In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs* (PLILP'97), Southampton, UK. Springer LNCS 1292, 1997, pp. 335–350

Teaching Functional and Logic Programming with a Single Computation Model

Michael Hanus

Informatik II, RWTH Aachen, D-52056 Aachen, Germany
hanus@informatik.rwth-aachen.de

Abstract. Functional and logic programming are often taught in different courses so that students often do not understand the relationships between these declarative programming paradigms. This is mainly due to the different underlying computation models—deterministic reduction and lazy evaluation in functional languages, and non-deterministic search in logic languages. We show in this paper that this need not be the case. Taking into account recent developments in the integration of functional and logic programming, it is possible to teach the ideas of modern functional languages like Haskell and logic programming on the basis of a single computation model. From this point of view, logic programming is considered as an extension of functional programming where ground expressions are extended to contain also free variables. We describe this computation model, the structure of a course based on it, and draw some conclusions from the experiences with such a course.

Keywords: Functional logic languages, lazy evaluation, narrowing, residuation, integration of paradigms

1 Introduction

Declarative programming is motivated by the fact that a higher programming level using powerful abstraction facilities leads to reliable and maintainable software. Thus, declarative programming languages are based on mathematical formalisms and completely abstract from many details of the concrete hardware and the implementation of the programs on this hardware. Since declarative programs strongly correspond to formulae of mathematical calculi, they simplify the reasoning (e.g., verification w.r.t. non-executable specifications), provide freedom in the implementation (e.g., use of parallel architectures), and reduce the program development time in comparison to classical imperative languages.

Unfortunately, declarative programming is currently split into two main fields based on different formalisms, namely functional programming (lambda calculus) and logic programming (predicate logic). As a consequence, functional and logic programming are usually taught in different courses so that students often do not understand the relationships between these declarative programming paradigms. This is mainly due to the different underlying computation models—deterministic reduction and lazy evaluation in functional languages, and non-deterministic search in logic languages. On the other hand, functional and logic languages have a common kernel and can be seen as different facets of a single idea. For instance, the use of algebraic data types instead of pointer structures, and the definition of local comprehensible cases by pattern matching and local definitions instead of complex procedures are emphasized in functional as well as logic programming. However, these commonalities are often hidden by the differences in the computation models and the application areas of these languages. The advantages of logic languages are in areas where computing with partial information and non-deterministic search is important (data bases, knowledge-based systems, optimization problems), where functional languages emphasize the improved abstraction and structuring facilities using referential transparency, higher-order functions and lazy evaluation. If students attend one course on functional programming and another course on logic programming, they know programming in the single languages, but it is often not clear for them that these are different facets of a common idea. As a result, they may not know how to choose the best of both worlds in order to solve a particular application problem.

In this paper we show how to overcome this problem. Our approach is the choice of a single language, an integrated functional logic language, for teaching functional and logic programming. We show that it is possible to teach the ideas of modern functional languages like Haskell and logic programming on the basis of a single language, provided that recent developments in the integration of functional and logic programming are taken into account. For this purpose we use the multi-paradigm language Curry [9, 11] which amalgamates functional, logic, and concurrent computation techniques. Since Curry subsumes many aspects of modern lazy functional languages like Haskell, it can be used to teach functional programming techniques without any reference to logic programming. Logic programming can be introduced at a later point as an extension of functional programming where ground expressions are extended to contain also free variables. The requirement to compute values for such free variables leads naturally to the introduction of non-determinism and search techniques. Curry's operational semantics is based on a single computation model, described in [9], which combines lazy reduction of expressions with a possibly non-deterministic binding of free variables occurring in expressions. Thus, pure functional programming, pure logic programming, and concurrent (logic) programming are obtained as particular restrictions of this model. Moreover, due to the use of an integrated functional logic language, we can choose the best of the two worlds in application programs. For instance, input/output (implemented in logic languages by side effects) can be handled with the monadic I/O concept [18] in a declarative way. Similarly, many other impure features of Prolog (e.g., arithmetic, cut) can be avoided by the use of functions.

In the next section, we introduce some basic notions and motivate the basic computation model of Curry. Some features of Curry are discussed in Section 3. Section 4 describes a course on declarative programming based on Curry's computational model. Section 5 contains our conclusions.

2 A Unified Model for Declarative Programming

This section introduces some basic notions of term rewriting [7] and functional logic programming [8] and recalls the computation model of Curry. In contrast to [9], we provide a more didactically oriented motivation of this computation model.

As mentioned above, a common idea of functional as well as logic programming is the use of algebraic data types instead of pointer structures (e.g., list terms instead of linked lists). Thus, the computational domain of declarative languages is a set of *terms* constructed from constants and data constructors. *Functions* (or *predicates* in logic programming, but we consider predicates as Boolean functions for the sake of simplicity) operate on terms and map terms to terms.

Formally, we consider a *signature* partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*.[1] We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for n-ary constructor and function symbols, respectively. A constructor c with arity 0 is also called a *constant*.[2]

We denote by $\mathcal{X}$ a set of variables (with elements x, y). An *expression* (*data term*) is a *variable* $x \in \mathcal{X}$ or an application $\varphi(e_1, \ldots, e_n)$ where $\varphi/n \in \mathcal{C} \cup \mathcal{F}$ ($\varphi/n \in \mathcal{C}$) and $e_1, \ldots, e_n$ are expressions (data terms).[3] We denote by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$ the set of all expressions and data terms, respectively. $\mathcal{V}ar(e)$ denotes the set of variables occurring in an expression e. An expression e is called *ground* if $\mathcal{V}ar(e) = \varnothing$. A *pattern* is an expression of the form $f(t_1, \ldots, t_n)$ where each variable occurs only once, $f/n \in \mathcal{F}$, and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A *head normal form* is a variable or an expression of the form $c(e_1, \ldots, e_n)$ with $c/n \in \mathcal{C}$.

A *position* p is a sequence of positive integers identifying a subexpression. $e|_p$ denotes the *subterm* or *subexpression* of e at position p, and $e[e']_p$ denotes the result of *replacing the subterm* $e|_p$ by the expression e' (see [7] for details).

A *substitution* is a mapping $\mathcal{X} \to \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$, where *id* denotes the identity substitution. Substitutions are extended to morphisms on expressions by $\sigma(\varphi(e_1, \ldots, e_n)) = \varphi(\sigma(e_1), \ldots, \sigma(e_n))$ for every expression $\varphi(e_1, \ldots, e_n)$. A substitution σ is called a *unifier* of two expressions e_1 and e_2 if $\sigma(e_1) = \sigma(e_2)$.

A (*declarative*) *program* $\mathcal{P}$ is a set of *rules* $l = r$ where l is a pattern and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. l and r are called left-hand side and right-hand side, respectively.[4] A rule is called a *variant* of another rule if it is obtained by a unique replacement of variables by other variables. In order to ensure well-definedness of functions, we require that $\mathcal{P}$ contains only trivial overlaps, i.e., if l_1=r_1 and l_2=r_2 are variants of rewrite rules and σ is a unifier for l_1 and l_2, then $\sigma(r_1) = \sigma(r_2)$ (*weak orthogonality*).

Example 1. If natural numbers are data terms built from the constructors `0` and `s`, we define the addition and the predicate "less than or equal to" as follows:

```
   0 + y = y                         0 ⩽ x     = true
s(x) + y = s(x+y)                 s(x) ⩽ 0     = false
                                  s(x) ⩽ s(y) = x⩽y
```

[1] We omit the types of the constructors and functions in this section since they are not relevant for the computation model. Note, however, that Curry is typed language with a Hindley/Milner-like polymorphic type system (see Section 3).

[2] Note that elementary built-in types like truth values (`true`, `false`), integers, or characters can also be considered as sets with (infinitely) many constants.

[3] We do not consider partial applications in this part since it is not relevant for the computation model. Such higher-order features are discussed in Section 3.

[4] For the sake of simplicity, we consider only unconditional rewrite rules in the main part of this paper. An extension to conditional rules is described in Section 3.

Since the left-hand sides are pairwise non-overlapping, the functions are well defined. $\square$

From a functional point of view, we are interested in computing *values* of expressions, where a value does not contain function symbols (i.e., it is a data term) and should be equivalent (w.r.t. the program rules) to the initial expression. The computation of a value can be done by applying rules from left to right. For instance, we can compute the value of `s(s(0))+s(0)` by applying the rules for addition to this expression:

$$\texttt{s(s(0))+s(0)} \to \texttt{s(s(0)+s(0))} \to \texttt{s(s(0+s(0)))} \to \texttt{s(s(s(0)))}$$

Formally, a *reduction step* is an application of a rule l=r to the subterm (*redex*) $t|_p$, i.e., $t \to s$ if $s = t[\sigma(r)]_p$ for some substitution σ with $\sigma(l) = t|_p$ (i.e., the left-hand side l of the selected rule must *match* the subterm $t|_p$).

In contrast to imperative languages, where the algorithmic control is explicitly contained in the programs by the use of various control structures, declarative languages abstract from the control issue since a program consists of rules and does not contain explicit information about the order to apply the rules. This makes the reasoning about declarative programs easier (program analysis, transformation, or verification) and provides more freedom for the implementor (e.g., transforming call-by-need into call-by-value, implementation on parallel architectures). On the other hand, a concrete programming language must provide a precise model of computation to the programmer. Thus, we can distinguish between different classes of functional languages. In an *eager functional language*, the selected redex in a reduction step is always an innermost redex, i.e., the redex is a ground pattern, where in *lazy functional languages* the selected redex is an outermost one. Innermost reduction may not compute a value of an expression in the presence of nonterminating rules, i.e., innermost reduction is not normalizing (we call a reduction strategy *normalizing* iff it always computes a value of an expression if it exists). Thus, we consider in the following outermost reduction, since it allows the computation with infinite data structures and provides more modularity by separating control aspects [13].

A subtle point in the definition of a lazy evaluation strategy is the selection of the "right" outermost redex. For instance, consider the rules of Example 1 together with the rule `f = f`. Then the expression `0+0`$\leqslant$`f` has two outermost redexes, namely `0+0` and `f`. If we select the first one, we compute the value `true` after one further outermost reduction step. However, if we select the redex `f`, we run into an infinite reduction sequence instead of computing the value. Thus, it is important to know which outermost redex is selected. Most lazy functional languages choose the leftmost outermost redex which is implemented by translating pattern matching into case expressions [23]. On the other hand, this may not be the best possible choice since leftmost outermost reduction is in general not normalizing (e.g., take the last example but swap the arguments of $\leqslant$ in the rules and the initial expression). It is well known that we can obtain a normalizing reduction strategy by reducing in each step a needed redex [12]. Although the computation of a needed redex is undecidable in general, there are relevant subclasses of programs where needed redexes can be effectively computed. For instance, if functions are inductively defined on the structure of data terms (so-called *inductively sequential functions* [3]), a needed redex can be simply computed by pattern matching. This is the basis of our computation model.

For this purpose, we organize all rules of a function in a hierarchical structure called definitional tree [3].[5] $\mathcal{T}$ is a *definitional tree with pattern* π iff the depth of $\mathcal{T}$ is finite and one of the following cases holds:

$\mathcal{T} = rule(l = r)$, where $l = r$ is a variant of a program rule such that $l = \pi$.
$\mathcal{T} = branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where p is a position of a variable in π, $c_1, \ldots, c_k$ are different constructors ($k > 0$), and, for all $i = 1, \ldots, k$, $\mathcal{T}_i$ is a definitional tree with pattern $\pi[c_i(x_1, \ldots, x_n)]_p$, where n is the arity of c_i and $x_1, \ldots, x_n$ are new variables.

A *definitional tree of an n-ary function f* is a definitional tree $\mathcal{T}$ with pattern $f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are distinct variables, such that for each rule $l = r$ with $l = f(t_1, \ldots, t_n)$ there is a node $rule(l' = r')$ in $\mathcal{T}$ with l variant of l'. In the following, we write $pat(\mathcal{T})$ for the pattern of a definitional tree $\mathcal{T}$, and DT for the set of all definitional trees. A function is called *inductively sequential* iff there exists a definitional tree for it. A program is inductively sequential if all defined functions are inductively sequential.

For instance, the predicate $\leqslant$ defined in Example 1 is inductively sequential, and a definitional tree for $\leqslant$ is (we underline the case variable in the pattern of each *branch* node):

$$\begin{array}{l} branch(\underline{\mathtt{x1}} \leqslant \mathtt{x2}, 1, rule(\mathtt{0} \leqslant \mathtt{x2}\ \mathtt{=}\ \mathtt{true}), \\ \qquad\qquad branch(\mathtt{s(x)} \leqslant \underline{\mathtt{x2}}, 2, rule(\mathtt{s(x)} \leqslant \mathtt{0}\ \ \ \ \ \ \mathtt{=}\ \mathtt{false}), \\ \qquad\qquad\qquad\qquad rule(\mathtt{s(x)} \leqslant \mathtt{s(y)}\ \mathtt{=}\ \mathtt{x} \leqslant \mathtt{y}\ ))) \end{array}$$

Intuitively, a definitional tree of a function specifies the strategy to evaluate a call to this function. If the tree is a *rule* node, we apply the rule. If it is a *branch* node, it is necessary to evaluate the subterm at the specified position (the so-called *needed* subterm) to head normal form in order to commit to one of the branches. Thus, in order to evaluate the expression `0+0`$\leqslant$`f` w.r.t. the previous definitional tree, the top branch node requires that the first subterm `0+0` must be evaluated to head normal form (in this case: `0`) in order to commit to the first branch.

Such a reduction strategy has the following advantages:

1. The strategy is normalizing, i.e., it always computes a value if it exists [3].
2. The strategy is independent on the order of rules. Note that pattern matching in traditional lazy functional languages implemented by case expressions is independent on the order of rules only for *uniform* programs [23] which is a strict subclass of inductively sequential programs.[6]

[5] We could also introduce our strategy by compiling all rules of a function into a case expression [23]. However, the use of definitional trees has the advantage that the structure of rules is not destroyed and the trees can be easily extended to more general classes of programs which become relevant later.

[6] *Uniform functions* are those functions where a definitional tree with a strict left-to-right order in the positions of the branches exists.

3. The definitional trees can be automatically generated from the left-hand sides of the rules [9] (similarly to the compilation of pattern matching into case expressions), i.e., there is no need for the programmer to explicitly specify the trees.
4. There is a strong equivalence between reduction with definitional trees and reduction with case expressions since definitional trees can be simply translated into case expressions [10]. However, reduction with definitional trees can be easily extended to more general strategies, as can be seen in the following.

Inductively sequential functions have the property that there is a single argument in the left-hand sides which distinguishes the different rules. In particular, functions defined by rules with overlapping left-hand sides, like the "parallel-or"

```
  true ∨ x     = true
     x ∨ true  = true
 false ∨ false = false
```

are not inductively sequential. However, it is fairly easy to extend definitional trees to cover also such functions. For this purpose, we introduce a further kind of nodes: a definitional tree $\mathcal{T}$ with pattern π can also have the form $or(\mathcal{T}_1, \mathcal{T}_2)$ where $\mathcal{T}_1$ and $\mathcal{T}_2$ are definitional trees with pattern π.[7] It is easy to see that a definitional tree with *or* nodes can be constructed for each defined function (see [9] for a concrete algorithm). For instance, a definitional tree for the parallel-or is

$$
\begin{aligned}
&or(branch(\underline{\texttt{x1}}\vee\texttt{x2}, 1, rule(\texttt{true}\vee\texttt{x2 = true}),\\
&\qquad\qquad\qquad\qquad branch(\texttt{false}\vee\underline{\texttt{x2}}, 2, rule(\texttt{false}\vee\texttt{false = false}))),\\
&\quad branch(\texttt{x1}\vee\underline{\texttt{x2}}, 2, rule(\texttt{x1}\vee\texttt{true = true})))
\end{aligned}
$$

The corresponding extension of the reduction strategy is a more subtle point. One possibility is the parallel reduction of the redexes determined by the different *or* branches of the tree. This is a deterministic and normalizing reduction strategy [3, 19] but requires some effort in the implementation. Since our computation model must include some kind of non-determinism in order to cover logic programming languages, we take a simpler strategy which non-deterministically computes all alternatives for *or* nodes, i.e., the evaluation strategy maps expressions into sets of expressions (see Appendix A for the precise formal definition). The solution taken in most lazy functional languages, i.e., the processing of overlapping rules in sequential order, is not considered since it destroys the equational reading of rules (i.e., modularity is lost since the rules cannot be understood independently).

Up to now, we have only considered functional computations where ground expressions are reduced to some value. In logic languages, the initial expression (usually an expression of Boolean type, called a *goal*) may contain free variables. A logic programming system should find values for these variables such that the goal is reducible to `true`. Fortunately, it requires only a slight extension of the reduction strategy introduced so far to cover non-ground expressions and variable instantiation (which also shows that the difference between functional and logic programming is not so large). Remember that there exists no reduction step if

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

a needed subterm is a free variable. Since the value of this variable is needed in order to proceed the computation, we non-deterministically bind the variable to the constructor required in the subtrees. For instance, if the function `f` is defined by the rules

```
f(a) = c
f(b) = d
```

(where `a`, `b`, `c`, `d` are constants), then the expression `f(x)` with the free variable `x` is evaluated to `c` or `d` if `x` is bound to `a` or `b`, respectively.

Unfortunately, one of the most important aspects, namely the instantiation of free variables, is not explicitly shown by the computation of values. Thus, we have to change our computational domain. Due to the presence of free variables in expressions, an expression may be reduced to different values by binding the free variables to different terms. In functional programming, one is interested in the computed *value*, whereas logic programming has the interest in the different bindings (*answers*). Thus, we define for our integrated framework an *answer expression* as a pair $\sigma \,[\!]\, e$ consisting of a substitution σ (the answer computed so far) and an expression e. An answer expression $\sigma \,[\!]\, e$ is *solved* if e is a data term. We sometimes omit the identity substitution in answer expressions, i.e., we write e instead of $id \,[\!]\, e$ if it is clear from the context.

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

For instance, if we consider the previous example, the evaluation of `f(x)` together with the different bindings for `x` is reflected by the following non-deterministic computation step:

$$\texttt{f(x)} \;\rightarrow\; \{\{\texttt{x} \mapsto \texttt{a}\} \,[\!]\, \texttt{c}\;,\; \{\texttt{x} \mapsto \texttt{b}\} \,[\!]\, \texttt{d}\}$$

For the sake of readability, we write the latter disjunctive expression in the form `{x=a}c | {x=b}d`. Similarly, the expression `f(b)` is reduced to `d` (which abbreviates a disjunctive expression with one element and the identity substitution).

A single *computation step* performs a reduction in exactly one expression of a disjunction (e.g., in the leftmost unsolved expression). This expression is reduced (with a possible variable instantiation) according to our strategy described so far. If the program is inductively sequential, i.e., the definitional trees do not contain *or* nodes, then this strategy is equivalent to the *needed narrowing* strategy [4]. Needed narrowing enjoys several optimality properties: every reduction step is needed, i.e., necessary to compute the final result, it computes the shorted possible derivations (if common subterms are shared) and a minimal set of solutions, and it is fully deterministic on ground expressions, i.e., in the functional programming case. If some definitional trees contain *or* nodes, optimality is lost (however, it is still optimal on the inductively sequential parts of the program), but the resulting strategy is sound and complete in the sense of functional and logic programming, i.e., all values and answers are computed [5].

The strategy described so far covers functional logic languages with a sound and complete operational semantics (i.e., based on narrowing [8]). However, it is still too restrictive for a general course on modern declarative languages due to the following reasons:

1. Narrowing and guessing of free variables should not be applied to all functions, since some functions (defined on recursive data structures) may not terminate if particular arguments are unknown.
2. The computation model requires the explicit definition of all functions by program rules. It is not clear how to connect primitive (external, predefined) functions where the rules are not explicitly given, like arithmetic, I/O etc.
3. Modern logic languages provide flexible selection rules (concurrent computations based on the synchronization on free variables).

All these features can be easily supported by allowing the delay of function calls if a particular argument is not instantiated. Thus, we allow the suspension of a function call if the value of some needed argument is unknown. For this purpose we extend the definition of *branch* nodes by an additional flag, i.e, a *branch* node has the form $branch(\pi, p, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)$ with $r \in \{rigid, flex\}$. A $flex$ annotation is treated as before, but a *rigid* annotation specifies that the evaluation of the function call is delayed if the branch argument is a free variable.

Since function calls may suspend, we need a mechanism to specify concurrent computations. For this purpose, we introduce a final extension of definitional trees: a definitional tree $\mathcal{T}$ with pattern π can also have the form $and(\mathcal{T}_1, \mathcal{T}_2)$ where the definitional trees $\mathcal{T}_1$ and $\mathcal{T}_2$ have the same pattern π and contain the same set of rules. An *and* node specifies the necessity to evaluate more than one argument position. The corresponding operational behavior is to try to evaluate one of these arguments. If this is not possible since the function calls in this argument are delayed, we proceed by trying to evaluate the other argument. This generalizes concurrent computation models for residuating logic programs [1, 21] to functional logic programs. For instance, the *concurrent conjunction* $\wedge$ of constraints[8] is defined by the single rule

$$\texttt{valid} \wedge \texttt{valid = valid} \qquad (R_\wedge)$$

together with the definitional tree

$$\begin{aligned}&and(branch(\underline{\texttt{x1}}\wedge\texttt{x2}, 1, rigid, branch(\texttt{valid}\wedge\underline{\texttt{x2}}, 2, rigid, rule(R_\wedge))),\\ &\quad branch(\texttt{x1}\wedge\underline{\texttt{x2}}, 2, rigid, branch(\underline{\texttt{x1}}\wedge\texttt{valid}, 1, rigid, rule(R_\wedge))))\end{aligned}$$

Due to the *and* node in this tree, an expression of the form $e_1 \wedge e_2$ is evaluated by an attempt to evaluate e_1. If the evaluation of e_1 suspends, an evaluation step is applied to e_2. If a variable responsible to the suspension of e_1 was bound during the last step, the left expression will be evaluated in the subsequent step. Thus, we obtain a concurrent behavior with an interleaving semantics.

The complete specification of this computation model is summarized in Appendix A.

[8] The auxiliary constructor `valid` denotes the result value of a solved constraint. In terms of our computation model, the equational constraint (cf. Section 3) `s(x)=s(s(0))` is reduced to the answer expression $\{$`x=s(0)`$\}$`valid`.

3 Curry: A Multi-Paradigm Declarative Language

Curry [9, 11] is a multi-paradigm declarative language aiming to integrate functional, logic, and concurrent programming paradigms. Curry's operational semantics is based on the computation model motivated and explained in the previous section. The operational behavior of each function is specified by its definitional tree which is automatically generated from the left-hand sides of the rewrite rules using a left-to-right pattern matching algorithm [9]. Non-Boolean functions are annotated with *rigid* branches, and predicates are annotated with *flex* branches (there are compiler pragmas to override these defaults; moreover, definitional trees can also be explicitly provided similarly to type annotations). This has the consequence that the operational behavior is nearly identical to lazy functional languages if the logic programming features are not used, and identical to logic programming if only predicates are defined. Thus, Curry is ideal to teach the concepts of functional and logic programming languages in a single course.

Beyond this computation model, Curry provides a parametrically polymorphic type system (the current implementation has a type inference algorithm for a Hindley/Milner-like type system; the extension to Haskell-like type classes is planned for a future version), modules, monadic I/O [18] etc. Basic arithmetic is provided by considering integer values, like "`42`" or "`-10`", as constants, and the usual operations on integers as primitive functions with *rigid* arguments, i.e., they are delayed until all arguments are known constants. For instance, the expression `3+5` is reduced to `8`, whereas `x+y` is delayed until `x` and `y` are bound by some other part of the program. Thus, they can act as passive constraints [2] providing for better constraint solvers than in pure logic programming [22] (e.g., by transforming "generate-and-test" into "test-and-generate"). Conceptually, primitive functions can be considered as defined by an infinite set of rules which provides a declarative reading for such functions [6]. In a similar way, any other external (side-effect free!) function can be connected to Curry.

Since functional logic languages are often used to solve equations between expressions containing defined functions, the notion of *equality* needs particular attention. It is well known (e.g., [16]) that the mathematical notion of equality (i.e., least congruence relation containing all instances of program rules) is not reasonable in the presence of nonterminating functions. In this context, the only sensible notion of equality, which is also used in functional languages, is *strict equality*, i.e., an *equational constraint* e_1=e_2 is satisfied if both sides e_1 and e_2 are reducible to a same data term. Operationally, an equational constraint e_1=e_2 is solved by evaluating e_1 and e_2 to unifiable data terms. Thus, an equation e_1=e_2 without occurrences of defined functions has the same meaning (unification) as in Prolog. The basic kernel of Curry only provides strict equations e_1=e_2 between expressions as constraints. Since it is conceptually fairly easy to add other constraint structures, future extensions of Curry will provide richer constraint systems to support constraint logic programming applications.

Conditional rules, in particular with *extra variables* (i.e., variables not occurring in the left-hand side) in conditions, are one of the essential features to provide the full power of logic programming. Although the basic computation model only supports unconditional rules, it can be easily extended to conditional rules following the approach taken in Babel [16]: consider a conditional rule[9] "l | $\{c\}$ = r" (where the constraint c is the condition) as syntactic sugar for the rule $l = (c \Rightarrow r)$, where the right-hand side is a *guarded expression*. The operational meaning of a guarded expression "$c \Rightarrow r$" is defined by the predefined rule

```
(valid ⇒ x)  =  x .
```

Thus, a guarded expression is evaluated by solving its guard. If this is successful, the guarded expression is replaced by the right-hand side r of the conditional rule.

Higher-order functions have been shown to be very useful to structure programs and write reusable software [13]. Although the basic computation model includes only first-order functions, higher-order features can be implemented by providing a (first-order) definition of the application function (as shown by Warren [24] for logic programming). Curry supports the higher-order features of current functional languages (partial function applications, lambda abstractions) by this technique, where the rules for the application function are implicitly defined. In particular, function application is *rigid* in the first argument, i.e., an application is delayed until the function to be applied is known (this avoids the expensive and operationally complex synthesis of functions by higher-order unification).

Further features of Curry include a committed choice construct, the encapsulation of search to get more control over the non-deterministic evaluation and to encapsulate non-deterministic computations between monadic I/O actions, and an interface to other constraint solvers. Since these features are not yet implemented, they have not been used in the course on declarative programming.

4 A Course on Declarative Programming

We have shown in Section 2 that the computation model of Curry comprises the models of functional as well as logic programming. Actually, one can obtain the computation models of a number of different declarative programming languages by particular restrictions on definitional trees (see [9] for details). Thus, from our point of view, it is the best available choice to teach functional and logic programming in a seamless way. In the following, we present the structure of a sample course based on this model. This course is not intended for beginners (since we do not discuss imperative or object-oriented programming techniques) but ideal for 2nd year or more advanced students.

"How to start?" is surely an important question for every course. To teach functional and logic programming concepts from the beginning is problematic from a didactic point of view. Thus, one has the alternative to start with functional programming and extend it with logic programming concepts, or vice versa. We take the first alternative since it may be easier to understand deterministic programming concepts than non-deterministic ones. Therefore, we start with functional programming and consider later logic programming as an extension of functional programming where ground expressions are extended to contain free variables and

[9] Constraints are enclosed in curly brackets. A Haskell-like guarded rule "l | b = r", where b is a Boolean expression, is considered as syntactic sugar for "l | {b=`true`} = r".

conditional rules may also contain extra variables. Thus, the course has the following structure:

Introduction: General ideas of declarative programming; referential transparency and substitution principle from mathematics (replacing equals by equals); problems of low-level control structures, side effects, and memory management in imperative languages.

Functional programming: Throughout this section, we compute only values, i.e., we do not use free variables in initial expressions or extra variables in conditions. For this kind of programs, free variables do not occur in computed expressions. The substitutions in the answer expressions are always the identity and, therefore, they are omitted. Moreover, function calls will never be delayed. Thus, the computational behavior of Curry is identical to a lazy functional language like Haskell (if function definitions are uniform, see below).

Basics: Function definitions; computing by reduction; strict vs. non-strict languages; if-then-else vs. conditional rules vs. pattern matching.

Here we use only elementary built-in data types like integers or Booleans, and Curry has a clear advantage over traditional narrowing-based functional logic languages, since Curry supports these built-in data types in a straightforward way (cf. Section 3).

Data types: Basic types; list and tuple types; function types; user-defined algebraic data types.

Since Curry has a Hindley-Milner-like type system and higher-order functions, all these topics can be treated as in modern functional languages.

Pattern matching: Patterns; influence of patterns to evaluation; compilation of pattern matching into case expressions; uniform definitions.

Linearity of patterns is a natural requirement. Therefore, students learn to program with linear patterns and put possible equalities in the condition part so that the left-linearity requirement of rules does not cause any difficulties also in the following logic programming part (note that left-linearity is a restriction from a logic programming point of view).

The translation of uniform definitions (cf. Footnote 6) into case expressions is straightforward. The most important problems in this section are non-uniform rules with overlapping left-hand sides. Since they are treated sequentially in traditional pattern matching compilers [23], the produced code is awkward. Moreover, the students partially got the impression that the declarative reading of programs is destroyed by this kind of sequential pattern matching. A parallel outermost reduction strategy (which could be provided by definitional trees but is not yet implemented) seems to be a better choice. Generally, one should better use if-then-else or negated conditions for programs based on sequential pattern matching and leave the optimizations (for instance, avoid repeated evaluations of identical conditions) to the compiler.

Higher-order functions: Generic functions; currying; control structures.

Since Curry provides the usual higher-order features from functional programming (cf. Section 3), this section can be taught like in traditional functional programming courses.

Type systems and type inference: Parametric polymorphism; type schemes; type inference.

Since Curry has currently only a Hindley/Milner-like type system, type classes and overloading are not discussed.

Lazy evaluation: Lazy reduction; head normal form; strictness; infinite data structures; stream-oriented programming.

As shown in Section 2, Curry's computation model is identical to lazy reduction if free variables do not occur. Thus, everything is identical to traditional functional programming.

Foundations: Reduction systems; lambda calculus; term rewriting.

This section formally defines important notions like confluence, normalization, critical pairs, orthogonality, or reduction strategies. Definitional trees (without *or* and *and* nodes) are introduced to define an efficient normalizing reduction strategy for inductively sequential programs. This is also the basis to explain the instantiation of variables in the next section.

Computing with partial information: Logic programming: Since Curry supports the (non-deterministic) computation with free variables, it is not necessary to switch to another language with apparently different concepts like Prolog. If free variables are bound, the substitutions in the answer expressions are no longer identity substitutions and, therefore, they are also shown to the user. Thus, it becomes clear in a natural way that in logic programming also the answer part is relevant.

Motivation: We start with a traditional deductive data base example (family relationships) to show the advantages of free variables in initial expressions. Since the same language is used, it is also clear that the concepts are similar. Due to the necessary instantiation of free variables, it becomes obvious that non-deterministic computations must be performed. After this elementary example, the advantages of extra variables in conditions and partial data structures (stepwise accumulation of information) are shown.

Computation with free variables: The idea of narrowing, soundness and completeness of narrowing, and the problems of (strict) equality are discussed. The important notion of unification can be reused from the section on type inference.

Logic programming: Logic programming is introduced as a special case of this general framework where only relations are allowed and all definitions have a flat structure (conditions in rules are conjunction of predicates). Resolution is defined as a special case of narrowing. Backtracking and logic programming techniques (e.g., generate-and-test) are introduced. All this can be done within the language Curry. The linearity of patterns is not a serious problem since non-linear patterns can be translated into linear ones by adding equations between variables in the condition part.

Narrowing strategies: The improvement of simple narrowing by introducing strategies is discussed. This leads to the needed narrowing strategy [4] which is optimal on inductively sequential programs and the basis of Curry's operational semantics. The problem of overlapping rules is discussed by introducing *or* nodes in definitional trees together with some optimizations for narrowing with *or* nodes (dynamic cut [15], parallel narrowing [5]). Now it becomes clear why a sequential implementation of overlapping rules cannot be taken since otherwise the binding of variables caused by the application of rules becomes unsound.

Residuation and concurrent programming: As motivated at the end of Section 2, the non-deterministic evaluation of partially instantiated function calls by narrowing should be avoided in some cases. Thus, the possible delayed evaluation of function calls by the residuation principle [1] is introduced and provides the opportunity to explain the treatment of primitive functions (like arithmetic, see Section 3). Moreover, concurrent programming techniques can be explained, e.g., concurrent objects can be modeled as processes with a stream of messages as input [20]. Since a function can delay if the input stream is not sufficiently instantiated, multiple agents can be simply implemented.

Curry's programming model: The computation model of Curry (cf. Section 2) is presented. A survey on other computation models for pure (lazy) functional, pure (concurrent) logic, narrowing- and residuation-based functional logic languages is given as restrictions of Curry's computation model. Different notions of equality (equality as constraint solving vs. equality as a test), the computation with higher-order functions and the evaluation of conditional rules are discussed.

Extensions: Since we introduced functional and logic programming with a single language, we can choose the best features of both paradigms for extensions which are necessary for application programs. For instance, *input/output* can be handled using the monadic I/O approach of functional languages [18], whereas optimization problems can be treated in the logic programming paradigm by adding particular constraint structures [14] (note that Curry's computation model can be easily extended to more general constraint structures by replacing the substitution part in answer expressions by constraints).

5 Conclusions

Functional and logic programming are often considered as separate programming paradigms and taught in different courses so that the common idea of declarative programming is sometimes lost. We have shown in this paper that this need not be the case if a single declarative language based on a unified computation model is taken into account. From this point of view, the difference between functional and logic programming is the difference between computation with full and partial information which also shows up in a difference in the (non-)determinism of programs. Most of the other ideas, like algebraic data structures, pattern matching, or local definitions, are similar in both paradigms. Thus, we developed a single course on declarative programming based on the computation model of the multi-paradigm language Curry. We taught such a course to advanced students in the 3rd/4th year which was very similar to that described in Section 4 (the minor differences were due to the fact that the full implementation of Curry was not yet available). Our experience with this course, also in comparison to previous courses on functional or logic programming, is quite motivating since we could concentrate on the best concepts and avoid other problematic topics. For instance, the description of the "cut" operator of Prolog could be completely avoided, since the pruning of the search space can be obtained by using functions instead of predicates or an explicit use of "if-then-else". Moreover, an integrated functional logic language leads to a natural amalgamation of programming techniques, e.g., conditions in function rules could be solved by a non-deterministic search in the presence of extra variables, or higher-order programming techniques can be more often applied in logic programming by partial applications of predicates to arguments [17].

Of course, we cannot address all topics discussed in traditional courses on functional or logic programming. For instance, meta-programming techniques, which are useful to implement new or modify existing computation models, are not yet considered. Another important topic in logic programming is the treatment of negative information by the "negation-as-failure" rule. Some basic applications of this rule (e.g., to check the disequality of data terms) can be treated in Curry by strict equality. However, more advanced applications require a better treatment of negative information. Such extensions are topics for future work.

References

1. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and Functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 17–23, 1987.
2. H. Aït-Kaci and A. Podelski. Functions as Passive Constraints in LIFE. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 4, pp. 1279–1318, 1994.
3. S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.
4. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, Portland, 1994.
5. S. Antoy, R. Echahed, and M. Hanus. Parallel Evaluation Strategies for Functional Logic Languages. In *Proc. of the Fourteenth International Conference on Logic Programming (ICLP'97)*. MIT Press (to appear), 1997.
6. 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.
7. 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.
8. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
9. 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.
10. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. In *Proc. Seventh International Conference on Rewriting Techniques and Applications (RTA'96)*, pp. 138–152. Springer LNCS 1103, 1996.
11. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1997.

12. G. Huet and J.-J. Lévy. Computations in Orthogonal Rewriting Systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pp. 395–443. MIT Press, 1991.
13. J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topcis in Functional Programming*, pp. 17–42. Addison Wesley, 1990.
14. J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, Vol. 19&20, pp. 503–581, 1994.
15. R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. *Theoretical Computer Science 142*, pp. 59–87, 1995.
16. J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.
17. L. Naish. Higher-order logic programming in Prolog. In *Proc. JICSLP'96 Workshop on Multi-Paradigm Logic Programming*, pp. 167–176. TU Berlin, Technical Report No. 96-28, 1996.
18. S.L. Peyton Jones and P. Wadler. Imperative Functional Programming. In *Proc. 20th Symposium on Principles of Programming Languages (POPL'93)*, pp. 71–84, 1993.
19. 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.
20. E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 2, pp. 251–273. MIT Press, 1987.
21. 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.
22. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.
23. P. Wadler. Efficient Compilation of Pattern-Matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pp. 78–103. Prentice Hall, 1987.
24. D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.

A Operational Semantics of Curry

The operational semantics of Curry is specified using the functions

$$\begin{array}{lll} cse & : \mathcal{T}(\mathcal{C}\cup\mathcal{F},\mathcal{X}) & \to \mathcal{D}\cup\{\bot\} \\ cs & : \mathcal{T}(\mathcal{C}\cup\mathcal{F},\mathcal{X})\times DT & \to \mathcal{D}\cup\{\bot\} \end{array}$$

(remember that $\mathcal{D}$ denotes the set of all disjunctions of answer expressions). The function *cse* performs a single computation step on an expression e. It computes a disjunction of answer expressions or the special constant $\bot$ indicating that no computation step is possible in e. As shown in Figure 1, *cse* attempts to apply a reduction step to the leftmost outermost function symbol in e by the use of *cs*, which is called with the appropriate subterm and a definitional tree with fresh variables for the leftmost outermost function symbol. *cs* is defined by a case distinction on the definitional tree. If it is a *rule* node, we apply this rule. If the definitional tree is an *and* node, we try to evaluate the first branch and, if this is not possible due to the suspension of all function calls, the second branch is

Computation step for a single (unsolved) expression:

$$\begin{array}{lll} cse(x) & = \bot & \text{for all variables } x \\ cse(f(e_1,\ldots,e_n)) & = cs(f(e_1,\ldots,e_n),\mathcal{T}) & \text{if } \mathcal{T} \text{ is a fresh definitional tree for } f \end{array}$$

$$cse(c(e_1,\ldots,e_n)) = \begin{cases} replace(c(e_1,\ldots,e_n),k,cse(e_k)) & \text{if } cse(e_1)=\cdots=cse(e_{k-1})=\bot\neq cse(e_k) \\ \bot & \text{if } cse(e_i)=\bot,\ i=1,\ldots,n \end{cases}$$

Computation step for an operation-rooted expression e:

$$cs(e,rule(l\,\texttt{=}\,r)) = \{id\,[\!]\,\sigma(r)\} \qquad \text{if } \sigma \text{ is a substitution with } \sigma(l)=e$$

$$cs(e,and(\mathcal{T}_1,\mathcal{T}_2)) = \begin{cases} cs(e,\mathcal{T}_1) & \text{if } cs(e,\mathcal{T}_1)\neq\bot \\ cs(e,\mathcal{T}_2) & \text{otherwise} \end{cases}$$

$$cs(e,or(\mathcal{T}_1,\mathcal{T}_2)) = \begin{cases} cs(e,\mathcal{T}_1)\cup cs(e,\mathcal{T}_2) & \text{if } cs(e,\mathcal{T}_1)\neq\bot\neq cs(e,\mathcal{T}_2) \\ \bot & \text{otherwise} \end{cases}$$

$$cs(e,branch(\pi,p,r,\mathcal{T}_1,\ldots,\mathcal{T}_k)) = \begin{cases} cs(e,\mathcal{T}_i) & \text{if } e|_p=c(e_1,\ldots,e_n) \text{ and } pat(\mathcal{T}_i)|_p=c(x_1,\ldots,x_n) \\ \varnothing & \text{if } e|_p=c(\cdots) \text{ and } pat(\mathcal{T}_i)|_p\neq c(\cdots), i=1,\ldots,k \\ \bot & \text{if } e|_p=x \text{ and } r=rigid \\ \bigcup_{i=1}^{k}\{\sigma_i\,[\!]\,\sigma_i(e)\} & \text{if } e|_p=x,\ r=flex, \text{ and } \sigma_i=\{x\mapsto pat(\mathcal{T}_i)|_p\} \\ replace(e,p,cse(e|_p)) & \text{if } e|_p=f(e_1,\ldots,e_n) \end{cases}$$

Derivation step for a disjunctive expression:

$$\{\sigma\,[\!]\,e\}\cup D \quad\to\quad \{\sigma_1\circ\sigma\,[\!]\,e_1,\ldots,\sigma_n\circ\sigma\,[\!]\,e_n\}\cup D$$

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

Fig. 1. Operational semantics of Curry

tried (this simple sequential strategy for concurrent computations could also be replaced by a more sophisticated strategy with a fair selection of threads). An *or* node produces a disjunction. The most interesting case is a *branch* node. Here we have to branch on the value of the top-level symbol at the selected position. If the symbol is a constructor, we proceed with the appropriate definitional subtree, if possible. If it is a function symbol, we proceed by evaluating this subexpression. If it is a variable, we either suspend (if the branch is *rigid*) or instantiate the variable to the different constructors. The auxiliary function *replace* puts a possibly disjunctive expression into a subterm:

$$replace(e,p,d) = \begin{cases} \{\sigma_1\,[\!]\,\sigma_1(e)[e_1]_p,\ldots,\sigma_n\,[\!]\,\sigma_n(e)[e_n]_p\} & \text{if } d=\{\sigma_1\,[\!]\,e_1,\ldots,\sigma_n\,[\!]\,e_n\} \\ \bot & \text{if } d=\bot \end{cases}$$

The overall computation strategy transforms disjunctive expressions. It takes a disjunct $\sigma\,[\!]\,e$ not in solved form and computes $cse(e)$. If $cse(e)=\bot$, then the computation of this expression *flounders* (i.e., this expression is not solvable). If $cse(e)$ is a disjunctive expression, we substitute it for $\sigma\,[\!]\,e$ composed with the old answer substitution.