ISLP91

91-92

A General Criterion for Avoiding Infinite Unfolding during Partial Deduction of Logic Programs

Maurice Bruynooghe, Danny De Schreye, Bern Martens
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Heverlee, Belgium
email: {maurice, dannyd, bern}@cs.kuleuven.ac.be

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.

1. Introduction

Since its introduction in logic programming by Komorowski in 1981 ([10]), partial evaluation has attracted the attention of many researchers in the field. Some, e.g. in [18], [24], [25], have addressed pragmatic issues related to the impurities of Prolog. Others, e.g. in [9], [13], [17], [21] and [22] were attracted by the perspective of eliminating the overhead associated with meta interpreters. Finally, the subject was put on a firm theoretical basis in [14].

Summarizing [14], a program is partially evaluated with respect to a goal $\leftarrow$ A by taking the resultants of an *incomplete* SLD-tree with $\leftarrow$ A as root. (For simplicity of exposition, we restrict ourselves to SLD ignoring the more general SLDNF case. See also section 5.) The problem discussed in this paper is how to construct this incomplete SLD-tree, in other words, how to unfold as much as possible while avoiding the construction of infinite branches. (Pushing unfolding to the limits might well yield an exponential growth of the program size and not necessarily improves performance. This issue is not discussed in the current paper.) Typical methods described in the literature rely on ad hoc techniques to avoid infinite branches (loops). For example, the authors of [1] mention four possible criteria for stopping unfolding : the goal is not unfolded when the selected literal is (1) a variant, (2) an instance, (3) more general, (4) unifies with a literal selected in a goal which is an ancestor of the current goal. This is insufficient to prevent infinite branches as the following program shows :

Length ([], N, N) ←
Length ([X | L], N, L_n) ← Length (L, s(N), L_n)
← Length ([a, b | L], 0, L_n)

Successive goals are :

← Length ([b | L], s(0), L_n)
← Length (L, s(s(0)), L_n)
← Length (L′, s(s(s(0))), L_n)

The goal :

← Length(L^n, s^n(s(s(0))), L_n)

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 [13], [20], [22].

In [19], the construction of the increasing term $s^n(0)$ is pushed out of the recursive call to avoid the failure of the loop detection test.

We propose a more systematic approach. It is rooted in well-known techniques for proving termination of programs (e.g. [7], [15]) based on well-founded sets. Applied on incomplete SLD-trees, the idea is to associate to them a *well-founded measure*, a monotone function from the successive nodes on a branch into a well-founded set - a set for which no infinite sequence of decreasing elements exists. However, in practice, it is rather difficult to find appropriate well-founded measures. A way out is suggested by the common wisdom that one should only compare states where the same recursive literal is selected. More precisely, the nodes of the SLD-tree can be partitioned into N classes $C_0, C_1, \cdots, C_N$ with N the number of recursive predicates. C_0 contains goals where a non-recursive predicate is selected. Such nodes do not need a well-founded measure as after a finite number of steps, either a recursive predicate must be selected, or the derivation terminates in success or failure. C_i $(i > 0)$ contains nodes where the i^{th} recursive predicate is selected and a well founded measure must be associated with nodes in C_i.

However, this is still inadequate as the following example shows.

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.

Consider the goal ← sums ([1, 2 | L], s). An obvious measure is the number of list constructors on the first argument. Unfolding yields :

← sum([1, 2 | L],s_1), sums ([2 | L], s_2)

The same measure can be used to control unfolding of sum. This eventually yields

$\leftarrow$ sum(L,s_1''), $s_1' = s_1'' + 2$, $s_1 = s_1' + 1$, sums([2 | L],s_2).

Unfolding sum once more will not decrease the number of list constructors in the first argument. However, sums ([2 | L],s_2) can be selected for unfolding. This gives rise to :

$\leftarrow$ sum(L, s_1''), $s_1' = s_1'' + 2$, $s_1 = s_1' + 1$, sum([2 | L], s_3), sums(L, s_4)

At this point it is reasonable to select sum([2 | L], s_3) for unfolding. However, the branch of the SLD-tree already has a goal where a variant of the same literal is being selected. The observation to be made is that the second unfolding of sums gives rise to a sum literal which is unrelated to the sum literal (and its descendent) created by the first unfolding of the sums literal and that it is undesirable to compare such unrelated sum literals, although they are selected on the same branch of the SLD tree.

The hierarchical prefoundings, formally introduced in the next section, capture this notion.

2. Hierarchical prefoundings.

In what follows, we frequently make use of strict order relations, denoted >. A strictly ordered set, S,> , will be called an s-poset, the corresponding order, >, an s-order.

Definition 2.1 An s-poset S,> is said to be *well-founded* if there is no infinite sequence of elements (s_1, s_2, $\cdots$) in S, such that $s_i > s_{i+1}$, for all $i \geq 1$.

Definition 2.2 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 $W,>_W$.

Given an SLD-tree τ, an s-poset can be associated to it in a natural way, by taking the tree structure itself as a strict ordering on the set of goals in the tree. 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_τ = { (G,i) | G is a goal of τ having i as its associated number in τ_0 }. The s-order is defined as (G,i) $>_\tau$ (G',j) if node i is an ancestor of node j in τ_0.

Definition 2.3 An SLD-tree τ is *well-founded* if there exists a well-founded measure f on $G_\tau,>_\tau$.

Paraphrasing a theorem in [15], we can formulate the following theorem.

Theorem 2.1 An SLD-tree is finite iff it is well-founded.

Proof. The if-part is trivial. 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}.

□

So, in order to ensure that the unfolding for the partial deduction terminates, we can first fix a useful well-founded measure and then control the construction of the SLD-tree by imposing that at each unfolding, the proposed measure must remain monotonic. Observe that the well-founded measures in Theorem 2.1 have a very specific form. For each derivation $(G_0,i_0), \cdots, (G_n,i_n), \cdots$ in τ, its image $\{f(G_0,i_0), \cdots, f(G_n,i_n), \cdots\}$ in W is a totally s-ordered subset of $W,>_W$. Although such a measure (at least for some $W,>_W$) always exists, in practice it is useful to allow more general measures, which still provide a sufficient condition for the termination of the unfolding, but are both easier to provide and more accurate in their control of the unfolding.

Our approach will be as follows. First, given a program P, an associated *prefounding* is defined. Such a prefounding is a pair consisting of a (possibly infinite) sequence of sets of goal statements in τ_0, $(C_0, C_1, C_2 \cdots)$, and of a sequence of (measure-) functions $(f_1, f_2, \cdots)$ associated to $(C_1, C_2 \cdots)$, where $f_i: C_i \rightarrow W_i, >_i$, and $W_i, >_i$ is a well-founded set.

On the basis of such a prefounding, an SLD-tree is constructed by maintaining as an invariant during the unfolding that each f_i must be a well-founded measure on C_i (unfolding of a goal statement in C_i is prohibited if the unfolding would violate the well-foundedness of the corresponding f_i).

In addition, the proposed prefounding will satisfy the following properties:

1. For each goal statement (G,i) in C_0 and for each infinite branch in τ_0 containing (G,i), there is a descendent of (G,i) belonging to some C_j, $j>0$, on the same branch (meaning that there is in fact no need to measure the goals in C_0).

2. For any SLD-tree τ, if each f_i is a well-founded measure on $C_i \cap \tau$, then there are only a finite number of distinct sets $C_i \cap \tau$.

On the basis of the sketched construction procedure and its properties, it should be intuitively clear that the construction will terminate and return a finite SLD-tree. We now formalize these ideas.

Throughout the remainder of 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.

Definition 2.4 Let $(G,i) = (\leftarrow A_1, \cdots, A_j, \cdots, A_n, i)$ be a node in an SLD-tree τ, let $R(G) = A_j$ be the call selected by the computation rule R, let $H \leftarrow B_1, \cdots, B_m$ 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, $(G',k) = ((\leftarrow A_1, \cdots, A_{j-1}, B_1, \cdots, B_m, A_{j+1}, \cdots, A_n)\theta, k)$. We say that $B_1\theta, \cdots, B_m\theta$ in G' are *direct descendents* of A_j in G and that A_j in G is a *direct ancestor* of $B_1\theta, \cdots, B_m\theta$ in G'.

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 and B an atom in G', A is an ancestor of B is denoted

as $A >_{pr} B$ (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 $R(G) >_{pr} R(G')$. By abuse of notation, we denote it as $(G,i) >_{pr} (G',k)$.

In [2], we introduce two different types of prefoundings. Here, due to space restrictions, we limit ourselves to the most expressive one.

Definition 2.5 A pair $((C_0, C_1, C_2 \cdots), (f_1, f_2, \cdots))$ is a *hierarchical prefounding* for τ_0, if :

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

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

   $C_0 = \{(G,i) \in \tau_0 \mid R(G) \in R_0\} \cup \{ (\Box,i) \mid i \in \mathbb{N} \}$.

2. C_0 contains no infinite sequence $(G_{i_1},i_1) >_{pr} (G_{i_2},i_2) >_{pr} \cdots$, such that for all m, $m \ge 1$, $R(G_{i_m})$ in (G_{i_m},i_m) is a direct ancestor of $R(G_{i_{m+1}})$ in $(G_{i_{m+1}},i_{m+1})$.

3. There exist a finite number of well-founded sets $W_1,>_1, \cdots, W_N,>_N$, such that for each $(G,i) \in \tau_0$ with $R(G) \in R_k$, f_i is a function, $f_i : C_i \to W_k$.

4. For all m, $n \ge 1$ and for each $(G,i) \in C_m \cap C_n$: $f_m(G,i) = f_n(G,i)$.

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

Definition 2.6 Let Term denote the set of terms in the first order language used to define the theory P. The *functor norm* is a function $|.| : \text{Term} \to \mathbb{N}$ defined as :

If $t = f(t_1, \cdots, t_n)$, $n > 0$
then $|t| = 1 + |t_1| + \cdots + |t_n|$
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 2.7 Let p be a predicate of arity n and $S = \{a_1, \cdots, a_m\}$, $1 \le a_k \le n$, $1 \le k \le m$, a set of argument positions for p. The *functor measure* with respect to p and S is a function :

$|.|_{p,S} : \{A \mid A \text{ an atom with predicate symbol } p\} \rightarrow \mathbb{N}$, defined as:

$|.|_{p,S}(p(t_1, \cdots, t_n)) = |t_{a_1}| + \cdots + |t_{a_m}|$.

We can extend the expressivity of the functions f_k somewhat by measuring subterms of arguments instead of just arguments. This is achieved by allowing selector functions, which we will denote as finite, non-empty sequences of positive integers connected with slashes, $i_1/i_2/\cdots/i_n$. It is straightforward to generalize Definition 2.7 by replacing the set of argument positions by a set of selector functions. We keep the notation $|.|_{p,S}$, where S is the set of selector functions.

Example 2.1 We provide a natural example of a hierarchical prefounding $((C_0, C_1, C_2 \cdots), (f_1, f_2, \cdots))$. P contains only a finite number of recursive predicates $p_1, \cdots, p_N$. Given the computation rule R, we define :

$R_k = \{R(G) \in R_{\tau_0} \mid R(G) \text{ is a call to } p_k\}$, where $k > 0$,
$R_0 = \{R(G) \in R_{\tau_0} \mid R(G) \text{ is a call to a non-recursive predicate}\}$.

As a consequence,

$C_0 = \{(G,i) \mid G \text{ a goal such that } R(G) \text{ is a call to a non-recursive predicate and } i \in \mathbb{N}\} \cup \{(\square,i) \mid i \in \mathbb{N}\}$.

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 is straightforward to verify the conditions in Definition 2.5.

Definition 2.5 is slightly more general than Example 2.1. This is because in general (e.g. when dealing with meta interpreters), more refined partitions of G_{τ_0} may be useful (see section 4).

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, corresponding to the recursive depth of the selected subgoal in each node. If in addition, τ is a subtree of τ_0, such that each f_i restricted to $C_i \cap \tau$ is a well-founded measure, then the number of hierarchical layers can be shown to be finite. This results in the following theorem.

Theorem 2.2 Let τ be an SLD-subtree of τ_0 and $((C_0, C_1, \cdots), (f_1, f_2, \cdots))$ a hierarchical prefounding for τ_0. If each f_i is a well-founded measure on $C_i \cap \tau$, $>_\tau$, then τ is finite.

Proof. Let $R_0, \cdots, R_N$ be a partition of R_{τ_0}, and let (G,i) be the associated partition of $((C_0, C_1, \cdots), (f_1, f_2, \cdots))$ 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 $R(G), R(G') \in R_k$. Observe that if (G,i) is covered by (G',j), then $C_i \subseteq C_j$.

Moreover, for each $(G,i) \in \tau_0 \setminus C_0$:

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

Assume that τ is not finite and that D is an infinite derivation in τ.

We will say that a sequence (G_{i_1}, i_1), $(G_{i_2}, i_2), \cdots$ 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 $(G_{i_1}, i_1), (G_{i_2}, i_2), \cdots$. By condition 2 of Definition 2.5, $>_{pr}$-subchains with elements in C_0 are of finite length. Thus, the chain contains an infinite subsequence, $(G_{j_1}, j_1) >_{pr} (G_{j_2}, j_2) >_{pr} \cdots$ 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, $((G_{q_1}, q_1), \cdots, (G_{q_r}, q_r))$, $r \leq N$, 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) is covered by (G_{q_s}, q_s), 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 $(G_{j_1}, j_1), (G_{j_2}, j_2), \cdots$ is infinite.

□

3. Constructing finite SLD-trees using hierarchical prefoundings

Here we show how the concept of a hierarchical prefounding can be used in a constructive way to produce finite SLD-trees. It is clear that the pair of potentially infinite sequences $((C_0, C_1, \cdots), (f_1, \cdots))$ can not serve as input for an algorithm. Instead, we assume that the finite partition $R_0, \cdots, R_N$ of R_{τ_0} is given, in addition to functions $F_1, \cdots, F_N$, such that

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

where $W_k, >_k$ is a well-founded s-poset.

Observe that due to condition 4 of Definition 2.5, we can associate such functions F_k, $k > 0$, to any given hierarchical prefounding $((C_0, C_1, \cdots), (f_1, \cdots))$. More precisely, for any (G,i) with $R(G) \in R_k$, $F_k(G,i) = f_i(G,i)$. Conversely, the pair $((R_0, \cdots, R_N), (F_1, \cdots, F_N))$ allows us to fully reconstruct the hierarchical prefounding.

Algorithm 3.1

Initialisation

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

While there exists a derivation D in τ and D $\notin$ Terminated **do**

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

If there exist $k > 0$, with $R(G) \in R_k$
and there exists an ancestor (G',j) of (G,i) in D, $>_\tau$
such that $R(G') \in R_k$
and $((R(G'),j), (R(G),i))$ is in the transitive closure of Pr
and $F_k(G,i) \geq_k F_k(G',j)$

then add D to Terminated

else

Let Derive (G,i) be the set of all derivation steps that can be performed, given (G,i), R and P,

If Derive $(G,i) = \phi$

then add D to terminated

else

Let Descend (R(G),i) be the set of all pairs $((R(G),i), (B\theta,j))$, where
- B is an atom in the body of a clause applied in an element of Derive (G,i)
- θ is the corresponding m.g.u.
- j is the number of the corresponding descendent of (G,i)

Expand D in τ with the derivation steps in Derive (G,i).
Add the elements of Descend (R(G),i) to Pr.

Endwhile

Proposition 3.1 Algorithm 3.1 terminates and returns a finite SLD-tree.

Proof. The proof is similar to that of Theorem 2.2.

In view of full automatization, three inputs to Algorithm 3.1 need to be fixed : the partition $R_0, \cdots, R_N$ of R_{τ_0}, the functions $F_1, \cdots, F_N$ and the computation rule R. For $R_0, \cdots, R_N$, the choice made in Example 2.1 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 $F_1, \cdots, F_N$ corresponding to the functions of Example 2.1 have the same advantage of generality, but here, an additional choice is

necessary to fix the argument sets S_{p_k} in each functor measure $F_k = |.|_{p_k,S_{p_k}}$. Finding appropriate sets of argument positions and a suitable computation rule R can be regarded as a search problem. Both the sets S_{p_k} 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 additions to Algorithm 3.1 that are appropriate.

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

When processing a leaf of a non-terminated derivation of τ, we dynamically decrease the argument sets $S_{p_1}, \cdots, S_{p_N}$ and fix the computation rule R, in such a way that :

1. We give preference to left-most subgoals.
2. Growing arguments in the selected atom with predicate symbol p_i are removed from the corresponding argument set S_{p_i} if this does not affect the well-foundedness of the SLD-tree constructed so far (under the new measures).
3. If no subset of S_{p_i} exists which satisfies the above condition, we adapt R to select the next subgoal (from left-to-right) and we return to step 2.
4. 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 3.1 is not affected, since we can only remove a finite number of argument positions. Also, we must point out that changing the computation rule may affect the order in which the answers are generated.

We illustrate this approach with the transpose program.

Example 3.1

transp(m,Nil) ← nullrows(m).
transp(m,r.n) ← makerow(m,r,k), transp(k,n).
makerow(Nil,Nil,Nil) ←.
makerow((h.t).u,h.v,t.w) ← makerow(u,v,w).
nullrows(Nil) ←.
nullrows(Nil.z) ← nullrows(z).

Take A equal to transp((a.b.Nil).x,y). We start off with an initial hierarchical prefounding $((R_0, R_{transp}, R_{makerow}, R_{nullrows}), (|.|_{transp,\{1,2\}}, |.|_{makerow,\{1,2,3\}}, |.|_{nullrows,\{1\}})$.
An initial fragment of the resulting SLD-tree is shown in the figure below.

In this figure, the predicate names have been abbreviated to their first character. Selected atoms under R are underlined. Each node is adorned with the functor

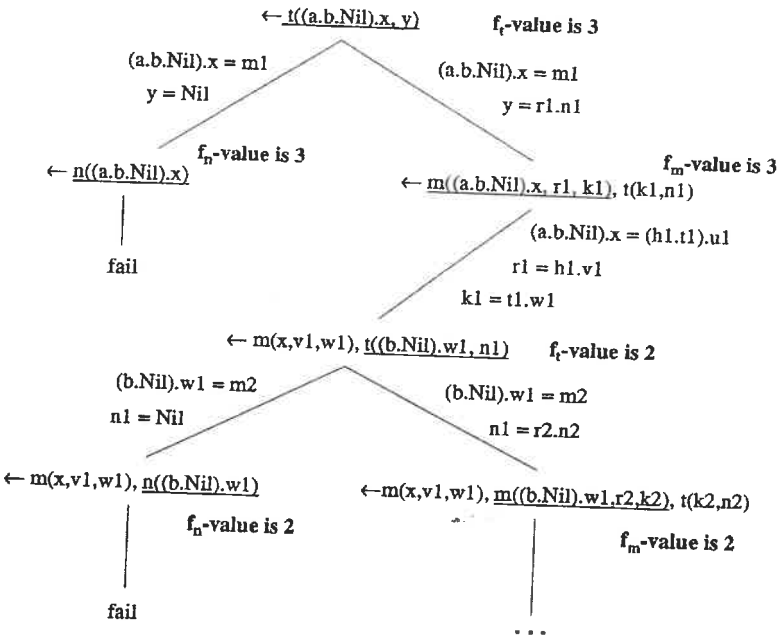


Fig. : Initial fragment of the SLD-tree constructed for ← transp((a.b.Nil).x, y).

measure values under the final argument sets, which in this case are equal to the original ones (only the first argument of makerow gets constructed in a derivation step, but it can not be removed from the argument set, due to ancestor derivations that depend on it).

Depending on whether the leaves of the generated SLD-tree are retained in the synthesis or omitted before synthesis (see [2] for a discussion on this trade-off), we (resp.) obtain the new clauses

transp1((a.b.Nil).Nil, (a.Nil). (b.Nil). Nil) ←
transp1((a.b.Nil).(h_3.t_3).u_3, (a.h_3.v_3).(b.v_2).Nil) ←
makerow(u_3,v_3,w_3), makerow(t_3.w_3,v_2,Nil).
transp1((a.b.Nil).(h_3.t_3).u_3, (a.h_3.v_3).(b.v_2).Nil) ←
makerow(u_3,v_3,w_3), makerow(t_3.w_3,v_2,Nil.z_2), nullrows(z_2).

or just the one new clause

transp2((a.b.Nil).x, (a.v_1). (b.v_2).Nil) ←
makerow(x,v_1,w_1), makerow(w_1,v_2,w_2), nullrows(w_2).

4. Application to meta interpreters

This section contains 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 [3]), 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 the section.

When a program has been identified as a meta interpreter, it receives a special treatment. Indeed, we want the parsing to be performed as much as possible during partial deduction ([12] and [21]). Therefore, the control of the unfolding should be based on properties of the object level goals, denoted by the meta level arguments. In addition, unfoldings performing only parsing should always be allowed. Consider for example the following vanilla-type meta interpreter :

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

We always want to unfold calls of the type solve((A,B)) or solve(true). Only calls of the type solve(A) should be checked. Assume that $p_1/n_1, \cdots, p_N/n_N$ are the recursive predicates of the object program. Then we define :

$R_k = \{R(G) \in R_{\tau_0} \mid R(G)$ is a call of type solve $(p_k(t_1, \cdots, t_{n_k}))\}$, where $k > 0$,

$R_0 = \{R(G) \in R_{\tau_0} \mid R(G)$ is a call of type solve $(p(t_1, \cdots, t_n))$,
where p/n is a non-recursive predicate $\}$
$\cup \{R(G) \in R_{\tau_0} \mid R(G)$ is a call of type solve((A,B)) $\}$
$\cup \{R(G) \in R_{\tau_0} \mid R(G)$ is a call of type solve(true) $\}$
$\cup \{R(G) \in R_{\tau_0} \mid R(G)$ is a call to clause/2 $\}$

and $F_k = |.|_{solve,S_k}$ where the initial S_k are of the form $\{1/1, 1/2, \cdots, 1/n_k\}$. Other meta interpreters, as for example

```
solve(true).
solve((G,Gs)):- clause(G,B),
                concat(B,Gs,Gs1),
                solve(Gs1).
```

can be treated similarly.

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.

We conclude this section with a brief discussion of how to deal 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.

5. Discussion

The use of well-founded orderings for proving termination properties is widely spread. It has been used for proving termination of programs [7], of production systems [15], of term rewriting systems [6] and it has already been used to control unfolding of logic programs. For example, [8] mentions a well-founded ordering, based on subterm ordering, to control the unfolding in the partial evaluator. In [16], Plümer defines a well-founded ordering for controlling unfolding, which he uses in the context of proving termination of logic programs.

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. As stated e.g. in [1], there is a very simple treatment for negated calls. They can be executed when ground and remain in the resultant when non-ground, although a better treatment, based on constructive negation [4], is 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. in [13], [18] and [25].

Typical heuristics proposed in the literature aim at loop detection and decide not to unfold a goal when a loop is suspected. [1] mentions four criteria, which we recalled in the introduction. In particular "instance-of" is also mentioned elsewhere, e.g. in [23] and [26]. In the introduction, we also presented an example showing that with any of these criteria, loops remain possible and we mentioned the use of the "instance-of" criterion combined with a depth bound in [13], [20] and [22]. Although these 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 (a simpler variant of) our framework. We refer to [2] for the details. Observe, however, that a direct application of our approach ensures termination for the mentioned example in a natural way, even without simulating any depth bound.

The authors of [19] do 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.

[18] also proposes several pragmatic loop detection tests, 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 2.

In the work reported here, we have separated the issue of controlling the unfolding from the (related) issues of the completeness and (in the case of SLDNF) correctness of the transformed program. We are currently working on a more general framework, integrating the use of well-founded measures for the control of unfolding with the conditions for completeness (and correctness) presented in [14].

Acknowledgements.

All authors are supported by the Belgian National Fund for Scientific Research. The work reported here was partially supported by ESPRIT BRA COMPULOG (project 3012).

We wish to thank Jef Daels and Kristof Verschaetse for technical assistance at various stages of this work. We also acknowledge several anonymous referees for their valuable comments.

References.

[1] K. Benkerimi and J.W. Lloyd, A Procedure for the Partial Evaluation of Logic Programs, Proc. NACLP90, eds. S.Debray and M.Hermenegildo, 1990, pp. 343-358.

[2] M. Bruynooghe, D. De Schreye, B. Martens, A General Criterion for Avoiding Infinite Unfolding during Partial Deduction, Report CW-126, Department of Computer Science, K.U.Leuven, 1991.

[3] A.D. Burt, P.M. Hill and J.W. Lloyd, Preliminary Report on the Logic Programming Language Gödel, University of Bristol, Computer Science Department, TR-90-02, (March 1990, Revised Oct. 1990).

[4] D. Chan and M. Wallace, A treatment of negation during partial evaluation, in Meta-Programming in Logic Programming, eds. H.D.Abramson and M.H.Rogers, Proc. Meta88, MIT Press, 1989, pp. 299-318.

[5] N. Dershowitz and Z. Manna, Proving Termination with Multiset Orderings, Communications of the ACM, August 1979, Volume 22, Number 8.

[6] N. Derhowitz, Termination of rewriting, Journal of Symbolic Computation, 3, 1987, pp. 69-116.

[7] R.W. Floyd, Assigning meanings to programs, Proc. Symp. in Applied Math., Vol.19, Amer. Math. Soc., Providence, R.I., pp. 19-32.

[8] H. Fujita and K. Furukawa, A self-applicable partial evaluator and its use in incremental compilation, New Generation Computing, 6 (2,3), 1988, pp. 91-118.

[9] J. Gallagher, Transforming Logic Programs By Specializing Interpreters, Proc. 7th European Conference on Artificial Intelligence, 1986, pp 109-122.

[10] H.J. Komorowski, A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, Technical Report LSST69, Linkoping university 1981.

[11] H.J. Komorowski, Synthesis of programs in the framework of partial deduction, Ser.A, No 81, Departments of Computer Science and Mathematics, Abo Akademi, Finland.

[12] A. Lakhotia and L. Sterling, Development of a Prolog Partial Evaluation System, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Feb. 1989 (working paper).

[13] G. Levi and G. Sardu, Partial Evaluation of Metaprograms in a Multiple Worlds Logic Language, New Generation Computing 1988,Vol.6,Nos2&3.

[14] J.W. Lloyd and J.C. Shepherdson, Partial Evaluation in Logic Programming, University of Bristol, Computer Science Department, CS-87-09 (Dec. 1987, Revised July 1989), to appear in J. Logic Programming.

[15] Z. Manna and S. Ness, On the termination of Markov algorithms, pp. 784-792 in Proc. 3rd Hawaii Int. Conf. on Syst. Sci., Honolulu, Hawaii (1970).

[16] L. Plümer, Termination proofs of logic programs, LNCS 446, Springer-Verlag, 1990.

[17] S. Safra and E. Shapiro, Meta Interpreters for Real, Proceedings of IFIP 86, H.-J. Kugler (ed), 1986, pp. 271 - 278.

[18] D. Sahlin, The Mixtus approach to automatic partial evaluation of full Prolog, Proc. NACLP90, eds. S.Debray and M.Hermenegildo, 1990, pp. 377-398.

[19] D.A. Smith and T.J. Hickey, Partial evaluation of a CLP language, Proc. NACLP90, eds. S.Debray and M.Hermenegildo, 1990, pp. 119-138.

[20] L. Sterling and R.D. Beer, Incremental flavor-mixing of meta-interpreters for expert system construction, Proc. 1986 Symp. on Logic Programming, IEEE Comp. Society Press, 1986, pp. 20-27.

[21] L. Sterling and R.D. Beer, Metainterpreters for expert system construction, J. Logic Programming 89, pp. 163-178.

[22] A. Takeuchi and K. Furukawa, Partial evaluation of PROLOG programs and its application to metaprogramming, Proceedings of IFIP 86, H.-J. Kugler (ed), 1986, pp. 415-420.

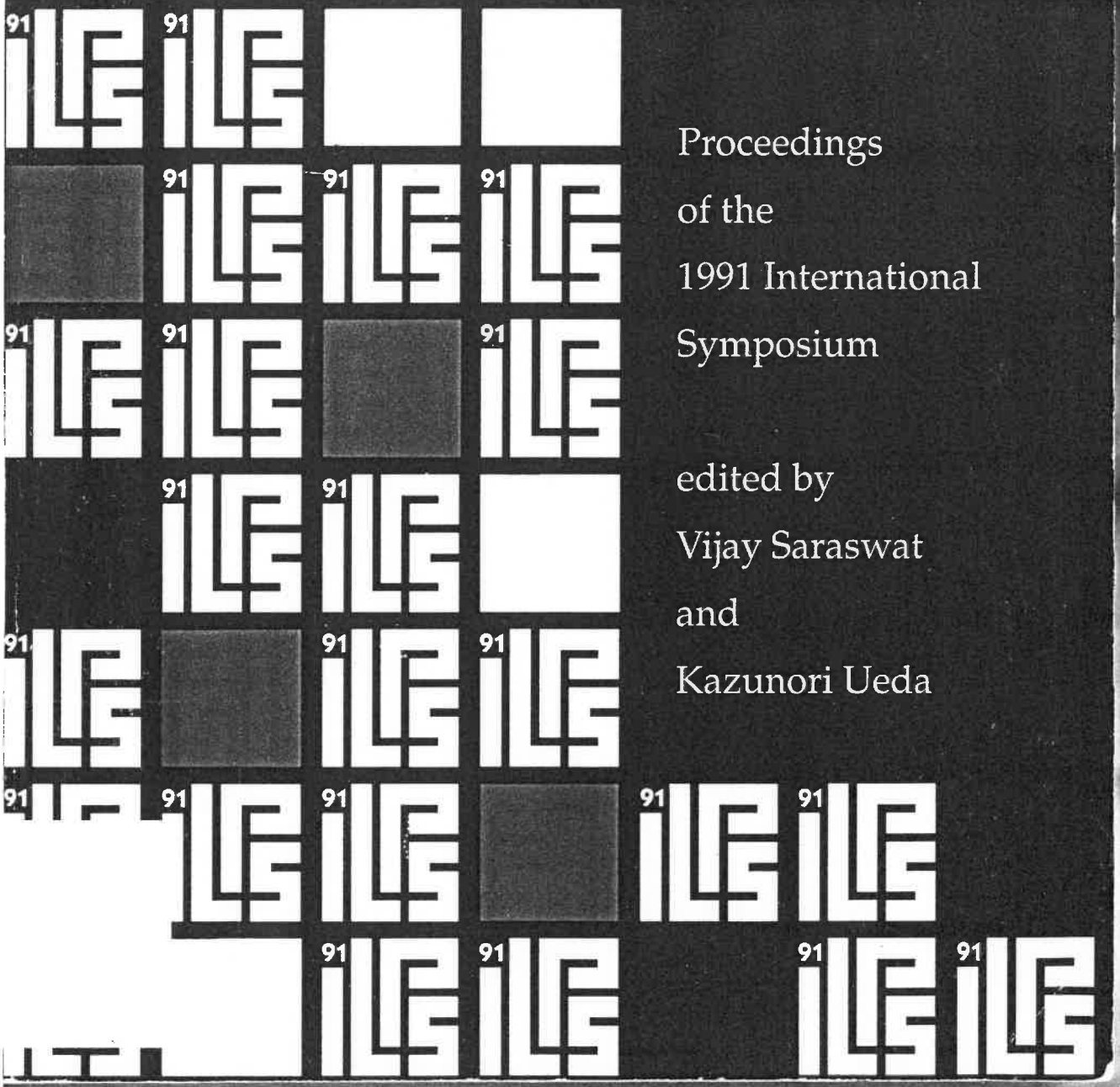
[23] F. Van Harmelen, The limitations of partial evaluation, in Logic-Based Knowledge Representation, eds. P. Jackson, H. Reichelt and F. Van Harmelen, MIT Press, 1989.

[24] R. Venken, 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, 84, pp. 91-100.

[25] R. Venken and B. Demoen, A partial evaluation system for Prolog: some practical considerations, New Generation Computing, 6 (2,3), 1988, pp. 279-290.

[26] L.U. Yalcinalp, L. Sterling, A. Lakhotia and A. Bansal, The COMPOSERS' Guide to Meta-Programming, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, Ohio, 44106, USA (internal report).

Logic Programming



Proceedings
of the
1991 International
Symposium

edited by
Vijay Saraswat
and
Kazunori Ueda