Implementation of Narrowing: The Prolog-Based Approach *

P.H. Cheong L. Fribourg

LIENS (URA 1327 CNRS)
45 rue d'Ulm, 75005 Paris — France
e-mail: {cheong,fribourg}@dmi.ens.fr

Abstract

We present the problem of integrating functional languages and logic languages. We explain why the narrowing-based techniques have so far prevailed as operational mechanisms for the functional logic interpreters. We then discuss various strategies of narrowing. Finally we explain how to simulate these strategies of narrowing using the leftmost SLD-resolution rule of Prolog, and compare some experimental results with those obtained with direct narrowing implementations.

1. Introduction

There has been a flurry of research on the integration of functional programming (FP) and logic programming (LP). A natural framework would be to consider the union of a set H of Horn clauses with a set E of conditional equations as a program. The declarative semantics of a program is then given by first-order logic with equality [26], that is, first-order logic extended with an equality symbol and the standard equality axioms. The operational semantics of a program is usually given by a system of inference rules that computes solutions to a given query. This system of inference rules should be sound and complete with respect to first-order logic with equality. Soundness ensures that all computed solutions are indeed correct, whereas completeness means that all solutions to a given query can be computed [25].

The simplest solution would be to retain the principle of ordinary resolution and to add equality axioms to the program. Unfortunately, this leads to a combinatorial explosion of the search space. This has motivated several proposals [31] [14] to extend

*This work has been partially supported by ESPRIT project BRA 3020.

syntactic unification involved in resolution to unification modulo the equational theory defined by E (called semantic unification). However, semantic unification applies only to a limited class of equational theories (for example, see [33]).

A second approach would view E as a set of oriented equations (rewrite rules), so that the arguments appearing in a call are either reduced in a call-by-value manner or in a lazy manner. FUNLOG [34] was an early experiment in this direction, later followed by LOG(F) [28]. The major drawbacks of these proposals are probably the incompleteness of the inference rules as well as the lack of a nice declarative semantics. An original solution to these problems is brought in [2] (see also chapter 2, part II, this volume). Here we will consider a third approach.

This approach consists in using a natural extension of term reduction called *narrowing* [20]. Narrowing consists in applying the minimal substitution to an expression in order to make it reducible, and then reduce it. The minimal substitution is found by unifying the expression with the lefthand side of a rewrite rule of E. For example, given the following rules :

$$plus(0, Y) = Y$$
$$plus(s(X), Y) = s(plus(X, Y))$$

the expression $plus(Z, W)$ can be narrowed to W, using the substitution $\{Z/0\}$, and to $s(plus(V, W))$, using the substitution $\{Z/s(V)\}$. Note that narrowing simulates resolution, provided that we view H as a set of conditional rewrite rules (see e.g. [21]). In other words, narrowing not only extends the basic computational mechanism of FP but also that of LP. It is then not surprising that narrowing-based techniques have so far prevailed as operational mechanisms for functional logic interpreters [1] [12] [15] [17] [23] [32]. Such a choice can often be nicely justified by rigorous completeness results. For instance, narrowing is a complete inference procedure in the case of a canonical system E [20].

In this paper, we shall explain how to efficiently implement various strategies of narrowing using Prolog evaluation. The paper is organized as follows. In section 2, we recall some theoretical results associated with narrowing and discuss various strategies. We propose in section 3 a language for functional logic programming. We then explain how to simulate the outermost strategy of narrowing (section 4) and the innermost strategy of narrowing (section 5) using Prolog resolution. Finally in section 6, we compare experimental results obtained with our Prolog-based approach with results obtained by direct narrowing implementation.

2. Narrowing Strategies

We have seen that the relative popularity of narrowing can be attributed to its ambivalence and its generality. Some critics may ask: is narrowing too powerful to be feasible in practice? To implement narrowing efficiently is certainly quite a difficult task, but a worth-while one, not only for the field of functional logic programming, but also for other closely related areas like algebraic programming, constraint logic programming, theorem proving, etc.

Narrowing has, in general, a high degree of nondeterminism. At each narrowing step, two choices have to be made, namely:

- the choice of the rewrite rule,
- the choice of the occurrence of the term to be reduced/narrowed (redex).

However, the number of choices at each step can be restricted by what we usually call a strategy refinement. There are numerous proposals of strategies in the literature, but we will only limit ourselves to a short survey of those that are suitable or widely-used in the domain of functional logic programming.

2.1. Basic narrowing

Basic narrowing goes back to Hullot [20]. The idea is to limit narrowing to subterms that are not introduced by instantiation. This amounts to preferring inner redexes to outer redexes (i.e. a kind of "inner-before-outer" strategy). It is of interest to note that basic narrowing is seldom used alone but often in conjunction with other refinements (one common refinement is leftmost-innermost basic narrowing [18]).

The completeness of basic narrowing has been studied by various authors, namely, [20] in the unconditional context, [19] and [16] in the conditional context; unconditional basic narrowing has also been studied in conjunction with rewriting [29]. It turns out that basic narrowing is complete for canonical term rewriting systems (TRS), and for level-canonical conditional term rewriting systems (CTRS) with extra-variables in the conditions. Recently, some counter-results have been discovered by [27]. For instance, basic narrowing is *not* complete for CTRS.

2.2. Normalized narrowing

Normalized narrowing can be traced back to [11]. The idea is to perform a normalization step before each narrowing step, which enforces the priority of determinate computations over nondeterminate ones.

It has often been remarked that normalized narrowing can turn an infinite narrowing tree into a finite one [9] [12]. However, completeness of normalized narrowing is hard to obtain, when it is used in conjunction with the basic strategy. A solution to this problem has been brought in [29], when the rewrite rules are unconditional.

The conditional case actually presents an additional difficulty, since reducibility is then undecidable. This means that, unless strong restrictions are imposed, there is no way to ensure the termination of the normalization step. Nontermination would delay the next narrowing step indefinitely, hence leading to incompleteness. A reasonable compromise in this case would consist in normalizing only via the *unconditional* rules (cf. SLOG [12]).

2.3. Selection narrowing

Selection narrowing has been introduced in two different contexts, namely, in the ordinary term context [30] [10], and also in the flattening context [5].

2.3.1. Redex-selection narrowing

In this variant, the idea is to select, at each step, only a redex for narrowing, while discarding other choices of redex. Of special interest are the *innermost* [12] and the *outermost* strategies [10] [36] which mimic the strict and lazy evaluation strategies known from FP.

Completeness of redex-selection narrowing has been studied by [30] and [10]. It turns out that innermost and outermost (redex-selection) narrowing are both incomplete. However, [10] has shown that in some cases completeness can be restored by simply transforming the given TRS into some homogeneous form. Lazy narrowing [32] and outer narrowing [36] can actually be understood as outermost redex-selection narrowing, modulo the transformation into homogeneous form.

2.3.2. Literal-selection narrowing

In this variant, programs and goals are considered as syntactic sugar for their flat form, and selection narrowing is simply taken to be SLD-resolution on the flat form. The flat form can be obtained by a process commonly known as flattening [22], which consists in replacing functional composition by logical conjunction (flattening will be presented in greater detail in following sections).

Again, two strategies mimic the strict and lazy evaluation strategies known from FP. The strict strategy can be simulated as the leftmost computation rule of Prolog [8]. The lazy strategy however requires a dynamic computation rule [24] [15].

Literal-selection narrowing can be considered as a refinement over basic narrowing regarding the search space issue [5]. Moreover, terms are shared and not duplicated, in the sense that when a term is narrowed, all its copies are also simultaneously narrowed to the same term. Although it is possible to introduce term sharing without recourse to flattening, the formal treatment involved would otherwise be quite tedious and clumsy.

3. Language proposal

In this paper we shall focus on the implementation of the outermost (lazy) and the innermost strategies of narrowing. As we have noted in the last section, these strategies can be implemented via SLD-resolution, by means of the flattening technique. Our aim is actually to implement these strategies by the leftmost computation rule of Prolog. There are two impediments to this aim.

First, lazy narrowing can be simulated if we use an outermost computation rule to select the literals [15]. The outermost computation rule is a dynamic computation rule, a priori not compatible with Prolog's leftmost rule. We shall however show how Prolog can adequately and sufficiently support the outermost rule, making unnecessary the use of specialized machines such as K-WAM [3].

Second, innermost narrowing is seldom used alone, but often in conjunction with normalization, to enforce the priority of determinate computations over nondeterminate ones. As rewriting is not entirely compatible with the flattening technique, new abstract machines like the A-WAM [18] has been specially designed. We shall discuss here a weaker notion of normalization, called simplification, that is more suited to be implemented in Prolog.

Following these ideas, we are able obtain portable implementations (in Prolog) for lazy or normalized innermost narrowing. Before devoting the remaining sections to the discussion of these simulations, let us first fix a syntax for our underlying language. The syntax we use here is only a subset of the languages SLOG [12], K-LEAF [15], BABEL [23] and ALF [18], but the extension to these languages is simple (see [6] [7]). In the following, we assume familiarity with logic programming [25].

3.1. First-order language

We consider a first order language that distinguishes between (defined) functions and (primitive) constructors.

A term formed uniquely from constructors and variables is called a *constructor* term. A term or a tuple of terms is said to be *linear* if it does not contain multiple occurrences of some variable. To simplify notations, we denote (possibly with subscripts and/or quotes) constructors by c; constructor terms by d or e; functions by f; predicates by p; terms by t or u; variables by S, T, U, V, W, X, Y, Z. Furthermore, an underlined symbol denotes a tuple. For example, $\underline{X}$ denotes a tuple of variables.

We use $Var(t)$ to denote the set of variables occurring in a term t, and $mgu(t,u)$ to denote the most general unifier of t and u.

3.2. Syntax

A *goal* is a formula $\leftarrow B_1,\ldots,B_n$ such that B_i $(i=1,\ldots,n)$ is either $p(\underline{t})$ or $f(\underline{t})=d$ with d a ground constructor term. A *clause* is a formula $A \leftarrow B_1,\ldots,B_m$ such that A is either $p(\underline{d})$ with $\underline{d}$ linear, or $f(\underline{d})=t$ with $\underline{d}$ linear and $Var(t)\subseteq Var(\underline{d})$, and such that $\leftarrow B_1,\ldots,B_n$ is a goal.

A *program* is a finite set of clauses such that any two clauses heads $f(\underline{d})=t$, $f(\underline{e})=u$ and any substitution σ verifying $\underline{d}\sigma=\underline{e}\sigma$ also verify $t\sigma=u\sigma$. This condition will be referred to as *weak-ambiguity.*

Note that, for the sake of presentation, we have omitted the strict equality of K-LEAF [15] or the weak equality of BABEL [23]. With respect to the syntax of SLOG [12] and ALF [18], we have imposed the linearity and the weak-ambiguity condition.

Example 3.1 *The following are some examples of clauses and goal:*

$$
\begin{array}{l}
plus(0,Y)=Y\leftarrow\\
plus(s(X),Y)=s(plus(X,Y))\leftarrow\\
int(X)=cons(X,int(s(X)))\leftarrow\\
first0(cons(0,Y))\leftarrow\\
\\
\leftarrow\ first0(int(plus(U,V)))
\end{array}
$$

The term $int(X)$ intuitively denotes the infinite list $[X,X+1,X+2,\ldots]$. The predicate $first0$ holds if its argument is a list headed by 0.

3.3. The flat form

The flat form of clauses and goals can be obtained by a process commonly known as flattening. Intuitively, this consists in replacing functional composition by logical conjunction. More formally, the flat form of goals and clauses is obtained by iteratively using the following flattening rules [5] [8] [22] [13] until none is applicable.

$$\frac{f(\underline{t})=u\leftarrow\ldots}{f(\underline{t})=u[\nu\leftarrow X]\leftarrow\ldots,u/\nu=X}\quad\text{if } u/\nu \text{ begins with a function}$$

$$\frac{\ldots,f(\underline{t})=u,\ldots}{\ldots,\underline{t}/\nu=X,f(\underline{t}[\nu\leftarrow X])=u,\ldots}\quad\text{if } \underline{t}/\nu \text{ begins with a function}$$

where X is a fresh variable, t/ν denotes the subterm of t at occurrence ν, and $t[\nu \leftarrow u]$ denotes the term obtained from t by replacing the subterm at occurrence ν by u.

Example 3.2 *The clauses and the goal in example 3.1 are flattened as*

$$
\begin{array}{l}
plus(0, Y) = Y \leftarrow \\
plus(s(X), Y) = s(Z) \leftarrow plus(X, Y) = Z \\
int(X) = cons(X, Y) \leftarrow int(s(X)) = Y \\
first0(cons(0, Y)) \leftarrow
\end{array}
$$

$$\leftarrow plus(U, V) = W, int(W) = W', first0(W')$$

Flattening preserves the declarative semantics of clauses and goals [5]. The flattening transformation allows to get rid of the transitivity and substitutivity axioms, but does not replace reflexivity and symmetry [4]. The symmetry axiom can be avoided thanks to the confluence property (assured by left-linearity and weak-ambiguity of the programs). However, the reflexivity axiom cannot be avoided in general. One possibility is to replace it by the more efficient *elimination* rule [15], or to discard it by providing sufficient conditions such as everywhere-definedness [12].

In what follows, we will essentially use the flat form in our exposition.

4. Implementation of lazy narrowing

In this section we explain how to implement lazy narrowing through SLD-resolution and an additional inference rule called *elimination* [15]. Intuitively, elimination corresponds to resolution using the reflexive axiom $X = X \leftarrow$. More formally, it can be defined as follows.

Elimination:

$$\frac{\leftarrow L_1, \ldots, L_{i-1}, f(\underline{d}) = X, L_{i+1}, \ldots, L_n}{\leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n} \quad \text{if } X \text{ does not occur in the } L_j\text{'s}$$

The SLD-resolution and the elimination rule constitute a *sound* and *complete* inference procedure for the language we consider here. For instance, the unique solution $U/0, V/0$ for the goal in 3.2 can be found by these rules.

Example 4.1

$$\leftarrow plus(U, V) = W, int(W) = W', \underline{first0(W')}$$

$$\Big|\ W'/cons(0, Y)$$

$$
\begin{array}{l}
\leftarrow plus(U,V) = W, \underline{int(W) = cons(0,Y)} \\
\qquad\qquad \Big| \; W/0 \\
\leftarrow \underline{plus(U,V) = 0}, int(s(0)) = Y \\
\qquad\qquad \Big| \; U/0, V/0 \\
\leftarrow \underline{int(s(0)) = Y} \\
\qquad\qquad \Big| \\
\qquad\qquad \leftarrow
\end{array}
$$

The underlined literal denotes the literal selected at each step or the refutation. The last step is elimination.

There is no need to backtrack on an elimination step [15]. The elimination rule transforms goals in simpler goals and should thus be given preference over the SLD-resolution rule. This means that an efficient computation rule should delay selecting $f(\underline{d}) = X$ for SLD-resolution unless it is sure that no elimination is applicable to it later on.

4.1. Simulation via Prolog

Instead of using variable annotations for expressing "outerness" as in [1], we shall use a different but equivalent formalism. We consider a paradigm in which all calls of the form $f(\underline{d}) = X$ are put to *sleep*, since they are good candidates for elimination. Sleeping literals are not chosen for resolution so long as their righthand sides are not instantiated. A sleeping literal will be awakened (or activated) when its righthand side is instantiated: it can be chosen for resolution (immediately or much later in the refutation sequence).

Example 4.2 *The clauses and goals in example 3.2 are written as:*

$$
\begin{array}{l}
plus(0,Y) = Y \leftarrow \\
plus(s(X),Y) = s(Z) \leftarrow [\![plus(X,Y) = Z]\!] \\
int(X) = cons(X,Y) \leftarrow [\![int(s(X)) = Y]\!] \\
first0(cons(0,Y)) \leftarrow \\
\\
\leftarrow [\![plus(U,V) = W]\!], [\![int(W) = W']\!], first0(W')
\end{array}
$$

where sleeping literals are enclosed by the $[\![$ *and* $]\!]$ *brackets.*

An *input variable* is a variable occurring in a literal $p(\underline{t})$ or in the lefthand side of a literal (equation) $s = t$. A literal L is said to be *functionally dependent* on another literal L', written $L \prec L'$ for short, if some variable of the righthand side of L is an input variable of L'. For example, we have $f(X) = Y \prec p(Y)$.

An active literal L in a goal G is said to be *outermost* iff L is not functionally dependent on any other active literals in G. An outermost computation rule is then a rule which selects outermost literals.

The outermost computation rule will go on selecting outermost literals until none is left. During this process, sleeping literals are activated in the resolvent when their righthand sides become instantiated to a nonvariable term. When no outermost literal is left, the remaining literals are then sleeping literals. These literals can be deleted *all at once*, due to the *multiple-elimination* rule below.

Multiple-Elimination:

$$\frac{\leftarrow L_1, \ldots, L_n}{\leftarrow} \quad \text{if the } L_i\text{'s are sleeping}$$

Example 4.3 *The annotated version of the refutation of example 4.1 is:*

$$\begin{array}{l}
\leftarrow [\![plus(U,V) = W]\!], [\![int(W) = W']\!], \underline{first0(W')} \\
\qquad\qquad \Big| \; W'/cons(0,Y) \\
\leftarrow [\![plus(U,V) = W]\!], \underline{int(W) = cons(0,Y)} \\
\qquad\qquad \Big| \; W/0 \\
\leftarrow \underline{plus(U,V) = 0}, [\![int(s(0)) = Y]\!] \\
\qquad\qquad \Big| \; U/0, V/0 \\
\leftarrow [\![int(s(0)) = Y]\!]
\end{array}$$

Finally, we obtain the empty goal by (multiple-)elimination.

4.1.1. Further processing of clauses

Sleeping equations are activated when their righthand sides become instantiated. This kind of dynamic activations can be simply handled by introducing a new function, declaratively defined by $eval(X) = X \leftarrow$, such that activations will only occur during an $eval$ call. (In particular, non $eval$ calls should not activate any sleeping equation.) To make use of $eval$, we further process the clauses by iteratively applying the following rules:

$$\frac{p(\ldots,d,\ldots) \leftarrow \cdots}{p(\ldots,Y,\ldots) \leftarrow eval(Y) = d,\ldots} \qquad \text{if } d \text{ is nonvariable}$$

$$\frac{f(\ldots,d,\ldots) = \cdots \leftarrow \cdots}{f(\ldots,Y,\ldots) = \cdots \leftarrow eval(Y) = d,\ldots} \qquad \text{if } d \text{ is nonvariable}$$

$$\frac{f(\underline{d}) = X \leftarrow \cdots}{f(\underline{d}) = Y \leftarrow \ldots, eval(X) = Y} \qquad \text{if } X \in Var(\underline{d})$$

$$\frac{\ldots, eval(X) = c(\ldots,d,\ldots),\ldots}{\ldots, eval(X) = c(\ldots,Y,\ldots), eval(Y) = d,\ldots} \qquad \text{if } d \text{ is nonvariable}$$

where $f \neq eval$ and Y is a fresh variable.

Note that these rules preserve the operational semantics of the clauses (given by SLD-resolution and elimination). The first three transformation rules are very much in the spirit of the notion of the homogeneous form [35]. The fourth rule is inspired from [28].

Example 4.4 *We process the clauses in example 4.2 into the required form:*

$$\begin{array}{l}
eval(X) = X \leftarrow \\
plus(X,Y) = Y' \leftarrow eval(X) = 0, [\![eval(Y) = Y']\!] \\
plus(X',Y) = s(Z) \leftarrow eval(X') = s(X), [\![plus(X,Y) = Z]\!] \\
int(X) = cons(X,Y) \leftarrow [\![int(s(X)) = Y]\!] \\
first0(X') \leftarrow eval(X') = cons(X,Y), eval(X) = 0
\end{array}$$

So far, we have put to sleep all calls of the form $f(\underline{d}) = X$. We now introduce an optimization which consists in not putting to sleep a $f(\underline{d}) = X$ call when it appears in a body of a clause like $g(\underline{e}) = X \leftarrow \ldots, f(\underline{d}) = X, \ldots$ For instance, the second clause of 4.4 should now read $plus(X,Y) = Y' \leftarrow eval(X) = 0, eval(Y) = Y'$. Such an optimization, together with the processing discussed in this section, allow us to have the following activation pattern.

Proposition 4.1 *Outermost resolution will activate a sleeping literal $[\![f(\underline{d}) = X]\!]$ if and only if the latter occurs in the parent goal (in particular, not in the input clause) and the selected literal has the form $eval(X) = e$, with e nonvariable.*

This proposition tells us that there is no activation of literals unless the selection literal has the form $eval(X) = e$, with X being the righthand side of some sleeping literal and e nonvariable. So the outermost resolution rule can be specialized as follows:

OR:

$$\frac{\leftarrow \ldots, L_{i-1}, L_i, L_{i+1}, \ldots}{\leftarrow (\ldots, L_{i-1}, B_1, \ldots, B_m, L_{i+1}, \ldots)\sigma} \quad \text{if} \begin{cases} L_i \text{ is not an } eval \text{ call} \\ A \leftarrow B_1, \ldots, B_m \text{ is a clause} \\ \sigma = mgu(A, L_i) \end{cases}$$

E1:

$$\frac{\leftarrow \ldots, L_{i-1}, eval(d) = e, L_{i+1}, \ldots}{\leftarrow (\ldots, L_{i-1}, L_{i+1}, \ldots)\sigma} \quad \text{if} \begin{cases} d \text{ is not the rhs of some sleeping } L_k \\ \sigma = mgu(d, e) \end{cases}$$

E2:

$$\frac{\leftarrow \ldots, L_{i-1}, eval(X) = e, L_{i+1}, \ldots, L_{j-1}, \llbracket f(\underline{d}) = X \rrbracket, L_{j+1}, \ldots}{\leftarrow (\ldots, L_{i-1}, L_{i+1}, \ldots, L_{j-1}, f(\underline{d}) = X, L_{j+1}, \ldots)\sigma} \quad \text{if} \begin{cases} e \text{ nonvariable} \\ \sigma = mgu(d, e) \end{cases}$$

where the selected literal (L_i in OR, $eval(d) = e$ in $E1$ and $eval(X) = e$ in $E2$) is always supposed to be outermost. Note that in the last rule, the literal $\llbracket f(\underline{d}) = X \rrbracket$ may appear in any position (i.e. either $j < i$ or $i > j$).

Example 4.5 *The refutation of example 4.3 now becomes:*

$$\leftarrow \llbracket plus(U, V) = W \rrbracket, \llbracket int(W) = W' \rrbracket, \underline{first0(W')}$$
$$|$$
$$\leftarrow \llbracket plus(U, V) = W \rrbracket, \llbracket int(W) = W' \rrbracket, \underline{eval(W') = cons(X, Y)}, eval(X) = 0$$
$$|$$
$$\leftarrow \llbracket plus(U, V) = W \rrbracket, \underline{int(W) = cons(X, Y)}, eval(X) = 0$$
$$|$$
$$\leftarrow \llbracket plus(U, V) = W \rrbracket, \llbracket int(s(W)) = Y \rrbracket, \underline{eval(W) = 0}$$
$$|$$
$$\leftarrow \underline{plus(U, V) = 0}, \llbracket int(s(0)) = Y \rrbracket$$
$$|$$
$$\leftarrow \underline{eval(U) = 0}, eval(V) = 0, \llbracket int(s(0)) = Y \rrbracket$$
$$|$$
$$\leftarrow \underline{eval(V) = 0}, \llbracket int(s(0)) = Y \rrbracket$$
$$|$$
$$\leftarrow \llbracket int(s(0)) = Y \rrbracket$$
$$|$$
$$\leftarrow$$

4.1.2. Dynamic ordering of goals

Thus far, the actual textual order of literals does not matter in our discussion. However, in order to pave the way for the compilation into Prolog, we must now introduce some notions of literal ordering. A goal (resp. a clause) of the form $\leftarrow B_1, \ldots, B_n$ (resp. $A \leftarrow B_1, \ldots, B_n$) is *ordered* iff for all literals B_i and B_j, $i < j$ implies B_i is not functionally dependent on B_j. The leftmost active literal in an ordered goal is always an outermost literal.

The *OR* rule preserves the order of goals in the sense that if the parent goal and the input clause are ordered, and if the leftmost active literal is chosen, then the resolvent is ordered. The *E1* rule preserves order in the same sense. However, the *E2* rule does not generally preserve order. But it can be guaranteed to preserve order provided that we always move the activated literal to the leftmost position in the resolvent.

According to the above discussion, the leftmost rule of Prolog can thus simulate the outermost rule provided that (1) statically goals and clauses are ordered, and (2) dynamically activated literals are moved to the leftmost positions. Point (2) can be implemented easily if sleeping literals can be directly accessed via some convenient term representation.

4.1.3. Extended term representation

Similarly to [3], we link a variable X directly to the sleeping literal that defines it, say $[\![f(\underline{d}) = X]\!]$, by writing it under the form $\#(S, f(\underline{d}) = X)$, where S is a control flag initially unset.

Example 4.6 *Clauses and goals are now represented as:*

$$
\begin{array}{l}
eval(X) = X \leftarrow \\
plus(X, Y) = Y' \leftarrow eval(X) = 0, eval(Y) = Y' \\
plus(X', Y) = s(\#(S, plus(X, Y) = Z)) \leftarrow eval(X') = s(X) \\
int(X) = cons(X, \#(S, int(s(X)) = Y)) \leftarrow \\
first0(X') \leftarrow eval(X') = cons(X, Y), eval(X) = 0
\end{array}
$$

$$\leftarrow first0(\#(S, int(\#(S', plus(U, V) = W)) = W'))$$

Note that the above clauses are ordered.

The control flags serve to interpret the #-terms. They are set to on when the associated equations are activated. In the new term representation, the rules *OR, E1* and *E2* simply become:

or:

$$\frac{\leftarrow L_1, \ldots, L_n}{\leftarrow (B_1, \ldots, B_m, L_2, \ldots, L_n)\sigma} \quad \text{if} \begin{cases} L_1 \text{ is not an } eval \text{ call} \\ A \leftarrow B_1, \ldots, B_m \text{ is a clause} \\ \sigma = mgu(A, L_1) \end{cases}$$

eval1:

$$\frac{\leftarrow eval(u) = e, L_2, \ldots, L_n}{\leftarrow (L_2, \ldots, L_n)\sigma} \quad \text{if} \begin{cases} u \text{ is a variable or its functor is a constructor} \\ \sigma = mgu(u, e) \end{cases}$$

eval2:

$$\frac{\leftarrow eval(\#(on, f(\underline{t}) = u)) = e, L_2, \ldots, L_n}{\leftarrow (L_2, \ldots, L_n)\sigma} \quad \text{if } \sigma = mgu(u, e)$$

eval3:

$$\frac{\leftarrow eval(\#(S, f(\underline{t}) = X)) = e, L_2 \ldots, L_n}{\leftarrow (f(\underline{t}) = X, L_2, \ldots, L_n)\sigma} \quad \text{if } \sigma = \{S/on, X/e\}$$

4.1.4. Translation into Prolog

We will now simulate the four previous inference rules in Prolog. The *or* rule is trivially simulated by Prolog. For instance, clauses (except the *eval* clause) and goals of the previous example can be simply treated as ordinary Prolog clauses and goals.

```
plus(X,Y)=Y1 :- eval(X)=0,eval(Y)=Y1.
plus(X1,Y)=s(#(_,plus(X,Y)=Z)) :- eval(X1)=s(X).
int(X)=cons(X,#(_,int(s(X))=Y)).
first0(X1) :- eval(X1)=cons(X,Y),eval(X)=0.

:- first0(#(_,int(#(_,plus(U,V)=_))=_)).
```

The rules for *eval* can be simulated by defining *eval* as a meta-predicate. Supposing that the constructors are `0`, `s` and `cons` and the functions are `plus` and `int`, we have the following definition for `eval`.

```
eval(X)=Y :- var(X),!,unify(X,Y).              % simulate eval1
eval(0)=Y :- unify(0,Y).                       % simulate eval1
eval(s(X))=Y :- unify(s(X),Y).                 % simulate eval1
eval(cons(X,Y))=Z :- unify(cons(X,Y),Z).       % simulate eval1

eval(#(S,plus(X,Y)=Z))=T :- S==on,unify(Z,T). % simulate eval2
eval(#(S,int(X)=Y))=Z :- S==on,unify(Y,Z).    % simulate eval2

eval(#(S,plus(X,Y)=Z))=T :-                    % simulate eval3
    var(S),unify(S,on),unify(Z,T),plus(X,Y)=Z.
eval(#(S,int(X)=Y))=Z)) :-                     % simulate eval3
    var(S),unify(S,on),unify(Y,Z),int(X)=Y.
```

where we have denoted syntactic unification by `unify`.

The cut in the first `eval` clause ensures that the input argument to the other `eval` clauses is always nonvariable. This in turn ensures that the other `eval` clauses correctly simulate the *eval* rules. The following optimizations are immediate:

- `unify` calls in the second, third and fourth clause can be unfolded.
- `var(S)` calls in the last two clauses are not necessary, provided we insert cuts (`!`) just after the `S==on` calls in the fifth and sixth clause (since `S==on` and `var(S)` are mutually exclusive).
- After deleting the `var(S)` calls, `unify` calls in the last two clauses can be unfolded.

Finally, we obtain:

```
eval(X)=Y :- var(X),!,unify(X,Y).
eval(0)=0.
eval(s(X))=s(X).
eval(cons(X,Y)=cons(X,Y)).
eval(#(S,plus(X,Y)=Z))=T :- S==on,!,unify(Z,T).
eval(#(S,int(X)=Y))=Z :- S==on,!,unify(Y,Z).
eval(#(on,plus(X,Y)=Z))=Z :- plus(X,Y)=Z.
eval(#(on,int(X)=Y))=Y :- int(X)=Y.
```

and the refutation:

```
:- first0(#(_,int(#(_,plus(U,V)=_))=_)).
     |
:- eval(#(_,int(#(_,plus(U,V)=_))=_))=cons(X,Y),eval(X)=0.
     |
:- int(#(_,plus(U,V)=_))=cons(X,Y),eval(X)=0.
     |
:- eval(#(_,plus(U,V)=_))=0.
     |
:- plus(U,V)=0.
     |
:- eval(U)=0,eval(V)=0.
     |
:- eval(V)=0.
     |
:-
```

5. Implementation of normalized innermost narrowing

In this section we shall present an interpreter called Prolog with Simplification that simulates normalized innermost narrowing. This interpreter makes use of leftmost SLD-resolution and a new rule called "simplification" [13]. Lefmost SLD-resolution constitutes a *sound* and *complete* inference procedure [12] if the given program is canonical (confluent and terminating) and its functions are everywhere-defined: a function f is said to be everywhere-defined if a rewriting step can be applied to every term of the form $f(\underline{d})$, where $\underline{d}$ is a tuple of ground constructor terms. The simplification rule does not affect soundness nor completeness but allows to cut down the search space [13].

5.1. Prolog with Simplification

Example 5.1 *Consider the following clauses and goals (in flat form):*

$$\begin{array}{l}
reverse(nil) = nil \leftarrow \\
reverse(cons(X, Xs)) = append(Y, cons(X, nil)) \leftarrow reverse(Xs) = Y \\
append(nil, Y) = Y \leftarrow \\
append(cons(X, Xs), Y) = [X|Z] \leftarrow append(Xs, Y) = Z
\end{array}$$

$$\leftarrow append(U, V) = T, append(T, W) = nil$$

Note that (the non flat version of) the above program is canonical and its functions are everywhere-defined [12]. A correct answer substitution for the goal is $\{U/nil, V/nil, W/nil\}$. *This can be computed by by the following leftmost SLD-refutation:*

$$\begin{array}{l}
\leftarrow append(U, V) = T, append(T, W) = nil \\
\qquad\qquad \big| \; U/nil, V/T \\
\leftarrow append(T, W) = nil \\
\qquad\qquad \big| \; T/nil, W/nil \\
\qquad\qquad \leftarrow
\end{array}$$

In general (leftmost) SLD-resolution simulates innermost narrowing [8]. Thus, the narrowing of the equational goal $\leftarrow append(append(X, Y), Z) = nil$ yields $\leftarrow append(Y, Z) = nil$ using the substitution $\{X/nil\}$; this corresponds to the resolution of the relational goal $\leftarrow append(X, Y) = U, append(U, Z) = nil$ which yields $\leftarrow append(Y, Z) = nil$ using the substitution $\{X/nil, U/Y\}$.

The normalization refinement is simulated here by means of an operation called *simplification*. Simplification is intuitively SLD-resolution restricted to literals which are "sufficiently instantiated". This has indeed a close connection to rewriting, since (1) SLD-resolution is equivalent to narrowing and (2) rewriting is also a restriction of narrowing to terms that are sufficiently instantiated.

A clause is called *deterministic* iff the lefthand side of its clause head is not unifiable with the lefthand sides of other clause heads. A literal is said to be *simplifiable* by a deterministic clause if the lefthand side of clause head matches the lefthand side of the literal. A goal is said to be *simplified* if it does not contain any simplifiable literal.

Simplification is then defined as follows:

One-step simplification:

$$\frac{\leftarrow L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_n}{\leftarrow (L_1, \ldots, L_{i-1}, B_1, \ldots, B_m, L_{i+1}, \ldots, L_n)\sigma}$$

if L_i is simplifiable by $A \leftarrow B_1, \ldots, B_m$ and $\sigma = mgu(A, L_i)$. Note that σ may be undefined (due to unification failure), in which case the goal is determinately replaced by "fail", which provokes immediate backtracking. Otherwise (σ is defined), the goal is determinately replaced by the resolvent.

Example 5.2 *Consider the previous SLD-derivation which, upon backtracking, generates the unsolvable goal* $\leftarrow append(U', V) = T', append(cons(X, T'), W) = nil$*. Although that goal causes an infinite loop in Prolog, it can be simplified to "fail", since the rightmost literal is simplifiable (by the second append clause) and there is unification failure.*

Simplification is however a weaker notion than rewriting, and the following example illustrates the difference.

Example 5.3 *Consider the following flat definition for multiplication:*

$$0 * Y = 0 \leftarrow$$
$$s(X) * Y = Z \leftarrow X * Y = W, X + W = Z$$
$$0 + Y = Y \leftarrow$$
$$\leftarrow s(X) + Y = s(Z) : -X + Y = Z$$

The goal $\leftarrow U + V = W, 0 * W = 0$ *is simplified as* $\leftarrow U + V = W$ *but not as the empty goal, while reduction on* $\leftarrow 0 * (U + V) = 0$ *directly yields the empty goal.*

In general, simplification is weaker than rewriting when there are rewrite rules that delete variables on the lefthand side (as above) or when there are conditional rewrite rules (our notion of deterministic clauses is too weak to take into account such rules).

In the following we explain how to simulate the simplification rule through a Prolog interpreter. We first describe a general meta-interpreter, then explain how to implement the simplification rule and in particular the embedded matching operation.

5.2. Meta-interpreter in Prolog

5.2.1. Difference-lists

The interpreter will be manipulating goals, sometimes taking them apart and sometimes joining them back together. A necessary operation is of course the concatenation operation, and that leads us to use the well-known difference-lists' technique here.

The difference-list representation of a clause $A \leftarrow B_1, \ldots, B_n$ is

$$clause(A, [B_1, \ldots, B_n|U] - U)$$

where `U` is some distinct variable.

For instance, the difference-list definition of `append` is:

```
clause(append(nil,Y)=Y,U-U).
clause(append(cons(X,Xs),Y)=cons(X,Z),[append(Xs,Y)=Z|U]-U).
```

The difference-list representation of a goal can be defined similarly. For example, $[B_1, \ldots, B_n|U] - U$ stands for the goal :- $B_1, \ldots, B_n$.

The interpreter therefore works on difference-lists and (leftmost) SLD-resolution is implemented as:

```
resolve(X-Y,X-Y) :- X==Y,!.              % empty goal
resolve([X|Y]-Z,U-Z) :- clause(X,U-Y).   % nonempty goal
```

5.2.2. Toplevel

To emphasize the priority of determinate computations over nondeterminate ones, goals are always simplified before any SLD-resolution step. This strategy is depicted in the following toplevel loop:

```
prolog_with_simp(X-Y) :- X==Y,!.          % empty goal
prolog_with_simp(X) :-                    % nonempty goal
     simplify(X,Y),                       % simplify first
     resolve(Y,Z),                        % then resolve
     prolog_with_simp(Z).
```

where `simplify(X,Y)` means that the goal `Y` is a simplified form of `X`.

It remains to discuss about the implementation of `simplify`.

5.2.3. Simplification

Goals are simplified in zero or more steps, by application of the (one-step) simplification rule, until the simplified form is obtained. The essential point here is that simplification can follow a left-to-right strategy. The application of the simplification rule on a literal will not affect the "simplification status" of literals to its left.

Proposition 5.1 *Consider a goal* $\leftarrow L_1, \ldots, L_n$ *which is one-step simplified into* $\leftarrow (L_1, \ldots, L_{i-1}, B_1, \ldots, B_m, L_{i+1}, \ldots, L_n)\sigma$*. Suppose that all literal* L_j *to the left of* L_i *(i.e.* $j < i$*) is already simplified. Then all literal* $L_j\sigma$ $(j < i)$ *is still simplified.*

This naturally justifies using a one-pass left-to-right strategy:

```
simplify(X-Y,X-Y) :- X==Y,!.
simplify([X|Y]-Z,U-W) :-
     simp_l(X,U-V),      % simplify the leftmost literal first
     simplify(Y-Z,V-W).  % then followed by the others
```

where `simp_l` stands for the simplification of a literal.

Whether a literal $f(s_1, \ldots, s_n) = s$ is simplifiable can be tested by matching the input arguments s_i against the input parameters t_i of some deterministic clause head $f(t_1, \ldots, t_n) = t$. If matching is successful, then the simplification process can proceed to simplify the clause body (from-left-to-right), with variable-bindings that are computed, as usual, by unification. However, it is only necessary to unify s against t, since the s_i has already been matched against the t_i.

The desired behavior can be simulated by writing every deterministic clause

$$f(t_1, \ldots, t_n) = t \text{ :- } l_1, \ldots, l_m$$

as

```
simp_l(f(X1,...,Xn)=X,U1-U) :-
     match(X1,t1),          % match X1 against t1
     ...
     match(Xn,tn),          % match Xn against tn
     !,                     % literal is simplifiable
     unify(X,t),            % syntactic unification
     simp_l(l1,U1-U2),      % simplify leftmost literal first
     simp_l(l2,U2-U3),      % followed by the second
     ...
     simp_l(lm,Um-U).       % followed by the last
```

where `X1,...,Xn,X,U1,...,Um,U` are distinct variables. Note that the cut (`!`) expresses the determinate nature of simplification.

The above clauses take into account the case where the given literal is simplifiable. So it is sufficient to complete that definition by:

```
simp_l(X,[X|Y]-Y).             % should be last simp_l clause
```

for the case where the given literal is already simplified.

5.2.4. Implementation of `match`

To implement the predicate `match` in Prolog, it is necessary to forbid the instantiation of the term to be matched. It is clear that such an "impure" feature can only be simulated via some impure primitives of Prolog. The idea of our implementation is very simple: to decompose matching calls into syntactic unification and some common (impure) Prolog primitives (i.e., `nonvar/1` and `==/2`). The following expansion rules are used to implement matching:

- "`match(X,c)`", where `c` is a constant, expands to "`X==c`".
- "`match(X,c(t1,...,tn))`" expands to "`nonvar(X),unify(X,c(Y1,...,Yn)),match(Y1,t1),...,match(Yn,tn)`", where the `Yi`'s are distinct variables.
- "`match(X,Y)`" expands to
  - "`unify(X,Y)`", if the occurrence of the variable `Y` is the first (leftmost) in the clause. In this case, `X` and `Y` can be immediately unified at compile-time.
  - "`X==Y`", if the occurrence of `Y` is not the first.

Let's illustrate with an example. The append clauses are deterministic clauses. The corresponding `simp_l` clauses are:

```
simp_l(append(A,B)=C,U-U) :- match(A,nil),
                             match(B,Y),!,
                             unify(C,Y).
simp_l(append(A,B)=C,U-V) :- match(A,cons(X,Xs)),
                             match(B,Y),!,
                             unify(C,cons(X,Z)),
                             simp_l(append(Xs,Y)=Z,U-V).
```

In the above clauses, to match `A` against a constant `nil` can be simulated by the call `A==nil`; to match `A` against a cons-structure `cons(X,Xs)` can be simulated by the call "`nonvar(A)`, `unify(A,cons(X,Xs))`"; and finally, to match `B` against `Y` can be simulated by the (compile-time) unification of `Y` and `B`. We thus obtain:

```
simp_l(append(A,B)=C,U-U) :- A==nil,!,unify(C,B).
simp_l(append(A,B)=C,U-V) :- nonvar(A),unify(A,cons(X,Xs)),!,
                             unify(C,[X|Z]),
                             simp_l(append(Xs,B)=Z,U-V).
```

6. Experimental Results

6.1. Outermost strategy

In section 4, we showed how the outermost computation rule can be implemented in Prolog. Since the outermost rule simulates lazy narrowing, we have obtained an implementation of lazy narrowing into Prolog. Some preprocessing of clauses and goals is required so that when Prolog interprets them, lazy narrowing is simulated.

The difference with the K-WAM approach [3] is that we handle the outermost rule at the interpretative level, whereas K-WAM handles it directly. On Sicstus Prolog, our approach is very comparable to the performances of emulated K-WAM.

Table 1. Benchmarks in milliseconds on VAX 8700

Query	Our approach	K-WAM
rev: 40 elements	70	82.5
revI: 40 elements	1800	1200
fibonacci: 15	150	285

rev : naive reverse
revI : naive reverse in inverted mode

6.2. Innermost strategy

Following the ideas of section 5, we have implemented a meta-interpreter in Sicstus Prolog and conducted some preliminary experiments (cf. tablee 1) to demonstrate its feasibility.

Table 2. Benchmarks in milliseconds on SUN 3/60

	Prolog with Simplification	Prolog
rev: 500 elements	16620	4540
revI: 60 elements	4820	1399
psort: 10 elements	30059	3417000
10-queens	3980	103600

psort : permutation sort of a list already sorted in inverse order

In the first two examples (rev and revI), the interpreter is about four times slower than Prolog. The difference stems from the interpretative overhead incurred by our approach, but that is more than compensated by impressive speedups in the case of generate-and-test programs (psort and queens). Benchmarks of table 3 shows that it is only slightly slower that the more complex approach taken in the A-WAM [18].

Table 3. Comparison with A-WAM on SUN 4

	Prolog with Simplification	A-WAM
rev: 30 elements	30	19
revI: 30 elements	350	210
psort: 8 elements	2450	1500

The difference with A-WAM is that simplification does not simulate full rewriting, either because it can't take into account the deletion of lefthand side variables, or because our notion of deterministic clauses is too weak to consider conditional term rewriting. However, rules that are conditional on simple tests (as in the case of quicksort) can actually be considered in a straightforward way (see [7]).

7. Conclusion

We have shown how Prolog can be used to simulate lazy narrowing and normalized innermost narrowing. It turns out that, without sacrificing portability, our method compares honorably with specialized abstract machines like the A-WAM and the

K-WAM. Moreover, taking into account the effort that goes into the development of standard Prolog compilers, we should be able to benefit immediately from major advances in in the field. In this sense, Prolog can be considered a suitable language for implementing narrowing.

References

[1] M. Bellia and G. Levi. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236, 1986.

[2] S. Bonnier and J. Maluszynski. Towards a clean amalgamation of logic programs with external procedures. In *Proceedings of the 5th International Conference on Logic Programming, Seattle*, 1988.

[3] P. Bosco, C. Cecchi, and C. Moiso. An extension of WAM for K-LEAF. In *Proceedings of the 6th International Conference on Logic Programming, Lisboa*, pages 318–333, 1989.

[4] P. Bosco, E. Giovannetti, C. Moiso, and C. Palamidessi. Comments on "Logic programming with equations". *Journal of Logic Programming*, 11:85–89, 1991.

[5] P. Bosco, E. Giovannetti, and G. Moiso. Refined strategies for semantic unification. In *Proceedings of TAPSOFT'87, Lecture Notes in Computer Science 150*, pages 276–290. Springer-Verlag, 1987.

[6] P. Cheong. Compiling lazy narrowing into Prolog. Technical Report 25, LIENS, 1990.

[7] P. Cheong and L. Fribourg. Efficient integration of simplification into Prolog. In *Proceedings of PLILP'91, Lecture Notes in Computer Science 528*, pages 359–370. Springer-Verlag, 1991.

[8] P. Deransart. An operational semantics of prolog programs. In *Programmation Logique, Perros-Guirec, CNET-lannion*, 1983.

[9] N. Dershowitz and A. Josephson. An implementation of narrowing, the RITE way. In *Proceedings of the IEEE Symposium on Logic Programming, Salt Lake City*, pages 31–40, 1986.

[10] R. Echahed. On completeness of narrowing strategies. In *Proceedings of CAAP 88, Lecture Notes in Computer Science 299*, pages 89–91. Springer-Verlag, 1988.

[11] M. Fay. First-order unification in an equational theory. In *Proceedings of the 4th Workshop on Automated Deduction, Austin*, pages 161–167, 1979.

[12] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proceedings of the IEEE Symposium on Logic Programming, Boston*, pages 172–184, 1985.

[13] L. Fribourg. Prolog with simplification. In K. Fuchi and M. Nivat, editors, *Programming of future generation computers*, pages 161–183. Elsevier Science Publishers B.V. (North Holland), 1988.

[14] J. Gallier and S. Raatz. Extending SLD-resolution to equational Horn clauses using E-unification. *Journal of Logic Programming*, 6:3–43, 1989.

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

[16] E. Giovannetti and C. Moiso. A completeness result for E-unification algorithms based on conditional narrowing. In *Proccedings of Foundations of Logic and Functional Programming*, volume 306, pages 318–334. Springer-Verlag, 1987.

[17] J. Goguen and J. Meseguer. EQLOG: equality, types and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Functional and Logic Programming*, pages 295–363. Prentice-Hall, 1986.

[18] M. Hanus. Compiling logic programs with equality. In *PLILP'90, Lecture Notes in Computer Science 456*, pages 387–401. Springer-Verlag, 1990.

[19] S. Hölldobler. From paramodulation to narrowing. In *Proceedings of the 5th International Conference on Logic Programming, Seattle*, pages 327–342, 1988.

[20] J. Hullot. Canonical forms and unification. In *Proceedings of the 5th Conference on Automated Deduction, Lecture Notes in Computer Science 87*, pages 318–334. Springer-Verlag, 1980.

[21] H. Hussmann. Unification in conditional equational theories. In *Proceedings of the EUROCAL '85 conference, Lecture Notes in Computer Science 204*, pages 543–553. Springer-Verlag, 1985.

[22] R. Kowalski. Logic programming. In *Proceedings of IFIP*, pages 133–145, 1983.

[23] H. Kuchen, R. Loogen, J. Moreno, and M. Rodríguez. Graph-based implementation of a functional logic language. In *ESOP' 90, Lecture Notes in Computer Science 432*, pages 279–290. Springer-Verlag, 1990.

[24] G. Levi, C. Palamidessi, P. Bosco, E. Giovannetti, and C. Moiso. A complete semantic characterization of K-LEAF, a logic language with partial functions. In *Proceedings of the IEEE Symposium on Logic Programming, San Francisco*, pages 318–327, 1987.

[25] L. Lloyd. *Foundations of logic programming.* Symbolic Computation Series. Sringer-Verlag, second edition, 1987.

[26] E. Mendelson. *Introduction to Mathematical Logic.* Van Nostrand, 1979.

[27] A. Middeldorp and E. Hamoen. Counterexamples to completeness results for basic narrowing, 1991. Technical Report, Deliverable D2.3, ESPRIT BASIC RESEARCH ACTION 3020.

[28] S. Narain. *LOG(F), an optimal combination of logic programming, rewriting and lazy evaluation.* PhD thesis, Department of Computer Science, University of California, Los Angeles, 1988.

[29] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7:295–317, 1989.

[30] Padawitz. Strategy controlled reduction and narrowing. In *Proceedings of Rewriting Techniques and Applications, Lecture Notes in Computer Science 256*, pages 242–255. Springer-Verlag, 1987.

[31] G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.

[32] U. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the IEEE Symposium on Logic Programming, Boston*, pages 138–151, 1985.

[33] J. Siekmann. Universal unification. In *7th Conference on Automated Deduction, Lecture Notes in Computer Science 170.* Springer-Verlag, 1984.

[34] P. Subrahmanyam and Y. You. Conceptual basis and evaluation strategies for integrating functional and logic programming. In *Proceedings of the IEEE Symposium on Logic Programming, Atlantic City*, pages 144–155, 1984.

[35] A. Yamamoto. A theoretical combination of SLD-resolution and narrowing. In *Proceedings of the 4th International Conference on Logic Programming, Melbourne*, pages 470–481, 1987.

[36] Y. You. Enumerating outer narrowing derivations for constructor-based term rewriting systems. *Journal of Symbolic Computation*, 7:319–343, 1989.