Proc. PLILP '93, LNCS 714, Springer Verlag 1993, pp. 184–200

A Demand Driven Computation Strategy for Lazy Narrowing*

Rita Loogen[1], Francisco López Fraguas[2], Mario Rodríguez Artalejo[2]

[1] RWTH Aachen, Lehrstuhl für Informatik II, 52056 Aachen, Germany
email: rita@informatik.rwth-aachen.de

[2] Universidad Complutense de Madrid, Departamento de Informática y Automática
Facultad de C.C. Matemáticas, Av. Complutense s/n, 28040 Madrid, Spain
email: {fraguas,mario}@dia.ucm.es

Abstract. Many recent proposals for the integration of functional and logic programming use conditional term rewriting systems (CTRS) as programs and narrowing as goal solving mechanism. This paper specifies a computation strategy for lazy conditional narrowing, based on the idea of transforming patterns into decision trees to control the computation. The specification is presented as a translation of CTRS into Prolog, which makes it executable and portable. Moreover, in comparison to related approaches, our method works for a wider class of CTRS.

1 Introduction

Many recent approaches to the integration of functional and logic programming take *conditional term rewriting systems* (CTRS) as programs and *narrowing* as goal solving mechanism. Narrowing is complete as an equation solving procedure, under suitable hypotheses, see e.g. [7]. In general, narrowing has a high degree of (don't know) nondeterminism, due to two different sources: the choice of the redex, and the choice of the rewriting rule. For this reason, unrestricted narrowing leads to a huge search space which makes it too expensive to implement.

A *narrowing strategy* is any well defined criterion which restricts narrowing by allowing only certain narrowing derivations and thus obtaining a smaller search space. Many strategies have been proposed and used for implementations of narrowing. An important property of a narrowing strategy is *completeness*, meaning that for every solution computed by unrestricted narrowing, the strategy is able to compute a more general solution. A survey of results about the completeness of different narrowing strategies can be found in [6].

To achieve an efficient implementation of narrowing it is not enough to choose a good narrowing strategy. For any fixed strategy and any given equational goal there is still a search space including in general many narrowing derivations.

* This research has been partially supported by the Spanish National Project TIC92–0793 "PDR", the Esprit BRA Working Group Nr. 6028 "CCL" and the grant In 20/6–1 from the German Research Community.

Any well defined procedure which searches for narrowing derivations and produces the corresponding solutions in some order will be called *control regime* in this paper. Two common control regimes are *depth-first* search and *breadth-first* search. It is well known that most control regimes (e.g., depth-first search) destroy completeness, even if the underlying strategy is complete. Under *computation strategy* for narrowing we understand any combination of some narrowing strategy and some control regime.

We are interested in narrowing strategies which, being complete without the hypothesis of termination of the TRS, are adequate for modelling the combination of lazy functional programming and logic programming. More specifically, we are interested in *lazy narrowing* [18]. Informally, lazy narrowing delays the selection of inner narrowing positions unless it is demanded by the patterns in the lhs of the rule which is going to be tried next. A related strategy is *outer narrowing* [22].

Several compiled implementations of lazy narrowing have appeared in the last years. Some of them, like e.g. [3], are based on Warren's abstract machine WAM [20], while others were designed as extensions of reduction machines [4, 12, 16, 21]. There exist also methods to translate lazy rewriting and lazy narrowing into Prolog [1, 2, 5, 11, 17] in such a way that Prolog's computation rule simulates the lazy strategy. Such approaches allow to execute lazy narrowing on top of any WAM-based Prolog implementation.

A common difficulty within all implementations of lazy narrowing is to find good control regimes, which avoid repeated evaluations of argument terms and 'minimize' the risk of nontermination. Our aim in this paper is to give a formal specification of a new *demand driven* control regime for lazy conditional narrowing, based on the idea of transforming patterns from the lhs of rules into decision trees to control the computation. Technically, we use a generalization of S. Antoy's [1, 2] *definitional trees* (originally designed for a smaller class of TRS). We present our specification as a translation of CTRS into Prolog, which makes it executable on top of any Prolog system.

The organization of this paper is as follows. In Sect. 2 we introduce a simple functional logic language based on CTRS and we give a specification of a lazy narrowing strategy in a Prolog-like notation. Section 3 explains the disadvantages of a naive control regime for lazy narrowing and presents a 'Prolog' specification of our new demand driven control regime. Section 4 contains a discussion of related work. Some conclusions are finally drawn in Sect. 5.

Due to lack of space, we have omitted some examples and optimizations. The interested reader may find them in [14], which will be sent on request.

2 The Lazy Narrowing Strategy

For our discussion of narrowing we are going to use a simple functional logic language (SFL for short) which is based on conditional rewrite rules and encompasses the expressive power of several more concrete languages, as e.g. K-LEAF [8] and BABEL [15].

2.1 SFL programs

We assume a first order signature $\langle DC, FS \rangle$ with the ranked alphabet $DC = \bigcup_{n \in \mathbb{N}} DC^n$ of *constructor symbols* and the disjoint ranked alphabet $FS = \bigcup_{n \in \mathbb{N}} FS^n$ of *function symbols*. In the following, letters $c, d, e \ldots$ are used for constructors and letters $f, g, h \ldots$ for function symbols. Given a countably infinite set of *variables* $X, Y, Z \ldots \in Var$, we build *terms* $s, t \ldots \in Term$ (using only variables and constructors) and *expressions* $e, l, r \ldots \in Exp$ (using variables, constructors and function symbols). We assume expressions to be well typed w.r.t. types declared for constructors and function symbols. For simplicity, types will not be mentioned explicitly.

Note that we reserve the word *term* for *constructor terms* (without occurrences of function symbols), which play for us the role of *normal forms*. Given any expression e, we write $|e|$ for the *shell* of e, obtained by replacing outermost subexpressions of the form $f(e_1, \ldots, e_m)$ by a special nullary constructor $\perp$ (not belonging to the signature).

For function symbols $f \in FS^n$ we consider *defining rules*, which must be *left linear* conditional equations of the following form:

$$f(t_1, \ldots, t_n) := e \Longleftarrow l_1{==}r_1, \ldots, l_m{==}r_m.$$

where $t_i \in Term$ (constructor term) $(1 \leq i \leq n)$, $e, l_i, r_i \in Exp$. Operationally, such equations will be used as *conditional rewrite rules*. The sign '==' in conditions stands for *strict equality*, meaning that a condition $l_i{==}r_i$ must be satisfied by narrowing l_i, r_i into unifiable *constructor terms*. An *SFL-program* is any finite set of defining rules obeying certain natural conditions which ensure *confluence*; see [9]. *Termination* is not required.
Goals for SFL-programs are systems of strict equations of the form

$$\Longleftarrow l_1{==}r_1, \ldots, l_m{==}r_m.$$

to be solved by narrowing. Note that evaluation of an expression e to yield a value can be triggered by a goal $\Longleftarrow e == R$, R being a new variable.

Example 1. Let $CS^0 = \{true, false, 0, [\,]\}$, $CS^1 = \{s\}$, $CS^2 = \{[.|.]\}$ and $FS^1 = \{from\}$, $FS^2 = \{leq, cut\}$.
A legal SFL-program is given by the following defining rules:

$$\begin{array}{lll} leq(0, Y) & := true. & (LEQ_1) \\ leq(s(X), 0) & := false. & (LEQ_2) \\ leq(s(X), s(Y)) & := leq(X, Y). & (LEQ_3) \end{array}$$

$$\begin{array}{llll} cut(N, [\,]) & := [\,]. & & (CUT_1) \\ cut(N, [X|Xs]) & := [X] & \Longleftarrow leq(N, X) == true. & (CUT_2) \\ cut(N, [X|Xs]) & := [X|cut(N, Xs)] & \Longleftarrow leq(N, X) == false. & (CUT_3) \end{array}$$

$$from(N) := [N|from(s(N))]. \quad (FROM)$$

A simple goal for this example is $\Longleftarrow cut(N, from(0)) == [0, s(0)]$, for which we may expect $\{N/s(0)\}$ as a computed answer. In the rest of the paper we refer to this program as "the running example".

2.2 Specification of Lazy Narrowing

In this subsection we specify a *lazy narrowing strategy* for SFL programs. Inspired by [5], we present our specification as a 'Prolog translation' PT that converts any given SFL program P into a set of clauses $PT(P)$. Our aim is that solutions in the SLD search space for $PT(P)$ specify solutions computed by lazy narrowing for P. For the moment, Prolog's control regime is abstracted away. Thus, clauses in $PT(P)$ are intended as 'don't know' nondeterministic alternatives (within each one of several mutually excluding cases expressed by means of cuts, as we shall see below).

Within $PT(P)$ we represent SFL variables and expressions as Prolog variables and terms. Goals and conditions of rules can also be represented as Prolog terms (using '==' and ',' as infix operators). The answer substitutions computed by narrowing are not made explicit by our specification; they are subsumed by Prolog's unification. $PT(P)$ consists of clauses for two main predicates: *hnf*, which narrows expressions into *head normal form*, and *solve*, which solves conditions (and thus, also goals) by lazy narrowing. These and some other auxiliary predicates are described in the sequel.

Computation of Head Normal Forms. The predicate $hnf(E, H)$ specifies that H is one of the possible results of narrowing the expression E into head normal form. The clauses for *hnf* are given in Fig. 1. Predicates $\#f$ correspond to the defined function symbols in P and are defined below.

Clauses for hnf**:**

$hnf(E, H)$:- $var(E), !, H = E.$

$hnf(c(E_1, \ldots, E_m), H)$:- $!, H = c(E_1, \ldots, E_m).$ % for each $c \in CS^m$ $(m \geq 0)$

$hnf(f(E_1, \ldots, E_n), H)$:- $!, \#f(E_1, \ldots, E_n, H).$ % for each $f \in FS^n$ $(n \geq 0)$

Fig. 1. Clauses for *hnf*

Rule Application. $\#f(E_1, \ldots, E_n, H)$ specifies the evaluation of the expression $f(E_1, \ldots, E_n)$ to the expression H (in head normal form) by lazy narrowing. We assume an $(n+1)$-ary predicate $\#f$ for each n-ary function f. The definition of the predicates $\#f$ uses the auxiliary predicates

- $unify(E, T)$ to unify the expression E and the linear term T, reducing E by lazy narrowing as much as demanded by the constructors occurring in T, and
- $solve(C)$ to solve condition C by lazy narrowing.

The formal specification is given in Fig. 2. Two straightforward optimizations can be considered to simplify the clauses for rule application. If the body expression e of a rule $f(t_1, \ldots, t_n) = e \Longleftarrow b$ is e.g. a term of the form $c(s_1, \ldots, s_m)$,

Clauses for $\#f$, corresponding to each function f:

don't know choice among

$\#f(E_1, \ldots, E_n, H) :- \ unify(E_1, t_1), \ldots, unify(E_n, t_n), solve(b), hnf(e, H).$

% for each defining rule $f(t_1, \ldots, t_n) = e \Longleftarrow b$ in P

Clauses for *unify*:

$unify(E, X) \quad :- \ var(X), !, X = E.$

$unify(E, c(T_1, \ldots, T_m)) :- \ !, hnf(E, c(E_1, \ldots, E_m)),$
$\quad unify(E_1, T_1), \ldots, unify(E_m, T_m).$

% for each $c \in CS^m$ $(m \geq 0)$

Clauses for *solve*:

$solve((L == R, C)) :- \ !, eq(L, R), solve(C).$

$solve(L == R) \quad :- \ eq(L, R).$

Fig. 2. Clauses for Rule Application

where c is some constructor, the clause corresponding to this rule can be taken to be the following:

$$\#f(E_1, \ldots, E_n, e) :- unify(E_1, t_1), \ldots, unify(E_n, t_n), solve(b).$$

If e is a term of the form $g(s_1, \ldots, s_m)$, where g is some function symbol, the clause corresponding to the rule can be taken to be

$$\begin{aligned}\#f(E_1, \ldots, E_n, H) :- \ & unify(E_1, t_1), \ldots, unify(E_n, t_n), \\ & solve(b), \#g(s_1, \ldots, s_m, H).\end{aligned}$$

Note that $hnf(E, c(E_1, \ldots, E_m))$ will also succeed if a variable is computed as head normal form of E.

Goal Solving with Incremental Occur Check. The condition solver *solve* depends on the predicate $eq(L, R)$, which solves the strict equation between expressions L and R by lazy narrowing (see Fig. 3). It is defined using the auxiliary predicates

- $eq\_hnf(HL, HR)$ to solve the strict equation between expressions HL and HR (in head normal form) by lazy narrowing
- $bind(X, H)$ to reduce expression H (initially in head normal form) to normal form by lazy narrowing, and bind variable X to the resulting term, being careful with the *occur check*, and
- $occurs\_not(X, E)$ to check that variable X does not occur in the shell of expression E.

Note that the occur check is interleaved with the evaluation of an expression to NF. After the evaluation of each subexpression to HNF, the occur check tests whether the variable, which is unified with the whole expression, occurs in the *shell* of the HNF (to check w.r.t. the whole HNF would be incorrect). If the occur check is successful, the attempt to solve the equation fails and there is no need to evaluate the remaining non-evaluated subexpressions.

Clause for *eq*:

$eq(L, R)$:− $hnf(L, HL), hnf(R, HR), eq\_hnf(HL, HR)$.

Clauses for *eq_hnf*:

$eq\_hnf(X, H)$:− $var(X), !, bind(X, H)$.

$eq\_hnf(H, X)$:− $var(X), !, bind(X, H)$.

$eq\_hnf(c(L_1, \ldots, L_m), c(R_1, \ldots, R_m))$

:− $!, eq(L_1, R_1), \ldots, eq(L_m, R_m)$.

% for each $c \in CS^m$ $(m \geq 0)$

Clauses for *bind*:

$bind(X, Y)$:− $var(Y), !, X = Y$.

$bind(X, c(E_1, \ldots, E_m))$

:− $!, occurs\_not(X, E_1), \ldots, occurs\_not(X, E_m)$,

$X = c(X_1, \ldots, X_m)$,

$hnf(E_1, H_1), bind(X_1, H_1), \ldots, hnf(E_m, H_m), bind(X_m, H_m)$.

% for each $c \in CS^m$ $(m \geq 0)$

Clauses for *occurs_not*:

$occurs\_not(X, Y)$:− $var(Y), !, X =\backslash= Y$.

% $X =\backslash= Y$ checks syntactic disequality.

$occurs\_not(X, c(E_1, \ldots, E_m))$:− $!, occurs\_not(X, E_1), \ldots, occurs\_not(X, E_m)$.

% for each $c \in CS^m$ $(m \geq 0)$.

$occurs\_not(X, E)$. % the shell of E is $\perp$.

Fig. 3. Specification of Equality

Example 2. The goal $eq(X, c(Y, c(X, g(X, Z))))$ will fail, because X occurs in the shell of the expression $c(Y, c(X, g(X, Z)))$, while solving $eq(X, c(Y, g(X, Z)))$ will require the evaluation of the function application $g(X, Z)$.

The translation $PT(P)$ consists of all the clauses described. We *define* lazy (conditional) narrowing as the narrowing strategy specified by $PT(P)$ by abstracting away Prolog's control regime, as explained above. Note that it differs from lazy narrowing as defined in [15]. The "lazy" strategy in [15] permits some computations with unneeded steps which are forbidden by the present specification.

On the other hand, $PT(P)$ can also be viewed as an executable Prolog program and taken as a compilation of P into Prolog. From this point of view, there are two main limitations: sharing is not supported, and the control regime is inherited from Prolog. The disadvantages of this control will be discussed in the next section. For the moment, we can already introduce some optimizations at the level of the PT translation.

First, we observe that calls to *unify* and *solve* can easily be unfolded by partial evaluation. This makes the clauses for these predicates superfluous. The rule application clauses for our running example, after performing these optimizations, are shown in Fig. 4.

Another possible optimization is related to *sharing*. On the implementation level, lazy narrowing should avoid the repeated evaluation of multiple occur-

Clauses for $\#f$, corresponding to each function f

do not know choice among

$$
\begin{array}{ll}
\#leq(A,B,true) & :-\ hnf(A,0). \\
\#leq(A,B,false) & :-\ hnf(A,s(X)),hnf(B,0). \\
\#leq(A,B,H) & :-\ hnf(A,s(X)),hnf(B,s(Y)),\#leq(X,Y,H).
\end{array}
$$

$$
\begin{array}{ll}
\#cut(A,Bs,[\,]) & :-\ hnf(Bs,[\,]). \\
\#cut(A,Bs,[X]) & :-\ hnf(Bs,[X|Xs]),eq(leq(A,X),true). \\
\#cut(A,Bs,[X|cut(A,Xs)]) & :-\ hnf(Bs,[X|Xs]),eq(leq(A,X),false).
\end{array}
$$

$$\#from(A,[A|from(s(A))]).$$

Fig. 4. Rule Application Clauses for Example 1

rences of an expression which has been *passed as actual parameter*, i.e., introduced by the application of some rule whose rhs is not linear. Following a technique introduced by Cheong [5], we can easily modify the PT translation so that it will support sharing. The same is true for the Prolog translation to be developed in the next section. In the rest of the paper, sharing will not be considered.

Other optimizations we have worked out include a more efficient version of the *occurs_not* predicate and an improved formulation of the clauses for predicates $\#f$, which has the effect of a *dynamic cut* for discarding alternative $\#f$ clauses in case of *deterministic functional computations*. The idea of dynamic cut for functional logic languages was introduced in [13]. The interested reader is referred to [14] for more details.

3 Control Regimes for Lazy Narrowing

Given some SFL-program P, the execution of $PT(P)$ as a Prolog program amounts to an execution of P by lazy narrowing *under a particular control regime*, which is actually very close to Prolog's one. In the sequel, we shall refer to this as the *naive control regime*.

3.1 Inadequacies of the naive control regime

The naive control regime has two main disadvantages already pointed out by other researchers; cfr. [10, 11].

(D1) When trying the different defining rules for a function f in order to narrow an expression $f(e_1, \ldots, e_n)$, it may happen that the reduction of some argument expression e_i to HNF is repeated.

(D2) There is "too much risk" that the attempt of solving a goal leads to a diverging computation. This is illustrated by the following example.

Example 3. Consider the SFL program P consisting of the three rules for leq from our running example, plus the following rules for the functions nth_even_nb, nth_odd_nb and $iter$:

$$\begin{array}{llll} nth\_even\_nb(N) := iter(N,0). & & iter(0,X) & := X. \\ nth\_odd\_nb(N) := iter(N,s(0)). & & iter(s(N),X) & := iter(N,s(s(X))). \end{array}$$

With respect to P, the goal "$\Longleftarrow leq(nth\_odd\_nb(N), s(s(s(0)))) == true$" has exactly the two solutions $N/0$ and $N/s(0)$. Solving this goal by naive lazy narrowing corresponds to the goal "$:-eq(leq(nth\_odd\_nb(N), s(s(s(0)))), true)$" using $PT(P)$ as a Prolog program. It is easily seen that Prolog's computation strategy leads to a nonterminating computation. This corresponds to the fact that the naive control tries first to reduce $nth\_odd\_nb(N)$ to the HNF 0 in order to apply rule LEQ_1 before trying the next rules. This behaviour is misfortunate, since the subexpression $nth\_odd\_nb(N)$ can be narrowed to infinitely many different terms in HNF, none of which is 0.

While disadvantage (D1) above is clearly important, the relevance of (D2) has to be relativized. In fact, most control regimes are based on incomplete search strategies (for the sake of a smaller space complexity) and thus run some risk of nontermination. On the other hand, a nonterminating behaviour can sometimes be avoided by some modification of the SFL program (e.g. changing the textual order of defining rules, or the defining rules themselves). For instance, replacing the second clause for $iter$ by "$iter(s(N),X) := s(s(iter(N,X)))$" would avoid the nontermination problem shown before.

As an improvement over the naive control regime, we are going to specify a so called *demand driven* control regime. The idea is roughly as follows: instead of trying the defining rules in textual order and restart the evaluation of argument expressions for each rule, we shall look for suitable argument expressions that can be evaluated first and then used for all rules.

The new control regime will be specified as a new Prolog translation $DPT(P)$ which will generate clauses for the predicates $\#f$ in a different way. Let us anticipate an example.

Example 4. The new DPT translation will produce the following clauses for the predicate $\#leq$ (compare with the clauses shown in Sect. 2):

$$\#leq(A,B,H) :- hnf(A,HA), \#leq_{\{1\}}(HA,B,H).$$

$$\begin{array}{l} \#leq_{\{1\}}(0,B,true). \\ \#leq_{\{1\}}(s(X),B,H) :- hnf(B,HB), \#leq_{\{1,2\}}(s(X),HB,H). \end{array}$$

$$\begin{array}{l} \#leq_{\{1,2\}}(s(X),0,false). \\ \#leq_{\{1,2\}}(s(X),s(Y),H) :- \#leq(X,Y,H). \end{array}$$

For this particular example, the execution of these clauses under Prolog's control avoids disadvantage (D1) and improves over (D2). Beware, however, that (D2) cannot be overcome completely. To see this, the reader may inspect the following goal (for the program from Example 3). Note that the demand driven control, as specified by the Prolog translation above, will fail to enumerate infinitely many solutions (though infinitely many others will be enumerated!):

$$\Longleftarrow leq(nth\_odd\_nb(N), nth\_even\_nb(M)) == true$$

3.2 Definitional Trees for SFL Programs

The new Prolog translation $DPT(P)$ anticipated in the last subsection is based on *definitional trees*, a tool we borrow from Antoy's work [1, 2]. Antoy has used definitional trees to define efficient normalization strategies for a class of unconditional TRS, properly included in the class of SFL programs. Here we present an extended notion of definitional tree which covers all SFL programs and serves as basis for the specification of the DPT translation, to come in the next subsection. Let us start by defining some auxiliary notions.

Definition 1. 1. A *call pattern* is any linear expression of the form $f(t_1, ..., t_n)$, where f is a function symbol and t_i are terms. A *generic call pattern* is any call pattern of the form $f(X_1, \ldots, X_n)$, where X_i are n different variables.

2. Let *cpt* be a call pattern and let l be the lhs of a defining rule. We say that l *matches cpt* iff l is an instance of *cpt* via some (necessarily linear) term-substitution. Moreover, l is a *variant* of *cpt* iff this term-substitution is a variable renaming.
3. Let $vpos(t)$ and $cpos(t)$ denote the set of *variable and constructor positions* in the term t, respectively. Let *cpt* be a call pattern which is matched by the lhs of at least one defining rule in a given SFL-program P. Let $lhs(cpt)$ be the set of all lhs of rules from P which match *cpt*. Let u belong to $vpos(cpt)$. We say:
   (a) u is *demanded by the lhs* l iff l has a constructor at position u.
   (b) u is *demanded* iff u is demanded by some l in $lhs(cpt)$.
   (c) u is *uniformly demanded* iff u is demanded by every l in $lhs(cpt)$.

Example 5. Consider our running example. Given the call pattern $leq(s(X), B)$, we see that position 1.1 is not demanded, while position 2 is uniformly demanded. If we consider the generic call pattern $leq(A, B)$, we find that position 1 is uniformly demanded, while position 2 is demanded, but not uniformly. Lastly, in the following call pattern no variable position is demanded: $cut(N, [X|Xs])$. Note that this is a variant of (actually identical to) the lhs of rules CUT_2 and CUT_3.

Now we are prepared to introduce definitional trees.

Definition 2 *(Definitional Trees).* Given an SFL-program with set of rules P and a function f, the *definitional tree* $dt(f, P)$ *of* f *w.r.t. to* P is built according to the following algorithm. We assume that P is given as an ordered set and speak of the textual order of rules. This information is used while building the tree.

1. We define $dt(f, P) := dt(f(X_1, \dots, X_n), P)$.
   This means that the dt of f is constructed as the dt of f's generic call pattern.
2. Definition of $dt(cpt, P)$, where cpt is a call pattern that is matched by the lhs of at least one rule in P, by recursion on the syntactic structure of cpt: Compute $VP := vpos(cpt)$ and distinguish the following cases:
   (a) Some position in VP is uniformly demanded.
   Let u be the leftmost such position (this choice is quite arbitrary)[3] and let X be the variable at position u in cpt. Let $c_1, \dots, c_m$ be the constructors occurring at position u in the lhs of rules that match cpt (we assume that these c_j are taken in textual order). For each c_j, build the new call pattern
   $$cpt_j = cpt[X/c_j(X_1, \dots, X_{r_j})]$$
   where X_k are fresh variables and r_j is the arity of c_j.
   Then the dt has the structure of a case distinction:
   $$\begin{array}{lll} dt(cpt, P) := (cpt & - & case\ X\ of \\ & & \quad c_1 : dt(cpt_1, P); \\ & & \quad c_2 : dt(cpt_2, P); \\ & & \quad \dots \\ & & \quad c_m : dt(cpt_m, P)) \end{array}$$
   (b) No position in VP is demanded.
   Then it must be the case that all the lhs which match cpt are variants of cpt. There may be one or more of them. Let $\langle cpt = e_i \Longleftarrow b_i \mid 1 \leq i \leq m \rangle$ be renamings of these $m \geq 1$ rules with lhs identical to cpt, taken in textual order. Then the dt has the structure of an alternative between applicable defining rules:
   $$\begin{array}{ll} dt(cpt, P) := (cpt \longrightarrow & \langle e_1 \Longleftarrow b_1 \text{ by } rl_1 \\ & \mid e_2 \Longleftarrow b_2 \text{ by } rl_2 \\ & \quad \dots \\ & \mid e_m \Longleftarrow b_m \text{ by } rl_m \rangle) \end{array}$$
   where $rl_1, \dots, rl_m$ are names of the rules.
   (c) Some position in VP is demanded, but no one is uniformly demanded.
   Let $u_1, \dots, u_k$ be those positions in VP which are demanded, taken in the same order as they occur as demanded positions in lhs's which match cpt.
   For each j, $1 \leq j \leq k$, let P_j be the subset of P consisting of those rules whose lhs's *matches* cpt and demands position u_j, and let $CASE_j$ be $dt(cpt, P_j)$.
   Moreover, let P_0 be the subset of P consisting of those rules whose lhs matches cpt but demands no position. If P_0 is not empty, let $REDUCE$

[3] Using type information, one should choose a position in which all constructors of the corresponding type occur.

be the definitional tree $dt(cpt, P_0)$. Then, the definitional tree for cpt has the structure

$$dt(cpt, P) = (cpt \ \text{—}\ try\ \ \langle ALT_1 \mid ALT_2 \mid \ldots \mid ALT_s\rangle)$$

where the alternatives ALT_r are precisely the $CASE_j$ and $REDUCE$ (if existing), taken in the order induced by the order of defining rules in the source program[4].

Definition 3 *(Uniform Definitional Trees).* A definitional tree is called *uniform* iff its construction does not pass through case (c) in Definition 1. In case that $dt(f, P)$ is uniform, we say that the definition for f given by the rules in P is *inductively uniform.* If this happens for every function f, we say that the program P is inductively uniform.

A comparison between our definitional trees and those of Antoy [1, 2] is included in Sect. 4 below.

Example 6. Our running example is an inductively uniform SFL-program. The definition of leq is one of Antoy's examples in [2]. The (uniform) dt of leq is as follows.

$$
\begin{array}{llll}
(leq(A,B) \text{ — } case\ A\ of & & & \\
 & 0: & (leq(0,B) & \longrightarrow \langle true \text{ by } LEQ_1\rangle; \\
 & s: & (leq(s(X),B) & \text{— } case\ B\ of \\
 & & & \quad 0: (leq(s(X),0) \\
 & & & \qquad \longrightarrow \langle false \text{ by } LEQ_2\rangle; \\
 & & & \quad s: (leq(s(X),s(Y)) \\
 & & & \qquad \longrightarrow \langle leq(X,Y) \text{ by } LEQ_3\rangle)))
\end{array}
$$

The dt of cut is also uniform, but it looks a bit differently. It includes a representation of the alternative between rules CUT_2 and CUT_3, whose lhs are variants (identical, in fact):

$$
\begin{array}{ll}
(cut(N,[X|Xs]) & \\
\quad \longrightarrow \langle [X] \Longleftarrow leq(N,X) == true & \text{by } CUT_2 \\
\qquad \mid [X|cut(N,Xs)] \Longleftarrow leq(N,X) == false & \text{by } CUT_3\rangle)
\end{array}
$$

As in the examples above, uniform definitional trees have always the property that every rule R for a function in an SFL-program occurs exactly once in a leaf $e \Longleftarrow b$ by R of the tree (though several rules may be attached at the same leaf). This property is lost in nonuniform definitional trees, as shown by the following example:

[4] The idea is that trying these alternatives covers all the possible lazy narrowing derivations starting from a term of the form cpt.

Example 7. Consider the "Prolog-like" SFL-program fragment given by the following rules

$$\begin{array}{lll} loves(john, mary) & := true. & (LV_1) \\ loves(mary, Y) & := true \Longleftarrow likes(Y, wine) == false. & (LV_2) \\ loves(X, mary) & := true \Longleftarrow loves(mary, X) == true. & (LV_3) \end{array}$$

Note that no variable position in the generic call pattern $loves(A, B)$ is demanded. The *dt* of *loves* is of the form

$$(loves(A, B) \;—\; try\langle ALT_1 \mid ALT_2\rangle)$$

where ALT_1 corresponds to rules LV_1, LV_2 and ALT_2 to rules LV_1, LV_3.
The structure of ALT_2 is as follows:

$$\begin{array}{l} (loves(A, B) \;—\; case\ B\ of \\ \qquad mary : (loves(A, mary) \;—\; try \\ \qquad\qquad \langle\ (loves(A, mary) \;—\; case\ A\ of \\ \qquad\qquad\qquad john : (loves(john, mary) \\ \qquad\qquad\qquad\qquad \longrightarrow\langle true \text{ by } LV_1\rangle)) \\ \qquad\qquad \mid (loves(A, mary) \longrightarrow \\ \qquad\qquad\qquad \langle true \Longleftarrow loves(mary, A) == true \text{ by } LV_3\rangle) \\ \qquad\qquad \rangle) \end{array}$$

The construction of ALT_1 is left to the reader. A graphical presentation of the whole definitional tree for *loves* is given in Fig. 5.

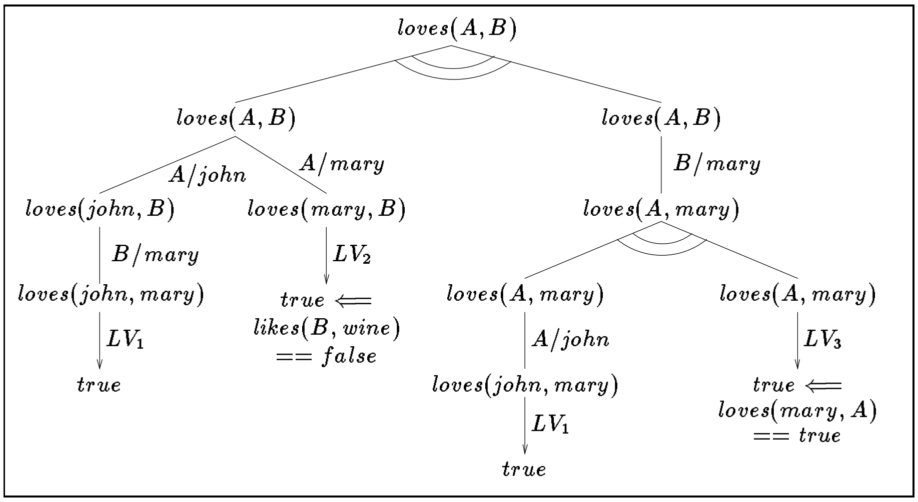

Fig. 5. Definitional Tree of *loves*

The inspection of other examples would show that nonuniform *dts* are often associated to functions with a 'parallel' flavour (e.g., the "parallel or" discussed in [2]). We can also find examples of nonuniform *dts* including uniform subtrees.

3.3 Prolog Specification of the Demand Driven Control

For a given SFL program P, we can now define the *demand driven* control regime as the search for solutions produced by the execution of the Prolog program $DPT(P)$. The Prolog translation $DPT(P)$, in turn, is defined as follows: For every n-ary function f we have a well defined definitional tree $dt(f, P)$. To every call pattern *cpt* occurring in this tree we associate an $(n+1)$-ary Prolog predicate $\#f_{CP}$, whose name depends uniquely on the set $CP = cpos(cpt)$ of constructor positions of *cpt*. By convention, we write $\#f_{\{\}}$ as $\#f$. Next, we give a recursive procedure which produces Prolog clauses from a given $dt(cpt, P)$, as follows:

(a) Assume $dt(cpt, P) = (cpt \;-\; case\ X\ of$

$$\begin{array}{l} c_1 : dt(cpt_1, P); \\ c_2 : dt(cpt_2, P); \\ \quad\ldots \\ c_m : dt(cpt_m, P)) \end{array}$$

where *cpt* is $f(t_1, \ldots, t_n)$. Take $CP = cpos(cpt), CPP = cpos(cpt_j)$ (which is the same for all $j, 1 \leq j \leq m$). Let HX be a fresh Prolog variable and build $(tt_1, \ldots, tt_n)$ as $(t_1, \ldots, t_n)[X/HX]$.
Then, the Prolog clauses for the tree $dt(cpt, P)$ consist of the clause

$$\#f_{CP}(t_1, \ldots, t_n, H) \ \text{:-}\ \ hnf(X, HX), \#f_{CPP}(tt_1, \ldots, tt_n, H).$$

followed by the clauses for the $dt(cpt_j, P)$.

(b) Assume now $dt(cpt, P) = (cpt \longrightarrow \langle e_1 \Longleftarrow b_1$ by rl_1

$$\begin{array}{ll} \mid e_2 \Longleftarrow b_2 & \text{by } rl_2 \\ \quad\ldots & \\ \mid e_m \Longleftarrow b_m & \text{by } rl_m\rangle) \end{array}$$

where *cpt* is $f(t_1, \ldots, t_n)$ and $cpos(cpt) = CP$. Then, the Prolog clauses for $dt(cpt, P)$ are

$$\begin{array}{c} \#f_{CP}(t_1, \ldots, t_n, H) \text{:-}\ solve(b_1), hnf(e_1, H). \\ \vdots \\ \#f_{CP}(t_1, \ldots, t_n, H) \text{:-}\ solve(b_m), hnf(e_m, H). \end{array}$$

in this order (which, by construction of the *dt*, corresponds to the textual order in the source SFL-program)[5].

(c) Assume $dt(cpt, P) = (cpt - try\ \langle ALT_1 \mid ALT_2 \mid \ldots \mid ALT_s\rangle)$. Then, the Prolog clauses for the tree $dt(cpt, P)$ are produced by merging the clauses obtained for $ALT_1, \ldots, ALT_s$ in such a way that clauses for the same predicate are taken together, but the order is respected. If some clause happens to have multiple occurrences, it suffices to keep one.

[5] The straightforward optimization we explained for $\#f$'s clauses in Subsect. 2.1 also applies here.

Finally, the translated program $DPT(P)$ consists of all the Prolog clauses associated to the definitional trees $dt(f,P)$ according to the procedure just defined, plus the Prolog clauses for the predicates hnf, eq, eq_hnf, $bind$ and $occurs\_not$, which are the same as in the PT-translation. The predicate *unify* becomes useless in the new translation scheme, and *solve* can be eliminated by partial evaluation, as before.

Considering again our running example, it is easily checked that the clauses for $\#leq$ we anticipated in Subsect. 3.1 are in fact produced by the DPT translation. The clauses for $\#cut$ and $\#from$ can be obtained similarly.
The next example illustrates the nonuniform case:

Example 8. The DPT translation generates the following clauses for the SFL-program from Example 7.

Note that we could optimize this Prolog code by discarding the clause marked with $(*)$. This is an instance of a general principle: dropping all but the first branches in a given dt which lead to equal leaves, optimizes the DPT translation[6].

$$
\begin{array}{l}
\#loves(A,B,H) :- hnf(A,HA), \#loves_{\{1\}}(HA,B,H).\\
\#loves(A,B,H) :- hnf(B,HB), \#loves_{\{2\}}(A,HB,H).\\
\#loves_{\{1\}}(john,B,H) \quad :- hnf(B,HB), \#loves_{\{1,2\}}(john,HB,H).\\
\#loves_{\{1\}}(mary,B,true) :- eq(likes(B,wine),false).\\
\#loves_{\{1,2\}}(john,mary,true).\\
\#loves_{\{2\}}(A,mary,H) \quad :- hnf(A,HA), \#loves_{\{1,2\}}(HA,mary,H). \quad (*)\\
\#loves_{\{2\}}(A,mary,true) :- eq(loves(mary,A),true).
\end{array}
$$

We close the section with some comments on two points of interest.

a) Managing control by definitional trees. Different permutations of the defining rules in a given SFL-program give rise to different definitional trees, which in turn determine different control regimes via the DPT-translation. For instance, rules LEQ_1, LEQ_2, LEQ_3 can be permuted in 6 different ways; only 4 different dt's arise from the permutations. Inspecting them can help to choose an order of defining rules which provides a better control. For instance, a definitional tree which delays recursive calls (or calls to other operations) to the rightmost branches is better for avoiding nonterminating narrowing computations. In the case of leq, the order LEQ_1, LEQ_2, LEQ_3 turns out to be optimal from this point of view. An implemented system could build the different dt's and use them to suggest optional reorderings of the rules.

b) Control versus Strategy. Since the notion of *laziness* applies to single narrowing computations, rather than to the search for alternative computations, it makes sense to ask whether two different control regimes are based on the same

[6] The reader may inspect the dt for *loves* to check what is going on.

narrowing strategy. We have the following claim, whose rigorous proof is left for future work:

Claim. DPT implements the same narrowing strategy as PT, under a different control. That is: the search spaces for $PT(P)$ and $DPT(P)$ include the same successful SLD refutations (up to the order of resolution steps), corresponding to the same successful narrowing computations (up to the order of narrowing steps).

Proof Idea. Note that we only claim a correspondence between the *successful* computations. The crucial point is that the $DPT(P)$ only expedites the evaluation of arguments which are uniformly demanded anyhow. $PT(P)$ starts the evaluation of arguments solely during the unification of an argument expression and the term pattern of a rule. But as uniformly demanded arguments are demanded by each defining rule of a function symbol, they will be evaluated in each successful narrowing derivation of the function application.

4 Related Work

Translations of functional logic languages into Prolog have also been presented in [5] and [11]. Cheong [5] has worked out a method for compiling programs written in the lazy functional plus logic language K-LEAF into Prolog. SFL and K-LEAF are very similar, and Prolog translations of K-LEAF programs according to Cheong's method are also similar to our translations $PT(P)$, with the following differences. Cheong justifies his translation method on the basis of a specific narrowing calculus for K-LEAF, while we take it as our specification of lazy narrowing. Cheong's treatment of strict equations is not completely correct (there are some problems related to the *occur check*). Cheong's translation supports sharing, but no dynamic detection of determinism.

[11] also discusses the disadvantages of lazy narrowing with the naive control scheme and presents a different demand driven strategy, which keeps the evaluation of arguments from left to right and the order of rules, which might be changed in our approach. The evaluation of demanded arguments is controlled by sophisticated demand patterns, which specify in more detail the amount of evaluation that is demanded by the term pattern of the function rules. Note that we distinguish only between no evaluation and evaluation to HNF. The demandedness notion of [11] is however not safe, as they view an argument as demanded whenever its evaluation is demanded by at least one defining rule.

Antoy [1, 2] introduces definitional trees and relates them to efficient normalization strategies for certain orthogonal and weakly orthogonal TRS. Inductively uniform *unconditional* SFL programs coincide with *inductively sequential* TRS, in Antoy's terminology[7]. For such programs, our definitional trees have the same

[7] We have preferred the term *inductively uniform* because it is not clear to us whether every inductively uniform SFL program admits a sequential reduction strategy. Moreover, this term suggests that inductive uniformity is a natural generalization of the uniformity notion from our previous paper [16].

structure as Antoy's. This is not always the case for other kinds of SFL programs (e.g. the program from Example 7).

Lower level (abstract machine based) implementations of lazy narrowing have been presented in [3, 4, 10, 16, 21]. The WAM-extension in [3] realizes the naive strategy. It has been the reference point of Cheong's work. In [16] the notion of uniform programs has been used to define a refined strategy. Our demand driven strategy generalizes and improves this approach.

In [10] a semantics based demandedness analysis has been used to determine demanded (strict) function arguments in order to evaluate such arguments before a function call. Moreover, a modified backtracking scheme to avoid the reevaluation of deterministic argument expressions is presented. The naive rule selection mechanism is kept in this approach.

[4] and [21] describe implementations of lazy narrowing which adapt the functional pattern matching compiler of [19] for rule selection. These approaches are very close to our approach. The case-expressions within the definitional trees are very similar to the case-expressions introduced by the pattern matching compiler. The most important difference is that the pattern matching compiler checks and evaluates arguments always from left to right.

5 Conclusions and Future Work

We presented a demand driven control regime for lazy narrowing computations based on a generalization of S. Antoy's approach of definitional trees [1, 2]. Our approach tends to avoid the disadvantages of lazy narrowing with the naive control scheme for most functional logic programs. The technique is especially appropriate for inductively uniform programs.

To simplify the presentation we considered only very elementary demandedness information. The incorporation of a more appropriate demandedness analysis in our scheme will be no problem and is planned in the future. Furthermore, we plan to develop a concrete implementation of the scheme in order to investigate its runtime behaviour in more detail.

Acknowledgement

While preparing the final version of the paper, we got some comments from Sergio Antoy and Michael Hanus. We are grateful for their interest in our work.

References

1. S. Antoy: *Lazy Evaluation in Logic*, Symp. on Programming Language Implementation and Logic Programming 1991, LNCS 528, Springer Verlag 1991, 371-382.
2. S. Antoy: *Definitional Trees*, Int. Conf. on Algebraic and Logic Programming (ALP) 92, LNCS 632, Springer Verlag 1992, 143–157.
3. P.G. Bosco, C. Cecchi, and C. Moiso: *An extension of WAM for K-LEAF*, 6th Int. Conf. on Logic Programming, Lisboa, 1989, 318-333.

4. M. M. T. Chakravarty, H. C. R. Lock: *The Implementation of Lazy Narrowing*, Symp. on Programming Language Implementation and Logic Programming 1991, LNCS 528, Springer Verlag 1991, 123–134.
5. P.H. Cheong: *Compiling lazy narrowing into Prolog*, Technical Report 25, LIENS, 1990, to appear in: Journal of New Generation Computing.
6. P.H. Cheong and L. Fribourg: *A survey of the implementations of narrowing*, in: J. Darlington and R. Dietrich (eds.) Declarative Programming. Workshops in Computing, Springer Verlag & BCS, 1992, 177-187.
7. M. Dershowitz, M. Okada: *A rationale for conditional equational rewriting*, Theoret. Comput. Sci. 75(1/2),1990,11–137.
8. E. Giovannetti, G. Levi, C. Moiso, C. Palamidessi: *Kernel LEAF: A Logic plus Functional Language*, Journal of Computer and System Sciences, Vol. 42, No. 2, Academic Press 1991, 139–185.
9. J.C. González Moreno, M.T. Hortalá González, M. Rodríguez Artalejo: *On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming*, to appear in: Computer Science Logic (CSL) 92, LNCS 702, Springer Verlag 1993 (15 pp.).
10. W. Hans, R. Loogen, St. Winkler: *On the Interaction of Lazy Evaluation and Backtracking*, Int. Symp. on Programming Language Implementation and Logic Programming (PLILP) 92, LNCS 631, Springer Verlag 1992.
11. J.A. Jiménez Martín, J. Mariño Carballo, J.J. Moreno Navarro: *Efficient Compilation of Lazy Narrowing into Prolog*, LOPSTR 92, LNCS, Springer Verlag 1992.
12. R. Loogen: *From reduction machines to narrowing machines*, TAPSOFT 91, CCPSD, LNCS 494, Springer Verlag 1991, 438-457.
13. R. Loogen, St. Winkler: *Dynamic Detection of Determinism in Functional Logic Languages*, Int. Symp. on Programming Language Implementation and Logic Programming (PLILP) 91, LNCS 528, Springer Verlag 1991, 335–346 (revised version to appear in TCS).
14. R. Loogen, F. J. López Fraguas, M. Rodríguez Artalejo: *A Demand Driven Computation Strategy for Lazy Narrowing*, Tech. Rep. Dep. Informática y Automática, UCM, Madrid, 1993.
15. J. J. Moreno Navarro, M. Rodríguez Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, Journal of Logic Programming Vol. 12, North Holland 1992, 191–223.
16. J.J. Moreno-Navarro, H. Kuchen, R. Loogen and M. Rodríguez-Artalejo: *Lazy narrowing in a graph machine*, Conf. on Algebraic and Logic Programming (ALP 90), LNCS 463, Springer Verlag 1990, 298–317.
17. S. Narain: *A technique for doing lazy evaluation in logic*, The Journal of Logic Programming, 3, 1986, 259-276.
18. U.S. Reddy: *Narrowing as the operational semantics of functional languages*, IEEE Symp. on Logic Programming, IEEE Comp. Soc. Press 1985, 138–151.
19. P. Wadler: *Efficient Compilation of Pattern-Matching*, Chapter 5 in: S. Peyton-Jones: *The Implementation of Functional Programming Languages*, Prentice Hall 1987.
20. D.H.D. Warren: *An abstract Prolog instruction set*, Technical Report 309, SRI International 1983.
21. D. Wolz: *Design of a compiler for lazy pattern driven narrowing*, 7th. Workshop on Specification of ADTs, LNCS 534, Springer Verlag 1991, 362-379.
22. J. H. You: *Enumerating Outer Narrowing Derivations for Constructor-Based Term Rewriting Systems*, Journal of Symbolic Computation 7, 1989, 319–341.