© Springer-Verlag
In *Proc. of the International Conference on Logic Programming, ICLP 2007.*
Springer LNCS 4670, pp. 45-75, 2007

Multi-paradigm Declarative Languages[⋆]

Michael Hanus

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

Abstract. Declarative programming languages advocate a programming style expressing the properties of problems and their solutions rather than how to compute individual solutions. Depending on the underlying formalism to express such properties, one can distinguish different classes of declarative languages, like functional, logic, or constraint programming languages. This paper surveys approaches to combine these different classes into a single programming language.

1 Introduction

Compared to traditional imperative languages, declarative programming languages provide a higher and more abstract level of programming that leads to reliable and maintainable programs. This is mainly due to the fact that, in contrast to imperative programming, one does not describe *how* to obtain a solution to a problem by performing a sequence of steps but *what* are the properties of the problem and the expected solutions. In order to define a concrete declarative language, one has to fix a base formalism that has a clear mathematical foundation as well as a reasonable execution model (since we consider a *programming language* rather than a *specification language*). Depending on such formalisms, the following important classes of declarative languages can be distinguished:

– *Functional languages:* They are based on the lambda calculus and term rewriting. Programs consist of functions defined by equations that are used from left to right to evaluate expressions.
– *Logic languages:* They are based on a subset of predicate logic to ensure an effective execution model (linear resolution). Programs consist of predicates defined by definite clauses. The execution model is goal solving based on the resolution principle.
– *Constraint languages:* They are based on constraint structures with specific methods to solve constraints. Programs consist of specifications of constraints of the considered application domain based on a set of primitive constraints and appropriate combinators. Constraint languages are often embedded in other languages where logic programming is a natural candidate. In this case, *constraint logic programming* [61] can be considered as a generalization of logic programming where unification on Herbrand terms is considered as a specific built-in constraint solver.

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

The different declarative programming paradigms offer a variety of programming concepts to the programmer. For instance, functional programming emphasizes generic programming using higher-order functions and polymorphic typing, and efficient and (under particular conditions) optimal evaluation strategies using demand-driven evaluation, which contributes to modularity in programming [59]. Logic programming supports the computation with partial information (logic variables) and nondeterministic search for solutions, where constraint programming adds efficient constraint solving capabilities for particular domains. Since all these features have been shown to be useful in application programming and declarative languages are based on common grounds, it is a natural idea to combine these worlds of programming into a single *multi-paradigm declarative language*. However, the interactions between the different features are complex in detail so that the concrete design of a multi-paradigm declarative language is non-trivial. This is demonstrated by many different proposals and a lot of research work on the semantics, operational principles, and implementation of multi-paradigm declarative languages since more than two decades. In the following, we survey some of these proposals.

One can find two basic approaches to amalgamate functional and logic languages: either extend a functional language with logic programming features or extend a logic language with features for functional programming. Since functions can be considered as specific relations, there is a straightforward way to implement the second approach: extend a logic language with syntactic sugar to allow functional notation (e.g., defining equations, nested functional expressions) which is translated by some preprocessor into the logic kernel language. A recent approach of this kind is [25] where functional notation is added to Ciao-Prolog. The language Mercury [83] is based on a logic programming syntax with functional and higher-order extensions. Since Mercury is designed towards a highly efficient implementation, typical logic programming features are restricted. 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 developed for functional logic programming languages [11, 43, 44] can not be applied. This condition has been relaxed in the language HAL [33] which adds constraint solving possibilities. However, Mercury as well as HAL are based on a strict operational semantics that does not support optimal evaluation as in functional programming. This is also true for Oz [82]. The computation model of Oz extends the concurrent constraint programming paradigm [78] with features for distributed programming and stateful computations. It provides functional notation but restricts their use compared to predicates, i.e., function calls are suspended if the arguments are not instantiated in order to reduce them in a deterministic way. Thus, nondeterministic computations must be explicitly represented as disjunctions so that functions used to solve equations require different definitions than functions to rewrite expressions. In some sense, these approaches do not exploit the semantical information provided by the presence of functions.

Extensions of functional languages with logic programming features try to retain the efficient demand-driven computation strategy for purely functional computations and add some additional mechanism for the extended features. For instance, Escher [63] is a functional logic language with a Haskell-like syntax [75] and a demand-driven reduction strategy. Similarly to Oz, function calls are suspended if they are not instantiated enough for deterministic reduction, i.e., nondeterminism must be expressed by explicit disjunctions. The operational semantics is given by a set of reduction rules to evaluate functions in a demand-driven manner and to simplify logical expressions. The languages Curry [41, 58] and TOY [66] try to overcome the restrictions on evaluating function calls so that there is no need to implement similar concepts as a function and a predicate, depending on their use. For this purpose, functions can be called, similarly to predicates, with unknown arguments that are instantiated in order to apply a rule. This mechanism, called *narrowing*, amalgamates the functional concept of reduction with unification and nondeterministic search from logic programming. Moreover, if unification on terms is generalized to constraint solving, features of constraint programming are also covered. Based on the narrowing principle, one can define declarative languages integrating the good features of the individual paradigms, in particular, with a sound and complete operational semantics that is optimal for a large class of programs [9].

In the following, we survey important concepts of such multi-paradigm declarative languages. As a concrete example, we consider the language Curry that is based on these principles and intended to provide a common platform for research, teaching, and application of integrated functional logic languages.

Since this paper is a survey of limited size, not all of the numerous papers in this area can be mentioned and relevant topics are only sketched. Interested readers might look into the references for more details. In particular, there exist other surveys on particular topics related to this paper. [38] is a survey on the development and the implementation of various evaluation strategies for functional logic languages that have been explored until a decade ago. [7] contains a good survey on more recent evaluation strategies and classes of functional logic programs. [77] 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 relevant in current languages. Section 3 discusses practical aspects of multi-paradigm languages. Section 4 contains references to applications of such languages. Finally, Section 5 contains our conclusions.

2 Foundations of Functional Logic Programming

2.1 Basic Concepts

In the following, we use functional programming as our starting point, i.e., we develop functional logic languages by extending functional languages with features for logic programming.

A *functional program* is a set of functions defined by *equations* or *rules*. A *functional computation* consists of replacing subexpressions by equal (w.r.t. the function definitions) subexpressions until no more replacements (or *reductions*) are possible and a value or normal form is obtained. For instance, consider the function `double` defined by[1]

```
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 functions later). In a similar way, one can evaluate nested expressions (where the subexpression to be replaced is underlined):

$$\underline{\texttt{double (1+2)}} \;\rightarrow\; \underline{\texttt{(1+2)}}\texttt{+(1+2)} \;\rightarrow\; \texttt{3+}\underline{\texttt{(1+2)}} \;\rightarrow\; \underline{\texttt{3+3}} \;\rightarrow\; \texttt{6}$$

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

$$\underline{\texttt{double (1+2)}} \;\rightarrow\; \texttt{(1+2)+}\underline{\texttt{(1+2)}} \;\rightarrow\; \underline{\texttt{(1+2)}}\texttt{+3} \;\rightarrow\; \underline{\texttt{3+3}} \;\rightarrow\; \texttt{6}$$

In this case, both derivations lead to the same result. This indicates a *fundamental property of declarative languages*, also termed *referential transparency*: 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 since one can also evaluate the argument of `double` before applying its defining equation:

$$\texttt{double }\underline{\texttt{(1+2)}} \;\rightarrow\; \underline{\texttt{double 3}} \;\rightarrow\; \underline{\texttt{3+3}} \;\rightarrow\; \texttt{6}$$

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, given 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 find appropriate evaluation strategies. A detailed account of the development of such strategies can be found in [38]. In the following, we will survey only 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 sup-

[1] For concrete examples in this paper, we use the syntax of Curry which is very similar to the syntax of Haskell [75], 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 a function f to an expression e is denoted by juxtaposition ("f e"). Moreover, binary operators like "`+`" are written infix.

port 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
```

Functions 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.:

$\texttt{not}\ \underline{\texttt{(not False)}} \quad \rightarrow \quad \underline{\texttt{not True}} \quad \rightarrow \quad \texttt{False}$

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 "`e:l`" consisting of a first element `e` and a list `l`:

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

The type "`List a`" is usually written as `[a]` and finite lists e_1`:`e_2`:`$\ldots$`:`e_n`:[]` are written as `[`e_1`,`e_2`,`$\ldots$`,`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 functions on lists. For instance, the behavior of a function `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`]` $= l$.[2] Based on this specification, one can define a function and verify that this definition satisfies the given specification (e.g., by inductive proofs as shown in [20]). 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
```

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

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 functions 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.

2.2 Narrowing

Techniques for goal solving are well developed in the area of logic programming. Since functional languages advocate the equational definition of functions, it is a natural idea to integrate both paradigms by adding an equality predicate to logic programs, leading to *equational logic programming* [73, 74]. 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 [81], 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 [18, 29] 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 survey, we consider a many-sorted *signature* Σ 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 [29] 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 functions 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 [38, 74, 81] 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 the restriction to constructor-based programs since this supports the development of efficient and practically useful evaluation strategies (see below).
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*. 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 [29] 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* σ with $\sigma(x_i) = t_i$ $(i = 1, \ldots, n)$ and $\sigma(x) = x$ for all other variables x. A substitution σ 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,[3] and σ 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 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 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 σ 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* σ 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 [38] 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

[3] 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.

applied rule. As shown in [9], 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 O     y = y
add (S x) y = S (add x y)

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

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

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

Further answers can be obtained by instantiating `v` to `(S...)`. This requires the evaluation of the subterm `(add w O)` 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 $\{\texttt{v} \mapsto \texttt{S z}, \texttt{w} \mapsto \texttt{O}\}$:

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

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

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

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 `v` to `S z` in the first narrowing step:

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

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 O)` 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 [9]. Furthermore, needed narrowing steps can be efficiently computed. Therefore, it has become the basis of modern functional logic languages.[4]

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

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 a function under evaluation (starting from an outermost function). 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 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 [4] 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 [5] considers patterns partially ordered by subsumption (the *subsumption ordering* on terms is defined by $t \leq \sigma(t)$ for a term t and substitution σ).

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 π (and π is called a *child* of π'), 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 π differ from each other only at a common position, called the *inductive position*, which is the position of a variable in π.[5]

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

[5] 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.

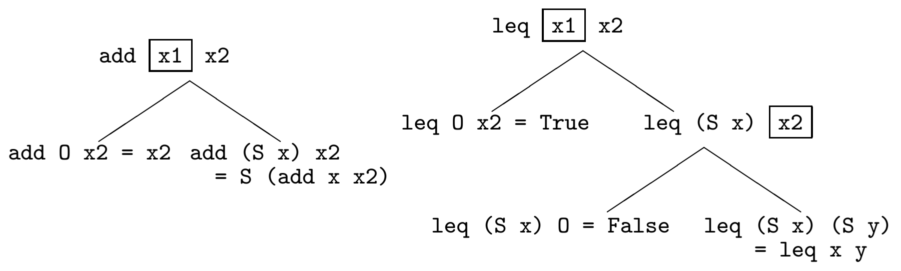

Fig. 1. Definitional trees of the operations `add` and `leq`

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 functions in functional logic programs are inductively sequential. Thus, needed narrowing is applicable to most functions, although extensions are useful for particular functions (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 contain the corresponding rules. For instance, the definitional trees of the operations `add` and `leq`, defined in Example 2, are illustrated in Figure 1.

The formal definition of needed narrowing is based on definitional trees and can be found in [9]. A definitional tree can be computed at compile time (see [7, 41] 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 π the following case distinction:

– If π is a leaf, we apply the corresponding rule.
– If π 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 π' of π having c at the inductive position, we proceed by examining π'. 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 π' of π and proceed with π'.
  3. If $t|_p$ is operation-rooted, we recursively apply the computation of a needed narrowing step to $\sigma(t|_p)$, where σ 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 $t_1 \texttt{=:=} t_2$ 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 so that reflexivity is not a feasible property of equational constraints (see [34] for a more detailed discussion on this topic).

Strict equality can be defined as a binary function 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.

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

Thus, it is sufficient to consider strict equality as any other function. 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 3.3).

Now we can state the main properties of needed narrowing. A (correct) *solution* for an equation $t_1 \texttt{=:=} t_2$ is a constructor substitution σ (note that constructor substitutions are desired in practice since a broader class of solutions would contain unevaluated or undefined expressions) if $\sigma(t_1) \texttt{=:=} \sigma(t_2) \overset{*}{\to} \texttt{Success}$. 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 ([9]). *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 σ is a solution for e.*
2. *(Completeness) For each solution σ 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 σ and σ' 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 of all possible narrowing derivations* if derivations on common subterms are shared (a standard implementation technique in non-strict functional languages) [9, 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 are inductively sequential, there are also functions where it is more convenient to relax this requirement. An 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 σ 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 derivation. `or` has no definitional tree and, thus, needed narrowing can not be applied. □

In rewriting, several normalizing strategies for weakly orthogonal TRSs have been proposed, like parallel outermost [72] or weakly needed [80] 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 [4, 8, 64]. 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. Weakly needed narrowing is no longer optimal in the sense of needed narrowing but sound and complete for weakly orthogonal TRS in the sense of Theorem 1 [8].

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 reasonable 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 *non-deterministic*. 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 can deal with such nondeterministic operations. If operations are interpreted as mappings from values into sets of values (actually, due to the presence of recursive non-strict functions, algebraic structures with cones of partially ordered sets are used instead of sets, see [36] 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 recent functional logic languages (whereas older languages put confluence requirements on their programs). The following example discusses this in more detail.

Example 5. 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 one reasons 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 a sorting function `psort` 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 based only on the left-hand sides of the rules but does not exploit confluence). Since in an expression like `sorted (perm xs)` the argument `(perm xs)` 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 [5, 36] for further examples). Antoy [5] shows that desirable properties of needed narrowing can be transferred to programs with nondeterministic functions 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 [5] for more details).

A subtle aspect of nondeterministic operations is their treatment if they are passed as arguments. For instance, consider the 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 rewrite derivations resulting in the values `0`, `1`, `1`, and `2`. The former behavior is referred to as *call-time choice* semantics [60] 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 [7] 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 5 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. Due to these reasons, current functional logic languages usually adopt the call-time choice semantics.

Conditional Rules. The narrowing strategies presented so far are 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 $s \,\texttt{=:=}\, t$. In order to apply weakly needed narrowing to conditional rules, one can transform a conditional rule of the form

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

into an unconditional rule

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

where the "*cond*itional" is defined by $cond(\texttt{Success}, x) \to x$. Actually, Antoy [6] has shown a systematic method to translate any conditional constructor-based TRS into an overlapping inductively sequential TRS performing equivalent computations.

2.3 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. [36] 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 [28].

To deal with non-strict functions, CRWL considers signatures $\Sigma_\perp$ that are extended by a special symbol $\perp$ to represent *undefined values.* For instance, $\mathcal{T}(\mathcal{C} \cup \{\perp\}, \mathcal{X})$ denotes the set of partial constructor terms, e.g., `1:2:`$\perp$ 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* $e_1 \mathtt{=:=} e_2$ 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:

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

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

(Bottom) $\quad e \to \perp \quad$ for any $e \in \mathcal{T}(\mathcal{C} \cup \mathcal{F} \cup \{\perp\}, \mathcal{X})$

(Restricted reflexivity) $\quad x \to x \quad$ for any variable $x \in \mathcal{X}$

(Decomposition) $\quad \dfrac{e_1 \to t_1 \ \cdots \ e_n \to t_n}{c(e_1, \ldots, e_n) \to c(t_1, \ldots, t_n)}$
for any $c/n \in \mathcal{C}$, $t_i \in \mathcal{T}(\mathcal{C} \cup \{\perp\}, \mathcal{X})$

(Function reduction) $\quad \dfrac{e_1 \to t_1 \ \cdots \ e_n \to t_n \quad C \quad r \to t}{f(e_1, \ldots, e_n) \to t}$
for any $f(t_1, \ldots, t_n) \to r \Leftarrow C \in [\mathcal{R}]_\perp$ and $t \neq \perp$

(Joinability) $\quad \dfrac{e_1 \to t \qquad e_2 \to t}{e_1 \mathtt{=:=} e_2} \quad$ for any total term $t \in \mathcal{T}(\mathcal{C}, \mathcal{X})$

The first rule specifies that $\perp$ approximates any expression. The condition $t \neq \perp$ 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 functions with a call-time choice semantics. Functions might have non-strict arguments that are not evaluated since the corresponding actual arguments can be derived to $\perp$ by the first rule. If the value of an argument is required to evaluate the right-hand side of a function's rule, it must be evaluated to a partial constructor term before it is passed to the right-hand side (since $[\mathcal{R}]_\perp$ 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, [36] defines a lazy narrowing calculus that reflects this behavior.

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 [27] and has been extended in various directions, e.g., higher-order operations [37], algebraic types [17], polymorphic types [35], failure [68], constraints [67] etc. An account on CRWL and its applications can be found in [77].

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 [63], Le Fun [2], Life [1], NUE-Prolog [71], and Oz [82]. Since the residuation principle evaluates function calls by deterministic reduction steps, nondeterministic search must be encoded by predicates [1, 2, 71] or disjunctions [63, 82]. 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 6. Consider Example 2 together with the operation

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

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

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

Thus, the solution $\{\texttt{y} \mapsto \texttt{S O}\}$ is computed by switching between the residuating function `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 functions. 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 functions 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 [39] for more details). On the other hand, residuation offers a concurrent evaluation principle with synchronization on logic variables (sometimes also called *declarative concurrency* [84]) and a conceptually clean method to connect *external functions* to declarative programs [21] (note that narrowing requires functions to be explicitly defined by rewrite rules). Therefore, it is desirable to integrate both principles in a single framework. This has been proposed in [41] 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 function under evaluation. For instance, if the branch of `add` in Figure 1 has a rigid tag, then `add` is evaluated as shown in Example 6.

3 Aspects of Multi-paradigm Languages

This section discusses some aspects of multi-paradigm languages that are relevant for their use in application programming. As before, we use the language Curry for concrete examples. Its syntax has been already introduced in an informal manner. Conceptually, a Curry program is 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 weakly needed narrowing with sharing and residuation. 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 [14, 15, 56]). Therefore, the programmer can concentrate on the declarative meaning of programs and needs less attention to the consequences of the particular evaluation strategy (see [45] for a more detailed discussion).

3.1 External Operations

Operations that are externally defined, i.e., not implemented by explicit rules, like basic arithmetic operators or I/O operations, can not be handled by narrowing. Therefore, residuation is an appropriate model to connect external operations in a conceptually clean way (see also [21]): their semantics can be considered as defined by a possibly infinite set of rules (e.g., see the definition of "+" in Section 2.1) whose behavior is implemented in some other programming language. Usually, external operations can not deal with unevaluated arguments possibly containing logic variables. Thus, the arguments of external operations are reduced to ground values before they are called. If some arguments are not ground but contain logic variables, the call is suspended until the variables are bound to ground values, which corresponds to residuation.

3.2 Higher-order Operations

The use of higher-order operations, i.e., operations that take other operations as arguments or yields them as results, is an important programming technique in functional languages so that it should be also covered by multi-paradigm declarative languages. Typical examples are the mapping of a function 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 [87] by defining a predicate *apply* that implements the application of an arbitrary function of the program to an expression. This technique is also known as "defunctionalization" [76] and enough to support the higher-order features of current functional languages (e.g., lambda abstractions can be replaced by new function definitions). An important difference to purely functional languages shows up when the function to be applied is a logic variable. In this case, one can instantiate this variable to all possible functions occurring in the program [37]. 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 [16, 35]. Another option is the instantiation of function variables to (well-typed) lambda terms in order to cover programs that can reason about bindings and block structure [55]. Since all these options might result in huge search spaces due to function instantiation and their feasibility and usefulness for larger application programs is not clear, one can also choose a more pragmatic solution: function application is rigid, i.e., it suspends if the functional argument is a logic variable.

3.3 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 general, a syntactic extension is not necessary to include constraints. For instance, the language Curry has no specific constructs for constraints but constraints are simply expressions of type `Success`, i.e., the equational constraint "`=:=`" is a function of type "`a -> a -> Success`", and the *concurrent conjunction* "`&`" on constraints that evaluates both arguments in a non-specified order (see Section 2.4) is a function of type "`Success -> Success -> Success`".

If 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 higher-order functions, 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. However, functional logic languages like Curry, Escher, or TOY use a more efficient implementation of strict equality. 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 can be avoided by binding one variable to the other, similar to logic programming.

One can easily integrate the features of constraint programming by adding basic constraints that deal with other constraint domains, like real arithmetic, Boolean, or finite domain constraints. Thus, typical applications of constraint logic programming can be covered and combined with features of lazy higher-order programming [10, 19, 30, 31, 67, 70, 77]. As an example demonstrating the compactness obtained by combining constraint programming with higher-order features, consider a solver for SuDoku puzzles[6] 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))
```

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

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 3×3 sub-matrices. Then, a SuDoku puzzle can be solved with these constraints by adding the usual domain and labeling constraints (see [49] for more details).

3.4 Function 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 *function 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 function pattern is interpreted as an abbreviation of the set of constructor terms that is the result of evaluating (by narrowing) the function 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, [13] proposes a specific demand-driven unification procedure for function pattern unification that can be implemented similarly to strict equality. Function 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 function patterns can be found in [13].

3.5 Encapsulating 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 [86]. The monadic

I/O approach is also taken in languages like Curry, Escher, or Mercury. 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 must be avoided. 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, one can introduce a primitive search operator [57, 79] that returns nondeterministic choices as data so that typical search operators of Prolog, like `findall`, `once`, or negation-as-failure, can be implemented using this primitive. Unfortunately, the combination with demand-driven evaluation and sharing causes some complications. For instance, in an expression like

```
let y = coin in findall (...y...)
```

it is not obvious whether the evaluation of `coin` (introduced outside but demanded inside the search operator) should be encapsulated or not. Furthermore, the order of the solutions might depend on the evaluation time. These and more peculiarities are discussed in [22] where another primitive search operator is proposed:

```
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 nontermination of the search operator if one is interested in only one solution.

3.6 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 compile functional logic languages with demand-driven evaluation strategies into Prolog (e.g., [3, 10, 26, 40, 62, 64]). Narrowing-based strategies can be compiled into pure Prolog whereas residuation (as necessary for external operations, see Section 3.1) demands for coroutining.[7] The compilation into Prolog 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.

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 various approaches to use other target languages than Prolog. For instance, [15] presents techniques to compile functional logic programs into Java programs that implement a fair search for solutions, and [23] proposes a translation of Curry programs into Haskell programs that offers the primitive search operator `getSearchTree` introduced in Section 3.5. *Virtual machines* to compile functional logic programs are proposed in [14, 56, 69].

4 Applications

Since multi-paradigm declarative languages amalgamate the most important declarative paradigms, their application areas cover the areas of languages belonging to the individual paradigms. Therefore, we discuss in this section only applications that demonstrate the feasibility and advantages of multi-paradigm declarative programming.

A summary of design patterns exploiting combined functional and logic features for application programming can be found in [11]. These patterns are unique to *functional logic* programming and can not 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 application 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 [11, 12].

The combination of functional and logic language features are exploited in [43] 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

[7] Note that external operations in Prolog do not use coroutining since they are implemented in a non-declarative way.

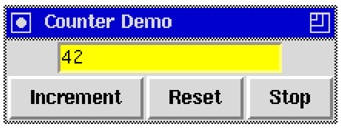

Fig. 2. A simple counter GUI

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 2. Using a 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 (second argument) to the referenced widget (first argument). 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 [44] 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 [52], constructing web-based interfaces for arbitrary applications [49] (there, the effectiveness of the multi-paradigm declarative programming style is demonstrated by a SuDoku solver with a web-based interface where the complete program consists of 20 lines of code), graphical programming environments [48, 54], and documentation tools [46]. Furthermore, there are proposals to use multi-paradigm languages for high-level distributed programming [42, 85], programming of embedded systems [50, 51], object-oriented programming [53, 82], or declarative APIs to databases [32, 47].

5 Conclusions

In this paper we surveyed the main ideas of multi-paradigm declarative languages, their foundations, and some practical aspects of such languages for application programming. As a concrete example, we used the multi-paradigm declarative language Curry. Curry amalgamates functional, logic, constraint, and concurrent programming features, it is based on strong foundations (e.g., soundness and completeness and optimal evaluation on inductively sequential programs) and it has been also used to develop larger applications. For the latter, Curry also offers useful features, like modules, strong typing, polymorphism, and declarative I/O, that are not described in this paper since they are not specific to multi-paradigm declarative programming (see [58] for such features).

We conclude with a summary of the advantages of combining different declarative paradigms in a single language. Although functions can be considered as predicates (thus, logic programming is sometimes considered as more general than functional programming), functional notation should not be used only as syntactic sugar: we have seen that the properties of functions (i.e., functional dependencies between input and output arguments) can be exploited to construct more efficient evaluation strategies without loosing generality. For instance, needed narrowing ensures soundness and completeness in the sense of logic programming and it is also optimal, whereas similar results are not available for pure logic programs. As a consequence, functional logic languages combine the flexibility of logic programming with the efficiency of functional programming. This leads to a more declarative style of programming without loosing efficiency. For instance, most functional logic languages do not have a Prolog-like "cut" operator since functions can be interpreted as a declarative replacement for it (see also [24, 65]). Moreover, searching for solutions with a demand-driven evaluation strategy results in a demand-driven search strategy that can considerably reduce the search space. Finally, narrowing can be appropriately combined with constraint solving and residuation. The latter provides for a declarative integration of external operations and concurrent programming techniques.

Acknowledgments

I am grateful to thank Sergio Antoy, Bernd Braßel, Sebastian Fischer and Germán Vidal for their comments related to this survey.

References

1. H. Aït-Kaci. An Overview of LIFE. In J.W. Schmidt and A.A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pp. 42–58. Springer LNCS 504, 1990.
2. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and Functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 17–23, San Francisco, 1987.

3. S. Antoy. 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.
4. S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.
5. 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.
6. 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.
7. S. Antoy. Evaluation Strategies for Functional Logic Programming. *Journal of Symbolic Computation*, Vol. 40, No. 1, pp. 875–903, 2005.
8. 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.
9. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
10. 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.
11. 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.
12. S. Antoy and M. Hanus. Concurrent Distinct Choices. *Journal of Functional Programming*, Vol. 14, No. 6, pp. 657–668, 2004.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
19. 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.
20. R.S. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

21. 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.
22. 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.
23. B. Braßel and F. Huch. Translating Curry to Haskell. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 60–65. ACM Press, 2005.
24. R. Caballero and Y. García-Ruiz. Implementing Dynamic-Cut in TOY. *Electronic Notes in Theoretical Computer Science*, Vol. 177, pp. 153–168, 2007.
25. A. Casas, D. Cabeza, and M.V. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation, and Higher-Order in LP Systems. In *Proc. of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006)*, pp. 146–162. Springer LNCS 3945, 2006.
26. 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.
27. 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.
28. 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.
29. 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.
30. 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.
31. 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 (to appear)*, 2007.
32. 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.
33. 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.
34. 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.
35. 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.

36. 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.
37. 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.
38. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
39. M. Hanus. Analysis of Residuating Logic Programs. *Journal of Logic Programming*, Vol. 24, No. 3, pp. 161–199, 1995.
40. 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.
41. 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.
42. 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.
43. 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.
44. 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.
45. 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.
46. 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.
47. M. Hanus. Dynamic Predicates in Functional Logic Programs. *Journal of Functional and Logic Programming*, Vol. 2004, No. 5, 2004.
48. 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.
49. 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.
50. M. Hanus and K. Höppner. Programming Autonomous Robots in Curry. *Electronic Notes in Theoretical Computer Science*, Vol. 76, 2002.
51. M. Hanus, K. Höppner, and F. Huch. Towards Translating Embedded Curry to C. *Electronic Notes in Theoretical Computer Science*, Vol. 86, No. 3, 2003.
52. 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.

53. 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.
54. M. Hanus and J. Koj. An Integrated Development Environment for Declarative Multi-Paradigm Programming. In *Proc. of the International Workshop on Logic Programming Environments (WLPE'01)*, pp. 1–14, Paphos (Cyprus), 2001. Also available from the Computing Research Repository (CoRR) at `http://arXiv.org/abs/cs.PL/0111039`.
55. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, Vol. 9, No. 1, pp. 33–75, 1999.
56. 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.
57. 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.
58. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, 2006.
59. J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pp. 17–42. Addison Wesley, 1990.
60. H. Hussmann. Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. *Journal of Logic Programming*, Vol. 12, pp. 237–255, 1992.
61. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111–119, Munich, 1987.
62. 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.
63. J. Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, No. 3, pp. 1–49, 1999.
64. 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.
65. R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. *Theoretical Computer Science 142*, pp. 59–87, 1995.
66. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
67. 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.
68. 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.
69. 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.

70. 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.
71. 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.
72. M.J. O'Donnell. *Computing in Systems Described by Equations*. Springer LNCS 58, 1977.
73. M.J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.
74. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.
75. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
76. J.C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*, pp. 717–740. ACM Press, 1972.
77. 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.
78. V.A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
79. C. Schulte and G. Smolka. Encapsulated Search for Higher-Order Concurrent Constraint Programming. In *Proc. of the 1994 International Logic Programming Symposium*, pp. 505–520. MIT Press, 1994.
80. 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.
81. 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.
82. 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.
83. 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.
84. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
85. P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 5, pp. 804–851, 1997.
86. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.
87. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.