A Unified Computation Model for Functional and Logic Programming

Michael Hanus

Informatik II, RWTH Aachen

D-52056 Aachen, Germany

hanus@informatik.rwth-aachen.de

Abstract

We propose a new computation model which combines the operational principles of functional languages (reduction), logic languages (non-deterministic search for solutions), and integrated functional logic languages (residuation and narrowing). This computation model combines efficient evaluation principles of functional languages with the problem-solving capabilities of logic programming. Since the model allows the delay of function calls which are not sufficiently instantiated, it also supports a concurrent style of programming. We provide soundness and completeness results and show that known evaluation principles of functional logic languages are particular instances of this model. Thus, our model is a suitable basis for future declarative programming languages.

1 Introduction

Declarative programming languages support the development of reliable software by providing abstraction facilities supported by closely underlying formal calculi. This simplifies the development and use of program transformation, analysis, and verification tools. Unfortunately, the field of declarative programming is split into two main fields, namely functional and logic programming. Although the objectives are similar, the methods are often different due to the different underlying computation models. In order to combine the advantages of both models into one language, there is a growing interest in integrating functional and logic languages (see [16] for a survey). Integrated functional logic languages offer features from functional programming (reduction of nested expressions, lazy evaluation, higher-order functions) and logic programming (logical variables, partial data structures, built-in search). Compared to purely functional languages, they have more expressive power due to the use of logical variables and built-in search mechanisms. Compared to purely logic languages, they have more efficient evaluation mechanisms due to the (deterministic!) reduction of functional expressions (see [14, 18] for discussions about the efficiency improvements of functional logic languages in comparison to Prolog). However, the research on integrating functional and logic programming has led to different computation models, since there are different ways to combine the search facilities of logic programming with efficient evaluation principles of functional programming. The most promising operational principles are residuation and narrowing.

Residuation is based on the idea to delay function calls until they are ready for deterministic evaluation. The residuation principle is used, for instance, in the languages Escher [21, 22], Le Fun [2], Life [1], NUE-Prolog [27], and Oz [29]. Since the residuation principle evaluates function calls by deterministic reduction steps, non-deterministic search must be encoded by predicates [1, 2, 27] or disjunctions [21].

Example 1.1 Consider the following rewrite rules and clauses defining the natural numbers and their addition:

$$\begin{array}{ll} 0 + X \to X & nat(0) \\ s(X) + Y \to s(X + Y) & nat(s(X)) \Leftarrow nat(X) \end{array}$$

A function call like $Z + 0$ is delayed since the first argument is not sufficiently instantiated. Thus, the following goal, consisting of an equation and a predicate call, is evaluated by delaying and awakening the function call (the function or predicate evaluated in the next step is underlined):

$$\begin{array}{lll} Z + 0 \doteq s(0) \wedge \underline{nat}(Z) & & \\ \quad \vdash_{\{Z \mapsto s(X)\}} & s(X)\underline{+}0 \doteq s(0) \wedge nat(X) & \\ \quad \vdash_{\{\}} & s(X + 0) \doteq s(0) \wedge \underline{nat}(X) & \\ \quad \vdash_{\{X \mapsto 0\}} & s(0\underline{+}0) \doteq s(0) & \\ \quad \vdash_{\{\}} & s(0) \doteq s(0) & \end{array}$$

In Proc. of the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), pp. 80–93, Paris, 1997.

Thus, the solution computed by residuation is $\{Z \mapsto s(0)\}$. □

The residuation principle preserves the deterministic nature of functions and provides concurrent computations with synchronization on logical variables. Unfortunately, it is incomplete, i.e., it is unable to compute solutions if arguments of functions are not sufficiently instantiated during the computation. Therefore, functional logic languages with a complete operational semantics, e.g., ALF [13], Babel [26], K-Leaf [11], LPG [7], or SLOG [10], are based on *narrowing*, a combination of the reduction principle of functional languages with unification for parameter passing. A narrowing step instantiates goal variables so that function calls become reducible.

Example 1.2 Consider the rules for addition of the previous example. To solve the equation $Z + 0 \doteq s(0)$, Z is instantiated to $s(X)$ and the second rule is applied. In a further step, X is instantiated to 0 and the first rule is applied:

$$\begin{array}{lll} Z + 0 \doteq s(0) & \leadsto_{\{Z \mapsto s(X)\}} & s(X + 0) \doteq s(0) \\ & \leadsto_{\{X \mapsto 0\}} & s(0) \doteq s(0) \end{array}$$

Thus, $\{Z \mapsto s(0)\}$ is the computed solution. Note that residuation cannot compute this solution but flounders on this goal. □

Narrowing provides completeness in the sense of functional programming (normal forms are computed if they exist) as well as logic programming (solutions are computed if they exist). In order to restrict the search space, to avoid unnecessary computations, and to support typical functional programming techniques, most recent work has concentrated on *lazy narrowing* strategies [4, 5, 11, 15, 20, 23, 25, 26, 28]. Among these, there is one strategy, called *needed narrowing* [4], which is optimal w.r.t. the length of derivations and the number of computed solutions (for a restricted class of programs, called inductively sequential systems, see Section 5.1).

Since each of these computation models have their own advantages, our aim is to combine them into a single framework. The difficulty in this combination is the fact that residuation rewrites functional expressions in a deterministic way and puts all non-determinism into the level of predicates, whereas narrowing allows non-deterministic steps also at the level of functions. Moreover, recent residuation-based languages, like Escher [21] or Oz [29], represent non-deterministic (don't know) choices by explicit disjunctions, whereas narrowing is usually defined with implicit disjunctions as in classical logic programming. A disadvantage of coding all non-determinism by predicates in residuation-based approaches is the loss of information about functional dependencies (which is, on the other hand, the well-known advantage of adding functions to logic programming). Therefore, our computation model represents non-deterministic choices by explicit disjunctions, and narrowing steps are applied to subterms but may generate new disjunctions. We show that narrowing and residuation are special cases of this framework. Hence, residuation can be viewed as a restriction of the potential non-determinism in the evaluation of partially instantiated function calls.

In the next section, we recall basic notions from functional logic programming. In Section 3, we specify our computation model. Soundness and completeness results are presented in Section 4. Section 5 compares this strategy with well-known strategies for functional logic programs. The compilation of a concrete programming language into our computation model is shown in Section 6. Finally, Section 7 discusses possible extensions of the basic model.

2 Preliminaries

We assume familiarity with basic notions of term rewriting [9] and functional logic programming [16]. 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. There is at least one sort *Bool* containing the 0-ary Boolean constructors *true* and $false$. The set of *terms* and *constructor terms* with *variables* (e.g., X, Y, Z) from $\mathcal{X}$ are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$. $\mathcal{V}ar(t)$ denotes the set of variables occurring in a term t. A term t is *ground* if $\mathcal{V}ar(t) = \emptyset$. A *pattern* is a term of the form $f(t_1, \ldots, t_n)$ where $f/n \in \mathcal{F}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A term is *operation-rooted* if it has an operation symbol at the root, otherwise it is called a *head normal form*. $root(t)$ denotes the symbol at the root of the term t. 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 [9] for details).

We denote by $X_1 = t_1, \ldots, X_n = t_n$ the *substitution* σ with $\sigma(X_i) = t_i$ $(i = 1, \ldots, n)$ and $\sigma(X) = X$ for all other variables. *id* denotes the identity substitution. 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 term t' is an *instance* of t if there is a substitution σ with $t' = \sigma(t)$. A *unifier* of two term s and t is a substitution σ with $\sigma(s) = \sigma(t)$.

A *functional logic program* is a *term rewriting system* $\mathcal{R}$ consisting of a set of *rewrite rules* $l \to r$ where

l is a linear[1] pattern and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. l and r are called *left-hand side* and *right-hand side*, respectively.[2] A rewrite rule is called a *variant* of another rule if it is obtained by a unique replacement of variables by other variables. In order to ensure well-definedness of functions (i.e., to ensure the confluence of the rewrite system), we require that $\mathcal{R}$ contains only trivial overlaps, i.e., if $l_1 \to r_1$ and $l_2 \to r_2$ are variants of rewrite rules and σ is a unifier for l_1 and l_2, then $\sigma(r_1) = \sigma(r_2)$ (*weak orthogonality*).

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{\mathcal{R}} s$ if there exist a position p in t, a rewrite rule $l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. A term t is called *irreducible* or in *normal form* if there is no term s with $t \to_{\mathcal{R}} s$.

Since we do not require terminating rewrite systems, normal forms may not exist. Therefore, the predicate $\doteq$ used in some examples is defined, like in functional languages, as the *strict equality* on terms, i.e., the equation $t_1 \doteq t_2$ is satisfied if t_1 and t_2 are reducible to the same ground constructor term. Since the strict equality can be defined as a binary Boolean function by a set of orthogonal rewrite rules, we do not consider it in a special way (see [4, 11, 26] for more details about strict equality).

Due to the presence of free variables in expressions, an expression may be reduced to different values by binding the free variables to different terms. In functional programming, one is interested in the computed *value*, whereas logic programming emphasizes the different bindings (*answers*). Thus, we define for our integrated framework an *answer expression* as a pair $\sigma \,[\!]\, e$ consisting of a substitution σ (the answer computed so far) and an expression e. An answer expression $\sigma \,[\!]\, e$ is *solved* if e is a constructor term. We sometimes omit the identity substitution in answer expressions, i.e., we write e instead of $id \,[\!]\, e$ if it is clear from the context.

Since more than one answer may exist for expressions containing free variables, in general, initial expressions are reduced to disjunctions of answer expressions. Thus, a *disjunctive expression* is a (multi-)set of answer expressions $\{\sigma_1 \,[\!]\, e_1, \ldots, \sigma_n \,[\!]\, e_n\}$, sometimes written as $(\sigma_1 \,[\!]\, e_1) \vee \ldots \vee (\sigma_n \,[\!]\, e_n)$. The set of all disjunctive expressions is denoted by $\mathcal{D}$.

3 A Unified Computation Model

A single *computation step* performs a reduction in exactly one unsolved expression of a disjunction (e.g., in the leftmost unsolved answer expression in Prolog-like implementations). If the computation step is deterministic, the expression is reduced to a new one. If the computation step is non-deterministic, the expression is replaced by a disjunction of new expressions. The precise behavior depends on the function calls occurring in the expression. For instance, consider the following rules defining a simple predicate:

$$\begin{array}{lcl} p(a) & \to & true \\ p(b) & \to & true \end{array}$$

The result of evaluating the goal $p(a)$ (*goals* are Boolean expressions) is $true$, whereas the goal $p(X)$ is evaluated to the disjunctive expression

$$(X = a \,[\!]\, true) \vee (X = b \,[\!]\, true) \ .$$

In order to avoid superfluous computation steps and to apply programming techniques of modern functional languages, nested expressions are evaluated lazily, i.e., the leftmost outermost function call could be selected in a computation step. However, there are functions which should only be evaluated if the arguments are sufficiently instantiated for deterministic reduction (e.g., test predicates). Thus, some insufficiently instantiated outermost function calls must be delayed. To provide a precise definition of the computational behavior of different functions, we use definitional trees.[3] A definitional tree is a hierarchical structure containing all rules of a defined function. $\mathcal{T}$ is a *definitional tree with pattern* π iff the depth of $\mathcal{T}$ is finite and one of the following cases holds:

- $\mathcal{T} = rule(l \to r)$, where $l \to r$ is a variant of a rule such that $l = \pi$.
- $\mathcal{T} = branch(\pi, o, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where o is an occurrence of a variable in π, $r \in \{rigid, flex\}$, $c_1, \ldots, c_k$ are different constructors of the sort of $\pi|_o$, for some $k > 0$, and, for all $i = 1, \ldots, k$, $\mathcal{T}_i$ is a definitional tree with pattern $\pi[c_i(X_1, \ldots, X_n)]_o$, where n is the arity of c_i and $X_1, \ldots, X_n$ are new variables.
- $\mathcal{T} = or(\mathcal{T}_1, \mathcal{T}_2)$, where $\mathcal{T}_1$ and $\mathcal{T}_2$ are definitional trees with pattern π.
- $\mathcal{T} = and(\mathcal{T}_1, \mathcal{T}_2)$, where $\mathcal{T}_1$ and $\mathcal{T}_2$ are definitional trees with pattern π.[4]

A *definitional tree of an n-ary function f* is a definitional tree $\mathcal{T}$ with pattern $f(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$

[1] A term is called *linear* if it does not contain multiple occurrences of one variable.

[2] We consider only unconditional rewrite rules in the main part of this paper. An extension to conditional rules is described in Section 7.2.

[3] Our notion is influenced by Antoy's work [3], but here we use an extended form of definitional trees.

[4] For the sake of simplicity, we consider only binary *or* and *and* nodes. The extension to such nodes with more than two subtrees is straightforward.

are distinct variables, such that for each rule $l \to r$ with $l = f(t_1, \ldots, t_n)$ there is a node $rule(l' \to r')$ in $\mathcal{T}$ with l variant of l'.[5] In the following, we write $pattern(\mathcal{T})$ for the pattern of a definitional tree $\mathcal{T}$, and DT for the set of all definitional trees.

Intuitively, a definitional tree of a function specifies the strategy to evaluate a call to this function. If the tree is a *rule* node, we apply the rule. If it is a *branch* node, it is necessary to evaluate the subterm at the specified position to head normal form in order to commit to one of the branches. If the actual value at this position is a logical variable, there are two possibilities. If the branch node is *rigid*, we delay the evaluation of this function call until the variable has been bound and proceed by evaluating some other function call. This corresponds to residuation [2, 21, 22]. If the branch node is of type $flex$, we instantiate the variable to the different constructors in the patterns of the subtrees, i.e., we have a non-deterministic step resulting in a disjunction. This corresponds to needed narrowing [4].

An *or* node in a definitional tree characterizes the situation where there is no unique argument to perform a branch. In this case we have a disjunction with two possibilities to proceed. An *and* node in a definitional tree specifies the necessity to evaluate more than one argument position. The corresponding operational behavior is an attempt to evaluate one of these arguments. If this is not possible since the function calls in this argument are delayed, we proceed by trying to evaluate the other argument. This is the generalization of concurrent computation models for residuating logic programs [1, 2, 29] to functional logic programs.

It is always possible to construct a definitional tree without *and* nodes for each function (concrete algorithms are described in [3, 23] and in Section 6). The *and* nodes are included only if concurrent computations should be allowed. To clarify the notion of definitional trees, we provide some examples in the following.

Example 3.1 The predicate "less than or equal to" for natural numbers can be specified by the following rules:

$$\begin{array}{rcl} 0 \leq X & \to & true \\ s(X) \leq 0 & \to & false \\ s(X) \leq s(Y) & \to & X \leq Y \end{array}$$

The definitional tree

$$\begin{array}{l} branch(X_1 \leq X_2, 1, flex, \\ \quad rule(0 \leq X_2 \to true), \\ \quad branch(s(X) \leq X_2, 2, flex, \\ \quad\quad rule(s(X) \leq 0 \to false), \\ \quad\quad rule(s(X) \leq s(Y) \to X \leq Y))) \end{array}$$

specifies needed narrowing [4] as the evaluation strategy for this function, i.e., to evaluate a call $t_1 \leq t_2$, we always evaluate t_1 to some head normal form since the first branch is performed on the first argument. If t_1 gets the value 0, we can apply the first rule, otherwise (if the value is $s(\cdots)$) t_2 must also be evaluated in order to apply some rule. □

Evaluation with definitional trees has some similarities to pattern matching in lazy functional languages like Haskell or Miranda [30]. Note, however, that definitional trees provide more powerful evaluation strategies than the simpler left-to-right pattern matching in current functional languages.

Example 3.2 Consider the rules

$$\begin{array}{rcl} f(0,0) & \to & 0 \\ f(X, s(N)) & \to & 0 \end{array}$$

and a non-terminating function $\bot$. Then the function call $f(\bot, s(0))$ is reduced to 0 using the definitional tree

$$\begin{array}{l} branch(f(X_1, X_2), 2, flex, \\ \quad branch(f(X_1, 0), 1, flex, rule(f(0,0) \to 0)), \\ \quad rule(f(X_1, s(N)) \to 0)) \ , \end{array}$$

whereas Miranda or Haskell do not terminate. In general, evaluation with definitional trees containing only *branch* and *rule* nodes always computes a normal form if it exists [3]. □

If there is no unique case distinction on some argument in the left-hand sides of the rewrite rules, it is necessary to include *or* nodes:

Example 3.3 Consider the rules

$$\begin{array}{rcl} 0 * X & \to & 0 \\ X * 0 & \to & 0 \end{array}$$

defining multiplication with 0. Then

$$\begin{array}{l} or(branch(X_1 * X_2, 1, rigid, rule(0 * X_2 \to 0)), \\ \quad branch(X_1 * X_2, 2, rigid, rule(X_1 * 0 \to 0))) \end{array}$$

is a definitional tree for this function. It specifies that $t_1 * t_2$ is reducible to 0 if one of the arguments is (reducible to) 0. Due to the *rigid* branches, the evaluation of the term $X * Y$ suspends. □

To provide concurrent computations, functions must be specified by definitional trees containing *and* nodes. In functional logic languages based on residuation (e.g., Escher [21], Le Fun [2], or LIFE [1]), functions are always deterministically evaluated or suspended, and non-determinism is covered by predicates. This behavior can easily be specified by *rigid* declarations for all non-Boolean functions and $flex$ declarations for Boolean functions. If all function calls are suspended

[5] To ensure completeness, we have to require that the sets of rules in the different subtrees of an *and* node are identical. Since we use *and* nodes only in conjunction with *rigid* branches in our examples, we can omit this requirement to simplify our examples.

in the leftmost subgoal, the computation proceeds by evaluating the next subgoal. This strategy can be simply specified by defining the conjunction as a Boolean function with a particular definitional tree:

Example 3.4 The conjunction is defined by

$$\begin{array}{rcl} true \wedge X & \to & X \\ X \wedge true & \to & X \end{array}$$

and the definitional tree

$$\begin{array}{l} and(branch(X_1 \wedge X_2, 1, rigid, rule(true \wedge X_2 \to X_2)), \\ \quad branch(X_1 \wedge X_2, 2, rigid, rule(X_1 \wedge true \to X_1))) \end{array}$$

Due to the *and* node in this tree, a goal of the form $t_1 \wedge t_2$ is evaluated by an attempt to evaluate t_1. If the evaluation of t_1 suspends, an evaluation step is applied to t_2. If a variable responsible to the suspension of t_1 was bound during the last step, the left expression will be evaluated in the subsequent step. A concurrent computation based on this definition of $\wedge$ has been shown in Example 1.1. □

After this informal presentation of our computation model, we provide a precise definition using the functions

$$cs : \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X}) \to \mathcal{D} \cup \{suspend\}$$

and

$$cst : \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X}) \times DT \to \mathcal{D} \cup \{suspend\} \ .$$

The function cs performs a single computation step on a term t. It computes a disjunction of answer expressions or the special constant *suspend* indicating that no reduction is possible in t. As shown in Figure 1, cs attempts to apply a reduction step to the leftmost outermost function symbol in t by the use of cst which is called with the appropriate subterm and the definitional tree for the leftmost outermost function symbol. cst is defined by a case distinction on the definitional tree. If it is a *rule* node, we apply this rule. If the definitional tree is an *and* node, we try to evaluate the first branch and, if this is not possible due to the suspension of all function calls, the second branch.[6] An *or* node produces a disjunction. To ensure completeness, we have to suspend the entire disjunction if one disjunct suspends (see Example 3.5 below). For a similar reason, we cannot commit to a disjunct which does not bind variables but we have to consider both alternatives (see [5] for a counter-example). The most interesting case is a *branch* node. Here we have to branch on the value of the top-level symbol at the selected position. If the symbol is a constructor, we proceed with the appropriate definitional subtree, if possible. If it is a function symbol, we proceed by evaluating this subterm. If it is a variable, we either suspend (if the branch is *rigid*) or instantiate the variable to the different constructors. The auxiliary function *compose* composes the result of a computation step with a substitution, and *replace* puts a possibly disjunctive expression into a subterm:

$$\begin{array}{l} compose(t, \mathcal{T}, \sigma) = \\ \quad \begin{cases} \{\sigma \,[]\, t\} & \text{if } cst(t, \mathcal{T}) = suspend \\ \{\sigma_1 \circ \sigma \,[]\, t_1, \ldots, \sigma_n \circ \sigma \,[]\, t_n\} & \\ \qquad \text{if } cst(t, \mathcal{T}) = \{\sigma_1 \,[]\, t_1, \ldots, \sigma_n \,[]\, t_n\} \end{cases} \\ \\ replace(t, o, suspend) = suspend \\ replace(t, o, \{\sigma_1 \,[]\, t_1, \ldots, \sigma_n \,[]\, t_n\}) = \\ \qquad \{\sigma_1 \,[]\, \sigma_1(t)[t_1]_o, \ldots, \sigma_n \,[]\, \sigma_n(t)[t_n]_o\} \end{array}$$

The overall computation strategy is a transformation $\xrightarrow{RN}$ on disjunctive expressions. It takes a disjunct $\sigma \,[]\, e$ not in solved form and computes $cs(e)$. If $cs(e) = suspend$, then the computation of this expression *flounders* and we cannot proceed (i.e., this expression is not solvable). If $cs(e)$ is a disjunctive expression, we substitute it for $\sigma \,[]\, e$ composed with the old answer substitution.

The following example shows that it is necessary to suspend a computation if one alternative in an *or* node suspends.

Example 3.5 Consider the predicate p defined by

$$\begin{array}{rcl} p(a, X) & \to & true \\ p(X, b) & \to & true \end{array}$$

Let

$$\mathcal{T}_p = or(\mathcal{T}_1, \mathcal{T}_2)$$

with

$$\mathcal{T}_1 = branch(p(X_1, X_2), 1, rigid, rule(p(a, X_2) \to true))$$

and

$$\mathcal{T}_2 = branch(p(X_1, X_2), 2, flex, rule(p(X_1, b) \to true))$$

be the definitional tree of p. Then $cst(p(X, X), \mathcal{T}_p) = suspend$ since $cst(p(X, X), \mathcal{T}_1) = suspend$ and $cst(p(X, X), \mathcal{T}_2) = (X = b \,[]\, true)$. Now consider the conjunction $p(X, X) \wedge X \doteq a$ ($\wedge$ is defined in Example 3.4). Since $p(X, X)$ suspends, the conjunction is evaluated by reducing the subgoal $X \doteq a$ to the answer expression $X = a \,[]\, true$. Thus, the conjunction is solved as follows:

$$\begin{array}{rcl} p(X, X) \wedge X \doteq a & \xrightarrow{RN} & X = a \,[]\, p(a, a) \wedge true \\ & \xrightarrow{RN} & X = a \,[]\, true \wedge true \\ & \xrightarrow{RN} & X = a \,[]\, true \end{array}$$

[6] For the sake of simplicity, we choose a simple sequential strategy for concurrent computations. However, it is also possible to provide a more sophisticated strategy with a fair selection of threads, e.g., as in Oz [29].

Computation step for a single (unsolved) expression t:

$$cs(X) = suspend \quad \text{for all variables } X$$

$$cs(f(t_1,\ldots,t_n)) = cst(f(t_1,\ldots,t_n),\mathcal{T}) \quad \text{if } \mathcal{T} \text{ is a definitional tree for } f$$

$$cs(c(t_1,\ldots,t_n)) = \begin{cases} replace(c(t_1,\ldots,t_n),k,cs(t_k)) & \text{if } cs(t_i)=suspend,\ i=1,\ldots,k-1, \text{ and } cs(t_k)\neq suspend \\ suspend & \text{if } cs(t_i)=suspend,\ i=1,\ldots,n \end{cases}$$

Computation step for an operation-rooted term t:

$$cst(t, rule(l \to r)) = \{id \,[\!]\, \sigma(r)\} \quad \text{if } \sigma \text{ is a substitution with } \sigma(l)=t$$

$$cst(t, and(\mathcal{T}_1,\mathcal{T}_2)) = \begin{cases} cst(t,\mathcal{T}_1) & \text{if } cst(t,\mathcal{T}_1)\neq suspend \\ cst(t,\mathcal{T}_2) & \text{otherwise} \end{cases}$$

$$cst(t, or(\mathcal{T}_1,\mathcal{T}_2)) = \begin{cases} cst(t,\mathcal{T}_1)\cup cst(t,\mathcal{T}_2) & \text{if } cst(t,\mathcal{T}_1)\neq suspend\neq cst(t,\mathcal{T}_2) \\ suspend & \text{otherwise} \end{cases}$$

$$cst(t, branch(\pi,o,r,\mathcal{T}_1,\ldots,\mathcal{T}_k)) = \begin{cases} cst(t,\mathcal{T}_i) & \text{if } t|_o = c(t_1,\ldots,t_n) \text{ and } pattern(\mathcal{T}_i)|_o = c(X_1,\ldots,X_n) \\ \emptyset & \text{if } t|_o = c(t_1,\ldots,t_n) \text{ and } pattern(\mathcal{T}_i)|_o \neq c(\cdots), i=1,\ldots,k \\ suspend & \text{if } t|_o = X \text{ and } r = rigid \\ \bigcup_{i=1}^{k} compose(\sigma_i(t),\mathcal{T}_i,\sigma_i) & \text{if } t|_o = X,\ r=flex, \text{ and } \sigma_i=\{X\mapsto pattern(\mathcal{T}_i)|_o\} \\ replace(t,o,cs(t|_o)) & \text{if } t|_o = f(t_1,\ldots,t_n) \end{cases}$$

Derivation step for a disjunctive expression:

$$(\sigma \,[\!]\, e) \vee D \quad \stackrel{RN}{\longrightarrow} \quad (\sigma_1 \circ \sigma \,[\!]\, e_1) \vee \ldots \vee (\sigma_n \circ \sigma \,[\!]\, e_n) \vee D$$

if $\sigma \,[\!]\, e$ is unsolved and $cs(e) = \{\sigma_1 \,[\!]\, e_1, \ldots, \sigma_n \,[\!]\, e_n\}$

Figure 1: Operational semantics of concurrent functional logic programming

However, if we ignore a suspension in one alternative of an *or* node and commit to the other alternative, we could evaluate $p(X,X)$ but loose the only answer:

$$\begin{array}{rcl} p(X,X) \wedge X \doteq a & \Rightarrow & X = b \,[\!]\, true \wedge b \doteq a \\ & \stackrel{RN}{\longrightarrow} & X = b \,[\!]\, b \doteq a \\ & \stackrel{RN}{\longrightarrow} & X = b \,[\!]\, false \end{array}$$

This explains our definition of a computation step in case of *or* nodes. □

4 Soundness and Completeness

We relate the results computed by our operational model to a rewriting semantics w.r.t. functional logic programs. This is sufficient since the equivalence of rewriting and (model-theoretic) validity is known for a rather general class of functional logic programs [12]. $\to_{\mathcal{R}}$ denotes the standard rewrite relation w.r.t. the given functional logic program $\mathcal{R}$ (as defined in Section 2). The next theorem shows the soundness of our computation model w.r.t. $\to_{\mathcal{R}}$ (we denote by $\to^*$ the transitive and reflexive closure of a binary relation $\to$).

Theorem 4.1 (Soundness) *If there is a derivation sequence*

$$id \,[\!]\, e \quad \stackrel{RN}{\longrightarrow}^* \quad \{\sigma_1 \,[\!]\, e_1, \ldots, \sigma_n \,[\!]\, e_n\}$$

then $\sigma_i(e) \to^*_{\mathcal{R}} e_i$ *for* $i = 1,\ldots,n$.

For the completeness we cannot expect a result similarly to logic programming (i.e., every correct answer is subsumed by a computed one) since expressions may be suspended due to the insufficient instantiation of arguments. Moreover, we cannot prove something like the inversion of the previous theorem since $\stackrel{RN}{\longrightarrow}$ evaluates only in a lazy manner so that some rewrite steps w.r.t. $\to_{\mathcal{R}}$ do not correspond to steps computed by $\stackrel{RN}{\longrightarrow}$. However, we can show that some part of a solution is

computed by $\stackrel{RN}{\longrightarrow}$, where a *solution* is a substitution that enables the reduction of an expression e to some constructor c (to see that this notion of solution covers the standard notion, take an equation $t_1 \doteq t_2$ for e and *true* for the final constructor c).

Theorem 4.2 (Completeness) *Let σ be a substitution and c a constructor such that $\sigma(e) \rightarrow^*_{\mathcal{R}} c$, and*

$$id \,[]\, e \stackrel{RN}{\longrightarrow}^* \{\sigma_1 \,[]\, e_1, \ldots, \sigma_n \,[]\, e_n\}$$

be a derivation sequence. Then there exists a substitution φ with $\sigma = \varphi \circ \sigma_i$, for some $i \in \{1, \ldots, n\}$, and $\varphi(e_i) \rightarrow^*_{\mathcal{R}} c$.

To show that the suspension of function calls is the only reason why solutions cannot be fully computed, we consider the subclass of functional logic programs where *rigid* branches do not occur. This result corresponds to completeness of lazy narrowing and requires a fair selection of unsolved disjuncts $\sigma \,[]\, e$ in the computation steps of $\stackrel{RN}{\longrightarrow}$ (i.e., every evaluable disjunct is evaluated at some time).

Theorem 4.3 (Completeness w.r.t. flexible functions) *Let $\mathcal{R}$ be a functional logic program where all branch nodes in the definitional trees are flexible, σ a substitution and c a constructor such that $\sigma(e) \rightarrow^*_{\mathcal{R}} c$. Then there exists a derivation sequence*

$$id \,[]\, e \stackrel{RN}{\longrightarrow}^* \{\sigma_1 \,[]\, e_1, \ldots, \sigma_n \,[]\, e_n\}$$

with $e_i = c$ and $\sigma = \varphi \circ \sigma_i$, for some $i \in \{1, \ldots, n\}$ and substitution φ.

5 Related Operational Models

The operational semantics presented in Section 3 should combine execution principles from functional programming and (concurrent) logic programming in a coherent way. In this section we show that well-known operational models can be derived from our model using particular restrictions expressed by definitional trees.

5.1 Narrowing

Narrowing combines reduction of functional expressions with unification for parameter passing. In order to restrict the search space and to avoid unnecessary evaluations, *lazy narrowing* strategies have been proposed. To specify the exact evaluation strategy, definitional trees have been used [4, 5, 18, 20, 23]. An optimal strategy can be obtained for a particular subclass of functional logic programs. A function is *inductively sequential* if it has a definitional tree without *or* and *and* nodes. A program is inductively sequential if all defined functions are inductively sequential. *Needed narrowing* [4] is an optimal lazy narrowing strategy for inductively sequential programs.

Proposition 5.1 *If the definitional tree for each function contains only rule and branch nodes and each branch node is flexible, then the strategy $\stackrel{RN}{\longrightarrow}$ is equivalent to needed narrowing.*[7]

For instance, the goal $X \leq X + X$ is reduced as follows ($\leq$ is defined in Example 3.1):

$$X \leq X + X \stackrel{RN}{\longrightarrow}$$
$$(X = 0 \,[]\, true) \vee (X = s(X') \,[]\, s(X') \leq s(X' + s(X')))$$

Thus, $\stackrel{RN}{\longrightarrow}$ produces only two alternatives and avoids unnecessary evaluations in contrast to simpler lazy narrowing strategies [26].

Since needed narrowing is optimal in the length of derivations and the number of computed solutions [4], $\stackrel{RN}{\longrightarrow}$ is a conservative extension of an optimal evaluation strategy for functional logic programs. If the definitional trees also contain *or* nodes, $\stackrel{RN}{\longrightarrow}$ is not optimal but similar to weakly needed narrowing strategies [5, 23]. The improvement of the latter strategies by incorporating deterministic simplification steps [15] or parallel reduction steps [5] is outside the scope of this paper. Note, however, that such improvements can also be specified and implemented using definitional trees [18].

5.2 Residuation

In functional logic languages based on residuation, functions are always evaluated in a deterministic way and all non-deterministic evaluations are covered by predicates. If a function call cannot be deterministically reduced due to insufficiently instantiated arguments, the call is suspended. This behavior can easily be specified by declaring all (non-Boolean) functions with *rigid* branches in their definitional trees. By definition of $\stackrel{RN}{\longrightarrow}$, this has the effect that a function call is suspended if the value of a variable argument is needed. Thus, we obtain the operational semantics of Le Fun [2] or Life [1] (see also Section 6). Similarly, Escher's evaluation strategy [22] is also subsumed. Moreover, definitional trees provide more flexible reduction strategies than the flat `NONVAR` declarations of Escher.

[7] Since needed narrowing consists of non-deterministic steps instead of generating disjunctions, equivalence means that the disjunctions produced by $\stackrel{RN}{\longrightarrow}$ steps are identical to all possible needed narrowing steps.

5.3 Functional Logic Programming vs. (Concurrent) Logic Programming

It is well known that functional logic programs can be mapped into pure logic programs by flattening nested function calls [8]. Flattening has also been used to extend (logic) languages with a functional syntax [27, 29]. Moreover, in residuation-based functional logic languages, it is necessary to flatten functional expressions if arguments of functions should be instantiated by unification with rewrite rules. Unfortunately, flattening has the risk to loose functional dependencies which are necessary to provide efficient evaluation strategies.

Example 5.2 The following rules define an infinite list of natural numbers and a function $first$ to select the first elements of a list:

$$\begin{array}{rcl} from(N) & \rightarrow & [N|from(s(N))] \\ first(0,L) & \rightarrow & [] \\ first(s(N),[E|L]) & \rightarrow & [E|first(N,L)] \end{array}$$

In order to solve the goal $first(X, from(X)) \doteq []$, $\stackrel{RN}{\longrightarrow}$ instantiates X to 0 (we ignore the second alternative $X = s(N)$ since it fails after two further steps):

$$\begin{array}{ll} & first(X, from(X)) \doteq [] \\ \stackrel{RN}{\longrightarrow} & (X = 0 \,[]\, [] \doteq []) \vee (X = s(N) \,[]\, \ldots) \\ \stackrel{RN^*}{\longrightarrow} & (X = 0 \,[]\, true) \end{array}$$

On the other hand, we obtain an infinite derivation sequence if we flatten the functions into predicates, since the predicate definitions

$$\begin{array}{l} from(N,[N|L]) \Leftarrow from(s(N),L) \\ first(0,L,[]) \\ first(s(N),[E|L],[E|R]) \Leftarrow first(N,L,R) \end{array}$$

cause the following infinite resolution sequence:

$$\begin{array}{l} first(X,L,[]), from(X,L) \\ \vdash\ from(0,L) \\ \vdash\ from(s(0),L1) \\ \vdash\ from(s(s(0)),L2) \\ \vdash\ \ldots \end{array}$$

□

This example shows the advantages of functional logic programming compared to logic programming. Functional languages provide efficient and in some cases optimal evaluation strategies. Such strategies can also be used in a more general framework if functional and logic languages are integrated.

5.4 Narrowing vs. Residuation

In the field of integrating functional and logic languages, there is a long debate about the "right" operational mechanism. Most proposals can be classified as residuation-based or narrowing-based. Both techniques have its own advantages and disadvantages. Residuation combines deterministic reduction of functions with partial data structures, provides concurrent computations with synchronization on logical variables, and allows a simple connection to external functions. Unfortunately, residuation is unable to compute a solution if arguments of functions are not sufficiently instantiated during the computation. Moreover, it is not clear whether this strategy is better than Prolog's resolution strategy since there are examples where residuation has an infinite search space whereas the equivalent (flattened) Prolog program has a finite search space [17]. On the other hand, narrowing is complete and optimality results for particular strategies are known.

The existing presentations of narrowing and residuation calculi are very different. Hence, one gets the impression that it is not possible to combine the advantages of these two worlds. However, we have shown by the definition of $\stackrel{RN}{\longrightarrow}$ that this need not be the case. In fact, the difference between narrowing and residuation in the definition of $\stackrel{RN}{\longrightarrow}$ is just one bit: if a branch node is flexible, narrowing steps are computed, and if a branch node is rigid, a function call may residuate. This provides also a technique to avoid the incompleteness of residuation: if an expression is completely evaluable by residuation (a sufficient criteria for this property can be found in [17]), we can evaluate it with rigid declarations, otherwise we apply narrowing to compute all solutions.

6 Generating Definitional Trees

Our computation model is based on the specification of a definitional tree for each operation. Although definitional trees are a high-level formalism to specify evaluation strategies, it is tedious to annotate each operation which its definitional tree in a concrete programming language. We discuss in this section the automatic generation of definitional trees from the left-hand sides of rewrite rules.

We assume that $\mathcal{R}$ is a functional logic program, i.e., a constructor-based weakly orthogonal term rewriting system. The generation of definitional trees for each function defined by $\mathcal{R}$ is not straightforward, since there may exist many non-isomorphic definitional trees for a single function representing different evaluation strategies. Thus, a concrete programming language where definitional trees can be omitted demands for a default

strategy to generate definitional trees. This is similar to other programming languages with pattern matching. For instance, lazy functional languages like Haskell or Miranda perform pattern matching from left to right [30]. Antoy [3] describes the generation of "bushy" definitional trees with a high number of *or* nodes, whereas the demand driven computation strategy of the functional logic language SFL [23] is based on generating *or* nodes only if it is unavoidable, which may result in a non-left-to-right pattern matching. In the following, we describe an algorithm to generate definitional trees based on the following default strategy:

1. Pattern-matching is performed from left to right.
2. *or* nodes are generated in case of a conflict between constructors and variables, i.e., if two rules have a variable and a constructor at the same position on the left-hand side.

This default strategy is reasonable since it does not generate any *or* node for most typical functional programs and is also used in current functional languages [30]. However, there are programs where this default does not generate the best possible results (see below).

To specify the algorithm, we define by

$$DP(\pi, R) = \{\ o \text{ position of a variable in } \pi \mid root(l|_o) \in \mathcal{C} \text{ for some } l \to r \in R\}$$

the set of *demanded positions* of a pattern π w.r.t. a set of rules R. For instance, the demanded positions of the pattern $X \leq Y$ w.r.t. the rules of Example 3.1 are $\{1, 2\}$ referring to the pattern variables X and Y. The *g*eneration of a definitional *t*ree for a pattern π and a set of rules R (where l is an instance of π for each $l \to r \in R$) is described by the function $gt(\pi, m, R)$ ($m \in \{flex, rigid\}$ determines the mode annotation in the generated *branch* nodes). We distinguish the following cases for gt:

1. If the position o is leftmost in $DP(\pi, R)$, $\{root(l|_o) \mid l \to r \in R\} = \{c_1, \ldots, c_k\}$ where $c_1, \ldots, c_k$ are different constructors with arities $n_1, \ldots, n_k$, and $R_i = \{l \to r \in R \mid root(l|_o) = c_i\}$, then

   $$gt(\pi, m, R) = branch(\pi, o, m, gt(\pi[c_1(X_{11}, \ldots, X_{1n_1})]_o, m, R_1), \ldots, gt(\pi[c_k(X_{k1}, \ldots, X_{kn_k})]_o, m, R_k))$$

   where X_{ij} are fresh variables. I.e., if all rules have a constructor at the leftmost demanded position, we generate a *branch* node.

2. If the position o is leftmost in $DP(\pi, R)$ and $R' = \{l \to r \in R \mid root(l|_o) \in \mathcal{C}\} \neq R$, then

   $$gt(\pi, m, R) = or(gt(\pi, m, R'), gt(\pi, m, R - R'))$$

   I.e., we generate an *or* node if the leftmost demanded position of the pattern is not demanded by the left-hand side of all rules.

3. If $DP(\pi, R) = \emptyset$ and $l \to r$ variant of some rule in R with $l = \pi$, then

   $$gt(\pi, m, R) = rule(l \to r)$$

   Note that all rules in R are variants of each other if there is no demanded position (this follows from the weak orthogonality of the rewrite system). For non-weakly orthogonal rewrite systems, which may occur if non-deterministic functions are allowed [12] or conditional rules are transformed into unconditional rules (see Section 7.2), the rules in R may not be variants. In this case we simply connect the different rules by *or* nodes.

If R is the set of all rules defining the n-ary function f, then a definitional tree for f is generated by computing $gt(f(X_1, \ldots, X_n), m, R)$. It is easy to see that this algorithm computes a definitional tree (without *and* nodes) for each function since the number of rules is reduced in each recursive call and it keeps the invariant that the left-hand sides of the current set of rules are always instances of the current pattern.

The algorithm gt generates the definitional trees shown in Examples 3.1, 3.3, and 3.5 (up to the different $flex/rigid$ annotations). It is conform with the evaluation strategy of functional languages like Haskell or Miranda, since it generates the definitional tree

$$or(branch(f(X_1, X_2), 1, rigid, branch(f(0, X_2), 2, rigid, rule(f(0, 0) \to 0))), branch(f(X_1, X_2), 2, rigid, rule(f(X_1, s(N)) \to 0)))$$

for the rules in Example 3.2. This tree is not optimal since it has a non-deterministic *or* node and always requires the evaluation of the first argument (in the first alternative) in contrast to the tree shown in Example 3.2.

We can avoid the generation of *or* nodes for inductively sequential functions if we relax the strict left-to-right evaluation strategy and select in case 1 of the definition of gt the leftmost position among all those demanded positions where *all* rules have a constructor in the left-hand side at this position. This algorithm, called gt' in the following, generates the definitional tree shown in Example 3.2.

In general, the algorithm gt' generates more powerful evaluation strategies than gt since it is ensured that, for ground terms, normal forms are always computed (if they exists) provided that the definitional trees contain only *branch* and *rule* nodes [3]. If the function definitions are *uniform* in the sense of [30], then gt and gt'

generate identical definitional trees. Since non-uniform definitions might occur in programs, it is reasonable to stick to a default strategy (like gt, i.e., left-to-right evaluation) but allow the programmer to explicitly specify the definitional tree for a function (in some nice syntactic notation). This approach is taken in Curry [19], a new declarative language intended to combine different computation models in the area of functional logic programming.

In a concrete programming language, it is also necessary to specify the modes occurring in the *branch* nodes (argument m in $gt(\pi, m, R)$), i.e., a default how to evaluate function calls with free variables as arguments. In functional logic languages based on narrowing, such arguments are instantiated to the different constructors occurring in the left-hand sides of rewrite rules, i.e., narrowing-based languages like Babel [26] or SFL [23] requires the default value $m = flex$. Functional logic languages based on residuation, like Escher [22], Le Fun [2], Life [1], or NUE-Prolog [27], do not allow any nondeterminism in function calls but only in predicate calls. Thus, these languages have a strict distinction between functions and predicates. We can obtain a similar operational behavior by generating definitional trees with $m = rigid$ for all functions and $m = flex$ for all predicates and adding the definition of conjunction as shown in Example 3.4. For instance, the function gt would generate the following definitional trees for the function $+$ and the predicate nat defined in Example 1.1:

$$\begin{array}{l} branch(X + Y, 1, \; rigid, \\ \qquad\qquad rule(0 + Y \to Y), \\ \qquad\qquad rule(s(X_1) + Y \to s(X_1 + Y))) \\ branch(nat(X), 1, flex, \\ \qquad\qquad rule(nat(0) \to true), \\ \qquad\qquad rule(nat(s(X_1)) \to nat(X_1))) \end{array}$$

These trees specify the computational behavior as shown in Example 1.1.

Although the delay of insufficiently instantiated function calls seems to be a reasonable evaluation strategy, it does not always lead to the expected results. For instance, consider the following rules defining the function *append* to concatenate two lists:

$$\begin{array}{rcl} append([], L) & \to & L \\ append([E|R], L) & \to & [E|append(R, L)] \end{array}$$

If the *branch* in the corresponding definitional tree is $rigid$, we can use this function to concatenate known lists, e.g., $append([a, b], [c, d])$ is reduced to $[a, b, c, d]$, but we cannot use this function to split a list: the goal $append(L, [b]) \doteq [a, b]$ is not reducible and the computation stops. One solution is to provide an additional definition of a split predicate (as proposed in Escher [22]) which is superfluous from a declarative point of view. Another solution is the use of a single definition by defining *append* as a predicate rather than a function. However, the necessary flat structure of predicates leads to a worse operational behavior. For instance, it is easy to verify that the goal $append(append(X, Y), Z) \doteq []$ has a finite search space w.r.t. $\xrightarrow{RN}$ if the definitional tree contains a flexible branch node. On the other hand, the corresponding flattened relational goal

$$append(X, Y, L) \wedge append(L, Z, [])$$

has an infinite search space w.r.t. the standard relational definition of list concatenation. Thus, it is often better to define functions rather than predicates even if the functions are computed in an inverted way using non-deterministic steps caused by flexible branch nodes.

7 Extensions to the Basic Model

We briefly discuss various extensions to our basic computation model.

7.1 Higher-Order Functions

One of the most important features of functional languages are higher-order functions. Our language of first-order rewrite systems can be easily extended to cover the higher-order features of existing functional languages. For this purpose it is sufficient to provide an application function $apply(F, X) \to F(X)$ where the first argument is required to be rigid, i.e., a function can be applied only if it is a known function, otherwise the application is delayed. Lambda abstractions in right-hand sides of rewrite rules can be handled in a similar way [31]. For applications beyond current higher-order functional languages, one can provide λ-abstractions as objects (i.e., also in left-hand sides of rules) and higher-order unification. It has been shown that definitional trees can also be used to control the evaluation in this case [20].

7.2 Conditional Rules

Logic programs are based on Horn clauses

$$p \Leftarrow q_1 \wedge \ldots \wedge q_n$$

which can be considered as conditional rewrite rules [6] of the form

$$p \to true \Leftarrow q_1 \wedge \ldots \wedge q_n \; .$$

Operationally, a conditional rule is applicable if the condition can be reduced to $true$ (i.e., these conditional rules belong to the class III_n in the terminology of [6]). Although conditional rules rarely occur in functional

logic programs (since the flattening of nested function calls into conjunctions of literals is not necessary in contrast to pure logic languages), conditional rules are sometimes useful to support a relational programming style as in the following example [23]:

$$\begin{array}{ll} loves(john, mary) & \to true \\ loves(mary, Y) & \to true \Leftarrow likes(Y, wine) \doteq false \\ loves(X, mary) & \to true \Leftarrow loves(mary, X) \doteq true \end{array}$$

As in the unconditional case, we have to ensure the well-definedness of functions specified by conditional rewrite rules. For this purpose, we require for each conditional rule $l \to r \Leftarrow c$ that l is a linear pattern and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ (note that we allow *extra variables* occurring in the condition c but not in the left-hand side l). However, we can relax the weak orthogonality requirement and replace it by the weaker *nonambiguity* requirement of Babel [26]. A term rewriting system $\mathcal{R}$ is *nonambiguous* if, for all rules $l_1 \to r_1 \Leftarrow c_2$ and $l_2 \to r_2 \Leftarrow c_2$ from $\mathcal{R}$ and unifier σ for l_1 and l_2, either $\sigma(r_1) = \sigma(r_2)$ (*weak orthogonality*) or $\sigma(c_1 \wedge c_2)$ is unsatisfiable (*incompatibility of guards*; see [26] for a decidable approximation of this condition).

Following the approach taken in Babel [26], we can easily extend our operational semantics to conditional rules by considering a conditional rule $l \to r \Leftarrow c$ as syntactic sugar for the rule $l \to (c \Rightarrow r)$, where the right-hand side is a *guarded expression*. The operational meaning of a guarded expression "$c \Rightarrow r$" is defined by the predefined rule

$$(true \Rightarrow X) \;\to\; X \;.$$

Thus, a guarded expression is evaluated by an attempt to reduce the condition to *true*. If the condition c is evaluable to *true*, the guarded expression $c \Rightarrow r$ is replaced by r, i.e., by the right-hand side of the conditional rule. If the condition is not evaluable to *true* and does not suspend, the disjunct containing this guarded expression will be deleted by definition of $\xrightarrow{RN}$, i.e., the application of the conditional rule fails and does not contribute to the final result.

The use of extra variables in conditions requires the search facilities of logic programming to compute appropriate values for the extra variables in order to solve the condition. For instance, the predicate $member(E, L)$, which is satisfied if E occurs in the list L, can be defined by the single conditional rule

$$member(E, L) \to true \;\Leftarrow\; append(L1, [E|L2]) \doteq L$$

if the list concatenation function *append* is defined as in Section 6. If the definitional tree for *append* has a flexible branch node, a goal like $member(X, [a, b, c])$ is evaluated by $\xrightarrow{RN}$ to the final disjunctive expression

$$(X = a \,[]\, true) \vee (X = b \,[]\, true) \vee (X = c \,[]\, true) \;.$$

Therefore, logic programming can be simply embedded into our framework. Note, however, that the use of functions instead of predicates may lead to more efficient evaluation strategies than SLD-resolution (see [14, 18] and Sections 5.3 and 6 for discussions about these advantages of functional logic languages).

7.3 Optimizing Disjunctive Computations

Most functions occurring in application programs are inductively sequential, i.e., they are defined by definitional trees containing only *branch* and *rule* nodes. Such functions have the nice property that ground function calls are evaluated by $\xrightarrow{RN}$ in a fully deterministic way, i.e., no disjunction occurs during the computation. The situation is quite different if functions are defined by rules with overlapping left-hand sides so that the corresponding definitional trees contain *or* nodes (e.g., as in Example 3.3). If such a function is evaluated, $\xrightarrow{RN}$ computes a disjunction whenever the computation does not suspend. Thus, the computation is non-deterministic even for ground terms. Although this behavior is conform with logic programming, from a functional programmer's point of view non-deterministic computation steps should not occur during ground term evaluation. However, current functional languages avoid this non-determinism with the risk of possible non-terminating computations. For instance, consider the rules for $*$ in Example 3.3. Functional languages with left-to-right pattern matching [30] compute the result 0 for the term $0 * \bot$ (where $\bot$ is a non-terminating function) but do not compute a result for the symmetric expression $\bot * 0$.

It is possible to improve our evaluation strategy without loosing completeness even in the presence of *or* nodes. For this purpose it is necessary to keep track of the positions where reduction steps are applied in the different alternatives of *or* nodes. Consider a node $or(\mathcal{T}_1, \mathcal{T}_2)$ where σ_i is the computed substitution and p_i is the position in the term where the rule is applied w.r.t. $\mathcal{T}_i$ $(i = 1, 2)$.[8]

1. If $\sigma_1 = id$ and p_1 is a prefix of p_2, i.e., p_2 is identical to or below p_1, then we define

   $$cst(t, or(\mathcal{T}_1, \mathcal{T}_2)) = cst(t, \mathcal{T}_1)$$

   (i.e., we ignore the second alternative, which is called *dynamic cut* in [24] if p_1 is the root position). This optimization is justified by the weak orthogonality property of the rewrite rules and can be efficiently implemented by checking the bindings

[8] For the sake of simplicity, we consider only the case that exactly one disjunct is computed in each alternative.

Strategy	Restrictions on definitional trees
Needed narrowing [4]	only *rule* and flexible *branch* nodes; optimal strategy w.r.t. length of derivations and number of computed solutions
Weakly needed narrowing [5, 23]	only *rule*, flexible *branch*, and *or* nodes
Simple lazy narrowing [26, 28] and resolution	particular definitional trees with flexible *branch* nodes (one *branch*/*rule* tree for each left-hand side, all rules connected by *or* nodes)
Lazy functional languages [30]	definitional trees with left-to-right pattern matching (generated by algorithm *gt* of Section 6); initial expression has no free variable
Residuation [1, 2, 21, 29]	rigid branches for non-Boolean functions; flexible branches for predicates (cf. Section 6)

Figure 2: Specification of different operational models by definitional trees

of the variables in the term after an application of a rewrite rule (see [24] for more details).

2. If $\sigma_1 = \sigma_2$ and the positions p_1 and p_2 are independent, we can perform both reduction steps in parallel in the independent subterms of t (instead of generating a disjunction for both alternatives). This optimization is justified by the completeness results for *parallel narrowing* [5] where it is shown that identical substitutions computed by different alternatives of *or* nodes can be merged.

Both optimizations together leads to an evaluation strategy with a purely deterministic behavior on ground terms (see [5] for more details).

8 Conclusion

We have presented a new computation model for functional logic programs which combines the most important execution principles of functional, logic, and functional logic languages. It is based on lazy reduction of expressions combined with the necessary instantiation of logical variables. In contrast to previous approaches for integrating functional and logic programming, our model is more flexible by combining the ideas of residuation and narrowing into a single coherent framework. Thus, it is the first approach to integrate a possibly non-deterministic evaluation of functions (narrowing) with a concurrent-oriented evaluation style (residuation). Other approaches to combine concurrent computation models and functional programming features, like Oz [29], often translate functions into predicates and, thus, loose functional dependencies and nested term structures which are often useful to provide efficient (optimal) evaluation strategies. Moreover, it is the first framework in this direction which provides for clear soundness and completeness results.

We have shown that the combination of narrowing- and residuation-based computation models is possible in a coherent way so that the individual principles can be obtained by simple restrictions of our semantics. To specify our computation model, we have used definitional trees, a hierarchical data structure containing the rules of a defined function. A definitional tree specifies the evaluation behavior for a single function. Although there are similarities to pattern matching in modern functional languages, definitional trees allow the specification of more powerful evaluation strategies. Moreover, we have extended them to describe the possibly concurrent computation of different arguments of a function. In a concrete implementation, it is not necessary to explicitly specify the definitional tree for each function. Since definitional trees without *and* nodes can be automatically generated using some default strategy (e.g., as described in Section 6), it is only necessary to explicitly specify the trees if one wants to change the default strategy or to include annotations for concurrent computations.

In order to emphasize the flexibility of our computational model, we show in Figure 2 the implementation of different evaluation strategies by putting particular restrictions on definitional trees.

Due to the flexibility of our computation model together with its simplicity (note that Figure 1 contains the entire specification of the strategy), this model is a suitable basis for languages which combine features from functional, logic, and concurrent[9] programming. Thus, this computation model is the basis of Curry [19], a new declarative language intended to combine recent developments in the area of functional logic programming.

The implementation of our computation model is

[9] Note that true concurrency requires an additional choice construct which we have not included since it destroys the completeness property of our model.

outside the scope of this paper. Note, however, that evaluation strategies based on definitional trees can be easily implemented in Prolog [18]. Disjunctions can be treated explicitly as in Escher [21] or Oz [29], or implicitly by backtracking as in Prolog. A first prototype implementation has been developed in Prolog.

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, 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.

[3] S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.

[4] 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.

[5] S. Antoy, R. Echahed, and M. Hanus. A Parallel Narrowing Strategy. Technical Report TR 96-1, Portland State University, 1996.

[6] J.A. Bergstra and J.W. Klop. Conditional Rewrite Rules: Confluence and Termination. *Journal of Computer and System Sciences*, Vol. 32, No. 3, pp. 323–362, 1986.

[7] 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.

[8] P.G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-Resolution. *Theoretical Computer Science 59*, pp. 3–23, 1988.

[9] 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.

[10] 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.

[11] 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.

[12] J.C. Gonzáles-Moreno, M.T. Hortalá-Gonzáles, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A Rewriting Logic for Declarative Programming. In *Proc. ESOP'96*, pp. 156–172. Springer LNCS 1058, 1996.

[13] 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.

[14] 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.

[15] M. Hanus. Combining Lazy Narrowing and Simplification. In *Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pp. 370–384. Springer LNCS 844, 1994.

[16] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.

[17] M. Hanus. Analysis of Residuating Logic Programs. *Journal of Logic Programming*, Vol. 24, No. 3, pp. 161–199, 1995.

[18] M. Hanus. Efficient Translation of Lazy Functional Logic Programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pp. 252–266. Springer LNCS 1048, 1995.

[19] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, 1995.

[20] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. In *Proc. Seventh International Conference on Rewriting Techniques and Applications (RTA'96)*, pp. 138–152. Springer LNCS 1103, 1996.

[21] J.W. Lloyd. Combining Functional and Logic Programming Languages. In *Proc. of the International Logic Programming Symposium*, pp. 43–57, 1994.

[22] J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, University of Bristol, 1995.

[23] R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 184–200. Springer LNCS 714, 1993.

[24] R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. *Theoretical Computer Science 142*, pp. 59–87, 1995.

[25] 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.

[26] 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.

[27] 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.

[28] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.

[29] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pp. 324–343. Springer LNCS 1000, 1995.

[30] P. Wadler. Efficient Compilation of Pattern-Matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pp. 78–103. Prentice Hall, 1987.

[31] D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.