THE INTEGRATION OF FUNCTIONS INTO LOGIC PROGRAMMING: FROM THEORY TO PRACTICE

MICHAEL HANUS

▷ Functional and logic programming are the most important declarative programming paradigms, and interest in combining them has grown over the last decade. Early research concentrated on the definition and improvement of execution principles for such integrated languages, while more recently efficient implementations of these execution principles have been developed so that these languages became relevant for practical applications. In this paper we survey the development of the operational semantics as well as the improvement of the implementation of functional logic languages. ◁

1. INTRODUCTION

Interest in the amalgamation of functional and logic programming languages has increased since the beginning of the last decade. Such integrated languages have advantages from the functional and the logic programming point of view. In comparison with pure functional languages, functional logic languages have more expressive power due to the availability of features like function inversion, partial data structures, and logical variables [109]. In comparison with pure logic languages, functional logic languages have a more efficient operational behavior since functions allow more deterministic evaluations than predicates. Hence the integration of functions into logic programming can avoid some of the impure control

Address correspondence to M. Hanus, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany. Email: michael@mpi-sb.mpg.de

This work has been partially supported by the German Ministry for Research and Technology (BMFT) under grant ITS 9103.
Received May 1993; accepted December 1993.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010

features of Prolog like the cut operator. These principal considerations were the motivation for integrating both language types.

Depending on the initial point of view, the integration of functional and logic programming languages has been tackled in two ways. From a functional programming point of view, logic programming aspects can be integrated into functional languages by permitting logical variables in expressions and replacing the matching operation in a reduction step by unification [109].[1] From a logic programming point of view, functions can be integrated into logic languages by combining the resolution principle with some kind of functional evaluation. Since we are mainly interested in logic programming, we concentrate this survey on the latter aspect. However, we want to point out that both views yield similar operational principles for the amalgamated languages.

The integration of functions into logic programming is very simple from a syntactic point of view. For this purpose, we have to extend the logic language by:

1. A method to define new functions.
2. A possibility to use these functions inside program clauses.

To realize the first point, we could allow the implementation of functions in an external (functional) language [14]. A more interesting alternative is the direct integration of function definitions into the logic language. For this purpose one has to permit program clauses defining the equality predicate. Equality "=" is a predefined predicate in Prolog systems which is satisfied iff both arguments are syntactically equal (i.e., syntactic unification of both arguments). Hence this predicate can be defined by the fact

```
X = X.
```

By admitting new clauses for "=", we are able to express that syntactically different terms are semantically equal. In particular, a function applied to some argument terms should be equal to its result. For instance, the following equality clauses define the semantics of the function `append` for concatenating lists (we use the Prolog notation for lists [122]):

```
append([],L)    = L.
append([E|R],L) = [E|append(R,L)].
```

Using these clauses for equality, we can prove that the term `append([1,2],[3])` is equal to `[1,2,3]`. Note that this method of defining functions is the same as in modern functional languages like Haskell [67], Miranda [127], or ML [63], where functions are defined by argument patterns. We can also define functions by conditional equations, where we may use arbitrary predicates in the conditions. For instance, the maximum function on naturals can be defined by the following conditional equations:

```
max(X,Y) = X :- X >= Y.
max(X,Y) = Y :- X =< Y.
```

[1] Other alternatives to integrate logic programming aspects into functional languages are set abstractions [26, 27, 112, 117, 118] or logical arrays [72].

Due to the logic part of the integrated language, the proof of the condition may require a search for the right instantiation of new variables occurring in the condition. This is shown in the following definition of a function computing the last element of a list:

```
last(L) = E :- append(_,[E]) = L.
```

If such conditional equations should be applied to compute the value of a functional expression, the validity of the condition must be proved. For instance, in order to evaluate the term `last([1,2])`, we have to find a solution to the equation `append(_,[E])=[1,2]`. Techniques to compute such solutions will be presented in Section 2.

After defining functions by equality clauses, the programmer can use these functions in expressions occurring in goals of the logic program. For instance, if the membership in a list is defined by the clauses

```
member(E,[E|L]).
member(E,[F|L]) :- member(E,L).
```

specifying the predicate `member`, we can use the `append` function in goals where list terms are required. In the goal

```
?- member(E,append([1],[2])).
```

the second argument is equal to the list `[1,2]` and therefore the two answers to this goal are `E=1` and `E=2`. This kind of amalgamated language is known as *logic programming with equality* and has a clearly defined *declarative semantics* [50, 71, 106]. It is similar to the well-known Horn clause logic [83], but with the difference that the equality predicate "`=`" is always interpreted as the identity on the carrier sets in all interpretations. Therefore, we omit the details of the declarative semantics in this survey.

The definition of the *operational semantics* is not so easy. In the last example the evaluation of the goal is obvious: first replace the functional term `append([1],[2])` by the equivalent result term `[1,2]` and then proceed with the goal `member(E,[1,2])` as in logic programming. However, what happens if the functional term contains free variables so that it cannot be evaluated to an equivalent term without the function call? For instance, consider the goal

```
?- append(L,[3,4]) = [1,2,3,4].
```

Clearly, the variable `L` should be instantiated to `[1,2]`, which is the unique solution to this equation, but how can we compute such solutions? In general, we have to compute unifiers w.r.t. the given equational axioms which is known as *E-unification* [44]. Replacing standard unification by E-unification in a resolution step yields a computational mechanism to deal with functions in logic programs [43, 49]. Unfortunately, E-unification can be a very hard problem even for simple equations (see [116] for a survey). For instance, if we state the associativity of the `append` function by the equation

```
append(append(L,M),N) = append(L,append(M,N)).
```

then it is known that the corresponding E-unification problem is decidable, but there may exist an infinite set of pairwise incomparable E-unifiers. Thus a complete E-unification procedure must enumerate all these unifiers. Moreover, it is also known that E-unification is undecidable even for simple equational axioms like distributivity and associativity of functions [115]. Therefore, van Emden and Yukawa [128] state that "one of the reasons why logic programming succeeded where other resolution theorem proving had failed ... was that in logic programming equality was avoided like the plague." Fortunately, there are restrictions on the definition of the equality predicate which are acceptable from a programming point of view and which ensure the existence of a usable E-unification algorithm.

In the beginning of research on amalgamated functional logic languages, many proposals were made to restrict the generality of the equality axioms and to develop appropriate execution principles (see [31] for a good collection of these proposals and [9] for a short survey). Since these execution principles seemed complicated and were not implemented as efficiently as pure logic languages, logic programmers were often doubtful about the integration of functions into logic programming. However, this has changed since new efficient implementation techniques have been developed for functional logic languages in recent years. In comparison to implementations of pure logic languages, these new techniques cause no overhead because of the presence of functions. Moreover, in many cases functional logic programs are more efficiently executed than their relational equivalents without using impure control features like "cut."

In the following text, we survey the operational principles and the implementation techniques of functional logic languages. Section 2 discusses the various operational semantics proposed for functional logic languages. We introduce basic notions by discussing computational methods for a rather general class of functional logic programs in Section 2.1. Then we consider the important subclass of constructor-based programs and discuss eager and lazy evaluation strategies in Sections 2.2 and 2.3. Section 2.4 highlights problems caused by conditional equations, and Section 2.5 introduces a completely different class of evaluation strategies which sacrifice completeness for the sake of efficiency. Implementations of these strategies are discussed in Section 3. Section 3.1 shows straightforward implementations by compiling into high-level languages, and Section 3.2 outlines the various low-level abstract machines developed for the execution of functional logic programs during the last few years.

2. OPERATIONAL PRINCIPLES FOR FUNCTIONAL LOGIC LANGUAGES

In order to give a precise definition of the operational semantics of functional logic languages and to fix the notation used in the rest of this paper, we recall basic notions from term rewriting [32] and logic programming [83].

If $\mathcal{F}$ is a set of *function symbols* together with their arity[2] and $\mathcal{X}$ is a countably infinite set of *variables*, then $\mathcal{T}(\mathcal{F},\mathcal{X})$ denotes the set of *terms* built from $\mathcal{F}$ and $\mathcal{X}$. If $t \notin \mathcal{X}$, then $\mathcal{H}ead(t)$ is the function symbol heading term t. $\mathcal{V}ar(t)$ is the set

[2] For the sake of simplicity, we consider only single-sorted programs in this paper. The extension to many-sorted signatures is straightforward [106]. We also assume that $\mathcal{F}$ contains at least one constant.

of variables occurring in a term t (similarly for the other syntactic constructions defined below, like literal, clause, etc.) A *ground term* t is a term without variables, i.e., $\mathcal{V}ar(t) = \emptyset$. A *substitution* σ is a homomorphism from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ into $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that its *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. We frequently identify a substitution σ with the set $\{x \mapsto \sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. The *composition of two substitutions* ϕ and σ is defined by $\phi \circ \sigma(x) = \phi(\sigma(x))$ for all $x \in \mathcal{X}$. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. A unifier σ is called *most general* (*mgu*) if for every other unifier σ', there is a substitution ϕ with $\sigma' = \phi \circ \sigma$. A *position* p in a term t is represented by a sequence of natural numbers, $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s (see [32] for details). If p and q are positions, we write $p \preceq q$ if p is a prefix of q. $p \cdot q$ denotes the concatenation of positions p and q.

Let $\rightarrow$ be a binary relation on a set S. Then $\rightarrow^*$ denotes the transitive and reflexive closure of the relation $\rightarrow$, and $\leftrightarrow^*$ denotes the transitive, reflexive and symmetric closure of $\rightarrow$. $\rightarrow$ is called *terminating* if there are no infinite chains $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow \cdots$. $\rightarrow$ is called *confluent* if for all $e, e_1, e_2 \in S$ with $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$, there exists an element $e_3 \in S$ with $e_1 \rightarrow^* e_3$ and $e_2 \rightarrow^* e_3$.

Let $\mathcal{P}$ be a set of *predicate symbols* including the binary equality predicate =. A *literal* $p(t_1, \ldots, t_n)$ consists of an n-ary predicate symbol applied to n argument terms. An *equation* is a literal with = as predicate symbol. We use the infix notation $t_1 = t_2$ for equations. A *clause* has the form

$$L_0 \texttt{ :- } L_1, \ldots, L_n.$$

($n \geq 0$), where $L_0, \ldots, L_n$ are literals. It is called (*conditional*) *equation* if L_0 is an equation, and *unconditional equation* if L_0 is an equation and $n = 0$.[3] Since unconditional equations $l = r$ and conditional equations $l = r \texttt{ :- } C$ will be used only from left to right, we call them (*rewrite*) *rules*, where l and r are the left- and right-hand side, respectively. A clause is a *variant* of another clause if it is obtained by a bijective replacement of variables by other variables. A *functional logic program* or *equational logic program* is a finite set of clauses. In the following text, we assume that P is a functional logic program.

2.1. A Sound and Complete E-Unification Method: Narrowing

If we have to evaluate a function applied to ground terms during unification in a functional logic program, we can simply evaluate this function call as in functional languages by applying appropriate rules to this call. For instance, the function call `append([],[2])` is evaluated by matching the left-hand side of the first rule for `append` against this call (this binds variable `L` in the equation to `[2]`) and replacing this function call by the instantiated right-hand side of the rule (i.e., `[2]`). This is called a *rewrite step*. Generally, $t \rightarrow_P s$ is a rewrite step if there exist a position p, a rule $l = r \in P$,[4] and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. In this

[3] The completeness of some particular operational semantics requires more conditions on conditional equations like the absence of extra variables in conditions. We will discuss these restrictions later.

[4] At the moment we consider only unconditional rules. The extension to conditional rules is discussed in Section 2.4.

case t is called *reducible* (at position p). The term t is *irreducible* or in *normal form* if there is no term t' with $t \rightarrow_P t'$. If the program P is known from the context, we omit the index $_P$ in the rewrite arrow. For instance, the ground term `append([1,2],[3])` is evaluated to its normal form `[1,2,3]` by the following three rewrite steps, provided that P contains the above rules defining `append`:

```
append([1,2],[3])   →   [1|append([2],[3])]
                    →   [1,2|append([],[3])]
                    →   [1,2,3]
```

If there is a function call containing free variables in arguments, then it is generally necessary to instantiate these variables to appropriate terms in order to apply a rewrite step. This can be done by using unification instead of matching in the rewrite step which is called *narrowing* [119]. Hence, in a narrowing step we unify a (nonvariable) subterm of the goal with the left-hand side of a rule and then we replace the instantiated subterm by the instantiated right-hand side of the rule. To be precise, we say a term t is *narrowable* into a term t' if:

1. p is a nonvariable position in t (i.e., $t|_p \notin \mathcal{X}$).
2. $l = r$ is a new variant[5] of a rule from P.
3. The substitution σ is a mgu of $t|_p$ and l.
4. $t' = \sigma(t[r]_p)$.

In this case, we write $t \leadsto_{[p,l=r,\sigma]} t'$ or simply $t \leadsto_{[l=r,\sigma]} t'$ or $t \leadsto_{\sigma} t'$ if the position or rule is clear from the context. If there is a narrowing sequence $t_0 \leadsto_{\sigma_1} t_1 \leadsto_{\sigma_2} \cdots \leadsto_{\sigma_n} t_n$, we write $t_0 \leadsto^*_{\sigma} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_2 \circ \sigma_1$. Thus in order to solve the equation `append(L,[2])=[1,2]`, we apply the second `append` rule (instantiating `L` to `[E|R]`) and then the first `append` rule (instantiating `R` to `[]`):

$$
\begin{array}{lll}
\texttt{append(L,[2])=[1,2]} & \leadsto_{\{\texttt{L}\mapsto\texttt{[E|R]}\}} & \texttt{[E|append(R,[2])]=[1,2]} \\
 & \leadsto_{\{\texttt{R}\mapsto\texttt{[]}\}} & \texttt{[E,2]=[1,2]}
\end{array}
$$

The final equation can be immediately proved by standard unification which instantiates `E` to `1`. Therefore, the computed solution is $\{$`L`$\mapsto$`[1]`$\}$.[6]

Narrowing is a sound and complete method to solve equations w.r.t. a confluent and terminating set of rules E. In order to state a precise proposition on soundness and completeness, we call an equation $s = t$ *valid* (w.r.t. an equation set E) if $s \leftrightarrow^*_E t$. By Birkhoff's completeness theorem, this is equivalent to the semantic validity of $s = t$ in all models of E. Therefore, we also write $s =_E t$ in this case. Now narrowing is a sound and complete E-unification method in the sense of the following theorem.[7]

[5]Similarly to pure logic programming, rules with fresh variables must be used in a narrowing step in order to ensure completeness.

[6]For the sake of readability, we omit the instantiation of clause variables in the substitutions of concrete narrowing derivations.

[7]Although we have defined rewrite and narrowing steps only on terms, it is obvious how to extend these definitions to literals and sequences of literals.

Theorem 2.1 (Hullot [69]). Let E be a finite set of unconditional equations so that $\rightarrow_E$ is confluent and terminating.

1. *(Soundness) If $s\!=\!t \rightsquigarrow^*_\sigma s'\!=\!t'$ and μ is a mgu for s' and t', then $\mu(\sigma(s)) =_E \mu(\sigma(t))$.*
2. *(Completeness) If $\sigma'(s) =_E \sigma'(t)$, then there exist a narrowing derivation $s\!=\!t \rightsquigarrow^*_\sigma s'\!=\!t'$, a mgu μ for s' and t', and a substitution ϕ with $\phi(\mu(\sigma(x))) =_E \sigma'(x)$ for all $x \in \mathcal{V}ar(s) \cup \mathcal{V}ar(t)$.*

The first proposition states that each substitution computed by narrowing is a unifier w.r.t. E, and the second proposition ensures that each unifier w.r.t. E is covered by a more general computed substitution. This theorem justifies narrowing as the basis to execute functional logic programs. The confluence requirement can often be established by applying a Knuth/Bendix completion procedure to transform a set of equations into a corresponding confluent one [77]. As an alternative there exist syntactic restrictions which ensure confluence (orthogonal rules) [32] (see also Section 2.3).

It is well known that the termination requirement for the completeness of narrowing can be dropped if the class of substitutions is restricted. A substitution σ is called *normalized* if $\sigma(x)$ is in normal form for all $x \in \mathcal{D}om(\sigma)$. If E is a finite set of unconditional equations so that $\rightarrow_E$ is confluent (and not necessarily terminating), then narrowing is complete w.r.t. normalized substitutions (i.e., the second proposition of Theorem 2.1 holds if σ' is normalized).

The difficulty in a narrowing derivation is the application of a suitable rule at an appropriate subterm in a goal. For instance, if we apply the first `append` rule to the goal `append(L,[2])=[1,2]`, we would obtain the new goal `[2]=[1,2]`, which is unsolvable. In general, there is no answer to this problem. In order to be complete and to find all possible solutions, Theorem 2.1 implies that *each* rule must be applied at *each* nonvariable subterm of the given goal. Hence using this simple narrowing method to execute functional logic programs yields a huge search space and many infinite paths even for simple programs. In order to use narrowing as a practical operational semantics, further restrictions are necessary which will be discussed in the following text.

An important restriction which has been known for a long time is *basic narrowing* [69]. This means that a narrowing step is only performed at a subterm which is not part of a substitution (introduced by previous unification operations), but belongs to an original program clause or goal. Basic narrowing can be defined by managing a set of basic positions. If

$$t_0 \rightsquigarrow_{[p_1,l_1\rightarrow r_1,\sigma_1]} t_1 \rightsquigarrow_{[p_2,l_2\rightarrow r_2,\sigma_2]} \cdots \rightsquigarrow_{[p_n,l_n\rightarrow r_n,\sigma_n]} t_n$$

is a narrowing derivation, then the sets $B_0, \ldots, B_n$ of *basic positions* are inductively defined by

$$\begin{array}{rcl} B_0 & = & \{p \mid p \text{ position in } t_0 \text{ with } t_0|_p \notin \mathcal{X}\}, \\ B_i & = & (B_{i-1}\backslash\{p \in B_{i-1} \mid p_i \preceq p\}) \\ & & \cup \{p_i \cdot p \mid p \text{ position in } r_i \text{ with } r_i|_p \notin \mathcal{X}\}, \quad i > 0. \end{array}$$

The above sequence is a *basic narrowing derivation* if $p_i \in B_{i-1}$ for $i = 1, \ldots, n$.

Example 2.1. Consider the following equation specifying a property of the reverse function:

```
rev(rev(L)) = L.
```

Applying this rule to the literal `rev(X)=X` yields the infinite narrowing derivation

$$\begin{array}{lll} \texttt{rev(X) = X} & \leadsto_{\{\texttt{X}\mapsto\texttt{rev(X1)}\}} & \texttt{X1 = rev(X1)} \\ & \leadsto_{\{\texttt{X1}\mapsto\texttt{rev(X2)}\}} & \texttt{rev(X2) = X2} \\ & \leadsto & \cdots. \end{array}$$

However, the second narrowing step is not basic since the subterm `rev(X1)` belongs to the substitution part introduced in the first step. In a basic narrowing derivation it is not allowed to reduce this term. Hence the only basic narrowing derivation of the same initial equation is

$$\texttt{rev(X)=X} \quad \leadsto_{\{\texttt{X}\mapsto\texttt{rev(X1)}\}} \quad \texttt{X1=rev(X1)}.$$

Since the last equation is not syntactically unifiable, there exists no solution to the initial equation. This example shows that the restriction to basic positions can reduce an infinite search space to a finite one.

Although the number of admissible narrowing positions is reduced and therefore the search space is smaller compared to simple narrowing, basic narrowing is sound and complete in the sense of Theorem 2.1. The important aspect of the basic strategy is that searching for narrowing positions inside substitutions for program variables is superfluous. All such positions must be present in the program, i.e., in the initial term or in the right-hand sides of rules. As we will see in Section 3.2.1, this is the key for an efficient compiler-based implementation of narrowing since the basic narrowing positions can be computed at compile time.

It is interesting to note that basic narrowing can give a sufficient criterion for the termination of all narrowing derivations:

Proposition 2.1 (Termination of narrowing [69]). Let $E = \{l_i \!=\! r_i \mid i = 1,\ldots,n\}$ be a finite set of unconditional equations so that $\rightarrow_E$ is confluent and terminating. If any basic narrowing derivation starting from any r_i terminates, then all basic narrowing derivations starting from any term are finite.

Therefore basic narrowing is a *decision procedure for E-unification* if the conditions of the last proposition hold. In particular, this is the case when all right-hand sides of the rules are variables as in Example 2.1.

The basic narrowing positions can be further restricted by also discarding those narrowing positions which are strictly left of the position used in a narrowing step. This strategy is called *left-to-right basic narrowing* and remains to be complete (see [64] for details). The set of admissible basic narrowing derivations can also be restricted by introducing redundancy tests like normalization properties of the computed substitutions. Using a sophisticated set of such tests one can obtain a narrowing procedure where each different narrowing derivation leads to different computed solutions (*LSE-narrowing* [12]).

2.2. *Narrowing Strategies for Constructor-Based Programs*

In order to implement a functional logic language based on basic narrowing we have to manage the set of basic positions and we try to apply all rules at all basic positions in each step. That yields a highly nondeterministic execution principle. On the other hand, pure functional languages deterministically select the position where rules are applied next (innermost position for eager languages and outermost position for lazy languages). An approach to achieve a similar strategy for functional logic languages is the partition of the set of function symbols into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{D}$ of *defined functions*. Constructors are used to build data types, whereas defined functions operate on these data types. Constructor terms (terms from $\mathcal{T}(\mathcal{C},\mathcal{X})$) are always irreducible, whereas defined functions are defined by rules. According to [41], we call a term *innermost* if it has the form $f(t_1,\ldots,t_n)$, where $f \in \mathcal{D}$ and $t_1,\ldots,t_n \in \mathcal{T}(\mathcal{C},\mathcal{X})$. A functional logic program is *constructor-based* if the left-hand side of each rule is an innermost term. In constructor-based programs, rules like

```
append(append(L,M),N) = append(L,append(M,N)).
rev(rev(L)) = L.
```

are excluded. However, the requirement for constructor-based programs is not a real restriction if we are interested in application programs rather than formulae specifying abstract properties of functions. This is also confirmed by the fact that this restriction on rules is also present in pure functional (and pure logic) programming languages.

In constructor-based functional logic programs, we can solve equations by *innermost narrowing* [41], which means that the narrowing position must be an innermost term. Innermost narrowing corresponds to eager evaluation (call-by-value) in functional languages. Since innermost narrowing requires the evaluation of inner terms even if it is not necessary to compute an E-unifier, the computed solutions are sometimes too specific. Therefore, innermost narrowing is incomplete, in general (in the sense of Theorem 2.1), as the following example shows.

Example 2.2. Consider the following rules where `a` is a constructor:

```
f(X) = a.
g(a) = a.
```

Since `f` is a constant function mapping all inputs to `a`, the identity substitution $\{\}$ is a solution of the equation `f(g(X))=a`. However, the only innermost narrowing derivation is

$$\texttt{f(g(X))=a} \;\leadsto_{\{\texttt{X}\mapsto\texttt{a}\}}\; \texttt{f(a)=a} \;\leadsto_{\{\}}\; \texttt{a=a}$$

i.e., innermost narrowing computes only the more specific solution $\{\texttt{X}\mapsto\texttt{a}\}$.

To formulate a completeness result, Fribourg [41] considered *ground substitutions*, i.e., substitutions σ with $\sigma(x)$ ground for all $x \in \mathcal{D}om(\sigma)$. Unfortunately, this is not sufficient for completeness even if the rules are confluent and terminating, because innermost narrowing has problems with partially defined functions. Fribourg presented various additional conditions to ensure completeness. The most important one is: innermost narrowing is complete if all functions are *totally defined*, i.e., the only irreducible ground terms are constructor terms. The next example shows the incompleteness of innermost narrowing in the presence of partial functions.

Example 2.3. Consider the following rules, where `a` and `b` are constructors:

```
f(a,Z) = a.
g(b) = b.
```

If we want to solve the equation `f(X,g(X))=a`, then there is the successful narrowing derivation

$$\texttt{f(X,g(X))=a} \;\; \rightsquigarrow_{\{\texttt{X}\mapsto\texttt{a}\}} \;\; \texttt{a=a}$$

by applying the first rule to the term `f(X,g(X))`, i.e., $\{\texttt{X}\mapsto\texttt{a}\}$ is a solution of the initial equation. However, this derivation is not innermost, and the only innermost narrowing derivation is not successful:

$$\texttt{f(X,g(X))=a} \;\; \rightsquigarrow_{\{\texttt{X}\mapsto\texttt{b}\}} \;\; \texttt{f(b,b)=a}$$

Therefore, innermost narrowing cannot compute the solution.

If E is a finite set of constructor-based unconditional equations so that $\to_E$ is confluent and terminating and all functions are totally defined, then innermost narrowing is complete w.r.t. ground substitutions (i.e., the second proposition of Theorem 2.1 holds if σ' is a ground substitution). The restriction to totally defined functions is not so serious from a practical point of view. In practice, most functions are totally defined, and irreducible innermost terms are usually considered as failure situations. If one wants to deal with partially defined functions, it is also possible to combine the innermost strategy with basic narrowing [66]. The idea of this *innermost basic narrowing* strategy is to skip over calls to partially defined functions by moving these calls to the substitution part. Due to the basic strategy, these calls need not be activated in subsequent computation steps. For a precise description we represent an equational literal in a goal by a skeleton and an environment part [66, 103]: the *skeleton* is an equation composed of terms occurring in the original program, and the *environment* is a substitution which has to be applied to the equation in order to obtain the actual literal. The initial equation E is represented by the pair $\langle E; \{\}\rangle$. If $\langle E; \sigma\rangle$ is a literal (E is the skeleton equation and σ is the environment), then a derivation step in the innermost basic narrowing calculus is one of the following two possibilities. Let p be an innermost position, i.e., $E|_p$ is an innermost term:

Narrowing: Let $l = r$ be a new variant of a rule such that $\sigma(E|_p)$ and l are unifiable with mgu σ'. Then $\langle E[r]_p; \sigma' \circ \sigma\rangle$ is the next literal derived by an *innermost basic narrowing* step.

Innermost reflection: Let σ' be the substitution $\{x \mapsto \sigma(E|_p)\}$, where x is a new variable. Then $\langle E[x]_p; \sigma' \circ \sigma\rangle$ is the next literal derived by an *innermost reflection* step (this corresponds to the elimination of an innermost redex [66] and is called "null narrowing step" in [18]).

Innermost basic narrowing is complete for a confluent and terminating constructor-based set of rules [66]. For instance, a solution of the equation `f(X,g(X))=a` w.r.t. the rules of Example 2.3 will be computed by an innermost reflection step followed by an innermost basic narrowing step:

$$\begin{array}{rcl} \langle \mathtt{f(X,g(X))=a};\{\}\rangle & \leadsto & \langle \mathtt{f(X,Y)=a};\{\mathtt{Y}\mapsto\mathtt{g(X)}\}\rangle \\ & \leadsto & \langle \mathtt{a=a};\{\mathtt{X}\mapsto\mathtt{a},\mathtt{Y}\mapsto\mathtt{g(a)}\}\rangle \end{array}$$

In order to get rid of the various innermost positions in a derivation step, it is possible to select exactly one innermost position for the next narrowing step similarly to the selection function in SLD-resolution (*selection narrowing* [18]). For instance, the operational semantics of the functional logic language ALF [53] is based on innermost basic narrowing with a leftmost selection strategy. This has the advantage that the position in the next derivation step is unique and can be precomputed by the compiler (see Section 3.2.1).

Unfortunately, all these improvements of the simple narrowing method are not better than SLD-resolution for logic programs since Bosco et al. [18] have shown that leftmost innermost basic narrowing is equivalent to SLD-resolution with the leftmost selection rule if we translate functional logic programs into pure logic programs by a flattening transformation (see also Section 3.1). Therefore, we need more sophisticated narrowing methods in order to obtain a real advantage of the integration of functions into logic programming. Fortunately, there are two essential improvements to eliminate unnecessary narrowing derivations. First of all, innermost narrowing strategies have the disadvantage that they continue computations at inner positions of an equation even if the outermost symbols are not unifiable. Therefore, they are too weak in practice.

Example 2.4. Consider the following rules that define the addition on natural numbers which are constructed by `0` and `s`:

```
0    + N = N.
s(M) + N = s(M+N).
```

Then there is the following infinite innermost narrowing derivation of the equation `X+Y=0`:

$$\mathtt{X+Y=0} \leadsto_{\{\mathtt{X}\mapsto\mathtt{s(X1)}\}} \mathtt{s(X1+Y)=0} \leadsto_{\{\mathtt{X1}\mapsto\mathtt{s(X2)}\}} \mathtt{s(s(X2+Y))=0} \leadsto \cdots$$

This derivation can be avoided if we check the outermost constructors of both sides of the derived equation: after the first narrowing step, the equation has the outermost symbols `s` and `0` at the left- and right-hand side, respectively. Since these symbols are different constructors, the equation can never be solved. Hence we could stop the derivation at that point.

The *rejection* rule motivated by this example is generally defined as follows:

Rejection: If the equation $s = t$ should be solved and there is a position p in s and t such that $\mathcal{H}ead(s|_p) \neq \mathcal{H}ead(t|_p)$ and $\mathcal{H}ead(s|_{p'}), \mathcal{H}ead(t|_{p'}) \in \mathcal{C}$ for all prefix positions $p' \preceq p$, then the equation is *rejected*, i.e., the narrowing derivation immediately fails.

Example 2.4 shows that the application of the rejection rule after each narrowing step is a useful optimization to reduce the search space of all narrowing derivations.

The rejection rule terminates a superfluous narrowing derivation if there are different constructors at the same outer position.[8] However, if there are defined function symbols around these constructors, the equation cannot be rejected since the defined functions may evaluate to the same term. Therefore, it is important to evaluate functions as soon as possible in order to apply the rejection rule and to eliminate useless derivations. For instance, consider the rules for addition of Example 2.4 together with the following rules defining a sum function on naturals:

```
sum(0)    = 0.
sum(s(N)) = s(N) + sum(N).
```

Then innermost narrowing applied to the equation `sum(X)=s(0)` has an infinite search space due to the following infinite narrowing derivation:

$$
\begin{array}{lll}
\texttt{sum(X) = s(0)} & \rightsquigarrow_{\texttt{X}\mapsto\texttt{s(N1)}} & \texttt{s(N1)+sum(N1) = s(0)} \\
 & \rightsquigarrow_{\texttt{N1}\mapsto\texttt{s(N2)}} & \texttt{s(s(N2))+(s(N2)+sum(N2)) = s(0)} \\
 & \rightsquigarrow_{\texttt{N2}\mapsto\texttt{s(N3)}} & \cdots
\end{array}
$$

The rejection rule cannot be applied since the head symbol of the left-hand side of the derived equations is always the defined function `+`. The situation can be improved if we evaluate the function call to `+` as soon as possible. That is, if the first argument to `+` is a term headed by the constructor `0` or `s`, we can rewrite this function call using the corresponding rule for `+`. Since rewriting does not bind free variables but replace terms by semantically equal terms, it is a solution preserving transformation. Moreover, repeated application of rewrite steps terminates due to the requirement for a terminating set of rewrite rules. Therefore, it is reasonable to rewrite both sides of the equation to normal form between narrowing steps. Such a narrowing method is called *normalizing narrowing* [40]. For instance, if we rewrite the second derived equation in the previous example to normal form, we can immediately terminate the narrowing derivation:

$$\texttt{s(s(N2))+(s(N2)+sum(N2))=s(0)} \;\to^*\; \texttt{s(s(N2+(s(N2+sum(N2)))))=s(0)}$$

The last equation is rejected since the first subterms of the left- and right-hand side are headed by the different constructors `s` and `0`.

Normalizing narrowing yields more determinism in narrowing derivations. Since the rules are required to be confluent and terminating, normal forms are unique and can be computed by any rewriting strategy. Therefore, rewriting can be implemented as a *deterministic* computation process like reductions in functional languages, whereas narrowing needs a *nondeterministic* implementation as in logic languages, i.e., normalizing narrowing unifies the operational principles of functional and logic programming languages in a natural way [35, 36].

The computation of the normal form before a narrowing step implements a strategy where we compute in a deterministic way as long as possible. This may

[8] [37] describes an extension of the rejection rule where the requirement for different constructors is weakened to incomparable function symbols.

reduce the search space since there are less and shorter normalizing narrowing derivations compared to simple narrowing.

Example 2.5. Consider the following rules for multiplication:

```
X * 0 = 0.
0 * X = 0.
```

Then there are two narrowing derivations of the equation `0*N=0`:

$$\texttt{0*N = 0} \leadsto_{[\texttt{X*0=0},\{\texttt{X}\mapsto\texttt{0},\texttt{N}\mapsto\texttt{0}\}]} \texttt{0 = 0},$$
$$\texttt{0*N = 0} \leadsto_{[\texttt{0*X=0},\{\texttt{X}\mapsto\texttt{N}\}]} \texttt{0 = 0},$$

but there is only one normalizing narrowing derivation since the left-hand side can immediately be rewritten to `0` using the second rule:

$$\texttt{0*N = 0} \;\rightarrow\; \texttt{0 = 0}$$

Thus the preference of deterministic computations can save a lot of time and space (see [55, 78] for benchmarks). If t is a large term, then normalizing narrowing immediately deletes t in the term $\texttt{0*}t$ by rewriting with the first rule, whereas an innermost narrowing strategy would evaluate this term by costly narrowing steps. The deletion of complete subterms has no correspondence in the equivalent logic programs. Hence normalizing narrowing is superior to SLD-resolution. This is due to the fact that rewriting operates on the term structure which is lost if functional logic programs are transformed into pure logic programs by flattening (cf. Section 3.1). The following example [41] shows the difference between normalizing narrowing and SLD-resolution.

Example 2.6. Consider the standard rules for the function `append` (cf. Section 1). Then the equation

```
append(append([0|V],W),Y) = [1|Z]
```

is rewritten to its normal form

```
[0|append(append(V,W),Y)] = [1|Z]
```

using the rules for `append`. This equation is immediately rejected since `0` and `1` are different constructors. The equivalent Prolog program

```
append([],L,L).
append([E|R],L,[E|RL]) :- append(R,L,RL).
?- append([0|V],W,L), append(L,Y,[1|Z])
```

causes an infinite loop for any order of literals and clauses [101].

The idea of normalizing narrowing can also be combined with the previously discussed improvements of simple narrowing. Fribourg has shown that *normalizing* *innermost narrowing* is complete under the same requirements of innermost narrowing [41]. *Normalizing basic narrowing* is discussed in [103, 110], and Hölldobler has shown completeness of *innermost basic narrowing with normalization* [66]. Normalization can be integrated into innermost basic narrowing derivations by applying, first of all, the following rule as long as possible to the literal $\langle E;\sigma\rangle$ consisting of the skeleton equation and the current substitution (note that the nondeterminism in this rule is *don't care*, i.e., it is sufficient to select nondeterministically one alternative and disregard all other possibilities):

Rewriting: Select a nonvariable position p in E and a new variant $l = r$ of a rule such that σ' is a substitution with $\sigma(E|_p) = \sigma'(l)$. Then $\langle E[\sigma'(r)]_p \ ; \ \sigma\rangle$ is the next goal derived by *rewriting*.

Innermost basic narrowing with normalization is superior to SLD-resolution since SLD-resolution is equivalent to innermost basic narrowing [18], but the normalization process may reduce the search space. In fact, it can be shown that any logic program can be transformed into a functional logic program so that the transformed program has at least the same efficiency as the original logic program but is more efficient in many cases [56]. Hence, one of the main motivations of integrating functions into logic programming has been achieved by the innermost basic narrowing strategy with normalization.

The normalization process between narrowing steps reduces the search space and prefers deterministic computations, but it also has one disadvantage. Since the whole goal must be reduced to normal form after each narrowing step, the normalization process may be costly. However, a careful analysis of this process shows that rewrite steps are only applicable at a few positions after a narrowing step: since the goal is in normal form before the narrowing step is applied and the narrowing step changes only small parts of the goal, rewrite steps can be restricted to a small number of positions in the narrowed goal in order to compute a new normal form. In particular, rewrite steps could only be applied to the replaced subterm (instantiated right-hand side of the applied equation) and to function calls in the goal where an argument variable has been instantiated by the narrowing step. Thus it is sufficient to start the normalization process at these positions, proceed from innermost to outermost positions, and immediately stop if no rewrite step can be performed at a position (since the outer part of the goal is already in normal form). A more detailed description of this incremental rewrite algorithm can be found in [57]. A further possibility to avoid rewrite attempts is the restriction of the set of rewrite rules. For instance, SLOG [41] does not use conditional equations (cf. Section 2.4) for normalization in order to avoid a recursive normalization process in conditions. Such a restriction does not influence the soundness and completeness of the operational semantics, but may increase the number of nondeterministic computations steps.

2.3. Lazy Narrowing Strategies

The narrowing strategies discussed so far correspond to eager evaluation strategies in functional programming. However, many modern functional languages like Haskell [67] or Miranda [127] are based on lazy evaluation principles (see [68] for a discussion on the advantages of lazy evaluation). A lazy strategy delays the evaluation of function arguments until their values are definitely needed to compute the result of the function call. Hence, lazy evaluation avoids unnecessary computations and allows us to deal with infinite data structures. For instance, consider the function `first(N,L)`, which computes the first `N` elements of a given list `L`:

```
first(0,L)        = [].
first(s(N),[E|L]) = [E|first(N,L)].
```

If we want to evaluate the function call `first(0,`t`)`, a lazy strategy does not evaluate t since it is not necessary in order to compute the result `[]`. This may avoid a lot of superfluous computations if the evaluation of t is expensive. Now consider the function `from(N)`, which computes the infinite list of naturals starting from `N`:

```
from(N) = [N|from(s(N))].
```

Then lazy evaluation of the function call `first(s(s(0)),from(0))` yields the result `[0,s(0)]`, whereas an eager evaluation of the same function call would not terminate.

In order to apply the idea of lazy evaluation to functional logic languages, there is another class of narrowing strategies that are motivated by this lazy functional programming point of view. A corresponding lazy strategy for narrowing is *outermost narrowing*, where the next narrowing position must be an outermost one. Unfortunately, this strategy is incomplete as the following example shows [38].

Example 2.7. Consider the following rules defining a function `f`:

```
f(0,0)    = 0.
f(s(X),0) = 1.
f(X,s(Y)) = 2.
```

We want to compute solutions of the equation `f(f(I,J),K)=0`. There is the following innermost narrowing derivation:

$$\texttt{f(f(I,J),K)=0} \;\leadsto_{\{\texttt{I}\mapsto\texttt{0},\texttt{J}\mapsto\texttt{0}\}}\; \texttt{f(0,K)=0} \;\leadsto_{\{\texttt{K}\mapsto\texttt{0}\}}\; \texttt{0=0}$$

Therefore, $\{\texttt{I}\mapsto\texttt{0}, \texttt{J}\mapsto\texttt{0}, \texttt{K}\mapsto\texttt{0}\}$ is a solution of the initial equation. Although the rewrite rules are confluent and terminating, there is only one outermost narrowing derivation using the last rule:

$$\texttt{f(f(I,J),K)=0} \;\leadsto_{\{\texttt{K}\mapsto\texttt{s(Y)}\}}\; \texttt{2=0}$$

Thus outermost narrowing cannot compute the above solution.

Echahed [38, 39] and Padawitz [105] have formulated strong restrictions to ensure the completeness of such outermost strategies.[9] In addition to confluence and termination of the rules, complete narrowing strategies must satisfy a *uniformity* condition. Uniformity means that the position selected by the narrowing strategy is a valid narrowing position for all substitutions in normal form applied to it. The outermost strategy is, in general, not uniform since in the last example the top

[9] To be more precise, they have investigated conditions for the completeness of any narrowing strategy. However, their most interesting applications are outermost strategies.

position of the term `f(f(I,J),K)` is not a valid narrowing position if we apply the substitution $\{\mathtt{K}\mapsto\mathtt{0}\}$ to this term. Echahed [38] has proposed a more constructive condition for the completeness of narrowing strategies: all functions must be totally defined and the left-hand sides of all rules must be pairwise *not strictly subunifiable*. The latter condition means that two subterms at the same position of two left-hand sides are not unifiable by a nontrivial mgu (see [38] for details). For instance, `f(0,0)` and `f(s(X),0)` are not strictly subunifiable (the mgu of the second arguments `0` and `0` is trivial), but `f(0,0)` and `f(X,s(Y))` are strictly subunifiable since the mgu of the first arguments is the nontrivial substitution $\{\mathtt{X}\mapsto\mathtt{0}\}$. Since the requirement for not strictly subunifiable left-hand sides is not satisfied by many functional logic programs, [38] also contains a method to transform a program where all functions are totally defined over the constructors into a program satisfying Echahed's conditions.

As mentioned above, lazy evaluation strategies should also support the use of infinite data structures. Since the presence of infinite data structures violates the termination requirement on the rewrite relation, narrowing strategies for terminating programs like outermost narrowing are not sufficient. Hence there are various proposals for *lazy narrowing* strategies which do not require the termination of the rewrite relation [27, 47, 95, 109]. Lazy narrowing differs from outermost narrowing in the fact that lazy narrowing permits narrowing steps at inner positions if the value at this position is needed in order to apply a narrowing rule at an outer position. For instance, if we want to solve the equation `f(f(I,J),K)=0` w.r.t. the rules of Example 2.7, we cannot apply the first rule `f(0,0)=0` at the root position of the left-hand side unless the first argument `f(I,J)` is evaluated to `0`. Since the value of the subterm `f(I,J)` is needed in order to decide the applicability of the first rule, lazy narrowing permits a narrowing step at the inner position. Hence a possible lazy narrowing derivation is

$$\mathtt{f(f(I,J),K)=0} \quad \rightsquigarrow_{\{\mathtt{I}\mapsto\mathtt{0},\mathtt{J}\mapsto\mathtt{0}\}} \quad \mathtt{f(0,K)=0} \quad \rightsquigarrow_{\{\mathtt{K}\mapsto\mathtt{0}\}} \quad \mathtt{0=0} \quad ,$$

which is also an innermost narrowing derivation. However, in general, an inner narrowing step is allowed only if it is demanded and contributes to some later narrowing step at an outer position (see [97] for an exact definition of a lazy narrowing redex).

In narrowing derivations, rules are always applied only in one direction. Hence the confluence of the associated rewrite relation is essential in order to ensure completeness. Since confluence is undecidable and cannot be achieved by completion techniques [77] if the rewrite relation is not terminating, functional logic languages with a lazy operational semantics have the following strong restrictions on the rules in order to ensure completeness [27, 47, 97]:

1. *Constructor-based:* The functional logic program is constructor-based.
2. *Left-linearity:* The functional logic program is left-linear, i.e., no variable appears more than once in the left-hand side of any rule.
3. *Free variables:* If $l=r$ is a rule, then $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$.
4. *Nonambiguity:* If $l_1=r_1$ and $l_2=r_2$ are two different rules, then l_1 and l_2 are not unifiable. Sometimes [97] this condition is relaxed to the requirement: if l_1 and l_2 are unifiable with mgu σ, then $\sigma(r_1)$ and $\sigma(r_2)$ are identical (*weak nonambiguity*).

The nonambiguity condition ensures that normal forms are unique if they exist, i.e., functions are uniquely defined. The strong nonambiguity condition excludes rules like in Example 2.5, whereas the weak nonambiguity condition excludes rules like

```
0    + N = N.
s(0) + N = s(N).
s(M) + N = s(M+N).
```

Due to the presence of nonterminating functions, *completeness results for lazy narrowing* are stated w.r.t. a domain-based declarative semantics of functional logic programs. For instance, consider the function defined by the single rule

```
f(X) = f(X).
```

A lazy narrowing derivation of the equation `f(0)=f(0)` does not terminate, and hence lazy narrowing would be incomplete w.r.t. the standard interpretation of equality. Therefore, some authors exclude defined functions in the right-hand side of goal equations [104] or include a decomposition rule [66], but most completeness results are established w.r.t. *strict equality*, i.e., the equality holds only if both sides are reducible to the same ground constructor term. As a consequence, strict equality does not have the reflexivity property $t = t$ for all terms t. In order to assign a denotation to terms like `f(0)`, the Herbrand universe is augmented with the constant $\perp$ representing undefined values, and then completed into a complete partial order (see [47, 97] for more details).

Since a lazy narrowing derivation requires a narrowing step at an inner position if the value is demanded at that position, it may be the case that values are demanded at different inner positions by different rules. For instance, consider again the rules of Example 2.7 and the given equation `f(f(I,J),K)=0`. If we try to apply the rule `f(0,0)=0` to solve this literal, then the value of the subterm `f(I,J)` is demanded, but it is not demanded if the rule `f(X,s(Y))=2` is applied. Hence a sequential implementation of lazy narrowing has to manage choice points for different narrowing positions as well as choice points for different rules. In order to simplify such an implementation and to avoid backtracking due to different narrowing positions, it is desirable to transform functional logic programs into a corresponding *uniform* program [95] which has the property that all rules are flat (i.e., each argument term of the left-hand side is a variable or a constructor applied to some variables) and pairwise not strictly subunifiable (cf. outermost narrowing). The implementation of the functional logic language BABEL proposed in [95] transforms the rules of Example 2.7 into the uniform program

```
f(X,0)    = g(X).
f(X,s(Y)) = 2.
g(0)      = 0.
g(s(X))   = 1.
```

where `g` is a new function symbol. Now it is clear that the evaluation of a function call of the form `f(`t_1`,`t_2`)` always demands the value of the second argument t_2.

In a sequential implementation of lazy narrowing using backtracking, problems may arise if the evaluation of a demanded argument yields infinitely many solutions. For instance, consider the rules [52]

```
one(0)    = s(0).
one(s(N)) = one(N).
```

that define the constant function `one`. Then there are infinitely many narrowing derivations of `one(X)` to the constructor term `s(0)` with the bindings $\{\mathtt{X}\mapsto\mathtt{0}\}$, $\{\mathtt{X}\mapsto\mathtt{s(0)}\}$, $\{\mathtt{X}\mapsto\mathtt{s(s(0))}\}$, and so on. As a consequence, a sequential lazy narrowing derivation of the equation `one(one(X))=s(0)` does not yield any result since the application of the first rule `one(0)=s(0)` requires the evaluation of the argument term `one(X)` to `0`. Since there are infinitely many evaluations of `one(X)` with result `s(0)`, the second rule is never tried. On the other hand, a sequential innermost narrowing implementation would compute the bindings enumerated above. This problem of a sequential implementation of lazy narrowing could be solved by a *mixed evaluation strategy* which combines lazy narrowing with innermost narrowing for demanded arguments. The value of the argument `one(X)` is demanded in the function call `one(one(X))` for all rules. Therefore, it is evaluated before any rule is selected. After this evaluation, the second rule is applied due to the result `s(0)` (see [52] for more details).

The previous examples show that a lazy narrowing strategy is more difficult to define than a lazy reduction strategy for the evaluation of pure functional programs. This is due to the fact that we have to choose the position as well as the applied equation in a lazy narrowing step. In contrast to reduction, applying different equations at a particular position may yield different solutions. Furthermore, the attempt to apply different equations may require different arguments to be evaluated. As a consequence, a simple lazy narrowing strategy runs the risk of performing unnecessary computations. The following example should explain this subtle point.

Example 2.8. Consider the following rules for comparing and adding natural numbers:

```
0    ≤ N    = true.            0    + N = N.
s(M) ≤ 0    = false.           s(M) + N = s(M+N).
s(M) ≤ s(N) = M ≤ N.
```

We want to solve the equation $\mathtt{X}\leq\mathtt{X+Y}=\mathtt{B}$ by lazy narrowing. A first solution could be computed by applying the first rule for $\leq$ without evaluating the subterm `X+Y`:

$$\mathtt{X}\leq\mathtt{X+Y}=\mathtt{B} \quad \rightsquigarrow_{\{\mathtt{X}\mapsto\mathtt{0}\}} \quad \mathtt{true}=\mathtt{B}$$

Thus $\{\mathtt{X}\mapsto\mathtt{0}, \mathtt{B}\mapsto\mathtt{true}\}$ is a solution of the initial equation. To compute a further solution, we attempt to apply the second or third rule for $\leq$, but in both cases it is necessary to evaluate the subterm `X+Y`. If we choose the rule `0+N=N` for the latter evaluation, we obtain the lazy narrowing derivation

$$\mathtt{X}\leq\mathtt{X+Y}=\mathtt{B} \quad \rightsquigarrow_{\{\mathtt{X}\mapsto\mathtt{0}\}} \quad \mathtt{0}\leq\mathtt{Y}=\mathtt{B} \quad \rightsquigarrow_{\{\}} \quad \mathtt{true}=\mathtt{B} \quad .$$

In the second narrowing step, only the first rule for $\leq$ is applicable. The computed solution $\{\mathtt{X}\mapsto\mathtt{0}, \mathtt{B}\mapsto\mathtt{true}\}$ is identical to the previous one, but the latter derivation contains a superfluous step: to compute this solution, it is not necessary to evaluate the subterm `X+Y`.

To avoid such unnecessary narrowing steps, it is possible to change the order of instantiating variables and applying rules: to evaluate a term of the form `X≤t` w.r.t. the previous example, at first we instantiate the variable `X` either to `0` or to `s(_)`, and then we decide whether it is necessary to evaluate the subterm `t`. The right instantiations of the variables and the choice of the appropriate rules can be determined by the patterns of the left-hand sides of the rules (see [7, 87] for more details). This strategy is called *needed narrowing* and is defined for so-called *inductively sequential* programs [7]. The *optimality of needed narrowing* w.r.t. the length of the narrowing derivations and the independence of computed solutions is shown in [7]. Another approach to avoid unnecessary computations in lazy narrowing derivations by using a sophisticated analysis of demanded arguments is presented in [96].

There are also other lazy evaluation strategies for functional logic programs which are slightly different from lazy narrowing presented so far. You [132] defined *outer narrowing* derivations which have the property that no later narrowing step at an outer position can be performed earlier in the derivation. Some lazy narrowing strategies generate outer narrowing derivations, but not vice versa, since outer narrowing is able to deal with partial functions which are not reducible to constructor terms. However, outer narrowing has the disadvantage that its definition refers to entire narrowing derivation, whereas lazy narrowing steps have a locally oriented definition. Therefore, outer narrowing requires a more complicated implementation.

The implementation of the functional logic language K-LEAF [47] is based on a translation into pure logic programs by flattening nested expressions (cf. Section 3.1). However, the flattened programs are not executed by Prolog's left-to-right resolution strategy, but by the *outermost resolution* strategy. This strategy selects a literal $f(t)=x$ for resolution only if the value of the result variable x is needed. It is related to lazy narrowing in the sense of the correspondence of narrowing derivations and resolution derivations [18].

One motivation for the integration of functions into logic programs is the opportunity to avoid nondeterministic computation steps during program execution in order to reduce the search space. In Section 2.2, we saw that this is possible w.r.t. eager narrowing strategies by the inclusion of a deterministic normalization process between narrowing steps. It is also possible to exploit the deterministic nature of functions in lazy narrowing derivations [88]: If a narrowing step in a lazy derivation is applied to a literal L and no variables from L are bound in this step, then all alternative rules can be discarded for this step due to the nonambiguity requirement of the rules. That is, in a sequential implementation the choice point for the alternative rules can be deleted. This determinism optimization may save space and time since some redundant narrowing steps are omitted. It should be noted that this optimization is a safe replacement of the Prolog "cut" operator because alternative clauses are discarded only if no solutions are lost. Since this is decided at run time, it is also called *dynamic cut* [88].

Since lazy narrowing avoids many unnecessary computations due to its outermost behavior, one could have the impression that the inclusion of a normalization process as in innermost narrowing has no essential influence on the search space, especially if the determinism optimization is carried out. However, normalization can avoid the creation of useless choice points in sequential implementations and reduce the search space for particular classes of programs as the following example shows.

Example 2.9. Consider the rules for the Boolean functions `even` and `or` (`0`, `s`, `false`, and `true` are constructors):

```
false or B     = B.
B     or false = B.                even(0)        = true.
true  or B     = true.             even(s(s(N))) = even(N).
B     or true  = true.
```

If we want to apply a lazy narrowing step to solve the equation

```
even(X) or true = true
```

we have to evaluate the subterm `even(X)` in order to decide the applicability of the first `or` rule. Unfortunately, there are infinitely many narrowing derivations of `even(X)` to the constructor `true` with the bindings {X↦0}, {X↦s(s(0))},.... Therefore, the search space of all possible lazy narrowing derivations is infinite. Moreover, a sequential implementation does not yield any result since the subsequent `or` rules are never tried. However, if we normalize the equation before applying any narrowing step, we would transform the initial equation into `true=true`, which is trivially satisfied. Thus the infinite search space would be reduced to a finite one.

Normalization between lazy narrowing steps is even more useful if inductive consequences are used. An *inductive consequence* is an equation which is valid in the least model of the program. For instance, the equation `N+0=N` is an inductive consequence w.r.t. Example 2.4, but it is not a logical consequence of the rules for addition. It has been shown that the application of inductive consequences is useful in normalizing innermost [41] and normalizing basic [103] narrowing derivations. If inductive consequences are applied, computed solutions are valid in the least model of the program (which is usually the intended model). Proposals to include normalization with inductive consequences into lazy evaluation strategies can be found in [33, 59]. The following example demonstrates the search space reduction using normalization with inductive consequences even for strongly nonambiguous rules:

Example 2.10. Consider the following rules for addition and multiplication on natural numbers:

```
0    + N = N.                      0    * N = 0.
s(M) + N = s(M+N).                 s(M) * N = N+(M*N).
```

Then there is the following lazy narrowing derivation of the equation `X*Y=s(0)`:

$$\begin{array}{lll} \texttt{X*Y=s(0)} & \leadsto_{[\texttt{s(M)*N=N+(M*N)},\{\texttt{X}\mapsto\texttt{s(M)}\}]} & \texttt{Y+(M*Y)=s(0)} \\ & \leadsto_{[\texttt{0+N=N},\{\texttt{Y}\mapsto\texttt{0}\}]} & \texttt{M*0=s(0)} \end{array}$$

The normalization of the last equation `M*0=s(0)` with the inductive consequence `X*0=0` yields the simplified equation `0=s(0)`, which is immediately rejected.[10] Due to the termination of this lazy narrowing derivation, the entire search space for this equation is finite. On the other hand, lazy narrowing without normalization or normalizing innermost narrowing with the same inductive consequence have infinite search spaces.

The last example shows the advantage of lazy narrowing with normalization. However, such strategies have been studied only for terminating rewrite rules [33, 59].

2.4. Conditional Equations

In the previous section, we discussed narrowing strategies to solve equations provided that functions are defined by unconditional equations. However, in many cases it is necessary or useful to define functions by conditional equations as the following definition of the maximum function shows:

```
max(X,Y) = X  :-  X >= Y.
max(X,Y) = Y  :-  X =< Y.
```

The declarative meaning of such conditional equations is inherited from standard logic programming: the equation must hold for each assignment that satisfies the conditions. To use conditional equations for term rewriting, various rewrite relations have been proposed. The most popular relation is based on the requirement that a conditional equation can be applied only if all equations in the condition part have a rewrite proof [75]. Hence, the rewrite relation in the presence of conditional equations is defined as follows. Let $l = r$ `:-` $t_1 = t'_1, \ldots, t_n = t'_n$ be a conditional equation,[11] t a term, and p a position in t. If there are a substitution σ with $t|_p = \sigma(l)$ and terms $u_1, \ldots, u_n$ with $t_i \to^* u_i$ and $t'_i \to^* u_i$ for $i = 1, \ldots, n$, then $t \to t[\sigma(r)]_p$. Note that this definition of conditional rewriting is recursive. Hence, the rewrite relation is undecidable for arbitrary conditional equations [75]. In order to obtain a decidable rewrite relation, it is often required that for all substitutions σ the terms $\sigma(t_i), \sigma(t'_i)$ in the condition must be smaller than the left-hand side $\sigma(l)$ w.r.t. a termination ordering [76] (see [32] for more references). If this is the case for all conditional equations, the program is called *decreasing* (other notions are *fair* or *simplifying* [76]).

If conditional equations are applied in narrowing derivations, it is also necessary to prove the conditions by narrowing rather than rewriting. Kaplan [76] and Hussmann [70] proposed narrowing calculi for conditional equations which have been adopted by many other researchers. The idea is to extend narrowing derivations to lists or multisets of equations and to add the equations in the condition part to the current equation list if the conditional equation is applied in a narrowing step:

[10] The addition of the inductive consequence `X*0=0` to the program rules is not reasonable since this would increase the search space, in general.

[11] For the sake of simplicity we consider only equations in the condition part; the extension to predicates in conditions is straightforward by representing predicates as Boolean functions.

Conditional narrowing: Let G be a given goal (list of equations), p be a position in G with $G|_p \notin \mathcal{X}$, and $l = r$ `:-` C be a new variant of a conditional equation such that $\sigma(G|_p)$ and l are unifiable with mgu σ. Then $\sigma(C, G[r]_p)$ is the next goal derived by *conditional narrowing*, i.e.,

$$G \leadsto_{[p,l=r\,\texttt{:-}C,\sigma]} \sigma(C, G[r]_p)$$

is a conditional narrowing step (C, G denotes the concatenation of the equation lists C and G).

A derivation in the conditional narrowing calculus successfully stops if there exists a mgu for all equations in the derived goal. Consider the standard rules for the function `append` (cf. Section 1) and the following conditional equation defining the function `last`:

```
last(L) = E  :-  append(R,[E]) = L.
```

A solution to the equation `last(L)=2` can be computed by the following derivation in the conditional narrowing calculus:

`last(L)=2` $\leadsto_{\{\}}$ `append(R,[E])=L, E=2` $\leadsto_{\{\texttt{R}\mapsto\texttt{[]}\}}$ `[E]=L, E=2`

The final equation list is unifiable with mgu $\{\texttt{E}\mapsto\texttt{2}, \texttt{L}\mapsto\texttt{[2]}\}$, which is a solution of the initial equation. Instead of computing the mgu for all equations in one step, we could successively eliminate each equation by a reflection step which is more appropriate in efficient implementations of functional logic languages. In this case, a narrowing derivation successfully stops if the list of equations is empty.

Reflection: If $E_1, \ldots, E_n$ is a given list of equations, $E_1 = s{=}t$, and there is a mgu σ for s and t, then $\sigma(E_2, \ldots, E_n)$ is the next goal derived by *reflection*, i.e.,

$$E_1, \ldots, E_n \leadsto_\sigma \sigma(E_2, \ldots, E_n)$$

is a step in the conditional narrowing calculus.

Similarly to the unconditional case, Hussmann [70] claimed soundness and completeness of the conditional narrowing calculus w.r.t. normalized substitutions if the associated term rewrite relation is confluent. However, conditional narrowing is much more complicated than unconditional narrowing since the proof of the equations in the condition part of a conditional equation requires a recursive narrowing process. Actually, subsequent work has shown that the use of conditional equations is more subtle, even if the term rewriting relation is confluent and terminating. One difficult problem is *extra variables in conditions*, i.e., variables in a condition which do not occur in the left-hand side of the conditional equation. Solving conditions with extra variables requires, in some cases, the computation of nonnormalized substitutions. Therefore, Hussmann's results do not hold in full generality.

Example 2.11. Consider the following set of clauses [48]:

```
a = b.
a = c.
b = c  :-  g(X,c) = g(b,X).
```

It is easy to check that the associated rewrite relation is confluent and terminating. The equation `b=c` is valid w.r.t. these clauses (there exists a rewrite proof using the last clause and instantiating the extra variable `X` to `a`), but the only derivation in the conditional narrowing calculus is not successful:

$$\begin{array}{lll} \texttt{b=c} & \rightsquigarrow_{\{\}} & \texttt{g(X,c)=g(b,X), c=c} \\ & \rightsquigarrow_{\{\}} & \texttt{g(Y,c)=g(b,Y), g(X,c)=g(c,X), c=c} \\ & \rightsquigarrow_{\{\}} & \cdots \end{array}$$

The condition `g(X,c)=g(b,X)` could be proved if the variable `X` were instantiated to the reducible term `a`, but the narrowing calculus does not support such instantiations.

The conditional narrowing calculus is complete if it is unnecessary to instantiate extra variables to reducible terms. A simple requirement to achieve this property is to forbid extra variables in conditions [66]. Hence, conditional narrowing is complete w.r.t. normalized substitutions if the set of conditional equations is confluent and does not contain extra variables. Conditional narrowing is complete for arbitrary substitutions if the set of conditional equations is confluent and terminating and does not contain extra variables [76].[12] If one wants to use extra variables in conditions, there are stronger criteria to ensure completeness (e.g., level-confluence [48], decreasing rules [34], or restricting the instantiation of extra variables to irreducible terms [107]), or it may be possible to transform the program into an equivalent one for which conditional narrowing is complete (e.g., Bertling and Ganzinger [11] proposed such a method).

Hölldobler [66] adapted the eager narrowing strategies for constructor-based programs (cf. Section 2.2) to conditional equations without extra variables. In particular, he showed completeness of conditional innermost basic narrowing with normalization in the presence of confluence and termination. However, he missed another problem of conditional equations which has been pointed out by Middeldorp and Hamoen [94]: the termination of the rewrite relation does not imply the termination of the entire rewrite process due to the recursive structure of rewrite proofs in the conditional case.

Example 2.12. Consider the following conditional equations:

```
even(X) = true  :-  odd(X) = false.
odd(X) = false  :-  even(X) = true.
```

The associated rewrite relation is terminating since at most one rewrite step can be performed to evaluate a term headed by `even` or `odd`. However, the conditional rewrite process, which has to check the validity of conditions, would loop due to the recursion in the conditions.

The difference between termination of the rewrite relation and termination of the conditional rewrite process raises no problems for simple narrowing, but it makes *basic conditional narrowing* incomplete as the next example shows.

[12]Kaplan was the first to prove this result for decreasing rules, but it holds also for nondecreasing conditional equations.

Example 2.13. Consider the following conditional equations [94]:

```
f(X) = a  :-  X = b, X = c.
d = b.
d = c.
b = c  :-  f(d) = a.
```

The associated rewrite relation is confluent and terminating. The equation `f(d)=a` is provable in the simple conditional narrowing calculus:

$$\begin{array}{lll} \texttt{f(d)=a} & \leadsto_{\{\}} & \texttt{a=a, d=b, d=c} \\ & \leadsto_{\{\}} & \texttt{a=a, b=b, d=c} \\ & \leadsto_{\{\}} & \texttt{a=a, b=b, c=c} \end{array}$$

However, this derivation is not basic since the term `d`, which belongs to the substitution part after the first narrowing step, is reduced in the second and third narrowing step. In fact, it can be easily shown that there is no successful basic conditional narrowing derivation starting from the initial equation, i.e., *basic conditional narrowing is incomplete* even in the presence of confluence and termination.

In order to ensure completeness for the important *basic* restriction of conditional narrowing, the additional *requirement for decreasing conditional equations* (see above) is sufficient, i.e., in each conditional equation the condition terms must be smaller than the left-hand side w.r.t. some termination ordering. This requirement excludes extra variables in conditions, but it is also used in tools for checking confluence of conditional equations to ensure the decidability of the rewrite relation [45] (although the confluence of decreasing conditional equations is only semidecidable [32]). Nevertheless, extra variables can often be included in decreasing conditional equations by generalizing the latter notion to *quasi-reductive* equations [11] or by restricting the instantiation of extra variables to irreducible terms in the definition of decreasing rules [107]. A good survey on the completeness results of (basic) conditional narrowing w.r.t. different classes of equational logic programs can be found in [94].

The discussion on the completeness problems w.r.t. conditional equations may give the impression that functional logic languages are less powerful than logic languages due to the restrictions on extra variables in conditions and decreasing equations. However, these restrictions are necessary only if one wants to use the full power of functional logic languages by specifying functions by overlapping equations. On the other hand, this case rarely occurs, since functional programmers often write programs with (weakly) nonambiguous equations. This is also required in functional logic languages with a lazy operational semantics (cf. Section 2.3). For instance, the functional logic language BABEL [97] allows extra variables in conditions, i.e., each rule $l = r$ `:-` C must satisfy only the weak variable condition $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ in addition to the usual constructor-based and left-linearity conditions (cf. Section 2.3). Moreover, weak nonambiguity is ensured by one of the following requirements on each pair of equations $l_1 = r_1$ `:-` C_1 and $l_2 = r_2$ `:-` C_2:

1. l_1 and l_2 do not unify.
2. σ is a most general unifier of l_1 and l_2 and $\sigma(r_1) = \sigma(r_2)$.

3. σ is a most general unifier of l_1 and l_2, and $\sigma(C_1)$ and $\sigma(C_2)$ are together unsatisfiable (see [97] for a computable approximation of the latter condition).

Note that there are no further restrictions like decreasing equations. Therefore, it is obvious that pure logic programming is a subset of BABEL since each relational clause L `:-` $L_1, \ldots, L_k$ can be translated into the rule

$$L = \texttt{true} \texttt{ :- } L_1 = \texttt{true}, \ldots, L_k = \texttt{true}$$

by representing predicates as Boolean functions. The latter conditional equations always satisfy condition 2 of BABEL's nonambiguity conditions. In this case, BABEL's operational semantics (lazy narrowing) corresponds to SLD-resolution, but with the additional feature of exploiting determinism by the dynamic cut [88] (cf. Section 2.3).

2.5. Incomplete Evaluation Principles

Although the narrowing principle is a sound and complete execution principle for functional logic programs which is more efficient than resolution for pure logic programs (provided that an appropriate narrowing strategy is chosen), it has one disadvantage in comparison to functional programming: if some argument value of a function call to be evaluated is not known, then a value must be guessed in a nondeterministic way. In order to avoid this nondeterminism in functional computations, several researchers have proposed reduction of functional expressions only if the arguments are sufficiently instantiated [3, 14, 102, 123]. They propose the evaluation of functions only if it is possible in a deterministic way, and all nondeterminism should be represented by predicates. In this case the basic operational semantics is SLD-resolution for predicates [83] with an extended unification procedure such that a function call in a term is evaluated before unifying this term with another term. For instance, consider the following definition of the predicate `square` which relates a number with its square value:

```
square(X, X*X).
```

If a solution of the literal `square(3,Z)` is computed, this literal must be unified with the literal of the `square` definition. Hence, `3` is unified with `X` in the first step. Thus `X` is bound to `3`. Then `Z` is unified with `3*3` (the instantiated second argument). Since the second term is a function call, it is evaluated to `9` and, therefore, `Z` is bound to `9`, which is the solution to this goal.

The important restriction in this modified unification process is that a function call is evaluated only if it does not contain a variable, i.e., if the function call is evaluable to a unique ground value.[13] Therefore, the precise definition of functions is irrelevant. Functions may be defined by rewrite rules [1, 102] or in a completely different language [14, 82]. The only requirement is that a function call must be evaluable if it does not contain variables and the result of the evaluation is a ground constructor term (or perhaps an error message).

[13]Some other languages based on this principle also allow evaluations with variables, but then it must be ensured that at most one possible alternative is applicable.

This evaluation principle seems to be preferable to the narrowing approaches since it preserves the deterministic nature of functions, but it is also obvious that it is an incomplete method. For instance, the goal

```
?- V=3, square(V,9)
```

can be successfully proved w.r.t. the above definition of `square`, but the logically equivalent goal

```
?- square(V,9), V=3
```

leads to a failure: the first literal cannot be proved, since `9` and the unevaluable function call `V*V` are not unifiable (as in Prolog we assume a left-to-right evaluation strategy for goals). In order to avoid these kinds of failures, the evaluation and unification of functions is delayed until the arguments will be instantiated to ground terms. This mechanism is called *residuation* in Le Fun [3] and is also used in a similar form in LIFE [1], NUE-Prolog [102], and Funlog [123]. It has also been used to connect a logic language with an existing functional language (S-unification [13, 14], P-unification [82]).

The residuation principle solves the first literal in the last goal by generating the *residuation* `9=V*V`, which will be proved or disproved as soon as the variable `V` becomes ground. After solving the second literal `V=3`, `V` will be bound to `3` and, therefore, the residuation `9=3*3` can be proved to be true. Hence the entire goal is true.

The delay principle for function evaluation is satisfactory in many cases, but it is still incomplete if functions are used in a logic programming manner as the following example shows.

Example 2.14. [58] Consider the function `append` of Section 1. Following the point of view of logic programming, the last element `E` of a given list `L` can be computed by solving the equation `append(_,[E])=L`. Since the first argument of the left-hand side of this equation will never be instantiated, residuation fails to compute the last element with this equation, whereas narrowing computes the unique value for `E`. Similarly, we specify by the equation `append(LE,[_])=L` a list `LE` which is the result of deleting the last element in the list `L`. Combining these two specifications, we define the reversing of a list by the following clauses:

```
rev([],[]).
rev(L, [E|LR]) :- append(LE,[E])=L, rev(LE,LR).
```

Now consider the literal `rev([a,b,c],R)`. Since the arguments of the calls to the function `append` are never instantiated to ground terms, the residuation principle cannot compute the valid answer `R=[c,b,a]`. In particular, there is an infinite derivation path using the residuation principle and applying the second clause infinitely many times (see Figure 1). The reason for this infinite derivation is the generation of more and more residuations for `append` by a repeated use of the second clause. At a particular point in the derivation these residuations are together unsolvable, but this is not detected by the residuation principle since the equations are simply delayed (hence they are sometimes called *passive constraints*

Current goal:	Current residuation:
`rev([a,b,c],R)`	$\emptyset$
Apply second rule for `rev`*:* `append(LE1,[E1])=[a,b,c], rev(LE1,LR1)`	$\emptyset$
Residuate function call: `rev(LE1,LR1)`	`append(LE1,[E1])=[a,b,c]`
Apply second rule for `rev`*:* `append(LE2,[E2])=LE1, rev(LE2,LR2)`	`append(LE1,[E1]) = [a,b,c]`
Residuate function call: `rev(LE2,LR2)`	`append(LE1,[E1])=[a,b,c],` `append(LE2,[E2])=LE1`
Apply second rule for `rev`*:* `append(LE3,[E3])=LE2, rev(LE3,LR3)`	`append(LE1,[E1])=[a,b,c],` `append(LE2,[E2])=LE1`
...	

FIGURE 1. Infinite derivation with the residuation principle

[4]). On the other hand, a functional logic language based on narrowing can solve this goal and has a finite search space [55]. Therefore, it is not true that avoiding nondeterministic functional computations by the residuation principle yields a better operational behavior in any case.

The last example raises the important question for a decidable class of programs for which the residuation principle is able to compute all answers. Since residuation depends on the instantiation of variables in function calls, an accurate characterization of such programs must analyze the possible run-time bindings of the variables. Program analysis methods tailored to such completeness questions can be found in [19, 21, 58].

2.6. Summary

In Table 1, we summarize the different execution principles for functional logic programs. Although the table enumerates an impressive number of different strategies, it is still incomplete, but it contains, from our point of view, the milestones and most relevant strategies to execute functional logic programs. In the table, we use the following abbreviations for the completeness requirements on the equational clauses:

- C: confluence
- T: termination
- CB: constructor-based
- TD: totally defined functions
- LFN: left-linearity, free variables, and nonambiguity (cf. Section 2.3)

Similarly to pure logic programming, the execution principles are complete if the

TABLE 1. Execution Principles for Functional Logic Languages

Operational Principle	Requirements
Simple narrowing [69, 119]	C, T
Basic narrowing [69]	C, T
Left-to-right basic narrowing [64]	C, T
LSE-narrowing [12]	C, T
Innermost narrowing [41]	C, T, CB, TD; complete w.r.t. ground substitutions
Innermost basic narrowing [66]	C, T, CB
Selection narrowing [18]	C, T, CB
Normalizing narrowing [40]	C, T, CB
Normalizing innermost narrowing [41]	C, T, CB, TD; complete w.r.t. ground substitutions
Normalizing basic narrowing [103, 110]	C, T
Normalizing innermost basic narrowing [66]	C, T, CB
Outermost narrowing [38]	C, T, CB, TD, not strictly subunifiable; complete w.r.t. ground substitutions
Outer narrowing [132]	CB, LFN
Lazy narrowing [27, 95, 109]	CB, LFN; complete w.r.t. strict equality
Needed narrowing [7]	CB, LFN, inductively sequential rules; complete w.r.t. strict equality
Outermost resolution [47]	CB, LFN; complete w.r.t. strict equality
Lazy unification with normalization [33, 59]	(Ground) C, T
Simple conditional narrowing [70, 76]	C, T; see [34, 48, 107] for extra variables
Basic conditional narrowing [66, 94]	C, T, decreasing rules
Innermost conditional narrowing [41]	C, T, CB, TD; complete w.r.t. ground substitutions
Innermost basic conditional narrowing [66]	C, T, decreasing rules
Residuation [3]	Incomplete in general, complete for particular programs [19, 58]

specified requirements are satisfied and a *fair* search strategy like breadth-first is used. If we use an unfair search strategy like depth-first implemented by backtracking (as done in most implementations of functional logic languages), nontermination may occur instead of computable answers.

3. IMPLEMENTING FUNCTIONAL LOGIC LANGUAGES

In this section we review methods used to implement functional logic languages. We restrict this overview to implementations on sequential architectures. Similarly to logic programming, functional logic languages can also be implemented on distributed architectures using concepts like AND- and OR-parallelism (see, for instance, [16, 81, 113]).

The first implementations of functional logic languages were based on interpreters written in high-level languages and thus could not compete with Prolog implementations based on the compilation of Prolog programs into low-level (abstract) machine code. For instance, early implementations of narrowing like the RAP system [46] or NARROWER [111], functional logic languages like LPG [10] or SLOG [41], which are based on normalizing innermost narrowing, and the RITE system [73], a system implementing normalizing narrowing by sharing common parts of different solutions, were implemented in high-level languages like Ada, Pascal, or LISP. However, during recent years more advanced implementations have been developed which achieve the same efficiency as implementations of functional or logic languages. In principal, there are two approaches for the efficient implementation of a functional logic language[14]:

1. Compilation into another high-level language for which efficient implementations exist [128].
2. Compilation into a low-level machine which is efficiently executable on conventional hardware.

In the following, we will discuss both alternatives in more detail.

3.1. Compilation into High-Level Languages

To implement a functional logic language, we need techniques to:

- Deal with logical variables and unification.
- Organize the search for successful derivations (backtracking in the sequential case).
- Apply rules at arbitrary subterms (in the presence of nested expressions).

Prolog offers built-in solutions for the first two requirements. Therefore, it is reasonable to use Prolog as a target language for compiling functional logic programs. Since Prolog applies rules only at the top level (to predicates) and not to subterms of a literal, we have to avoid nested expressions in the target programs. This can be done by *flattening* the program. A conditional equation $l = r$ `:-` $L_1, \ldots, L_n$ is *flattened* as follows:

[14] We do not consider the possibility of constructing special hardware, since this alternative seems unreasonable.

1. If r contains the term $f(t_1, \ldots, t_n)$, where f is a defined function, replace this term in r by a new variable Z and add the new condition $f(t_1, \ldots, t_n) = Z$. Flatten the new clause.
2. If some L_i contains the subterm $f(t_1, \ldots, t_n)$, where f is a defined function and this subterm is not the left-hand side in case of an equation, replace this subterm by a new variable Z and add the new condition $f(t_1, \ldots, t_n) = Z$. Flatten the new clause.

In a similar way, any other goal and relational clause containing defined function symbols is flattened. Such a flattening procedure has been used in this or a slightly modified form in [8, 18, 92, 124, 128] to implement functional logic languages via SLD-resolution.

Example 3.1. The flattened form of the rules for `append` and `member` (cf. Section 1) and the goal literal `member(E,append([1],[2]))` is

```
append([],L)    = L.
append([E|R],L) = [E|Z] :- append(R,L) = Z.
member(E,[E|L]).
member(E,[F|L]) :- member(E,L).
?- append([1],[2]) = L, member(E,L).
```

This program can be executed by SLD-resolution if we add the clause

```
X = X.
```

for unifying both sides of an equation after the evaluation of functions.[15]

If the left-to-right order of the new equations generated during flattening equals the innermost-to-outermost positions of the corresponding subterms, then it can be shown [18] that applying left-to-right SLD-resolution to the flattened program corresponds to leftmost innermost basic narrowing w.r.t. the original functional logic program. Hence, resolution combined with flattening has the same soundness and completeness properties as narrowing.

The idea of flattening functional logic programs can also be applied to implement the residuation principle (cf. Section 2.5). Since residuation delays the evaluation of functions until the arguments are sufficiently instantiated, a Prolog system with coroutining [101] is necessary. In this case, clauses are flattened as described above, and for each function a delay declaration is added which forces the delay of function calls until arguments are instantiated such that at most one clause is applicable to the function call. An implementation with NU-Prolog as the target language is described in [102], and an implementation using delay predicates to connect an existing functional language to a Prolog system with coroutining is described in [74]. It is also possible to implement lazy evaluation strategies by flattening if Prolog's evaluation strategy is slightly modified [15, 23].

The advantage of implementing narrowing by flattening is its simplicity: functional logic programs can be flattened by a simple preprocessor and then executed

[15] If the symbol "=" is predefined to denote syntactic equality as in most Prolog systems, we have to use another operator symbol for equality.

by a Prolog system. Due to the existing sophisticated Prolog implementations, we obtain an efficient implementation of a functional logic language with relatively little effort. However, this method also has an important disadvantage. While functional languages compute values in a deterministic way, our implementation is always nondeterministic since functions are mapped into predicates. For instance, if the rules for multiplication of Example 2.5 are given, then a functional language would deterministically evaluate the term `0*0` to `0` using one of the rules. On the other hand, a Prolog system would apply both rules, i.e., it computes in a nondeterministic way. Inserting cuts or delay declarations in a Prolog program may improve the efficiency, but it reduces the applicability of the logic program, in general. Moreover, cuts or delay declarations cannot avoid simple infinite loops as the following example demonstrates.

Example 3.2. Consider the rules for the function `append` (cf. Section 1) and the literal

```
append(append(X,Y),Z) = []
```

The solution $\{$`X`$\mapsto$`[]`, `Y`$\mapsto$`[]`, `Z`$\mapsto$`[]`$\}$ is computed in two narrowing steps using the first rule for `append`. However, if the second `append` rule is applied to the inner subterm, `X` is instantiated to `[E|R]` and

```
append([E|append(R,Y)],Z) = []
```

is the derived equation. A normalizing narrowing strategy transforms the last equation into its normal form `[E|append(append(R,Y),Z)]=[]`, which is immediately rejected. Hence, an infinite derivation does not occur. On the other hand, the execution of the flattened goal

```
?- append(X,Y) = T, append(T,Z) = [].
```

w.r.t. the flattened program (cf. Example 3.1) generates the new goal

```
?- append(R,Y) = T1, append([E|T1],Z) = [].
```

if the second clause for `append` is applied to the first literal. Hence, Prolog runs into an infinite loop, which could be avoided only if the second literal is proved before the first one.

Hence, a logic language with an operational semantics that prefers the evaluation of deterministic literals (i.e., literals having at most one matching clause) would avoid the infinite loop in the last example. The Andorra computation model [62] or Prolog with Simplification [42] have this property. Therefore, the flattening technique yields better results if the target of the transformation is a logic programming system with an extended computation model. If an efficient implementation of such an extended computation model is not available, it is also possible to simulate it with a standard Prolog system by a simple meta-interpreter. Cheong and Fribourg [24] developed a method to reduce the overhead of the meta-interpreter by using partial evaluation techniques.

Nevertheless, flattening of functional logic programs into the Andorra computation model or into Prolog with Simplification is less powerful than normalization due to the following reasons:

1. Normalization can delete subterms if there are rules with variables in the left-hand side that do not occur in the right-hand side (e.g., `0*X=0`). The application of such rules during normalization would correspond to the deletion of literals in the flattened program.
2. Normalization evaluates terms even if more than one rule is applicable. For instance, the term `0*0` is normalized to `0` w.r.t. the multiplication rules of Example 2.5, which are not deterministic in the sense of [42, 62].

Example 3.3. Consider the following program for computing the maximum of two natural numbers:

```
max(X,Y) = Y  :-  le(X,Y).
max(X,Y) = X  :-  le(Y,X).
le(0,N).
le(s(M),s(N)) :- le(M,N).
```

If we compute the maximum of two identical numbers, e.g., we want to solve the equation `max(s(s(0)),s(s(0)))=Z`, then the solution {`Z`↦`s(s(0))`} would be computed in a unique way in a functional language or by normalizing narrowing. However, applying SLD-resolution, Prolog with Simplification, or the Andorra computation model to this program (it is already in flat form) yields the same solution twice because both `max` rules are applicable to this equation.

The last examples have shown the limitations of the flattening approach: it is not ensured that functional expressions are reduced in a purely deterministic way if all arguments of a function are ground values. This important property of functional languages is not preserved since the information about functional dependencies is lost by flattening. Moreover, flattening restricts the chance to detect deterministic computations by the dynamic cut (cf. Section 2.3) which is relevant especially in the presence of conditional equations [88]. Therefore, several new implementation techniques have been developed for functional logic languages. The characteristic of these new approaches is the use of low-level abstract machines and the compilation of functional logic programs into the code of these machines. In the next section we sketch the basic ideas of these abstract machines.

3.2. Compilation into Abstract Machines

The use of "abstract machines" is a well-known technique for the efficient implementation of functional and logic languages on standard hardware. On the one hand, abstract machines have a low-level architecture so that they can be simply compiled or efficiently emulated on standard hardware. On the other hand, the architecture of abstract machines is tailored to the execution of a particular high-level language, and this simplifies the compilation process in comparison to a direct compilation into real machine code. There are a lot of proposals for abstract machines to execute pure functional or logic languages. Since functional logic languages are extensions of pure functional or logic languages, it is a natural idea to extend one of the existing abstract machines to execute functional logic programs. In the following, we will see that this has been successfully translated into action.

3.2.1. Extending Abstract Machines for Logic Languages. Most Prolog implementations are based on the "*Warren Abstract Machine*" (WAM) [2, 130] or on a refinement of it. The WAM supports logical variables, unification, application of clauses, and backtracking. This is also necessary in any implementation of a functional logic language and thus there are several proposals to extend the WAM in order to deal with narrowing and functional computations. As discussed in the previous section, one possible implementation of narrowing is flattening and applying SLD-resolution. If a lazy evaluation principle as in K-LEAF [47] is implemented, it is necessary to apply a modified resolution strategy where a literal is activated only if it is needed.

Example 3.4. [15] Consider the following functional logic program which is already in flat form:

```
p(1,2) :- q(0).
q(0).
f(1) = 1.
f(2) = 1.
```

In order to prove the literal `p(f(X),X)`, it is transformed into the flat form:

```
?- p(V,X), f(X) = V.
```

The new variable `V`, which is introduced during flattening, is also called *produced variable*. The outermost resolution strategy selects an equational literal only if the value of its produced variable is required. Hence, the literal `p(V,X)` is selected in the first resolution step. Applying the first clause yields the bindings $\sigma = \{\mathtt{V}\mapsto\mathtt{1}, \mathtt{X}\mapsto\mathtt{2}\}$ and the derived goal

```
?- q(0), f(2) = 1.
```

Since the produced variable `V` has been instantiated, the literal $\sigma(\mathtt{f(X)=V})$ is selected in the next step (instead of `q(0)`). The application of the fourth clause to this literal generates the new goal `q(0)`, which is immediately proved by the second clause.

To implement this selection strategy, it is necessary to link a produced variable v with the corresponding equational literal $f(t)=v$. This is implemented in the K-WAM [15], an extension of the WAM to implement the outermost resolution strategy of K-LEAF. In the K-WAM each produced variable v contains a pointer to its equational literal $f(t)=v$. If v is instantiated to a nonvariable term during unification, the corresponding literal $f(t)=v$ is added to a global list (*force list*). The literals in the force list are proved immediately after the unification of the head literal. Therefore, the only changes to the WAM are a new representation for produced variables, a modification in the unification procedure to deal with produced variables, and a switch to the force list after the head unification. All other aspects are fully inherited from the WAM.

It is interesting to note that Cheong [23] showed that the outermost strategy can also be implemented in Prolog without any modification of the WAM. For this purpose, it is necessary to compile K-LEAF programs into Prolog programs by changing the representation of terms (in particular, produced variables) and adding clauses for the evaluation of these new terms (see [23] for details).

We have seen in Section 3.1 that the flattening approach is problematic if a deterministic computation principle like normalization is included. Fortunately, it is relatively easy to extend the WAM to a direct inclusion of narrowing and normalization. To describe the necessary extensions, we recall the main data areas of the WAM:

Code area: Contains the WAM code of the compiled program.
Local stack: Contains environments (for clause invocations) and choice points.
Heap: Contains term structures.
Trail: Contains references to variables that have been bound during unification. These variables must be unbound in case of backtracking.

The WAM has primitive instructions for unification, clause invocation, and backtracking. Each clause is translated into a sequence of unification instructions for the clause head, followed by a sequence of calls to the predicates in the clause body. Thus goals are represented by instruction sequences and not by proper data structures. On the other hand, narrowing and normalization manipulates the term structure: subterms are replaced by right-hand sides of rules. Hence, a WAM-based implementation of narrowing must support such term manipulations. One possible approach is implemented in the A-WAM [53, 55], an extension of the WAM to implement the functional logic language ALF. The operational semantics of ALF is based on SLD-resolution for predicates combined with normalizing innermost basic narrowing for functions which can be specified by conditional equations [53]. To support term manipulation, the A-WAM has instructions to replace terms in the heap by new terms. These replacements are also stored on the trail in order to undo them in case of backtracking. Using these new instructions, function rules can be compiled similarly to clauses for predicates.

The main problem for the efficient implementation of an innermost narrowing strategy is the access to the current leftmost innermost subterm in the next narrowing step. A simple solution would be a dynamic search through the term. Obviously, this is too slow. Fortunately, the compiler can determine this position since we use a *basic* narrowing strategy. Recall that in basic narrowing, all narrowing positions must belong to the initial goal or to the right-hand side of some rule, but not to the substitution part. Consequently, the compiler can compute the basic positions in leftmost innermost order. For instance, if `f(g(X),h(Y))` is the right-hand side of some rule, then the basic positions are the ones belonging to the subterms `g(X)`, `h(Y)`, `f(g(X),h(Y))` (in leftmost innermost order). In addition to the WAM, the A-WAM has an *occurrence stack*, where the basic positions of the current literal are stored in leftmost innermost order, i.e., the top element of this stack is always the leftmost innermost position of the current literal. The compiler generates instructions for the manipulation of the occurrence stack. For instance, if a rule with no defined function symbol on the right-hand side is applied, like `0+X=X`, then the compiler generates a `pop` instruction for the occurrence stack in the translated code of this rule. Similarly, `push` instructions are generated for right-hand sides containing defined function symbols.[16] The `push` and `pop` instructions are generated along

[16]Note that only occurrences of defined function symbols are stored on the occurrence stack since the program is constructor-based and there are no rules for constructors.

with the usual term building instructions of the WAM and cause no real overhead. The advantage of this approach is the access of the next narrowing position in constant time.

Altogether, a rule $l = r$ is translated into the following scheme of A-WAM instructions:

⟨unify the left-hand side l with the current subterm⟩
⟨replace the current subterm by the right-hand side r⟩
⟨update the occurrence stack (delete or add occurrences)⟩
⟨proceed with normalization/narrowing at new innermost occurrence⟩

The rules used for normalization are translated in a similar way, but the unification of the left-hand side is replaced by matching (unification without instantiating goal variables). Moreover, choice points are not generated during normalization due to its deterministic nature.

The normalization process before each narrowing step causes a problem since it tries to simplify the current term by applying normalization rules from innermost to outermost positions in the term. If no normalization rule can be applied to a subterm, the next innermost position is tried, i.e., an element is popped from the occurrence stack. This is necessary as the following example shows: If the only rules for the functions `f` and `g` are

```
f(Z) = 0.
g(0) = 0.
```

then the term `g(X)` cannot be rewritten (only narrowing could be applied), but the term `f(c(g(X)))` can be simplified to `0`.

Hence the normalization process pops all elements from the occurrence stack and, therefore, the stack is empty when normalization is finished and a narrowing rule should be applied. Now, in order to avoid a dynamic search for the appropriate innermost occurrence, the A-WAM has a second stack (*copy occurrence stack*) for storing the deleted occurrences. This stack contains all occurrences if normalization is finished and the original occurrence stack is empty. Thus the occurrence stack can be reinstalled by a simple block-copy operation.

The advantage of the A-WAM is its efficiency in comparison to the original WAM: a dynamic search inside the term structure can be avoided and the code of the compiled functional logic programs is very similar to the WAM code of the corresponding logic programs obtained by flattening (see [55] for examples). The overhead of the occurrence stack manipulation is small (around 5%), and the execution of pure functional programs is comparable with implementations of functional languages due to the deterministic normalization process (see [55] for benchmarks). In Sections 2.2 and 3.1 we saw that a normalizing narrowing strategy is more efficient than SLD-resolution for the flattened programs since the deterministic normalization process can reduce the search space. These theoretical considerations can be proved, in practice, if an efficient implementation of normalizing narrowing like the A-WAM is available. For instance, in the "permutation sort" program, a list is sorted by enumerating all permutations and checking whether they are sorted. The relational version of the program ([122], p. 55) enumerates *all* permutations, whereas in the functional version not all permutations are enumerated since the

TABLE 2. Normalizing Narrowing vs. SLD-Resolution: Permutation Sort

Program:	$n = 6$	$n = 8$	$n = 10$
Pure logic program ([122], p. 55)	0.65	37.92	3569.50
Functional logic program ([41], p. 182)	0.27	1.43	7.43

generation of a permutation is stopped (by normalizing the goal to "fail") if two consecutive elements `X` and `Y` have the wrong ordering `Y<X` ([41], p. 182). As a consequence, the A-WAM yields the execution times in seconds on a Sun4 to sort the list `[`n`,...,2,1]` for different values of n as shown in Table 2 [55]. In such typical "generate-and-test" programs, the normalization process performs the test part and the narrowing steps perform the generate part of the program. Due to the strategy of normalizing narrowing, the test part is merged into the generate part, which yields a more efficient control strategy than SLD-resolution for equivalent logic programs. This is achieved in a purely clean and declarative way without any user annotations to control the proof strategy. More details on this control aspect can be found in [56].

Although the permutation sort example is only a toy program, larger applications have been implemented in ALF in order to test the suitability of normalizing narrowing as an operational semantics for functional logic languages. It turns out that the normalization process between narrowing steps is not an overhead even if it does not reduce the search space: most computations are performed by normalization, and narrowing steps are applied only at some few positions. Hence, rewrite steps are the rule and narrowing steps are the exception, in practice. This is similar to the experience that in practical logic programming most computations are functional. Therefore, functional logic languages can help to implement these functional subcomputations in a more efficient way.

Mück [98] has also developed a technique to compile narrowing into a WAM-like architecture. Although he has not included normalization and an efficient management of occurrences in his framework, the proposed method can be used to derive some of the instructions of an abstract narrowing machine in a systematic way: he has shown how functional logic programs can be translated into low-level instructions using partial evaluation techniques.

3.2.2. Extending Abstract Machines for Functional Languages. Another alternative to implement functional logic languages is the extension of abstract machines used for the implementation of pure functional languages. If the functional logic language is based on some kind of narrowing, the necessary extensions are the implementation of logical variables, unification, and backtracking.

Loogen [86] has extended a reduction machine to implement a subset of the functional logic language BABEL [97]. *Reduction machines* are designed to compile functional languages. Their main components are a stack of environments (local variables and actual arguments) for function calls and a heap or graph structure to store data terms. The evaluation process is controlled by the stack, i.e., the stack contains the environments for function calls in innermost order if an eager evaluation strategy is implemented. In order to implement an innermost narrowing strategy, Loogen has extended such a reduction machine by variable nodes in the graph to represent logical variables and by choice points in the environment stack and a trail to organize backtracking. The overall structure of this narrowing machine is similar to the WAM, but with an explicit data stack to pass arguments and results of function calls. This data stack allows a better management of choice points. Since normalization is not included, defined function symbols need not be represented in the heap. They are directly translated into `call` instructions of the reduction machine. For instance, an expression like `f(g(X))` is translated into the instructions

```
load X       % load contents of X on the data stack
call g/1     % call code of function g with one argument
call f/1     % call code of function f with one argument
```

(see [86] for a formal specification of the machine and the compilation process). The resulting code is very similar to the WAM code obtained by flattening the functional logic program (as described in Section 3.1) and translating the logic program as usual. However, the proposed narrowing machine has an important optimization in comparison to the WAM: if the application of a rule does not bind any goal variables, then the choice point corresponding to this rule is discarded so that alternative rules are not tried (*dynamic cut*, cf. Section 2.3). This is implemented by a `pop` instruction which checks the variable bindings after the unification of the left-hand side of the rule [88]. Due to this optimization pure functional computations without logical variables are performed with the same deterministic behavior as in pure functional languages. However, there remains a small overhead since choice points are generated and then immediately deleted. As discussed in Section 2.2, pure innermost narrowing is too weak for many applications due to nonterminating derivations. Therefore, [86] also outlines an implementation of lazy narrowing by introducing suspension nodes in the heap representing unevaluated function calls.

Chakravarty and Lock [22] have proposed an abstract machine for lazy narrowing which is an extension of a stack-based reduction machine used to implement functional languages with a lazy evaluation principle. The instruction code of their JUMP machine is a block-structured intermediate language so that classical code generation techniques can be applied. The main data areas of their machine are a stack for activations records of functions and choice points, a heap to store environments and closures representing logical variables and unevaluated function calls, and a trail to store bindings which must be reset in case of backtracking. Constructor terms, logical variables, and suspended function calls are treated in a similar way: their current value is obtained by jumping to their code address, which eliminates overhead of tag testing as in the WAM. Another difference is the choice point organization. While the WAM creates a choice point if there is more than one rule applicable to a predicate, the JUMP machine creates a choice point for each logical variable during unification of a function call with the left-hand side of a rule. This requires a transformation of the given rules into a set of nonsubunifiable rules (cf. Section 2.3). The advantage of this choice point organization is that ground function calls are automatically computed in a deterministic way. On the other hand, several choice points are created for a function call with several unbound variables. The JUMP machine can also be used to implement innermost narrowing by using another compilation scheme. Lock [85] has proposed a mixed implementation scheme where argument evaluation is implemented by lazy narrowing or innermost narrowing depending on some kind of strictness information for the arguments of a function.

The main component of the narrowing machines described so far is a stack which contains local data for each function call and choice points. The structure of this stack controls the execution order. The global nature of this stack makes it difficult to base a parallel implementation on it. In functional programming it has been shown that a decentralized graph structure is more appropriate for parallel implementations. Hence Kuchen et al. [79] have proposed a graph-based abstract machine for an innermost narrowing implementation of the language BABEL [97]. The main component of their BAM machine is a graph containing task nodes for each evaluation of a function call. Each task node contains local management information like local code address, return address etc., the list of arguments and local variables of the function call, and a local trail to organize backtracking. The intention of this machine is to support AND-parallelism [81]; hence, backtracking is included. Further elements of the graph are special nodes to represent logical variables, constructors (data terms), and partial function applications (BABEL supports curried functions where some arguments are omitted in a function call). The instruction set of this machine consists of local instructions like loading local registers, unifying variables or constructors, creating new graph nodes, etc., and process instructions to activate and terminate tasks. In a sequential implementation of this machine there is always one active task identified by a global pointer. A parallel extension of this machine to support AND-parallelism on a shared memory multiprocessor is described in [81]. [95] describes an extension of the sequential BAM to support a lazy narrowing strategy.

Wolz [131] proposed another graph-based abstract machine for the implementation of lazy narrowing. The machine LANAM is an extension of an abstract machine for lazy term rewriting and has also many similarities to the WAM. The main motivation for the graph-based architecture is the sharing of data structures and unevaluated expressions in order to avoid multiple evaluations. The implemented lazy narrowing strategy requires neither constructor-based programs nor nonambiguous rules as other lazy narrowing strategies (cf. Section 2.3). All rules for a function symbol are compiled into a decision tree representing the applicable rules. Initially, all function symbols with defining rules are potentially evaluable. If a function cannot be evaluated since no rule is applicable, it is marked as a constructor that cannot be further evaluated. To apply a rule to an expression, the arguments of the expression corresponding to the nonvariable arguments of the rule are evaluated to their head normal form (a term with a constructor at the top). This process continues on subterms of the arguments as long as the rule has nested argument patterns. Due to this evaluation strategy, a transformation of the source program into a uniform program by flattening the left-hand sides of the rules (cf. Section 2.3) is not necessary. An early detection of nonapplicable rules is supported by a particular strategy to select arguments for evaluation. However, completeness results for the overall lazy narrowing strategy are not provided.

Most of the various abstract narrowing machines discussed above are highly optimized to obtain an efficient implementation of the chosen narrowing strategy. As a result, the correctness of these implementations is hard to prove. To achieve a verifiable implementation of a functional logic language, Mück [99] has proposed the CAMEL narrowing machine which is based on the categorical abstract machine (CAM) [25], a relatively simple but efficient abstract machine for the execution of functional languages. The CAM has three data areas (code area, value stack, value area) and a small set of plain instructions. Mück has slightly extended the CAM by a heap to store logical variables, choice points in the value stack to handle backtracking, and some new instructions for unification and backtracking. These extensions enable a simple scheme to compile functional logic programs based on innermost narrowing into CAMEL instructions. In order to achieve the efficiency of sophisticated narrowing implementations, it is necessary to optimize the CAMEL by several refinement steps. Although this approach is not yet implemented, it may be useful to verify and simplify existing narrowing implementations.

3.3. Summary

The most important techniques proposed for the efficient implementation of functional logic languages are summarized in Table 3. These implementations have shown that it is possible to implement functional logic languages in an efficient way provided that:

- An appropriate operational semantics is chosen.
- Implementation principles known from pure functional and logic programming languages are adapted.

If these two items are carefully selected, functional logic languages have the same efficiency as pure functional or pure logic languages. This is due to the fact that the implementations are similar to the pure languages if the additional features of the amalgamated language are not used. For instance, the A-WAM extends the WAM by several new instructions and a new data structure (occurrence stack). These new instructions and the data structure are used only if defined functions are present in the program. Thus the compiled code is identical to the WAM code as described in [2, 130] for pure logic programs without defined functions. As an example from the other extreme, consider the JUMP machine which is an extension of an abstract machine used for the efficient implementation of functional languages (spineless tagless G-machine). If logical variables do not occur during run time, no choice point will be generated and the behavior is the same as for a pure functional program. However, if features from both programming paradigms are used in the proposed implementations of functional logic languages, the advantage of the amalgamated approach shows up. The knowledge about functional dependencies is used in the implementation to reduce the nondeterminism, e.g., by the inclusion of a deterministic normalization process or by the inclusion of a dynamic cut.

Although there are many differences between the various abstract machines due to the implemented narrowing strategies and the different starting points, it is interesting to see that there is a common kernel in the proposed abstract machines which is also present in the WAM: the code area for the program, the heap to store logical variables and evaluated expressions, a (local) stack to store environments and choice points, and a trail to store variable bindings and other changes in the term structure that must be reset in case of backtracking. Due to the similarity to the WAM and other "classical" abstract machines, there are many possibilities to improve the current implementations of functional logic languages by applying optimization techniques for Prolog implementations (e.g., [29, 65, 126]). However, more advanced compilation techniques which depend on a global analysis of the

TABLE 3. Efficient Implementations of Functional Logic Languages

Implementation	Implementation principle	Operational semantics
[18, 128]	Flattening and resolution	Innermost basic narrowing
Flang [91, 92]	Flattening and resolution	Innermost narrowing
NUE-Prolog [102]	Flattening and resolution with coroutining	Residuation
GAPLog [74]	Flattening and resolution with coroutining	Residuation (S-unification)
Prolog with Simplification [24]	Partial evaluation and resolution	Resolution and simplification
K-WAM [15]	WAM-extension	Outermost resolution
A-WAM [53, 55]	WAM-extension	Innermost basic narrowing with normalization
SBAM [52, 86]	Extended stack-based reduction machine	Innermost narrowing and lazy narrowing
JUMP [22, 85]	Extended stack-based reduction machine	Lazy narrowing and innermost narrowing
BAM [79]	Extended graph-based reduction machine	Innermost narrowing
LBAM [95]	Extended graph-based reduction machine	Lazy narrowing
PBAM [81]	Extended graph-based reduction machine	AND-parallel innermost narrowing
LANAM [131]	Extension of a lazy term rewriting machine	Lazy narrowing
CAMEL [99]	CAM-extension	Innermost narrowing

program [93, 125, 129] require the development of new program analysis methods for functional logic programs [60].

4. CONCLUSIONS

The research on functional logic languages during the last decade has shown that functional and logic languages can be amalgamated without loosing the efficiency of current implementations of functional or logic languages. The amalgamated languages have more expressive power in comparison to functional languages and a better operational behavior in comparison to logic languages. Therefore, the original motivation for the research in this area has been satisfied. This goal has been achieved in two basic steps:

1. *The execution principles for functional logic languages have been refined.* The most important operational principle is narrowing, a combination of resolution from logic programming and term reduction from functional programming. Since narrowing is highly inefficient in its simplest form, much work has been carried out to restrict the admissible narrowing derivations without loosing completeness. The development of these refined strategies was the precondition for the efficient implementation of functional logic languages.
2. *Implementation techniques known from functional and logic languages have been extended to implement functional logic languages.* Due to the refined operational principles, only slight extensions are necessary. The overhead introduced by these extensions is small or disappears if the new features (functions in case of logic programs and logical variables in case of functional programs) are not used. Moreover, the use of functions yields a more efficient behavior in comparison to pure logic programs.

In this survey we have tried to sketch and to relate the various developments of the last decade. Nevertheless, we could not cover all aspects on the integration of functional and logic languages. There are many further topics which have been partly addressed in the past and which are interesting for future work. These include:

- Better implementation by using program analysis techniques [5, 6, 20, 30, 58, 60].
- Distributed implementations [16, 81, 113].
- Development of programming environments like debugging tools [61].
- Integration of other features like types [1, 54, 114, 121], constraints [1, 28, 80, 84, 89, 90, 92], or higher-order functions [17, 51, 54, 100, 108, 120].

The author is grateful to Herbert Kuchen, Rita Loogen, and an anonymous referee for their detailed comments on a previous version of this paper. Sergio Antoy, Bernd Bütow, Rachid Echahed, Hendrik Lock, Andy Mück, Peter Padawitz, and Christian Prehofer have given further valuable comments to improve the paper.

REFERENCES

1. H. Aït-Kaci. An Overview of LIFE. In J.W. Schmidt and A.A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pp. 42–58. Springer LNCS 504, 1990.
2. H. Aït-Kaci. *Warren's Abstract Machine*. MIT Press, 1991.
3. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and Functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 17–23, San Francisco, 1987.
4. H. Aït-Kaci and A. Podelski. Functions as Passive Constraints in LIFE. Research Report 13, DEC Paris Research Laboratory, 1991.
5. M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Inconsistency for Incremental Equational Logic Programming. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 443–457. Springer LNCS 631, 1992.
6. M. Alpuente, M. Falaschi, M.J. Ramis, and G. Vidal. Narrowing Approximations as an Optimization for Equational Logic Programs. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 391–409. Springer LNCS 714, 1993.
7. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, Portland, 1994.
8. R. Barbuti, M. Bellia, G. Levi, and M. Martelli. LEAF: a Language which Integrates Logic, Equations and Functions. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pp. 201–238. Prentice Hall, 1986.
9. M. Bellia and G. Levi. The Relation between Logic and Functional Languages: A Survey. *Journal of Logic Programming (3)*, pp. 217–236, 1986.
10. D. Bert and R. Echahed. Design and Implementation of a Generic, Logic and Functional Programming Language. In *Proc. European Symposium on Programming*, pp. 119–132. Springer LNCS 213, 1986.
11. H. Bertling and H. Ganzinger. Completion-Time Optimization of Rewrite-Time Goal Solving. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 45–58. Springer LNCS 355, 1989.
12. A. Bockmayr, S. Krischer, and A. Werner. An Optimal Narrowing Strategy for General Canonical Systems. In *Proc. of the 3rd Intern. Workshop on Conditional Term Rewriting Systems*, pp. 483–497. Springer LNCS 656, 1992.
13. S. Bonnier. Unification in Incompletely Specified Theories: A Case Study. In *Mathematical Foundations of Computer Science*, pp. 84–92. Springer LNCS 520, 1991.
14. S. Bonnier and J. Maluszynski. Towards a Clean Amalgamation of Logic Programs with External Procedures. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 311–326. MIT Press, 1988.
15. P.G. Bosco, C. Cecchi, and C. Moiso. An extension of WAM for K-LEAF: a WAM-based compilation of conditional narrowing. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 318–333. MIT Press, 1989.
16. P.G. Bosco, C. Cecchi, C. Moiso, M. Porta, and G. Sofi. Logic and Functional Programming on Distributed Memory Architectures. In *Proc. Seventh International Conference on Logic Programming*, pp. 325–339. MIT Press, 1990.
17. P.G. Bosco and E. Giovannetti. IDEAL: An Ideal Deductive Applicative Language. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 89–94, Salt Lake City, 1986.
18. P.G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-Resolution. *The-*

oretical Computer Science 59, pp. 3–23, 1988.

19. J. Boye. S-SLD-resolution – An Operational Semantics for Logic Programs with External Procedures. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 383–393. Springer LNCS 528, 1991.
20. J. Boye. Avoiding Dynamic Delays in Functional Logic Programs. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 12–27. Springer LNCS 714, 1993.
21. J. Boye, J. Paakki, and J. Maluszynski. Synthesis of directionality information for functional logic programs. In *Proc. Third International Workshop on Static Analysis*, pp. 165–177. Springer LNCS 724, 1993.
22. M.M.T. Chakravarty and H.C.R. Lock. The Implementation of Lazy Narrowing. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 123–134. Springer LNCS 528, 1991.
23. P.H. Cheong. Compiling lazy narrowing into Prolog. Technical Report 25, LIENS, Paris, 1990. To appear in Journal of New Generation Computing.
24. P.H. Cheong and L. Fribourg. Efficient Integration of Simplification into Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 359–370. Springer LNCS 528, 1991.
25. G. Cousineau. The Categorical Abstract Machine. In G. Huet, editor, *Logical Foundations of Functional Programming*, pp. 25–45. Addison Wesley, 1990.
26. J. Darlington, A.J. Field, and H. Pull. The unification of functional and logic languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pp. 37–70. Prentice Hall, 1986.
27. J. Darlington and Y. Guo. Narrowing and unification in functional programming - an evaluation mechanism for absolute set abstraction. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 92–108. Springer LNCS 355, 1989.
28. J. Darlington, Y. Guo, and H. Pull. A New Perspective on Integrating Functional and Logic Languages. In *Proc. Fifth Generation Computer Systems*, pp. 682–693, 1992.
29. S.K. Debray. Register Allocation in a Prolog Machine. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 267–275, Salt Lake City, 1986.
30. S.K. Debray and D.S. Warren. Functional Computations in Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, pp. 451–481, 1989.
31. D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, 1986.
32. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pp. 243–320. Elsevier, 1990.
33. N. Dershowitz, S. Mitra, and G. Sivakumar. Equation Solving in Conditional AC-Theories. In *Proc. of the 2nd International Conference on Algebraic and Logic Programming*, pp. 283–297. Springer LNCS 463, 1990.
34. N. Dershowitz and M. Okada. Conditional Equational Programming and the Theory of Conditional Term Rewriting. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 337–346, 1988.
35. N. Dershowitz and D.A. Plaisted. Logic Programming cum Applicative Programming. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 54–66, Boston, 1985.
36. N. Dershowitz and D.A. Plaisted. Equational Programming. In J.E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11*, pp. 21–56. Oxford Press, 1988.
37. N. Dershowitz and G. Sivakumar. Solving Goals in Equational Languages. In *Proc. 1st Int. Workshop on Conditional Term Rewriting Systems*, pp. 45–55. Springer

LNCS 308, 1987.
38. R. Echahed. On Completeness of Narrowing Strategies. In *Proc. CAAP'88*, pp. 89–101. Springer LNCS 299, 1988.
39. R. Echahed. Uniform Narrowing Strategies. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 259–275. Springer LNCS 632, 1992.
40. M.J. Fay. First-Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction*, pp. 161–167, Austin (Texas), 1979. Academic Press.
41. L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.
42. L. Fribourg. Prolog with Simplification. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pp. 161–183. Elsevier Science Publishers, 1988.
43. J.H. Gallier and S. Raatz. Extending SLD-Resolution to Equational Horn Clauses Using E-Unification. *Journal of Logic Programming (6)*, pp. 3–43, 1989.
44. J.H. Gallier and W. Snyder. Complete Sets of Transformations for General E-Unification. *Theoretical Computer Science*, Vol. 67, pp. 203–260, 1989.
45. H. Ganzinger. A Completion Procedure for Conditional Equations. *J. of Symb. Computation*, Vol. 11, pp. 51–81, 1991.
46. A. Geser and H. Hussmann. Experiences with the RAP system – a specification interpreter combining term rewriting and resolution. In *Proc. of ESOP 86*, pp. 339–350. Springer LNCS 213, 1986.
47. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.
48. E. Giovannetti and C. Moiso. A completeness result for E-unification algorithms based on conditional narrowing. In *Proc. Workshop on Foundations of Logic and Functional Programming*, pp. 157–167. Springer LNCS 306, 1986.
49. J.A. Goguen and J. Meseguer. Eqlog: Equality, Types, and Generic Modules for Logic Programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pp. 295–363. Prentice Hall, 1986.
50. J.A. Goguen and J. Meseguer. Models and Equality for Logical Programming. In *Proc. of the TAPSOFT '87*, pp. 1–22. Springer LNCS 250, 1987.
51. J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming. In *Proc. CSL'92*, pp. 216–230. Springer LNCS 702, 1992.
52. W. Hans, R. Loogen, and S. Winkler. On the Interaction of Lazy Evaluation and Backtracking. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 355–369. Springer LNCS 631, 1992.
53. M. Hanus. Compiling Logic Programs with Equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 387–401. Springer LNCS 456, 1990.
54. M. Hanus. A Functional and Logic Language with Polymorphic Types. In *Proc. Int. Symposium on Design and Implementation of Symbolic Computation Systems*, pp. 215–224. Springer LNCS 429, 1990.
55. M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 344–365. Springer LNAI 567, 1991.
56. M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 1–23. Springer LNCS 631, 1992.

57. M. Hanus. Incremental Rewriting in Narrowing Derivations. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 228–243. Springer LNCS 632, 1992.
58. M. Hanus. On the Completeness of Residuation. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming*, pp. 192–206. MIT Press, 1992.
59. M. Hanus. Lazy Unification with Simplification. In *Proc. 5th European Symposium on Programming*, pp. 272–286. Springer LNCS 788, 1994.
60. M. Hanus. Towards the Global Optimization of Functional Logic Programs. In *Proc. 5th International Conference on Compiler Construction*, pp. 68–82. Springer LNCS 786, 1994.
61. M. Hanus and B. Josephs. A Debugging Model for Functional Logic Programs. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 28–43. Springer LNCS 714, 1993.
62. S. Haridi and P. Brand. Andorra Prolog: An Integration of Prolog and Committed Choice Languages. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 745–754, 1988.
63. R. Harper, D.B. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, University of Edinburgh, 1986.
64. A. Herold. Narrowing Techniques Applied to Idempotent Unification. SEKI-Report SR 86-16, Univ. Kaiserslautern, 1986.
65. T. Hickey and S. Mudambi. Global Compilation of Prolog. *Journal of Logic Programming*, Vol. 7, pp. 193–230, 1989.
66. S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNCS 353, 1989.
67. P. Hudak and P. Wadler. Report on the Functional Programming Language Haskell. *SIGPLAN Notices*, Vol. 27, No. 5, 1992.
68. J. Hughes. Why Functional Programming Matters. Technical Report 16, Programming Methodology Group, University of Goteborg, 1984.
69. J.-M. Hullot. Canonical Forms and Unification. In *Proc. 5th Conference on Automated Deduction*, pp. 318–334. Springer LNCS 87, 1980.
70. H. Hussmann. Unification in Conditional-Equational Theories. In *Proc. EUROCAL '85*, pp. 543–553. Springer LNCS 204, 1985.
71. J. Jaffar, J.-L. Lassez, and M.J. Maher. A theory of complete logic programs with equality. *Journal of Logic Programming*, Vol. 1, pp. 211–223, 1984.
72. R. Jagadeesan, K. Pingali, and P. Panangaden. A Fully Abstract Semantics for a First-Order Functional Language with Logic Variables. *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 577–625, 1991.
73. A. Josephson and N. Dershowitz. An Implementation of Narrowing. *Journal of Logic Programming (6)*, pp. 57–77, 1989.
74. A. Kågedal and F. Kluźniak. Enriching Prolog with S-Unification. In J. Darlington and R. Dietrich, editors, *Workshop on Declarative Programming*, pp. 51–65. Springer, Workshops in Computing, 1991.
75. S. Kaplan. Conditional Rewrite Rules. *Theoretical Computer Science 33*, pp. 175–193, 1984.
76. S. Kaplan. Simplifying conditional term rewriting systems: Unification, termination, and confluence. *Journal of Symbolic Computation*, Vol. 4, No. 3, pp. 295–334, 1987.
77. D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pp. 263–297. Pergamon Press, 1970.
78. S. Krischer and A. Bockmayr. Detecting Redundant Narrowing Derivations by the LSE-SL Reducibility Test. In *Proc. RTA'91*. Springer LNCS 488, 1991.

79. H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph-based Implementation of a Functional Logic Language. In *Proc. ESOP 90*, pp. 271–290. Springer LNCS 432, 1990.
80. H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Implementing a Lazy Functional Logic Language with Disequality Constraints. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming*. MIT Press, 1992.
81. H. Kuchen, J.J. Moreno-Navarro, and M.V. Hermenegildo. Independent AND-Parallel Implementation of Narrowing. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 24–38. Springer LNCS 631, 1992.
82. G. Lindstrom, J. Maluszynski, and T. Ogi. Our LIPS Are Sealed: Interfacing Functional and Logic Programming Systems. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 428–442. Springer LNCS 631, 1992.
83. J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.
84. H. Lock, A. Mück, and T. Streicher. A Tiny Constraint Functional Logic Programming Language and Its Continuation Semantics. In *Proc. 5th European Symposium on Programming*, pp. 439–453. Springer LNCS 788, 1994.
85. H.C.R. Lock. *The Implementation of Functional Logic Programming Languages*. PhD thesis, Technical University of Berlin, 1992. Also available as GMD Report 208, Oldenbourg Verlag, München.
86. R. Loogen. Relating the Implementation Techniques of Functional and Functional Logic Languages. *New Generation Computing*, Vol. 11, pp. 179–215, 1993.
87. R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. PLILP'93*. Springer LNCS, 1993.
88. R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 335–346. Springer LNCS 528, 1991.
89. F.J. López Fraguas. A General Scheme for Constraint Functional Logic Programming. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 213–227. Springer LNCS 632, 1992.
90. F.J. López Fraguas and M. Rodríguez-Artalejo. An Approach to Constraint Functional Logic Programming. Technical Report DIA/91/4, Universidad Complutense, Madrid, 1991.
91. A.V. Mantsivoda. Flang: a functional-logic language. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 257–270. Springer LNAI 567, 1991.
92. A.V. Mantsivoda. Flang and its Implementation. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 151–166. Springer LNCS 714, 1993.
93. C.S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming (1)*, pp. 43–66, 1985.
94. A. Middeldorp and E. Hamoen. Counterexamples to Completeness Results for Basic Narrowing. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 244–258. Springer LNCS 632, 1992.
95. J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. In *Proc. Second International Conference on Algebraic and Logic Programming*, pp. 298–317. Springer LNCS 463, 1990.
96. J.J. Moreno-Navarro, H. Kuchen, J. Marino-Carballo, S. Winkler, and W. Hans. Efficient Lazy Narrowing Using Demandedness Analysis. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Pro-*

gramming, pp. 167–183. Springer LNCS 714, 1993.

97. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.
98. A. Mück. Compilation of Narrowing. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 16–29. Springer LNCS 456, 1990.
99. A. Mück. CAMEL: An Extension of the Categorical Abstract Machine to Compile Functional/Logic Programs. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 341–354. Springer LNCS 631, 1992.
100. G. Nadathur and D. Miller. An Overview of λProlog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 810–827. MIT Press, 1988.
101. L. Naish. *Negation and Control in Prolog.* Springer LNCS 238, 1987.
102. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 15–26. Springer LNCS 528, 1991.
103. W. Nutt, P. Réty, and G. Smolka. Basic Narrowing Revisited. *Journal of Symbolic Computation*, Vol. 7, pp. 295–317, 1989.
104. S. Okui and T. Ida. Lazy Narrowing Calculi. Report ISE-TR-92-97, Univ. of Tsukuba, 1992.
105. P. Padawitz. Strategy-Controlled Reduction and Narrowing. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 242–255. Springer LNCS 256, 1987.
106. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.
107. P. Padawitz. Generic Induction Proofs. In *Proc. of the 3rd Intern. Workshop on Conditional Term Rewriting Systems*, pp. 175–197. Springer LNCS 656, 1992.
108. Z. Qian. Higher-Order Equational Logic Programming. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 254–267, Portland, 1994.
109. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.
110. P. Réty. Improving basic narrowing techniques. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 228–241. Springer LNCS 256, 1987.
111. P. Réty, C. Kirchner, H. Kirchner, and P. Lescanne. NARROWER: a new algorithm for unification and its application to Logic Programming. In *First Conference on Rewriting Techniques and Applications*, pp. 141–157, Dijon, 1985. Springer LNCS 202.
112. J.A. Robinson and E.E. Sibert. LOGLISP: Motivation, Design and Implementation. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pp. 299–313. Academic Press, 1982.
113. M. Sato. Quty: A Concurrent Language Based on Logic and Function. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 1034–1056. MIT Press, 1987.
114. D.W. Shin, J.H. Nang, S.R. Maeng, and J.W. Cho. A Typed Functional Extension of Logic Programming. *New Generation Computing*, Vol. 10, pp. 197–221, 1992.
115. J. Siekmann and P. Szabó. Universal Unification and a Classification of Equational Theories. In *Proc. 6th Conference on Automated Deduction*, pp. 369–389. Springer LNCS 138, 1982.
116. J.H. Siekmann. An Introduction to Unification Theory. In R.B. Banerji, editor, *Formal Techniques in Artificial Intelligence*, pp. 369–425. Elsevier Science

Publishers, 1990.

117. F.S.K. Silbermann and B. Jayaraman. Set Abstraction in Functional and Logic Programming. In *Fourth International Conference on Functional Programming and Computer Architecture*, pp. 313–326, 1989.

118. F.S.K. Silbermann and B. Jayaraman. A Domain-theoretic Approach to Functional and Logic Programming. Technical Report TUTR 91-109, Tulane University, 1991. Also in Journal of Functional Programming, Vol. 2, No. 3, 1992, pp. 273-321.

119. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *Journal of the ACM*, Vol. 21, No. 4, pp. 622–642, 1974.

120. G. Smolka. Fresh: A Higher-Order Language Based on Unification and Multiple Results. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pp. 469–524. Prentice Hall, 1986.

121. G. Smolka. TEL (Version 0.9) Report and User Manual. SEKI Report SR-87-11, FB Informatik, Univ. Kaiserslautern, 1988.

122. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

123. P.A. Subrahmanyam and J.-H. You. FUNLOG: a Computational Model Integrating Logic Programming and Functional Programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pp. 157–198. Prentice Hall, 1986.

124. H. Tamaki. Semantics of a logic programming language with a reducibility predicate. In *Proc. 1984 Symposium on Logic Programming*, pp. 259–264, 1984.

125. A. Taylor. LIPS on a MIPS: Results form a Prolog Compiler for a RISC. In *Proc. Seventh International Conference on Logic Programming*, pp. 174–185. MIT Press, 1990.

126. A.K. Turk. Compiler Optimizations for the WAM. In *Proc. Third International Conference on Logic Programming (London)*, pp. 657–662. Springer LNCS 225, 1986.

127. D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, pp. 1–16. Springer LNCS 201, 1985.

128. M.H. van Emden and K. Yukawa. Logic Programming with Equations. *Journal of Logic Programming (4)*, pp. 265–288, 1987.

129. P.L. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Univ. of California Berkeley, 1990. Report No. UCB/CSD 90/600.

130. D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, 1983.

131. D. Wolz. Design of a Compiler for Lazy Pattern Driven Narrowing. In *Recent Trends in Data Type Specification*, pp. 362–379. Springer LNCS 534, 1990.

132. J.-H. You. Enumerating Outer Narrowing Derivations for Constructor-Based Term Rewriting Systems. *Journal of Symbolic Computation*, Vol. 7, pp. 319–341, 1989.