Needed Narrowing in Prolog

Sergio Antoy

Portland State University

TR 96-2

May 17, 1996

Abstract

We describe the implementation of needed narrowing deployed in a compiler of a functional-logic language and present a few related concepts and results. Our implementation is obtained by translating rewrite rules into Prolog source code and optionally applying a set of optimizations to this code. We benchmark the effectiveness of each individual optimization. We show that our implementation is more efficient than all other previously proposed similar implementations. We measure both execution times, as is customarily done, and memory allocation that turns out to be a significant factor. We solve equations using a *semi*-strict equality relation that generalizes classic strict equality with sometimes a smaller search space. We give a new, more declarative and accessible formulation of a definitional tree, a crucial concept of our approach, and we present a simple algorithm to build definitional trees. We briefly explore a notion of simplification that is applicable to and sometimes beneficial for computations in inductively sequential systems, where classic simplification is not applicable.

This work has been supported in part by the National Science Foundation under grant CCR-9406751.
Author's address: Dept. of Computer Science, P.O. Box 751, Portland, OR 97207, `antoy@cs.pdx.edu`.

1 Introduction

We describe the implementation of an advanced narrowing strategy, *needed narrowing* [3], in Prolog. Our implementation has been deployed in an extension [4] of the Gödel compiler [12], which translates Gödel source code into Prolog source code. Our implementation is high-level, portable, similar in scope to previously proposed implementations [1, 5, 6, 13, 15, 17] and more efficient.

Narrowing is a widely used procedure for performing functional computations in a logic programming environment [10]. The appeal of narrowing for this task stems from its ability to evaluate functional expressions containing uninstantiated logic variables. Functions are defined by rewriting [7, 14]. A narrowing step rewrites a functional expression after possibly instantiating some of its variables. A narrowing strategy selects in an expression both the instantiation of its variables and its redex, and is aimed at ensuring desirable properties such as the efficiency and completeness of computations.

Narrowing is employed in functional-logic programming mainly to solve equations. Consider the following rewrite system, where upper case identifiers denote the variables.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
0 + X & \rightarrow & X & & & \\
s(X) + Y & \rightarrow & s(X + Y) & \mathit{double}(X) & \rightarrow & X + X \\
0 \leqslant \_ & \rightarrow & \mathit{true} & \mathit{half}(0) & \rightarrow & 0 \\
s(\_) \leqslant 0 & \rightarrow & \mathit{false} & \mathit{half}(s(0)) & \rightarrow & 0 \\
s(X) \leqslant s(Y) & \rightarrow & X \leqslant Y & \mathit{half}(s(s(X))) & \rightarrow & s(\mathit{half}(X))
\end{array}$$

For example, narrowing computes solutions to the equation $X \leqslant Y + Z \approx \mathit{true}$ as follows. There are three (most general) substitutions that create a redex in the equation: $\{X \mapsto 0\}$, $\{Y \mapsto 0\}$, and $\{Y \mapsto s(Y_1)\}$. The first substitution yields $0 \leqslant Y + Z \approx \mathit{true}$ which is reduced to $\mathit{true} \approx \mathit{true}$ by the first rule of $\leqslant$ and eventually to *true*. Thus, $\{X \mapsto 0\}$ is a solution of the equation. The other substitutions allow us to rewrite the right subterm of $\leqslant$. For example, the third substitution yields $X \leqslant s(Y_1) + Z \approx \mathit{true}$ which is reduced to $X \leqslant s(Y_1 + Z) \approx \mathit{true}$ by the second rule of $+$. Further narrowing steps applied to this equation compute solutions that composed with $\{Y \mapsto s(Y_1)\}$ compute solutions of the initial equation.

In Section 2 we make precise the scope of our discussion. In Section 3 we recall the needed narrowing strategy. In Section 4 we describe the generation of Prolog code that solves equations by needed narrowing. In Section 5 we describe several transformations of the Prolog code intended to improve the efficiency of computations. In Section 6 we measure the efficiency of our technique. In Section 7 we compare our technique with related work. In Section 8 we discuss future directions of our work.

2 Scope

We consider *many-sorted*, *constructor based*, *inductively sequential* rewrite systems. The first two concepts are standard notions of rewriting defined, e.g., in [7, 14]. The third concept is defined in [2] and recalled in the next section.

We consider a rewrite system with rules $\mathcal{R}$ and signature Σ. $\mathcal{R}$ may be non-terminating, hence some terms over Σ may not have a normal form. Under these conditions it is appropriate to define the validity of an equation as follows. The *equality rules* of Σ are defined below, where $\wedge$ is a new right associative, infix symbol and there is one rule for each constructor c in Σ.

$$\begin{array}{rcll} \mathit{true} \wedge X & \rightarrow & X & \\ c \approx c & \rightarrow & \mathit{true} & \forall c/0 \in \Sigma \\ c(X_1, \ldots, X_n) \approx c(Y_1, \ldots, Y_n) & \rightarrow & (X_1 \approx Y_1) \wedge \cdots \wedge (X_n \approx Y_n) & \forall c/n \in \Sigma \end{array}$$

The equality rules define an equivalence on terms referred to as *strict equality* and commonly used when computations may not terminate [8, 16]. A solution of an equation $t \approx u$ is a substitution σ such that $\sigma(t \approx u)$ rewrites to *true* using both $\mathcal{R}$ and the equality rules of Σ. A literal application of the equality rules may lead to an inefficient enumeration of all the narrowing derivations of a term. We correct this problem, in practice, by extending the above rules with

$$X \approx X \quad \rightarrow \quad \mathit{true}$$

and imposing that only constructor substitutions are acceptable for X.

We limit our discussion to unconditional rewrite systems. However, extending our discussion to conditional rewrite systems, e.g., as done in [15], does not create significant problems. Indeed, our implementation is applied conditional systems in [4].

3 Needed Narrowing

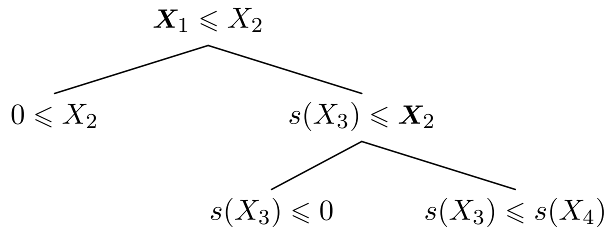
Needed Narrowing [3] is an optimal strategy in the sense that only unavoidable steps are performed to find the solutions of equations. It is defined for *inductively sequential* rewrite systems. Each operation in these systems is associated with a structure called *definitional tree*. Below, we recall this concept, central to our approach, in a more declarative form than in [2]. An algorithm to build definitional trees is presented in Appendix A.

A *pattern* is a term of the form $f(t_1, \ldots, t_n)$, where f is a defined operation and, for all i, t_i is a constructor term. A *definitional tree* of an operation f is a non-empty set $\mathcal{T}$ of patterns partially ordered by subsumption and having the following properties up to renaming of variables.

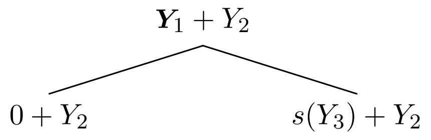
- [root property] The minimum element, referred to as the *root*, of $\mathcal{T}$ is $f(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are distinct variables.
- [leaves property] The maximal elements, referred to as the *leaves*, of $\mathcal{T}$ are all and only (variants of) the left hand sides of the rules defining f. Non-maximal elements are referred to as *branches*.
- [parent property] If π is a pattern of $\mathcal{T}$ different from the root, there exists in $\mathcal{T}$ a unique pattern π' strictly preceding π such that there exists no other pattern strictly between π and π'. π' is referred to as the *parent* of π and π as a *child* of π'.
- [induction property] All the children of a same parent differ from each other only at the position of a variable, referred to as *inductive*, of their parent.

There exist operations with no definitional tree, and operations with more than one definitional tree, examples are in [2]. The existence of a definitional tree of a function f is decidable and simple in most practical situations. An operation is called *inductively sequential* if it has a definitional tree. A rewrite system is called *inductively sequential* if all its operations are inductively sequential. The inductive sequentiality of a rewrite system $\mathcal{R}$ is obtained at no cost by establishing a priority on the rules of $\mathcal{R}$ based on either textual ordering or specificity, as done in most functional languages with pattern matching.

The adjective "inductive" is motivated by the fact that, for completely defined operations, the children of a pattern are obtained by "doing a data type induction" on the inductive variable of their parent. This variable is typeset in boldface in the examples. The trees of the operations $\leqslant$ and $+$ are pictorially shown below.



$\boldsymbol{X}_1 \leqslant X_2$
$0 \leqslant X_2$
$s(X_3) \leqslant \boldsymbol{X}_2$
$s(X_3) \leqslant 0$
$s(X_3) \leqslant s(X_4)$



Definitional trees make it particularly easy to operationally define needed narrowing. Let $t = f(t_1, \ldots, t_k)$ be an operation-rooted term to narrow. We most-generally unify t with some non-deterministically chosen maximal pattern π in a definitional tree $\mathcal{T}$ of f. Let σ be one such unifier. If π is a leaf of $\mathcal{T}$, $\sigma(t)$ is a redex and we reduce it. If π is a branch of $\mathcal{T}$, we consider the subterm u of $\sigma(t)$ matched by the inductive variable of π. u cannot be a variable. If u is operation-rooted, we attempt to narrow u to a head normal form. If u is constructor-rooted, we fail, since it can be shown [3] that $\sigma(t)$ cannot be narrowed at the top, which is necessary to solve any equation of which t is a side.

For example, consider the term $t = X \leqslant Y + Z$ discussed in our introductory example. Let $\mathcal{T}$ be the definitional tree of the operation $\leqslant$ shown earlier. The maximal elements of $\mathcal{T}$ that unify with t are $0 \leqslant X_2$ and $s(X_3) \leqslant X_2$. The first pattern is a leaf, thus, $t\{X \mapsto 0\}$ can be reduced at the root. The second pattern is a branch and X_2 is its inductive variable, thus, we attempt to narrow $Y + Z$, the subterm of $s(X_3) \leqslant Y + Z$ matching X_2. As seen in this example, needed narrowing does not always chooses most general unifiers.

4 Code Generation

Our implementation maps each defined operation f of an inductively sequential rewrite system $\mathcal{R}$, into a predicate f_0 of a Prolog program. If the type of f is $t_1 \times t_2 \times \cdots \times t_n \to t$, then the type of f_0 is $t_1 \times t_2 \times \cdots \times t_n \times t$. The predicate f_0 invokes a finite set of auxiliary predicates $f_1, f_2, \ldots$ depending on f, too. If $f_i(u_1, \ldots, u_n, u)$ succeeds, $i \geqslant 0$, then u is a head normal form (minimal in a sense not discussed in this paper) of $f(u_1, \ldots, u_n)$.

We define $f_0, f_1, \ldots$ as the output of the procedure *CodeGen*, defined shortly, on input f. *CodeGen* takes two arguments: p and $\mathcal{N}$. p is a natural number indexing the predicate identifier f and $\mathcal{N}$ is a subtree of a definitional tree of f. *CodeGen* is recursive. The values of p and $\mathcal{N}$ in the initial call to *CodeGen* are 0 and $\mathcal{T}$, a definitional tree of f. The value of p in recursive calls to *CodeGen* is the next yet unused index in the generation of $f_0, f_1, \ldots$.

A *shallow constructor term* is a term $c(X_1, \ldots, X_m)$, where c is a constructor, $m \geqslant 0$ is the arity of

c and $X_1, \ldots, X_m$ are fresh variables. If $m = 0$, the term is simply c. We call *shallow constructor term enumeration* of sort s an enumeration $c_1, \ldots, c_j$ of all the shallow constructor terms of a sort s, up to variable renaming.

A *head normal form* is a term t such that no descendant of t will ever be a redex. In constructor based system "useful" head normal forms are constructor-rooted terms. In the following discussion, we consider uninstantiated variables head normal forms, too, since both our implementation and the Gödel compiler in which it is embedded instantiate variables to constructor-rooted terms only.

The computation of *CodeGen* depends on whether or not $\mathcal{N}$ is a leaf of $\mathcal{T}$. Several optimizations of the code produced by *CodeGen* will be discussed in the next section.

1. Case $\mathcal{N}$ is a branch of $\mathcal{T}$
Let $\pi = f(t_1, \ldots, t_n)$ be the pattern at the root of $\mathcal{N}$, let t_i be the inductive variable of the pattern, and let s be the sort of t_i. The clauses generated by *CodeGen* are called to evaluate to head normal form a functional expression $exp = f(u_1, \ldots, u_n)$ that is known to unify with π[1]. The first problem is to find whether π is maximal among the patterns of $\mathcal{T}$ that unify with *exp*. Let u denote the i-th argument of *exp*. i.e., the subterm of *exp* matching t_i. We consider three subcases:

- u is operation-rooted: then π is maximal.
- u is a variable: then all the children of $\mathcal{N}$ contain maximal patterns unifying with *exp*.
- u is constructor-rooted: then at most one child of $\mathcal{N}$ contains maximal patterns unifying with *exp* (exactly one child if f is completely defined).

Thus, the clauses generated by *CodeGen* must determine which of the above subcases u satisfies and in each case dispatch an appropriate clause to continue the computation of needed narrowing of *exp*.

The detection of the subcase satisfied by u is performed via unification by a set of clauses whose heads differ only at the i-th argument. In these clauses, shown below, $c_1, \ldots, c_j$ is a shallow constructor term enumeration of sort s, and R is a new variable.

$$\begin{array}{lr}
f_p(t_1, \ldots, X, \ldots, t_n, R) \text{ :- } var(X),\ !,\ \ldots & 0 \\
f_p(t_1, \ldots, c_1, \ldots, t_n, R) \text{ :- } !,\ \ldots & 1 \\
\ldots & \\
f_p(t_1, \ldots, c_j, \ldots, t_n, R) \text{ :- } !,\ \ldots & j \\
f_p(t_1, \ldots, X, \ldots, t_n, R) \text{ :- } \ldots & j+1
\end{array}$$

The cut makes these clauses mutually exclusive. Now, we describe the code of the yet undefined portions of the bodies.

Clause number 0 is completed as follows

$$f_p(t_1, \ldots, X, \ldots, t_n, R) \text{ :- } var(X),\ !,\ f_{p_0}(t_1, \ldots, X, \ldots, t_n, R).$$

where f_{p_0} is a new predicate defined by j clauses identical to clauses 1 to j of f_p, but without the cut. As shown shortly, this code executes a descent down some non-deterministically chosen path of $\mathcal{N}$ where some maximal patterns unifying with *exp* can be found.

[1]This fact is obviously true for the root pattern of $\mathcal{T}$ and can be shown to be invariant through recursive calls of *CodeGen*.

The k-th clause, for k in $1, \ldots, j$, is either

$$f_p(t_1, \ldots, c_k, \ldots, t_n, R) \coloneq !,\ f_{p_k}(t_1, \ldots, c_k, \ldots, t_n, R).$$

or

$$f_p(t_1, \ldots, c_k, \ldots, t_n, \_) \coloneq !,\ \mathit{fail}.$$

The first clause is generated when $\mathcal{N}$ has a child, say $\mathcal{N}_k$, whose pattern has a variant of c_k at the position of the inductive variable of π. In this case, f_{p_k} is a new predicate generated by the call $\mathit{CodeGen}(p_k, \mathcal{N}_k)$. This code executes a descent down the unique path of $\mathcal{N}$ where some maximal patterns unifying with *exp* can be found. Otherwise, *CodeGen* generates the second clause. In this case it can be shown that *exp* cannot be narrowed to a head normal form because f is not completely defined.

The last clause of f_p is executed only when the i-th argument of *exp*, unified with the variable X, is an operation-rooted term. Clause number $j+1$ is defined as follows

$$f_p(t_1, \ldots, X, \ldots, t_n, R) \coloneq \mathit{ftp}_s(X, H), f_{p_{j+1}}(t_1, \ldots, H, \ldots, t_n, R).$$

where ftp_s, defined later, evaluates the functional expression instantiating X to head normal form and binds this value to Y, and $f_{p_{j+1}}$ is a new predicate defined by $j+1$ clauses identical to clauses 0 to j of f_p.

In our examples, we use more expressive names than indexing the translated function's identifier. Below, we show the predicate `add` generated from the operation + defined in the introduction. The clauses defining `add_10` are shown later. The clauses defining `add_1s` are similarly computed. The clauses defining `add_1v` and `add_1h` are easy to compute from the definition of `add`.

```
add(A,B,C) :- var(A), !, add_1v(A,B,C).
add(0,B,C) :- !, add_10(0,B,C).
add(s(A),B,C)) :- !, add_1s(s(A),B,C).
add(A,B,C) :- ftp_nat(A,H), add_1h(H,B,C).
```

2. Case $\mathcal{N}$ is a leaf of $\mathcal{T}$

Let $\pi = f(t_1, \ldots, t_n)$ be the pattern of $\mathcal{N}$ and $l \to r$ the rule of $\mathcal{R}$ such that l and π are variants. Let σ be a renaming substitution such that $f(t_1, \ldots, t_n) \to r' = \sigma(l \to r)$. We consider three exhaustive and mutually exclusive subcases, i.e., whether r' is a variable, or a constructor-rooted term, or an operation-rooted term.

2.1 Case r' is a variable, say X. There exists a unique occurrence of X in $t_1, \ldots, t_n$. Let s be the sort of X and let $c_1, \ldots, c_j$ be a shallow constructor term enumeration of sort s. *CodeGen* generates the following $j+2$ clauses

$$\begin{array}{lr}
f_p(t_1, \ldots, t_n, X) \coloneq \mathit{var}(X), !. & 0 \\
f_p(t_{11}, \ldots, t_{1n}, c_1) \coloneq !. & 1 \\
\cdots & \\
f_p(t_{j1}, \ldots, t_{jn}, c_j) \coloneq !. & j \\
f_p(t_1, \ldots, t_n, R) \coloneq \mathit{ftp}_s(X, H), f_q(t'_1, \ldots, t'_n, R). & j+1
\end{array}$$

where t_{hk}, $1 \leqslant h \leqslant j$, $1 \leqslant k \leqslant n$, is identical to t_k except for the instantiation of X to c_h, t'_k is identical to t_k except for the replacement of X with H, and f_q is a new predicate defined by $j+1$ clauses equal to clauses 0 to j of f_p.

If one of these clauses succeeds, the last argument of f_p is a head normal form. This is obvious for clauses 0 to j. For clause $j+1$, this property is a consequence of the definitions of ftp_s, presented shortly, and of f_q.

For example, the clauses generated by the first rule of + are

```
add_10(0,B,B) :- var(B), !.
add_10(0,0,0) :- !.
add_10(0,s(B),s(B)) :- !.
add_10(0,B,C) :- ftp_nat(B,H), add_10_2h(0,H,C).
```

2.2 Case r' is a constructor-rooted term. *CodeGen* generates the following single clause.

$$f_p(t_1, \ldots, t_n, r').$$

If this clause succeeds, the last argument of f_p is obviously a head normal form.

2.3 Case r' is an operation-rooted term. Let $r' = g(u_1, \ldots, u_m)$. *CodeGen* generates the following single clause, where R is a fresh variable and g_0 is the predicate generated by *Codegen* on input g.

$$f_p(t_1, \ldots, t_n, R) :- g_0(u_1, \ldots, u_m, R).$$

If this clause succeeds, then it can be shown by induction on its proof tree that R is instantiated to a head normal form.

Definition of ftp

We have used in the previous clauses a family of predicates, ftp_s, where s is a sort, that we now define. For each each sort s, for each defined operation g whose range is s, we define the following clause of ftp_s.

$$ftp_s(g(X_1, \ldots, X_n), R) :- !,\ g_0(X_1, \ldots, X_n, R).$$

This clause is executed during the evaluation of *exp* when it is necessary to evaluate a subexpression of the form $g(t_1, \ldots, t_n)$. The execution of this clause invokes the predicate g_0 generated by *CodeGen* on input g. If this clause succeeds, R is instantiated to a head normal form of $g(t_1, \ldots, t_n)$.

For example, the clauses of ftp_{nat} for the operations + and *half* are shown below, where, as usual, we denote the binary addition with its standard infix operator, and the corresponding predicate generated by *CodeGen* with `add`.

```
ftp_nat(A+B,H) :- !, add(A,B,H).
ftp_nat(half(A),H) :- !, half(A,H).
```

Strict Equality

Equations are solved by narrowing them to *true*. The equality rules presented in Section 2 define a set of operations, $\wedge$ and $\approx_s$, where s ranges over the sorts of Σ, that are inductively sequential. *CodeGen* applied to these operations generates Prolog clauses that can be used to compute head normal forms. Indeed, the only head normal form they compute is *true*, thus, *CodeGen* gives us the Prolog code for solving equations with no additional machinery.

In practice, we use a "specialized" version of *CodeGen*, that improves the code in several ways. In particular

- The family of Prolog predicates $\approx_{si}$ is binary. These predicates succeed rather than returning *true* in a third argument.
- If both arguments of $\approx_{s0}$ are uninstantiated variables, then $\approx_{s0}$ unifies these variables and succeeds. Since our approach only substitutes variables with constructor terms[2], this computation is consistent with our discussion in Section 2.
- $\wedge_0, \wedge_1, \ldots$ are not generated. In their places we use the boolean conjunction already available in Prolog.
- We apply to $\approx_{si}$ the optimizations discussed in the next section and an additional optimization, discussed later, applicable to functions that have more than one definitional tree.

The code implementing the strict equality predicate for natural numbers is shown in Appendix B.

5 Optimizations

In this section we describe some transformations of the code generated by *CodeGen* that are intended to improve the efficiency of computations. Except where explicitly stated, we describe each optimization independently of the other. In many cases, several optimizations can be applied to the same predicate. Constructor elimination and first argument indexing are borrowed from [11]. Some optimizations are not straightforward and we have only the space to sketch them and show an example.

Rule call elimination

CodeGen on input of a leaf node outputs a predicate defined by only one clause in two of the three cases discussed earlier. A call to this clause is easily unfolded. For example, the second rule of + generates only

```
add_1s(s(A),B,s(A+B)).
```

The predicate `add_1s` is called from one clause of `add`, namely

```
add(s(A),B,C)) :- !, add_1s(s(A),B,C).
```

We unfold this call and avoid the definition of `add_1s` altogether. The result simply becomes

```
add(s(A),B,s(A+B)) :- !.
```

Constructor elimination

Our implementation may generate predicates all of whose invocations and defining clause heads have a same argument rooted by a same constructor, possibly a constant. In this case, the argument may be replaced by the sequence of the argument's arguments. For example, an auxiliary predicate, that we identify with `half_1s`, generated by the rules of the defined operation *half*, is defined by

```
half_1s(s(A),B) :- var(A), !, ...
half_1s(s(0),B) :- !, ...
```

[2]This property also holds for Gödel, whose compiler has been extend with our implementation.

```
half_1s(s(s(A)),B) :- !, ...
half_1s(s(A),B) :- ftp_nat(A,H), ...
```

The first argument of `half_1s` is always rooted by the constructor s. `half_1s` is optimized as follows, assuming that suitable changes are made in the bodies of these clauses and in the call to this predicate.

```
half_1s(A,B) :- var(A), !, ...
half_1s(0,B) :- !, ...
half_1s(s(A),B) :- !, ...
half_1s(A,B) :- ftp_nat(A,H), ...
```

Constant arguments can be entirely eliminated. For example, the predicate `add_10` discussed in the previous section becomes

```
add_10(B,B) :- var(B), !.
add_10(0,0) :- !.
add_10(s(B),s(B)) :- !.
add_10(B,C) :- ftp_nat(B,H), add_10_2h(H,C).
```

First argument indexing

Most Prolog compilers index the clauses of a predicate by hashing on the principal functor of its first argument. If a pattern π has an inductive variable, the predicate's arguments can be permuted so that the argument corresponding to the inductive variable of π is the first. For example, the clauses of $\leqslant$ when the first argument is rooted by s are invoked by

```
leq(s(A),B,C) :- !, leq_1s(s(A),B,C).
```

and are defined by

```
leq_1s(s(A),B,C) :- var(B), !, ...
leq_1s(s(A),0,C) :- !, ...
leq_1s(s(A),s(B),C) :- !, ...
leq_1s(s(A),B,C) :- ftp_nat(B,H), ...
```

Swapping the arguments of `leq_1s` takes better advantage of indexing, i.e.,

```
leq_1s(B,s(A),C) :- var(B), !, ...
leq_1s(0,s(A),C) :- !, ...
leq_1s(s(B),s(A),C) :- !, ...
leq_1s(B,s(A),C) :- ftp_nat(B,H), ...
```

This optimization can be performed also for the clauses generated by a rule whose right hand side is a variable, e.g., the first rule of $+$.

Ftp elimination

The definition of the *ftp* family of predicates can be eliminated by instantiating the inductive argument in the head of a predicate with all the operation-rooted patterns. For example, the last clause of `add` can be replaced by the following clauses.

```
add(X+Y,B,C) :- !, add(X,Y,H), add_1h(H,B,C).
add(half(A),B,C) :- !, half(A,H), add_1h(H,B,C).
...
```

Eager discrimination

The case analysis performed by the code output by *CodeGen* on the inductive variable of an argument may be anticipated to the calling function, when the called function appears on the right hand side of a rule. For example, *double* is implemented by a single clause

```
double(A,B) :- add(A,A,B).
```

The first argument of `add` corresponds to the inductive variable of the pattern from which `add` is generated. We anticipate to `double` the case analysis that would be performed by `add` on this argument. In conjunction with the rule call elimination optimization, the predicate `add` is no longer invoked by `double`. The resulting clauses are shown below.

```
double(X,Y) :- var(X), !, double_1v(X,Y).
  double_1v(0,0).                                  % add(0,0,Y)
  double_1v(s(X),s(X+s(X))).                       % add(s(X),s(X),Y)
double(0,0) :- !.                                  % add(0,0,Y)
double(s(X),s(X+s(X))) :- !.                       % add(s(X),s(X),Y)
double(A,B) :- !, ftp_nat(A,H), double_1h(H,B).
  double_1h(X,Y) :- var(X), !, double_1v(X,Y).
  double_1h(0,0) :- !.                             % add(0,0,Y)
  double_1h(s(X),s(X+s(X))) :- !.                  % add(s(X),s(X),Y)
```

Variable unflattening

Patterns with constructor-term arguments that are non-shallow yield predicates with clauses that are never invoked when a non-shallow argument is matched by a variable, since variables are instantiated only by shallow constructor terms through our implementation. Clauses that are never invoked can be eliminated and the remaining clause can be generated as if the subtree involved in the generation were flat. For example, the code generated for the operation *half* is

```
half(A,B) :- var(A), !, half_1v(A,B).
  half_1v(0,B) :- half_10(0,B).
  half_1v(s(A),B) :- half_1s(s(A),B).
...
```

The definition of `half_1s` includes clauses where the first argument's argument is not instantiated to a variable. These clauses are never executed. The predicate `half_1v` is defined as follows to avoid the resulting potential inefficiency.

```
half_1v(0,B) :- half_10(0,B).
half_1v(s(0),B) :- half_1s0(s(0),B).
half_1v(s(s(A)),B) :- half_1ss(s(s(A)),B).
```

6 Benchmark

We present a benchmark intended both to assess the benefits of the optimizations discussed in Section 5, and to compare our implementation with other similar implementations. Our measures are obtained using the popular SICStus 2.1 Prolog interpreter/compiler. All code is compiled. Since system time measures are subject to some imprecision, the values that we report in the following tables are the averages of 12 executions of the same goals.

We report both the relative execution time and the absolute amount of allocated memory for finding the first solution of the following equations originally proposed in [11, Sect. 7]. As in [11], natural numbers are represented by $0/s$-terms. The function *one* is defined in [11, Example 2].

$$\begin{array}{rr}
10000 \leqslant 10000 + 10000 \approx \mathit{true} & E_1 \\
1000 \leqslant X + X \approx \mathit{true} & E_2 \\
400 + X \leqslant (X + 200) + X \approx \mathit{true} & E_3 \\
2000 \leqslant 1000 + (X + X) \approx \mathit{true} & E_4 \\
\mathit{double}(\mathit{double}(\mathit{one}(10000))) \approx X & E_5
\end{array}$$

The following table summarizes our findings with respect to execution time. Lines 1 to 6 measure the effects of a each optimization. A tabular entry contains the time of the computation as percent of the time taken by the unoptimized code. Line 7 refers to the code with all the optimizations combined. Line 8 shows, in the same scale, the performance of the code proposed in [11]. The comparison with several other implementations of narrowing in Prolog can be inferred using the benchmarks in [11], where it is shown that Hanus's code is the fastest.

		E_1	E_2	E_3	E_4	E_5	Aver.
1	Rule call elim.	83	77	83	77	61	76
2	Constr. elim.	72	65	76	66	77	72
3	Indexing	72	80	78	79	100	82
4	Ftp elim.	93	94	93	94	84	92
5	Eager discr.	–	–	–	–	25	–
6	Var. Unflat.	–	–	–	–	–	–
7	All opt.	36	35	43	35	11	32
8	Hanus code	56	79	63	79	27	61

Eager discrimination, line 5, is effective only for E_5. Variable unflattening, line 6, is not effective for the above equations, but, e.g., reduces by 25% the computation time to solve $\mathit{half}(X) \approx k$, where k is an integer literal.

The following table summarizes our findings with respect to memory allocation. Each tabular entry shows, in megabytes, the amount of memory allocated during the computation of the first solution of each equation.

		E_1	E_2	E_3	E_4	E_5
0	Plain	0.680	0.054	0.052	0.130	4.000
1	Rule call elim.	0.440	0.034	0.035	0.086	0.800
2	Constr. elim.	0.280	0.022	0.027	0.058	0.800
3	Indexing	0.680	0.054	0.052	0.130	4.000
4	Ftp elim.	0.680	0.054	0.052	0.130	4.000
5	Eager discr.	0.680	0.054	0.052	0.130	1.600
6	Var. Unflat.	0.680	0.054	0.052	0.130	4.000
7	All opt.	0.280	0.022	0.027	0.058	0.800
8	Hanus code	0.280	0.022	0.027	0.058	0.800

7 Related Work

The last ten years have seen a wealth of results on the foundations and applications of narrowing. The main thrust of this paper is the implementation of narrowing in Prolog, which has received considerable attention within this area [1, 5, 6, 13, 15, 17]. Our benchmark indicates that our implementation is almost twice as fast as Hanus's implementation [11]. A similar benchmark in [11] indicates that Hanus's implementation is faster, sometimes substantially, of all other previously published similar implementations. Hanus's implementation is partly inspired by [15]. Both [11, 15] also propose implementations of narrowing for the larger class of the constructor based, *weakly orthogonal* rewrite systems and also include *sharing*, which avoids multiple evaluations of equal terms. Extending our implementation to this larger class of systems and including sharing is possible as well using, e.g., the same approaches as [11, 15].

There are some noticeable differences between our approach and [11]. Our implementation adopts a less strict notion of equality that reduces the size of the search space in some cases. Our implementation allows us to perform more optimizations. Our implementation takes better advantage of mode information. Similar to [11] we consider an uninstantiated variable to be a head normal form. However, we process variables and constructor-rooted terms with different predicates, since there are choice points that must be created for variables, but can be avoided for constructor-rooted terms. This discrimination is not performed in [11]. Our implementation makes better use of the built-in unification, where [11, 15] define a predicate, *hnf/2*, to evaluate an expression *exp* to a head normal form. If *exp* is already a head normal form, *hnf/2* returns it unchanged. *Hnf/2* detects by unification whether to evaluate *exp*. Our implementation defines a similar predicate, $ftp_s/2$. However, we perform the unification required to discover whether to evaluate *exp* before calling $ftp_s/2$. Thus, we save the cost of the call when no evaluation of *exp* is necessary.

8 Future work

Hanus [9] has shown that adding deterministic rewrite steps to certain non-eager narrowing strategies may reduce infinite search spaces to finite ones. Unfortunately, needed narrowing cannot be improved by this technique. The benefits of this idea seems to originate from postponing the non-deterministic instantiation of variables in favor of more deterministic computations. We have explored some extensions of Hanus's idea suitable to needed narrowing.

Terms containing a function with distinct definitional trees may have distinct needed narrowing steps. In this case, the choice of the step may affect the efficiency of a computation. Following Hanus's lead we choose, at run-time, the "more deterministic" step. An opportunity for this strategy is offered by the ubiquitous strict equality. When one and only one side of an equation is a variable, we postpone its instantiation until the other side has been evaluated to head normal form. This strategy makes the instantiation deterministic and yields a marginal efficiency improvement.

Commutative operations, even if they do not have distinct definitional trees, offer a similar opportunity. Consider, for example, the evaluation of $t + u$ to head normal form. If t is a variable, our and most other implementations non-deterministically instantiate it. In this situation, if u is a head normal form, Hanus has shown the benefits of applying simplifying rewrite rules that do not require the (immediate) instantiation of t. We investigate a slightly more general approach. We define a partial ordering relation on terms, referred to as *more deterministic than*, and, if u is more deterministic than t, we evaluate $u + t$ instead of $t + u$. For suitable definitions of "more deterministic than" this approach encompasses both delaying the non-deterministic instantiation of a variable in an equation and applying a simplifying rewrite rule to an addition. Note that commuting the arguments of a commutative operation, such as addition, may change the set of the solutions of an equation. We argue, however, that this situation should never occur in well-behaved programs.

Below we show the results of a preliminary benchmark of this idea. We generate Prolog clauses from the rules of + as usual, however, before executing these clauses, we swap the arguments of + if the second argument is more deterministic than the first argument. The "more deterministic than" ordering ranks constructor-rooted terms as the most deterministic ones, variables as the least deterministic ones, and operation-rooted terms in between these to groups. This is more general than using simplifying rules. The next table shows both time and memory allocation results.

	E_1	E_2	E_3	E_4	E_5
Time	220	5	3	8	155
Memory	100	145	115	117	100

The values of each tabular entry show the performance of simplifying code as a percentage of non-simplifying code. In three equations the speed up is remarkable. Furthermore, the search spaces of both E_2 and E_4 become finite. The simplifying code computes all the solutions of these equation in a fraction of the time needed to compute the first solution without simplification.

Acknowledgement

I am very grateful to M. Hanus, R. Loogen, J. J. Moreno Navarro, and M. Rodríguez Artalejo for their interest in my work and for generously exchanging, during the last several years, programs and ideas on the subject of this paper. I am indebted to M. Hanus for showing me, well before [11], that the technique that I proposed in [1] was not as efficient as I had thought—his observation motivated my work—and to B. Barry for implementing the benchmark harness and making many valuable comments on the form and content of this paper.

Appendix A

We describe an algorithm that builds a definitional tree of an operation f from f's rules's left hand sides. The merits of this algorithm are its simplicity and the fact that the only information it requires is to distinguish variables from non-variable symbols. This was a crucial consideration for coding the algorithm within an existing compiler with encapsulated information. The algorithm is potentially inefficient, since it relies on non-deterministic choices. The algorithm's non-determinism can be limited or entirely eliminated at the expense of further computation (and a more complicated description). We assume that the rewrite system is left linear and non-overlapping.

There are two major steps, referred to as "chaining" and "merging", in the computation of a definitional tree of f. The first step builds, for each left hand side l of a rule of f, a "chain" of l, which is a convenient representation of what could be, if some constraints were ignored, a path from the root of a tree of f down to l. More precisely, a *chain* is a sequence $[l_0, p_1, l_1, p_2, \ldots, p_k, l_k]$ such that $l_0 = f(X_1, \ldots, X_n)$, l_k is a variant of l, and for all $j = 1, \ldots, k$, p_k is the position of a shallow constructor term in l_k and l_{k-1} is obtained from l_k by replacing this term with a variable.

For example, consider the operation $\leqslant$ defined in the introduction. A chain of $s(X) \leqslant s(Y)$, the left hand side of the third rule of $\leqslant$, is $[X_1 \leqslant X_2, 1, s(X_3) \leqslant X_2, 2, s(X_3) \leqslant s(X_4)]$. Ignoring the positions, this chain is indeed a path from the root of a tree of $\leqslant$ down to a variant of $s(X) \leqslant s(Y)$. Note, however, that $[X_1 \leqslant X_2, 2, X_1 \leqslant s(X_4), 1, s(X_3) \leqslant s(X_4)]$ is also a chain, but there is no tree of $\leqslant$ in which this chain represents a path.

The second step of the algorithm verifies whether all the chains of f can be "merged" into a definitional tree. If the merging succeeds, a tree of f is generated in the process. Merging chains is a recursive computation that takes a set S of "shortened" chains of f such that the head elements of the chains are variants of each other[3]. A chain is *shortened* by removing an even number of elements from its front. If S is a singleton and its chain contains only one term, say l, then the merging succeeds, and the (sub)tree $\mathcal{T}_S$ generated by S consists of the leaf l. Otherwise, we check whether the first position (second element) is the same in all the chains of S.

If this condition is false, the merging step fails. If this condition is true (remember that all the chains in S begin with variant terms), a variant of the head of a chain in S is the root of the (sub)tree $\mathcal{T}_S$ generated by S. Then, each chain in S is shortened by removing its first two elements, the leading term and the following position. The shortened chains of S are partitioned into subsets such that the heads of the chains in each subset are variants. Each resulting subset is merged to give a child of $\mathcal{T}_S$.

For example, an initial set of chains of $\leqslant$ is $\{[X_1 \leqslant X_2, 1, 0 \leqslant X_2], [Y_1 \leqslant Y_2, 1, s(Y_3) \leqslant Y_2, 2, s(Y_3) \leqslant 0], [Z_1 \leqslant Z_2, 1, s(Z_3) \leqslant Z_2, 2, s(Z_3) \leqslant s(Z_4)]\}$. Since the first position of each chain is the same, i.e., 1, $X_1 \leqslant X_2$ is the root of the tree. The chains are stripped of their first two elements and partitioned into the two sets, $\{[0 \leqslant X_2]\}$ and $\{[s(Y_3) \leqslant Y_2, 2, s(Y_3) \leqslant 0], [s(Z_3) \leqslant Z_2, 2, s(Z_3) \leqslant s(Z_4)]\}$. The first set yields the leftmost leaf of the tree of $\leqslant$ shown in Section 3. The second set can be merged too and yields the entire right subtree of the root.

Appendix B

The following code implements the strict equality predicate on natural numbers that we described in the text. If both arguments of `strict_eq_nat` are variables, they are unified and the predicate succeeds. This policy, which deviates from the standard definition of strict equality, is consistent with the property that the rest of our implementation instantiates variables only to constructor terms. All the optimizations discussed in Section 5 are included in the following code.

[3]All the chains in the initial set of chains begin with variant terms, and this property is invariant for any set of this kind subsequently processed during the merging.

```
strict_eq_nat(X,Y) :- var(X), !, sen_r_2v(Y,X).
  sen_r_2v(Y,X) :- var(Y), !, X=Y.          % semi strict, really
  sen_r_2v(0,0) :- !.
  sen_r_2v(s(Y),s(X)) :- !, sen_r_2v(Y,X).
  sen_r_2v(Y,X) :- ftp_nat(Y,H), sen_r_1h_2v(H,X).
    sen_r_1h_2v(Y,X) :- var(Y), !, X=Y.     % semi strict, really
    sen_r_1h_2v(0,0) :- !.
    sen_r_1h_2v(s(Y),s(X)) :- sen_r_2v(Y,X).
strict_eq_nat(0,Y) :- !, sen_10(Y).
  sen_10(0) :- !.
  sen_10(s(_)) :- !, fail.
  sen_10(Y) :- ftp_nat(Y,0).
strict_eq_nat(s(X),Y) :- !, sen_r_2s(Y,X).
  sen_r_2s(Y,X) :- var(Y), !, Y=s(Z), strict_eq_nat(X,Z).
  sen_r_2s(0,_) :- !, fail.
  sen_r_2s(s(Y),X) :- !, strict_eq_nat(X,Y).
  sen_r_2s(Y,X) :- ftp_nat(Y,s(Z)), strict_eq_nat(X,Z).
strict_eq_nat(X,Y) :- ftp_nat(X,H), sen_1h(H,Y).
  sen_1h(X,Y) :- var(X), !, sen_r_2v(Y,X).
  sen_1h(0,Y) :- !, sen_10(Y).
  sen_1h(s(X),Y) :- sen_r_2s(Y,X).
```

References

[1] S. Antoy. Lazy evaluation in logic. In *Proc. of the 3th Intl. Symp. on Programming Language Implementation and Logic Programming*, pages 371–382. Springer LNCS 528, 1991.

[2] S. Antoy. Definitional trees. In *Proc. of the 4th Intl. Conf. on Algebraic and Logic programming*, pages 143–157. Springer LNCS 632, 1992.

[3] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.

[4] B. J. Barry. Needed narrowing as the computational strategy of evaluable functions in an extension of Gödel. Master's thesis, Portland State University, June 1996.

[5] P.H. Cheong. Compiling lazy narrowing into prolog. Technical Report 25, LIENS, 45, rue d'Ulm, 7505 Paris, France, 1990.

[6] P.H. Cheong and L. Fribourg. Implementation of narrowing: The Prolog-based approach. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic programming languages: constraints, functions, and objects*, pages 1–20. MIT Press, 1993.

[7] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North Holland, Amsterdam, 1990.

[8] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: a logic plus functional language. *The Journal of Computer and System Sciences*, 42:139–185, 1991.

[9] M. Hanus. Combining lazy narrowing and simplification. In *Proc. 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 370–384. Springer LNCS 844, 1994.

[10] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[11] M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.

[12] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, 1993.

[13] J.A. Jiménez-Martín, J. Mariño-Carballo, and J. J. Moreno-Navarro. Efficient compilation of lazy narrowing into prolog. In *LOPSTR'92*, 1992.

[14] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1–112. Oxford University Press, 1992. Previous version: Term rewriting systems, Technical Report CS-R9073, Stichting Mathematisch Centrum, Amsterdam, 1990.

[15] R. Loogen, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *PLILP'93*, pages 184–200, Tallinn, Estonia, August 1993. Springer LNCS 714.

[16] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[17] M. H. van Emden and K. Yukawa. Logic programming with equations. *The Journal of Logic Programming*, 4:265–288, 1987.