In *Proc. 11th International Workshop on Functional and (Constraint) Logic Programming, WFLP 2002* (M. Falaschi, Ed.), Grado (Italy), pp. 7–20, 2002

An Operational Semantics for Declarative Multi-Paradigm Languages*

E. Albert[1] M. Hanus[2] F. Huch[2] J. Oliver[1] G. Vidal[1]

[1] DSIC, UPV, Camino de Vera s/n, E-46022 Valencia, Spain
{ealbert,fjoliver,gvidal}@dsic.upv.es
[2] Institut für Informatik, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany
{mh,fhu}@informatik.uni-kiel.de

Abstract. In this paper we define an operational semantics for functional logic languages covering notions like laziness, sharing, concurrency, non-determinism, etc. Such a semantics is not only important to provide appropriate language definitions to reason about programs and check the correctness of implementations but it is also a basis to develop language-specific tools, like program tracers, profilers, optimizers, etc. First, we define a "big-step" semantics in natural style to relate expressions and their evaluated results. Since this semantics is not sufficient to cover concurrency, search strategies, or to reason about costs associated to particular computations, we also define a "small-step" operational semantics covering the features of modern functional logic languages.

1 Introduction

This paper is motivated by the fact that there does not exist a precise definition of an operational semantics covering all aspects of modern declarative multi-paradigm programming languages, like laziness, sharing, concurrency, logical variables, non-determinism, etc. For instance, the report on the multi-paradigm language Curry [13] contains a fairly precise operational semantics but covers sharing only informally. The operational semantics of the functional logic language Toy [18] is based on narrowing and sharing (without concurrency) but the formal definition is based on a narrowing calculus [10] which does not include a particular pattern-matching strategy. However, the latter becomes important if one wants to reason about costs of computations (see [4] for a discussion about narrowing strategies and calculi). [14] contains an operational semantics for a lazy narrowing strategy but it addresses neither concurrency nor aspects of search strategies.

In order to provide an appropriate basis for reasoning about programs, correctness of implementations, optimizations, or costs of computations, we define

* This work has been partially supported by CICYT TIC 2001-2705-C03-01, by Acción Integrada Hispano-Alemana HA2001-0059, by Acción Integrada Hispano-Austriaca HU2001-0019, by Acción Integrada Hispano-Italiana HI2000-0161, and by the DFG under grant Ha 2457/1-2.

an operational semantics covering the important aspects of current functional logic languages. For this purpose, we proceed in two steps. First, we introduce a *natural* semantics which defines the intended results by relating expressions to values. This "big-step" semantics is non-deterministic and does not cover all aspects (e.g., concurrency, search strategies). However, it accurately models sharing which is important not only to reason about the space behavior of programs (as in [17]) but also for the correctness of computed results in the presence of non-confluent function definitions [10]. Then, we provide a more implementation-oriented semantics based on the definition of individual computation steps. After stating the equivalence of the non-deterministic version of this "small-step" semantics with the natural semantics, we extend it to cover concurrency and search strategies. This final semantics is the formal basis to reason about operational aspects of programs, for instance, in order to develop appropriate debugging tools. It is a basis to provide a comprehensive definition of Curry (in contrast to [11, 13] which contain only partial definitions). One can use it to prove the correctness of implementations by further refinements, as done in [22]. Furthermore, one can count the costs (time/space) associated to particular computations in order to justify optimizations [1, 3, 23] or to compare different search strategies.

2 Foundations

A main motivation for this work is to provide a foundation for developing programming tools (like profilers, debuggers, optimizers) for declarative multi-paradigm languages. In order to be concrete, we consider Curry [11, 13] as our source language. Curry is a modern multi-paradigm language amalgamating in a seamless way the most important features from functional, logic, and concurrent programming. Its operational semantics is based on an execution model that combines lazy evaluation with non-determinism and concurrency. This model has been introduced in [11] without formalizing the sharing of common subterms.

Basically, a Curry program is a set of function definitions (and data definitions for the sake of typing, which we ignore here). Each function is defined by rules describing different cases for input arguments. For instance, the conjunction on Boolean values (`True`, `False`) can be defined by the rules

```
and True  x  =  x
and False x  =  False
```

(data constructors usually start with uppercase and function application is denoted by juxtaposition). Since Curry supports logic programming features, there are no limitations w.r.t. overlapping rules. In particular, one can also have non-confluent rules to define functions that yield more than one result for a given input (these are called *non-deterministic* or *set-valued functions*). For instance, the following function non-deterministically returns one of its arguments:

```
choose x y  =  x
choose x y  =  y
```

A subtle question is the meaning of such definitions if function calls are passed as parameters, e.g., the set of possible values of "`double (choose 1 2)`" w.r.t. the definition "`double x = x+x`." Similarly to [10], Curry follows the "call-time choice" semantics where all descendants of a subterm are reduced to the same value in a derivation, i.e., the previous expression reduces non-deterministically to one of the values `2` or `4` (but not to `3`). This choice is consistent with a lazy evaluation strategy where all descendants of a subterm are shared [17]. It is the purpose of this paper to describe the combination of laziness, sharing, and non-determinism in a precise and understandable manner.

We consider programs where functions are defined by *rules* of the form "$f\,t_1 \ldots t_n$ `=` e" where f is a function, $t_1, \ldots, t_n$ are data terms (i.e., without occurrences of defined functions), the *left-hand side* $f\,t_1 \ldots t_n$ is linear (i.e., without multiple occurrences of variables), and e is a well-formed *expression*. A rule is applicable if its left-hand side matches the current call. Functions are evaluated lazily so that the operational semantics of Curry is a conservative extension of lazy functional programming. It extends the optimal evaluation strategy of [5] by concurrent programming features. These are supported by a concurrent conjunction operator "`&`" on constraints (i.e., expressions of the built-in type `Success`). In particular, a constraint of the form "c_1 `&` c_2" is evaluated by solving both constraints c_1 and c_2 concurrently.

In order to provide an understandable operational description, we assume that programs are translated into a "flat" form, which is a convenient standard representation for functional logic programs. The flat form makes the pattern matching strategy explicit by the use of case expressions, which is important for the operational reading; moreover, source programs can be easily translated into this flat form [12]. The syntax for programs in flat form is as follows:

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

where P denotes a program, D a function definition, p a pattern and e an arbitrary expression. A program P consists of a sequence of function definitions D such that the left-hand side has pairwise different variable arguments. The right-hand side is an expression e composed by variables (e.g., x, y, z,...), data constructors (e.g., a, b, c,...), function calls (e.g., f, g,...), case expressions, disjunctions (e.g., to represent set-valued functions), and let bindings where the local variables $x_1, \ldots, x_n$ are only visible in $e_1, \ldots, e_n, e$. A case expression has the form[1] $(f)\mathit{case}\ e\ \mathit{of}\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$, where e is an expression, $c_1, \ldots, c_k$ are different constructors, and $e_1, \ldots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are locally introduced and bind the corresponding variables of

[1] We write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$ and $(f)\mathit{case}$ for either fcase or case.

the subexpression e_i. The difference between *case* and *fcase* shows up when the argument e is a free variable: *case* suspends whereas *fcase* nondeterministically binds this variable to the pattern in a branch of the case expression and proceeds with the appropriate branch. Let bindings are in principle not required for translating Curry programs but they are convenient to express sharing without the use of complex graph structures. Operationally, let bindings introduce new structures in memory that are updated after evaluation, which is essential for lazy computations. As an example, we show the translation of the functions `and` and `choose` into the flat form:

$$\begin{array}{lcl} \texttt{and}(\texttt{x},\texttt{y}) & = & \textit{case}\ \texttt{x}\ \textit{of}\ \{\texttt{True} \to \texttt{y};\ \ \texttt{False} \to \texttt{False}\ \} \\ \texttt{choose}(\texttt{x},\texttt{y}) & = & \texttt{x}\ \textit{or}\ \texttt{y} \end{array}$$

Laziness (or neededness) of computations will show up in the description of the behavior of function calls and case expressions. In a function call, parameters are not evaluated but directly passed to the function's body. In a case expression, the form of the outermost symbol of the case argument is required; therefore, the case argument should be evaluated to a *head normal form* (i.e., a variable or an expression with a constructor at the top). Consequently, our operational semantics will describe the evaluation of expressions only to head normal form. This is not a restriction since the evaluation to normal form or the solving of equations can be reduced to head normal form computations (see [12]). Similarly, the higher-order features of current functional languages can be reduced to first-order definitions by introducing an auxiliary "apply" function [24]. Therefore, we base the definition of our operational semantics on the flat form described above. This is also consistent with current implementations which use the same intermediate language [6].

Extra variables are those variables in a rule which do not occur in the left-hand side. Such extra variables are intended to be instantiated by constraints in conditions or right-hand sides. They are usually introduced in Curry programs by a declaration of the form `let x free in...` As Antoy [4] pointed out, the use of extra variables in a functional logic language causes no conceptual problem if these extra variables are renamed whenever a rule is applied. We will model this renaming similar to the renaming of local variables in let bindings. For this purpose, we assume that all extra variables x are explicitly introduced in flat programs by a (circular) let binding of the form $let\ x = x\ in\ e$. Throughout this paper, we call such variables which are bound to themselves *logical variables*. For instance, an expression $x + y$ with logical variables x and y is represented as $let\ x = x, y = y\ in\ x + y$. It is interesting to note that circular bindings are also used in implementations of Prolog to represent logic variables [25].

3 A Natural Semantics for Functional Logic Programs

In this section, we introduce a natural (big-step) semantics for functional logic programs which is in the midway between a (simple) denotational semantics and a (complex) operational semantics for a concrete abstract machine. Our

semantics is non-deterministic and accurately models sharing. Let us illustrate the effect of sharing by means of an example.

Example 1. Consider the following (flat) program:

$$\begin{array}{lcl} \texttt{foo(x)} & = & \texttt{addB(x,x)} \\ \texttt{bit} & = & 0 \ or \ 1 \\ \texttt{addB(x,y)} & = & case\ \texttt{x}\ of\ \{0 \to 0;\ 1 \to case\ \texttt{y}\ of\ \{0 \to 1;\ 1 \to \texttt{BO}\}\} \end{array}$$

Under a sharing-based implementation, the computation of "`foo(`e`)`" must evaluate the expression e only once. Therefore, the evaluation of the goal `foo(bit)` must return either `0` or `BO` (binary overflow). Note that, without sharing, the results would be `0`, `1`, or `BO`.

The definition of our semantics mainly follows the natural semantics defined in [17] for the lazy evaluation of functional programs. In this (higher-order) functional semantics, the *let* construct is used for the creation and sharing of *closures* (i.e., functional objects created as the value of lambda expressions). The key idea in Launchbury's natural semantics is to describe the semantics in two parts: a "normalization" process—which consists in converting the λ-calculus into a form where the creation and sharing of closures is made explicit—followed by the definition of a simple semantics at the level of closures. Similarly, we also describe our (first-order) semantics for functional logic programs in two separated phases. In the first phase, we apply a normalization process in order to ensure that the arguments of functions and constructors are always variables (not necessarily different) and that all bound variables are completely fresh variables.

Definition 1 (normalization). *The normalization of an expression e proceeds in two stages. First, we flatten all the arguments of function (or constructor) calls by means of the mapping e^*, which is defined inductively as follows:*

$$\begin{array}{rl} x^* = & x \\ h(x_1,\ldots,x_n)^* = & h(x_1,\ldots,x_n) \\ h(x_1,\ldots,x_{i-1},e_i,e_{i+1},\ldots,e_n)^* = & let\ x_i = e_i^*\ in\ h(x_1,\ldots,x_{i-1},x_i,e_{i+1},\ldots,e_n)^* \\ & where\ e_i\ is\ not\ a\ variable\ and\ x_i\ is\ a\ fresh\ variable \\ (let\ \{\overline{x_k = e_k}\}\ in\ e)^* = & let\ \{\overline{x_k = e_k{}^*}\}\ in\ e^* \\ (e_1\ or\ e_2)^* = & e_1{}^*\ or\ e_2{}^* \\ ((f)case\ e\ of\ \{\overline{p_k \to e_k}\})^* = & (f)case\ e^*\ of\ \{\overline{p_k \mapsto e_k{}^*}\} \end{array}$$

Here, h denotes either a constructor or a function symbol.

The second stage consists in applying α-conversion in order to have fresh variable names for all bound variables in e. The extension of this normalization process to programs is straightforward.

Normalization introduces different let constructs for each non-variable argument. Trivially, this could be modified in order to produce one single let with the bindings for all non-variable arguments of a function (or constructor) call, which we assume for the subsequent examples.

For the definition of our semantics, we consider that both the program and the expression to be evaluated have been previously normalized as in Definition 1.

(VarCons) $\Gamma[x \mapsto t] : x \;\Downarrow\; \Gamma[x \mapsto t] : t$

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

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

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

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

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

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

(Guess) $\dfrac{\Gamma : e \;\Downarrow\; \Delta : x \quad \Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i) \;\Downarrow\; \Theta : v}{\Gamma : fcase\ e\ of\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v}$
where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are fresh

Fig. 1. Natural Semantics for Functional Logic Programs

Example 2. Consider again the program and goal of Example 1. Their normalized versions are as follows:

$$\begin{array}{lcl} \texttt{foo(x)} & = & \texttt{addB(x,x)} \\ \texttt{bit} & = & \texttt{0}\ or\ \texttt{1} \\ \texttt{addB(x,y)} & = & case\ \texttt{x}\ of\ \{0 \to 0;\ 1 \to case\ \texttt{y}\ of\ \{0 \to 1;\ 1 \to \texttt{B0}\}\} \end{array}$$

$$let\ \ \texttt{x1} = \texttt{bit}\ \ in\ \ \texttt{foo(x1)}$$

The state transition semantics is defined in Figure 1. Our rules obey the following naming conventions:

$$\Gamma, \Delta, \Theta \in Heap\ =\ Var \to Exp \qquad v \in Value\ ::=\ x \mid c(\overline{e_n})$$

Furthermore, we use x, y to denote variable names, t for constructor-rooted terms, and e for arbitrary expressions. A *heap* is a partial mapping from variables to expressions. The *empty heap* is denoted by $[\,]$. A *value* is a constructor-rooted term or a logical variable (w.r.t. the associated heap). We use judgements of the form "$\Gamma : e \;\Downarrow\; \Delta : v$", which should be interpreted as "the expression e in the context of the heap Γ evaluates to the value v with the (modified) heap Δ". Let us briefly explain the rules of our semantics in Figure 1.

(VarCons). In order to evaluate a variable which is bound to a constructor-rooted term in the heap, we simply reduce the variable to this term. The heap remains unchanged.

(VarExp). This rule achieves the effect of sharing. If the variable to be evaluated is bound to some expression in the heap, then the expression is evaluated and the heap is updated with the computed value; finally, we return this value as the result. In contrast to [17], we do not remove the binding for the variable from the heap; this becomes useful to generate fresh variable names easily. [22] solves this problem by introducing a variant of Launchbury's relation which is labeled with the names of the already used variables. The only disadvantage of our approach is that *black holes* (a detectably self-dependent infinite loop) are not detected at the semantical level. However, this does not affect to the natural semantics since black holes have no value.

(Val). For the evaluation of a value, we return it without modifying the heap.

(Fun). This rule corresponds to the unfolding of a function call. The result is obtained by reducing the right-hand side of the corresponding rule.

(Let). In order to reduce a let construct, we add the bindings to the heap and proceed with the evaluation of the main argument of the let. Note that we rename the variables introduced by the let construct with fresh names in order to avoid variable name clashes.

(Or). This rule non-deterministically evaluates an *or* expression by either evaluating the first argument or the second argument.

(Select). This rule corresponds to the evaluation of a case expression whose argument reduces to a constructor-rooted term. In this case, we select the appropriate branch and, then, proceed with the evaluation of the expression in this branch by applying the corresponding matching substitution.

(Guess). This rule corresponds to the evaluation of a flexible case expression whose argument reduces to a logical variable. Rule Guess non-deterministically binds this variable to one of the patterns and proceeds with the evaluation of the corresponding branch. Renaming of pattern variables is also necessary in order to avoid variable name clashes. Additionally, we update the heap with the (renamed) logical variables of the pattern.

A proof of a judgement corresponds to a derivation sequence using the rules of Figure 1. Given a normalized program $\mathcal{R}$ and a normalized expression e (to be evaluated), the *initial configuration* has the form "$[] : e$." We say that a derivation is *successful* if it computes a value. If we try to construct a derivation (starting from the left-bottom), then this may *fail* because of two different situations: there may be no finite proof that a reduction is valid—which corresponds to an infinite loop—or there may be no rule which applies in a (sub-part) of the proof. In the latter case, we have two possibilities: either rule Select is not applicable because there is no matching branch or rule Guess cannot be applied because a logical variable has been obtained as the argument of a rigid case expression. The natural semantics of Figure 1 does not distinguish between all the above failures. However, they will become observable in the small-step operational semantics.

The following result states that our natural semantics only computes values.

Lemma 1. *If $\Gamma : e \Downarrow \Delta : v$, then either v is rooted by a constructor symbol or it is a logical variable in Δ (i.e., $\Delta[v] = v$).*

4 A Small-Step Operational Semantics

From an operational point of view, an evaluation in the natural semantics builds a *proof* for "$[] : e_0 \Downarrow \Gamma : e_1$" in a bottom-up manner, whereas a computation by using a small-step semantics builds a sequence of states [22]. In order to transform a natural (big-step) semantics into a small-step one, we need to represent the *context* of sub-proofs in the big-step semantics. For instance, when applying the VarExp rule, a sub-proof for the premise is built. The context (i.e., the rule) indicates that we must update the heap Δ at x with the computed value v for the expression e. This context must be made explicit in the small-step semantics. In our case, the context is extensible (i.e., if P' is a sub-proof of P, then the context of P' is an extension of the context of P) and, thus, the representation of the context is made by a *stack*.

A configuration $\Gamma : e$ consists of a heap Γ and an expression e to be evaluated. Now, a *state* (or *goal*) of the small-step semantics is a triple (Γ, e, S), where Γ is the current heap, e is the expression to be evaluated (often called the *control* of the small-step semantics), and S is the stack which represents the current context. *Goal* denotes the domain $Heap \times Control \times Stack$.

The complete small-step semantics is presented in Figure 2. Let us briefly describe the transition rules. Rule varcons is perfectly analogous to rule VarCons in the natural semantics. In rule varexp, the evaluation of a variable x which is bound to an expression e (which is neither constructor-rooted nor a logical variable) proceeds by evaluating e and, then, adding to the stack the reference to variable x. Here, the stack S is a list (the empty stack is denoted by $[]$). When a variable x is on top of the stack, rule val updates the heap with $x \mapsto v$ once a value v is computed. Rules fun, let and or are quite similar to their counterparts in the natural semantics. Rule case initiates the evaluation of a case expression by evaluating the case argument and pushing the alternatives $(f)\{\overline{p_k \to e_k}\}$ on top of the stack. If we reach a constructor-rooted term, then rule select is used to select the appropriate branch and continue with the evaluation of this branch. If we reach a logical variable, then rule guess is used to non-deterministically choose one alternative and continue with the evaluation of this branch; moreover, the heap is updated with the binding of the logical variable to the corresponding pattern.

In order to evaluate an expression e, we construct an *initial goal* of the form $([\,], e, [\,])$ and apply the rules of Figure 2. We denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$. A derivation $([\,], e, [\,]) \Longrightarrow^* (\Gamma, e', S)$ is *successful* if e' is a head normal form (i.e., the computed *value*) and S is the empty list. The computed *answer* can be extracted from Γ by composing the bindings for the logical variables in the initial expression e. The equivalence of the small-step semantics and the natural semantics is stated in the following theorem.

Theorem 1. $([\,], e, [\,]) \Longrightarrow^* (\Delta, v, [\,])$ *if and only if* $[\,] : e \Downarrow \Delta : v$.

Rule		*Heap*	*Control*	*Stack*
varcons		$\Gamma[x \mapsto t]$	x	S
	$\Longrightarrow$	$\Gamma[x \mapsto t]$	t	S
varexp		$\Gamma[x \mapsto e]$	x	S
	$\Longrightarrow$	$\Gamma[x \mapsto e]$	e	$x : S$
val		Γ	v	$x : S$
	$\Longrightarrow$	$\Gamma[x \mapsto v]$	v	S
fun		Γ	$f(\overline{x_n})$	S
	$\Longrightarrow$	Γ	$\rho(e)$	S
let		Γ	$let\ \{\overline{x_k = e_k}\}\ in\ e$	S
	$\Longrightarrow$	$\Gamma[\overline{y_k \mapsto \rho(e_k)}]$	$\rho(e)$	S
or		Γ	$e_1\ or\ e_2$	S
	$\Longrightarrow$	Γ	e_i	S
case		Γ	$(f)case\ e\ of\ \{\overline{p_k \to e_k}\}$	S
	$\Longrightarrow$	Γ	e	$(f)\{\overline{p_k \to e_k}\} : S$
select		Γ	$c(\overline{y_n})$	$(f)\{\overline{p_k \to e_k}\} : S$
	$\Longrightarrow$	Γ	$\rho(e_i)$	S
guess		$\Gamma[x \mapsto x]$	x	$f\{\overline{p_k \to e_k}\} : S$
	$\Longrightarrow$	$\Gamma[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}]$	$\rho(e_i)$	S

where in varexp: e is not constructor-rooted and $e \neq x$
val: v is constructor-rooted or a variable with $\Gamma[y] = y$
fun: $f(\overline{y_n}) = e \in \mathcal{R}$ and $\rho = \{\overline{y_n \mapsto x_n}\}$
let: $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ fresh
or: $i \in \{1, 2\}$
select: $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$
guess: $i \in \{1, \ldots k\},\ p_i = c(\overline{x_n}),\ \rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ fresh

Fig. 2. Non-Deterministic Small-Step Semantics for Functional Logic Programs

5 A Deterministic Operational Semantics

The semantics presented in the previous section is still non-deterministic. In implementations of functional logic languages, this non-determinism is implemented by some search strategy. For debugging or profiling functional logic programs, it is necessary to model search strategies as well. Therefore, we extend the relation $\Longrightarrow$ as follows: $\Longrightarrow \subseteq \ \mathit{Goal} \times \mathit{Goal}^*$. The idea is that a computation step yields a sequence consisting of all possible successor states instead of non-deterministically selecting one of these states. Non-determinism occurs only in the rules or and guess of Figure 2. Thus, the deterministic semantics consists of all rules of Figure 2 except for the rules or and guess which are replaced by the deterministic versions of Figure 3. The main difference is that, in the deterministic versions, all possible successors are listed in the result of $\Longrightarrow$.

With the use of sequences, a search strategy (denoted by "$\circ$") can be defined as a function which composes two sequences of goals. The first sequence represents the new goals resulting from the last evaluation step. The second sequence represents the old goals which must be still explored. For example, a (left-to-

Rule	*Heap*	*Control*	*Stack*		(*Heap* × *Control* × *Stack*)*
or	Γ	e_1 *or* e_2	S	$\Longrightarrow$	$(\Gamma, e_1, S)\ (\Gamma, e_2, S)$
guess	$\Gamma[x \mapsto x]$	x	$f\{\overline{p_k \to e_k}\} : S$	$\Longrightarrow$	$(\Gamma[x \mapsto \rho_1(p_1), \overline{y_{n_1} \mapsto y_{n_1}}], \rho_1(e_1), S)$
					$\vdots$
					$(\Gamma[x \mapsto \rho_k(p_k), \overline{y_{n_k} \mapsto y_{n_k}}], \rho_k(e_k), S)$

where in guess: $p_i = c_i(\overline{x_{ni}})$, $\rho_i = \{\overline{x_{ni} \mapsto y_{ni}}\}$, and $\overline{y_{ni}}$ fresh

Fig. 3. Deterministic Small-Step Semantics for Functional Logic Programs

right) depth-first search strategy ($\circ_d$) and a breadth-first search strategy ($\circ_b$) can be specified as follows:

$$w \circ_d v = wv \quad \text{and} \quad w \circ_b v = vw$$

A small-step operational semantics (including search) which computes the first leaf in the search tree w.r.t. a search function "$\circ$" can be defined as the smallest relation $\longrightarrow \subseteq \mathit{Goal}^* \times \mathit{Goal}^*$ satisfying

$$\textsf{(Expand)} \quad \frac{g \implies G}{g\ G' \longrightarrow G \circ G'} \qquad \text{where } g \in \mathit{Goal} \text{ and } G, G' \in \mathit{Goal}^*$$

The evaluation starts with the initial goal $g_0 = ([\,], e_0, [\,])$ where e_0 is the expression to be evaluated. The relation $\longrightarrow$ is deterministic and it may reach four kinds of *final* states:

Solution: when $\Longrightarrow$ does not yield a successor because the first goal is a solution, i.e., it has the form $(\Gamma, v, [\,])$, where v is the computed value.

Suspension: when $\Longrightarrow$ does not yield a successor because the expression of the first goal is a rigid case expression with a logical variable in the argument position. This situation represents a suspended goal and will be discussed in more detail in the next section.

Fail: when $\Longrightarrow$ does not yield a successor because the value in the case expression of the first goal does not match any of the patterns.

No more goals: All goals have been explored and there are no solutions.

In order to distinguish the different situations, we add a label to the relation $\longrightarrow$ which classifies the leaves of the search tree. The label is computed by means of the following function:

$$\mathit{type}(\Gamma, e, S) = \begin{cases} \mathit{SUCC} & \text{if } e = v,\ S = [\,] \\ \mathit{SUSP} & \text{if } e = x,\ S = \{\overline{p_k \to e_k}\} : S',\ \text{and } \Gamma[x] = x \\ \mathit{FAIL} & \text{if } e = c(\overline{y_n}),\ S = (f)\{\overline{p_k \to e_k}\} : S', \\ & \text{and } \forall i = 1, \ldots, k.\ p_i \neq c(\ldots) \\ \mathit{EXPAND} & \text{otherwise} \end{cases}$$

With this function we can now define the complete evaluation of an expression:

$$\textsf{(Expand)} \ \frac{g \implies G}{g\ G' \overset{\mathit{EXPAND}}{\longrightarrow} G \circ G'} \qquad \textsf{(Discard)} \ \frac{g \not\Longrightarrow}{g\ G' \overset{\mathit{type}(g)}{\longrightarrow} G'}$$

where $g \in \mathit{Goal}$ and $G, G' \in \mathit{Goal}^*$. The (decidable) condition $g \not\Longrightarrow$ of the rule Discard means that none of the rules for $\Longrightarrow$ matches. In this case, $\longrightarrow$ does not perform an *EXPAND* step as the following lemma states. It can be shown by a simple case analysis over $\Longrightarrow$:[2]

Lemma 2. *If* $g_0 \longrightarrow^* g\ G'$ *and* $g \not\Longrightarrow$, *then* $\mathit{type}(g) \neq \mathit{EXPAND}$.

Now, one can extract the information of interest from the set of (possibly infinite) derivations. For example, the set of all solutions is defined by

$$\mathit{solutions}(g_0) = \{g \mid g_0 \longrightarrow^* g\ G \quad \stackrel{\mathit{SUCC}}{\longrightarrow} \quad G\}\ .$$

6 Adding Concurrency

Modern declarative multi-paradigm languages like Curry support concurrency. This makes multi-threading with communication on shared logical variables possible. The simplest semantics for concurrency is *interleaving*, which is usually defined at the level of a small-step semantics. The definition of a concurrent natural semantics would be much more complicated because of the additional don't-care non-determinism of interleaving.

For the formalization of concurrency, we extend the expressions and stacks in the goals to sequences of expressions and stacks, i.e., $\mathit{Goal} = \mathit{Heap} \times (\mathit{Control} \times \mathit{Stack})^*$. Each element of $(\mathit{Control} \times \mathit{Stack})^*$ represents a thread which can non-deterministically perform actions (which is the idea of the interleaving semantics). New threads are created with the conjunction operator `&` by extending the sequence with a new thread. The heap is a global entity for all threads in a goal, thus threads communicate with each other by means of variable bindings in this global heap.

The rules for the concurrent semantics are presented in Figure 4, where $T, T' \in (\mathit{Control} \times \mathit{Stack})^*$. The following possibilities for discarding a goal are distinguished in the context of the interleaving semantics:

(Fail) A goal fails if one of its threads fails.
(Succ) A goal is a solution if all threads terminate successfully.
(Deadlock) At least one thread suspends and all other ones suspend or succeed.

Our concurrent semantics is *indeterministic* (i.e., don't-care non-deterministic). An evaluation represents one trace of the system. During the evaluation of a goal, several threads may suspend and later be awoken by variable bindings produced by other threads. Then a step with $\Longrightarrow$ is again possible for the awoken process. A goal is only discarded in any of the three cases discussed above. Note that only the *FAIL* case can be interleaved with *EXPAND* steps while in the other two cases there is no alternative successor.

The rule Expand allows computation steps in an arbitrary thread of the first goal. If such a step is don't-know non-deterministic, i.e., yields more than one

[2] We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$ including all labels.

$$\text{(Expand)}\quad \frac{(\Gamma, e, S) \implies (\Gamma_1, e_1, S_1) \ldots (\Gamma_n, e_n, S_n)}{(\Gamma, T(e,S)T') : G \overset{EXPAND}{\longrightarrow} (\Gamma_1, T(e_1,S_1)T') \ldots (\Gamma_n, T(e_n,S_n)T') \circ G}$$

$$\text{(Fork)}\quad \frac{-}{(\Gamma, T(e_1 \& e_2, S)T') : G \overset{EXPAND}{\longrightarrow} (\Gamma, T(e_1,S)(e_2,S)T') \circ G}$$

$$\text{(Fail)}\quad \frac{type(\Gamma, e, S) = FAIL}{(\Gamma, T(e,S)T') : G \overset{FAIL}{\longrightarrow} G} \qquad \text{(Succ)}\quad \frac{\forall 1 \le i \le k : \ type(\Gamma, e_i, S_i) = SUCC}{(\Gamma, (e_1,S_1) \ldots (e_k,S_k)) : G \overset{SUCC}{\longrightarrow} G}$$

$$\text{(Deadlock)}\quad \frac{\begin{array}{c}\forall 1 \le i \le k : \ type(\Gamma, e_i, S_i) \in \{SUCC, SUSP\} \\ \text{and} \ \ \exists 1 \le j \le k : \ type(\Gamma, e_j, S_j) = SUSP\end{array}}{(\Gamma, (e_1,S_1) \ldots (e_k,S_k)) : G \overset{SUSP}{\longrightarrow} G}$$

Fig. 4. Concurrent Semantics for Multi-Paradigm Programs

goal, the entire process structure is copied. Although this is necessary to compute all solutions, it could be more efficient to perform a non-deterministic step only if a deterministic step in another thread is not possible. This strategy corresponds to *stability* in AKL [15] and Oz [21] and could be also specified in our framework.

7 Conclusions and Related Work

We have presented an operational semantics for functional logic languages based on lazy evaluation with sharing, concurrency, and non-determinism implemented by some search strategy. Such a semantics is important for a precise definition of various constructs of a language, like the combination of set-valued functions, laziness, and concurrency, as well as tools related to operational aspects of a language, like profilers and debuggers. We have developed our semantics in several steps. First, we transformed programs into a normalized form in order to make explicit the pattern matching strategy, common subexpressions, etc. Then, we defined for these normalized programs a natural semantics covering laziness, sharing, and non-determinism. We also presented a non-deterministic small-step semantics and proved its equivalence with the natural semantics. Finally, we refined the small-step semantics to describe the concepts not covered by the first semantics, namely search strategies and concurrency. Thus, the final semantics is an appropriate basis to define concrete functional logic programming languages like Curry, where mainly the handling of predefined functions needs to be added.

As far as we know, this is the first attempt of a rigorous operational description covering all these language features. Nevertheless, we want to compare our work with some other related approaches. Launchbury [17], Sestoft [22], and Sansom and Peyton Jones [20] gave similar descriptions for purely lazy functional languages where logic programming features and concurrency are not covered. [16] and [9] contain operational and denotational descriptions of Prolog with

the main emphasis on covering the backtracking strategy and the "cut" operator. Although the modeling of backtracking by the use of a goal sequence has some similarities with our description, laziness, sharing, and concurrency are not covered there. The same holds for Börger's descriptions of Prolog's operational semantics (e.g., [7, 8]) which consist of various small-step semantics for the different language constructs. Podelski and Smolka [19] defined an operational semantics for constraint logic programs with coroutining in order to specify the interaction of backtracking, cut, and coroutining. Their modeling of coroutining via "pools" is related to our model of concurrency, but demand-driven evaluation and sharing is not contained in their semantics. Operational semantics for functional logic programs with sharing can be found in [10, 14]. However, [10] does not model the pattern-matching strategy used in real implementations, and [14] does not model search strategies and concurrency (but allows partial applications in patterns).

In order to obtain a complete operational description of a practical language like Curry, one has to add descriptions for solving equational constraints and evaluating external functions and higher-order applications to the semantics presented in this paper. Since this is orthogonal to the other operational aspects (sharing, concurrency), it can be easily added in the style of [13]. Indeed, we have implemented an interpreter for Curry based on the operational description shown in this paper (see [2]). The interpreter is written in Haskell [20] and, thus, it can be easily adapted to Curry in order to obtain a meta-interpreter for Curry. In addition to our small-step semantics, the implementation also provides equational constraints and a garbage collector on the heap to execute larger examples. The results are quite encouraging.

For future work, we plan to use this semantics to define a cost-augmented semantics (which is useful for profiling), to develop debugging and optimization tools (like partial evaluators), and to check or derive new implementations (like in [22]) for Curry.

References

1. E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of LOPSTR 2000*, pages 103–124. Springer LNCS 2042, 2001.
2. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. An Operational Semantics for Declarative Multi-Paradigm Languages. Technical report, 2002. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
3. E. Albert and G. Vidal. Symbolic Profiling of Multi-Paradigm Declarative Languages. In *Proc. of LOPSTR 2001*. Springer LNCS, 2002. To appear.
4. S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of PPDP 2001*, pages 199–206. ACM Press, 2001.
5. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
6. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of FroCoS'2000*, pages 171–185. Springer LNCS 1794, 2000.

7. E. Börger. A logical operational semantics of full Prolog. Part I: Selection core and control. In *Proc. CSL'89*, pages 36–64. Springer LNCS 440, 1990.
8. E. Börger. A logical operational semantics of full Prolog. Part II: Built-in predicates for database manipulations. In *Mathematical Foundations of Computer Science '90*, pages 1–14. Springer LNCS 452, 1990.
9. S.K. Debray and P. Mishra. Denotational and Operational Semantics for Prolog. *Journal of Logic Programming (5)*, pages 61–91, 1988.
10. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An Approach to Declarative Programming based on a Rewriting Logic. *Journal of Logic Programming*, 40:47–87, 1999.
11. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of POPL'97*, pages 80–93. ACM, New York, 1997.
12. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
13. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.
14. T. Hortalá-González and E. Ullán. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. of FLOPS 2001*, pages 216–232. Springer LNCS 2024, 2001.
15. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proc. of ILPS'91*, pages 167–183. MIT Press, 1991.
16. N.D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. In S. Tärnlund, editor, *Proc. of ICLP'84*, pages 281–288, 1984.
17. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of POPL'93*, pages 144–154. ACM Press, 1993.
18. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
19. A. Podelski and G. Smolka. Operational Semantics of Constraint Logic Programs with Coroutining. In *Proc. of ICLP'95*, pages 449–463. MIT Press, 1995.
20. P.M. Sansom and S.L. Peyton-Jones. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, 1997.
21. C. Schulte and G. Smolka. Encapsulated Search for Higher-Order Concurrent Constraint Programming. In *Proc. of ILPS'94*, pages 505–520. MIT Press, 1994.
22. P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, 1997.
23. G. Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of PEPM'02*, pages 52–62. ACM Press, 2002.
24. D. H. D. Warren. Higher-Order Extensions to Prolog – Are they needed? *Machine Intelligence*, 10. Ellis Horwood, 1982.
25. D.H.D. Warren. An Abstract Prolog Instruction Set. Technical note 309, SRI International, Stanford, 1983.