In *Proc. of the 1997 Joint Conference on Declarative Programming (APPIA-GULP-PRODE'97)*, pp. 9–24, Grado (Italy), 1997

A Unified Computation Model for Declarative Programming

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

Abstract

We propose a new computation model which combines the operational principles of functional languages (reduction), logic languages (non-deterministic search for solutions), and integrated functional logic languages (residuation and narrowing). This computation model combines efficient evaluation principles of functional languages with the problem-solving capabilities of logic programming. Since the model allows the delay of insufficiently instantiated function calls, it also supports a concurrent style of programming. We show that many known evaluation principles of declarative languages are particular instances of our model. This computation model is the basis of Curry, a multi-paradigm language which combines functional, logic and concurrent programming styles. We conclude with a description of some features of Curry.

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. For instance, pointers are avoided and replaced by the use of algebraic data types, and complex procedures are split into easily comprehensible parts using pattern matching and local definitions. 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 mathematical formalisms, namely functional programming (lambda calculus) and logic programming (predicate logic). This has negative consequences w.r.t. to teaching (usually, there are different courses on functional programming and logic programming, and students do not see many similarities between them), research (each field has its own community, conferences, and journals, and sometimes similar solutions are developed twice), and applications (each field has its own application areas and some effort has been done to show that one paradigm can cover applications of the other paradigm [37] instead of showing the advantages of declarative programming in various application fields). The separation is mainly due to the different underlying computations 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 pointers, 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.

In this paper we want to show how to overcome this problem. Our approach is the choice of a single computation model which combines lazy reduction of expressions with a possibly non-deterministic binding of free variables occurring in expressions. Since it is a conservative extension of an optimal evaluation strategy for integrated functional logic languages [5], it combines the problem-solving capabilities of logic programming with optimal reduction strategies known from functional programming for a large class of programs. Moreover, in order to avoid uncontrolled non-determinism during the evaluation of particular expressions and to provide a simple connection to externally defined functions, function calls may be suspended until the arguments are sufficiently instantiated. 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 [31] in a fully declarative way. Similarly, most of the other impure features of Prolog (e.g., arithmetic, cut) can be avoided by the use of functions.

This computation model is the basis of the multi-paradigm language Curry [16, 17, 19]. Apart from this new model, Curry offers many other features useful for practical programming, like a type and a module system, higher-order functions, arithmetic, declarative I/O etc. The development of Curry is an international initiative intended to provide a common basis for functional logic languages and further research and developments in this area. More details can be found in the Curry home page: `http://www-i2.informatik.rwth-aachen.de/~hanus/curry/`

In the next section, we introduce some basic notions and motivate the basic computation model of Curry. Properties of this model are briefly discussed in Section 3. Section 4 outlines some features of Curry, and Section 5 contains our conclusions.

2 A Unified Computation Model for Declarative Programming

In this section we introduce the basic computation model of Curry, where we use a slightly different description than in its original presentation [16]. We motivate it by a stepwise extension of a rewrite model to features from logic and concurrent programming. The complete formal specification of our computation model is summarized in Appendix A.

2.1 Term Rewriting

Firstly, we introduce some basic notions of term rewriting [11] and functional logic programming [15].

As mentioned in the previous section, a common idea of functional as well as logic programming is the use of algebraic data types instead pointers. 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 throughout this paper 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] Usually, there are at least the 0-ary Boolean constructors `true` and `false`.

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 $\mathcal{V}ar(e) = \emptyset$. 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 in an expression. $e|_p$ 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 [11] 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*). However, it is also possible to drop this restriction and allow non-deterministic functions since such functions can be evaluated by a non-deterministic rewrite principle [13], which is part of this computation model (cf. Section 2.3).

Example 2.1 If natural numbers are data terms built from the constructors `0` and `s`, the following rules define the addition and the predicate "less than or equal to" for natural numbers:

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

Since the left-hand sides are pairwise non-overlapping, the functions are well defined. □

[1] For the sake of simplicity, 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 4).

[2] Note that elementary built-in types like truth values, 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 4.

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

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 value can be computed 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:

`s(s(0))+s(0)` $\rightarrow$ `s(s(0)+s(0))` $\rightarrow$ `s(s(0+s(0)))` $\rightarrow$ `s(s(s(0)))`

Formally, a *reduction step* is an application of a rule l=r to the subterm (*redex*) $t|_p$, i.e., $t \rightarrow s$ 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 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 [22].

2.2 Lazy Evaluation and Pattern Matching

A subtle point in the definition of a lazy evaluation strategy in combination with pattern matching is the selection of the "right" outermost redex. For instance, consider the rules of Example 2.1 together with the rule `f = f`. Then the expression `0+0`$\leq$`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 [38]. 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 $\leq$). It is well known that we can obtain a normalizing reduction strategy by reducing in each step a needed redex [21]. 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* [4]), 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 [4].[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:

$\mathcal{T} = rule(l\,\texttt{=}\,r)$, where $l\,\texttt{=}\,r$ 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 $l\,\texttt{=}\,r$ with $l = f(t_1, \ldots, t_n)$ there is a node $rule(l'\,\texttt{=}\,r')$ 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 $\leq$ defined in Example 2.1 is inductively sequential, and a definitional tree for $\leq$ is:

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

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 to head normal form in order to commit to one of the branches. Thus, in order to evaluate the expression `0+0`$\leq$`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.

Formally, if e is an expression with a function f at the top[6] and $\mathcal{T}$ is a definitional tree for f, then $e \to e'$ is a reduction step iff $e' = cs(e, \mathcal{T})$, where the partial function cs ("*c*omputation *s*tep") is defined as follows:

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

$$cs(e, branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_k))$$
$$= \begin{cases} cs(e, \mathcal{T}_i) & \text{if } e|_p = c(e_1, \ldots, e_n) \text{ and } pat(\mathcal{T}_i)|_p = c(x_1, \ldots, x_n) \\ e[e']_p & \text{if } e|_p = f'(\cdots),\ \mathcal{T}' \text{ is a definitional tree of } f', \text{ and } cs(e|_p, \mathcal{T}') = e' \end{cases}$$

This definition of a reduction strategy has the following advantages:

1. The strategy is normalizing, i.e., it always computes a value if it exists.

2. The strategy is independent on the order of rules. Note that pattern matching in traditional lazy functional languages implemented by case expressions [38] is independent on the order of rules only for *uniform* programs [38] which is a strict subclass of inductively sequential programs.[7]

[5] We could also introduce our strategy by compiling all rules of a function into a case expression [38]. 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] If the expression has a constructor at the top, consider the leftmost outermost subexpression which has a function at the top.

[7] *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 [16] (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 easily translated into case expressions (see [18] for details). However, reduction with definitional trees can be easily extended to more general strategies, as can be seen in the following.

2.3 Overlapping Rules and Non-deterministic Rewriting

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"

$$\begin{array}{rcl} \texttt{true} \vee \texttt{x} & = & \texttt{true} \\ \texttt{x} \vee \texttt{true} & = & \texttt{true} \\ \texttt{false} \vee \texttt{false} & = & \texttt{false} \end{array}$$

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.[8] It is easy to see that a definitional tree with *or* nodes can be constructed for each defined function (see [16] for a concrete algorithm). For instance, a definitional tree for the parallel-or is

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

The corresponding extension of the reduction strategy is a more subtle point. The following extension of *cs* processes the branches of the *or* nodes in a sequential manner:

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

This corresponds to the implementation of overlapping rules in most lazy functional languages, i.e., rules in such languages cannot be read as equalities between the left- and right-hand side but must be read as sequences where the latter rules are discarded if a rule can be successfully applied. This has the advantage that some negative conditions in subsequent rules can be avoided, but it leads to a more operational than declarative reading of programs (i.e., some kind of modularity is lost since the rules cannot be understood independently). A further disadvantage is that no value is computed if the computation with the first branch does not terminate and only the second branch leads to the result. To overcome this problem, we could replace the sequential implementation by a non-deterministic one, i.e., we assume that *cs* maps expressions into sets of expressions ($cs : \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X}) \times DT \to 2^{\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})}$) and define

$$cs(e, or(\mathcal{T}_1, \mathcal{T}_2)) = cs(e, \mathcal{T}_1) \cup cs(e, \mathcal{T}_2)$$

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

By reducing all expressions in parallel, it is ensured that a value will eventually be computed if it exists. Another alternative is the parallel reduction of independent subexpressions which is a deterministic and normalizing reduction strategy [34]. This can also be defined by a modification of *cs* so that a set of redex positions is computed by the use of definitional trees and all these redexes are reduced in parallel (see [4, 6] for more details). Since our computation model must include some kind of non-determinism in order to cover logic programming languages, we take the first alternative and assume in the following that *cs* maps expressions into sets of expressions.

2.4 Computing with Non-ground Expressions

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 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 from an operational point of view). The current definition of *cs* is undefined if we have to branch on 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. Thus, we could extend the definition of *cs* by the following case:[9]

$$cs(e, branch(\pi, p, \mathcal{T}_1, \ldots, \mathcal{T}_k)) = \bigcup_{i=1}^{k} cs(\sigma_i(e), \mathcal{T}_i) \quad \text{if } e|_p = x \text{ and } \sigma_i = \{x \mapsto pat(\mathcal{T}_i)|_p\}$$

For instance, if the function `f` is defined by the rules

```
f(a) = a
f(b) = b
```

(where `a` and `b` are constants), then the expression `f(x)` with the free variable `x` is evaluated by *cs* as follows:

$$\texttt{f(x)} \rightarrow \{\texttt{a},\texttt{b}\}$$

Unfortunately, one of the most important aspects, namely the instantiation of free variables, is not explicitly shown in this computation step. 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

[9] In order to ensure completeness, we also have to ensure that the definitional tree taken to evaluate a function symbol has always fresh variables.

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:

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

For the sake of readability, we write the latter disjunctive expression in the form `{x=a}a | {x=b}b`. Similarly, the expression `f(b)` is reduced to `b` (which is an abbreviation for 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 [5]. 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 [6].

2.5 Equality and Constraints

Functional logic languages are able to solve equations containing defined functions. For instance, consider the function `+` defined in Example 2.1 and the equation `x+0=s(0)`. Using the computation model presented so far, this equation can be solved by evaluating the left-hand side `x+0` to the answer expression `{x=s(0)}s(0)` (here we omit the other alternatives in the disjunction). Since the resulting equation is trivial, the equation is valid w.r.t. the computed answer `{x=s(0)}`.

Thus, we could solve an equation by reducing both sides to unifiable terms. However, it is well known [12, 27] that this notion of equality is not reasonable in the presence of nonterminating functions. The only sensible notion of equality which is also used in functional languages, is the *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. As a consequence, if both sides are undefined (nonterminating), then the strict equality does not hold. Operationally, an equational constraint e_1=e_2 is solved by evaluating e_1 and e_2 to unifiable data terms. The equational constraint could also be solved in an incremental way by an interleaved lazy evaluation of the expressions and binding of variables to constructor terms [25].

Equational constraints are different from standard Boolean functions since they are checked for satisfiability. For instance, the equational constraint `x=s(0)` is satisfiable if the variable `x` is bound to `s(0)`. However, the evaluation of `x=s(0)` does not deliver a Boolean value `true` or `false`, since the latter value would require a binding of `x` to all values different from `s(0)` (which could be expressed if we use a richer constraint system than substitutions, for instance, disequality constraints [7]). This is sufficient since, similarly to logic programming, constraints are only used in conditions of equations (cf. Section 4) which must be checked for satisfiability.

If we want to check the equality of two fully known expressions, we can reduce both sides to ground constructor terms and check their identity. This *test equality* can be specified as any other Boolean function by the following rules (where `==` and `&&` are infix operators):

$$\begin{array}{ll}
\texttt{c == c = true} & \forall \texttt{c}/0 \in \mathcal{C} \\
\texttt{c(x}_1\texttt{,...,x}_n\texttt{) == c(y}_1\texttt{,...,y}_n\texttt{) = x}_1\texttt{==y}_1\texttt{ \&\&...\&\& x}_n\texttt{==y}_n & \forall \texttt{c}/n \in \mathcal{C} \\
\texttt{c(x}_1\texttt{,...,x}_n\texttt{) == d(y}_1\texttt{,...,y}_m\texttt{) = false} & \forall \texttt{c}/n, \texttt{d}/m \in \mathcal{C} \text{ with } \texttt{c}/n \neq \texttt{d}/m \\
\texttt{true \&\& x = x} & \\
\texttt{false \&\& x = false} &
\end{array}$$

For instance, the test "`s(0)==s(0)`" reduces to `true`, whereas the test "`s(0)==0`" reduces to `false`. In order to avoid an infinite set of solutions for insufficiently instantiated tests like `x==y`, the evaluation of a test equality is suspended if one side is a free variable (i.e., `==` is *rigid* in both arguments, cf. Section 2.6). Therefore, the test equality can be used where Boolean values are required (e.g., in the condition part of `if-then-else`), whereas equational constraints can only be applied in the condition of a program rule. In terms of concurrent constraint programming languages [33], `==` and `=` correspond to *ask* and *tell* equality constraints, respectively. This is also justified by the fact that a test e_1==e_2 is suspended if one side is a variable, whereas an equational constraint e_1=e_2 is checked for satisfiability and propagates new variable bindings.

Note that 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 [26], future extensions of Curry will provide richer constraint systems to support constraint logic programming applications.

2.6 Concurrent Computations

The strategy described so far covers functional logic languages with a sound and complete operational semantics (i.e., based on narrowing [15]). However, it is still too restrictive to cover all important aspects of 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. For this purpose we extend the function *cs* so that the evaluation of some function call may suspend, i.e., *cs* has the type

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

A function call may be suspended if the value of some (needed) argument is unknown. Thus, 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 *rigid* annotation specifies that the evaluation of the function call is suspended if the branch argument is a free variable. This is expressed by the following new definition of cs for the case of free variables:

$$cs(e, branch(\pi, p, r, \mathcal{T}_1, \ldots, \mathcal{T}_k)) \\ = \begin{cases} \cdots \\ suspend & \text{if } e|_p = x \text{ and } r = rigid \\ \bigcup_{i=1}^{k} \{\sigma_i \,[\!]\, \sigma_i(e)\} & \text{if } e|_p = x,\ r = flex, \text{ and } \sigma_i = \{x \mapsto pat(\mathcal{T}_i)|_p\} \end{cases}$$

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, 2, 35] to functional logic programs. For instance, the *concurrent conjunction* of constraints $\wedge$ is defined by the single rule[10]

$\texttt{valid} \wedge \texttt{valid = valid}$

together with the definitional tree

$$and(branch(\texttt{x1}\wedge\texttt{x2}, 1, rigid, branch(\texttt{valid}\wedge\texttt{x2}, 2, rigid, rule(\texttt{valid}\wedge\texttt{valid = valid}))), \\ branch(\texttt{x1}\wedge\texttt{x2}, 2, rigid, branch(\texttt{x1}\wedge\texttt{valid}, 1, rigid, rule(\texttt{valid}\wedge\texttt{valid = valid}))))$$

Due to the *and* node in this tree, a constraint of the form $t_1 \wedge t_2$ is evaluated by an attempt to evaluate t_1. If the evaluation of t_1 suspends, an evaluation step is applied to t_2. If a variable responsible to the suspension of t_1 was bound during the last step, the left expression will be evaluated in the subsequent step. Thus, we obtain a concurrent behavior with an interleaving semantics.

This fairly simple model for concurrent computations is able to cover applications of Prolog systems with coroutining [29]. For instance, if `gen` is a predicate or constraint which instantiates its arguments with potential solutions (i.e., `gen` is defined with flexible branch nodes) and `test` checks whether the argument is a correct solution (i.e., `test` is defined with rigid branch nodes), then a constraint like "`gen(X)` $\wedge$ `test(X)`" corresponds to a "generate-and-test" solution whereas "`test(X)` $\wedge$ `gen(X)`" specifies a "test-and-generate" solution where the test is activated as soon as its argument is sufficiently instantiated.

It is also interesting to note that this model is able to cover recent developments in parallel functional computation models like Eden [9] or Goffin [10]. For instance, a constraint of the form "`x=f(t1)` $\wedge$ `y=g(t2)` $\wedge$ `z=h(x,y)`" specifies a potentially concurrent computation of the functions `f`, `g` and `h` where the function `h` can proceed its computation only if the arguments have been bound by evaluating the expressions `f(t1)` and `g(t2)` (provided that `h` is rigid in all arguments).

The advantage of this computation model is the clear separation between sequential and concurrent parts. Sequential computations, which could be considered as the basic units of a program, could be expressed as usual functional (logic) programs, and they can be composed to concurrent computation units via concurrent conjunctions of

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

constraints. Since constraints could be passed as arguments or results of functions (like any other data object or function), it is possible to specify general operators to create flexible communication architectures similarly to Goffin [10]. Thus, the same abstraction facilities could be used for sequential as well as concurrent programming. On the other hand, the clear separation between sequential and concurrent computations supports the use of efficient and optimal evaluation strategies for the sequential parts, where similar techniques for the concurrent parts are not available. This is in contrast to other, more fine-grained concurrent computation models like AKL [23], CCP [33], or Oz [35].

3 Properties of the Computation Model

Detailed soundness and completeness results for the operational model presented in the previous section can be found in [16]. Due to the possible suspension of function calls, we cannot expect strong completeness results as in logic programming. However, it can be shown that all computed answers are correct and no answer is lost during the computation. Moreover, if all definitional trees have flexible branches, then a completeness result similar to logic programming holds.

This computation model subsumes various known evaluation principles for declarative programming languages, which can be seen by particular restrictions of the form of definitional trees (see [16] for a more detailed discussion). For instance, if the definitional trees only contains $rule$ and flexible $branch$ nodes, we obtain the optimal *needed narrowing* strategy [5]. This shows that the computation model is a conservative extension of an optimal evaluation strategy for functional logic programs. The addition of *or* nodes supports function definitions with overlapping left-hand sides and results in the *weakly needed narrowing* strategy [6, 25], which is a widely used strategy in current narrowing-based lazy functional logic languages. *Simple lazy narrowing* [27, 32] or SLD-resolution can be obtained by connecting all trees for each rule by *or* nodes. The lazy evaluation strategy of functional languages like Haskell [20] performs pattern matching from left to right [38] and, therefore, it can be implemented by definitional trees with "left-to-right"-oriented branch nodes. The extension of this functional kernel with equational constraints leads to recent computation models for parallel functional languages [9, 10]. Finally, the effect of residuation [1, 2] is obtained by marking all branches of predicates as flexible, and all branches of non-Boolean functions as rigid.

Figure 1 summarizes the necessary restrictions on the form of definitional trees in order to obtain a particular strategy.

4 Curry: A Multi-Paradigm Declarative Language

Curry [17, 19] 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 Section 2. The operational behavior of each function is specified by its definitional tree. Since it it tedious to specify the definitional trees for all functions, they are automatically generated from the left-hand sides of the rewrite rules using a left-to-right pattern matching algorithm [16]. Non-Boolean functions are annotated with $rigid$ branches, and predicates (i.e., functions with Boolean result type) are annotated with $flex$ branches (there are compiler pragmas to override these defaults; moreover, definitional trees can also be explicitly provided

Strategy	Restrictions on definitional trees
Needed narrowing [5]	only *rule* and flexible *branch* nodes; optimal strategy w.r.t. length of derivations and number of computed solutions
Weakly needed narrowing [6, 25]	only *rule*, flexible *branch*, and *or* nodes
Simple lazy narrowing [27, 32] and SLD-resolution	particular definitional trees with flexible *branch* nodes (a *branch*/*rule* tree for each left-hand side, all rules connected by *or* nodes)
Lazy functional languages [38]	definitional trees with left-to-right pattern matching; initial expression has no free variable
Residuation [1, 2, 24, 35]	rigid branches for non-Boolean functions; flexible branches for predicates
Parallel functional languages [9, 10]	definitional trees with left-to-right pattern matching; parallelism via equational constraints

Figure 1: Specification of different operational models by definitional trees

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.

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 [39] is planned for a future version), a module system, and a declarative concept for input/output operations based on the monadic I/O concept from functional programming [31].

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 [3] providing for better constraint solvers than in pure logic programming [36] (e.g., by transforming "generate-and-test" into "test-and-generate", cf. Section 2.6). Conceptually, primitive functions can be considered as defined by an infinite set of rules which provides a declarative reading for such functions [8]. In a similar way, any other external (side-effect free!) function can be connected to Curry.

Higher-order functions has been shown to be very useful to structure programs and write reusable software [22]. Although the basic computation model includes only first-order functions, Warren [40] has shown that the higher-order features of functional programming can be implemented by providing a (first-order) definition of the application function. 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 [28]).

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 [27]: consider a conditional rule[11] "l | $\{c\}$ = r" (where the condition c is a constraint) 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 an attempt to solve the condition. If this is successful, the guarded expression is replaced by the right-hand side r of the conditional rule.

Further features of Curry, which are under development, include a committed choice construct, the encapsulation of search to get more control over the non-deterministic evaluation, and an interface to other constraint solvers.

5 Conclusions

Functional and logic programming are often considered as separate programming paradigms and 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 programming 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, lazy evaluation, or local definitions, are similar in both paradigms. Additionally, some problematic "non-logical" features of Prolog can be avoided in the integrated language. For instance, I/O operations with side effects can be replaced by monadic I/O operations, and the use of the "cut" operator of Prolog could be avoided, since the pruning of the search space can be obtained by using functions instead of predicates [14] 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 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 [30].

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] 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.

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

[11] Constraints are enclosed in curly brackets. Thus, a Haskell-like guarded rule "l | b = r", where b is a Boolean expression, can be considered as syntactic sugar for the conditional rule "l | $\{b$=true$\}$ = r".

[5] 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.

[6] 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.

[7] P. Arenas-Sánchez, A. Gil-Luezas, and F.J. López-Fraguas. Combining Lazy Narrowing with Disequality Constraints. In *Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pp. 385–399. Springer LNCS 844, 1994.

[8] 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.

[9] S. Breitinger, R. Loogen, and Y. Ortega-Mallen. Concurrency in Functional and Logic Programming. In *Fuji International Workshop on Functional and Logic Programming.* World Scientific Publ., 1995.

[10] M.M.T. Chakravarty, Y. Guo, M. Köhler, and H.C.R. Lock. Goffin - Higher-Order Functions Meet Concurrent Constraints. *Science of Computer Programming (to appear)*, 1997.

[11] 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.

[12] 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.

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

[14] M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 1–23. Springer LNCS 631, 1992.

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

[16] 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.

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

[18] 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.

[19] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1997.

[20] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell (Version 1.2). *SIGPLAN Notices*, Vol. 27, No. 5, 1992.

[21] 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.

[22] J. Hughes. Why Functional Programming Matters. In D.A. Turner, editor, *Research Topcis in Functional Programming*, pp. 17–42. Addison Wesley, 1990.

[23] S. Janson and S. Haridi. An Introduction to AKL: A Multi-Paradigm Programming Language. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Constraint Programming*, NATO ASI Series, pp. 414–449. Springer, 1994.

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

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

[26] F.J. López Fraguas. A General Scheme for Constraint Functional Logic Programming. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 213–227. Springer LNCS 632, 1992.

[27] 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.

[28] G. Nadathur and D. Miller. An Overview of λProlog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 810–827. MIT Press, 1988.

[29] L. Naish. *Negation and Control in Prolog.* Springer LNCS 238, 1987.

[30] 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.

[31] 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.

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

[33] V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[34] 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.

[35] 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.

[36] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[37] P. Wadler. How to Replace Failure by a List of Successes. In *Functional Programming and Computer Architecture.* Springer LNCS 201, 1985.

[38] 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.

[39] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL'89*, pp. 60–76, 1989.

[40] 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

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

The function *cse* performs a single computation step on an expression e. It computes a disjunction of answer expressions or the special constant *suspend* indicating that no reduction is possible in e. As shown in Figure 2, *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 the definitional tree 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

Computation step for a single (unsolved) expression:

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

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

Computation step for an operation-rooted expression e:

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

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

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

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

Derivation step for a disjunctive expression:

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

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

Figure 2: Operational semantics of Curry

suspension of all function calls, the second branch.[12] An *or* node produces a disjunction. To ensure completeness, we have to suspend the entire disjunction if one disjunct suspends [16]. For a similar reason, we cannot commit to a disjunct which does not bind variables but we have to consider both alternatives (see [6] for a counter-example). The most interesting case is a *branch* node. Here we have to branch on the value of the top-level symbol at the selected position. If the symbol is a constructor, we proceed with the appropriate definitional subtree, if possible. If it is a function symbol, we proceed by evaluating this subterm. If it is a variable, we either suspend (if the branch is *rigid*) or instantiate the variable to the different constructors. The auxiliary function *replace* puts a possibly disjunctive expression into a subterm:

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

The overall computation strategy is a transformation on disjunctive expressions. It takes a disjunct $\sigma \,[\!]\, e$ not in solved form and computes $cse(e)$. If $cse(e) = suspend$, then the computation of this expression *flounders* and we cannot proceed (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.

[12] For the sake of simplicity, we choose a simple sequential strategy for concurrent computations. However, it is also possible to provide a more sophisticated strategy with a fair selection of threads, e.g., as in Oz [35].