# Functional Logic Programming: From Theory to Curry*

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

**Abstract.** Functional logic programming languages combine the most important declarative programming paradigms, and attempts to combine these paradigms have a long history. The declarative multi-paradigm language Curry is influenced by recent advances in the foundations and implementation of functional logic languages. The development of Curry is an international initiative intended to provide a common platform for the research, teaching, and application of integrated functional logic languages. This paper surveys the foundations of functional logic programming that are relevant for Curry, the main features of Curry, and extensions and applications of Curry and functional logic programming.

## 1 Introduction

Compared to traditional imperative languages, functional as well as logic languages provide a higher and more abstract level of programming that leads to reliable and maintainable programs. Although the motivations are similar in both paradigms, the concrete languages differ due to their different foundations, namely the lambda calculus and first-order predicate logic. Thus, it is a natural idea to combine these worlds of programming into a single paradigm, and attempts for doing so have a long history. However, the interactions between functional and logic programming features are complex in detail so that the concrete design of an integrated functional logic language is a non-trivial task. This is demonstrated by a lot of research work on the semantics, operational principles, and implementation of functional logic languages since more than two decades. Fortunately, recent advances in the foundation and implementation of functional logic languages have shown reasonable principles that lead to the design of practically applicable programming languages. The declarative multi-paradigm language Curry[1] [69,92] is based on these principles. It is developed by an international initiative of researchers in this area and intended to provide a common platform for the research, teaching, and application of integrated functional logic languages. This paper surveys the foundations of functional logic programming that are relevant for Curry, design decisions and main features of

* This work was partially supported by the German Research Council (DFG) under grants Ha 2457/5-1 and Ha 2457/5-2 and the NSF under grant CCR-0218224.

[1] http://www.curry-language.org

Curry, implementation techniques, and extensions and applications of functional logic programming.

Since this paper is intended to be a compact survey, not all of the numerous papers in this area can be mentioned and the relevant topics are only sketched. Interested readers might look into the cited references for more details. In particular, there exist other surveys on particular topics related to this paper. [66] is a survey on the development and the implementation of various evaluation strategies for functional logic languages that have been explored until more than a decade ago. [15] contains a good survey on more recent evaluation strategies and classes of functional logic programs. The survey [119] is more specialized but reviews the efforts to integrate constraints into functional logic languages.

The rest of this paper is structured as follows. The next main section introduces and reviews the foundations of functional logic programming that are used in current functional logic languages. Section 3 discusses important aspects of the language Curry. Section 4 surveys the efforts to implement Curry and related functional logic languages. Sections 5 and 6 contain references to further extensions and applications of functional logic programming, respectively. Finally, Section 7 contains our conclusions with notes about related languages.

## 2 Foundations of Functional Logic Programming

### 2.1 Basic Concepts

Functional logic languages are intended to combine the most important features of functional languages (algebraic data types, polymorphic typing, demand-driven evaluation, higher-order functions) and logic languages (computing with partial information, constraint solving, nondeterministic search for solutions). A *functional program* is a set of *functions* or *operations* defined by *equations* or *rules*. A *functional computation* consists of replacing subexpressions by equal (w.r.t. the defining equations) subexpressions until no more replacements (or *reductions*) are possible and a value or normal form is obtained. For instance, consider the operation `double` defined by[2]

```
double x = x+x
```

The expression "`double 1`" is replaced by `1+1`. The latter can be replaced by `2` if we interpret the operator "`+`" to be defined by an infinite set of equations, e.g., `1+1=2`, `1+2=3`, etc (we will discuss the handling of such operations later). In a similar way, one can evaluate nested expressions (where the replaced subexpression is underlined):

`double (1+2)` → `(1+2)+(1+2)` → `3+(1+2)` → `3+3` → `6`

[2] For concrete examples in this paper, we use the Curry syntax which is very similar to the syntax of Haskell [117], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of an operation $f$ to an expression $e$ is denoted by juxtaposition ("$f\ e$"). Moreover, binary operators like "`+`" are written infix.

There is also another order of evaluation if we replace the arguments of operators from right-to-left:

```
double (1+2)  →  (1+2)+(1+2)  →  (1+2)+3  →  3+3  →  6
```

In this case, both derivations lead to the same result. This indicates a fundamental property of declarative languages: the value of a computed result does not depend on the order or time of evaluation due to the absence of side effects. This simplifies the reasoning about and maintenance of declarative programs.

Obviously, these are not all possible evaluation orders. Another one is obtained by evaluating the argument of `double` before applying its defining equation:

```
double (1+2)  →  double 3  →  3+3  →  6
```

In this case, we obtain the same result with less evaluation steps. This leads to questions about appropriate *evaluation strategies*, where a strategy can be considered as a function that determines for an expression the next subexpression to be replaced: Which strategies are able to compute values for which classes of programs? As we will see, there are important differences in case of recursive programs. If there are several strategies, which strategies are better w.r.t. the number of evaluation steps, implementation effort, etc? Many works in the area of functional logic programming have been devoted to finding appropriate evaluation strategies. A detailed account of the development of such strategies can be found in [66]. In the following, we will only survey the strategies that are relevant for current functional logic languages.

Although functional languages are based on the lambda calculus that is purely based on function definitions and applications, modern functional languages offer more features for convenient programming. In particular, they support the definition of algebraic data types by enumerating their *constructors*. For instance, the type of Boolean values consists of the constructors `True` and `False` that are declared as follows:

```
data Bool = True | False
```

Operations on Booleans can be defined by pattern matching, i.e., by providing several equations for different argument values:

```
not True  = False
not False = True
```

The principle of replacing equals by equals is still valid provided that the actual arguments have the required form, e.g.:

```
not (not False)  →  not True  →  False
```

More complex data structures can be obtained by recursive data types. For instance, a list of elements, where the type of elements is arbitrary (denoted by the type variable `a`), is either the empty list "`[]`" or the non-empty list "`x:xs`" consisting of a first element `x` and a list `xs`. Hence, lists can be defined by

```
data List a = [] | a : List a
```

For conformity with Haskell, the type "`List a`" is usually written as `[a]` and finite lists $e_1$`:`$e_2$`:...:`$e_n$`:[]` are written as `[`$e_1$`,`$e_2$`,...,`$e_n$`]`. We can define operations on recursive types by inductive definitions where pattern matching supports the convenient separation of the different cases. For instance, the concatenation operation "`++`" on polymorphic lists can be defined as follows (the optional type declaration in the first line specifies that "`++`" takes two lists as input and produces an output list, where all list elements are of the same unspecified type):

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : xs++ys
```

Beyond its application for various programming tasks, the operation "`++`" is also useful to specify the behavior of other operations on lists. For instance, the behavior of an operation `last` that yields the last element of a list can be specified as follows: for all lists $l$ and elements $e$, `last` $l = e$ iff $\exists xs : xs$ `++[`$e$`]` $= l$.[3] Based on this specification, one can define an operation and verify that this definition satisfies the given specification (e.g., by inductive proofs as shown in [34]). This is one of the situations where functional logic languages become handy. Similarly to logic languages, functional logic languages provide search for solutions for existentially quantified variables. In contrast to pure logic languages, they support equation solving over nested functional expressions so that an equation like $xs$`++[`$e$`]` $=$ `[1,2,3]` is solved by instantiating $xs$ to the list `[1,2]` and $e$ to the value `3`. For instance, in Curry one can define the operation `last` as follows:

```
last l | xs++[e] =:= l = e   where xs,e free
```

Here, the symbol "`=:=`" is used for *equational constraints* in order to provide a syntactic distinction from defining equations. Similarly, *extra variables* (i.e., variables not occurring in the left-hand side of the defining equation) are explicitly declared by "`where...free`" in order to provide some opportunities to detect bugs caused by typos. A *conditional equation* of the form $l$ `|` $c$ `=` $r$ is applicable for reduction if its condition $c$ has been solved. In contrast to purely functional languages where conditions are only evaluated to a Boolean value, functional logic languages support the *solving* of conditions by guessing values for the unknowns in the condition. As we have seen in the previous example, this reduces the programming effort by reusing existing operations and allows the direct translation of specifications into executable program code. The important question to be answered when designing a functional logic language is: How are conditions solved and are there constructive methods to avoid a blind guessing of values for unknowns? This is the purpose of narrowing strategies that are discussed next.

[3] The exact meaning of the equality symbol is omitted here since it will be discussed later.

### 2.2 Narrowing

Techniques for goal solving are well developed in the area of logic programming. Since functional languages advocate the equational definition of operations, it is a natural idea to integrate both paradigms by adding an equality predicate to logic programs, leading to *equational logic programming* [93,115,116]. On the operational side, the resolution principle of logic programming must be extended to deal with replacements of subterms. *Narrowing*, originally introduced in automated theorem proving [125], is a constructive method to deal with such replacements. For this purpose, defining equations are interpreted as rewrite rules that are only applied from left to right (as in functional programming). In contrast to functional programming, the left-hand side of a defining equation is *unified* with the subterm under evaluation. In order to provide more detailed definitions, some basic notions of term rewriting [31,48] are briefly recalled. Although the theoretical part uses notations from term rewriting, its mapping into the concrete programming language syntax should be obvious.

Since we ignore polymorphic types in the theoretical part of this paper, we consider a many-sorted *signature* $\Sigma$ 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. Given a set of variables $\mathcal{X}$, the set of *terms* and *constructor terms* are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. 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 term is *operation-rooted* (*constructor-rooted*) if its root symbol is an operation (constructor). A *head normal form* is a term that is not operation-rooted, i.e., a variable or a constructor-rooted term.

A *pattern* is a term of the form $f(d_1, \ldots, d_n)$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A *term rewriting system* (TRS) is set of rewrite rules, where an (unconditional) *rewrite rule* is a pair $l \to r$ with a linear pattern $l$ as the *left-hand side* (*lhs*) and a term $r$ as the *right-hand side* (*rhs*). Note that this definition reflects the specific properties of functional logic programs. Traditional term rewriting systems [48] differ from this definition in the following points:

1. We have required that the left-hand sides must be linear patterns. Such rewrite systems are also called *constructor-based* and exclude rules like

   ```
   (xs ++ ys) ++ zs  =  xs ++ (ys ++zs)          (assoc)
   last (xs ++ [e])  =  e                        (last)
   ```

   Although this seems to be a restriction when one is interested in writing equational specifications, it is not a restriction from a programming language point of view, since functional as well as logic programming languages enforces the same requirement (although logic languages do not require linearity of patterns, this can be easily obtained by introducing new variables and adding equations for them in the condition; conditional rules are discussed below). Often, non-constructor-based rules specify properties of operations rather than providing a constructive definition (compare rule *assoc* above that specifies the associativity of "`++`"), or they can be transformed

into constructor-based rules by moving non-constructor terms in left-hand side arguments into the condition (e.g., rule *last*). Although there exist narrowing strategies for non-constructor-based rewrite rules (see [66,116,125] for more details), they often put requirements on the rewrite system that are too strong or difficult to check in universal programming languages, like termination or confluence. An important insight from recent works on functional logic programming is that the restriction to constructor-based programs is quite reasonable since this supports the development of efficient and practically useful evaluation strategies (see below). Although narrowing has been studied for more general classes of term rewriting systems, those extensions are often applied to areas like theorem proving rather than programming (e.g., [52]).

2. Traditional rewrite rules $l \to r$ require that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. A TRS where all rules satisfy this restriction is also called a *TRS without extra variables.*[4] Although this makes sense for rewrite-based languages, it limits the expressive power of functional logic languages (see the definition of `last` in Section 2.1). Therefore, functional logic languages usually do not have this variable requirement, although some theoretical results have only been proved under this requirement.

In order to formally define computations w.r.t. a TRS, we need a few further notions. A *position* $p$ in a term $t$ is represented by a sequence of natural numbers. Positions are used to identify particular subterms. Thus, $t|_p$ 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 [48] for details). A *substitution* is an idempotent mapping $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ where the *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. Substitutions are obviously extended to morphisms on terms. We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* $\sigma$ with $\sigma(x_i) = t_i$ $(i = 1, \ldots, n)$ and $\sigma(x) = x$ for all other variables $x$. A substitution $\sigma$ is *constructor* (*ground constructor*), if $\sigma(x)$ is a constructor (ground constructor) term for all $x \in \mathcal{D}om(\sigma)$.

A *rewrite step* $t \to_{p,R} t'$ (in the following, $p$ and $R$ will often be omitted in the notation of rewrite and narrowing steps) is defined if $p$ is a position in $t$, $R = l \to r$ is a rewrite rule with fresh variables,[5] and $\sigma$ is a substitution with $t|_p = \sigma(l)$ and $t' = t[\sigma(r)]_p$. The instantiated lhs $\sigma(l)$ is also called a *redex* (*red*ucible *ex*pression). A term $t$ is called *irreducible* or in *normal form* if there is no term $s$ with $t \to s$. $\overset{*}{\to}$ denotes the reflexive and transitive closure of a relation $\to$.

Rewrite steps formalize functional computation steps with pattern matching as introduced in Section 2.1. The goal of a sequence of rewrite steps is to compute

[4] In case of conditional rules, which are discussed later, the condition is considered as belonging to the right-hand side so that variables occurring in the condition but not in the left-hand side are also extra variables.

[5] In classical traditional term rewriting, fresh variables are not used when a rule is applied. Since we consider also rules containing extra variables in right-hand sides, it is important to replace them by fresh variables when the rule is applied.

a normal form. A *rewrite strategy* determines for each rewrite step a rule and a position for applying the next step. A *normalizing strategy* is one that terminates a rewrite sequence in a normal form, if it exists. Note, however, that normal forms are not necessarily the interesting results of functional computations, as the following example shows.

*Example 1.* Consider the operation

```
idNil [] = []
```

that is the identity on the empty list but undefined for non-empty lists. Then, a normal form like "`idNil [1]`" is usually considered as an error rather than a result. Actually, Haskell reports an error for evaluating the term "`idNil [1+2]`" rather than delivering the normal form "`idNil [3]`". □

Therefore, the interesting results of functional computations are *constructor terms* that will be also called *values*. Evaluation strategies used in functional programming, such as lazy evaluation, are not normalizing, as the previous example shows.

Functional logic languages are able to do more than pure rewriting since they instantiate variables in a term (also called *free* or *logic variables*) so that a rewrite step can be applied. The combination of variable instantiation and rewriting is called *narrowing*. Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if $p$ is a non-variable position in $t$ (i.e., $t|_p$ is not a variable) and $\sigma(t) \to_{p,R} t'$. Since the substitution $\sigma$ is intended to instantiate the variables in the term under evaluation, one often restricts $\mathcal{D}om(\sigma) \subseteq \mathcal{V}ar(t)$. 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 $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (where $\sigma = \{\}$ in the case of $n = 0$). Since in functional logic languages we are interested in computing *values* (constructor terms) as well as *answers* (substitutions), we say that the narrowing derivation $t \leadsto_\sigma^* c$ *computes the value* $c$ *with answer* $\sigma$ if $c$ is a constructor term.

The above definition of narrowing is too general for a realistic implementation since it allows arbitrary instantiations of variables in the term under evaluation. Thus, all possible instantiations must be tried in order to compute all possible values and answers. Obviously, this does not lead to a practical implementation. Therefore, older narrowing strategies (see [66] for a detailed account) were influenced by the resolution principle and required that the substitution used in a narrowing step must be a most general unifier of $t|_p$ and the left-hand side of the applied rule. As shown in [19], this condition prevents the development of optimal evaluation strategies. Therefore, most recent narrowing strategies relax this traditional requirement but provide another constructive method to compute a small set of unifiers in narrowing steps, as we will see below. The next example shows the non-optimality of narrowing with most general unifiers.

*Example 2.* Consider the following program containing a declaration of natural numbers in Peano's notation and two operations for addition and a "less than or equal" test (the pattern "`_`" denotes an unnamed *anonymous variable*):

```
data Nat = O | S Nat
```

```
add 0     y = y
add (S x) y = S (add x y)

leq 0     _     = True                                    (leq1)
leq (S _) 0     = False                                   (leq2)
leq (S x) (S y) = leq x y                                 (leq3)
```

Consider the initial term "`leq v (add w 0)`" where `v` and `w` are free variables. By applying rule $leq_1$, `v` is instantiated to `0` and the result `True` is computed:

$$\texttt{leq v (add w 0)} \leadsto_{\{\texttt{v}\mapsto\texttt{0}\}} \texttt{True}$$

Further answers can be obtained by instantiating `v` to `(S...)`. This requires the evaluation of the subterm `(add w 0)` in order to allow the application of rule $leq_2$ or $leq_3$. For instance, the following narrowing derivation computes the value `False` with answer $\{\texttt{v}\mapsto\texttt{S z}, \texttt{w}\mapsto\texttt{0}\}$:

$$\texttt{leq v (add w 0)} \leadsto_{\{\texttt{w}\mapsto\texttt{0}\}} \texttt{leq v 0} \leadsto_{\{\texttt{v}\mapsto\texttt{S z}\}} \texttt{False}$$

However, we can also apply rule $leq_1$ in the second step of the previous narrowing derivation and obtain the following derivation:

$$\texttt{leq v (add w 0)} \leadsto_{\{\texttt{w}\mapsto\texttt{0}\}} \texttt{leq v 0} \leadsto_{\{\texttt{v}\mapsto\texttt{0}\}} \texttt{True}$$

Obviously, the last derivation is not optimal since it computes the same value as the first derivation with a less general answer and needs one more step. This derivation can be avoided by instantiating variable `v` to `S z` in the first narrowing step:

$$\texttt{leq v (add w 0)} \leadsto_{\{\texttt{v}\mapsto\texttt{S z},\ \texttt{w}\mapsto\texttt{0}\}} \texttt{leq (S z) 0}$$

Now, rule $leq_1$ is no longer applicable, as intended. However, this first narrowing step contains a substitution that is not a most general unifier between the evaluated subterm `(add w 0)` and the left-hand side of some rule for `add`. □

**Needed Narrowing.** The first narrowing strategy that advocated the use of non-most general unifiers and for which optimality results have been shown is needed narrowing [19]. Furthermore, needed narrowing steps can be efficiently computed. Therefore, it has become the basis of modern functional logic languages.[6]

Needed narrowing is based on the idea to perform only narrowing steps that are in some sense necessary to compute a result (such strategies are also called *lazy* or *demand-driven*). For doing so, it analyzes the left-hand sides of the rewrite rules of an operation under evaluation (starting from an outermost operation). If there is an argument position where all left-hand sides are constructor-rooted, the corresponding actual argument must be also rooted by one of the constructors in order to apply a rewrite step. Thus, the actual argument is evaluated to head

[6] Concrete languages and implementations add various extensions in order to deal with larger classes of programs that will be discussed later.

normal form if it is operation-rooted and, if it is a variable, nondeterministically instantiated with some constructor.

*Example 3.* Consider again the program of Example 2. Since the left-hand sides of all rules for `leq` have a constructor-rooted first argument, needed narrowing instantiates the variable `v` in "`leq v (add w 0)`" to either `0` or `S z` (where `z` is a fresh variable). In the first case, only rule $leq_1$ becomes applicable. In the second case, only rules $leq_2$ or $leq_3$ become applicable. Since the latter rules have both a constructor-rooted term as the second argument, the corresponding subterm `(add w 0)` is recursively evaluated to a constructor-rooted term before applying one of these rules. □

Since there are TRSs with rules that do not allow such a reasoning, needed narrowing is defined on the subclass of *inductively sequential* TRSs. This class can be characterized by definitional trees [12] that are also useful to formalize and implement various narrowing strategies. Since only the left-hand sides of rules are important for the applicability of needed narrowing, the following characterization of definitional trees [13] considers patterns partially ordered by subsumption (the *subsumption ordering* on terms is defined by $t \leq \sigma(t)$ for a term $t$ and substitution $\sigma$).

A *definitional tree* of an operation $f$ is a non-empty set $T$ of linear patterns partially ordered by subsumption having the following properties:

*Leaves property:* The maximal elements of $T$, called the *leaves*, are exactly the (variants of) the left-hand sides of the rules defining $f$. Non-maximal elements are also called *branches*.

*Root property:* $T$ has a minimum element, called the *root*, of the form $f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are pairwise distinct variables.

*Parent property:* If $\pi \in T$ is a pattern different from the root, there exists a unique $\pi' \in T$, called the *parent* of $\pi$ (and $\pi$ is called a *child* of $\pi'$), such that $\pi' < \pi$ and there is no other pattern $\pi'' \in \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ with $\pi' < \pi'' < \pi$.

*Induction property:* All the children of a pattern $\pi$ differ from each other only at a common position, called the *inductive position*, which is the position of a variable in $\pi$.[7]

An operation is called inductively sequential if it has a definitional tree and its rules do not contain extra variables. A TRS is inductively sequential if all its defined operations are inductively sequential. Intuitively, inductively sequential functions are defined by structural induction on the argument types. Purely functional programs and the vast majority of operations in functional logic programs are inductively sequential. Thus, needed narrowing is applicable to most operations, although extensions are useful for particular operations (see below).

It is often convenient and simplifies the understanding to provide a graphic representation of definitional trees, where each inner node is marked with a pattern, the inductive position in branches is surrounded by a box, and the leaves

[7] There might be more than one potential inductive position when constructing a definitional tree. In this case one can select any of them since the results about needed narrowing do not depend on the selected definitional tree.

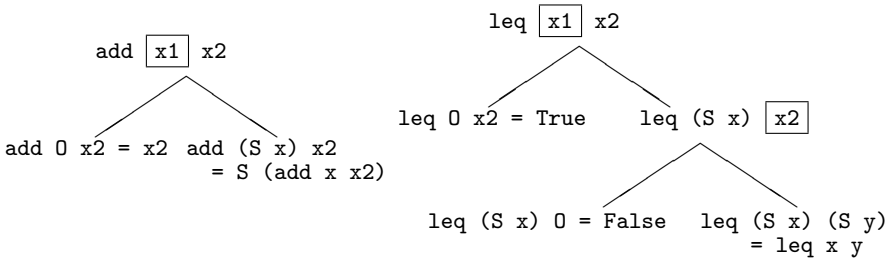


**Fig. 1.** Definitional trees of the operations `add` and `leq`

contain the corresponding rules. For instance, the definitional trees of the operations `add` and `leq`, defined in Example 2, are illustrated in Figure 1. Definitional trees have also a strong correspondence to traditional pattern matching by case expressions in functional languages, as we will see later.

The formal definition of needed narrowing is based on definitional trees and can be found in [19]. A definitional tree can be computed at compile time (see [15,69] for algorithms to construct definitional trees) and contains all information for the efficient implementation of the decisions to be made at run time (compare Example 3). Intuitively, a needed narrowing step is applied to an operation-rooted term $t$ by considering a definitional tree (with fresh variables) for the operation at the root. The tree is recursively processed from the root until one finds a maximal pattern that unifies with $t$. Thus, to compute a needed narrowing step, one starts with the root pattern of the definitional tree and performs at each level with pattern $\pi$ the following case distinction:

- If $\pi$ is a leaf, we apply the corresponding rule.
- If $\pi$ is a branch and $p$ its inductive position, we consider the corresponding subterm $t|_p$:
  1. If $t|_p$ is rooted by a constructor $c$ and there is a child $\pi'$ of $\pi$ having $c$ at the inductive position, we proceed by examining $\pi'$. If there is no such child, we fail, i.e., no needed narrowing step is applicable.
  2. If $t|_p$ is a variable, we nondeterministically instantiate this variable by the constructor term at the inductive position of a child $\pi'$ of $\pi$ and proceed with $\pi'$.
  3. If $t|_p$ is operation-rooted, we recursively apply the computation of a needed narrowing step to $\sigma(t|_p)$, where $\sigma$ is the instantiation of the variables of $t$ performed in the previous case distinctions.

As discussed above, the failure to compute a narrowing step in case (1) is not a weakness but advantageous when we want to compute values. For instance, consider the term $t =$ `idNil [1+2]` where the operation `idNil` is as defined in Example 1. A normalizing strategy performs a step to compute the normal form `idNil [3]` whereas needed narrowing immediately fails since there exists no value as a result. Thus, the early failure of needed narrowing avoids wasting resources.

As a consequence of the previous behavior, the properties of needed narrowing are stated w.r.t. constructor terms as results. In particular, the equality symbol "`=:=`" in goals is interpreted as the *strict equality* on terms, i.e., the equation $t_1$ `=:=` $t_2$ is satisfied iff $t_1$ and $t_2$ are reducible to the same ground constructor term. In contrast to the mathematical notion of equality as a congruence relation, strict equality is not reflexive. Similarly to the notion of result values, this is intended in programming languages where an equation between functional expressions that do not have a value, like "`idNil [1] =:= idNil [1]`", is usually not considered as true. Furthermore, normal forms or values might not exist (note that we do not require terminating rewrite systems) so that reflexivity is not a feasible property of equational constraints (see [60] for a more detailed discussion on this topic).

Strict equality can be defined as a binary operation by the following set of (inductively sequential) rewrite rules. The constant `Success` denotes a solved (equational) constraint and is used to represent the result of successful evaluations.[8]

$$\begin{array}{rcll} c \texttt{ =:= } c & = & \texttt{Success} & \forall c/0 \in \mathcal{C} \\ c\ x_1 \ldots x_n \texttt{ =:= } c\ y_1 \ldots y_n & = & x_1\texttt{=:=}y_1\ \texttt{\&...\&}\ x_n\texttt{=:=}y_n & \forall c/n \in \mathcal{C}, n > 0 \\ \texttt{Success \& Success} & = & \texttt{Success} & \end{array}$$

Thus, it is sufficient to consider strict equality as any other operation. Concrete functional logic languages provide more efficient implementations of strict equality where variables can be bound to other variables instead of instantiating them to ground terms (see also Section 3.2).

Now we can state the main properties of needed narrowing. A (correct) *solution* for an equation $t_1$ `=:=` $t_2$ is a constructor substitution $\sigma$ (note that constructor substitutions are desired in practice since a broader class of solutions would contain unevaluated or undefined expressions) if $\sigma(t_1)$ `=:=` $\sigma(t_2) \stackrel{*}{\rightarrow}$ `Success`. Needed narrowing is sound and complete, i.e., all computed solutions are correct and for each correct solution a possibly more general one is computed, and it does not compute redundant solutions in different derivations:

**Theorem 1 ([19]).** *Let $\mathcal{R}$ be an inductively sequential TRS and $e$ an equation.*

1. *(Soundness) If $e \leadsto^*_\sigma$ `Success` is a needed narrowing derivation, then $\sigma$ is a solution for $e$.*
2. *(Completeness) For each solution $\sigma$ of $e$, there exists a needed narrowing derivation $e \leadsto^*_{\sigma'}$ `Success` with $\sigma'(x) \leq \sigma(x)$ for all $x \in \mathcal{V}ar(e)$.*
3. *(Minimality) If $e \leadsto^*_\sigma$ `Success` and $e \leadsto^*_{\sigma'}$ `Success` are two distinct needed narrowing derivations, then $\sigma$ and $\sigma'$ are independent on $\mathcal{V}ar(e)$, i.e., there is some $x \in \mathcal{V}ar(e)$ such that $\sigma(x)$ and $\sigma'(x)$ are not unifiable.*

Furthermore, in successful derivations, needed narrowing computes only steps that are necessary to obtain the result and, consequently, it computes the *shortest*

[8] Since narrowing is used to solve equations, it does not compute solutions such that an equation is *not* satisfied. This is the motivation to use the specific constant `Success` rather than the Boolean values `True` and `False` as the outcome of equation solving.

*of all possible narrowing derivations* if derivations on common subterms are shared (a standard implementation technique in non-strict functional languages) [19, Corollary 1]. Needed narrowing is currently the only narrowing strategy with such strong results. Therefore, it is an adequate basis for modern functional logic languages, although concrete implementations support extensions that are discussed next.

**Weakly Needed Narrowing.** Inductively sequential TRS are a proper subclass of (constructor-based) TRSs. Although the majority of function definitions is inductively sequential, there are also operations where it is more convenient to relax this requirement. The next interesting superclass are *weakly orthogonal TRSs*. These are rewrite systems where left-hand sides can overlap in a semantically trivial way. Formally, a TRS without extra variables (recall that we consider only left-linear constructor-based rules) is *weakly orthogonal* if $\sigma(r_1) = \sigma(r_2)$ for all (variants of) rules $l_1 \to r_1$ and $l_2 \to r_2$ and substitutions $\sigma$ with $\sigma(l_1) = \sigma(l_2)$.

*Example 4.* A typical example of a weakly orthogonal TRS is the *parallel-or*, defined by the rules:

```
or True  _     = True          (or1)
or _     True  = True          (or2)
or False False = False         (or3)
```

A term like "`or` $s$ $t$" could be reduced to `True` whenever one of the arguments $s$ or $t$ evaluates to `True`. However, it is not clear which of the arguments should be evaluated first, since any of them could result in a nonterminating rewriting or narrowing derivation. `or` has no definitional tree and, thus, needed narrowing cannot be applied. □

In rewriting, several normalizing strategies for weakly orthogonal TRSs have been proposed, such as parallel outermost [114] or weakly needed [122] rewriting that are based on the idea to replace several redexes in parallel in one step. Since strategies for functional logic languages already support nondeterministic evaluations, one can exploit this feature to extend needed narrowing to a *weakly needed narrowing* strategy. The basic idea is to generalize the notion of definitional trees to include *or-branches* which conceptually represent a union of definitional trees [12,18,100]. If such an or-branch is encountered during the evaluation of a narrowing step, weakly needed narrowing performs a nondeterministic guess and proceeds with the subtrees below the or-branches.

*Example 5.* Consider again the rules for the operation `or` shown in Example 4 and the operation `f` defined by

```
f 0 = True
```

One can construct separate definitional trees for the rule sets $\{or_1, or_3\}$ and $\{or_2\}$ and join them by an or-branch. Then there are the following different weakly needed narrowing derivations based on this *generalized definitional tree* for the term "`or (f x) (f x)`":

```
or (f x) (f x) ⇝{x↦0} or True (f 0) ⇝{} True
or (f x) (f x) ⇝{x↦0} or True (f 0) ⇝{} or True True ⇝{} True
or (f x) (f x) ⇝{x↦0} or (f 0) True ⇝{} or True True ⇝{} True
or (f x) (f x) ⇝{x↦0} or (f 0) True ⇝{} True
```

□

Obviously, weakly needed narrowing is no longer optimal in the sense of needed narrowing. However, it is sound and complete for weakly orthogonal TRS in the sense of Theorem 1 [18].

Weakly needed narrowing can be improved by computing weakly needed narrowing steps in parallel, discarding steps with non-minimal substitutions and replacing several outermost redexes in parallel. The resulting strategy, called *parallel narrowing* [18], computes only one derivation for Example 5 and has the general property (beyond soundness and completeness) that it behaves deterministically (i.e., without choices) on ground terms. However, the computation of parallel narrowing steps is quite complex (see [18] for details) so that it has not been integrated in existing functional logic languages, in contrast to weakly needed narrowing that is implemented in languages such as Curry [69,92] or TOY [101].

**Overlapping Inductively Sequential Systems.** Inductively sequential and weakly orthogonal TRSs are *confluent*, i.e., each term has at most one normal form. This property is sensible for functional languages since it ensures that operations are well defined (partial) functions in the mathematical sense. Since the operational mechanism of functional logic languages is more powerful due to its built-in search mechanism, in this context it makes sense to consider also operations defined by non-confluent TRSs. Such operations are also called *nondeterministic*. The prototype of such a nondeterministic operation is a binary operation "`?`" that returns one of its arguments:

```
x ? y = x
x ? y = y
```

Thus, the expression "`0 ? 1`" has two possible results, namely `0` or `1`.

Since functional logic languages already handle nondeterministic computations, they are also capable of dealing with such nondeterministic operations. To provide a reasonable semantics for functional logic programs, constructor-based rules are sufficient but confluence is not required [62]. If operations are interpreted as mappings from values into sets of values (actually, due to the presence of recursive non-strict operations, algebraic structures with cones of partially ordered sets are used instead of sets, see [62] for details), one can provide model-theoretic and proof-theoretic semantics with the usual properties (minimal term models, equivalence of model-theoretic and proof-theoretic solutions, etc). Thus, functional logic programs with nondeterministic operations are still in the design space of declarative languages. Moreover, nondeterministic operations have advantages w.r.t. demand-driven evaluation strategies so that they became a

standard feature of current functional logic languages, whereas older languages, like ALF [65], Babel [109], K-Leaf [60], or SLOG [58], put confluence requirements on their programs. The following example discusses this in more detail.

*Example 6.* Based on the binary operation "`?`" introduced above, one can define an operation `insert` that nondeterministically inserts an element at an arbitrary position in a list:

```
insert e []     = [e]
insert e (x:xs) = (e : x : xs) ? (x : insert e xs)
```

Exploiting this operation, one can define an operation `perm` that returns an arbitrary permutation of a list:

```
perm []     = []
perm (x:xs) = insert x (perm xs)
```

One can already see an important property when reasoning about nondeterministic operations: the computation of results is arbitrary, i.e., one result is as good as any other. For instance, if one evaluates `perm [1,2,3]`, any permutation (e.g., `[3,2,1]` as well as `[1,3,2]`) is an acceptable result. If one puts specific conditions on the results, the completeness of the underlying computational model (e.g., INS, see below) ensures that the appropriate results meeting these conditions are selected.

For instance, one can use `perm` to define an operation `psort` to sort a list based on a "partial identity" function `sorted` that returns its input list if it is sorted:

```
sorted []                                = []
sorted [x]                               = [x]
sorted (x1:x2:xs) | leq x1 x2 =:= True = x1 : sorted (x2:xs)

psort xs = sorted (perm xs)
```

Thus, `psort xs` returns only those permutations of `xs` that are sorted. The advantage of this definition of `psort` in comparison to traditional "generate-and-test" solutions becomes apparent when one considers the demand-driven evaluation strategy (note that one can apply the weakly needed narrowing strategy to such kinds of programs since this strategy is only based on the left-hand sides of the rules but does not exploit confluence). Since in an expression like `sorted (perm xs)` the argument of `sorted` is only evaluated as demanded by `sorted`, the permutations are not fully computed at once. If a permutation starts with a non-ordered prefix, like `S 0 : 0 : perm xs`, the application of the third rule of `sorted` fails and, thus, the computation of the remaining part of the permutation (which can result in $n!$ different permutations if $n$ is the length of the list `xs`) is discarded. The overall effect is a reduction in complexity in comparison to the traditional generate-and-test solution. □

This example shows that nondeterministic operations allow the transformation of "generate-and-test" solutions into "test-of-generate" solutions with a lower complexity since the demand-driven narrowing strategy results in a demand-driven

construction of the search space (see [13,62] for further examples). Antoy [13] shows that desirable properties of needed narrowing can be transferred to programs with nondeterministic operations if one considers *overlapping inductively sequential systems*. These are TRSs with inductively sequential rules where each rule can have multiple right-hand sides (basically, inductively sequential TRSs with occurrences of "`?`" in the top-level of right-hand sides), possibly containing extra variables. For instance, the rules defining `insert` form an overlapping inductively sequential TRS if the second rule is interpreted as a single rule with two right-hand sides ("`e:x:xs`" and "`x : insert e xs`"). The corresponding strategy, called *INS* (*inductively sequential narrowing strategy*), is defined similarly to needed narrowing but computes for each narrowing step a set of replacements. INS is a conservative extension of needed narrowing and optimal modulo nondeterministic choices of multiple right-hand sides, i.e., if there are no multiple right-hand sides or there is an oracle for choosing the appropriate element from multiple right-hand sides, INS has the same optimality properties as needed narrowing (see [13] for more details).

A subtle aspect of nondeterministic operations is their treatment if they are passed as arguments. For instance, consider the nondeterministic operation `coin` defined by

```
coin = 0 ? 1
```

and the expression "`double coin`" (where `double` is defined as in Section 2.1). If the argument `coin` is evaluated (to `0` or `1`) before it is passed to `double`, we obtain the possible results `0` and `2`. However, if the argument `coin` is passed unevaluated to `double`, we obtain after one rewrite step the expression `coin+coin` which has four possible further rewrite derivations resulting in the values `0`, `1`, `1`, and `2`. The former behavior is referred to as *call-time choice* semantics [94] since the choice for the desired value of a nondeterministic operation is made at call time, whereas the latter is referred to as *need-time choice* semantics. There are arguments for either of these semantics depending on the programmer's intention (see [15] for more examples).

Although call-time choice suggests an eager or call-by-value strategy, it fits well into the framework of demand-driven evaluation where arguments are shared to avoid multiple evaluations of the same subterm. For instance, the actual subterm (e.g., `coin`) associated to argument `x` in the rule "`double x = x+x`" is not duplicated in the right-hand side but a reference to it is passed so that, if it is evaluated by one subcomputation, the same result will be taken in the other subcomputation. This technique, called *sharing*, is essential to obtain efficient (and optimal) evaluation strategies. If sharing is used, the call-time choice semantics can be implemented without any further machinery. Furthermore, in many situations call-time choice is the semantics with the "least astonishment". For instance, consider the reformulation of the operation `psort` in Example 6 to

```
psort xs = idOnSorted (perm xs)

idOnSorted xs | sorted xs =:= xs = xs
```

Then, for the call `psort xs`, the call-time choice semantics delivers only sorted permutations of `xs`, as expected, whereas the need-time choice semantics delivers all permutations of `xs` since the different occurrences of `xs` in the rule of `idOnSorted` are not shared. For instance, to evaluate the call `psort [3,2,1]`, one has to verify that the condition

```
sorted (perm [3,2,1]) =:= perm [3,2,1]
```

of `idOnSorted` is satisfied (see below for more details about conditional rules). This can be shown by reducing both occurrences of "`perm [3,2,1]`" to the list `[1,2,3]`. Since the condition is satisfied, the call `idOnSorted (perm [3,2,1])` will be reduced to `perm [3,2,1]` w.r.t. the need-time choice semantics. Thus, one finally obtains all permutations of the input list.

Due to these reasons, current functional logic languages usually adopt the call-time choice semantics.

**Conditional Rules.** The narrowing strategies presented so far are only defined for rewrite rules without conditions, although some of the concrete program examples indicate that conditional rules are convenient in practice. Formally, a *conditional rewrite rule* has the form $l \to r \Leftarrow C$ where $l$ and $r$ are as in the unconditional case and the condition $C$ consists of finitely many equational constraints of the form $s \texttt{=:=} t$. Due to the interpretation of equational constraints as strict equalities, one can define a rewrite step with a conditional rule similar to the unconditional case with the additional requirement that each equational constraint in the condition of an applicable rule must be *joinable*, i.e., both sides of the equation must be reducible to the same ground constructor term.[9] A more precise definition will be provided in Section 2.3.

To extend narrowing to conditional rules, one can define narrowing steps on *equational goals*, i.e., (multi)sets of equations, where an application of a conditional rule adds new conditions to the equational goal. However, to obtain an efficient implementation, functional logic languages often use another technique. As discussed before, efficient narrowing strategies exploit the structure of the left-hand sides of rewrite rules to decide its applicability. In order to do the same for conditional rules, one can consider conditions as part of the right-hand side. This can be achieved by transforming a conditional rule of the form

$$l \;\to\; r \;\Leftarrow\; s_1 \texttt{=:=} t_1 \;\texttt{\&}\ldots\texttt{\&}\; s_n \texttt{=:=} t_n$$

into an unconditional rule

$$l \;\to\; \mathit{cond}(s_1 \texttt{=:=} t_1 \;\texttt{\&}\ldots\texttt{\&}\; s_n \texttt{=:=} t_n,\;\; r)$$

where the "*cond*itional" is defined by $\mathit{cond}(\texttt{Success}, x) \to x$. Since overlapping inductively sequential TRSs allow rules with multiple right-hand sides, one can transform also sets of conditional rules with identical left-hand sides, in contrast to pure term rewriting with confluence requirements where only restricted

[9] The recursion in this intuitive definition of conditional rewriting can be avoided by an iterative definition using levels for rewriting conditions, see [33].

subsets of conditional rules can be transformed into unconditional ones (e.g., [33]). Actually, Antoy [14] has shown a systematic method to translate any conditional constructor-based TRS into an overlapping inductively sequential TRS performing equivalent computations.

For restricted subsets of conditional rules, other transformations that allow the application of more sophisticated narrowing strategies are possible. For instance, in [17] it is shown how to transform any weakly orthogonal conditional TRS into an unconditional TRS so that the weakly needed and parallel narrowing strategies are sound and complete on the transformed programs. The application of parallel narrowing to the transformed programs has the effect that conditions are evaluated in parallel so that nondeterministic evaluation steps are completely avoided on ground terms.

**Further Works.** Although weakly needed narrowing or INS are reasonable narrowing strategies for rather general classes of functional logic programs, further works investigated improvements for specific classes of TRSs. For instance, [97] proposes a refinement of definitional trees if there is more than one inductive position (e.g., in operations like "`=:=`" and "`&`" defined above). This is exploited to implement needed narrowing in a way that reduces the number of nondeterministic choices. [50,51] proposes natural narrowing as a refinement of weakly needed narrowing by incorporating a better treatment of demandedness properties.

Since the formal reasoning about sophisticated narrowing strategies could be fairly complex, *narrowing calculi* have been studied. Usually, such calculi are defined by a set of inference rules on equational goals so that properties like soundness and completeness can be shown by proving invariants w.r.t. the application of inference rules. This simplifies the proof of properties of narrowing techniques but has the disadvantage that a connection to efficient implementations required for real languages is more difficult to establish. Examples for such narrowing calculi are LNC [107] for confluent TRSs, OINC [95] for orthogonal TRSs and goals with ground normal forms as right-hand sides, or CLNC [62] as the narrowing equivalent to CRWL (see below). LNC and OINC do not require constructor-based TRSs. This has useful applications for applicative TRSs [111] in order to study narrowing calculi for programs with higher-order operations.

### 2.3 Constructor-based Rewriting Logic

As discussed in the previous section on overlapping inductively sequential TRS, sharing becomes important for the semantics of nondeterministic operations. This has the immediate consequence that traditional equational reasoning is no longer applicable. For instance, the expressions `double coin` and `coin+coin` are not equal since the latter can reduce to `1` while this is impossible for the former w.r.t. a call-time choice semantics. In order to provide a semantical basis for such general functional logic programs, González-Moreno et al. [62] have proposed the rewriting logic *CRWL* (Constructor-based conditional ReWriting Logic) as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and nondeterministic operations and call-time

choice semantics. This logic has been also used to link a natural model theory as an extension of the traditional theory of logic programming and to establish soundness and completeness of narrowing strategies for rather general classes of TRSs [47].

To deal with non-strict operations, CRWL considers signatures $\Sigma_\bot$ that are extended by a special symbol $\bot$ to represent *undefined values*. For instance, $\mathcal{T}(\mathcal{C} \cup \{\bot\}, \mathcal{X})$ denotes the set of partial constructor terms, e.g., `1:2:`$\bot$ denotes a list starting with elements `1` and `2` and an undefined rest. Such *partial terms* are considered as finite approximations of possibly infinite values. CRWL defines the deduction of two kinds of basic statements: *approximation statements* $e \to t$ with the intended meaning "the partial constructor term $t$ approximates the value of $e$", and *joinability statements* $e_1$ `=:=` $e_2$ with the intended meaning that $e_1$ and $e_2$ have a common *total* approximation $t \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ with $e_1 \to t$ and $e_2 \to t$, thus modeling strict equality with terms containing variables. To model call-time choice semantics, rewrite rules are only applied to partial *values*. Hence, the following notation for *partial constructor instances* of a set of (conditional) rules $\mathcal{R}$ is useful:

$$[\mathcal{R}]_\bot = \{\sigma(l \to r \Leftarrow C) \mid l \to r \Leftarrow C \in \mathcal{R}, \sigma : \mathcal{X} \to \mathcal{T}(\mathcal{C} \cup \{\bot\}, \mathcal{X})\}$$

Then CRWL is defined by the following set of inference rules (where the program is represented by a TRS $\mathcal{R}$):

| | | |
|---|---|---|
| Bottom: | $e \to \bot$ | $e \in \mathcal{T}(\mathcal{C} \cup \mathcal{F} \cup \{\bot\}, \mathcal{X})$ |
| Restricted reflexivity: | $x \to x$ | $x \in \mathcal{X}$ |
| Decomposition: | $\dfrac{e_1 \to t_1 \ \cdots \ e_n \to t_n}{c(e_1, \ldots, e_n) \to c(t_1, \ldots, t_n)}$ | $c/n \in \mathcal{C}$, $t_i \in \mathcal{T}(\mathcal{C} \cup \{\bot\}, \mathcal{X})$ |
| Function reduction: | $\dfrac{e_1 \to t_1 \ \cdots \ e_n \to t_n \quad C \quad r \to t}{f(e_1, \ldots, e_n) \to t}$ | $f(t_1, \ldots, t_n) \to r \Leftarrow C \in [\mathcal{R}]_\bot$ and $t \neq \bot$ |
| Joinability: | $\dfrac{e_1 \to t \qquad e_2 \to t}{e_1$ `=:=` $e_2}$ | $t \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ |

Rule (Bottom) specifies that $\bot$ approximates any expression. The condition $t \neq \bot$ in rule (Function reduction) avoids unnecessary applications of this rule since this case is already covered by the first rule. The restriction to partial constructor instances in this rule formalizes non-strict operations with a call-time choice semantics. Operations might have non-strict arguments that are not evaluated since the corresponding actual arguments can be derived to $\bot$ by rule (Bottom). If the value of an argument is required to evaluate the right-hand side of a rule, it must be evaluated to a partial constructor term before it is passed to the right-hand side (since $[\mathcal{R}]_\bot$ contains only partial constructor instances), which corresponds to a call-time choice semantics. Note that this does not prohibit the use of lazy implementations since this semantical behavior can be enforced by sharing unevaluated expressions. Actually, [62] defines a lazy narrowing calculus that reflects this behavior.

| | | |
|---|---|---|
| Fapp | $f(\sigma(t_1),\ldots,\sigma(t_n)) \ \to_l \ \sigma(r)$ | $f(t_1,\ldots,t_n) \to r \in \mathcal{R}$, $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{C},\mathcal{X})$ |
| LetIn | $g(\ldots,e,\ldots) \ \to_l$ $\texttt{let}\ x = e\ \texttt{in}\ g(\ldots,x,\ldots)$ | $e = f(\ldots)$ ($f \in \mathcal{F}$) or $e = \texttt{let}\ldots$ $g \in \mathcal{C} \cup \mathcal{F}$, $x \in \mathcal{X}$ fresh variable |
| Flat | $\texttt{let}\ x = (\texttt{let}\ y = e_1\ \texttt{in}\ e_2)\ \texttt{in}\ e_3 \ \to_l$ $\texttt{let}\ y = e_1\ \texttt{in}\ (\texttt{let}\ x = e_2\ \texttt{in}\ e_3)$ | $y$ does not appear free in $e_3$ |
| Bind | $\texttt{let}\ x = t\ \texttt{in}\ e \ \to_l \ \sigma(e)$ | $t \in \mathcal{T}(\mathcal{C},\mathcal{X})$, $\sigma = \{x \mapsto t\}$ |
| Elim | $\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \ \to_l \ e_2$ | $x$ does not appear free in $e_2$ |
| Contx | $e[e_1]_p \ \to_l \ e[e_2]_p$ | $e_1 \to_l e_2$ and $p$ position in $e$ |

**Fig. 2.** Rules for let-rewriting [103]

CRWL can be used as the logical foundation of functional logic languages with non-strict nondeterministic operations. It is a basis for the verification of functional logic programs [45] and has been extended in various directions, e.g., higher-order operations [63], algebraic types [30], polymorphic types [61], failure [104], constraints [102] etc. An account on CRWL and its applications can be found in [119].

As discussed in [103], a disadvantage of CRWL is its high level of abstraction: CRWL relates expressions to computed (partial) results but misses a one-step evaluation mechanism similarly to rewriting for functional programs or narrowing for functional logic programs. Thus, it is sometimes difficult to use CRWL to reason about computations in functional logic languages. To overcome this drawback, López-Fraguas et al. [103] proposed specific reduction relations conform with CRWL (for simplicity, we consider here only rules without conditions). The following reduction relation $\rightarrowtail$ is similarly to standard rewriting but restricts the reduction of operations to situations where the arguments are partial terms. Furthermore, any expression can be approximated by $\bot$.

$$\begin{aligned} e[f(t_1,\ldots,t_n)]_p &\rightarrowtail e[r]_p \quad \text{if } f(t_1,\ldots,t_n) \to r \in [\mathcal{R}]_\bot \text{ and } p \text{ position in } e \\ e &\rightarrowtail e[\bot]_p \quad \text{if } p \text{ is a position in } e \end{aligned}$$

CRWL and the relation $\rightarrowtail$ are equivalent in the sense that CRWL and $\rightarrowtail$ relates the same partial terms to each expression [103].

This reduction relation is more appropriate to reason about computations. For instance, it has been applied in [79] to approximate call patterns in functional logic computations. On the negative side, this reduction relation allows a nondeterministic choice between reducing or approximating a call to some operation which leads to a large computation space. Furthermore, the order of reduction steps does not reflect the typical demand-driven order of evaluation steps. Therefore, López-Fraguas et al. [103] proposed *let-rewriting*, i.e., rewriting on expressions containing let-bindings which denote arguments that need to be evaluated in order to reduce some operation. For this purpose, *let-expressions* are expressions where the extended form "$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$" is also permitted ($x$ is visible in $e_2$ but not in $e_1$, i.e., lets are not recursive). The let-rewriting

relation $\rightarrow_l$ is defined by the rules in Figure 2 (we omit the precise definition of free variable occurrences and substitutions on let-expressions since they are standard). In contrast to CRWL, let-rewriting does not use $\bot$ to approximate expressions that are not demanded. Instead, such expressions are moved from an argument position to a let-binding (LetIn) which can be eliminated (Elim) if they are not demanded. Thus, a function call is reduced if the arguments do not contain any operation (Fapp) which reflects the call-time choice semantics. The equivalence of CRWL and let-rewriting is shown in [103]. There it is also shown that let-rewriting is equivalent to standard rewriting for deterministic programs. Let-rewriting does not enforce any reduction strategy. This will be considered in Section 2.5 where a more strategy-oriented semantics will be discussed.

### 2.4 Residuation

Although narrowing extends soundness and completeness results of logic programming to the general framework of functional logic programming, it is not the only method that has been proposed to integrate functions into logic programs. An alternative technique, called *residuation*, is based on the idea to delay or suspend function calls until they are ready for deterministic evaluation. The residuation principle is used, for instance, in the languages Escher [99], Le Fun [2], Life [1], NUE-Prolog [110], and Oz [126]. Since the residuation principle evaluates function calls by deterministic reduction steps, nondeterministic search must be encoded by predicates [1,2,110] or disjunctions [99,126]. Moreover, if some part of a computation might suspend, one needs a primitive to execute computations concurrently. For instance, the conjunction of constraints "`&`" needs to evaluate both arguments to `Success` so that it is reasonable to do it concurrently, i.e., if the evaluation of one argument suspends, the other one is evaluated.

*Example 7.* Consider Example 2 together with the operation

```
nat 0     = Success
nat (S x) = nat x
```

If the operation `add` is evaluated by residuation, i.e., suspends if the first argument is a variable, the expression "`add y 0 =:= S 0 & nat y`" is evaluated as follows:

$$
\begin{array}{lll}
\texttt{add y 0 =:= S 0 \& }\underline{\texttt{nat y}} & \rightarrow_{\{\texttt{y}\mapsto\texttt{S x}\}} & \underline{\texttt{add (S x) 0}}\texttt{ =:= S 0 \& nat x} \\
 & \rightarrow_{\{\}} & \underline{\texttt{S (add x 0) =:= S 0}}\texttt{ \& nat x} \\
 & \rightarrow_{\{\}} & \texttt{add x 0 =:= 0 \& }\underline{\texttt{nat x}} \\
 & \rightarrow_{\{\texttt{x}\mapsto\texttt{0}\}} & \underline{\texttt{add 0 0}}\texttt{ =:= 0 \& Success} \\
 & \rightarrow_{\{\}} & \underline{\texttt{0 =:= 0}}\texttt{ \& Success} \\
 & \rightarrow_{\{\}} & \underline{\texttt{Success \& Success}} \\
 & \rightarrow_{\{\}} & \texttt{Success}
\end{array}
$$

Thus, the solution $\{\texttt{y} \mapsto \texttt{S 0}\}$ is computed by switching between the residuating operation `add` and the constraint `nat` that instantiates its argument to natural numbers. □

Narrowing and residuation are quite different approaches to integrate functional and logic programming. Narrowing is sound and complete but requires the nondeterministic evaluation of function calls if some arguments are unknown. Residuation might not compute some result due to the potential suspension of evaluation but avoids guessing on operations. From an operational point of view, there is no clear advantage of one of the strategies. One might have the impression that the deterministic evaluation of operations in the case of residuation is more efficient, but there are examples where residuation has an infinite computation space whereas narrowing has a finite one (see [67] for more details). On the other hand, residuation offers a concurrent evaluation principle with synchronization on logic variables (sometimes also called *declarative concurrency* [128]) and a conceptually clean method to connect *external operations* to declarative programs [35] (note that narrowing requires operations to be explicitly defined by rewrite rules). Therefore, it is desirable to integrate both principles in a single framework. This has been proposed in [69] where residuation is combined with weakly needed narrowing by extending definitional trees with branches decorated with a *flexible*/*rigid* tag. Operations with flexible tags are evaluated as with narrowing whereas operations with rigid tags suspend if the arguments are not sufficiently instantiated. The overall strategy is similar to weakly needed narrowing with the exception that a rigid branch with a free variable in the corresponding inductive position results in the suspension of the operation under evaluation. For instance, if the branch of `add` in Figure 1 has a rigid tag, then `add` is evaluated as shown in Example 7.

### 2.5 Flat Programs

The constructor-based rewriting logic defines the meaning of functional logic programs without referring to a concrete evaluation strategy. However, reasoning about the behavior of programs (e.g., program analysis), optimizing programs (e.g., partial evaluation), or building language specific tools (e.g., debuggers, profilers) demands for a more detailed description of the operational semantics of programs. On the one hand, such a description should reflect all details of the program execution, like pattern matching, sharing, binding logic variables, etc. On the other hand, it should be high level so that properties of programs can be formally derived.

It has been shown that such a description can be better based on an intermediate *flat representation of programs* rather than on the source-level functional logic programs. Figure 3 shows the syntax of such a flat language which has been successfully applied for this purpose. Flat programs contain an explicit representation of pattern matching (*case*/*fcase* corresponds to branches in definitional trees, *or* represents a choice between definitional trees in the case of rules with overlapping left-hand sides). The difference between *case* and *fcase* corresponds to residuation and narrowing: when the argument $e$ evaluates to a free variable, *case* suspends whereas *fcase* nondeterministically binds this variable to a pattern in a branch of the case expression.

| | | |
|---|---|---|
| $P$ | $::= D_1 \ldots D_m$ | (program) |
| $D$ | $::= f(x_1, \ldots, x_n) = e$ | (function definition) |
| $e$ | $::= x$ | (variable) |
| | $\mid c(e_1, \ldots, e_n)$ | (constructor call) |
| | $\mid f(e_1, \ldots, e_n)$ | (function call) |
| | $\mid$ *case* $e$ *of* $\{p_1 \to e_1; \ldots; p_n \to e_n\}$ | (rigid case) |
| | $\mid$ *fcase* $e$ *of* $\{p_1 \to e_1; \ldots; p_n \to e_n\}$ | (flexible case) |
| | $\mid e_1$ *or* $e_2$ | (disjunction) |
| | $\mid$ *let* $\{x_1 = e_1, \ldots, x_n = e_n\}$ *in* $e$ | (let binding) |
| $p$ | $::= c(x_1, \ldots, x_n)$ | (pattern) |

**Fig. 3.** Syntax for flat programs

Let bindings as shown in Figure 3 are in principle not required for translating functional logic programs into flat programs. However, they can be used to translate extended classes of programs containing circular data structures and are convenient to express sharing without the use of complex graph structures [49,64]. Operationally, let bindings introduce new structures in memory that are updated after evaluation, which is essential for lazy computations [98]. Furthermore, let bindings are also useful to represent free variables in expressions by a direct circular binding of the form "*let* $\{x = x\}$ *in* $e$".

For instance, the operations `add` and "`?`" defined in Section 2.2 have the following flat representations:

```
add x y  =  fcase x of { 0 → y;  S z → S (add z y) }
x ? y    =  x or y
```

[87] defines a mapping between definitional trees and flat programs and shows the equivalence of needed narrowing and outermost narrowing on flat programs. A precise description of (weakly needed) narrowing and residuation with sharing is given in [3] as an extension of Launchbury's natural semantics for lazy evaluation [98]. For this purpose, one considers only *normalized* flat programs, i.e., programs where the arguments of constructor and function calls are always variables. Any flat program can be normalized by introducing new variables by let expressions [3]. For instance, the expression "`double coin`" is normalized into "*let* {`x` = `coin`} *in* `double x`". In order to model sharing, the variables are interpreted as references into a heap where new let bindings are stored and function calls are updated with their evaluated results.

To be more precise, a *heap*, denoted by $\Gamma, \Delta$, or $\Theta$, is a partial mapping from variables to expressions (the *empty heap* is denoted by `[]`). The value associated to variable $x$ in heap $\Gamma$ is denoted by $\Gamma[x]$. $\Gamma[x \mapsto e]$ denotes a heap $\Gamma'$ with $\Gamma'[x] = e$ and $\Gamma'[y] = \Gamma[y]$ for all $x \neq y$. We use this notation either as a condition or as an update of a heap. A logic variable $x$ that is unbound in $\Gamma$ is represented by a circular binding of the form $\Gamma[x] = x$.

Using heap structures, one can provide a high-level description of the operational behavior of residuation and demand-driven narrowing with call-time

VarCons $\quad \Gamma[x \mapsto t] : x \;\Downarrow\; \Gamma[x \mapsto t] : t$ where $t$ is constructor-rooted

VarExp $\quad \dfrac{\Gamma[x \mapsto e] : e \;\Downarrow\; \Delta : v}{\Gamma[x \mapsto e] : x \;\Downarrow\; \Delta[x \mapsto v] : v}$ where $e$ is not constructor-rooted and $e \neq x$

Val $\quad \Gamma : v \;\Downarrow\; \Gamma : v$ where $v$ is constructor-rooted or a variable with $\Gamma[v] = v$

Fun $\quad \dfrac{\Gamma : \rho(e) \;\Downarrow\; \Delta : v}{\Gamma : f(\overline{x_n}) \;\Downarrow\; \Delta : v}$ where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$

Let $\quad \dfrac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \;\Downarrow\; \Delta : v}{\Gamma : \mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e \;\Downarrow\; \Delta : v}$ where $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ are fresh variables

Or $\quad \dfrac{\Gamma : e_i \;\Downarrow\; \Delta : v}{\Gamma : e_1\ \mathit{or}\ e_2 \;\Downarrow\; \Delta : v}$ where $i \in \{1, 2\}$

Select $\quad \dfrac{\Gamma : e \;\Downarrow\; \Delta : c(\overline{y_n}) \qquad \Delta : \rho(e_i) \;\Downarrow\; \Theta : v}{\Gamma : (f)\mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v}$ where $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$

Guess $\quad \dfrac{\Gamma : e \;\Downarrow\; \Delta : x \qquad \Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i) \;\Downarrow\; \Theta : v}{\Gamma : \mathit{fcase}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v}$

where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are fresh variables

**Fig. 4.** Natural semantics of normalized flat programs [3]

choice in form of a natural semantics (also called big-step semantics). The natural semantics uses judgements of the form "$\Gamma : e \;\Downarrow\; \Delta : v$" with the meaning that in the context of heap $\Gamma$ the expression $e$ evaluates to value (head normal form) $v$ and produces a modified heap $\Delta$. Figure 4 shows the rules defining this semantics w.r.t. a given normalized flat program $P$ ($\overline{o_k}$ denotes a sequence of objects $o_1, \ldots, o_k$). The rules VarCons and VarExp retrieve expressions from the heap: VarCons retrieves values whereas the expressions retrieved by VarExp are further evaluated. In order to avoid the reevaluation of the same expression, VarExp updates the heap with the computed value, which models sharing. Values (i.e., head normal forms) are just returned by rule Val. Fun unfolds function calls by evaluating the right-hand side after binding the formal parameters to the actual ones. Let introduces new bindings in the heap and renames the variables in the expressions with the fresh names introduced in the heap. Or nondeterministically evaluates one of its arguments. Finally, Select and Guess deal with *case* expressions. If the first argument of *case* evaluates to a constructor-rooted

term, Select evaluates the corresponding branch of the *case* expression, otherwise (if the argument evaluates to an unbound variable), Guess nondeterministically binds the argument to one of the patterns of the *case* expression and continues with the corresponding branch.

By introducing a stack to model the context of a computation, one can also define an equivalent small-step semantics which can be enriched with more details of realistic implementations, such as search strategies, concurrency, external operations etc (see [3] for details).

The flat representation of programs and its operational semantics has been used for various language-oriented tools (e.g., compilers [20,27], partial evaluators [4,5], trace-oriented debuggers [40], profilers [39]) and extended in various ways (e.g., higher-order functions [87], memoization [53], encapsulated search [38], computation costs [39]).

Flat programs can be considered as a kernel language for functional logic programming since programs written in concrete functional logic languages like Curry with all its syntactic sugar can be automatically translated into flat programs. It is interesting to note that the language of flat programs is *not* minimal since it contains two concepts that can be simulated by each other: logic variables and overlapping rules (i.e., disjunctions expressed by *or*). For instance, a rule like

```
x ? y = x or y
```

can be expressed without *or* by introducing a logic variable `z` that ranges over two data constructors `I0` and `I1`:

```
x ? y = let {z = z} in fcase z of { I0 → x; I1 → y }
```

On the other hand, logic variables can be eliminated by defining *nondeterministic generator operations* for each type. For instance, a generator for type `Nat` defined in Example 2 is the operation `genNat` defined by

```
genNat = O ? S genNat
```

so that `genNat` evaluates to all possible values of type `Nat`. Now each occurrence of a logic variable can be replaced by a corresponding generator, e.g., the expression

```
let {x = x} in leq x (S O)
```

can be transformed into

```
let {x = genNat} in leq x (S O)
```

without changing the computed results. These equivalences have been used in implementations of functional logic languages [41,42]. Further details can be found in [24].

## 3 Language Concepts: Curry

After the review of recent results and techniques for functional logic programming, this section shows how they influenced the design of a concrete programming language. For this purpose, we consider Curry [69,92] (the relation to other languages will be briefly discussed in Section 7), a functional logic language based on many of the concepts introduced so far. The development of Curry is the outcome of an international initiative of researchers in the area of functional logic programming with the goal to provide a common standard for the research, teaching, and application of integrated functional logic languages.

The syntax of Curry is very similar to the syntax of Haskell [117] and has been already introduced in an informal manner. Curry is a polymorphically typed language with a Hindley/Milner-like type system supporting type inference [46]. Since the type concept is fairly standard and orthogonal to the other issues of the language, it is not explicitly addressed in the following. Therefore, this section is devoted to discuss concepts and design decisions that are unique to Curry.

### 3.1 Semantics

A Curry program is formally a constructor-based TRS. Thus, its *declarative semantics* is given by the rewriting logic CRWL, i.e., operations and constructors are non-strict with a call-time choice semantics for nondeterministic operations.

The *operational semantics* is based on an extension of needed narrowing on generalized definitional trees with sharing and residuation. The precise description is based on normalized flat programs as already shown in Section 2.5. Thus, for (flexible) inductively sequential operations, which form the vast majority of operations in application programs, the evaluation strategy is optimal w.r.t. the length of derivations and number of computed solutions and always computes a value if it exists (in case of nondeterministic choices only if the underlying implementation is fair w.r.t. such choices, as [26,27,88]). Therefore, the programmer can concentrate on the declarative meaning of programs and needs less attention to the consequences of the particular evaluation strategy (see [73] for a more detailed discussion). The following example shows that Curry is an improvement compared to Haskell which does not have a similar behavior for all inductively sequential operations.

*Example 8.* Consider the inductively sequential operation `f` defined by

```
f 0 []     = 0
f x (y:ys) = y
```

and a nonterminating operation $\bot$. Then the expression "`f` $\bot$ `[1]`" has the value `1`, but Haskell does not terminate on this expression due to the strict left-to-right top-down pattern matching strategy. Furthermore, if the operation `g` is defined by

```
g x = 0
g 1 = 1
```

in Haskell the expression "`g 1`" is evaluated to `0` although the second equation indicates that `1` is also an acceptable result. As a consequence, program rules in Haskell cannot be interpreted as equations but all the rules defining an operation in a Haskell program must be passed through a complex pattern-matching compiler [129] in order to understand their meaning. □

As discussed above, *external operations* not implemented by explicit rules, like basic arithmetic operators or I/O operations, cannot be handled by narrowing. Therefore, Curry exploits *residuation* to connect external operations in a conceptually clean way (see also [35]). Since external operations can not usually deal with unevaluated arguments possibly containing logic variables, the arguments of external operations are reduced to a ground value before the operation is evaluated. If some arguments are not ground but contain logic variables, the function call is suspended until the variables are bound to ground values. The *concurrent conjunction* "`&`" on constraints is the basic concurrency operator that evaluates both arguments in a non-specified order to success.

The discussion of residuation-based languages (see Section 2.4) might give the impression that residuation is useful for user-defined operations. Therefore, previous versions of Curry had also the possibility to define operations as "rigid". However, it turned out that this is unnecessary in practice, since the suspension of operations often caused more complications than their active application through narrowing (exceptions are related to concurrent objects and ports for distributed programming, see below). Moreover, the optimality of needed narrowing ensures that the argument guessing is restricted to a minimal part. Therefore, all user-defined operations are evaluated by narrowing and only external operations and conditionals like "`if-then-else`" or "`case-of`" are evaluated by residuation. The latter is motivated by the fact that conditionals are often used as guards to prevent infinite recursion. A useful primitive to define general "suspension" combinators for concurrent programming is the predefined operation `ensureNotFree` that returns its argument evaluated to head normal form but suspends as long as the result is a logic variable.

### 3.2 Constraints

Functional logic languages are able to solve equational constraints. As shown in Section 2.2, such constraints occur in conditions of conditional rules and are intended to restrict the applicability of the rewrite rule, i.e., a replacement with a conditional rule is only performed if the condition has been shown to be satisfied (e.g., compare the definition of `last` in Section 2.1). Thus, constraints are solved when conditional rules are applied. In terms of concurrent constraint programming languages [121], solving constraints in conditions corresponds to *tell* constraints. The dual operation, *ask*, is used in conditionals like "`if-then-else`". Curry distinguishes these different uses by different types: `Success` and `Bool`.

Equational constraints are expressions of type `Success`. Since constraints are ordinary expressions, they are first-class values that can be passed in arguments or data structures. For instance, the following "constraint combinator" takes a

list of constraints as input and creates a new constraint that is satisfied if all constraints in the input list are satisfied:

```
allValid :: [Success] -> Success
allValid []     = success
allValid (c:cs) = c & allValid cs
```

Here, `success` is not a constructor but denotes the trivial constraint that is always satisfied. Exploiting the higher-order features of Curry (see below), one can define it also by

```
allValid = foldr (&) success
```

Note that the constructor `Success` was introduced in Section 2.2 only to provide a rewrite-based definition of strict equality. It is not available in Curry where a more efficient implementation of strict equality is used. The main difference shows up when an equational constraint "`x =:= y`" between two logic variables `x` and `y` is solved. Solving it with the rewrite rules shown in Section 2.2, `x` and `y` are nondeterministically bound to ground constructor terms which usually results in an infinite search space. This is avoided in Curry by binding one variable to the other, similar to logic programming.

Hence, the type `Success` is a type without constructors but with a few basic constraints like `success` and "`=:=`" and a concurrent conjunction "`&`" to combine constraints into larger units. Actually, one can consider `Success` as equivalent to the functional type "*ConstraintStore* $\rightarrow$ *ConstraintStore*" mapping a constraint store into a new constraint store. Then, the trivial constraint `success` is the identity mapping and a constraint like `x=:=2` maps a constraint store into a new one which is extended by the binding of `x` to `2`. Constraint stores are implicitly chained through a derivation, cloned in nondeterministic steps, and extended when evaluating a condition of a rule. This view has been used in [106] to connect a solver for real arithmetic constraints to a Curry implementation. By adding basic constraints that deal with other constraint domains, like real arithmetic or finite domain constraints, typical applications of constraint logic programming can be covered and combined with features of lazy higher-order programming [20,54,55,102,106,119].

The condition of a rule is any expression of type `Success`, i.e., it is not only a conjunction of equational constraints but can also be constructed by constraint combinators like `allValid`. By contrast, the condition in an "`if-then-else`" must be an expression of type `Bool` since two different values (`True` and `False`) are required to select the `then` or `else` branch according to the result of the Boolean test. For this purpose, Curry also supports a *test equality* predicate "`==`" of type "`a->a->Bool`" to check the equality of two *ground* constructor terms. In contrast to "`=:=`", a call to "`==`" is suspended if an argument contains logic variables so that the equality cannot be decided without instantiating these variables. Hence, one can consider "`==`" as a rigid operation defined by the rules

$$
\begin{array}{rcll}
c \texttt{ == } c & \texttt{=} & \texttt{True} & \forall c/0 \in \mathcal{C} \\
c\ x_1 \ldots x_n \texttt{ == } c\ y_1 \ldots y_n & \texttt{=} & x_1\texttt{==}y_1\ \texttt{\&\&...\&\&}\ x_n\texttt{==}y_n & \forall c/n \in \mathcal{C}, n > 0 \\
c\ x_1 \ldots x_n \texttt{ == } d\ y_1 \ldots y_m & \texttt{=} & \texttt{False} & \forall c/n \neq d/m \in \mathcal{C} \\
\texttt{True \&\& x} & \texttt{=} & \texttt{x} & \\
\texttt{False \&\& x} & \texttt{=} & \texttt{False} &
\end{array}
$$

As an alternative to the distinction between equational constraints in conditions and Boolean tests in conditionals, one might also use equational constraints in conditionals, as, for instance, done in the purely narrowing-based language TOY [101]. This demands for the negation of constraints so that a conditional "`if` $c$ `then` $e_1$ `else` $e_2$" is evaluated by nondeterministically evaluating $c \wedge e_1$ or $\neg c \wedge e_2$. Actually, this is implemented in TOY by the use of *disequality constraints*. However, the complexity of the handling of disequality constraints puts more demands on the implementation side.

### 3.3 Higher-order Operations

The use of higher-order operations, i.e., operations that take other operations as arguments or yield them as results, is an important programming technique in functional languages so that it should be covered also by functional logic languages. Typical examples are the mapping of an operation to all elements of a list (`map`) or a generic accumulator for lists (`foldr`):

```
map :: (a->b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs

foldr :: (a->b->b) -> b -> [a] -> b
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Logic languages often provide higher-order features through a transformation into a first-order program [131] by defining a predicate *apply* that implements the application of an arbitrary operation of the program to an expression. This technique is also known as "defunctionalization" [118] and enough to support the higher-order features of current functional languages (e.g., lambda abstractions can be replaced by new function definitions). Therefore, this solution is also used in Curry.

As an example, consider a program containing the unary operation `not` and the binary operations `add` and `leq`. Then, one can define the meaning of *apply* by the following rules:

```
apply not     x = not x                    (apply1)
apply add     x = add x                    (apply2)
apply (add x) y = add x y                  (apply3)
apply leq     x = leq x                    (apply4)
apply (leq x) y = leq x y                  (apply5)
```

Thus, a *partially applied function call*, i.e., a $n$-ary operation called with less than $n$ arguments, is considered as a constructor-rooted, i.e., not further evaluable, term (one can also make this distinction clear by introducing new constructor symbols for such partial applications). Thus, the first argument in each rule for *apply* is always a constructor-rooted term. If an $n$-ary function call with $n-1$ arguments is applied to its final argument, the operation is evaluated (e.g., as in the rules $apply_1$, $apply_3$, $apply_5$). This explicit definition has been used in Prolog-based implementations of functional logic languages [20].

An important difference to purely functional languages shows up when the operation to be applied (i.e., the first argument of *apply*) is a logic variable. In this case, one can instantiate this variable to all possible operations occurring in the program [63]. Since this might result also in instantiations that are not intended w.r.t. the given types, one can restrict these instantiations to well-typed ones which requires to keep type information at run time [29,61]. Another option is the instantiation of variables denoting functions to (well-typed) lambda terms in order to cover programs that can reason about bindings and block structure [87]. Since all these options might result in huge search spaces due to function instantiation, and the feasibility and usefulness for larger application programs is not clear, Curry chooses a more pragmatic solution: function application *apply* is rigid, i.e., it suspends if the first functional argument is a logic variable. For this behavior, we can avoid the explicit introduction of rules for *apply*: it can be considered as a primitive operation with a meaning that is specified by extending the natural semantics of Figure 4 with the following rule (where partially applied function calls are considered as constructor-rooted terms in the rules VarCons and Val):

$$\textsf{Apply}\quad \frac{\Gamma : x \;\Downarrow\; \Delta : \varphi(\overline{x_k}) \qquad \Delta : \varphi(\overline{x_k}, y) \;\Downarrow\; \Theta : v}{\Gamma : apply\ x\ y \;\Downarrow\; \Theta : v}$$

where $\varphi$ is either a constructor or an $n$-ary operation with $k < n$.

### 3.4 Encapsulated Search

An essential difference between functional and logic computations is their determinism behavior. Functional computations are deterministic. This enables a reasonable treatment of I/O operations by the monadic approach where I/O actions are considered as transformations on the outside world [130]. The monadic I/O approach is also taken in Curry. However, logic computations might cause (don't know) nondeterministic choices, i.e., a computation can be cloned and continued in two different directions. Since one can not clone the entire outside world, nondeterministic choices during monadic I/O computations are not allowed and lead to a run-time error in Curry. Since this might restrict the applicability of logic programming techniques in larger applications, there is a clear need to *encapsulate nondeterministic search* between I/O actions. For this purpose, [89] proposes the addition of a primitive search operator

```
try :: (a->Success) -> [a->Success]
```

that takes a constraint abstraction, e.g., `(\x->x=:=coin)`, as input, evaluates it until the first nondeterministic step occurs, and returns the result: the empty list in case of failure, a list with a single element in case of success, or a list with at least two elements representing a nondeterministic choice. For instance, `try (\x->x=:=coin)` evaluates to `[\x->x=:=0,\x->x=:=1]`. Based on this primitive, one can define various search strategies to explore the search space and return its solutions. [105] shows an implementation of this primitive.

Although typical search operators of Prolog, like `findall`, `once`, or negation-as-failure, can be implemented using the primitive `try`, it became also clear that the combination with demand-driven evaluation and sharing causes further complications. For instance, in an expression like

```
let y=coin in try (\x->x=:=y)
```

it is not obvious whether the evaluation of `coin` (introduced outside but demanded inside the search operator) should be encapsulated or not. Hence, the result of this expression might depend on the evaluation order. For instance, if `coin` is evaluated before the `try` expression, it results in two computations where `y` is bound to `0` in one computation and to `1` in the other computation. Hence, `try` does not encapsulate the nondeterminism of `coin` (this is also the semantics of `try` implemented in [105]). However, if `coin` is evaluated inside the capsule of `try` (because it is not demanded before), then the nondeterminism of `coin` is encapsulated. These and more peculiarities are discussed in [38]. Furthermore, the order of the solutions might depend on the textual order of program rules or the evaluation time (e.g., in parallel implementations). Therefore, [38] contains a proposal for another primitive search operator:

```
getSearchTree :: a -> IO (SearchTree a)
```

Since `getSearchTree` is an I/O action, its result (in particular, the order of solutions) depends on the current environment, e.g., time of evaluation. It takes an expression and delivers a search tree representing the search space when evaluating the input:

```
data SearchTree a = Or [SearchTree a] | Val a | Fail
```

Based on this primitive, one can define various concrete search strategies as tree traversals. To avoid the complications w.r.t. shared variables, `getSearchTree` implements a *strong encapsulation view*, i.e., conceptually, the argument of `getSearchTree` is cloned before the evaluation starts in order to cut any sharing with the environment. Furthermore, the structure of the search tree is computed lazily so that an expression with infinitely many values does not cause the non-termination of the search operator if one is interested in only one solution. More details about search trees and their operational semantics can be found in [38,41].

Although these concepts are sufficient to encapsulate nondeterministic computations to avoid nondeterminism in I/O operations, it is often also desired to collect all the values of an expression in some data structure at arbitrary computation points, e.g., to accumulate all values, to compute a minimal value, or to check whether some constraint has no solution (similarly to "negation as

failure" in logic programming). As mentioned above, the initial concepts for encapsulation in functional logic languages have the drawback that their result might depend on the degree of evaluation of the argument (which is difficult to grasp in non-strict languages). A solution to this problem is presented in [25] by the introduction of *set functions*. For each defined operation $f$, $f_{\mathcal{S}}$ denotes its corresponding set function. In order to be independent of the evaluation order, $f_{\mathcal{S}}$ encapsulates only the nondeterminism caused by evaluating $f$ except for the nondeterminism caused by evaluating the arguments to which $f$ is applied. For instance, consider the operation `decOrInc` defined by

```
decOrInc x = (x-1) ? (x+1)
```

Then "$\texttt{decOrInc}_{\mathcal{S}}$ `3`" evaluates to (an abstract representation of) the set $\{2, 4\}$, i.e., the nondeterminism originating from `decOrInc` is encapsulated into a set. However, "$\texttt{decOrInc}_{\mathcal{S}}$ `(2?5)`" evaluates to two different sets $\{1, 3\}$ and $\{4, 6\}$ due to its nondeterministic argument, i.e., the nondeterminism originating from the argument is not encapsulated but produces different sets. It is shown in [25] that the results of set functions do not depend on the evaluation order so that the disadvantages of the earlier approaches are avoided. [104,120] contain similar proposals but with the restriction to test only the failure of expressions. There, an operation `fails` computes the set of all values of its argument expression and returns true, if this set is empty, or false, otherwise.

## 4 Implementation

The definition of needed narrowing and its extensions shares many similarities with pattern matching in functional or unification in logic languages. Thus, it is reasonable to use similar techniques to implement functional logic languages. Due to the coverage of logic variables and nondeterministic search, one could try to translate functional logic programs into Prolog programs in order to exploit the implementation technology available for Prolog. Actually, there are various approaches to implement functional logic languages with demand-driven evaluation strategies in Prolog (e.g., [11,20,44,68,96,100]). A common idea is the translation of source operations into predicates that compute only the *head normal form* (i.e., a constructor-rooted term or a variable) of a call to this operation. Thus, an $n$-ary operation could be translated into a predicate with $n + 1$ arguments where the last argument contains the head normal form of the evaluated call. For instance, the list concatenation "`++`" defined in Section 2.1 can be translated into the following Prolog predicate `conc`:

```
conc(Xs,Ys,H) :- hnf(Xs,HXs), conc_1(HXs,Ys,H).
conc_1([],Ys,H) :- hnf(Ys,H).
conc_1([X|Xs],Ys,[X|conc(Xs,Ys)]).
```

Since the first argument of "`++`" is an inductive position, its value is needed and, hence, computed by the predicate `hnf` before it is passed to the predicate `conc_1` implementing the pattern matching on the first argument. Since the

right-hand side of the second rule of "++" is already in head normal form, no further evaluation is necessary. In the first rule of `conc_1`, it is unknown at compile time whether the second argument `Ys` is already in head normal form. Therefore, the evaluation to head normal form is enforced by the predicate `hnf`. The goal `hnf(`$t$`,`$h$`)` evaluates any term $t$ to its head normal form $h$. Some of the clauses defining `hnf` are:

```
hnf(V,V) :- var(V), !.
hnf([],[]).
hnf([X|Xs],[X|Xs]).
...
hnf(conc(Xs,Ys),H) :- conc(Xs,Ys,H).
...
```

Using this scheme, there is a straightforward transformation of needed narrowing and its extensions into Prolog. However, this scheme does not implement sharing where it is required that each function call should be evaluated at most once. This can be achieved by representing function calls as "suspensions" that contain two further arguments: one indicates whether the suspension has been already evaluated and the other contains the head normal form. Thus, the second rule of `conc_1` has then the form

```
conc_1([X|Xs],Ys,[X|susp(conc(Xs,Ys),E,H)]).
```

and the definition of `hnf` has the additional clause

```
hnf(susp(Call,E,H),H) :- var(E) -> hnf(Call,H), E=ready ; true.
```

Another implementation of sharing is proposed in [20] where only variables with multiple occurrences in right-hand sides are shared instead of function calls. In order to implement residuation, coroutining features of modern Prolog implementation can be exploited (see [20] for details).

The transformation of functional logic programs into Prolog programs has many advantages. It is fairly simple to implement, one can use constraint solvers available in many Prolog implementations in application programs, and one can exploit the advances made in efficient implementations of Prolog (depending on the Prolog implementation, one can improve the efficiency of the above code by a couple of minor transformations). Thus, one obtains with a limited effort an implementation that can be used for larger applications with a comparable efficiency than other more low-level implementations (e.g., [81,101]).

Despite these advantages, the transformation into Prolog has the drawback that one is fixed to Prolog's backtracking strategy to implement nondeterministic search. This hampers the implementation of encapsulated search or fair search strategies. Therefore, there are also various approaches to use other target languages than Prolog. For instance, [27] presents techniques to compile Curry programs into Java programs that implement a fair search for solutions. A translation of Curry programs into Haskell programs is proposed in [41,42] which offers a primitive operator to encapsulate search, similarly to `getSearchTree` introduced in Section 3.4. **Virtual machines** to compile Curry programs are

proposed in [26,88,105]. In particular, [26,88] implement a fair (global) search for solutions, and [105] covers the implementation of encapsulated search.

Beyond the compilation of programs into particular target languages or virtual machines, the implementation of programming languages has many other facets that have been considered also for functional logic languages. For instance, **partial evaluation** is a powerful compile-time technique to optimize source-level programs. [9] contains a general framework for the partial evaluation of functional logic programs. It has been specialized to the case of needed narrowing in [10] where the superiority of needed narrowing has been shown also for partial evaluation. To provide a practical implementation of a partial evaluator covering all features of Curry, [5] shows that this is possible if the partial evaluator is based on the flat representation introduced in Section 2.5.

To understand the run-time behavior of functional logic programs, specific tools are required since it is well known that even the operational behavior of purely functional programs with lazy evaluation is difficult to understand [112]. This demands for tools specifically designed to show operational aspects of functional logic programs. Thus, a few activities into this direction have started. Since traditional tracing tools, although provided in practical systems, are often not helpful, the objective of **debugging tools** is usually a representation of operational aspects that are more oriented towards the program text rather than execution steps. For instance, COOSy [36] is a tool to observe the evaluation of individual expressions, operations, or logic variables in a program. It records the observable events during run time and presents the corresponding results (computed values, variable bindings etc) with a separate viewer. TeaBag [28] provides another view that connects the activities of virtual machine with the source program under execution. Other debugging tools are more oriented towards the semantics of functional logic programs. For instance, [40] describes a formal semantics for a trace-based debugging tool, [39] proposes a profiling tool based on a cost-augmented semantics of functional logic programs, [113] proposes a debugging approach based on dynamic slicing, and [6,43] contain approaches to declarative debugging based on the ideas developed in the area of logic programming [123].

## 5 Extensions

The language Curry described so far is based on the theoretical foundations on functional logic programming surveyed in Section 2. Thus, it is a basis to show the feasibility and usefulness of these concepts in practice. Nevertheless, various extensions to this base language have been explored in recent years. In this section, we review some of them: constraints, functional patterns, and support for distributed programming. Other aspects, which are not discussed below, are default rules [108], failure [104,120], inductive programming [56], tabling and memoization [16,53], connecting databases [57,75], or proof tools for the verification of functional logic programs [45].

### 5.1 Constraints

The integration of constraints has been already mentioned. Curry provides equational constraints that are solved in conditions. Further constraint domains, like real arithmetic, Boolean, or finite domains, can be supported by adding basic constraints for these domains (e.g., see [20,32,54,55,102,106,119] for some examples). It has been shown [54,55] that functional logic languages are good frameworks to solve constraint problems in a high-level and maintainable way.

As an example demonstrating the compactness obtained by combining constraint programming and higher-order features, consider a solver for SuDoku puzzles[10] with finite domain constraints. If we represent the SuDoku matrix `m` as a list of lists of finite domain variables, the "SuDoku constraints" can be easily specified by

```
allValid (map allDifferent m) &
allValid (map allDifferent (transpose m)) &
allValid (map allDifferent (squaresOfNine m))
```

where `allDifferent` is the usual constraint stating that all variables in its argument list must have different values, `transpose` is the standard matrix transposition, and `squaresOfNine` computes the list of $3 \times 3$ sub-matrices. Then, a SuDoku puzzle can be solved with these constraints by adding the usual domain and labeling constraints (see [77] for more details).

### 5.2 Functional Patterns

We have discussed in Section 2.2 the fundamental requirement of functional languages for *constructor-based* rewrite systems. This requirement is the key for practically useful implementations and excludes rules like

```
last (xs ++ [e])  =  e                                    (last)
```

The non-constructor pattern `(xs ++ [e])` in this rule can be eliminated by moving it into the condition part (see Section 2.1):

```
last l | xs++[e] =:= l = e   where xs,e free              (lastc)
```

However, the strict equality used in (*lastc*) has the disadvantage that all list elements are completely evaluated. Hence, an expression like `last [failed,3]` (where `failed` is an expression that has no value) leads to a failure. This disadvantage can be avoided by allowing *functional patterns*, i.e., expressions containing defined functions, in arguments of a rule's left-hand side so that (*last*) becomes a valid rule. In order to base this extension on the existing foundations of functional logic programming as described so far, a functional pattern is interpreted as an abbreviation of the set of constructor terms that is the result of

[10] A SuDoku puzzle consists of a $9 \times 9$ matrix of digits between 1 and 9 so that each row, each column, and each of the nine $3 \times 3$ sub-matrices contain pairwise different digits. The challenge is to find the missing digits if some digits are given.

evaluating (by narrowing) the functional pattern. Thus, rule (*last*) abbreviates the following (infinite) set of rules:

```
last [x] = x
last [x1,x] = x
last [x1,x2,x] = x
...
```

Hence, the expression `last [failed,3]` reduces to `3` w.r.t. these rules. In order to provide a constructive implementation of this concept, [23] proposes a specific demand-driven unification procedure for functional pattern unification that can be implemented similarly to strict equality. Functional patterns are a powerful concept to express transformation problems in a high-level way. Concrete programming examples and syntactic conditions for the well-definedness of rules with functional patterns can be found in [23]. [80] exploits functional patterns for a declarative approach to process XML documents.

### 5.3 Distributed Programming

Distributed systems are of great importance but programming them is a non-trivial task. Since Curry has already features for concurrent programming via residuation, it provides a good basis that can be extended for distributed programming. For this purpose, [70] proposes *port constraints*. Conceptually, a *port* is a constraint between a multiset (of messages) and a list that is satisfied if all elements in the multiset occur in the list and vice versa. Clients use a primitive constraint `send` $m\,p$ that extends a port $p$ by an element (message) $m$. A server is just a recursive operation that processes the list of messages of a port and waits until the tail of the list is instantiated with the next message, i.e., it is rigid w.r.t. the message list. By sending messages containing logic variables, the server can instantiate them so that answers can be easily returned without explicit return channels. By making ports accessible through symbolic names similar to URLs, clients can send messages to servers running on an arbitrary host. As shown in [70], the use of ports provides a high-level concept to implement distributed systems and to integrate existing declarative programs fairly easy into distributed environments.

The port concept can be also used to implement *distributed objects* in Curry. It is well known from concurrent logic programming [124] that objects can be easily implemented as predicates processing a stream of incoming messages. The object's internal state is a parameter that may change in each recursive call that processes a message. By creating such object predicates with their own ports, one immediately obtains distributed objects that can reside on various hosts. To avoid the explicit modeling of objects by the programmer, [85] proposes ObjectCurry, a syntactic extension of Curry which allows the direct definition of templates that play the role of classes in conventional object-oriented languages. A template defines a local state and messages that modify the state and send messages to other objects. Templates can be directly transformed into standard

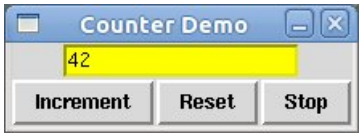


**Fig. 5.** A simple counter GUI

Curry code by a preprocessor. To provide inheritance between templates, the preprocessor implements an extended type system that includes subtyping.

## 6 Applications

Although most of the published work on functional logic programming is related to foundational aspects, functional logic languages, in particular Curry, have been used in various applications in order to demonstrate the feasibility and advantages of functional logic programming. A summary of design patterns exploiting combined functional and logic features for application programming can be found in [21]. These patterns are unique to functional logic programming and cannot be directly applied in other paradigms. For instance, the *constraint constructor* pattern exploits the fact that functional logic languages can deal with failure so that conditions about the validity of data represented by general structures can be encoded directly in the data structures rather than in applications programs. This frees the application programs from dealing with complex conditions on the constructed data. Another pattern, called *locally defined global identifier*, has been used to provide high-level interfaces to libraries dealing with complex data, like programming of dynamic web pages or graphical user interfaces (GUIs) (see below). This pattern exploits the fact that functional logic data structures can contain logic variables which are globally unique when they are introduced. This is helpful to create local structures with globally unique identifiers and leads to improved abstractions in application programs. Further design patterns and programming techniques are discussed in [21,22].

The combination of functional and logic language features are exploited in [71] for the high-level programming of GUIs. The hierarchical structure of a GUI (e.g., rows, columns, or matrices of primitive and combined widgets) is represented as a data term. This term contains call-back functions as event handlers, i.e., the use of functions as first-class objects is natural in this application. Since event handlers defined for one widget should usually influence the appearance and contents of other widgets (e.g., if a slider is moved, values shown in other widgets should change), GUIs have also a logical structure that is different from its hierarchical structure. To specify this logical structure, logic variables in data structures are handy, since a logic variable can specify relationships between different parts of a data term. As a concrete example, consider the simple counter GUI shown in Figure 5. Using a Curry library designed with these ideas, one can specify this GUI by the following data term:

```
Col [Entry [WRef val, Text "0", Background "yellow"],
     Row [Button (updateValue incrText val) [Text "Increment"],
          Button (setValue val "0")         [Text "Reset"],
          Button exitGUI                    [Text "Stop"]]]
   where val free
```

The hierarchical structure of the GUI (a column with two rows) is directly reflected in the tree structure of this term. The first argument of each `Button` is the corresponding event handler. For instance, the invocation of `exitGUI` terminates the GUI, and the invocation of `setValue` assigns a new value to the referenced widget. For this purpose, the logic variable *val* is used. Since the attribute `WRef` of the entry widget defines its origin and it is used in various event handlers, it appropriately describes the logical structure of the GUI, i.e., the dependencies between different widgets. Note that other (more low level) GUI libraries or languages (e.g., Tcl/Tk) use strings or numbers as widget references which is potentially more error prone.

Similar ideas are applied in [72] to provide a high-level programming interface for web applications (dynamic web pages). There, HTML terms are represented as data structures containing event handlers associated to submit buttons and logic variables referring to user inputs in web pages that are passed to event handlers. These high-level APIs have been used in various applications, e.g., to implement web-based learning systems [84], constructing web-based interfaces for arbitrary applications [77,78], graphical programming environments [76], documentation tools [74], and web frameworks [86] for Curry. Furthermore, Curry has also been used for embedded systems programming [82,83] with specific libraries and application-specific compilers.

## 7 Conclusions and Related Languages

In this paper we surveyed foundations of functional logic programming and their practical realization in the declarative multi-paradigm language Curry. Curry is currently the only functional logic language which is based on such strong foundations (e.g., soundness and completeness and optimal evaluation on inductively sequential programs) and that has been also used to develop larger applications. Nevertheless, there exist languages with similar goals. We briefly discuss some of them and relate them to Curry.

The language **TOY** [101] has strong connections to Curry since it is based on similar foundations (rewriting logic CRWL, demand-driven narrowing). In contrast to Curry, it is purely narrowing-based and does not cover residuation or concurrency. As a consequence, there is no distinction between constraints and Boolean expressions, but disequality constraints are used to implement the negation of equations in conditional expressions. Similarly to some implementations of Curry, TOY supports constraints over finite domains or real numbers. In addition to Curry, TOY allows higher-order patterns in the left-hand sides of program rules. Since residuation is not included in TOY, the connection with ex-

ternal operations is rather ad hoc. Furthermore, TOY does not provide a concept to encapsulate search.

**Escher** [99] is a residuation-based functional logic language. Nondeterminism is expressed by explicit disjunctions. The operational semantics is given by a set of reduction rules to evaluate operations in a demand-driven manner and simplify logical expressions. Due to its different computation model, the conditions under which completely evaluated answers can be computed are not clear.

The language **Oz** [126] is based on a computation model that extends the concurrent constraint programming paradigm [121] with features for distributed programming and stateful computations. Similarly to Escher, nondeterministic computations must be explicitly represented as disjunctions so that operations used to solve equations require different definitions than operations to rewrite expressions. In contrast to Escher and Curry, the base semantics is strict so that optimal evaluations are not directly supported.

The functional logic language **Mercury** [127] restricts logic programming features in order to provide a highly efficient implementation. In particular, predicates and functions must have distinct modes so that their arguments are either ground or unbound at call time. This inhibits the application of typical logic programming techniques, like computation with partially instantiated structures, so that some programming techniques for functional logic programming [21,71,72] cannot be applied in Mercury. This condition has been relaxed in the language **HAL** [59]. However, both languages are based on a strict operational semantics that does not support optimal evaluations.

Although many encouraging results have been obtained in recent years, the development of functional logic languages is ongoing and there are many topics for future work:

**Semantics and language concepts:** The notion of strict equality, although similar to functional languages, is for some applications too restrictive so that a more flexible handling is often desirable. Are more powerful higher-order features useful, and how can they be treated (from a semantical and implementation point of view)? Are there other concepts for concurrency and distribution together with a formal model? How can existing constraint solvers be integrated in a generic way, and which kinds of constraint domains and solvers are useful? More powerful type systems (e.g., type classes, subtypes, dependent types) and concepts for object-orientation beyond existing ones can be considered. Is the incorporation of modes useful? Are there appropriate concepts for meta-programming beyond existing approaches (e.g., libraries of the PAKCS distribution [81])?

**Implementation:** More efficient implementations, in particular, of advanced concepts such as encapsulated search, concurrency, fair scheduling, parallelism. Compilation into various target languages or target architectures, e.g., multi-core or embedded processors. Implementation of useful concepts from related languages, like Haskell's type classes, genericity, memoization. Program optimization, e.g., by powerful transformations or for restricted classes of programs. Domain-specific compilation for particular application

domains (e.g., constraints, web programming, embedded or pervasive systems). Better environments for program development. More domain-specific libraries and APIs, standardization of libraries (e.g., for Curry) to improve compatibility of different implementations, standard interfaces to external operations.

**Analysis and transformation:** Only a few approaches exist for the analysis of functional logic programs (e.g., [7,8,37,67,79,90,91,132]) so that this area deserves more studies, like termination analyses, abstract interpretation frameworks, analysis of particular properties (e.g., determinism, suspension, modes). Similarly, more powerful and practically applicable methods for transforming programs are required, like optimizing source and intermediate programs, more advanced program specialization, refactoring, and also general transformation frameworks.

**Debugging:** Some works done in this area have been already mentioned, but more work is required to provide practically useful support tools, like tracers, declarative debuggers, program slicers, or profilers for functional logic programs, integrated debugging environments, techniques and strategies for program correction and program verification.

Results and advances in these areas are also useful to support the development of more applications implemented with functional logic languages.

### Acknowledgments

I am grateful to Harald Ganzinger who put me on this research track and created a productive research environment in his group that lead to my most important contributions in this area. Furthermore, I would like to thank Sergio Antoy, Bernd Braßel, Germán Vidal, and the anonymous reviewers for their constructive remarks on a previous version of this paper.

## References

1. H. Aït-Kaci. An Overview of LIFE. In J.W. Schmidt and A.A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pp. 42–58. Springer LNCS 504, 1990.
2. H. Aït-Kaci, 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. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 795–829, 2005.
4. E. Albert, M. Hanus, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of the 7th International Conference on Logic for Programming and Automated Reasoning (LPAR 2000)*, pp. 381–398. Springer LNCS 1955, 2000.
5. E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluator for a Multi-Paradigm Declarative Language. *Journal of Functional and Logic Programming*, Vol. 2002, No. 1, 2002.

6. M. Alpuente, F.J. Correa, and M. Falaschi. A Debugging Scheme for Functional Logic Programs. *Electronic Notes in Theoretical Computer Science*, Vol. 64, 2002.
7. M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Unsatisfiability for Equational Logic Programming. *Journal of Logic Programming*, Vol. 22, No. 3, pp. 223–254, 1995.
8. M. Alpuente, M. Falaschi, and G. Vidal. A Compositional Semantic Basis for the Analysis of Equational Horn Programs. *Theoretical Computer Science*, Vol. 165, No. 1, pp. 133–169, 1996.
9. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 4, pp. 768–844, 1998.
10. M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. *Theory and Practice of Logic Programming*, Vol. 5, No. 3, pp. 273–303, 2005.
11. S. Antoy. Non-Determinism and Lazy Evaluation in Logic Programming. In *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'91)*, pp. 318–331. Springer Workshops in Computing, 1991.
12. S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.
13. S. Antoy. Optimal Non-Deterministic Functional Logic Computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pp. 16–30. Springer LNCS 1298, 1997.
14. S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 199–206. ACM Press, 2001.
15. S. Antoy. Evaluation Strategies for Functional Logic Programming. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 875–903, 2005.
16. S. Antoy and Z.M. Ariola. Narrowing the Narrowing Space. In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, pp. 1–15. Springer LNCS 1292, 1997.
17. S. Antoy, B. Braßel, and M. Hanus. Conditional Narrowing without Conditions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pp. 20–31. ACM Press, 2003.
18. S. Antoy, R. Echahed, and M. Hanus. Parallel Evaluation Strategies for Functional Logic Languages. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pp. 138–152. MIT Press, 1997.
19. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
20. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pp. 171–185. Springer LNCS 1794, 2000.
21. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.
22. S. Antoy and M. Hanus. Concurrent Distinct Choices. *Journal of Functional Programming*, Vol. 14, No. 6, pp. 657–668, 2004.
23. S. Antoy and M. Hanus. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pp. 6–22. Springer LNCS 3901, 2005.

24. S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pp. 87–101. Springer LNCS 4079, 2006.
25. S. Antoy and M. Hanus. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pp. 73–82. ACM Press, 2009.
26. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A Virtual Machine for Functional Logic Computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pp. 108–125. Springer LNCS 3474, 2005.
27. S. Antoy, M. Hanus, B. Massey, and F. Steiner. An Implementation of Narrowing Strategies. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pp. 207–217. ACM Press, 2001.
28. S. Antoy and S. Johnson. TeaBag: A Functional Logic Language Debugger. In *Proc. 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)*, pp. 4–18, Aachen (Germany), 2004. Technical Report AIB-2004-05, RWTH Aachen.
29. S. Antoy and A. Tolmach. Typed Higher-Order Narrowing without Higher-Order Strategies. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pp. 335–352. Springer LNCS 1722, 1999.
30. P. Arenas-Sánchez and M. Rodríguez-Artalejo. A Semantic Framework for Functional Logic Programming with Algebraic Polymorphic Types. In *Proc. CAAP'97*, pp. 453–464. Springer LNCS 1214, 1997.
31. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
32. R. Berghammer and S. Fischer. Implementing Relational Specifications in a Constraint Functional Logic Language. *Electronic Notes in Theoretical Computer Science*, Vol. 177, pp. 169–183, 2007.
33. 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.
34. R.S. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.
35. S. Bonnier and J. Maluszynski. Towards a Clean Amalgamation of Logic Programs with External Procedures. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 311–326. MIT Press, 1988.
36. B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing Functional Logic Computations. In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pp. 193–208. Springer LNCS 3057, 2004.
37. B. Braßel and M. Hanus. Nondeterminism Analysis of Functional Logic Programs. In *Proceedings of the International Conference on Logic Programming (ICLP 2005)*, pp. 265–279. Springer LNCS 3668, 2005.
38. B. Braßel, M. Hanus, and F. Huch. Encapsulating Non-Determinism in Functional Logic Computations. *Journal of Functional and Logic Programming*, Vol. 2004, No. 6, 2004.
39. B. Braßel, M. Hanus, F. Huch, J. Silva, and G. Vidal. Run-Time Profiling of Functional Logic Programs. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pp. 182–197. Springer LNCS 3573, 2005.

40. B. Braßel, M. Hanus, F. Huch, and G. Vidal. A Semantics for Tracing Declarative Multi-Paradigm Programs. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'04)*, pp. 179–190. ACM Press, 2004.
41. B. Braßel and F. Huch. On a Tighter Integration of Functional and Logic Programming. In *Proc. APLAS 2007*, pp. 122–138. Springer LNCS 4807, 2007.
42. B. Braßel and F. Huch. The Kiel Curry System KiCS. In *Applications of Declarative Programming and Knowledge Management*, pp. 195–205. Springer LNAI 5437, 2009.
43. R. Caballero and M. Rodríguez-Artalejo. DDT: a Declarative Debugging Tool for Functional-Logic Languages. In *Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, pp. 70–84. Springer LNCS 2998, 2004.
44. P.H. Cheong and L. Fribourg. Implementation of Narrowing: The Prolog-Based Approach. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic programming languages: constraints, functions, and objects*, pp. 1–20. MIT Press, 1993.
45. J.M. Cleva, J. Leach, and F.J. López-Fraguas. A logic programming approach to the verification of functional-logic programs. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pp. 9–19. ACM Press, 2004.
46. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pp. 207–212, 1982.
47. R. del Vado Virseda. A Demand-Driven Narrowing Calculus with Overlapping Definitional Trees. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pp. 253–263. ACM Press, 2003.
48. 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.
49. R. Echahed and J.-C. Janodet. Admissible Graph Rewriting and Narrowing. In *Proc. Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pp. 325–340, 1998.
50. S. Escobar. Refining Weakly Outermost-Needed Rewriting and Narrowing. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pp. 113–123. ACM Press, 2003.
51. S. Escobar. Implementing Natural Rewriting and Narrowing Efficiently. In *Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, pp. 147–162. Springer LNCS 2998, 2004.
52. S. Escobar, J. Meseguer, and P. Thati. Narrowing adn Rewriting Logic: from Foundations to Applications. *Electronic Notes in Theoretical Computer Science*, Vol. 177, pp. 5–33, 2007.
53. S. España and V. Estruch. A Memoizing Semantics for Functional Logic Languages. In *Proc. ESOP 2004*, pp. 109–123. Springer LNCS 2986, 2004.
54. A.J. Fernández, M.T. Hortalá-González, and F. Sáenz-Pérez. Solving Combinatorial Problems with a Constraint Functional Logic Language. In *Proc. of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, pp. 320–338. Springer LNCS 2562, 2003.

55. A.J. Fernández, M.T. Hortalá-González, F. Sáenz-Pérez, and R. del Vado-Vírseda. Constraint Functional Logic Programming over Finite Domains. *Theory and Practice of Logic Programming*, Vol. 7, No. 5, pp. 537–582, 2007.
56. C. Ferri, J. Hernández, and M.J. Ramírez. Incremental Learning of Functional Logic Programs. In *Proc. of the 5th International Symposium on Functional and Logic Programming (FLOPS 2001)*, pp. 233–247. Springer LNCS 2024, 2001.
57. S. Fischer. A Functional Logic Database Library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 54–59. ACM Press, 2005.
58. 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.
59. M.J. García de la Banda, B. Demoen, K. Marriott, and P.J. Stuckey. To the Gates of HAL: A HAL Tutorial. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 47–66. Springer LNCS 2441, 2002.
60. 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.
61. J.C. Gonzáles-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. Polymorphic Types in Functional Logic Programming. *Journal of Functional and Logic Programming*, Vol. 2001, No. 1, 2001.
62. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, Vol. 40, pp. 47–87, 1999.
63. J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. A Higher Order Rewriting Logic for Functional Logic Programming. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pp. 153–167. MIT Press, 1997.
64. A. Habel and D. Plump. Term Graph Narrowing. *Mathematical Structures in Computer Science*, Vol. 6, No. 6, pp. 649–676, 1996.
65. 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.
66. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
67. M. Hanus. Analysis of Residuating Logic Programs. *Journal of Logic Programming*, Vol. 24, No. 3, pp. 161–199, 1995.
68. 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.
69. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
70. M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pp. 376–395. Springer LNCS 1702, 1999.
71. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.

72. M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.
73. M. Hanus. Reduction Strategies for Declarative Programming. In B. Gramlich and S. Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 57. Elsevier Science Publishers, 2001.
74. M. Hanus. CurryDoc: A Documentation Tool for Declarative Programs. In *Proc. 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, pp. 225–228. Research Report UDMI/18/2002/RR, University of Udine, 2002.
75. M. Hanus. Dynamic Predicates in Functional Logic Programs. *Journal of Functional and Logic Programming*, Vol. 2004, No. 5, 2004.
76. M. Hanus. A Generic Analysis Environment for Declarative Programs. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 43–48. ACM Press, 2005.
77. M. Hanus. Type-Oriented Construction of Web User Interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pp. 27–38. ACM Press, 2006.
78. M. Hanus. Putting Declarative Programming into the Web: Translating Curry to JavaScript. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pp. 155–166. ACM Press, 2007.
79. M. Hanus. Call Pattern Analysis for Functional Logic Programs. In *Proceedings of the 10th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'08)*, pp. 67–78. ACM Press, 2008.
80. M. Hanus. Declarative Processing of Semistructured Web Data. Technical Report 1103, Christian-Albrechts-Universität Kiel, 2011.
81. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2010.
82. M. Hanus and K. Höppner. Programming Autonomous Robots in Curry. *Electronic Notes in Theoretical Computer Science*, Vol. 76, 2002.
83. M. Hanus, K. Höppner, and F. Huch. Towards Translating Embedded Curry to C. *Electronic Notes in Theoretical Computer Science*, Vol. 86, No. 3, 2003.
84. M. Hanus and F. Huch. An Open System to Support Web-based Learning. In *Proc. 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, pp. 269–282. Technical Report DSIC-II/13/03, Universidad Politécnica de Valencia, 2003.
85. M. Hanus, F. Huch, and P. Niederau. An Object-Oriented Extension of the Declarative Multi-Paradigm Language Curry. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pp. 89–106. Springer LNCS 2011, 2001.
86. M. Hanus and S. Koschnicke. An ER-based Framework for Declarative Web Programming. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, pp. 201–216. Springer LNCS 5937, 2010.
87. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, Vol. 9, No. 1, pp. 33–75, 1999.
88. M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming*, Vol. 1999, No. 6, 1999.

89. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.
90. M. Hanus and F. Steiner. Type-based Nondeterminism Checking in Functional Logic Programs. In *Proc. of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pp. 202–213. ACM Press, 2000.
91. M. Hanus and F. Zartmann. Mode Analysis of Functional Logic Programs. In *Proc. 1st International Static Analysis Symposium*, pp. 26–42. Springer LNCS 864, 1994.
92. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.curry-language.org`, 2011.
93. S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNCS 353, 1989.
94. H. Hussmann. Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. *Journal of Logic Programming*, Vol. 12, pp. 237–255, 1992.
95. T. Ida and K. Nakahara. Leftmost outside-in narrowing calculi. *Journal of Functional Programming*, Vol. 7, No. 2, pp. 129–161, 1997.
96. J.A. Jiménez-Martin, J. Marino-Carballo, and J.J. Moreno-Navarro. Efficient Compilation of Lazy Narrowing into Prolog. In *Proc. Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR'92)*, pp. 253–270. Springer Workshops in Computing Series, 1992.
97. P. Julián Iranzo and C. Villamizar Lamus. Analysing Definitional Trees: Looking for Determinism. In *Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, pp. 55–69. Springer LNCS 2998, 2004.
98. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pp. 144–154. ACM Press, 1993.
99. J. Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, No. 3, pp. 1–49, 1999.
100. R. Loogen, F. López 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.
101. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
102. F.J. López-Fraguas, M. Rodríguez-Artalejo, and R. del Vado Virseda. A lazy narrowing calculus for declarative constraint programming. In *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pp. 43–54. ACM Press, 2004.
103. F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A Simple Rewrite Notion for Call-time Choice Semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pp. 197–208. ACM Press, 2007.
104. F.J. López-Fraguas and J. Sánchez-Hernández. A Proof Theoretic Approach to Failure in Functional Logic Programming. *Theory and Practice of Logic Programming*, Vol. 4, No. 1, pp. 41–74, 2004.
105. W. Lux. Implementing Encapsulated Search for a Lazy Functional Logic Language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pp. 100–113. Springer LNCS 1722, 1999.

106. W. Lux. Adding Linear Constraints over Real Numbers to Curry. In *Proc. of the 5th International Symposium on Functional and Logic Programming (FLOPS 2001)*, pp. 185–200. Springer LNCS 2024, 2001.
107. A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, Vol. 167, No. 1,2, pp. 95–130, 1996.
108. J.J. Moreno-Navarro. Default Rules: An Extension of Constructive Negation for Narrowing-based Languages. In *Proc. Eleventh International Conference on Logic Programming*, pp. 535–549. MIT Press, 1994.
109. 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.
110. 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.
111. K. Nakahara, A. Middeldorp, and T. Ida. A Complete Narrowing Calculus for Higher-Order Functional Logic Programming. In *Proc. of the 7th International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'95)*, pp. 97–114. Springer LNCS 982, 1995.
112. H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, Vol. 4, No. 3, pp. 337–370, 1994.
113. C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pp. 123–134. ACM Press, 2004.
114. M.J. O'Donnell. *Computing in Systems Described by Equations*. Springer LNCS 58, 1977.
115. M.J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.
116. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.
117. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
118. J.C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*, pp. 717–740. ACM Press, 1972.
119. M. Rodríguez-Artalejo. Functional and Constraint Logic Programming. In *Constraints in Computational Logics: Theory and Applications (CCL'99)*, pp. 202–270. Springer LNCS 2002, 2001.
120. J. Sánchez-Hernández. Constructive Failure in Functional-Logic Programming: From Theory to Implementation. *Journal of Universal Computer Science*, Vol. 12, No. 11, pp. 1574–1593, 2006.
121. V.A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
122. R.C. Sekar and I.V. Ramakrishnan. Programming in Equational Logic: Beyond Strong Sequentiality. *Information and Computation*, Vol. 104, No. 1, pp. 78–109, 1993.
123. E. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.
124. E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pp. 251–273. MIT Press, 1987.

125. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *Journal of the ACM*, Vol. 21, No. 4, pp. 622–642, 1974.
126. 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.
127. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, Vol. 29, No. 1-3, pp. 17–64, 1996.
128. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
129. 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.
130. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.
131. D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.
132. F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In *Proc. of the 4th International Symposium on Static Analysis (SAS'97)*, pp. 141–156. Springer LNCS 1302, 1997.