate the following theorem.

Theorem 2.1

An SLD⁻-tree is finite iff it is well-founded.

Proof

Let τ be the SLD⁻-tree. For the if-part, assuming that τ has an infinite derivation, we can construct a sequence $(G_0, i_0) >_\tau (G_1, i_1) >_\tau \dots$ in $G_\tau, >_\tau$. Applying the well-founded measure f to it, we get an infinite sequence $f(G_0, i_0) >_w f(G_1, i_1) >_w \dots$ in $W, >_w$. For the only-if-part, since $G_\tau, >_\tau$ is a well-founded s-poset itself, take f equal to the identity function, 1_{G_τ} . $\square$

Although in this section the emphasis is on providing declarative characterisations of finite SLD-trees, leaving the operational use of these characterisations for the next section, we already point out how the two will be linked, using the simple well-founded SLD⁻-tree characterisation as an example. Operationally, we start off by fixing any function $f: \tau_0 \rightarrow W, >_w$, where $W, >_w$ is some well-founded s-poset. Such a function is called a *prefounding for τ_0* . For an initial $\tau = \{(\leftarrow A, 1)\}$, the restriction of f to the domain τ , $f|_\tau$, is obviously monotonic. Then, we control the construction of the SLD⁻-tree, τ , by imposing that each unfolding must preserve monotonicity of $f|_\tau$. The resulting SLD⁻-tree will be the largest SLD⁻-subtree τ of τ_0 such that $f|_\tau$ is monotonic. Clearly, it is well-founded through f and therefore finite. This explains the terminology *prefounding for τ_0* . The final finite SLD-tree produced as a basis for the partial deduction is the SLD-tree τ^+ corresponding to τ .

Of course, the problem of obtaining *sensibly* expanded SLD-trees has now been shifted to that of finding sensible functions on τ_0 . Although for each finite SLD-subtree of τ a corresponding function on τ_0 exists, it may be hard to generate useful functions that lead to sensible SLD-trees. Therefore, we introduce more refined characterisations of finite SLD-trees, in which several measures are combined. Each measure will focus on a separate part of the recursive inference expressed in the SLD-tree. As a result, it will be more easy to generate

them.

The inspiration for these more general measures comes from certain criteria which seem reasonable in practice to control unfolding. When building an SLD-tree, we tend to compare the subgoal selected for unfolding with the selected subgoals in the ancestor goals of the same derivation. Especially if two such subgoals are calls to the same predicate, a comparison may allow to avoid loops in the derivation. Within the context of our approach, this corresponds to the observation that it is not really necessary to have a measure that decreases at each derivation step, but that the measure should decrease between each two goals in the same derivation, where a subgoal of a same recursive predicate was selected for unfolding.

A second motivation to allow more liberal measures comes from the application of partial deduction to meta interpreters. Typically, in vanilla-type meta interpreters,

```
solve(true).
solve((A, B)):- solve(A), solve(B).
solve(A):- clause(A, B), solve(B).
```

we are not really interested in comparing the measure assigned to two goals, unless the selected subgoals are both of type `solve(A)`. More in particular, one almost never wants to stop the unfolding of a subgoal of type `solve((A, B))` or `solve(true)`. Therefore, measures assigned to goals where such subgoals are selected should not interfere with the control of the unfolding. Better still, no measure should be attached to them.

These two considerations lead to the following generalized notion of foundedness.

Definition 2.7

An SLD-tree τ is *subset-wise founded* if

- (1) There exists a finite number of sets, $C_0, \dots, C_N$, such that $G_\tau = \bigcup \{C_i \mid i \leq N\}$.
- (2) For each $i = 1, \dots, N$, there exists a well-founded measure $f_i: C_i, >_\tau \rightarrow W_{i, >_i}$.
- (3) For each pair (G, k) in C_0 and each derivation D in τ containing (G, k) :
 - either D is finite
 - or there exists a descendent (G', j) of (G, k) in D such that $(G', j) \in C_i$, for some $i > 0$.

This definition makes both of the observations above explicit:

- Not every two goals in a same derivation must be comparable. The measure only needs to decrease if the goals belong to the same set C_i .
- Some goals (the ones in C_0) can be disregarded completely, if we can make sure that they have a descendent within the same derivation, which

is either contained in a set with measure, C_i , $i > 0$, or which is a leaf.

Example 2.1

Our notational conventions are: variables start with a lower case character, constants with a capital. The dot notation $x.y$ is used to represent a list with head x and tail y .

Consider the following simple program, defining the relation $\text{contains_pal}/2$ where $\text{contains_pal}(x, y)$ holds if y is an initial sublist of a list x and y is a palindrome.

$\text{contains_pal}(x, y) \leftarrow \text{i_sublist}(y, x), \text{palindrome}(y).$

$\text{i_sublist}(\text{Nil}, z) \leftarrow$

$\text{i_sublist}(h.t, h.s) \leftarrow \text{i_sublist}(t, s).$

$\text{palindrome}(p) \leftarrow \text{reverse}(p, p).$

$\text{reverse}(u, v) \leftarrow \text{reverse_acc}(u, \text{Nil}, v).$

Fig. 1 Subset-wise founded SLD-tree for $\leftarrow \text{contains_pal}(3.x, y).$

reverse_acc(Nil, acc, acc) $\leftarrow$
reverse_acc(k.n, acc, m) $\leftarrow$ reverse_acc(n, k.acc, m).

Let:

$C_0 = \{(G, k) \mid R(G, k) \text{ is a call to either contains_pal/2, palindrome/1 or reverse/2 and } k \in \mathbb{N}\} \cup \{(\square, k) \mid k \in \mathbb{N}\},$
 $C_1 = \{(G, k) \mid R(G, k) \text{ is a call to i_sublist/2 and } k \in \mathbb{N}\},$
 $C_2 = \{(G, k) \mid R(G, k) \text{ is a call to reverse_acc/3 and } k \in \mathbb{N}\}.$

For the functions f_i , $i = 1, 2$, let

$f_1: C_1 \rightarrow \mathbb{N}$, $f_1(G, k)$ = the number of functor symbols in the second (input) argument of $R(G, k)$ (which is the input argument of i_sublist),

$f_2: C_2 \rightarrow \mathbb{N}$, $f_2(G, k)$ = the number of functor symbols in the first argument of $R(G, k)$ (the input argument of reverse_acc).

Finally, assume that R , in each goal, selects the left-most atom for which at least one argument position is a non-variable (in the example, every goal contains at least one such atom). A maximally large subset-wise founded SLD⁻-tree τ for $\leftarrow \text{contains_pal}(3.x, y)$, with respect to the given pair of sets and functions and this computation rule is drawn in Fig. 1. Selected atoms under R are underlined.

Observe that every well-founded SLD⁻-tree τ is subset-wise founded. Just take $N = 1$, $C_1 = G_\tau$, $C_0 = \phi$, $f_1 = f$.

Theorem 2.2

An SLD⁻-tree is finite iff it is subset-wise founded.

Proof

Let τ be the SLD⁻-tree. If τ is finite, then it is well-founded and therefore subset-wise founded. Conversely, let τ be subset-wise founded and assume that it is infinite. Since τ contains an infinite derivation D and G_τ is included in the union of $C_0, \dots, C_N$, there exists a set C_i , such that $C_i \cap D$ is infinite. On $C_1, \dots, C_N$ we have a well-founded measure. Thus, i must be 0.

Now, since D is infinite, by Definition 2.7, for each (G, k) in $C_0 \cap D$, there exists a (G', j) in some $C_i \cap D$, $i > 0$, such that $(G, k) >_\tau (G', j)$. Because $C_0 \cap D$ is infinite and strictly decreasing in $G_\tau, >_\tau$, this allows us to construct an infinite number of elements in $\bigcup \{C_i \mid i > 0\}$. Thus, one of the sets $C_i \cap D$ must be infinite, which contradicts the existence of f_i . $\square$

Notice that as a consequence, the notions of a finite, a well-founded and a subset-wise founded SLD⁻-tree coincide. This makes us return to the question why it is useful to introduce the subset-wise founded measures at all. The reason is that interesting subsets of goal statements can be defined and that simple well-founded measures can be provided on them. This is especially useful if we aim to use the given theorems operationally, in the sense that we first define sets

and measures, and consecutively construct the SLD^- -tree under the constraints imposed by the measures. Finding sensible and useful well-founded measures on the entire tree is often more complex.

§3 Constructing Finite SLD-Trees Using Well-Founded Measures

In this section, we illustrate how the definitions of Section 2 can be applied operationally to construct useful SLD-trees. We introduce two algorithms and illustrate their use with some simple examples. Some issues on complete automation are postponed until Section 4.

Throughout the section, P denotes a definite program, $\leftarrow A$ a definite goal with one atom, R a computation rule for $P \cup \{\leftarrow A\}$ and τ_0 the complete SLD-tree for $P \cup \{\leftarrow A\}$ under R .

3.1 Finite Prefoundings

As announced in the previous section, the operational use of both characterizations of finiteness is based on the notion of *prefoundings*. The definition of a *finite prefounding*, $((C_0, \dots, C_N), (f_1, \dots, f_N))$ for τ_0 , below, is essentially the same as the conditions imposed on the pair $((C_0, \dots, C_N), (f_1, \dots, f_N))$ in the definition of a subset-wise founded SLD^- -tree τ , except that:

- τ is replaced by τ_0 (in particular, $(C_0, \dots, C_N)$ is a covering of τ_0),
- $f_i: C_i \rightarrow W_{i, > i}$, $i > 0$, is not required to be monotonic.

Given such a prefounding, it will again be easy to construct a subset-wise founded SLD^- -tree τ , by unfolding every goal for as long as all functions f_i , $i > 0$, remain monotonic on the restricted domains $C_i \cap \tau$.

Definition 3.1

A pair $((C_0, \dots, C_N), (f_1, \dots, f_N))$ is a *finite prefounding* for τ_0 , if

- (1) Each C_i , $i \leq N$, is a set of pairs (G, k) , such that G is a goal with predicate symbols in $P \cup \{\leftarrow A\}$, $k \in \mathbb{N}$ and $G_{\tau_0} \subseteq \bigcup \{C_i \mid i \leq N\}$.
- (2) For each $i > 0$, f_i is a function, $f_i: C_i \rightarrow W_{i, > i}$, where $W_{i, > i}$ is a well-founded s-poset.
- (3) For each pair (G, k) in C_0 and each derivation D in τ_0 containing (G, k) :
 - either D is finite
 - or there exists a descendent (G', j) of (G, k) in D , such that $(G', j) \in C_i$, for some $i > 0$.

In order to give an example of a finite prefounding, we first introduce a class of practical measure functions.

Definition 3.2

Let Term denote the set of terms in the first order language used to define the theory P . The *functor norm* is a function $|\cdot|: \text{Term} \rightarrow \mathbb{N}$ defined as:

If $t = f(t_1, \dots, t_n)$, $n > 0$
 then $|t| = 1 + |t_1| + \dots + |t_n|$
 else $|t| = 0$

The functor norm counts the number of functors in a given term. Alternatively, we could have chosen to count the number of constants as well. However, we do not believe that this leads to an improved measure function with respect to termination properties.

Definition 3.3

Let p be a predicate of arity n and $S = \{a_1, \dots, a_m\}$, $1 \leq a_k \leq n$, $1 \leq k \leq m$, a set of argument positions for p . The *functor measure* with respect to p and S is a function:

$$\begin{aligned} |\cdot|_{p,S} : \{A \mid A \text{ an atom with predicate symbol } p\} &\rightarrow \mathbb{N}, \text{ defined as:} \\ |\cdot|_{p,S}(p(t_1, \dots, t_n)) &= |t_{a_1}| + \dots + |t_{a_m}|. \end{aligned}$$

Example 3.1

We provide a natural example of a finite prefounding $((C_0, \dots, C_N), (f_1, \dots, f_N))$. The given program P contains only a finite number of recursive predicates $p_1, \dots, p_N$. Given the computation rule R , we define:

$$\begin{aligned} C_i &= \{(G, k) \mid G \text{ a goal such that } R(G, k) \text{ is a call to predicate } p_i \text{ and} \\ &\quad k \in \mathbb{N}\}, \text{ for } i > 0, \\ C_0 &= \{(G, k) \mid G \text{ a goal such that } R(G, k) \text{ is a call to a non-recursive} \\ &\quad \text{predicate and } k \in \mathbb{N}\} \cup \{(\square, k) \mid k \in \mathbb{N}\} \end{aligned}$$

Clearly, $G_{\tau_0} \subseteq \bigcup \{C_i \mid i \leq N\}$. Also, if (G, k) is in a derivation D of τ_0 , then either D is finite or there is a descendent (G', j) of (G, k) in D , such that $R(G', j)$ is an atom with a recursive predicate (independent of whether or not G is in C_0). So, the conditions (1) and (3) of Definition 3.1 are fulfilled.

The functor measure induces in a natural way functions f_i on the classes C_i , $i > 0$. For each class C_i , let S_i be a set of argument positions of interest for the predicate p_i . Essentially, these will be the input arguments for p_i (in Section 4 we discuss how they can be determined in an automated way). Then, define f_i to be $|\cdot|_{p_i, S_i}$.

More concretely, the pair $((C_0, C_1, C_2), (f_1, f_2))$ of Example 2.1 is a particular instance of this generic example. Here, $f_1 = |\cdot|_{\text{isublist}, \{2\}}$ and $f_2 = |\cdot|_{\text{reverse_acc}, \{1\}}$.

Given a finite prefounding, $((C_0, \dots, C_N), (f_1, \dots, f_N))$ the following algorithm computes a finite SLD⁻subtree τ of τ_0 , which is subset-wise founded with respect to the s-posets $C_0 \cap \tau, >_\tau, \dots, C_N \cap \tau, >_\tau$ and the functions $f_1, \dots, f_N$ restricted to these s-posets. The basic criterion for unfolding used in the algorithm is that each of the functions $f_1, \dots, f_N$ should invariantly remain monotonic on its domain.

Algorithm 3.1**Initialisation**

$$\tau = \{ \{ (\leftarrow A, 1) \} \},$$

$$\text{Terminated} = \phi$$

While there exists a derivation $D \in \tau$ and $D \notin \text{Terminated}$ **do**

Let (G, k) be the leaf of D ,

Let $\text{Derive}(G, k)$ be the set of all direct $>_{\tau_0}$ -descendents,
 (G', j) of (G, k) in G_{τ_0} ,

Let $\text{Decrease}(G, k)$ be the set of all $(G', j) \in \text{Derive}(G, k)$, such that
 for all $i > 0$, such that $(G', j) \in C_i$,
 for all $(G'', 1) \in D \cap C_i$:
 $f_i(G', j) <_i f_i(G'', 1)$,

If $\text{Decrease}(G, k) = \phi$

then add D to Terminated

else replace τ by $\tau \setminus D \cup \{D \cup \{(G', j)\} \mid (G', j) \in \text{Decrease}(G, k)\}$

Endwhile

Example 3.2

For the `contains_pal` program of Example 2.1 and using the finite preordering of Example 3.1, the SLD⁻-tree produced by Algorithm 3.1 is the one in Fig. 1. The corresponding SLD-tree, τ^+ , contains one additional node,

$$\leftarrow \text{i_sublist}(k_4.n_4, x), \text{reverse_acc}(n_4, k_4.3.\text{Nil}, 3.k_4.n_4)$$

which is a child of

$$\leftarrow \text{i_sublist}(t_1, x), \text{reverse_acc}(t_1, 3.\text{Nil}, 3.t_1)$$

with a unification expressed in the equations $\{t_1 = k_4.n_4, 3.\text{Nil} = \text{acc}_4, 3.t_1 = m_4\}$.

A partial deduction algorithm could produce the following new clauses on the basis of τ^+ :

$$\begin{aligned} \text{contains_pall}(3.x, \text{Nil}) &\leftarrow \\ \text{contains_pall}(3.x, 3.\text{Nil}) &\leftarrow \\ \text{contains_pall}(3.x, 3.k_4.n_4) &\leftarrow \text{sublist}(k_4.n_4, x), \text{reverse_acc}(n_4, k_4.3. \\ &\quad \text{Nil}, 3.k_4.n_4). \end{aligned}$$
Proposition 3.1

Algorithm 3.1 terminates. The resulting finite SLD⁻-tree τ is subset-wise founded with respect to the sets $C_0 \cap \tau, \dots, C_N \cap \tau$ and the well-founded measures $f_1, \dots, f_N$ restricted to these sets.

Proof

Due to similar arguments as in the proof of Theorem 2.2, the algorithm terminates. We omit the details. To prove subset-wise foundedness, observe that by the first condition of Definition 3.1, the sets $C_0 \cap \tau, \dots, C_N \cap \tau$ cover τ , as required by the first condition of Definition 2.7. For each of the s-posets $C_i \cap \tau, i > 0$, the corresponding f_i is monotonic by construction, so condition (2) of Definition 2.7 is satisfied. The third and last condition of Definition 2.7 is identical to the third condition in Definition 3.1 on $C_0, \dots, C_N$. $\square$

Alternatively, we could formulate a completely similar algorithm, such that the resulting SLD⁻-tree is subset-wise founded, with the exception of its leaves. This is achieved by performing one additional derivation step in each derivation for which the stopping criterion is met.

The choice between these alternatives can be related to a trade-off on the level of the synthesized program (see Ref. 26) for a partial deduction algorithm based on well-founded SLD-trees), as is illustrated in the following simple example.

Example 3.3

Let P be the `append/3` program,

```
append(Nil, l, l) ←
append(h.t, k, h.l) ← append(t, k, l).
```

We use the finite prefounding of Example 3.1, with $N = 1, C_1 = \{(G, i) \mid R(G, i) \text{ has predicate symbol } \text{append/3}\}, C_0 = \{(\square, i) \mid i \in \mathbb{N}\}$ and $f_1 = \cdot|_{\text{append/3, l1}}$. $A = \text{append}(1.2.x, y, z)$ and R is any computation rule. Using Algorithm 3.1, only 2 derivation steps are performed (leading up to the first branching point in τ_0). Introducing a new predicate `append1/3`, the following new clause could be generated:

```
append1(1.2.x, y, 1.2.l2) ← append(x, y, l2).
```

An alternative algorithm, systematically performing one unfolding beyond well-foundedness with respect to $((C_0, C_1), (f_1))$ could produce the new clauses:

```
append2(1.2.Nil, y, 1.2.y) ←
append2(1.2.h3.t3, y, 1.2.h3.l3) ← append(t3, y, l3).
```

The example is representative for the kind of difference that the resulting programs produced by the two approaches will reveal in general. In the case of a variant of Algorithm 3.1, one more unfolding is performed after all 'input' has been consumed. Therefore, more new clauses are generated, distinguishing between the different clauses of the given program P that are applicable to the one-but-last goal in the tree.

3.2 Application to Meta Interpreters

Our second motivation for introducing both the concept of a subset-wise founded SLD⁻-tree and that of a finite prefounding is the application of partial deduction to meta interpreters. As recognized in Refs. 32) and 22), it is certainly desirable that if partial deduction is applied to meta interpreters, it should at least perform the parsing task. This results in the somewhat ad hoc rule that for interpreters of the vanilla-type (see Section 2), calls to solve/1, where the argument is a conjunction of multiple object level atoms, should always be unfolded. As opposed to this, for calls to solve/1 with an argument containing only one object level atom, the control of the unfolding should be based on properties of the atom itself.

This idea fits well within the context of a finite prefounding. Assume that $p_1/n_1, \dots, p_N/n_N$ are the recursive predicates of the object program. We introduce the sets

$$C_1 = \{(G, k) \mid R(G, k) \text{ is an atom of type solve}(p_i(t_1, \dots, t_{n_i})), \\ \text{where } t_j, j = 1, \dots, n_i \text{ are any terms, and } k \in \mathbb{N}\}$$

$$C_0 = \{(G, k) \mid R(G, k) \text{ is an atom of type solve}(p(t_1, \dots, t_n)), \\ \text{where } p/n \text{ is a non-recursive predicate and } t_j, j = 1, \dots, n \\ \text{terms, and } k \in \mathbb{N}\} \\ \cup \{(G, k) \mid R(G, k) \text{ is an atom of type solve}((A, B)) \text{ and} \\ k \in \mathbb{N}\} \\ \cup \{(G, k) \mid R(G, k) \text{ is an atom of type clause}(A, B) \text{ and} \\ k \in \mathbb{N}\} \\ \cup \{(\square, k) \mid k \in \mathbb{N}\}$$

Again, functor measures are appropriate for the functions f_i on these sets, although we would want more expressivity in the sense that subterms of arguments should be measured instead of just arguments. This is achieved by allowing selector functions.

Definition 3.4

A selector function s is denoted as a finite, non-empty sequence of positive integers connected with slashes, $n_1/n_2/\dots/n_k$. Let Atom be the set of all atoms, the selector function s is a function: $\text{Atom} \cup \text{Term} \rightarrow \text{Term}$, recursively defined as:

If $s = n$ and $n \leq m$
then $s(r(t_1, \dots, t_m)) = t_n$
else if $s = n_1/n_2/\dots/n_k$, $n_1 \leq m$ and
 $n_2/\dots/n_k(t_{n_1})$ is defined,
then $s(r(t_1, \dots, t_m)) = n_2/\dots/n_k(t_{n_1})$
else $s(r(t_1, \dots, t_m))$ is undefined.

It is straightforward to generalize Definition 3.3 of a functor measure with

respect to a predicate p and a set of argument positions S to that of a functor measure with respect to p and a set of selector functions S .

Then, given a set of argument positions, S_i , of interest for each recursive predicate p_i/m_i of the object program, let $S_i' = \{1/a_1, \dots, 1/a_k\}$, where $S_i = \{a_1, \dots, a_k\}$. Define

$$f_i = \lambda \cdot \lambda_{\text{solve}, S_i'}, i = 1, \dots, N$$

In view of full automation of partial deduction applied to meta interpreters, several other issues need to be addressed, such as:

- how to recognize meta interpreters,
- how to distinguish in general between the meta predicates that perform the parsing and the arguments within them that contain the object program goals,
- how to avoid loops for those predicates in the meta interpreter that perform tasks different from the parsing.

We return to these issues in Section 4.

3.3 Hierarchical Prefoundings

The class of finite prefoundings introduced in Example 3.1, although rather easy to generate automatically and practical for many examples, have one clear disadvantage. We illustrate the problem with the following simple program.

Example 3.4

```

sums(Nil, Nil) ←
sums(h.t, sumall.sumt) ← sum(h.t, sumall), sums(t, sumt).

sum(Nil, 0) ←
sum(n.m, s) ← sum(m, sm), s is sm + n.

```

The predicate $\text{sums}(\text{list1}, \text{list2})$ yields true, if list2 is the list of all sums that can recursively be made for the numbers in list1 , the numbers in its tail, and so on. Let A be equal to $\text{sums}(1.2.\text{Nil}, x)$ and assume that the natural finite prefounding $((C_{\text{sums}}, C_{\text{sum}}, C_0), (\lambda \cdot \lambda_{\text{sums}, \{1\}}, \lambda \cdot \lambda_{\text{sum}, \{1\}}))$, where $C_0 = \{(G, i) \mid R(G, i) \text{ has predicate symbol is/2}\} \cup \{(\square, i) \mid i \in \mathbb{N}\}$, is used.

The subset-wise founded SLD⁻-tree produced by Algorithm 3.1 under the standard computation rule is shown in Fig. 2.

It is clear that the unfolding is stopped too early. The reason is that the f_{sum} -value for the atom $\text{sum}(2.\text{Nil}, \text{sumall}_2)$ is compared with the f_{sum} -value for $\text{sum}(\text{Nil}, \text{sm}_2)$, which is 0. However, although both atoms have the same predicate symbol, it would be more appropriate not to compare them, because they are not recursive descendents of each other within the proof tree for $\leftarrow$

Fig. 2 Subset-wise founded SLD-tree for $\leftarrow \text{sums}(1.2.\text{Nil}, x)$.
The leaf (between brackets) is not included.

$\text{sum}(1.2.\text{Nil}, x)$.

The problem occurs systematically when we have clauses of the type:

$$\begin{aligned} p(\dots) &\leftarrow q(\dots), p(\dots). \\ q(\dots) &\leftarrow q(\dots). \end{aligned}$$

or

$$p(\dots) \leftarrow p(\dots), p(\dots).$$

In both cases, the SLD-tree τ_0 contains derivations with selected subgoals for the same predicate that are part of different branches in the proof tree. Clearly, many familiar logic programs are of this type (e.g. naive reverse, permutation sort, quick sort, n-queens).

For the sums-program, the problem can easily be solved by providing a different finite prefounding. We simply replace the set C_{sum} by two new sets:

$$C_{\text{sum},1} = \{(G, i) \in C_{\text{sum}} \mid R(G, i) \text{ is a descendent of } \leftarrow \text{sum}(1.2.\text{Nil}, \text{sumall}_1) \text{ in the proof tree}\}$$

$$C_{\text{sum},2} = \{(G, i) \in C_{\text{sum}} \mid R(G, i) \text{ is a descendent of } \leftarrow \text{sum}(2.\text{Nil}, \text{sumall}_2) \text{ in the proof tree}\}$$

However, not only is this impractical in view of automation, but the reason why it works in this example is that τ_0 is finite. In general, if we want to cover τ_0 with sets C_i , such that goals with selected subgoals originating from different branches of the proof tree are placed in different sets C_i , then we need an infinite number of such sets. This motivates the introduction of the concept *hierarchical prefounding*. Before giving its definition, we introduce some additional terminology. With slight abuse of notation, in what follows, we will occasionally refer to pairs (G, i) as goals or nodes in τ or τ_0 (instead of G_τ or G_{τ_0}).

Definition 3.5

Let $(G, i) = (\leftarrow A_1, \dots, A_j, \dots, A_n, i)$ be a node in an SLD⁻-tree τ , let $R(G, i) = A_j$ be the call selected by the computation rule R , let $H \leftarrow B_1, \dots, B_m$ be a clause whose head unifies with A_j and let $\theta = \text{mgu}(A_j, H)$ be the most general unifier. (G, i) has a son (G', k) in τ , $(G', k) = ((\leftarrow A_1, \dots, A_{j-1}, B_1, \dots, B_m, A_{j+1}, \dots, A_n) \theta, k)$. Let (G'', l) be a descendent of (G', k) in τ with $R(G'', l) = B_i \theta \psi$, for some $i \leq m$ and ψ the composition of all mgu's on the subderivation from (G', k) to (G'', l) . We say that $B_i \theta \psi$ in (G'', l) is a *direct descendent* of A_j in (G, i) and that A_j in (G, i) is a *direct ancestor* of $B_i \theta \psi$ in (G'', l) .

Stated otherwise, the selected atoms of two goals are in the direct descendent relation if, in the proof tree associated to τ , the two nodes labeled by these atoms are connected by an arc.

Definition 3.5 (Continued)

The binary relations *descendent* and *ancestor*, defined on atoms in goals, are the transitive closures of the direct descendent and direct ancestor relations respectively. For A an atom in (G, i) and B an atom in (G', j) , A is an ancestor of B is denoted as $A >_{\text{pr}} B$ (pr stands for proof tree).

The relations descendent and ancestor on pairs of atoms in goals of τ induce in a natural way descendent and ancestor relations on τ . Let (G, i) and (G', k) be in τ . We say that (G, i) is a *proper ancestor* of (G', k) , if $(G, i) >_{\tau_0} (G', k)$ and $R(G, i) >_{\text{pr}} R(G', k)$. By abuse of notation, we denote it as $(G, i) >_{\text{pr}} (G', k)$.

The notion of a *hierarchical prefounding for τ_0* , defined below, differs from a finite prefounding in essentially two ways: First, it is more specific in the sense that, instead of classifying (with possible overlaps) the goals of τ_0 into a finite number of sets C_i , we now initially partition the selected subgoals of the goals in τ_0 into a finite number of sets R_k and, in a second phase and for each R_k , we classify (with possible overlaps) the goals with a selected subgoal in R_k into a number of sets C_i . In this second phase, the classification is completely deterministic. Two goals (G, n) and (G', m) of R_k are placed in a same C_i , $i > 0$,

if one of them is a proper ancestor of the other (in other words, if either $(G, n) >_{pr} (G', m)$ or $(G', m) >_{pr} (G, n)$). Second, in another respect, hierarchical prefoundings are more general than finite ones, because the number of sets C_i is allowed to be infinite.

Definition 3.6

A pair $((C_0, C_1, C_2, \dots), (f_1, f_2, \dots))$ is a *hierarchical prefounding* for τ_0 , if:

- (1) There exists a finite partition $R_0, \dots, R_N$ of $R_{\tau_0} = \{R(G, i) \mid (G, i) \in \tau_0\}$, such that for all $(G, i) \in \tau_0 \setminus C_0$:

$$C_i = \{(G, i)\} \cup \{(G', j) \in \tau_0 \mid \exists k: 1 \leq k \leq N, \text{ such that } R(G, i), R(G', j) \in R_k \text{ and } (G, i) >_{pr} (G', j) \text{ or } (G', j) >_{pr} (G, i)\}$$

$$C_0 = \{(G, i) \in \tau_0 \mid R(G, i) \in R_0\} \cup \{(\square, i) \mid i \in \mathbb{N}\}.$$
- (2) There exist a finite number of well-founded sets $W_1, >_1, \dots, W_N, >_N$, such that for each $(G, i) \in \tau_0$ with $R(G, i) \in R_k$, f_i is a function, $f_i: C_i \rightarrow W_k$, and for all $i, j > 0$, $f_i|_{C_i \cap C_j} = f_j|_{C_i \cap C_j}$.
- (3) C_0 contains no infinite sequence $(G_{i_1}, i_1) >_{pr} (G_{i_2}, i_2) >_{pr} \dots$, such that for all $m, m \geq 1$, $R(G_{i_m}, i_m)$ in (G_{i_m}, i_m) is a direct ancestor of $R(G_{i_{m+1}}, i_{m+1})$ in $(G_{i_{m+1}}, i_{m+1})$.

Before giving an example, a comment on the relation between the third condition above and condition (3) in Definition 3.1 seems appropriate. Without providing an explicit definition of the notion of a proof tree, the condition above could be reformulated as follows:

For each pair (G, k) in C_0 and for each branch B in the proof tree associated to τ_0 containing $R(G, k)$:

- either B is finite
- or there exists a descendent (G', j) of (G, k) with $R(G', j)$ a descendent of $R(G, k)$ in B , such that $(G', j) \in C_i$, for some $i > 0$.

Observe that this formulation is completely similar to the one in condition (3) of Definition 3.1, with derivation D in τ_0 replaced by branch B of the corresponding proof tree.

Example 3.5

Assume that the recursive predicates of P are $p_1, \dots, p_N$. Define:

$$R_k = \{R(G, i) \in R_{\tau_0} \mid R(G, i) \text{ is a call to } p_k\}, \text{ where } k > 0,$$

$$R_0 = \{R(G, i) \in R_{\tau_0} \mid R(G, i) \text{ is a call to a non-recursive predicate}\}.$$

As a consequence, C_0 is equal to the set C_0 of Example 3.1. For each node $(G, i) \in \tau_0 \setminus C_0$, C_i is the set of all the ancestors and descendents of (G, i) , such that R selects a call from them to the same predicate as it does from (G, i) and that their selected atom is a $>_{pr}$ -ancestor or descendent of the atom selected from (G, i) . Now assume that the functions f_i are functor measures on the selected atom of a goal (G', j) of C_i . Then, it should be clear that, in order to decide on the

unfolding of $R(G, j)$, the ancestor goals of (G, j) in C_i (and their selected atoms) are precisely what we want to compare with.

We refer to Example 3.6 for an explicit instance of the class of examples introduced here.

Definition 3.6 is slightly more general than Example 3.5. This is because in general (e.g. when dealing with meta interpreters), more refined partitions of R_{τ_0} may be useful (see Sections 3.2 and 4.3).

It is clear that the pair of potentially infinite sequences $((C_0, C_1, \dots), (f_1, \dots))$ cannot be considered as a practical specification of the prefounding. In particular, it can not be used as input to an algorithm computing finite SLD-trees. Instead, hierarchical prefoundings will be specified through a finite partition $R_0, \dots, R_N$ of R_{τ_0} , in addition to functions $F_1, \dots, F_N$, such that

$$F_k: \{(G, i) \in \tau_0 \mid R(G, i) \in R_k\} \rightarrow W_{k, >_k},$$

where $W_{k, >_k}$ is a well-founded s-poset.

Observe that due to condition (2) of Definition 3.6, we can associate such functions $F_k, k > 0$, to any given hierarchical prefounding $((C_0, C_1, \dots), (f_1, \dots))$. More precisely, for any (G, i) with $R(G, i) \in R_k$, $F_k(G, i) = f_i(G, i)$. Conversely, the pair $((R_0, \dots, R_N), (F_1, \dots, F_N))$ allows us to fully reconstruct the hierarchical prefounding, where, for each (G, i) with $R(G, i) \in R_k$, $f_i = F_k|_{C_i}$.

The terminology *hierarchical prefounding* refers to the fact that if such a prefounding exists, then the nodes of τ_0 can be partitioned into hierarchical layers. This is described formally in the following definitions.

Definition 3.7

Let $((C_0, C_1, \dots), (f_1, f_2, \dots))$ be a hierarchical prefounding of τ_0 , with associated partition $R_0, \dots, R_N$ of R_{τ_0} , and let (G, i) and (G', j) be in $\tau_0 \setminus C_0$. We say that (G, i) is covered by (G', j) if $(G', j) >_{pr} (G, i)$ and there exists a $k > 0$, such that $R(G, i), R(G', j) \in R_k$.

Observe that if (G, i) is covered by (G', j) , then $C_i \subseteq C_j$. Moreover, for each $(G, i) \in \tau_0 \setminus C_0$:

$$C_i = \{(G, i)\} \cup \{(G', j) \mid (G', j) \text{ is covered by } (G, i) \text{ or } (G, i) \text{ is covered by } (G', j)\}$$

Definition 3.8

Let $((C_0, C_1, \dots), (f_1, f_2, \dots))$ be a hierarchical prefounding of τ_0 , with associated partition $R_0, \dots, R_N$ of R_{τ_0} . We define a function *layer*: $\tau_0 \rightarrow \{0, 1, \dots, N\}$ by:

If $(G, i) \in C_0$, then $\text{layer}(G, i) = 0$
 else if (G, i) is covered by (G', j)
 then $\text{layer}(G, i) = \text{layer}(G', j)$
 else $\text{layer}(G, i) = \max(\{0\} \cup \{\text{layer}(G', j) \mid (G', j) >_{pr} (G, i)\})$
 + 1

Notice that the range of the layer function is indeed $\{0, 1, \dots, N\}$. There are only $N + 1$ classes R_k and the layer value does not increase between two nodes (G', j) and (G, i) , such that (G, i) is covered by (G', j) .

It is due to the existence of this finite number of hierarchical layers that a procedure for constructing finite SLD-trees can be formulated. The underlying idea is that if the classes C_i corresponding to nodes of each layer k are well-founded (through f_i), then—using the hierarchical layer-structure—we can prove that only a finite number of mutually non-covered classes of each layer k exist. As a consequence (since “covered” implies “contained in”) we can find a finite number of classes that contain the entire SLD⁻-tree. So, the tree has become subset-wise founded.

With the exception of the construction procedure itself, this is formalized in the following theorem.

Theorem 3.1

Let τ be an SLD⁻-subtree of τ_0 and $((C_0, C_1, \dots), (f_1, f_2, \dots))$ a hierarchical prefounding for τ_0 . If each f_i is a well-founded measure on $C_i \cap \tau, >_\tau$, then there exist a finite number of sets $C_{i_1}, \dots, C_{i_M}$ in $\{C_1, C_2, \dots\}$, such that τ is subset-wise founded with respect to $((C_0, C_{i_1}, \dots, C_{i_M}), (f_{i_1}, \dots, f_{i_M}))$.

Proof

We show directly that τ is finite. Assume that it is not finite and that D is an infinite derivation in τ .

We will say that a sequence $(G_{i_1}, i_1), (G_{i_2}, i_2), \dots$ in D is a $>_{pr}$ -chain, if for each m , (G_{i_m}, i_m) is a direct $>_{pr}$ -ancestor of $(G_{i_{m+1}}, i_{m+1})$.

Since D is infinite, the corresponding proof tree must have an infinite branch, so that D itself contains an infinite $>_{pr}$ -chain $(G_{i_1}, i_1), (G_{i_2}, i_2), \dots$. By condition (3) of Definition 3.6, $>_{pr}$ -subchains with elements in C_0 are of finite length. Thus, the chain contains an infinite subsequence, $(G_{j_1}, j_1) >_{pr} (G_{j_2}, j_2) >_{pr} \dots$ with nodes in $\tau \setminus C_0$.

Since there only exist a finite number of classes R_k , there exist a finite number of members of this sequence, $((G_{q_1}, q_1), \dots, (G_{q_r}, q_r))$, $r \leq N$, which are the maximal elements for the sequence under the covered-s-order. Thus, for all other members (G_{j_i}, j_i) , $i \geq 1$, in the sequence

$$(G_{j_i}, j_i) \text{ is covered by } (G_{q_s}, q_s), \text{ for some } s, 1 \leq s \leq r.$$

For each (G_{q_s}, q_s) , $1 \leq s \leq r$, the associated set $C_{q_s} \cap \tau$ is well-founded through f_{q_s} by assumption. Therefore, $C_{q_s} \cap \tau$ is finite. Furthermore, nodes covered by (G_{q_s}, q_s) are in $C_{q_s} \cap \tau$. This contradicts the fact that the sequence $(G_{j_1}, j_1), (G_{j_2}, j_2), \dots$ is infinite.

By Theorem 2.2, we may conclude that τ is subset-wise founded. In the remainder of the proof we provide a minimal number of sets $C_{i_1}, \dots, C_{i_M}$ in $\{C_1, C_2, \dots\}$, such that τ is subset-wise founded with respect to $((C_0, C_{i_1}, \dots, C_{i_M}), (f_{i_1}, \dots, f_{i_M}))$. For any n , $1 \leq n \leq N$, let

$$S_n = \{(G, i) \in \tau \mid \text{layer}(G, i) = n \text{ and there exists a } k, 1 \leq k \leq N, \\ \text{such that } R(G, i) \in R_k \text{ and there does not exist a node } \\ (G', j) \in \tau, \text{ such that } (G, i) \text{ is covered by } (G', j)\}$$

Now, let $(C_{i_1}, \dots, C_{i_M})$ be the sequence of all classes of the given hierarchical prefounding such that their corresponding nodes (G_{i_j}, i_j) are in S_k , for some $k \leq N$. Let $(f_{i_1}, \dots, f_{i_M})$ be the associated sequence of functions, $f_{i_j}: C_{i_j} \rightarrow W_{i_j}, >_{i_j}$. We prove that τ is subset-wise founded with respect to $((C_0 \cap \tau, C_{i_1} \cap \tau, \dots, C_{i_M} \cap \tau), (f_{i_1}, \dots, f_{i_M}))$.

Clearly, each of the sets $C_q \cap \tau$ satisfies $C_q \cap \tau \subseteq G_\tau$. To prove that $G_\tau \subseteq \bigcup \{C_q \cap \tau \mid q \in \{i_1, \dots, i_M, 0\}\}$, observe that each node $(G, i) \in \tau$ which is not in $\bigcup \{S_k \mid k \leq N\}$ is either in C_0 or is covered by a node $(G', j) \in \bigcup \{S_k \mid k \leq N\}$. If the latter is the case, then $C_i \subseteq C_j$. So, $G_\tau \subseteq \bigcup \{C_q \cap \tau \mid q \in \{i_1, \dots, i_M, 0\}\}$ and condition (1) of Definition 2.7 is fulfilled.

Each $C_q \cap \tau$, $q \neq 0$, has the well-founded measure f_q by assumption, satisfying condition (2) of Definition 2.7.

Finally, condition (3) of Definition 2.7 on C_0 directly follows from condition (3) of Definition 3.6. $\square$

Slightly abusing our terminology, in the sequel we will sometimes call an SLD⁻-tree τ subset-wise founded with respect to the pair $((R_0, \dots, R_N), (F_1, \dots, F_N))$ determined by $((C_0, C_1, \dots), (f_1, f_2, \dots))$.

We conclude the section by showing how the concept of a hierarchical prefounding can be used in a constructive way to produce finite SLD-trees. For the algorithm constructing τ , we assume that the hierarchical prefounding is given as a pair $((R_0, \dots, R_N), (F_1, \dots, F_N))$.

First, we refine our notion of *covering* as follows.

Definition 3.9

Let (G, i) and (G', j) be two distinct nodes of τ_0 . We say that (G', j) is a *direct covering ancestor* of (G, i) , if

- (G, i) is covered by (G', j) and
- for any other node (G'', k) in τ_0 , such that (G, i) is covered by (G'', k) , (G', j) is also covered by (G'', k) .

In Ref. 26) the terminology *youngest covering ancestor* is used for nearly the same concept. Observe that if a direct covering ancestor exists, then it is unique.

In the algorithm below, we assume that the proof tree is constructed simultaneously with the SLD⁻-tree. Without making this construction explicit, we make use of the $>_{pr}$ -relation, derivable from the (partially generated) proof tree.

Algorithm 3.2**Initialisation**

$\tau = \{\{\leftarrow A, 1\}\},$
 $\text{Pr} = \phi, \{*\text{in Pr, the } >_{\text{pr}}\text{-relation will be constructed*}\}$
 $\text{Terminated} = \phi$

While there exists a derivation $D \in \tau$ and $D \notin \text{Terminated}$ **do**

Let (G, k) be the leaf of D ,
 Let $\text{Derive}(G, k)$ be the set of all direct $>_{\tau_0}$ -descendents,
 (G', j) of (G, k) in τ_0 ,
 Let $\text{Decrease}(G, k)$ be the set of all $(G', j) \in \text{Derive}(G, k)$ such that
 if (G'', l) is a direct covering ancestor for (G', j) in D
 and $n \in \{1, \dots, N\}$ be such that $R(G', j), R(G'', l) \in R_n$,
 then $F_n((G'', l)) >_n F_n((G', j))$

If $\text{Decrease}(G, k) = \phi$
then add D to terminated
else
 Replace τ by $\tau \setminus D \cup \{D \cup \{(G', j)\} \mid (G', j) \in \text{Decrease}(G, k)\}$
 Extend the Pr-relation accordingly.

Endwhile

Proposition 3.2

Algorithm 3.2 terminates. There exist a finite number of sets $C_0, \dots, C_M$ and functions $f_1, \dots, f_M$, such that the resulting finite SLD^- -tree τ is subset-wise founded with respect to $C_0 \cap \tau, \dots, C_M \cap \tau$ and the well-founded measures $f_1, \dots, f_M$ defined on these sets.

Proof

The proof of the termination of the algorithm is similar to the proof of Theorem 3.1. We omit it. The subset-wise foundedness follows directly from Theorem 3.1.

□

Example 3.6

We illustrate this approach with the transpose program, which was also studied in Refs. 17) and 2).

$\text{transp}(m, \text{Nil}) \leftarrow \text{nullrows}(m).$
 $\text{transp}(m, r.n) \leftarrow \text{makerow}(m, r, k), \text{transp}(k, n).$
 $\text{makerow}(\text{Nil}, \text{Nil}, \text{Nil}) \leftarrow$
 $\text{makerow}((h.t).u, h.v, t.w) \leftarrow \text{makerow}(u, v, w).$
 $\text{nullrows}(\text{Nil}) \leftarrow$
 $\text{nullrows}(\text{Nil}.z) \leftarrow \text{nullrows}(z).$

Let A be equal to $\text{transp}((a.b.\text{Nil}).x, y)$. We define the hierarchical prefounding as an instance of the generic hierarchical prefounding proposed in Example 3.5. Since there are only three recursive predicates, transp , makerow and nullrows , this gives rise to four sets R_i . The corresponding functions F_i , $i > 0$, are the functor measures associated to the selected subgoals and the set of input argument positions for these atoms. Thus, we get $((R_0, R_{\text{transp}}, R_{\text{makerow}}, R_{\text{nullrows}}), (|\cdot|_{\text{transp},\{1\}}, |\cdot|_{\text{makerow},\{1\}}, |\cdot|_{\text{nullrows},\{1\}}))$. For the computation rule, R , we assume a rule which, from each goal, selects an atom with an associated functor measure value smaller than the functor measure of the selected atom in the direct covering ancestor (if it exists). If several such atoms exist, it selects the left-most

Fig. 3 SLD-tree generated by Algorithm 3.2 for $\leftarrow \text{transp}((a.b.\text{Nil}).x, y)$

of these atoms with functor measure larger than 0, or, if they all have functor measure 0, the left-most atom.

The SLD⁻-tree resulting from Algorithm 3.2 is shown in Fig. 3.

In this figure, the predicate names have been abbreviated to their first character. Atoms selected under R are underlined. With the exception of failure and success nodes, the goals are represented as triplets (G, i, j), where i is an identifier of the node in τ_0 and j is either the identifier of (G, i)'s direct covering ancestor or (if no such ancestor exists) has the value '_'. Each node labeled with a goal which is not in C_0 is also adorned with the value of its measure under the appropriate f_i .

Observe that the tree is an SLD-tree, so that $\tau^+ = \tau$ (the nodes derivable from the leaf on the right hand side both contain only atoms that violate well-foundedness under the given prefoundings).

From the tree, the following new clauses can be inferred:

$$\begin{aligned} &\text{transpl}((a.b.\text{Nil}).\text{Nil}, (a.\text{Nil}).(b.\text{Nil}).\text{Nil}) \leftarrow \\ &\text{transpl}((a.b.\text{Nil}).(h_3.h_4.t_4).u_3, (a.h_3.v_3).(b.h_4.v_4).\text{Nil}) \leftarrow \\ &\quad \text{makerow}(u_3, v_3, w_3), \text{makerow}(w_3, v_4, w_4), \text{nullrows}(t_4, w_4). \end{aligned}$$

The amount of specialisation obtained in these clauses is slightly less than in the clauses produced by the algorithm in Ref. 2). This is due to the use of a different computation rule, which, in this case, allows some extra specialisation. However, in general, it does not guarantee the construction of a finite SLD-tree. (The length program in Section 4.1 is an example.) Our computation rule is a special case of the general rule that we will introduce in Section 4.2. Ref. 26) contains some further comments on the use of both computation rules in the context of partial deduction.

Alternatively, if we apply Algorithm 3.1, using an instance of the finite prefoundings introduced in Example 3.1, the construction of the SLD⁻-tree is stopped at the last branching of the tree (node 10 in Fig. 3). In this case, the only new clause one would derive is:

$$\begin{aligned} &\text{transp2}((a.b.\text{Nil}).x, (a.v_1).(b.v_2).\text{Nil}) \leftarrow \\ &\quad \text{makerow}(x, v_1, w_1), \text{makerow}(w_1, v_2, w_2), \text{nullrows}(w_2). \end{aligned}$$

This is precisely the result obtained in Ref. 17):

§4 Applicability and Automation of the Approach

In this section, we first show how the technique can be seen as a general framework for controlling the unfolding in partial deduction. We illustrate how existing techniques can easily be simulated and improved upon using well-founded measures. After that, we discuss its automation. We extend Algorithm 3.2, fixing the choice of the argument sets used in the functor measures of Example 3.6 and also fixing the computation rule. Finally, we further discuss the applicability of our method to meta interpreters.

4.1 Related Techniques

Typical heuristics proposed in the literature aim at loop detection and decide not to unfold a goal when a loop is suspected. Ref. 3) mentions four criteria. A literal L is not unfolded when there is, higher up in the same branch of the SLD-tree, a node with selected literal L' and respectively, L is a variant, is an instance, is more general, or unifies with L' . In particular "instance-of" is also mentioned by other authors, e.g. Refs. 35), 38).

The following program shows that with any of these techniques, loops are still possible.

```
Length([], N, N) ←
Length([X|L], N, Ln) ← Length(L, s(N), Ln).
← Length([a, b|L], 0, Ln)
```

Successive goals are:

```
Length([b|L], s(0), Ln)
Length(L, s(s(0)), Ln)
Length(L', s(s(s(0))), Ln)
```

The goal:

```
Length(Ln, sn(s(s(0))), Ln)
```

does not satisfy any of the above criteria and can still be unfolded. Many authors apply an ad-hoc depth bound to prevent such infinite unfoldings^{23,33,31}.

Although the mentioned heuristics for loop detection are quite unsatisfactory (the $>_{pr}$ relation is not taken into account when comparing two nodes on the SLD-branch and depth bounds are very ad-hoc), they can easily be simulated in our framework. We illustrate this for the "instance-of" rule. Use of a finite prefounding with a single class is sufficient. The measure to be associated with a goal which is the i th step in a derivation is N if the selected literal is an instance of a previously selected one (i.e. unfolding that literal yields a not well-founded measure) and is $\max(0, N - i)$ otherwise (i.e. a derivation is limited to N steps). Alternatively, a slightly more natural simulation can be done with a finite prefounding having one class for each recursive predicate and one for the other predicates. Observe, however, that a direct application of our approach ensures termination for the above example in a natural way, even without simulating any depth bound.

Ref. 29) also proposes several pragmatic loop prevention tests (see also the discussion of Ref. 5) below), going from simply restricting the number of times the selected goal can be a call to the same predicate, to more sophisticated ones, comparing the successive calls to the same predicate by using some measure function, which is related to the functor measure we introduced in Section 3.1.

Ref. 30) does not use a depth bound either, but a delayed unification

technique to avoid building an infinite sequence of “diverging” calls. This corresponds to not including certain arguments in the measure-function with this difference that failing to do delayed unification on those arguments will cause loops while including those arguments in the measure function will severely restrict the unfolding.

Recently, Bol⁵⁾ presented a systematic and formal account of loop checks for logic programs. He discerns between sound and complete checks. The former ensure that no solutions are removed and have mainly been focussed on in work on run-time termination analysis, the latter guarantee termination, possibly at the cost of deleting some solutions. It is illustrated how, in the context of unfolding for partial deduction, complete checks are most important. (Since complete checks sometimes remove non-loops as well, this approach is also referred to as *loop prevention* rather than *loop detection*.) A sound check can be added to improve run-time termination properties of the partially deduced program. Finally, both the approaches in Ref. 29) and in the present paper are formalised and discussed in a common framework for loop prevention.

Also related is the extensive amount of recent work on compile-time analysis techniques for the detection of (non-)terminating programs and goals (e.g. Refs. 34), 8), 4), 1), 27), 11)). These techniques as well make use of well-founded measures (often referred to as *level mappings* in this context) to prove termination. However, important differences are that compile-time termination analysis

- provides proofs of finiteness *for complete SLD-trees* τ_0 for the given program and goal and is not concerned with characterisation of finiteness for (incomplete) SLD-trees.
- only relies on the source code of the program, or at the most, a finite abstract representation of its evaluation (e.g. through abstract interpretation), to prove termination (where, in view of the application to partial deduction, we can rely on a concrete partial evaluation).

These differences, more generally, represent the typical distinction between compile-time termination analysis and loop checking or loop prevention.

Still, termination analysis could be used to further optimise our approach. Consider a goal

$$\leftarrow p(x_1.x_2.x_3.\text{Nil})$$

and assume that it has a direct covering ancestor

$$\leftarrow p(y_1.y_2.y_3.z)$$

so that, under the functor measure, the f_i value of both goals is the same. By Algorithm 3.2, the former goal should not be included in τ . However, assume that in a preprocessing, using compile-time termination analysis, we proved that

the procedure defining $p/1$ terminates for any goal with its single argument bound to a list of fixed length. Then, we could safely add the goal to τ . Moreover, we should allow full evaluation of any such goal (for instance, by placing it, together with all its descendents, in C_0).

The relationship with the termination criteria used in techniques related to partial evaluation, such as the program specialisation introduced in Ref. 18) and compiling control ^{6),10)} is less obvious. The reason is that in addition to the termination of unfolding, there is a second fundamental problem for the automation of partial deduction: one must be able to derive a finite number of SLD-trees such that the set of resultants obtained for these SLD-trees form a complete transformation for the given program (and for the goals under consideration). Program specialisation and compiling control use global analysis techniques with the aim of solving both the termination and the completeness problem in one single step. Because the solutions to the two problems are interleaved, it is impossible—at this stage—to provide a clear comparison with our technique, which only addresses termination of unfolding.

4.2 Automation

Next, we discuss the automation of the proposed approach. As a basic procedure for the construction of the SLD⁻-tree, we select Algorithm 3.2, since it provides a more refined treatment of recursion than Algorithm 3.1.

In view of full automation, three inputs to the algorithm need to be fixed: the partition $R_0, \dots, R_N$ of R_{τ_0} , the functions $F_1, \dots, F_N$ and the computation rule R . For $R_0, \dots, R_N$, the choice made in Example 3.5 is appropriate. Every recursive predicate p has its associated equivalence class R_p , whereas calls to non-recursive predicates are placed in R_0 and are therefore always unfolded. This type of partitioning can be trivially automated and is quite general.

The functor measures $F_1, \dots, F_N$ corresponding to the functions of Example 3.5 have the same advantage of generality, but here, an additional choice is necessary to fix the argument sets S_p in each functor measure $F_p = |\cdot|_{p, S_p}$. Finding appropriate sets of argument positions and a suitable computation rule R can be regarded as a search problem. Both the sets S_p and R are dynamically generated during the construction of the SLD⁻-tree in order to consume as much data as possible. We do not provide the actual algorithm, but instead point out the changes to Algorithm 3.2 that are needed.

Initially, for each recursive predicate p/n in the program P , S_p is taken equal to its entire set of argument positions, $\{1, \dots, n\}$.

All further changes are situated at the level of defining the sets $\text{Decrease}(G, k)$. Given are (see Algorithm 3.2) a leaf (G, k) of a non-terminated derivation D and $\text{Derive}(G, k)$, the set of all direct $>_{\tau_0}$ -descendents of (G, k) . In the adapted algorithm, $\text{Decrease}(G, k)$ is defined constructively, through an iteration over all elements of $\text{Derive}(G, k)$. As a side-effect of the iteration, the sets S_p and the computation rule R are fixed.

Let $\text{Decrease}(G, k) = \phi$

For all (G', j) in $\text{Derive}(G, k)$ **do**

Let $p(t_1, \dots, t_n)$ be the left-most atom of (G', j) , such that, under the assumption that $R(G', j) = p(t_1, \dots, t_n)$, the following holds:

- S_p can be partitioned into $S_{p_{\text{new}}}$ and $S_{p_{\text{remove}}}$
- $S_{p_{\text{remove}}}$ is the subset of S_p , such that
 - if** there is no direct covering ancestor of (G', j) in D
 - or if** the direct covering ancestor (G'', l) of (G', j) in D satisfies $|R(G'', l)|_{p, S_p} > |p(t_1, \dots, t_n)|_{p, S_p}$
 - then** $S_{p_{\text{remove}}} = \phi$
 - else if** (G'', l) is the direct covering ancestor of (G', j) in D
 - then**

$$S_{p_{\text{remove}}} = \{a \in S_p \mid |p(t_1, \dots, t_n)|_{\{a\}} > |R(G'', l)|_{\{a\}}\}$$
- τ remains subset-wise founded with respect to $((R_0, R_{p_1}, \dots, R_p, \dots, R_{p_N}), (|\cdot|_{p_1, S_{p_1}}, \dots, |\cdot|_{p, S_{p_{\text{new}}}}, \dots, |\cdot|_{p_N, S_{p_N}}))$

If such a $p(t_1, \dots, t_n)$ exists

then

- If** $|p(t_1, \dots, t_n)|_{S_{p_{\text{new}}}} = 0$,
- and** there exists a left-most atom $q(s_1, \dots, s_m)$ on the right of $p(t_1, \dots, t_n)$, satisfying the above conditions and with $|q(s_1, \dots, s_m)|_{S_{q_{\text{new}}}} > 0$
- then** let $R(G', j) = q(s_1, \dots, s_m)$,
- replace S_q by $S_{q_{\text{new}}}$ and add $R(G', j)$ to $\text{Decrease}(G, k)$
- else** let $R(G', j) = p(t_1, \dots, t_n)$,
- replace S_p by $S_{p_{\text{new}}}$ and add $R(G', j)$ to $\text{Decrease}(G, k)$

The remainder of Algorithm 3.2 is left unchanged.

What is expressed in this adaptation to Algorithm 3.2 is that we dynamically decrease the argument sets and fix the computation rule, in such a way that:

- (1) We give preference to left-most goals
- (2) Growing arguments in the selected atom are removed from its argument set if this does not affect the well-foundedness of the SLD^- -tree constructed so far (under the new measures)
- (3) If possible, we postpone the unfolding of atoms with functor measure 0. This avoids early branchings in the SLD^- -tree under construction (a call with measure 0 usually matches all clauses).

Observe that, although we continuously change the hierarchical prefounding, the termination of Algorithm 3.2 is not affected, since we can only remove a finite number of argument positions.

Examples 4.1

It is difficult to produce one example to illustrate the different aspects of the above algorithm. Instead, we briefly point out a number of examples for which the algorithm is particularly suited.

One of the algorithm's features is that it tries to focus automatically on the argument positions of interest. For instance, given the `reverse_acc/2` program (Example 2.1) and a goal $\leftarrow \text{reverse_acc}(3.x, \text{Nil}, y)$, the second argument position will be removed from $S_{\text{reverse_acc}}$ in the first derivation step.

A second feature is the sensible way in which the computation rule is generated. The SLD-tree in Fig. 3 for the `transp/2` program is an example of a tree produced by the above algorithm. It illustrates how early branching in the tree is avoided by delaying the selection of insufficiently instantiated atoms.

A final important feature is that procedures consuming input from different argument positions can easily be dealt with. As an example, for mergesort, at each derivation step, input is consumed from either one of two input lists. The algorithm will take both arguments into account.

4.3 Meta Interpreters

We conclude this section with a brief discussion of the additional considerations that can be made for a program that is a meta interpreter.

First, there is the problem of identifying a program as a meta interpreter. Most meta interpreters written in Prolog amalgamate the object and the meta level language. Such programs can be automatically recognized by looking for the use of the built-in predicate "clause". Meta interpreters that strictly separate the two languages are also possible. Here, all object level code is denoted by meta level terms. In the logic programming language Gödel (see Ref. 7)), meta programs are always of the latter kind. There, they can automatically be told apart from object level term handling programs through the required argument type declarations. In Prolog, however, this is not the case and therefore, this type of meta interpreters can not easily be detected. We return to this issue at the end of this section.

When a program has been identified as a meta interpreter, it receives a special treatment as was illustrated in Section 3.2. Indeed, as was mentioned there, we want the parsing to be performed as much as possible during partial deduction. Therefore, the control of the unfolding should be based on the properties of the object level goals, denoted by the meta level arguments. In addition, unfoldings performing only parsing should always be allowed. We believe it is possible to generalize the approach described in Section 3.2, on the basis of these considerations. Consider for example the following meta interpreter, which is slightly more complex than the one treated in Section 3.2.

```
solve(true).
solve((G, Gs)):- clause(G, B),
```

concat(B, Gs, Gsl),
solve(Gsl).

Here, we want to focus attention on the arguments of the first conjunct in solve((A, B)) calls. Assuming that $p_1/n_1, \dots, p_N/n_N$ are the recursive predicates of the object program, we define:

$$R_k = \{R(G, i) \in R_{r_0} \mid R(G, i) \text{ is a call of type } \text{solve}((p_k(t_1, \dots, t_{n_k}), B))\}, \text{ where } 0 < k \leq N,$$

$$\begin{aligned} R_0 = & \{R(G, i) \in R_{r_0} \mid R(G, i) \text{ is a call of type } \text{solve}((p(t_1, \dots, t_n), B)), \\ & \text{where } p/n \text{ is a non-recursive predicate}\} \\ & \cup \{R(G, i) \in R_{r_0} \mid R(G, i) \text{ is a call of type } \text{solve}(\text{true})\} \\ & \cup \{R(G, i) \in R_{r_0} \mid R(G, i) \text{ is a call to clause/2}\} \end{aligned}$$

$$R_{N+1} = \{R(G, i) \in R_{r_0} \mid R(G, i) \text{ is a call to concat/3}\}$$

and $F_k = \lambda \cdot \lambda \text{solve}, S_k$ where the initial S_k are of the form $\{1/1/1, 1/1/2, \dots, 1/1/n_k\}$ for $0 < k \leq N$.

Notice, however, that such an approach is only correct if the parsing is correct, in the sense that the meta interpreter terminates when executed with a completely instantiated goal. Moreover, the goal with respect to which the partial deduction is performed, should not contain meta level variables. How to lift these restrictions, how to check for loops at the meta level as well as at the object level, and how to automatically treat the parsing—as much as possible—separately, are subjects of ongoing research.

Next, we return to the issue of dealing with Prolog meta interpreters that represent object theories by meta level terms. Here, we would need techniques allowing to prove at compile time that certain arguments of predicates take their values in a set of terms that can be partitioned in a finite number of classes. Each of these classes should be a set of terms starting with the same functor (denoting an object level predicate). Only then, control on the level of the object program would be feasible. Type inference through abstract interpretation seems promising to provide such information. More generally—even outside the context of meta programming—this could prove to be a useful refinement of the techniques presented in this paper. This issue will need further research too. Notice, finally, that similar techniques may be required to fix the initial S_k for meta interpreters of the amalgamated type.

§5 Discussion

The use of well-founded orderings for proving termination properties is well known. It has been used for proving termination of programs,¹⁵⁾ of production systems,²⁵⁾ of term rewriting systems¹³⁾ and it has already been used to control unfolding of logic programs. For example, Ref. 16) mentions a well-founded ordering, based on subterm ordering to control the unfolding in their

partial evaluator. The work on termination analysis of logic programs can also be mentioned here. Particularly Ref. 27) explicitly defines a well-founded ordering for controlling unfolding. Finally, similar problems arise with partial evaluation in a functional framework, where one has to decide on unfolding of function calls. In the context of MIX, Ref. 19) mentions a simple well-founded strategy, where a call is unfolded when attention shifts to a smaller part (a subterm) of some argument.

Our contribution consists of introducing a class of more powerful and widely applicable well-founded orderings and devising a procedure which automatically selects the most promising one.

We have concentrated on pure logic programs, even excluding negation. In Ref. 3), it is proposed that during unfolding, negated calls can be executed when ground and remain in the resultant when non-ground. This of course jeopardises termination, since termination of "ordinary" ground logic program execution is not guaranteed in general. One solution is restricting attention to specific subclasses of programs (the work on compile-time termination analysis, cited above, is relevant here). However, a better treatment, based on constructive negation⁹⁾, seems feasible. We have also ignored pragmatic issues, related to the partial evaluation of Prolog, as there are: cuts, side effects, the preservation of the order of answers. These topics are extensively discussed by other authors, e.g. Refs. 23), 37) and 29).

As mentioned in the introduction, a method for constructing finite SLD-trees is only one of the components in a partial deduction procedure for logic programs. A sound, complete and terminating algorithm for partial deduction based on the work presented here, is described in Ref. 26). Its main advantage is a clear separation between the issues of termination of unfolding on the one hand, and soundness and completeness of the partial deduction on the other hand. As mentioned above, this contrasts with the method proposed in Ref. 18) where both concerns are addressed simultaneously.

In future work we aim to develop additional generic examples of hierarchical prefoundings that extend the power of the ones proposed in Example 3.5, Section 4.2 and Section 4.3. The functions $f_i: C_i \rightarrow W, >$ that we proposed in this paper do not take advantage of all the information contained in the goal (G, i) they are applied to, but only *measure* some (sub)argument(s) of the *selected atom* $R(G, i)$. This is in particular undesirable in cases where the computation rule R enforces a coroutining behaviour on two (or more) procedures. Typically, what we may expect in such cases are goals of the form:

$\leftarrow \text{produce}(1.2.3.x, y), \text{consume}(y)$

Unfolding the producer may instantiate the consumer to some extent, e.g.:

$\leftarrow \text{produce}(2.3.x, y_1), \text{consume}(f(1).y_1)$

Activating the consumer may lead to a new goal, such as e.g.:

$\leftarrow \text{produce}(2.3.x, y_1), \text{consume}(y_1)$

However, after performing an additional production step, leading e.g. to

$\leftarrow \text{produce}(3.x, y_2), \text{consume}(f(2).y_2)$

with the generic measures proposed earlier, it may very well be the case that further unfolding of the consumer is stopped at this point. In order to alleviate this problem, it is useful to consider functions f_i that measure terms in argument positions of several atoms in a goal. In the produce-consume example above, such functions might for example take the form

$$f_i: C_i \rightarrow \mathbb{N}^2, >^2$$

$$f_i = (|\cdot|_{\text{produce}, \{1\}}, |\cdot|_{\text{consume}, \{1\}}),$$

where $>^2$ is the lexicographical order superimposed on the usual order on $\mathbb{N}, >$. Notice that we could also have formulated the above example with only one predicate symbol, $\text{produce_consume}/2$, having the same first argument as produce and the argument of consume as its second argument. Here again—although in this case the selected atom is the entire goal—the generic measure functions proposed in this paper lack power (unfolding is likely to be stopped at the same point as above) and measure functions similar to the f_i function above would be more appropriate. Automated generation of such generalised measures is the subject of further work.

Acknowledgements

We wish to thank Jef Daels and Kristof Verschaetse for technical assistance at various stages of this work.

References

- 1) Apt, K. R. and Pedreschi, D., "Studies in Pure Prolog: Termination," *Proc. Esprit Symposium on Computational Logic* (J. W. Lloyd, ed.), pp. 150-176, 1990.
- 2) Benkerimi, K. and Lloyd, J. W., "A Procedure for the Partial Evaluation of Logic Programs," *Technical Report, TR-89-04*, Department of Computer Science, University of Bristol, Great-Britain, 1989.
- 3) Benkerimi, K. and Lloyd, J. W., "A Procedure for the Partial Evaluation of Logic Programs," *Proc. NACLP90* (S. Debray and M. Hermenegildo, eds.), pp. 343-358, 1990.
- 4) Bezem, M., "Characterising Termination of Logic Programs with Level Mappings," *Proc. NACLP89* (E. L. Lusk and R. A. Overbeek, eds.), pp. 69-80, 1989. Revised version will appear in *Journal of Logic Programming*.
- 5) Bol, R. N., "Loop Checking in Logic Programming," *Ph.D. thesis*, CWI, Amsterdam, 1991.
- 6) Bruynooghe, M., De Schreye, D. and Krekels, B., "Compiling Control," *Journal of Logic Programming*, pp. 135-162, 1989.
- 7) Burt, A. D., Hill, P. M. and Lloyd, J. W., "Preliminary Report on the Logic Programming Language Gödel," *Technical Report, TR-90-02*, Computer Science Department,

University of Bristol, March 1990. Revised Oct. 1990.

- 8) Cavedon, L., "Continuity, Consistency, and Completeness Properties for Logic Programs," *Proc. ICLP89*(G. Levi and M. Martelli, eds.), pp. 571-584, 1989.
- 9) Chan, D. and Wallace, M., "A Treatment of Negation during Partial Evaluation," in *Meta-Programming in Logic Programming*(H. D. Abramson and M. H. Rogers, eds.), *Proc. Meta88*, MIT Press, pp. 299-318, 1989.
- 10) De Schreye, D. and Bruynooghe, M., "On the Transformation of Logic Programs with Instantiation Based Computation Rules," *J. Symbolic Computation*, pp. 125-154, 1989.
- 11) De Schreye, D., Verschaeetse, K. and Bruynooghe, M., "A Framework for Analysing the Termination of Definite Logic Programs," in *Proc. FGCS92*, 1992.
- 12) Dershowitz, N. and Manna, Z., "Proving Termination with Multiset Orderings," *Communications of the ACM*, 22, 8, Aug. 1979.
- 13) Dershowitz, N., "Termination of Rewriting," *Journal of Symbolic Computation*, 3, pp. 69-116, 1987.
- 14) Fitting, M. *First-Order Logic and Automated Theorem Proving*, Springer-Verlag, New York, 1990.
- 15) Floyd, R. W., "Assigning Meanings to Programs," *Proc. Symp. in Applied Math.*, 19, Amer. Math. Soc., Providence, R. I., pp. 19-32, 1967.
- 16) Fujita, H. and Furukawa, K., "A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation," *New Generation Computing*, 6, 2-3, pp. 91-118, 1988.
- 17) Gallagher, J., "Transforming Logic Programs by Specializing Interpreters," *Proc. 7th European Conference on Artificial Intelligence*, pp. 109-122, 1986.
- 18) Gallagher, J. and Bruynooghe, M., "The Derivation of an Algorithm for Program Specialization," in *Proc. ICLP90*(D. H. D. Warren and P. Szeredi, eds.), pp. 732-746, 1990. Revised version in *New Generation Computing*, 9, 3-4, pp. 305-334, 1991.
- 19) Jones, N. D., Sestoft, P. and Sondergaard, H., "MIX: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation," *Lisp and Symbolic Computation* 2, pp. 9-50, 1989.
- 20) Komorowski, H. J., "A Specification of an Abstract Prolog Machine and Its Application to Partial Evaluation," *Technical Report, LSST69*, Linkoping University, 1981.
- 21) Komorowski, H. J., "Synthesis of Programs in the Framework of Partial Deduction," *Ser. A, No. 81*, Departments of Computer Science and Mathematics, Abo Akademi, Finland, 1989.
- 22) Lakhotia, A. and Sterling, L., "Development of a Prolog Partial Evaluation System," in *The Practice of Prolog*(L. Sterling, ed.), MIT Press, 1990.
- 23) Levi, G. and Sardu, G., "Partial Evaluation of Metaprograms in a Multiple Worlds Logic Language," *New Generation Computing*, 6, 2-3, pp. 227-247, 1988.
- 24) Lloyd, J. W. and Shepherdson, J. C., "Partial Evaluation in Logic Programming," *J. Logic Programming*, 11, 3-4, pp. 217-242, 1991.
- 25) Manna, Z. and Ness, S., "On the Termination of Markov Algorithms," in *Proc. 3rd Hawaii Int. Conf. on Syst. Sci.*, Honolulu, Hawaii, pp. 784-792, 1970.
- 26) Martens, B., De Schreye, D. and Bruynooghe, M., "Sound and Complete Partial Deduction with Unfolding Based on Well-Founded Measures," in *Proc. FGCS92*, 1992.
- 27) Plümer, L., "Termination Proofs of Logic Programs," *LNCS446*, Springer-Verlag, 1990.
- 28) Safra, S. and Shapiro, E., "Meta Interpreters for Real," *Proceedings of IFIP86*(H.-J. Kugler, ed.), pp. 271-278, 1986.
- 29) Sahlin, D., "The Mixtus Approach to Automatic Partial Evaluation of Full Prolog," *Proc. NACLP90*(S. Debray and M. Hermenegildo, eds.), pp. 377-398, 1990.
- 30) Smith, D. A. and Hickey, T. J., "Partial Evaluation of a CLP Language," *Proc. NACLP90*(S. Debray and M. Hermenegildo, eds.), pp. 119-138, 1990.
- 31) Sterling, L. and Beer, R. D., "Incremental Flavor-Mixing of Meta-Interpreters for Expert

- System Construction," *Proc. 1986 Symp. on Logic Programming*, IEEE Comp. Society Press, pp. 20-27, 1986.
- 32) Sterling, L. and Beer, R. D., "Metainterpreters for Expert System Construction," *J. Logic Programming*, 6, 1-2, pp. 163-178, 1989.
 - 33) Takeuchi, A. and Furukawa, K., "Partial Evaluation of Prolog Programs and Its Application to Metaprogramming," *Proceedings of IFIP86* (H.-J. Kugler, ed.), pp. 415-420, 1986.
 - 34) Ullman, J. D. and Van Gelder, A., "Efficient Tests for Top-Down Termination of Logical Rules," *J. ACM*, 35, 2, pp. 345-373, 1988.
 - 35) Van Harmelen, F., "The Limitations of Partial Evaluation," in *Logic-Based Knowledge Representation* (P. Jackson, H. Reichgelt and F. Van Harmelen, eds.), MIT Press, 1989.
 - 36) Venken, R., "A Prolog Meta Interpreter for Partial Evaluation and Its Application to Source to Source Transformation and Query Optimization," in *ECAI-84: Advances in Artificial Intelligence*, North-Holland, Pisa, pp. 91-100, 1984.
 - 37) Venken, R. and Demoen, B., "A Partial Evaluation System for Prolog: Some Practical Considerations," *New Generation Computing*, 6, 2-3, pp. 279-290, 1988.
 - 38) Yalcinalp, L. U., Sterling, L., Lakhoria, A. and Bansal, A., "The COMPOSERS' Guide to Meta-Programming," *internal report*, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, USA, 1990.

Maurice Bruynooghe, Prof. Dr. ir.: Belgian 'National Fund for Scientific Research' and Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, 3001 Heverlee, Belgium

Research interests: His research interests cover a broad range of topics on Logic Programming and Artificial Intelligence. He has worked on implementation techniques for Prolog, on intelligent backtracking techniques for Prolog and for solving combinatorial search problems, on program transformation techniques for logic programs, on planning and temporal reasoning, on concept learning in a predicate logic framework and on abstract interpretation for logic programs. Currently, his main interests are in abstract interpretation for constraint logic programming and in inductive logic programming.

Danny De Schreye, Prof. Dr.: Belgian 'National Fund for Scientific Research' and Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, 3001 Heverlee, Belgium

Research interests: He is active on several research topics in Logic Programming. His main interest in this area is in program transformation. On this topic, he has worked on the development of control compilation techniques and on partial deduction. His second interest is in program termination. Here, his work is on providing formal characterisations of terminating programs and on the application of static analysis techniques to the development of automatic termination tools. Further interests are in the semantics of meta-programs, in the design of sound and practical extensions of SLDNF with abduction and their applications to temporal reasoning, and in improved concepts and tools for finite domain CLP.

Bern Martens: Belgian 'National Fund for Scientific Research' and Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, 3001 Heverlee, Belgium

Research interests: He has studied knowledge representation and the role of logic in artificial intelligence. Within logic programming, he is mostly interested in deductive databases, where he has contributed to the work on checking integrity constraints, and in program transformation. Currently, his main research activities are in partial deduction and in meta-programming. In the former field, he focuses on methods for automatic sound and complete partial deduction of logic programs, including good approaches to the control of unfolding. His efforts in the latter field have mainly been directed towards providing a sensible semantics for untyped meta-programs.