Analysing Definitional Trees: Looking for Determinism[1]

Pascual Julián Iranzo and Christian Villamizar Lamus

Departamento de Informática
Universidad de Castilla–La Mancha
Ciudad Real, Spain
`Pascual.Julian@uclm.es` `cvillami@inf-cr.uclm.es`

Abstract. This paper describes how high level implementations of (needed) narrowing into Prolog can be improved by analysing definitional trees. First, we introduce a refined representation of definitional trees that handles properly the knowledge about the inductive positions of a pattern. The aim is to take advantage of the new representation of definitional trees to improve the aforementioned kind of implementation systems. Second, we introduce selective unfolding transformations, on determinate atom calls in the Prolog code, by examining the existence of what we call "deterministic (sub)branches" in a definitional tree. As a result of this analysis, we define some generic algorithms that allow us to compile a functional logic program into a set of Prolog clauses which increases determinism and incorporates some refinements that are obtained by *ad hoc* artifices in other similar implementations of functional logic languages. We also present and discuss the advantages of our proposals by means of some simple examples.

Keywords: Functional logic programming, narrowing strategies, implementation of functional logic languages, program transformation.

1 Introduction

Functional logic programming [14] aims to implement programming languages that integrate the best features of both functional programming and logic programming. Most of the approaches to the integration of functional and logic languages consider term rewriting systems as programs and some narrowing strategy as complete operational mechanism. Laziness is a valuable feature of functional logic languages, since it increases the expressive power of this kind of languages: it supports computations with infinite data structures and a modular programming style. Among the different lazy narrowing strategies, needed narrowing [7] has been postulated optimal from several points of view: i) it is correct and complete, with regard to strict equations and constructor substitutions answers, for the class of inductively sequential programs (see, forward,

---

[1] Supported by CICYT TIC 2001-2705-C03-01, Acción Integrada Hispano-Italiana HI2000-0161, and Acción Integrada Hispano-Alemana HA2001-0059.

Definition 2); ii) it computes minimal length derivations, if common variables are shared; and iii) no redundant answers are obtained. Some of these optimality properties have also been established for a broader class of term rewriting systems defining non–deterministic functions [4]. Needed narrowing addresses computations by means of some structures, namely definitional trees [2], which contain all the information about the program rules. These structures allow us to select a position of the term which is being evaluated and this position points out to a reducible subterm that is "unavoidable" to reduce in order to obtain the result of the computation. It is accepted that the framework for declarative programing based on non–deterministic lazy functions of [20] also uses definitional trees as part of its computational mechanism. In recent years, a great effort has been done to provide the integrated languages with high level implementations of this computational model into Prolog (see for instance [3, 9, 15, 18] and [21]). This paper investigates how an analysis of definitional trees can introduce improvements in the quality of the Prolog code generated by these implementation systems.

The paper is organized as follows: Section 2 recalls some basic notions we use in the rest of the sections. In Section 3 we describe a refined representation of definitional trees and we give an algorithm for their construction in the style of [17]. Section 4 introduces two new translation techniques: Section 4.1 discusses how to take advantage of the new representation of definitional trees to improve (needed) narrowing implementations; Section 4.2 presents an algorithm, guided by the structure of a definitional tree, which is able to produce the same effect as if a determinate unfolding transformation was applied on the compiled Prolog code. Section 5 presents some experiments that show the effectiveness of our proposals. Section 6 discuses the relation of our techniques to other research on functional logic programming and logic programming. Finally, Section 7 contains our conclusions.

2 Preliminaries

We consider first order expressions or *terms* built from symbols of the set of variables $\mathcal{X}$ and the set of function symbols $\mathcal{F}$ in the usual way. The set of terms is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We sometimes write $f/n \in \mathcal{F}$ to denote that f is a n–ary function symbol. If t is a term different from a variable, $\mathcal{R}oot(t)$ is the function symbol heading t, also called the *root symbol* of t. A term is *linear* if it does not contain multiple occurrences of the same variable. $\mathcal{V}ar(o)$ is the set of variables occurring in the syntactic object o. We write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$.

A *substitution* σ is a mapping from the set of variables to the set of terms, with finite *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$. We denote the identity substitution by *id*. We define the composition of two substitutions σ and θ, denoted $\sigma \circ \theta$ as usual: $\sigma \circ \theta(x) = \hat{\sigma}(\theta(x))$, where $\hat{\sigma}$ is the extension of σ to the domain of the terms. A *renaming* is a substitution ρ such that there exists the inverse substitution ρ^{-1} and $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = id$.

A term t is *more general* than s (or s is an *instance* of t), in symbols $t \leq s$, if $(\exists\sigma)\ s = \sigma(t)$. Two terms t and t' are *variants* if there exists a renaming ρ such that $t' = \rho(t)$. We say that t is *strictly* more general than s, denoted $t < s$, if $t \leq s$ and t and s are not variants. The quasi–order relation "$\leq$" on terms is often called *subsumption order* and "$<$" is called *strict subsumption order.*

Positions of a term t (also called *occurrences*) are represented by sequences of natural numbers used to address subterms of t. The concatenation of the sequences p and w is denoted by $p.w$. Two positions p and p' of t are *comparable* if $(\exists w)\ p' = p.w$ or $p = p'.w$, otherwise are *disjoint* positions. Given a position p of t, $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the result of replacing the subterm $t|_p$ by the term s. Let $\overline{p_n}$ be a sequence of disjoint positions of a term t, $t[s_1]_{p_1} \ldots [s_n]_{p_n}$ denotes the result of simultaneously replacing each subterm $t|_{p_i}$ by the term s_i, with $i \in \{1, \ldots, n\}$.

2.1 Term rewriting systems

We limit the discussion to unconditional term rewriting systems[1]. A *rewrite rule* is a pair $l \rightarrow r$ with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The terms l and r are called the *left–hand side* (lhs) and *right–hand side* (rhs) of the rewrite rule, respectively. A *term rewriting system* (TRS) $\mathcal{R}$ is a finite set of rewrite rules.

We are specially interested in TRSs whose associate signature $\mathcal{F}$ can be partitioned into two disjoint sets $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ where $\mathcal{D} = \{\mathcal{R}oot(l) \mid (l \rightarrow r) \in \mathcal{R}\}$ and $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions* or *operations.* Terms built from symbols of the set of variables $\mathcal{X}$ and the set of constructors $\mathcal{C}$ are called *constructor terms.* A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{D}$ and $\overline{d_n}$ are constructor terms. A term $f(\overline{x_n})$, where $\overline{x_n}$ are different variables, is called generic pattern. A TRS is said to be *constructor–based* (CB) if the lhs of its rules are patterns. For CB TRSs, a term t is a *head normal form* (hnf) if t is a variable or $\mathcal{R}oot(t) \in \mathcal{C}$.

A TRS is said to be *left–linear* if for each rule $l \rightarrow r$ in the TRS, the lhs l is a linear term. We say that a TRS is *non–ambiguous* or *non–overlapping* if it does not contain critical pairs (see [10] for a standard definition of critical pair). Left–linear and non–ambiguous TRSs are called *orthogonal* TRSs.

Inductively sequential TRSs are a proper subclass of CB orthogonal TRSs. The definition of this class of programs make use of the notion of *definitional tree.* For the sake of simplicity and because further complications are irrelevant for our study, in the following definition, we ignore the *exempt* nodes that appear in the original definition of [2] and also the *or*–nodes of [18] used in the implementation of Curry [17]. Note also, that or–nodes lead to parallel definitional trees and thus out of the class of inductively sequential systems.

Definition 1. [Partial definitional tree]
Given a CB TRS $\mathcal{R}$, $\mathcal{P}$ is a partial definitional tree *with pattern π if and only if one of the following cases hold:*

---

[1] This is not a true limitation for the expressiveness of a programming language relaying on this class of term rewriting systems [5].

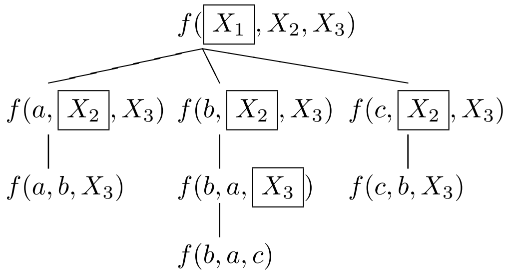

Fig. 1. Definitional tree for the function "f"of Example 1

1. *$\mathcal{P} = rule(\pi, l \to r)$, where π is a pattern and $l \to r$ is a rewrite rule in $\mathcal{R}$ such that π is a variant of l.*
2. *$\mathcal{P} = branch(\pi, o, \overline{\mathcal{P}_k})$, where π is a pattern, o is a variable position of π (called* inductive position*), $\overline{c_k}$ are different constructors, for some $k > 0$, and for all $i \in \{1, \dots, k\}$, $\mathcal{P}_i$ is a partial definitional tree with pattern $\pi[c_i(\overline{x_n})]_o$, where n is the arity of c_i and $\overline{x_n}$ are new variables.*

From a declarative point of view, a partial definitional tree $\mathcal{P}$ can be seen as a set of linear patterns partially ordered by the strict subsumption order "$<$" [4]. Given a defined function f/n, a *definitional tree of f* is a partial definitional tree whose pattern is a generic pattern and its leaves contain variants of all the rewrite rules defining f.

Example 1. Given the rules defining the function $f/3$

$$R_1 : f(a, b, X) \to r_1, \quad R_2 : f(b, a, c) \to r_2, \quad R_3 : f(c, b, X) \to r_3.$$

a definitional tree of f is:

$$\begin{aligned}
&branch(f(X_1, X_2, X_3), 1,\\
&\qquad branch(f(a, X_2, X_3), 2, rule(f(a, b, X_3), R_1)),\\
&\qquad branch(f(b, X_2, X_3), 2, branch(f(b, a, X_3), 2, rule(f(b, a, c), R_2))),\\
&\qquad branch(f(c, X_2, X_3), 2, rule(f(c, b, X_3), R_3)))
\end{aligned}$$

Note that there can be more than one definitional tree for a defined function. It is often convenient and simplifies understanding to provide a graphic representation of definitional trees, where each node is marked with a pattern and the inductive position in branches is surrounded by a box. Figure 1 illustrates this concept.

Definition 2. [Inductively Sequential TRS]
A defined function f is called inductively sequential *if it has a definitional tree. A rewrite system $\mathcal{R}$ is called inductively sequential if all its defined functions are inductively sequential.*

In this paper we are mainly interested in inductively sequential TRSs (or proper subclasses of them) which are called *programs*.

2.2 Definitional trees and Narrowing Implementations into Prolog

Most of the relevant implementations of functional logic languages, which use needed narrowing as operational mechanism, are based on the compilation of the programs written in these languages into Prolog [9, 15, 18, 19]. These implementation systems may be thought as a translation process that essentially consists in the following:

1. An algorithm to transform the program rules in a functional logic program into a set of definitional trees (See [18] and [17] for some of those algorithms).
2. An algorithm that takes the definitional trees as an input parameter and visits their nodes, generating a Prolog clause for each visited node. Since definitional trees contain all the information about the original program as well as information to guide the (optimal) pattern matching process during the evaluation of expressions, the set of generated Prolog clauses is able to simulate the intended narrowing strategy being implemented.

In the case of functional logic programs with a needed narrowing semantics, a generic algorithm for the translation of definitional trees into a set of clauses is given in [15]. When we apply that algorithm to the definitional tree of function f in Example 1, we obtain the following set of Prolog clauses:

```
% Clause for the root node: it exploits the first inductive position
f(X1, X2, X3, H) :- hnf(X1, HX1), f_1(HX1, X2, X3, H).

% Clauses for the remainder nodes:
f_1(a, X2, X3, H):- hnf(X2, HX2), f_1_a_2(HX2, X3, H).
f_1_a_2(b, X3, H):- hnf(r1, H).

f_1(b, X2, X3, H):- hnf(X2, HX2), f_1_b_2(HX2, X3, H).
f_1_b_2(a, X3, H):- hnf(X3, HX3), f_1_b_2_a_3(HX3, H).
f_1_b_2_a_3(c, H):- hnf(r2, H).

f_1(c, X2, X3, H):- hnf(X2, HX2), f_1_c_2(HX2, X3, H).
f_1_c_2(b, X3, H):- hnf(r3, H).
```

where `hnf(T, H)` is a predicate that is true when `H` is the hnf of a term `T`. For this example, the clauses defining the predicate `hnf` are:

```
% Evaluation to head normal form (hnf).
hnf(T, T) :- var(T), !.
hnf(f(X1, X2, X3), H) :- !, f(X1, X2, X3, H).
hnf(T, T). % otherwise the term T is a hnf;
```

The meaning of these set of clauses is very easy to understand. For evaluating a term $t = f(t_1, t_2, t_3)$ to a hnf, first, it is necessary to evaluate (to a hnf) the substerms of t at the inductive positions of the patterns in the definitional tree associated with f (in the order dictated by that definitional tree — see Figure 1). Hence, for our example: we compute the hnf of t_1 and then the hnf of t_2; if b is the hnf of t_1 and a is the hnf of t_2, we have to compute the hnf of t_3; if the hnf of t_3 is c then the hnf of t will be the hnf of r_2 else the computation fails (see the

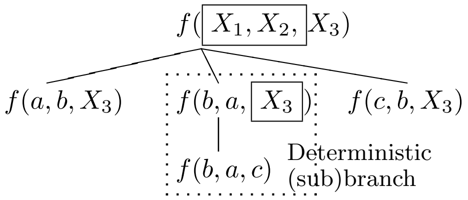

Fig. 2. Refined definitional tree for the function "f"of Example 1

sixth clause). On the other hand, if the hnf of t_1 is a or c it suffices to evaluate t_2 to a hnf, disregarding t_3, in order to obtain the final value. This evaluation mechanism conforms with the needed narrowing strategy of [7], as it has been formally demonstrated in [1].

3 A Refined Representation of Definitional Trees

As we have just seen, building definitional trees is the first step of the compilation process in high level implementations of needed narrowing into Prolog. Therefore, providing a suitable representation structure for the definitional trees associated with a functional logic program may be an important task in order to improve those systems. In this section we give a refined representation of definitional trees that saves memory allocation and is the basis for further improvements.

It is noteworthy that the function f of Example 1 has two definitional trees: the one depicted in Figure 1 and a second one obtained by exploiting position 2 of the generic pattern $f(X_1, X_2, X_3)$. Hence, this generic pattern has two inductive positions. We can take advantage of this situation if we "simultaneously" exploit these two positions to obtain the definitional tree depicted in the Figure 2. This new representation cuts the number of nodes of the definitional tree from eight to five nodes. Note also that this kind of representation reduces the number of possible definitional trees associated with a function. Actually, using the new representation, there is only one definitional tree for f.

The main idea of the refinement is as follows: when a pattern has several inductive positions, exploit them altogether. Therefore we need a criterion to detect inductive positions. This criterion exists and it is based on the concept of uniformly demanded position of [18].

Definition 3. [Uniformly demanded position]
Given a pattern π and a TRS $\mathcal{R}$, Let be $\mathcal{R}_\pi = \{l \to r | (l \to r) \in \mathcal{R} \land \pi \le l\}$. A variable position p of the pattern π is said to be: (i) demanded *by a lhs l of a rule in $\mathcal{R}_\pi$ if $\mathcal{R}oot(l|_p) \in \mathcal{C}$. (ii)* uniformly demanded *by $\mathcal{R}_\pi$ if p is demanded for all lhs in $\mathcal{R}_\pi$.*

We write $\mathcal{UDP}os(\pi)$ to denote the set of uniformly demanded positions of the pattern π. The following proposition establishes a necessary condition for a position of a pattern to be an inductive position.

Proposition 1. *Let $\mathcal{R}$ be an inductively sequential TRS and let π be the pattern of a branch node of a definitional tree $\mathcal{P}$ of a function defined in $\mathcal{R}$. If o is an inductive position of π then o is uniformly demanded by $\mathcal{R}_\pi$.*

Proof. We proceed by contradiction. Assume o is not uniformly demanded by $\mathcal{R}_\pi$. Hence, there must exist some $(l \to r) \in \mathcal{R}_\pi$ such that $\mathcal{R}oot(l|_o) = c \in \mathcal{C}$, and some $(l' \to r') \in \mathcal{R}_\pi$ such that $l'|_o \in \mathcal{X}$. Since o is the inductive position of the branch node whose pattern is π, by definition of definitional tree, $\pi < \pi[c(\overline{x_n})]_o \leq l$ and $\pi \leq l'$. Therefore it is impossible to built a partial definitional tree with leaves l and l' by exploiting the position o, which contradicts the hypothesis that o is an inductive position.

The converse proposition is more involving to formulate but not difficult to establish. In the following, given two partial definitional trees $\mathcal{P}_1$ and $\mathcal{P}_2$, we say $\mathcal{P}_1 \preceq \mathcal{P}_2$ if and only if $\mathcal{P}_1 = \mathcal{P}_2$ or $\mathcal{P}_1 \prec \mathcal{P}_2$, where $\mathcal{P}_1 \prec \mathcal{P}_2$ if $\mathcal{P}_1$ is a proper subtree of $\mathcal{P}_2$.

Proposition 2. *Let $\mathcal{R}$ be an inductively sequential TRS. Let $\mathcal{P}$ a partial definitional tree, with pattern π, and o a variable position of π. If o is uniformly demanded by $\mathcal{R}_\pi$ then there exists a partial definitional tree $\mathcal{P}' \preceq \mathcal{P}$, with pattern π', such that o is an inductive position of π'.*

Proof. By structural induction on the shape of $\mathcal{P}$.

- Base case: $\mathcal{P}$ is a leaf.

  Then, the proposition vacuously holds, since the pattern π of $\mathcal{P}$ does not have uniformly demanded positions (π is a variant of a lhs of a rule R and $\mathcal{R}_\pi$ is a singleton containing the rule R).
- Inductive case: $\mathcal{P}$ is a branch.

  If o is uniformly demanded by $\mathcal{R}_\pi$ then, by definition, o is a variable position of π and for each rule $(l_i \to r_i) \in \mathcal{R}_\pi$, $\mathcal{R}oot(l_i|_o) = c_i \in \mathcal{C}$. Hence, it is possible to build the set of (child) patterns $\pi[c_i(\overline{x_n})]_o$ (where n is the arity of c_i and $\overline{x_n}$ are new variables). Therefore, o is a candidate to be an inductive position. Two cases may occur:
  1. In fact, o is the inductive position of the pattern π and the proof is done, since $\mathcal{P}' = \mathcal{P}$.
  2. Assume $o'' \neq o$ is the inductive position of π. Then, by definition of partial definitional tree, there exists a pattern $\pi'' = \pi[c_i(\overline{x_n})]_{o''}$, which is the pattern of a definitional tree $\mathcal{P}'' \prec \mathcal{P}$. Clearly, o is a variable position of π'' and it is uniformly demanded by the set of rules $\mathcal{R}_{\pi''} \subseteq \mathcal{R}_\pi$. Therefore, by the inductive hypothesis, there exists a partial definitional tree $\mathcal{P}' \prec \mathcal{P}''$, with pattern π', and o is the inductive position of π'. This concludes the proof.

Hence, the concept of uniformly demanded position and Proposition 1 give us a syntactic criterion to detect if a variable position of a pattern is an inductive position or not and, therefore, a guideline to built a definitional tree: (i) Given a branch node, select a uniformly demanded position of its pattern; fix it as an inductive position of the branch node and generate the corresponding child nodes. (ii) If the node doesn't have uniformly demanded positions then there two possibilities: the node is a leaf node, if it is a variant of a lhs of the considered TRS, or it is a "failure" node, and it is impossible to build the definitional tree. The following algorithm, in the style of [17], uses this scheme to build a refined partial definitional tree $\mathit{rpdt}(\pi, \mathcal{R}_\pi)$ for a pattern π and rules $\mathcal{R}_\pi = \{l \to r \mid (l \to r) \in \mathcal{R} \wedge \pi \leq l\}$:

1. If $\mathcal{UDP}os(\pi) = \emptyset$ and there is only one rule $(l \to r) \in \mathcal{R}_\pi$ and a renaming ρ such that $\pi = \rho(l)$:
$$\mathit{rpdt}(\pi, \mathcal{R}_\pi) = \mathit{rule}(\pi, \rho(l) \to \rho(r));$$
2. If $\mathcal{UDP}os(\pi) \neq \emptyset$ and for all $(c_{i_1}, \ldots, c_{i_m}) \in \mathcal{C}_\pi$, $\mathcal{P}_i = \mathit{rpdt}(\pi_i, \mathcal{R}_{\pi_i}) \neq \mathtt{fail}$:
$$\mathit{rpdt}(\pi, \mathcal{R}_\pi) = \mathit{branch}(\pi, \overline{o_m}, \overline{\mathcal{P}_k});$$
where $\overline{o_m}$ is the sequence of uniformly demanded positions in $\mathcal{UDP}os(\pi)$, $\mathcal{C}_\pi = \{(c_{i_1}, \ldots, c_{i_m}) | (l_i \to r_i) \in \mathcal{R}_\pi \wedge \mathcal{R}oot(l_i|_{o_1}) = c_{i_1} \wedge \ldots \wedge \mathcal{R}oot(l_i|_{o_m}) = c_{i_m}\}$, $k = |\mathcal{C}_\pi| > 0$, $\pi_i = \pi[c_{i_1}(\overline{x_{n_{i_1}}})]_{o_1} \ldots [c_{i_m}(\overline{x_{n_{i_m}}})]_{o_m}$ and $\overline{x_{n_{i_1}}}, \ldots, \overline{x_{n_{i_m}}}$ are new variables.
3. Otherwise, $\mathit{rpdt}(\pi, \mathcal{R}_\pi) = \mathtt{fail}$.

Given an inductively sequential TRS $\mathcal{R}$ and a n–ary defined function f in $\mathcal{R}$, the definitional tree of f is $\mathit{rdt}(f, \mathcal{R}) = \mathit{rpdt}(\pi_0, \mathcal{R}_{\pi_0})$ where $\pi_0 = f(\overline{x_n})$. Note that, for an algorithm like the one described in [17] the selection of the inductive positions of the pattern π is non–deterministic, if $\mathcal{UDP}os(\pi) \neq \emptyset$. Therefore, it is possible to build different definitional trees for an inductively sequential function, depending on the inductive position which is selected. On the contrary, our algorithm deterministically produces a single definitional tree for each inductively sequential function. Note also that it matches the more informal algorithm that appears in [17] when, for each branch node, there is only one inductive position.

We illustrate the previous algorithm and last remarks by means of a new example.

Example 2. Given the rules defining the function $f/2$

$$R_1 : f(0, 0) \to 0, R_2 : f(s(X), 0) \to s(0), R_3 : f(s(X), s(s(Y))) \to f(X, Y).$$

the last algorithm builds the following definitional tree for f:

$$\begin{array}{l} \mathit{branch}(f(X_1, X_2), (1, 2), \\ \qquad \mathit{rule}(f(0, 0), R_1), \\ \qquad \mathit{rule}(f(s(X_3), 0), R_2), \\ \qquad \mathit{branch}(f(s(X_3), s(X_4)), (2.1), \mathit{rule}(f(s(X_3), s(s(X_5))), R_3)) \end{array}$$

which is depicted in Figure 3. The algorithm for generating definitional trees of [17] may build two definitional trees for f (depending on whether position 1 or position 2 is selected as the inductive position of the generic pattern $f(X_1, X_2)$). Both of these trees have seven nodes, while the new representation of Figure 3 reduces the number of nodes of the definitional tree to five nodes.

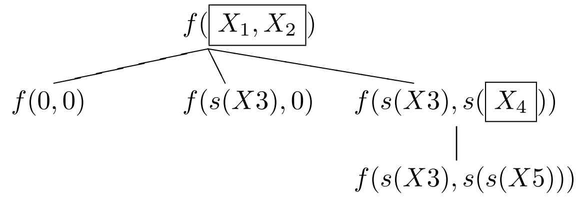

Fig. 3. Refined definitional tree for the function “f”of Example 2

As it has been proposed in [9], it is possible to obtain a simpler translation scheme of functional logic programs into Prolog if definitional trees are first compiled into *case expressions*. That is, functions are defined by only one rule where the lhs is a generic pattern and the rhs contains case expressions to specify the pattern matching of actual arguments. The use of case expressions doesn't invalidate our argumentation. Thus, we can transform the definitional tree of Example 2 in the following case expression:

$$
\begin{array}{ll}
f(X_1, X_2) = \texttt{case}\ \ (X_1, X_2)\ \ \texttt{of} & \\
\qquad\qquad\quad (0, 0) & \to 0 \\
\qquad\qquad\quad (s(X_3), 0) & \to s(0) \\
\qquad\qquad\quad (s(X_3), s(X_4)) & \to \texttt{case}\ \ (X_4)\ \ \texttt{of} \\
 & \quad\; s(X_5) \to f(X_3, X_5)
\end{array}
$$

A case expression, like this, will be evaluated by reducing a tuple of arguments to their hnf and matching them with one of the patterns of the case expression.

4 Improving Narrowing Implementations into Prolog

This section discuses two improvements in the translation of non-strict functional logic programs into Prolog which are based in the analysis of definitional trees. These translation techniques can be applied jointly or separately.

4.1 Translation Based on Refined Definitional Trees

The refined representation of definitional trees introduced in Section 3 is very close to the standard representation of definitional trees, but it is enough to provide further improvements in the translation of functional logic programs into Prolog.

It is easy to adapt the translation algorithm that appears in [15] to use our refined representation of definitional trees as input.

$$
\begin{array}{l}
Trans(\overline{\mathcal{T}_n}, p) := Trans(\mathcal{T}_1), \ldots, Trans(\mathcal{T}_n); \\
Trans(rule(\pi, \pi \to r), p) := \\
\qquad produceCode : \\
\qquad \boxed{f_p(\overline{t_m}, \texttt{H})\ \texttt{:-}\ \ \texttt{hnf}(r, \texttt{H}).}
\end{array}
$$

```
$Trans(branch(\pi,(o_1,\ldots,o_n),\overline{\mathcal{T}'}),p) :=$
    $produceCode:$
        $f_p(\overline{t_m},$H$)$ :-
        hnf($X_1$, $Y_1$), ..., hnf($X_n$, $Y_n$), $f_{p\cup\{o_1,\ldots,o_n\}}(\overline{t'_m},$H$)$.

    $Trans(\overline{\mathcal{T}'},p\cup\{o_1,\ldots,o_n\});$
where $\pi = f(\overline{t_m})$, $\pi|_{o_1} = X_1,\ldots,\pi|_{o_n} = X_n$ and $\pi[Y_1]_{o_1}\ldots[Y_n]_{o_n} = f(\overline{t'_m})$.
```

If we apply this slightly different algorithm to the refined definitional tree of Figure 2, we obtain the following set of clauses, where the inductive positions 1 and 2 are exploited altogether:

```
% Clause for the root node:
f(X1, X2, X3, H) :- hnf(X1, HX1), hnf(X2, HX2), f_1_2(HX1, HX2, X3, H).

% Clauses for the remainder nodes:
f_1_2(a, b, X3, H):- hnf(r1, H).

f_1_2(b, a, X3, H):- hnf(X3, HX3), f_1_2_b_a(HX3, H).
f_1_2_b_a(c, H):- hnf(r2, H).

f_1_2(c, b, X3, H):- hnf(r3, H).
```

where we have cut the number of clauses with regard to the standard representation into Prolog (of the rules defining function f) presented in Section 2.2. The number of clauses is reduced in the same proportion the number of nodes of the standard definitional tree for f were cut. As we are going to show in Section 5, this refined translation technique is able to improve the efficiency of the implementation system.

On the other hand, it is important to note that the kind of improvements we are mainly studying in this subsection can not be obtained by an unfolding transformation process applied to the set of clauses produced by the standard algorithm of [15]: In fact, it is not possible to obtain the above set of clauses by an unfolding transformation of the set of clauses shown in Section 2.2.

4.2 Selective Unfolding Transformations

The analysis of definitional trees provides further opportunities for improving the translation of inductively sequential programs into Prolog. For instance, we can take notice that the definitional tree of function f in Example 1 has a “deterministic” (sub)branch, that is, a (sub)branch whose nodes have only one child (see Figure 2). This knowledge can be used as an heuristic guide for applying determinate unfolding transformation steps selectively.

Note that, for the example we are considering, the clauses:

```
f_1_2(b, a, X3, H):- hnf(X3, HX3), f_1_2_b_a(HX3, H). %% (C1)
f_1_2_b_a(c, H):- hnf(r2, H).                         %% (C2)
```

can be merged into:

```
f_1_2(b, a, X3, H):- hnf(X3, c), hnf(r2, H).         %% (C')
```

by applying a safe unfolding transformation in the style of Tamaki and Sato [23] but restricting ourselves to determinate atoms [11, 12] (i.e., an atom that matches exactly one clause head in the Prolog code): we get clause `C1` (the unfolded clause) and we select the atom `f_1_2_b_a(HX3, H)` in its body; this atom call is unifiable with the head of clause `C2` (the *unique* unfolding clause for this atom call), with most general unifier $\sigma = \{\texttt{HX3/c}\}$ (actually, a matcher); Therefore, we can perform a transformation step where `C1` and `C2` are instantiated applying σ, the atom call is unfolded and, afterwards, clauses `C1` and `C2` are replaced by `C'`.

This selective unfolding is preferable to a generalized (post–compilation) unfolding transformation process[2] which may degrade the efficiency of the compiled Prolog code. Moreover, this selective unfolding transformation can be easily integrated inside the compilation procedure described in [15]. It suffices to introduce an additional case in order to treat deterministic (sub)branches:

```
$Trans(\overline{\mathcal{T}_n}, p) := Trans(\mathcal{T}_1), \ldots, Trans(\mathcal{T}_n);$
  ⋮
$Trans(branch(\pi, o, \mathcal{T}^1), p) :=$
    if $\mathcal{T}^n = branch(\pi_n, o_n, \overline{\mathcal{T}'})$
      $produceCode$ :
        $f_p(\overline{t_m}$, H) :-
        hnf($X$, $\pi_1|_o$), hnf($\pi_1|_{o_1}$, $\pi_2|_{o_1}$), $\ldots$, hnf($\pi_{n-1}|_{o_{n-1}}$, $\pi_n|_{o_{n-1}}$),
        hnf($\pi_n|_{o_n}$, Y), $f_{p\cup\{o,o_1,\ldots,o_n\}}(\overline{t'_m}$, H).
      $Trans(\overline{\mathcal{T}'}, p \cup \{o, o_1, \ldots, o_n\});$
     esle if $\mathcal{T}^n = rule(\pi_n, \pi_n \to r)$
      $produceCode$ :
        $f_p(\overline{t_m}$, H) :-
        hnf($X$, $\pi_1|_o$), hnf($\pi_1|_{o_1}$, $\pi_2|_{o_1}$), $\ldots$, hnf($\pi_{n-1}|_{o_{n-1}}$, $\pi_n|_{o_{n-1}}$),
        hnf($r$, H).
where $\pi = f(\overline{t_m})$, $\pi|_o = X$, $\mathcal{T}^1, \ldots, \mathcal{T}^n$ is the sequence of nodes in the
deterministic (sub)branch with $\mathcal{T}^i = branch(\pi_i, o_i, \mathcal{T}^{i+1})$, and $\pi_n[\texttt{Y}]_{o_n} = f(\overline{t'_m})$.
  ⋮
```

Roughly speaking, the new case in the algorithm of [15] can be understood as follows. If there exists a deterministic (sub)branch visit its nodes in descending order forcing that the evaluation (to hnf) of the subterms at the inductive position o of a term be the flat constructor at position o of the child node. Proceed in this way until: i) a non deterministic node is reached; or ii) a leaf node is

---

[2] That is, a transformation process where non determinate atom calls are unfolded too.

reached and, in this case, evaluate the rhs of the rule to its hnf and stop the translation.

The last algorithm allows some improvements we have omitted by the lack of simplicity. First, it is possible to eliminate redundant arguments. Second, it is possible to exploit rule nodes (i.e., an atom call like `hnf(`r`, H)`) to perform an additional determinate unfolding step[3]. Having all this in consideration, the following example illustrates the algorithm.

Example 3. Given the rules defining the partial function *even*:

$$R_1 : even(0) \to true, \qquad R_2 : even(s(s(X)) \to even(X).$$

The definitional tree for that function is:

$$\begin{array}{l} branch(even(X_1), 1, \\ \qquad rule(even(0), R_1), \\ \qquad branch(even(s(X_2)), 1.1, rule(even(s(s(X_3)), R_2)))) \end{array}$$

and the Prolog code generated by the $Trans$ algorithm is:

```
% Evaluation to head normal form (hnf).
hnf(even(X1), H) :- !, even(X1, H).

% Clause for the root node: it exploits the first inductive position
even(X1, H) :- hnf(X1, HX1), even_1(HX1, H).

even_1(0, true).
% Clause for the deterministic (sub)branch:
even_1(s(X2), H) :- hnf(X2, s(X3)), even(X3, H).
```

Note as the determinate call `hnf(even(X3), H)` has been unfolded (into the call `even(X3, H)` using the first rule for evaluating a hnf).

Therefore, our $Trans$ algorithm, guided by the structure of a definitional tree, is able to reproduce the effect of a post-compilation unfolding transformation when it is applied selectively on determinate atom calls in the standard compiled Prolog code.

5 Experiments

We have made some experiments to verify the effectiveness of our proposals. We have instrumented the Prolog code obtained by the compilation of simple Curry programs by using the `curry2prolog` compiler of PAKCS [8] (an implementation of the multi–paradigm declarative language Curry [17]). We have introduced our translation techniques in the remainder Prolog code. For our first translation technique, the one using the refined representation of definitional trees, the results of the experiments are shown in Table 1. Runtime and memory occupation were measured on a Sun4 Sparc machine, running sicstus v3.8 under SunOS

---

[3] These improvements are implemented in the `curry2prolog` compiler of PAKCS [8] for the standard cases.

v5.7. The "`Speedup`" column indicates the percentage of execution time saved by our translation technique. The values shown on that column are the percentage of the quantity computed by the formula $(t_1 - t_2)/t_1$, where t_1 and t_2 are the average runtimes, for several executions, of the proposed terms (goals) and Prolog programs obtained when we don't use (t_1) and we use (t_2) our translation technique. The "`G. stack Imp.`" column reports the improvement of memory occupation for the computation. We have measured the percentage of global stack allocation. The amount of memory allocation measured between each execution remains constant. Most of the benchmark programs are extracted from [17] and the standard prelude for Curry programs with slight modifications[4]. For the benchmark programs `family` and `xor` we evaluate all outcomes. The natural numbers are implemented in Peano notation, using `zero` and `succ` as constructors of the sort. In the `zip` and `zip3` programs the input terms $L1$ and $L2$ are lists of length 9.

Table 1. Runtime speed up and memory usage improvements for some benchmark programs and terms.

`Benchmark`	`Term`	`Speedup`	`G. stack Imp.`
`family`	$grandfather(\_,\_)$	19.9%	0%
`geq`	$geq(100000, 99999)$	4.6%	16.2%
`geq`	$geq(99999, 100000)$	4.3%	16.2%
`xor`	$xor(\_,\_)$	18.5%	0%
`zip`	$zip(L1, L2)$	3.6%	5.5%
`zip3`	$zip3(L1, L2, L2)$	4.5%	10%
	`Average`	9.2%	7.9%

Regarding the second translation technique, the one which implements selective unfolding transformations, for the benchmark program of Example 3 we obtain an average speedup of 11.7% and an improvement in memory usage of 14.7% for the term (goal) $even(16000)$.

Finally, the function f in Example 2 is a (rather artificial) function where we can apply both of our translation techniques, allowing us to illustrate their effect acting jointly and separately. Table 2 shows some different benchmarks, where we are comparing the efficiency of the Prolog code generate by the `curry2prolog` compiler with regard to the Prolog code generated by our first translation technique (`ref`), the Prolog code generated by our second translation technique (`det`) and the Prolog code generated when we apply both of them (`all`). The results show that the effect of our techniques is accumulative.

---

[4] For example, `zip` (resp. `zip3`) is adapted for combining two (resp. three) lists of elements of equal length into one list of pairs (resp. triples) of the corresponding elements. However, this function also may be useful in a practical context (see [16], page 280).

Table 2. Runtime speed up and memory usage improvements for the function f in Example 2.

	$f(50000, 100000)$		$f(100000, 50000)$	
`Benchmark`	`Speedup`	`G. stack Imp.`	`Speedup`	`G. stack Imp.`
`ref`	9.5%	11.4%	9.1%	10.8%
`det`	7.7%	11.4%	7.1%	10.8%
`all`	18.9%	22.8%	18.3%	21.6%

More detailed information about the experiments and benchmark programs can be found in `http://www.inf-cr.uclm.es/www/pjulian/publications.html`.

6 Discussion and Related Work

In this section we discuss some important issues and we put them in relation to other research on functional logic programming and logic programming when it is convenient.

Elimination of *ad hoc* artifices. It is noteworthy that, in some cases, the benefits of our first translation scheme are obtained in an *ad hoc* way in actual needed narrowing into Prolog implementation systems. For instance, the standard definition of the strict equality used in non–strict functional logic languages is [13, 21]:

$$\begin{aligned} c == c &\rightarrow true \\ c(\overline{X_n}) == c(\overline{Y_n}) &\rightarrow X_1 == Y_1 \&\& \ldots \&\& X_n == Y_n \end{aligned}$$

where c is a constructor of arity 0 in the first rule and arity $n > 0$ in the second rule. There is one of these rules for each constructor that appears in the program we are considering. Clearly, the strict equality has an associate definitional tree whose pattern $(X_1 == X_2)$ has two uniformly demanded positions (positions 1 and 2) and, therefore, it can be translated using our first technique, that produces a set of Prolog clauses similar to the one obtained by the `curry2prolog` compiler. In fact, the `curry2prolog` compiler translates these rules into the following set of Prolog clauses[5]:

```
hnf(A==B,H):-!,seq(A,B,H).

seq(A,B,H):-hnf(A,F),hnf(B,G),seq_hnf(F,G,H).

seq_hnf(true,true,H):-!,hnf(true,H).
seq_hnf(false,false,H):-!,hnf(true,H).
seq_hnf(c,c,H):-!,hnf(true,H).
seq_hnf(c(A1,...,Z1),c(A2,...,Z2),H):-!,
```

---

[5] Note that, we have simplified the code produced by the `curry2prolog` compiler in order to increase its readability and facilitate the comparison with our proposal.

```
hnf(&&(A1==A1,&&(B1==B2,&&(...,&&(Z1==Z2,true)))),H).
```

Thus, the `curry2prolog` compiler produces an optimal representation of the strict equality which is treated as a special system function with an *ad hoc* predefined translation into Prolog, instead of using the standard translation algorithm which is applied for the translation of user defined functions.

Failing derivations. Our first contribution, as well as the overall theory of needed evaluation, is interesting for computations that succeed. However it is important to say that some problems may arise when a computation does not terminate or fails. For example, given the (partial) function

$$f(a, a) \to a$$

the standard compilation into Prolog is:

```
f(A,B,C)   :- hnf(A,F), f_1(F,B,C).
f_1(a,A,B) :- hnf(A,E), f_1_a_2(E,B).
f_1_a_2(a,a).
```

while our first translation technique produces:

```
f(A,B,C) :- hnf(A,F), hnf(B,G), f_1(F,G,C).
f_1(a,a,a).
```

Now, if we want to compute the term `f(b, expensive_term)`, the standard implementation detects the failure after the computation of the first argument. On the other hand, the new implementation computes the expensive term (to hnf) for nothing. Of course, the standard implementation has problems too —e.g. if we compute the term `f(expensive_term, b)`, it also computes the expensive term (to hnf)—, but it may have a better behavior on this problem. Thus, in a sequential implementation, the performance of our first translation technique may be in danger when subterms, at uniformly demanded positions, are evaluated (to hnf) jointly with an other subterm whose evaluation (to hnf) produces a failure. An alternative to overcome this practical disadvantage is to evaluate these subterms in parallel, introducing monitoring techniques able to detect the failure as soon as possible and then to stop the streams of the computation.

Clause indexing and direct implementation into Prolog. Clause indexing is a technique, used in the implementation of Prolog compilers, that aims to reduce the number of clauses on which unification with a goal is performed. In general, indexing techniques are based on the inspection of the outermost function symbol of one o more arguments in a clause head. If the predicate symbol and the respective indexed symbols of the clause head and the goal coincide, then the clause is selected as part of the *filtered set*. Afterwards, the set of clauses in the filtered set (presumably smaller than the original one) is attempted to unify with the goal. Most sophisticated indexing techniques as those described in [22] perform indexing on all non variable symbols of a clause head (losing no significant structural information). Also, these techniques are able to obtain the unifier during the indexing process. Although it seems to have some similarities between indexing techniques and the standard operational mechanism of functional logic languages, there is a big difference: in the context of pure logic languages terms are *dead* structures. However, in the context of this work, the concept of evaluation strategy relays on the existence and manipulation of nested *alive* terms. The needed narrowing strategy, as defined in [7], is an application from terms and partial definitional trees to sets of triples (position, rule, substitution), where each triple gives the position of a term, the rule of the program and the unifier substitution (not necessarily a most general one) used in a narrowing step. Our work is concerned in the optimization of certain implementation techniques of needed narrowing into Prolog.

On the other hand, it is possible a direct representation of a function into Prolog which is often more efficient, since term structures with nested functions calls are not generated. However, a direct implementation corresponds to a call-by-value strategy, that lacks some valuable properties (as the ability of handle infinite data structures or a good termination behavior) [15].

Determinate unfolding. Determinate unfolding [11, 12] has been proposed as a way to ensure that the specialization of a logic program will never duplicate computations. The advantages of determinate unfolding transformations, in the context of the implementation of functional logic languages into Prolog, were suggested in [15] and [9]. They proposed to apply determinate unfolding as a post-compilation process but actually, in the `curry2prolog` compiler, determinate unfolding steps are only applied to unfold the atom calls produced by rule nodes. The novel of our proposal is that it exploits all opportunities for determinate unfolding in a systematic way and it is embedded inside the compilation process.

7 Conclusions

In this paper we have introduced a refined representation of definitional trees that eliminates the indeterminism in the selection of definitional trees in the context of the needed narrowing strategy (Actually, there is only one refined definitional tree for each inductively sequential function). We have defined two translation techniques based in the analysis of (refined) definitional trees. Although the results of the experiments section reveals a good behavior of these translation techniques, it is difficult to evaluate which may be their impact over the whole system, since the improvements appear when we can detect patterns that have several uniformly demanded positions or the existence of deterministic (sub)branches in a (refined) definitional tree. Nevertheless, our work shows that there is a potential for the improvement of actual (needed) narrowing implementation systems: we obtain valuable improvements of execution time and memory allocation when our translation techniques are relevant. For the case of inductively sequential functions without the features aforementioned, our translation schemes are conservative and don't produce runtime speedups or memory allocation improvements. Although failing derivations are rather a problematic case where the performance of our first translation technique may be in danger, we can deal with these problem by introducing concurrent computations, in order to guarantee that slowdowns, with regard to standard implementations of needed narrowing into Prolog, are not produced. Hence, the occurrence of several inductive position in a pattern can be considered as a signal for exploiting implicit parallelism.

On the other hand, our simple translation techniques are able to eliminate some *ad hoc* artifices in actual implementations of (needed) narrowing into Prolog, providing a systematic and efficient translation mechanism. Moreover, the ideas we have just developed can be introduced with a modest programming effort in standard implementations of needed narrowing into Prolog (such as the PAKCS [8] implementation of Curry) and in other implementations based on the use of definitional trees (e.g., the implementation of the functional logic language $\mathcal{TOY}$[19]), since they don't modify their basic structures.

Acknowledgements

We gratefully acknowledge Sergio Antoy for clarifying us some aspects of the theoretical/practical dimension of the failing derivations problem. Also, we thank Ginés Moreno for many useful discussions on the advantages of introducing selective unfolding transformations.

References

1. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Uniform Lazy Narrowing. *Journal of Logic and Computation*, 13(2):27, March/April 2003.
2. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, Berlin, 1992.
3. S. Antoy. Needed Narrowing in Prolog. Technical Report Technical Report 96-2, Portland State University, 1996. Full version of extended abstract in [6].
4. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, volume 1298 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, Berlin, 1997.
5. S. Antoy. Constructor-based conditional narrowing. In *Proc. of 3rd Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'01).* Springer LNCS, 2001.
6. S. Antoy. Needed Narrowing in Prolog. In *Proc. of the 8th Int'l Symp. on Prog. Lang., Implementations, Logics, and Programs*, Lecture Notes in Computer Science, pages 473–474. Aachen University, Germany, Sept. 1996.
7. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, July 2000.
8. S. Antoy, M. Engelke, M. Hanus, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. Pakcs 1.5 : The Portland Aachen Kiel Curry System User Manual. Technical Report Version of May, 25, University of Kiel, Germany, 2003. Available from URL: `http://www.informatik.uni-kiel.de/ pakcs/`.

9. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into prolog. In *Proc. Third International Workshop on Frontiers of Combining Systems, FroCoS 2000*, pages 171–185. Springer LNCS 1794, 2000.
10. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
11. J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. ACM, New York, 1993.
12. J. Gallagher and M. Bruynooghe. Some Low-Level Source Transformations for Logic Programs. In M. Bruynooghe, editor, *Proc. of 2nd Workshop on Meta-Programming in Logic*, pages 229–246. Department of Computer Science, KU Leuven, Belgium, 1990.
13. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.
14. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
15. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation, LOPSTR'95*, pages 252–266. Springer LNCS 1048, 1995.
16. M. Hanus and F. Huch. An open system to support web-based learning. In *Proceedings of the 12th International Workshop on Functional and (Constraint) Logic Programming, WFLP 2003, Valencia, Spain, June 12-13, 2003*, pages 269–282. Universidad Politécnica de Valencia, 2003.
17. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry`, 1999.
18. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93, Tallinn (Estonia)*, pages 184–200. Springer LNCS 714, 1993.
19. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
20. J. G. Moreno, M. H. González, F. López-Fraguas, and M. R. Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *Journal of Logic Programming*, 1(40):47–87, 1999.
21. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.
22. R. Ramesh, I. Ramakrishnan, and D. Warren. Automata–Driven Indexing of Prolog Clauses. *Journal of Logic Programming*, 23(2):151–202, 1995.
23. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.