Refined Definitional Trees and Prolog Implementations of Narrowing

Pascual Julián Iranzo[1]

Departamento de Informática
Universidad de Castilla–La Mancha
Ciudad Real, Spain
Pascual.Julian@uclm.es

Abstract. This paper describes how high level implementations of (needed) narrowing into Prolog can be improved by introducing a refined representation of definitional trees that handles properly the knowledge about the inductive positions of a pattern. We define some generic algorithms that allow us to transform a functional logic program into a set of Prolog clauses which 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 proposal by means of some simple examples.

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

1 Introduction

Functional logic programming [12] 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, 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

[1] Supported by CICYT TIC 2001-2705-C03-01, Acción Integrada Hispano-Italiana HI2000-0161, Acción Integrada Hispano-Alemana HA2001-0059, and the Valencian Research Council under grant GV01-424.

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 [17] 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, 8, 13, 15] and [18]). Most of these implementation systems mainly rely on a two-phase transformation procedure that consists of: a suitable representation structure for the definitional trees associated with a functional logic program; and an algorithm that takes the above representation of definitional trees as an input parameter and translates it into a set of Prolog clauses able to simulate the narrowing strategy being implemented.

This paper investigates how a refined representation of definitional trees can introduce improvements in the quality of the code generated by the transformation scheme we have just described.

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 building them in the style of [14]. Section 4 discusses how to translate functional logic programs into Prolog, taking advantage of the new representation of definitional trees to improve (needed) narrowing implementations. Section 5 presents some experiments that show the effectiveness of our proposal. Section 6 contains our conclusions. Finally we briefly discuss the lines of future work.

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, $t[s_1]_{p_1} \dots [s_n]_{p_n}$ denotes the result of simultaneously replacing each subterm $t|_{p_i}$ by the term s_i, with $i \in \{1, \dots, n\}$.

2.1 Term rewriting systems

We limit the discussion to unconditional term rewriting systems[1]. A *rewrite rule* is a pair $l \to r$ 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 $\mathcal{D} = \{\mathcal{R}oot(l) \mid (l \to r) \in \mathcal{R}\}$ 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 $l \to r$ 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 [15] used in the implementation of Curry [14]. 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. *$\mathcal{P} = rule(\pi, l \to r)$, where π is a pattern and $l \to r$ is a rewrite rule in $\mathcal{R}$ such that π is a variant of l.*

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

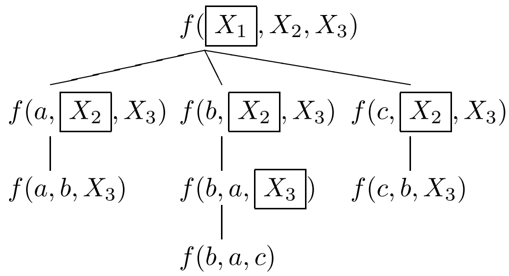

Fig. 1. Definitional tree for the function "f"of Example 1

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 $i \in \{1, \ldots, k\}$, $\mathcal{P}_i$ 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:

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

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 [8, 13, 15, 16]. 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 [15] and [14] 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 [13]. 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 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].

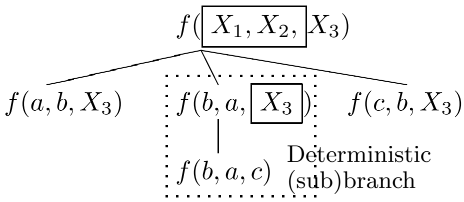

Fig. 2. Refined definitional tree for the function "f" of Example 1

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 to 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 [15].

Definition 3. [Uniformly demanded position]
Given a pattern π and a TRS $\mathcal{R}$, Let be $\mathcal{R}_\pi = \{l \to r | (l \to r) \in \mathcal{R} \wedge \pi \leq l\}$. 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.

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 [14], uses this scheme to build a refined partial definitional tree $rpdt(\pi, \mathcal{R}_\pi)$ 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 $\pi = \rho(l)$:

   $$rpdt(\pi, \mathcal{R}_\pi) = rule(\pi, \rho(l) \to \rho(r));$$

2. If $\mathcal{UDP}os(\pi) \neq \emptyset$ and for all $(c_{i_1}, \ldots, c_{i_m}) \in \mathcal{C}_\pi$, $\mathcal{P}_i = rpdt(\pi_i, \mathcal{R}_{\pi_i}) \neq \texttt{fail}$:

   $$rpdt(\pi, \mathcal{R}_\pi) = branch(\pi, \overline{o_m}, \overline{\mathcal{P}_k});$$

   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, $rpdt(\pi, \mathcal{R}_\pi) = \texttt{fail}$.

Given an inductively sequential TRS $\mathcal{R}$ and a n–ary defined function f in $\mathcal{R}$, the definitional tree of f is $rdt(f, \mathcal{R}) = rpdt(\pi_0, \mathcal{R}_{\pi_0})$ where $\pi_0 = f(\overline{x_n})$. Note that, for an algorithm like the one described in [14] 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 [14] 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 $f/2$

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

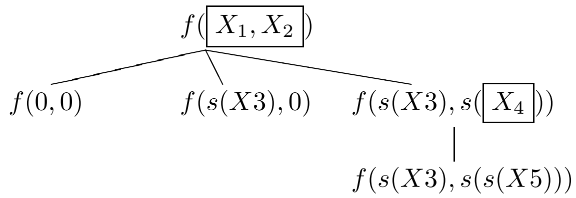

Fig. 3. Refined definitional tree for the function "f"of Example 2

the last algorithm builds the following definitional tree for f:

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

which is depicted in Figure 3. The algorithm for generating definitional trees of [14] 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.

As it has been proposed in [8], 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 expresions 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:

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

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

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 [13] to use our refined representation of definitional trees as input. If we apply this slightly different algorithm to the refined definitional tree of Figure 2, we obtain the following set of clauses:

```
% 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 the next section, this refined translation technique is able to improve the efficiency of the implementation system.

Note that the analysis of definitional trees provide 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 unfolding transformation steps selectively. Hence, 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).
f_1_2_b_a(c, H):- hnf(r2, H).
```

can be transformed in:

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

We think this selective unfolding is preferable to the more costly and generalized (post–compilation) unfolding transformation process suggested in [13] and [8].

On the other hand, it is important to note that the kind of improvements we are mainly studying in this work can not be obtained by an unfolding transformation process applied to the set of clauses produced by the standard algorithm of [13]: In fact, it is not possible to obtain the above set of clauses by unfolding transformation of the set of clauses shown in Section 2.2.

5 Experiments

We have made some experiments to verify the effectiveness of our proposal. We have instrumented the Prolog code obtained by the compilation of simple Curry programs by using the `curry2prolog` compiler of Pakcs [9] (an implementation of the multi–paradigm declarative language Curry [14]). We have introduced our translation technique in the remainder Prolog code. 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 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 in each execution remains constant. Most of the benchmark programs are extracted from [14] and the standard prelude for Curry programs with slight modifications. 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%

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 Conclusions

Although the results of the preceding section reveals a good behavior of our translation technique, it is difficult to evaluate what may be its impact over the whole system, since the improvements appear when we can detect patterns which have several uniformly demanded positions. For the case of inductively sequential functions without this feature, our translation scheme is conservative and doesn't produce runtime speedups or memory allocation improvements. On the other hand it is noteworthy that, in some cases, the benefits of our 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 [11, 18]:

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

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 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[2]:

```
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):-!,
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.

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

$$f(a, a) \rightarrow a$$

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 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

[2] 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) 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 translation technique may be endanger 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.

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 technique is relevant, without an appreciable slowdown in the cases where it is not applicable. Also, our simple translation technique is 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 [9] implementation of Curry) and in other implementations based on the use of definitional trees (e.g., the implementation of the functional logic language $\mathcal{TOY}$[16]), since they don't modify their basic structures.

7 Future Work

Failing derivations are rather a problematic case where the performance of our translation technique may be endanger. We want to deal with these problem in order to guarantee that slowdowns, with regard to standard implementations of needed narrowing into Prolog, are not produced. Also we like to study how *clause indexing* [19], in the context of Prolog implementation, relates with our work.

On the other hand, we aim to investigate how definitional trees may be used as a guide to introduce selective program transformation techniques.

Acknowledgements

I gratefully acknowledge the anonymous referees for many useful suggestions and Sergio Antoy for clarifying me some aspects of the theoretical/practical dimension of this problem.

References

1. M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Uniform Lazy Narrowing. *Journal of Logic and Computation* 13(2), 2003. Short preliminary version in Proc. of WFLP 2002, available at `http://www.inf-cr.uclm.es/www /pjulian`

2. S. Antoy. Definitional trees. In *Proc. of ALP'92*, volume 632 of *LNCS*, pages 143–157. Springer-Verlag, 1992.
3. S. Antoy. Needed Narrowing in Prolog. 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 ALP'97*, volume 1298 of *LNCS*, pages 16–30. Springer-Verlag, 1997.
5. S. Antoy. Constructor-based conditional narrowing. In *Proc. of (PPDP'01)*. Springer LNCS, 2001.
6. S. Antoy. Needed Narrowing in Prolog. In *Proc. of PLILP'96*, *LNCS*, pages 473–474. 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 and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. FroCoS 2000*, pages 171–185. Springer LNCS 1794, 2000.
9. S. Antoy, M. Hanus, J. Koj, P. Niederau, R. Sadre, and F. Steiner. Pacs 1.3 : The Portland Aachen Kiel Curry System User Manual. Technical Report Version of December, 4, University of Kiel, Germany, 2000. Available from URL: `http://www.informatik. uni-kiel.de/~pakcs/`
10. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
11. 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.
12. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
13. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. LOPSTR'95*, pages 252–266. Springer LNCS 1048, 1995.
14. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www. informatik.uni-kiel.de/~curry`, 1999.
15. R. Loogen, F. López-Fraguas, and M. Rodríguez - Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of PLILP'93*, pages 184–200. Springer LNCS 714, 1993.
16. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
17. 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.
18. 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.
19. R. Ramesh and I. V. Ramakrishnan and D. S. Warren. Automata–Driven Indexing of Prolog Clauses. *Journal of Logic Programming*, 23(2):151–202. 1995.