A Deterministic Operational Semantics for Functional Logic Programs*

Elvira Albert[1] Michael Hanus[2] Frank Huch[2]

Javier Oliver[1] Germán 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. This paper introduces a deterministic operational semantics for functional logic programs including notions like laziness, sharing, concurrency, non-deterministic functions, etc. Our semantic description is important not only 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. Starting from a "big-step" semantics in natural style which relates expressions and their evaluated results—but it is not sufficient to cover concurrency, search strategies, or to reason about costs associated to particular computations—, we define a "small-step" operational semantics which actually covers the advanced 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 functional logic languages, like laziness, sharing, concurrency, logical variables, non-deterministic functions, higher-order, constraints, etc. For instance, the report on the multi-paradigm language Curry [12] contains a fairly precise operational semantics but covers sharing only informally. The operational semantics of the functional logic language Toy [16] is based on narrowing and sharing (without concurrency) but the formal definition is based on a narrowing calculus [9] which does not include a particular pattern-matching strategy. However, the latter becomes important if one wants to reason about costs of computations (see [6] for a discussion about narrowing strategies and calculi). [13] contains an operational semantics for a lazy narrowing strategy but it addresses neither concurrency nor aspects of search strategies.

* 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.

In order to provide theoretical foundations for reasoning about programs, correctness of implementations, optimizations, or costs of computations, we define an operational semantics covering the important aspects of current functional logic languages. We start from a *natural* semantic description which defines the intended results by relating expressions to values [2]. 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 [15]) but also for the correctness of computed results in the presence of non-confluent function definitions [9]. Our main contribution is the formalization of a more implementation-oriented semantics based on the definition of individual computation steps. Then, we extend such a semantics to cover concurrency and search strategies. The resulting semantic description provides a formal framework to reason about operational aspects of programs, for instance, in order to develop appropriate debugging tools. In particular, it is a basis to provide a comprehensive definition of Curry (in contrast to [10, 12] which contain only partial definitions). One can use it to prove the correctness of implementations by further refinements, as done in [19]. Furthermore, one can count the costs (time/space) associated to particular computations in order to justify optimizations [1, 5, 20] or to compare different search strategies.

The paper is organized as follows. In the next section we introduce some foundations for understanding the subsequent development. Section 3 recalls a semantic description for functional logic programs in natural style. This is refined in Section 4 to a semantics describing individual execution steps. Section 5 shows a deterministic version of the latter in order to make the search strategy explicit. In Section 6, we add concurrency in order to yield the final semantics for declarative multi-paradigm programs. Finally, we discuss in Section 7 an implementation of our semantics and conclude in Section 8.

2 Foundations

A main motivation for this work is to provide theoretical foundations for developing programming tools (like profilers, debuggers, optimizers) for declarative multi-paradigm languages. In order to be concrete, we consider Curry [10, 12] 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 [10] without formalizing the sharing of common subterms. The accurate definition of the latter aspect is one of the purposes of the subsequent sections.

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 as a result:

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

A subtle question is the meaning of such definitions if function calls are passed as parameters. Consider, e.g., the definition "`double x = x+x`" and the expression "`double (choose 1 2)`". Similarly to [9], 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 [15]. It is our purpose 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*.[1] 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 [7] 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 description; moreover, source programs can be automatically

[1] Although Curry allows rules with conditions, we will only consider unconditional rules for the sake of simplicity. This is not a real restriction since conditional rules can be translated into unconditional ones by the introduction of auxiliary functions, see [6].

translated into this flat form [11]. The syntax for programs in flat form can be summarized as follows:

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

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[2]

$$(f)\mathit{case}\ e\ \mathit{of}\ \{c_1(\overline{x_{n_1}}) \to e_1; \ldots; c_k(\overline{x_{n_k}}) \to e_k\}$$

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 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:

$$
\begin{array}{lll}
\mathtt{and}(\mathtt{x},\mathtt{y}) & = & \mathit{case}\ \mathtt{x}\ \mathit{of}\ \{\mathtt{True} \to \mathtt{y};\ \ \mathtt{False} \to \mathtt{False}\ \} \\
\mathtt{choose}(\mathtt{x},\mathtt{y}) & = & \mathtt{x}\ \mathit{or}\ \mathtt{y}
\end{array}
$$

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

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

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 [11]). Similarly, the higher-order features of current functional languages can be reduced to first-order definitions by introducing an auxiliary "apply" function [21]. 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 [8].

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 [6] 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 let binding of the form $\mathit{let}\ x = x\ \mathit{in}\ e$. 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 $\mathit{let}\ x = x, y = y\ \mathit{in}\ x + y$. Thus, such a (circular) let binding indicates logical variables in our semantic treatment. Let us note that circular bindings are also used in implementations of Prolog to represent logic variables [22].

3 A Natural Semantics

In this section, we recall a natural (big-step) semantics for lazy functional logic programs [2] which is in the midway between a (simple) denotational semantics and a (complex) operational semantic description for a concrete abstract machine. This 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:

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

Under a sharing-based implementation, the computation of "`foo(`e`)`" must evaluate the expression e only once. Therefore, the evaluation of the expression `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 the big-step semantics mainly follows the natural semantics defined in [15] 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, the (first-order) semantics for lazy functional logic programs is described in two separated phases. In the first phase, a normalization process is applied 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. *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:*

$$\begin{aligned}
x^* &= x \\
h(x_1,\ldots,x_n)^* &= h(x_1,\ldots,x_n) \\
h(x_1,\ldots,x_{i-1},e_i,e_{i+1},\ldots,e_n)^* &= \mathit{let}\ x_i = e_i^*\ \mathit{in}\ h(x_1,\ldots,x_{i-1},x_i,e_{i+1},\ldots,e_n)^* \\
&\quad\ \mathit{where}\ e_i\ \mathit{is\ no\ variable\ and}\ x_i\ \mathit{fresh} \\
(\mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e)^* &= \mathit{let}\ \{\overline{x_k = {e_k}^*}\}\ \mathit{in}\ e^* \\
(e_1\ \mathit{or}\ e_2)^* &= {e_1}^*\ \mathit{or}\ {e_2}^* \\
((f)\mathit{case}\ e\ \mathit{of}\ \{\overline{p_k \to e_k}\})^* &= (f)\mathit{case}\ e^*\ \mathit{of}\ \{\overline{p_k \mapsto {e_k}^*}\}
\end{aligned}$$

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 one new let construct 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 the natural semantics, we consider that both the program and the expression to be evaluated have been previously normalized.

Example 2. The normalization of the program and expression of Example 1 returns the program unchanged and the following expression:

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

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

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

where Var and Exp represent the domain of variables and expressions, respectively. 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

$$\text{(VarCons)} \quad \Gamma[x \mapsto t] : x \Downarrow \Gamma[x \mapsto t] : t$$

$$\text{(VarExp)} \quad \frac{\Gamma[x \mapsto e] : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v} \quad \begin{array}{l}\text{where } e \text{ is not constructor-rooted} \\ \text{and } e \neq x\end{array}$$

$$\text{(Val)} \quad \Gamma : v \Downarrow \Gamma : v \quad \begin{array}{l}\text{where } v \text{ is constructor-rooted} \\ \text{or a variable with } \Gamma[x] = x\end{array}$$

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

$$\text{(Let)} \quad \frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \Downarrow \Delta : v}{\Gamma : let\ \{\overline{x_k = e_k}\}\ in\ e \Downarrow \Delta : v} \quad \begin{array}{l}\text{where } \rho = \{\overline{x_k \mapsto y_k}\} \\ \text{and } \overline{y_k} \text{ are fresh variables}\end{array}$$

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

$$\text{(Select)} \quad \frac{\Gamma : e \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : \rho(e_i) \Downarrow \Theta : v}{\Gamma : (f)case\ e\ of\ \{\overline{p_k \to e_k}\} \Downarrow \Theta : v} \quad \begin{array}{l}\text{where } p_i = c(\overline{x_n}) \\ \text{and } \rho = \{\overline{x_n \mapsto y_n}\}\end{array}$$

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

Fig. 1. Natural Semantics for Lazy Functional Logic Programs

"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 in Figure 1.

(VarCons). If a variable is bound to a constructor-rooted term in the heap, this rule simply returns the associated 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, this value is returned as the result. In contrast to [15], the binding for the variable is not removed from the heap; this becomes useful to generate fresh variable names easily.

(Val). For the evaluation of a value, this rule trivially returns 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, this rule adds the bindings to the heap and, then, proceeds with the evaluation of the main argument of the let. Note that the variables introduced by the let construct are renamed with fresh names to avoid variable name clashes.

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

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\mathtt{x2} \mapsto \mathtt{bit}] : 1 \;\Downarrow\; [\mathtt{x2} \mapsto \mathtt{bit}] : 1}\;\mathsf{Val}}{[\mathtt{x2} \mapsto \mathtt{bit}] : 0 \mathrel{or} 1 \;\Downarrow\; [\mathtt{x2} \mapsto \mathtt{bit}] : 1}\;\mathsf{Or}}{[\mathtt{x2} \mapsto \mathtt{bit}] : \mathtt{bit} \;\Downarrow\; [\mathtt{x2} \mapsto \mathtt{bit}] : 1}\;\mathsf{Fun}}{[\mathtt{x2} \mapsto \mathtt{bit}] : \mathtt{x2} \;\Downarrow\; [\mathtt{x2} \mapsto 1] : 1}\;\mathsf{VarExp} \qquad \boxed{\mathtt{sub\text{-}proof}}}{[\mathtt{x2} \mapsto \mathtt{bit}] : \mathit{case}\ \mathtt{x2}\ \mathit{of}\ \{0 \to 0;\ 1 \to \mathit{case}\ \mathtt{x2}\ldots\} \;\Downarrow\; [\mathtt{x2} \mapsto 1] : \mathtt{B0}}\;\mathsf{Select}}{[\mathtt{x2} \mapsto \mathtt{bit}] : \mathtt{addB(x2,x2)} \;\Downarrow\; [\mathtt{x2} \mapsto 1] : \mathtt{B0}}\;\mathsf{Fun}}{[\mathtt{x2} \mapsto \mathtt{bit}] : \mathtt{foo(x2)} \;\Downarrow\; [\mathtt{x2} \mapsto 1] : \mathtt{B0}}\;\mathsf{Fun}}{[\,] : \mathit{let}\ \mathtt{x1} = \mathtt{bit}\ \mathit{in}\ \mathtt{foo(x1)} \;\Downarrow\; [\mathtt{x2} \mapsto 1] : \mathtt{B0}}\;\mathsf{Let}
$$

where `sub-proof` has the following form:

$$
\dfrac{\dfrac{}{[\mathtt{x2} \mapsto 1] : \mathtt{x2} \;\Downarrow\; [\mathtt{x2} \mapsto 1] : 1}\;\mathsf{VarCons} \qquad \dfrac{}{[\mathtt{x2} \mapsto 1] : \mathtt{B0} \;\Downarrow\; [\mathtt{x2} \mapsto 1] : \mathtt{B0}}\;\mathsf{Val}}{[\mathtt{x2} \mapsto 1] : \mathit{case}\ \mathtt{x2}\ \mathit{of}\ \{0 \to 1;\ 1 \to \mathtt{B0}\} \;\Downarrow\; [\mathtt{x2} \mapsto 1] : \mathtt{B0}}\;\mathsf{Select}
$$

Fig. 2. Big-step semantics of Example 2

(Select). This rule corresponds to the evaluation of a case expression whose argument reduces to a constructor-rooted term. In this case, the appropriate branch is selected and, then, the evaluation of the expression in this branch proceeds 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. It 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, the heap is updated 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 "$[\,] : e$." A derivation is *successful* if it computes a value. The computed *answer* can be extracted by composing the bindings for the logical variables in the initial expression e. If we try to construct a derivation (starting from the left-bottom), then it 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 of the next section.

Figure 2 illustrates the sharing behavior of the semantic description with one of the possible (non-deterministic) derivations for the program and expression of Example 2. Let us note that the heap in the final configuration, $[\mathtt{x2} \mapsto 1] : \mathtt{B0}$, does not contain bindings for the variable `x1` of the initial expression (due to the renaming of local variables in let expressions), which means that the computed answer is the empty substitution.

4 A Small-Step Semantics

From an operational point of view, an evaluation in the natural semantics builds a *proof* for "$[\,] : e_0 \Downarrow \Gamma : e_1$", whereas a computation by using a small-step semantics builds a sequence of states [19]. 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* [19] (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 3. 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 and 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 3. We denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$. A derivation $([\,], e, [\,]) \Longrightarrow^* (\Gamma, e', S)$ is *successful* if e' is in head normal form (the computed *value*) and S is the empty list. Similarly to the natural semantics, 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 as follows [2]:

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

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
	$\Longrightarrow \Gamma$	$\rho(e)$	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
	$\Longrightarrow \Gamma$	e_i	S
case	Γ	$(f)case\ e\ of\ \{\overline{p_k \to e_k}\}$	S
	$\Longrightarrow \Gamma$	e	$(f)\{\overline{p_k \to e_k}\} : S$
select	Γ	$c(\overline{y_n})$	$(f)\{\overline{p_k \to e_k}\} : S$
	$\Longrightarrow \Gamma$	$\rho(e_i)$	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. 3. 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 actual 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: $\Longrightarrow \subseteq$ *Goal* $\times$ *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 3. Thus, the deterministic semantics, presented in [3], consists of all rules of Figure 3 except for the rules or and guess which are replaced by the deterministic versions of Figure 4. The main difference is that, in the deterministic versions, all possible successors are listed in the result of $\Longrightarrow$.

Rule	*Heap*	*Control*	*Stack*		$(\mathit{Heap} \times \mathit{Control} \times \mathit{Stack})^*$
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: $p_i = c_i(\overline{x_{n\,i}})$, $\rho_i = \{\overline{x_{n\,i} \mapsto y_{n\,i}}\}$, and $\overline{y_{n\,i}}$ fresh

Fig. 4. Deterministic Small-Step Semantics for Functional Logic Programs

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

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

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

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

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. Furthermore, the computed answer can be obtained from the bindings in Γ for the variables of the initial expression e_0.

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:

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

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

$$(\mathsf{Expand})\ \frac{g \Longrightarrow G}{g\ G' \stackrel{EXPAND}{\longrightarrow} G \circ G'} \qquad (\mathsf{Discard})\ \frac{g \not\Longrightarrow}{g\ G' \stackrel{type(g)}{\longrightarrow} G'} \qquad \begin{pmatrix} g \in Goal \text{ and} \\ G, G' \in Goal^* \end{pmatrix}$$

The (decidable) condition $g \not\Longrightarrow$ of 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 $\Longrightarrow$):[3]

Lemma 1. *If $g_0 \longrightarrow^* g\ G'$ and $g \not\Longrightarrow$, then $type(g) \neq EXPAND$.*

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

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

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., $Goal = Heap \times (Control \times Stack)^*$. Each element of $(Control \times Stack)^*$ represents a thread and these threads 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, presented in [3], appear in Figure 5, where $T, T' \in (Control \times Stack)^*$. 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 threads suspend or succeed.

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

$$\textsf{(Expand)}\ \frac{(\Gamma, e, S) \ \Longrightarrow\ (\Gamma_1, e_1, S_1) \dots (\Gamma_n, e_n, S_n)}{(\Gamma, T(e,S)T') : G \ \overset{EXPAND}{\longrightarrow}\ (\Gamma_1, T(e_1, S_1)T') \dots (\Gamma_n, T(e_n, S_n)T') \circ G}$$

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

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

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

Fig. 5. Concurrent Semantics for Multi-Paradigm Programs

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.

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 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 [14] and Oz [18] and could also be specified in our framework without any problem.

We conjecture that $\longrightarrow$ is confluent up to variable renaming because the heap can only be extended. If the variable bindings of different threads in the shared heap clash, then this will happen in any scheduling policy since there is no committed choice construct.

7 Implementation

Our semantic description does not only provide the theoretical foundation to reason about lazy functional logic programs, but it can also be used as a basis to implement interpreters, debuggers, and optimization tools in a high-level manner. In order to get confidence in the latter aspect, we have implemented an interpreter for Curry based on the operational description shown in this paper and a depth-first search strategy. The interpreter is written in Haskell [17] and, thus, it can be easily adapted to Curry in order to obtain a meta-interpreter for Curry. The entire implementation consists of a front-end to compile Curry programs into the flat form introduced in Section 2 and an evaluator for expressions based on our small-step semantics. The implementation of the heap uses balanced search trees to ensure efficient access and update operations. 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. Standard functional programs are executed (using the Glasgow Haskell compiler) with approximately 24000 reductions per second on a 1.3 GHz Linux-PC (AMD Athlon with 256 KB cache). For logic programs involving search more than 2000 non-deterministic steps are executed per second. Although our interpreter is much slower than compilers based on back-ends implemented in low-level (non-declarative) languages, it is comparable to other meta-interpreters. In particular, it is faster than previous meta-interpreters for Curry (e.g., [4]) due to an improved handling of variable sharing. Thus, our implementation can be an appropriate basis for developing further tools like program optimizers based on partial evaluators, visualization tools, etc.

8 Conclusions and Future 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 semantic description 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, like profilers and debuggers. Moreover, we have proved its equivalence with the natural semantics of [2]. Our semantics provides an appropriate foundation to define lazy functional logic programming languages like Curry.

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. This is subject of ongoing work [3].

For future work, we plan to enhance this operational semantics with the computation of cost information (which is useful, e.g., for profiling [5, 17]). Furthermore, it could be interesting to use our operational semantics as a basis to develop debugging and optimization tools (like partial evaluators [4]), and to check or derive new implementations (like in [19]) for Curry.

References

1. E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of the 10th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'00)*, pages 103–124. Springer LNCS 2042, 2001.
2. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. An Operational Semantics for Functional Logic Languages. *Electronic Notes in Theoretical Computer Science*, ENTCS:76, 2002.
3. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Lazy Functional Logic Programs. In *Proc. of Workshop on Reduction Strategies in Rewriting and Programming (WRS'02)*, pages 97–112, 2002.

4. E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.
5. E. Albert and G. Vidal. Symbolic Profiling of Multi-Paradigm Declarative Languages. In *Proc. of Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'01)*, pages 148–167. Springer LNCS 2372, 2002.
6. S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206. ACM Press, 2001.
7. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
8. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the Int'l Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.
9. 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.
10. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of ACM Symp. on Principles of Programming Languages*, pages 80–93. ACM, New York, 1997.
11. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
12. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.
13. T. Hortalá-González and E. Ullán. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. of the 5th Int'l Symp. on Functional and Logic Programming (FLOPS 2001)*, pages 216–232. Springer LNCS 2024, 2001.
14. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proc. 1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.
15. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of ACM Principles of Programming Languages*, pages 144–154. ACM Press, 1993.
16. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
17. 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.
18. C. Schulte and G. Smolka. Encapsulated Search for Higher-Order Concurrent Constraint Programming. In *Proc. of the 1994 International Logic Programming Symposium*, pages 505–520. MIT Press, 1994.
19. P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, 1997.
20. G. Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 52–62. ACM Press, 2002.
21. D. H. D. Warren. Higher-Order Extensions to Prolog – Are they needed? In Michie Hayes-Roth and Pao, editors, *Machine Intelligence*, volume 10. Ellis Horwood, 1982.
22. D.H.D. Warren. An Abstract Prolog Instruction Set. Technical note 309, SRI International, Stanford, 1983.