An Implementation of Narrowing Strategies

Sergio Antoy
antoy@cs.pdx.edu

Bart Massey
bart@cs.pdx.edu

Department of Computer Science
Portland State University
P.O. Box 751, Portland, OR 97207
U.S.A.

Michael Hanus
mh@informatik.uni-kiel.de

Frank Steiner
fst@informatik.uni-kiel.de

Institut für Informatik
Christian-Albrechts-Universität Kiel
Olshausenstr. 40, D-24098 Kiel
Germany

ABSTRACT

This paper describes an implementation of narrowing, an essential component of implementations of modern functional logic languages. These implementations rely on narrowing, in particular on some optimal narrowing strategies, to execute functional logic programs. We translate functional logic programs into imperative (Java) programs without an intermediate abstract machine. A central idea of our approach is the explicit representation and processing of narrowing computations as data objects. This enables the implementation of operationally complete strategies (i.e., without backtracking) or techniques for search control (e.g., encapsulated search). Thanks to the use of an intermediate and portable representation of programs, our implementation is general enough to be used as a common back end for a wide variety of functional logic languages.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Multiparadigm Languages*

General Terms

Languages, Design, Theory, Experimentation

Keywords

Functional logic, narrowing, Curry, XML, Java

1. INTRODUCTION

This paper describes an implementation of narrowing for overlapping inductively sequential rewrite systems [5]. Narrowing is the essential computational engine of functional logic languages (see [14] for a survey on such languages and their implementations). An implementation of narrowing translates a program consisting of rewrite rules into executable code. This executable code currently falls into two categories: Prolog predicates (e.g., [4, 12, 15, 27]) or instructions for an abstract machine (e.g., [11, 19, 26, 29]). Although these approaches are relatively simple, in both cases, several layers of interpretation separate the functional logic program from the hardware intended to execute it. Obviously, this situation does not lead to efficient execution.

In this paper we investigate a different approach. We translate a functional logic program into an imperative program. Our target language is Java, but we make limited use of specific object-oriented features, such as inheritance and dynamic polymorphism. Replacing Java with a lower-level target language, such as C or machine code, would be a simple task.

In Section 2 we briefly introduce the aspects of functional logic programming relevant to our discussion. In Section 3 we review background information for the key concepts presented in this paper. In Section 4 we describe the elements and the characteristics of our implementation of narrowing. In Section 5 we describe aspects of our compilation process, as well as execution issues such as input, output and tracing/debugging that may greatly affect the usability of a system. In Section 6 we summarize current efforts toward the implementation of functional logic languages, particularly w.r.t. implementations of narrowing and how they compare to our work. Section 7 sketches planned extensions to our framework, and Section 8 offers some conclusions.

2. FUNCTIONAL LOGIC PROGRAMS

Functional logic languages combine the operational principles of two of the most important declarative programming paradigms, namely functional and logic programming (see [14] for a survey). Efficient demand-driven functional computations are amalgamated with the flexible use of logical variables, providing for function inversion and search for solutions. Functional logic languages with a sound and complete operational semantics are usually based on narrowing (originally introduced in automated theorem proving [32]) which combines reduction (from the functional part) and variable instantiation (from the logic part). A *narrowing* *step* instantiates variables of an expression and applies a reduction step to a redex of the instantiated expression. The instantiation of variables is usually computed by unifying a subterm of the entire expression with the left-hand side of some program equation.

EXAMPLE 1. *Consider the following rules defining the $\leq$ predicate* **leq** *on natural numbers which are represented by terms built from* **zero** *and* **succ**:

```
leq(zero,Y)           = true
leq(succ(X),zero)     = false
leq(succ(X),succ(Y))  = leq(X,Y)
```

The expression **leq(succ(M),Y)** *can be evaluated (i.e., reduced to a value) by instantiating* **Y** *to* **succ(N)** *to apply the third equation, followed by the instantiation of* **M** *to* **zero** *to apply the first equation:*

$$\texttt{leq(succ(M),Y)} \leadsto_{\{\texttt{Y}\mapsto\texttt{succ(N)}\}} \texttt{leq(M,N)} \leadsto_{\{\texttt{M}\mapsto\texttt{zero}\}} \texttt{true}$$

Narrowing provides completeness in the sense of logic programming (computation of all answers, i.e., substitutions leading to successful evaluations) as well as functional programming (computation of values). Since simple narrowing can have a huge search space, a lot of effort has been made to develop sophisticated narrowing strategies without losing completeness (see [14]). *Needed narrowing* [7] is based on the idea of evaluating only subterms which are *needed* in order to compute a result. For instance, in a term like **leq(**t_1**,**t_2**)**, it is always necessary to evaluate t_1 (to some variable or constructor-rooted term) since all three rules in Example 1 have a non-variable first argument. On the other hand, the evaluation of t_2 is only needed if t_1 is of the form **succ(**t**)**. Thus, if t_1 is a free variable, needed narrowing instantiates it to a constructor term, here **zero** or **succ(V)**. Depending on this instantiation, either the first equation is applied or the second argument t_2 is evaluated. Needed narrowing is currently the best narrowing strategy for first-order (inductively sequential) functional logic programs due to its optimality properties w.r.t. the length of derivations and the independence of computed solutions, and due to the possibility of efficiently implementing needed narrowing by pattern matching and unification [7]. Moreover, it has been extended in various directions, e.g., higher-order functions and λ-terms as data structures [18], overlapping rules [5], and concurrent computations [16].

Needed narrowing is complete, in the sense that for each solution to a goal there exists a narrowing derivation computing a more general solution. However, most of the existing implementations of narrowing lack this property since they are based on Prolog-style backtracking. Since backtracking is not fair in exploring all derivation paths, some solutions might not be found in the presence of infinite derivations, i.e., these implementations are incomplete from an operational point of view. An important property of our implementation is its operational completeness, i.e., all computable answers are eventually computed by our implementation.

3. BACKGROUND

Since pattern matching is an essential feature of existing functional logic languages, term rewriting systems (TRSs) are an adequate formal model for functional logic programs. Therefore, we review in the following some notions from term rewriting [9].

We consider a (*many-sorted*) *signature* partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for n-ary constructor and operation symbols, respectively. As usual, *terms* are built from these symbols and *variables* (e.g., x, y, z). A *constructor term* is a term without operation symbols. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is *ground* if $\mathcal{V}ar(t) = \varnothing$. A term is *linear* if it does not contain multiple occurrences of one variable.

A *pattern* is a term of the form $f(d_1, \ldots, d_n)$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n$ are constructor terms. A term is *operation-rooted* (*constructor-rooted*) if it has an operation (constructor) symbol at the root. 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.

We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$) and $\sigma(x) = x$ for all other variables x. Substitutions are extended to morphisms on terms by $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$ for every term $f(t_1, \ldots, t_n)$.

A set of *rewrite rules* $l = r$ such that l is not a variable and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* (*lhs*) and the *right-hand side* (*rhs*) of the rule, respectively. A TRS $\mathcal{R}$ is *left-linear* if l is linear for all $l = r \in \mathcal{R}$. A TRS is *constructor based* (CB) if each lhs l is a pattern. In the remainder of this paper, a functional logic *program* is a left-linear CB-TRS.

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists a position p in t, a rewrite rule $R = l = r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$ (p and R will often be omitted in the notation of a computation step). The instantiated lhs $\sigma(l)$ is called a *redex* and the instantiated rhs $\sigma(r)$ is called the *reduct* of this redex. A (constructor) *head normal form* is either a variable or a constructor-rooted term. A term t is called *irreducible* or in *normal form* if there is no term s with $t \to s$. $\to^+$ denotes the transitive closure of $\to$ and $\to^*$ denotes the reflexive and transitive closure of $\to$.

To evaluate terms containing variables, narrowing non-deterministically instantiates the variables so that a rewrite step is possible. Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \to_{p,R} t'$. We denote by $t_0 \leadsto^*_\sigma t_n$ a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$. Since we are interested in computing *values* (constructor terms) as well as *answers* (substitutions) in functional logic programming, we say that the narrowing derivation $t \leadsto^*_\sigma c$ *computes the result* c *with answer* σ if c is a constructor term. The evaluation to ground constructor terms (and not to arbitrary expressions) is the intended semantics of functional languages and also of most functional logic languages.

A challenge in the design of functional logic languages is the definition of a "good" narrowing strategy, i.e., a restriction on the narrowing steps issuing from t, without losing completeness. In the following, we briefly outline the needed narrowing strategy (a formal description can be found in [7]).

Needed narrowing extends Huet and Lévy's notion of a needed reduction [23] and is defined on *inductively sequential programs* [3]. Roughly speaking, in an inductively sequen-

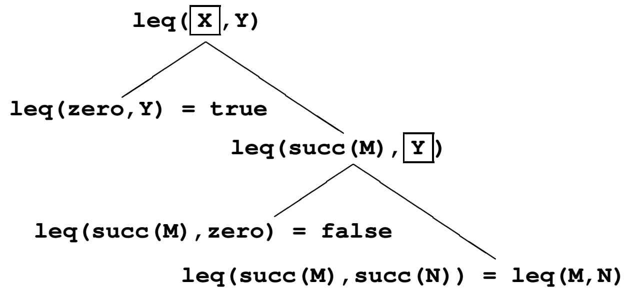

Figure 1: Definitional tree for the operation `leq` of Example 1

tial program the rules for each function can be organized in a tree-like structure (*definitional tree* [3]). The leaves contain all (and only) the rules defining the function. The inner nodes have a discriminating argument, also called an *inductive position*: all child nodes have different constructor symbols at this position. For instance, the definitional tree for the function **`leq`** in Example 1 is illustrated in Figure 1; the inductive position is marked by a surrounding box.

The computation of a needed narrowing step is guided by the definitional tree for the root of the operation-rooted term t. If t is a leaf node, we reduce it with the rule at this leaf. Otherwise, we check the subterm corresponding to the inductive position of the branch: if it is a variable, it is (non-deterministically) instantiated to the constructor of a child; if it is already a constructor, we proceed with the corresponding child; if it is a function, we evaluate it (to head normal form) by recursively applying needed narrowing.

4. IMPLEMENTATION OF NEEDED NARROWING

In this section we describe the main ideas of our implementation of narrowing. We implement a strategy, referred to as *INS* [5], proven sound and complete for the class of the overlapping inductively sequential rewrite systems. In these systems, the left-hand sides of the rewrite rules defining an operation can be organized in definitional trees. However, an operation may have distinct rewrite rules with the same left-hand side (modulo renaming of variables): operation **`coin`** (Section 4.8), is one example. To ease the understanding of our work, we first describe the implementation of rewrite computations in inductively sequential rewrite systems. We then describe the extensions that lead to narrowing in overlapping inductively sequential rewrite systems.

4.1 Overview

The overall goals of our implementation are speed of execution and operational completeness. The following principles guide our implementation and are instrumental in achieving the goal.

1. A reduction step replaces a redex of a term with its reduct. A term is represented as a tree-like data structure. The execution of a reduction updates only the portion of this data structure affected by the replacement. Thus, the cost of a reduction is independent of its context. We call this principle *in-place* replacement.
2. Only somewhat needed steps are executed. We use the qualifier "somewhat" because different notions of *need* have been proposed for different classes of rewrite systems. We execute a particular kind of steps that for reductions in orthogonal systems is known as *root-needed* [30]. Thus, reductions that are a priori useless are never performed. We call this principle *useful step.*
3. *Don't know* non-deterministic reductions are executed in parallel. Both narrowing computations (in most rewrite systems) and reductions (in interesting rewrite systems) are non-deterministic. Without some form of parallel execution, operational completeness would be lost. We call this principle *operational completeness.*

In inductively sequential rewrite systems, and when computations are restricted to rewriting, it is relatively easy to faithfully implement all the above principles. In fact, our implementation does it. However, our environment is considerably richer. We execute *narrowing* computations in *overlapping* inductively sequential rewrite systems. In this situation, two complications arise. The non-determinism of narrowing and/or of overlapping rules imply that a redex may have several replacements. In these situations, there cannot be a single in-place replacement. Furthermore, the steps that we compute in *overlapping* inductively sequential rewrite systems are needed, but only modulo non-deterministic choices [5]. Hence, some step may not be needed in the strict sense of [7, 23], but we may not be able to know by feasible means which steps.

The architecture of our implementation is characterized by *terms* and *computations.* Both terms and computations are organized into tree-like linked (dynamic) structures. A *term* consists of a *root symbol* applied to zero or more *arguments* which are themselves terms. A *computation* consists of a stack of *terms* that identify reduction steps. All the terms in the stack, with the possible exception of the top, are not yet redexes, but will eventually become redexes, and be reduced, before the computation is complete. In terms, links go from a parent to its children, whereas in computations links go from children to their parent.

A graphical representation of these objects is shown in Figure 2. In this figure, the steps to the left represent the terms in the stack of the computation. $Step_0$ is the bottom of the stack: it cannot be executed before $Step_1$ is executed. Likewise $Step_1$ cannot be executed before $Step_2$ is executed.

To ease understanding, we begin with an account of our implementation of rewriting computations in inductively sequential rewrite systems. Although non-trivial, this implementation is simple enough to inspire confidence in both its correctness and efficiency. Then, we generalize the discussion to larger classes of rewrite systems and finally to narrowing computations and argue why both correctness and efficiency of this initial implementation are preserved by these extensions.

4.2 Symbol representation

Symbols are used to represent terms. A *symbol* is an object that contains two pieces of information: a *name* and a *kind.* Since there is no good reason to have more than one instance of a given symbol in a program, each distinct symbol is implemented as an immutable singleton object. The *name* is a string. The *kind* is a tag that classifies a symbol. For now, the tag is either "defined operation" or "data constructor". Additional tags will be defined later to compute with larger classes of rewrite systems. The tag of a symbol is used to dispatch computations that depend on the classification of a symbol. Of course, we could dispatch these computations by dynamic polymorphism, i.e., by defining an abstract method overridden by subclasses. Often, these methods would consist of a few statements that use the environment of the caller. A tag avoids both a proliferation of small methods and the inefficiency of passing around the environment. Furthermore, this architecture supports implementations in objectless target languages as well.

Nevertheless, in our Java architecture, class *symbol* has subclasses such as *operation* and *constructor*. In particular, there is one subclass of *operation* for each defined operation f of a functional logic program. This class, according to our second principle, contains the code for the execution of a useful step of any term rooted by f. Operations are defined by rewrite rules. We use the following rules in the examples to come.

```
add (zero, Y)      = Y
add (succ (X), Y) = succ (add (X, Y))

positive (zero)      = false
positive (succ (_)) = true
```

4.3 Term representation

Terms of user-defined type contain two pieces of information: the *root* of the *term*, which is a *symbol*, and the *arguments* of the *root*, which are *terms* themselves. Terms of builtin types contain specialized information, e.g., terms of the builtin type *int* contain an *int*. This situation suggests defining a common base class and a specialization of this class for each appropriate type of term. However, this is in conflict with the fact that according to the first principle of our implementation, a *term* is a mutable object. In Java, the class of an object cannot change during execution.

Therefore, we implement a *term* as a bridge pattern. A term delegates its functionality to a representation. Different types, such as user-defined types, builtin types, and variables are represented differently. All the representations provide a common functionality. The representation of a term object can change at run-time and thus provide mutability of both value and behavior as required by the implementation.

4.4 Computation representation

A *computation* is an object abstracting the necessity to execute a sequence of specific reduction steps in a term. Class *computation* contains two pieces of information:

1. A *stack of terms* to be contracted (reduced at the root). The terms in the stack are not redexes except, possibly, the top term. Each term in the stack is a subterm of the term below it, and must be reduced to a constructor-rooted term in order to reduce the term below it. Therefore, the elements of the stack in a computation may be regarded as steps as well. The underpinning theoretical justification of this stack of steps is in the proof of Th. 24 of the extended version of [5]. We ensure that every term in the stack eventually will be contracted. To achieve this aim, if a complete strategy cannot execute a step in an operation-rooted term, it reduces the term to the special value *failure*.
2. A set of *bookkeeping information*. For example, this information includes the number of steps executed by the computation and the elapsed time. An interesting bookkeeping datum is the state of a computation. Computations being executed are in a *ready* state. A computation's state becomes *exhausted* after the computation has been executed and it has been determined that no more steps will be executed at the root of the bottom-most term of the stack. Before becoming exhausted a computation state may be either *result* or *failure*. Later, we will extend our model of computation with residuation. With the introduction of residuation, a new state of a computation, *flounder*, is introduced as well.

Loosely speaking, an initial computation is created for an initial top-level expression to evaluate. This expression is the top and only term of the stack of this computation. If the top term t is not a redex, a subterm of t needed to contract t is placed on the stack and so on until a redex is found. A redex on top of the stack is replaced by its reduct. If the reduct is constructor-rooted, the stack is popped (its top element is discarded).

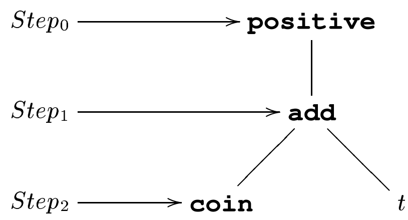

Figure 2: Snapshot of a computation of term `positive(add(coin,`t`))`

4.5 Search space representation

The search space is a queue of computations which are repeatedly selected for processing. The machinery of a queue and fair selection is not necessary for rewriting in inductively sequential rewrite systems. For these systems, computations are strictly sequential and consequently a single (possibly implicit) stack of steps would suffice. However, the architecture that we describe not only accommodates the extensions from rewriting to narrowing and/or from inductively sequential rewrite systems to the larger classes that are coming later, but it allows us to compute more efficiently.

A computation serves two purposes: (1) finding maximal operation-rooted subterms t of the top-level term to evaluate and (2) reducing each t to head normal form. The pseudo-code of Figure 3 sketches part (2), which is the most challenging. Some optimizations would be possible, but we avoid them for the sake of clarity.

Since inductively sequential rewrite systems are confluent, replacing in-place a subterm u of a term t with u's reduct does not prevent reaching t's normal form. When a term has a result this result is found, since repeated contractions of needed redexes are normalizing.

4.6 Sentinel

The first extension to the previous model is the introduction of a "sentinel" at the root of the top-level expression being evaluated. For this, we introduce a distinguished

```
while the queue is not empty
| select a ready computation k from the queue
| let t be the term at the top of k's stack
| switch on the root of t
| | case t is operation-rooted
| | | switch on the reducibility of t
| | | | case t is a redex
| | | | | replace t with its reduct
| | | | | put k back into the queue
| | | | case t is not a redex
| | | | | switch on s, a maximal needed subterm of t
| | | | | | case s exists
| | | | | | | push s on k's stack
| | | | | | | put k back into the queue
| | | | | | case s does not exist
| | | | | | | stop the computation, no result exists
| | | | | endswitch
| | | endswitch
| | case t is constructor-rooted
| | | pop k's stack
| | | if k's stack is not empty
| | | | put k back into the queue
| endswitch
endwhile
```

Figure 3: Procedure to evaluate a term to a head normal form

symbol called *sentinel* that takes exactly one argument of any kind. If t is the term to evaluate, our implementation evaluates *sentinel*(t) instead. Thus, this is the actual term of the initial computation. Symbol *sentinel* has characteristics of both an operation and a constructor. Similar to an operation, the stack of the initial computation contains *sentinel*(t), but similar to a constructor, *sentinel*(t) cannot be contracted for any t. Having a sentinel has several advantages. The strategy works with the sentinel by means of implicit rewrite rules that always look for an internal needed redex and never contract the *sentinel*-rooted term itself. Also, using a sentinel saves frequent tests similar to using a sentinel in many classic algorithms, e.g., sorting.

4.7 Failure

The second extension to the previous model is concerned with the possibility of a "failure" of a computation. A failure occurs when a term has no constructor normal form. The computation detects a failure when the strategy, which is complete, finds no useful steps (redexes) in an operation-rooted term.

The pseudo-code presented earlier simply terminates the computation when it detects a failure. For the extensions discussed later it is more convenient to explicitly represent failures in a term. This allows us, e.g., to clean up computations that cannot be completed and to avoid duplicating certain computations. To this purpose we introduce a new symbol called *failure*. The *failure* symbol is treated as a constant constructor.

Suppose that u is an operation-rooted term. If the strategy finds no step in u, it evaluates u to *failure*. A *failure* symbol is treated as a constructor during the pattern matching process. Implicit rewrite rules for each defined operation rewrite any term t to *failure* when a *failure* occurs at a needed position of t. For example, we perform the following reduction:

add (failure, v**)** $\rightarrow$ **failure**

With these implicit rewrite rules, an inner occurrence of *failure* in a term propagates up to the sentinel, which can thus report that a computation has no result. The explicit representation of failing computations is also important in performing non-deterministic computations.

4.8 Non-determinism

The third extension to the previous model is concerned with non-determinism. In our work, non-determinism is expressed by rewrite rules with identical left-hand sides, but distinct right-hand sides. A textbook example of a non-deterministic defined operation is:

```
coin = zero
coin = succ (zero)
```

This operation differs from the previous ones in that a given term, say $s =$ **coin**, has two distinct reducts.

The most immediate problem posed by non-deterministic operations is that if s occurs in some term t and we replace in-place s with one of its replacements, we may lose a result that could be obtained with another replacement. If a term such as s becomes the top of the stack of a computation k, we change the state of k to *exhausted* and we start two or more new computations. Each new computation, say k', begins with a stack containing a single term obtained by one of the several possible reductions of s.

The procedure described above can be optimized in many ways. We mention only the most important one that we have implemented — the sharing of subterms disjoint from s. We show this optimization in an example. Suppose that the top-level term being evaluated is:

add (coin, t**)**

The non-determinism of **coin** gives rise to the computation of the following two terms:

add (zero, t**)**
add (succ (zero), t**)**

These terms are evaluated concurrently and independently. However, term t in the above display is shared rather than duplicated. Sharing improves the efficiency of computations since only one term, rather than several equal copies, is constructed and possibly evaluated. In some situations, a shared term may occur in the stacks of two independent computations and be concurrently evaluated by each computation. This approach avoids a common problem of backtracking-based implementations of functional logic languages, in which t will be evaluated twice if it is needed during the evaluation of both **add** terms shown above.

4.9 Rewrite rules

The final relevant portion of our architecture is the implementation of rewrite rules. All the rules of an ordinary defined operation f are translated into a single Java method. This method implicitly uses a definitional tree of f to compare constructor symbols in inductive positions of the tree with corresponding occurrences in an f-rooted term t to reduce. Let k_t be a computation in the queue, *ready* the state of k_t, and t the term on the top of k_t's stack. The following case breakdown defines the code that needs to be generated.

1. If t is a redex with a single reduct, then t is replaced in-place by its reduct.

2. If t is a redex with several reducts, then a new computation is started for each reduct. The state of k_t is changed to *exhausted*.

3. If in a needed position of t there is *failure*, then t is considered a redex as well and it is replaced in-place by *failure*.

4. If in a needed position of t there is an operation-rooted ordinary term s, then s is pushed on the stack of k_t.

5. The last case to consider is when operation f is incompletely defined and no needed subterm is found in t. In this case, t is replaced in-place by *failure*.

4.10 Narrowing

At this point we are ready to discuss the extension of our implementation to narrowing. A narrowing step instantiates variables in a way very similar to a non-deterministic reduction step. For example, suppose that *allnat* is an operation defined by the rules:

```
allnat = zero
allnat = succ (allnat)
```

Narrowing term **add(X,t)**, where **X** is an uninstantiated variable and t is any term, is not much different from reducing **add(allnat,t)**.

There are two key differences in the handling of variables w.r.t. non-deterministic reductions: (1) we must keep track of variable bindings to construct the *computed answer* at the end of a computation, and (2) if a given variable occurs repeatedly in a term being evaluated, the replacement of a variable with its binding must replace all the occurrences. We solve point (1) by storing the binding of a variable in a computation. Point (2) is simply bookkeeping. We represent substitutions "incrementally." A computation computes both a value (for the functional part) and an answer (for the logic part). The answer is a substitution. In most cases, a narrowing step produces several distinct bindings for a variable. Each of these bindings increments a previously computed substitution. For example, suppose that the expression to narrow is:

add (X, Y) = t

for some term t. Some computation may initially bind **X** to **zero**. Later on, a narrowing step may bind **Y** independently to both **zero** and **succ(Y$_1$)**. These bindings will "add" to the previous one. The previous binding is shared, which saves both memory and execution time.

4.11 Parallelism

Our implementation includes a form of parallelism known as *parallel-and*. And-parallel steps do not affect the soundness or completeness of the strategy, *INS*, underlying our implementation, but in some cases they may significantly reduce the size of the narrowing space of a computation — possibly from infinite to finite. The *parallel-and* operation is handled explicitly by our implementation. If a computation k leads to the evaluation of t **&** u, where t and u are terms and "**&**" denotes the parallel-and operation, then steps of both t and u are scheduled. This requires to change the stack of a computation into a tree-like structure. The set of leaves of this tree-like structure replaces the top of the stack previously discussed.

As soon as one of these parallel steps has to be removed from the tree, which means that its term argument has been reduced to a constructor term c (including *failure*), the parent of the step is reconsidered. Depending on c's value, either the parent term is reduced (to a *failure* if $c = failure$) and the other parallel steps are removed, or (if $c = success$) the computation of the other parallel steps continues normally.

4.12 Residuation

Residuation is a computational mechanism that delays the evaluation of a term containing an uninstantiated variable in a needed position [1]. Similar to narrowing, it supports the integration of functional programming with logic programming by allowing uninstantiated variables in functional expressions. However, in contrast to narrowing it is incomplete, i.e., unable to find all the solutions of some problems. Residuation is useful for dealing with built-in types such as numbers [10]. Residuation is meaningful only when a computation has several steps executing in parallel. If a computation has only one step executing, and this step residuates, the computation cannot be completed and it is said to *flounder*.

Operations that residuate are called *rigid*, whereas operations that narrow are called *flexible*. A formal model for the execution of programs defining both rigid and flexible operations is described in [16]. Our implementation already has the necessary infrastructure to accommodate this model. When a step s residuates on some variable V, we store (a reference to) s in V, mark s as *residuating* and continue the execution of the other steps. When V is bound, we remove the *residuating* mark from s so that s can be executed as any other step. If all the steps of a computation are *residuating*, the computation *flounders*.

5. THE COMPILATION PROCESS

The main motivation of this new implementation of narrowing is to provide a generic back end that can be used by functional logic languages based on a lazy evaluation strategy. Current work [6] shows that any narrowing computation in a left-linear constructor-based conditional rewrite system can be simulated, with little or no loss of efficiency, in an overlapping inductively sequential rewrite system, hence by our implementation. Therefore, our implementation can be used by languages such as Curry [21], Escher [25] and Toy [28].

To support this idea, our implementation works independently of any concrete source language. The source programs of our implementation are functional logic programs where all functions are defined at the top level (i.e., no local declarations) and the pattern-matching strategy is explicit. This language, called FlatCurry, has been developed as an intermediate language for the Curry2Prolog compiler [8] in the Curry development system PAKCS [17] and is used for various other purposes, e.g., meta-programming and partial evaluation [2]. Basically, a FlatCurry program is (apart from data type and operator declarations) a list of function declarations where each function f is defined by a single rule of the form $f(x_1, \ldots, x_n) = e$, i.e., the left-hand side consists of pairwise different variable arguments and the right-hand side is an expression containing case expressions for pattern matching.

To be more precise, an expression can take any of the forms shown in Figure 4. The *shallow patterns* p_i occurring in case expressions have the form $c(x_1, \ldots, x_n)$, i.e., all

x	(variable)
$c(e_1,\ldots,e_n)$	(constructor)
$f(e_1,\ldots,e_n)$	(function call)
case e_0 *of* $\{p_1 \to e_1; \ldots; p_n \to e_n\}$	(rigid case)
fcase e_0 *of* $\{p_1 \to e_1; \ldots; p_n \to e_n\}$	(flexible case)
$or(e_1, e_2)$	(choice)
$partcall(f, e_1, \ldots, e_k)$	(partial application)
$apply(e_1, e_2)$	(application)
$constr(\{x_1,\ldots,x_n\}, e)$	(constraint)
$guarded(\{x_1,\ldots,x_n\}, e_1, e_2)$	(guarded expression)

Figure 4: FlatCurry expressions

```
leq(X,Y) = fcase X of {
    zero → true;
    succ(M) → fcase Y of {
        zero → false;
        succ(N) → leq(M,N)
    }
}
```

Figure 5: Encoding of Example 1 in FlatCurry

case branches are constructors applied to pairwise distinct (fresh) variables. Any inductively sequential program can be translated into FlatCurry rules whose right-hand side consists of only constructor applications, function applications and case expressions [18]. For instance, the function **leq** of Example 1 is represented in FlatCurry as shown in Figure 5.

The other options for expressions are used for the extensions of inductively sequential programs that occur in various functional logic languages. For instance, *or* expressions are used to represent non-deterministic choices (see Section 4.8), rigid case expressions for residuation, i.e., functions which suspend on insufficiently instantiated arguments (see Section 4.12), (partial) applications for higher-order functions (which can be implemented by a transformation into first-order rules, see [34]), and guarded expressions for conditional rules[1].

Although FlatCurry was originally designed as an intermediate language to compile and manipulate Curry programs, it should be clear that it can also be used for various other declarative languages (e.g., Haskell-like lazy languages with strict left-to-right pattern matching can be compiled by generating appropriate case expressions). Our back end accepts a syntactic representation of FlatCurry programs in XML format[2] so that other functional logic languages can be compiled into this implementation-independent format. XML is becoming the format of choice for exchanging structured information, such as external representations of compiled programs, between different programs and non-homogeneous systems. Our choice of this format is intended to easily accommodate a variety of source languages and to maximize the usability of our back end. Figure 6 shows the XML code for the FlatCurry representation of **leq** given above.

Our compiler, which is fully implemented in Curry, reads an XML representation and compiles it into a Java program following the ideas described in Section 4. Recall that every function is represented by a subclass of *operation*. For each function, we define a method *expand* which will expand a function call according to its rules and depending on its arguments (Sections 4.9, 4.10).

To show the simplicity of our compiled code, we provide an excerpt of the *expand* method for **leq** in Figure 7 which is generated from the XML representation given above. According to Section 4.9, we must decide whether **leq(**t_1, t_2**)** is a redex. This expression is a redex if t_1 is a variable (we must narrow) or **zero** (we apply the first rule). If t_1 equals **succ(..)**, we must do the same check for the second argument. If t_1 fails, so does **leq**. If t_1 is a function call, we must evaluate it first. For the sake of simplicity, we show pseudo-code, which reflects the basic structure and is very similar to the real Java code.

To use our back end for a functional logic language, it is only necessary to compile programs from this language to a XML representation according to the FlatCurry DTD. For instance, our compiler can be used as a back end for Curry since Curry programs can be translated into this XML representation with PAKCS [17]. Again, it is worth emphasizing that FlatCurry can encode more than just Curry programs or needed narrowing, because the evaluation strategy is compiled into the case expressions. For instance, FlatCurry is a superset of TFL, which is used as an intermediate representation for a Toy-like language based on the CRWL paradigm (Constructor-based conditional ReWriting Logic) [22].

The computation engine is designed to work with the *read-eval-print* loop typical of many functional, logic and functional logic interpreters. In our Java implementation, the computation engine and the read-eval-print loop are threads that interact with each other in a producer/consumer pattern. When a computed expression (value plus answer) becomes available, the computation engine notifies the read-eval-print loop while preserving the state of the narrowing space. The read-eval-print loop presents the results to the user and waits. The user may request further results or terminate the computation. If the user requests a new result, the read-eval-print loop notifies the computation engine to further search the narrowing space. Otherwise, the narrowing space is discarded.

Currently we provide a naive trace facility that is useful to debug both user code and our own implementation. Since the computations originating from a goal are truly concurrent, as is necessary to ensure operational completeness, and since some terms are shared between computations, the trace is not always easy to read. Computations are identified by a unique *id*. We envision a tool, conceptually and structurally well separated from the computation engine, that collects the interleaved traces of all computations, separates them, and presents each trace in a different window for each computation. This tool may have a graphical user interface to select which computations to see and/or interact with.

6. RELATED WORK

In this section we discuss and compare other approaches to functional logic language implementation (see [14] for a survey). Our approach provides an operationally complete

[1] See `http://www.informatik.uni-kiel.de/~curry/flat/` for more details.

[2] The DTD for the XML FlatCurry representation is available from `http://www.informatik.uni-kiel.de/~curry/flatcurry.dtd`.

```
<func name="leq" arity="2">
 <functype>... </functype>                       // the type of the function
 <rule>                                          // the rule for the function
   <lhs> <var>X</var> <var>Y</var> </lhs>        // two arguments, enumerated
   <rhs>
     <case type="flex">                          // evaluate by narrowing
       <var>X</var>                              // switch on first argument
       <branch>
         <pattern name="zero" />                 // if it matches Zero...
         <comb type="ConsCall" name="true" />    // ...reduce to True
       </branch>
       <branch>
         <pattern name="succ">                   // if it matches succ(M)...
           <var>M</var>
         </pattern>
         <case type="flex">                      // ...then go on with second argument
            code for matching the second argument
         </case>
       </branch>  </case>  </rhs>  </rule>  </func>
```

Figure 6: XML code for leq

```
expand (Computation comp) {
    term = comp.getTerm();                      // get the term from top of the stack
    X = term.getArg(0);                         // get first argument
    Y = term.getArg(1);                         // get second argument
    switch on kind of X                         // case X of ...
    case variable:                              // do narrowing: bind to patterns
        X.bindTo(zero);
        spawn new computation for leq(zero,Y);
        X.bindTo(succ(M));
        spawn new computation for leq(succ(M),Y);
        comp.setExhausted();                    // this computation is exhausted
    case constructor:                           // argument is constructor-rooted,
        switch on kind of constructor           // thus do pattern matching
        case zero:                              // apply first rule:
            term.update(true);                  // replace term with true
        case succ:                              // case X of succ(M) → case Y of...
          recursive case for switching on Y
    case failure:                               // the needed subterm has failed,
        term.update(failure)                    // thus leq fails, too
    case operation:                             // X is a function call, thus
        comp.pushOnStack(X);                    // evaluate this call first
}
```

Figure 7: Simplified pseudo-code for the expand method of leq

and efficient architecture for implementing narrowing which can potentially accommodate sophisticated concepts, e.g., the combination of narrowing and residuation, encapsulated search or committed choice. As some recent narrowing-based implementations of functional logic languages show, most implementations that include these concepts lack completeness or are inefficient.

One common approach to implement functional logic languages is the transformation of source functional logic programs into Prolog programs. This approach is favored for its simplicity since Prolog has most of the features of functional logic languages: logical variables, unification, and non-determinism implemented by backtracking. However, the challenge in such an implementation is the implementation of a sophisticated evaluation strategy that exploits the presence of functions in the source programs. Different implementations of this kind are compared and evaluated in [15] where it is demonstrated that needed narrowing is efficiently implemented in a (strict) language such as Prolog and that this implementation is superior to other narrowing strategies. Therefore, most of the newer proposals to implement functional logic languages in Prolog are based on needed narrowing [4, 8, 15, 27]. In contrast to our implementation of narrowing, all of these efforts are operationally incomplete (i.e., existing solutions might not be found due to infinite derivation paths) since they are based on Prolog's depth-first search mechanism. The same drawback also occurs in implementations of functional logic languages based on abstract machines (e.g., [11, 26, 29, 22]) since these abstract machines use backtracking to implement non-determinism.

An exception is the Curry2Java compiler [19] which is based on an abstract machine implementation in Java but uses independent threads to implement non-deterministic choices. If these threads are fairly evaluated (which can be ensured by specific instructions), infinite derivations in one branch do not prevent finding solutions in other branches. Our approach is more flexible since it does not depend on threads, but it can control to any degree of granularity the scheduling of steps in distinct computations. This eases the implementation of problem-specific search strategies at the top level, whereas Curry2Java is restricted to encapsulated search [20].

Our implementation is the subject of active investigation in several directions. Thus, we are not specifically concerned with its efficiency at this time. Rather, we are studying architectures that easily integrate concepts and ideas that have been proposed for functional logic programming. Efficiency is an important issue, though, and we expect that it will be a strong point of our implementation due to the direct translation into an imperative language without the additional control layers of an abstract machine. While we have attempted to select an efficient architecture, we have not paid much attention to detailed optimization of our implementation, and we do not expect top speed as long as we compile to Java. We performed only a limited number of benchmarks to get a feel for where we stand.

For the functional evaluation, we evaluated the naive reverse of a list of 1200 elements (400 only for comparing Curry2Java). To benchmark non-determinism we evaluated **`add x y =:= peano300`**, where **`peano300`** denotes the term encoding 300 in unary notation and the infix operator **`=:=`** denotes the strict equality with unification. This goal is solved by creating 301 parallel computations by narrowing on the **`add`** operation.

The two fastest available implementations of needed narrowing, to the best of our knowledge, are the Curry2Prolog compiler of the PAKCS system and the *Münster Curry Compiler (MCC)* [29]. The Curry2Java back end (C2J), included in the PAKCS system, is not as fast, but is the fastest available correct and complete implementation of needed narrowing. We have also compared our approach to a Java-based implementation of Prolog: Jinni [33] is the fastest engine in the naive reverse benchmark among the Java-based Prolog implementations compared in [13]. Table 1 shows execution times, in seconds, for simple benchmarks on a PIII-900 MHz Linux machine. These results show that our engine is currently the fastest *complete* implementation of narrowing. In all likelihood, its speed is partially due to the elimination of the overhead paid by Curry2Java for computing with an abstract machine. In comparison with Jinni, we perform better in the **`rev`**$_{1200}$ benchmark, where the number of reduction steps is more or less the same for needed narrowing and SLD-resolution. For the **`add`** benchmark, we evaluate the goal **`add(X,Y,peano300)`** in Jinni. Due to the rules for strict equality with unification, even an optimized implementation of needed narrowing will perform at least twice as many reduction steps for **`add x y =:= peano300`** as a SLD-resolution of **`add(X,Y,peano300)`**. However, we are still faster than Jinni in this benchmark, too. Curry2Prolog and MCC are faster than our approach by a factor 8 for **`rev`** and by factor 20 for **`add`**. This is to be expected. Backtracking-based implementations are simpler and faster because they sacrifice completeness. Additionally, Curry2-Prolog is executed by the highly optimized SICStus Prolog compiler, and the abstract machine of MCC is written in C, while our implementation is executed by the JVM. We expect that if our implementation were optimized and/or coded in C, it would offer performance competitive with these incomplete systems while retaining completeness.

A factor of 8-20 speedup over Java for a C implementation is reasonable and supported by the results of [19]. The authors have shown that a C++ implementation of the Curry2Java abstract machine was more than 50 times faster than the same implementation in Java. We do not expect a similar improvement because we have already eliminated the interpretation layer of the abstract machine, and because the results of [19] were obtained with JDK 1.1 while we use JDK 1.3. The latter is more efficient. However, we are confident that there are still considerable opportunities for improving the efficiency of our implementation. We plan to work on this aspect, but only after resolving the architectural issues related to the inclusion of search and concurrency features which are discussed in the next section.

Table 1: Execution times for simple benchmarks on several FLP engines

	Ours	C2J	MCC	PAKCS	Jinni
`rev`$_{400}$	0.69	2.6			
`rev`$_{1200}$	5.5	N/A	0.69	0.68	45.9
`add`$_{300}$	2.1	16.2	0.12	0.09	2.5

7. FURTHER EXTENSIONS

A very interesting feature for modern functional logic languages is encapsulated search [20]. Although this feature is not yet included in our implementation, our architecture is ready to accommodate it.

Encapsulated search uses a *search operator* to explicitly control different branches of a non-deterministic computation. It relies on a data structure to encode search goals and their non-deterministic splitting. This structure supports different search strategies and controls failures. Additionally, it prevents non-determinism from splitting the global computation, which is crucial to avoid conflicts with irreversible I/O operations. Complete encapsulated search strategies rely on another key feature, committed choice [24]. Losely speaking, different branches of a computation are evaluated in parallel. When one branch finds a solution, the other branches are discarded. The combination of the search operator and committed choice is necessary for implementing *complete* encapsulated search strategies [31, 20].

To ensure completeness, it is necessary to distinguish between local and global computations in three aspects. Non-deterministic steps of a goal cannot split the global computation. If a goal either fails or succeeds we must take special actions like encoding the result in a data structure or killing some other local computations (if the committed choice is involved). The third aspect concerns variable binding. Global variables, i.e., variables not introduced by search, cannot be bound by search, because different local computations can share a global variable. Different bindings of any such variable in local computations would be inconsistent in the global computation.

We know of only two attempts at narrowing-based implementations of encapsulated search. The Münster Curry Compiler implements the search operator, but it lacks committed choice. Thus, complete search algorithms cannot be coded in MCC. Curry2Java provides both the search operator and committed choice. Curry2Java employs threads for non-deterministic search, thus it faces the problem of integrating local search into an architecture which was not designed for explicit control. This problem has not yet been solved and its solution is not near.

In our architecture, it should be much easier to implement and integrate encapsulated search and committed choice because we have explicit and direct control of computation. Computations are designed to be nested, which eases introducing local computations. A crucial aspect of the implementation of encapsulated search is the distinction between local and global variables. This can be solved by making a computation log a variable as local when it is introduced inside this computation, e.g., by evaluating a local declaration of a free variable. This method was successfully used in Curry2Java.

The implementation of committed choice should be even easier than the search operator. While the search operator must encode all possible branches after a non-deterministic step in a data structure, committed choice can discard all other possibilities if it has found one successful branch. If the computation encounters a function call which should be evaluated by committed choice, a new queue of computations (Section 4.5) is created for goals to be evaluated in parallel. These local computations follow the rules for local variable bindings described above. When a non-deterministic step occurs in one of the computations, we just add new computations to the queue. This local queue is similar to the global one, except that when a computation succeeds, we delete the entire local queue and continue with a single goal. Thus, the explicit control of computation in our architecture allows us to implement both encapsulated search and committed choice with modest extensions.

Another advantage of our model is the potential for an *efficient* complete encapsulated search strategy. The search operator and committed choice must be combined to realize a complete encapsulated search strategy, but such algorithms are highly inefficient because committed choice will repeatedly spawn many local computations which are soon killed again. In our model, we could realize an efficient algorithm with minimal effort. We just need to create a local queue of computations in which we evaluate a search goal. In contrast to the global queue, we need to take care of local variable bindings, and we must return the solutions as a list of search goals, which can be done lazily. However, these are all just changes to the global queue. Thus, we could provide a lazy, efficient and complete encapsulated search algorithm which avoids the inefficiency of combining search operators and committed choice, i.e., the repeated spawning and killing of local computations.

8. CONCLUSION

We described the architecture of an engine for functional logic computations. Our engine implements an efficient, sound and complete narrowing strategy, *INS*, and integrates this strategy with other features, e.g., residuation and and-parallelism, desirable in functional logic programming. Our implementation is operationally complete, easy to extend (e.g., by external resources like constraint libraries) and general enough to be used as a back end for a variety of languages. Although our work is still evolving, simple benchmarks show that it is the fastest complete implementation of narrowing currently available: it has strong potential for further improvement in both performance and functionality.

9. ACKNOWLEDGEMENTS

This research has been partially supported by the NSF DAAD under grant INT-9981317 and the German Research Council (DFG) under grant Ha 2457/1-2.

10. AVAILABILITY

Our implementation and supporting material is available under the GNU Public License at `http://nmind.cs.pdx.edu`.

11. REFERENCES

[1] H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.

[2] E. Albert, M. Hanus, and G. Vidal. A practical partial evaluator for a multi-paradigm declarative language. In *Proc. 5th Intl. Symposium on Functional and Logic Programming (FLOPS '01)*, pages 326–342. Springer LNCS 2024, 2001.

[3] S. Antoy. Definitional trees. In *Proc. 3rd Intl. Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.

[4] S. Antoy. Needed narrowing in Prolog. Technical report 96-2, Portland State University, 1996.

[5] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. Intl. Conference on Algebraic and Logic Programming (ALP '97)*, pages 16–30. Springer LNCS 1298, 1997.

[6] S. Antoy. Constructor-based conditional narrowing. In *Principles and Practice of Declarative Programming, (PPDP'01)*, Sept. 2001. (In this volume).

[7] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal ACM*, 47(4):776–822, 2000. Previous version in *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, 1994.

[8] S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. 3rd Intl. Workshop on Frontiers of Combining Systems (FroCoS '00)*, pages 171–185. Springer LNCS 1794, 2000.

[9] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[10] 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)*, pages 311–326. MIT Press, 1988.

[11] M. Chakravarty and H. Lock. Towards the uniform implementation of declarative languages. *Computer Languages*, 23(2-4):121–160, 1997.

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

[13] E. Denti, A. Omicini, and A. Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In *Practical Aspects of Declarative Languages (PADL)*, pages 184–198. Springer LNCS 1990, 2001.

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

[15] M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth Intl. Workshop*

on Logic Program Synthesis and Transformation, pages 252–266. Springer LNCS 1048, 1995.

[16] M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[17] M. Hanus, S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.3: The Portland Aachen Kiel Curry System User Manual. Technical report, University of Kiel, Germany, 2000. Available at `http://www.informatik.uni-kiel.de/~pakcs`.

[18] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[19] M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, 1999(6), 1999.

[20] M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint Intl. Symposium PLILP/ALP '98)*, pages 374–390. Springer LNCS 1490, 1998.

[21] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.

[22] T. Hortala-Gonzalez and E. Ullan. An abstract machine based system for a lazy narrowing calculus. In *Proc. 5th Intl. Symposium on Functional and Logic Programming (FLOPS '01)*, pages 216–232. Springer LNCS 2024, 2001.

[23] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–443. MIT Press, 1991.

[24] S. Janson. *AKL – A Multiparadigm Programming Language*. PhD thesis, Swedish Institute of Computer Science, 1994.

[25] J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3):1–49, 1999.

[26] R. Loogen. Relating the implementation techniques of functional and functional logic languages. *New Generation Computing*, 11:179–215, 1993.

[27] R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. 5th Intl. Symposium on Programming Language Implementation and Logic Programming*, pages 184–200. Springer LNCS 714, 1993.

[28] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proceedings of RTA '99*, pages 244–247. Springer LNCS 1631, 1999.

[29] W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji Intl. Symposium on Functional and Logic Programming (FLOPS '99)*, pages 100–113. Springer LNCS 1722, 1999.

[30] A. Middeldorp. Call by need computations to root-stable form. In *Proc. 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 94–105, 1997.

[31] C. Schulte and G. Smolka. Encapsulated search for higher-order concurrent constraint programming. In *Proc. 1994 Intl. Logic Programming Symposium*, pages 505–520. MIT Press, 1994.

[32] J. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[33] P. Tarau. Jinni. Available at `http://www.binnetcorp.com/Jinni/`, 2001.

[34] D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.