An Operational Semantics for Declarative Multi-Paradigm Languages ⋆

Elvira Albert[1] Michael Hanus[2] Frank Huch[2]
Javier Oliver[3] Germán Vidal[3]

[1] *DSIP, UCM, Avda. Complutense s/n, E-28040 Madrid, Spain*
elvira@fdi.ucm.es

[2] *Institut für Informatik, CAU Kiel, Olshausenstr. 40, D-24098 Kiel, Germany*
{mh,fhu}@informatik.uni-kiel.de

[3] *DSIC, UPV, Camino de Vera s/n, E-46022 Valencia, Spain*
{fjoliver,gvidal}@dsic.upv.es

Abstract

Practical declarative multi-paradigm languages combine the main features of functional, logic and concurrent programming (e.g., laziness, sharing, higher-order, logic variables, non-determinism, search strategies). Usually, these languages also include interfaces to external functions as well as to constraint solvers. In this work, we introduce the first formal description of an operational semantics for realistic multi-paradigm languages covering all the aforementioned features in a precise and understandable manner. We also provide a deterministic version of the operational semantics which models search strategies explicitly. This deterministic semantics becomes essential to develop language-specific tools like program tracers, profilers, optimizers, etc. Finally, we extend the deterministic semantics in order to model concurrent computations. An implementation of the complete operational semantics has been undertaken.

1 Introduction

Modern declarative multi-paradigm languages combine the best features of functional, logic and concurrent programming. The definition of a precise operational semantics for these languages is not an easy task since one must cover notions like sharing, logical variables, non-determinism, search strategies, concurrency, etc., as well as the interactions among them. Recently, a se-

⋆ This work has been partially supported by CICYT TIC 2001-2705-C03-01, by the MCYT under grants HA2001-0059, HU2001-0019 and HI2000-0161, and by the DFG under grant Ha 2457/1-2.

mantic description for lazy functional logic languages has been given [2]. This work defined a rigorous operational description for functional logic languages based on lazy evaluation with sharing and non-determinism. Furthermore, it presented two characterizations of the operational semantics: a high-level description in natural style and a more detailed small-step semantics. The equivalence between both characterizations is also proved in that paper.

However, in order to obtain a complete operational semantics of a practical language (like Curry [14] or Toy [18]), one has to add descriptions for modeling search strategies and concurrency, for solving equational constraints, evaluating external functions and higher-order features. These extensions are orthogonal to the other operational aspects (sharing, non-determinism) and have not been formally described yet. It is the aim of this paper to provide an operational description for declarative multi-paradigm languages covering all the aforementioned features in a precise and understandable manner.

The starting point in this work is the (non-deterministic) "small-step" operational semantics of [2]. Firstly, we present an extension of this semantics in order to cover practical features like integer and floating point numbers, external functions (e.g., arithmetic operators), predefined constraints (unification) and higher-order functions. Then, we provide a deterministic version of the small-step semantics which makes the search strategy explicit. This deterministic description constitutes a formal basis to reason about implementation-oriented aspects of programs, e.g., to develop appropriate tracing, profiling, and debugging tools. For instance, one can instrument this semantics in order to count the costs (time/space) associated to particular computations (similarly to, e.g., [1,4,21,24]). This is useful to formally quantify the improvements achieved by a concrete program optimization and to compare different search strategies. Note that this approach would not be possible by considering a non-deterministic semantics, since it cannot properly describe the computation paths associated to a particular search strategy. Finally, we consider the use of threads to model concurrent computations and extend the previous semantics accordingly. Thus, we obtain a complete deterministic semantics which supports multi-threading with communication on logical variables.

To the best of our knowledge, this work is the first attempt to formally define the complete operational semantics of a realistic multi-paradigm language like Curry [14]. An implementation of this semantics has been undertaken. It can be useful to test language extensions, to check program optimizations, or to derive programming tools by designing instrumented versions.

This paper is organized as follows. In the next section we briefly describe the considered language. Section 3 recalls the small-step semantics of [2]. This is extended in Section 4 to cover the additional features of declarative multi-paradigm languages. Section 5 presents a deterministic version of the semantics and Section 6 adds concurrency so that the final semantics covers all the important features. In Section 7, we describe an implementation of our semantics and conclude in Section 8 with a comparison to related work.

2 The Language

As we have already mentioned, a main motivation for this work is to provide foundations for developing programming tools (like profilers, tracers, optimizers, etc.) for declarative multi-paradigm languages. In order to be concrete, we consider Curry [12,14] 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.

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* of the form "$f\, t_1 \ldots t_n$ `=` e" where f is a function, $t_1, \ldots, t_n$ are data (or constructor) 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] For instance, the addition on binary numbers, built from constructors `0`, `1` and `BO` (binary overflow), can be defined by the following rules:

```
addB 0 y  =  y

addB 1 0  =  1

addB 1 1  =  BO
```

(data constructors usually start with upper case letter and function application is denoted by juxtaposition). A rule is applicable if its left-hand side matches the current application. Functions are evaluated lazily so that the operational semantics of Curry is a conservative extension of lazy functional programming.

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, Curry programs can be automatically translated into this flat form [13]. The syntax for programs in flat form is summarized in Figure 1. We use P to denote a program, D a function definition, p a pattern and $e \in \mathit{Exp}$ 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 from $\mathit{Var} = \{x, y, z, \ldots\}$, data constructors (e.g., a, b, c,...), function symbols (e.g., f, g, h,...), case expressions, disjunctions (e.g., to represent *non-deterministic* or *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

[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 [5].

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

Fig. 1. Flat Representation for Programs

expression has the following 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.

As an example, we show the translation of function "`addB`" into flat form:

$$\texttt{addB(x,y)} \quad = \quad \mathit{case}\ \texttt{x}\ \mathit{of}\ \{\texttt{0} \to \texttt{y};\ \texttt{1} \to \mathit{case}\ \texttt{y}\ \mathit{of}\ \{\texttt{0} \to \texttt{1};\ \texttt{1} \to \texttt{B0}\}\}$$

Let bindings are in principle not required for translating Curry programs but they are convenient to express the *sharing* of subterms without the use of complex graph structures (like, e.g., [8,11]). Operationally, let bindings introduce new structures in memory that are updated after evaluation, which is essential in the presence of lazy computations and non-deterministic functions. We also assume that all *extra variables* (i.e., variables in the right-hand side of a function which do not appear in the left-hand side) are explicitly introduced in flat programs by a direct circular let binding. For instance, if the right-hand side e contains an extra variable x, it is denoted by $\mathit{let}\ x = x\ \mathit{in}\ e$. Throughout this paper, we call such variables which are bound to themselves *logical variables*. Our representation of logical variables does not exclude the use of other circular data structures, as in $\mathit{let}\ \texttt{x} = \texttt{1} : \texttt{x}\ \mathit{in}\ \ldots$

Curry also includes a number of practical features which will be described in the remaining of this section. In particular, Curry extends the optimal

[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*.

evaluation strategy of [6] by concurrent programming features. These are supported by a concurrent conjunction operator "`&`" on constraints (i.e., expressions of the built-in type `Success`). For instance, a constraint of the form "c_1 `&` c_2" is evaluated by solving both constraints c_1 and c_2 concurrently. Elementary constraints are `Success`, which is always satisfied, and *equational constraints* e_1 `=:=` e_2 between two expressions. The latter is satisfied if both expressions are reducible to a same ground constructor term (i.e., we consider the so-called *strict equality* [9,19]). Operationally, an equational constraint e_1 `=:=` e_2 is solved by evaluating e_1 and e_2 to unifiable constructor terms.

Higher-order features in Curry include partial function applications and lambda abstractions. In our (first-order) flat representation, higher-order functions are translated into applications to an auxiliary function *apply* [25]. This distinguished function can easily be defined by means of ordinary program rules (see the discussion in Section 4.3). However, the evaluation of higher-order applications containing free variables as functions is not allowed (i.e., such applications are suspended to avoid the use of higher-order unification [13]). Moreover, Curry also allows the use of functions which are not defined in the user's program (*external* functions), like arithmetic operators, basic input/output facilities, etc.

Let us illustrate some of the above features with an example. Consider the following rule defining a function to concatenate two lists (where `[]` denotes the empty list and `z:zs` a list with first element `z` and tail `zs`):

```
conc(xs,ys)  =  case xs of { []     -> ys;
                           (z:zs)   -> z : conc(zs,ys) }
```

Now, the equational constraint "`conc(p,s) =:= [1,2,3]`" is solved by instantiating variables `p` and `s` to lists so that their concatenation yields the list `[1,2,3]`. Thus, we can define a constraint which is satisfied if `p` is a prefix of the list `xs` as follows:

```
prefix(p,xs)  =  let s=s in conc(p,s) =:= xs
```

In order to show an example for higher-order programming, we define a higher-order constraint, `satisfyAll`, which takes a unary constraint `c` and a list `xs` as input; it is satisfied if all elements of `xs` satisfy the constraint `c`:

```
satisfyAll(c,zs)  =  case zs of { []     -> Success;
                                (x:xs)   -> apply(c,x)
                                            & satisfyAll(c,xs) }
```

where we use *apply* to denote function application. Now, we can combine this definition with our previous definition of `prefix` in order to compute a common prefix of a list of strings (strings are considered as lists of characters):

```
commonPrefix(p,xs)  =  satisfyAll(prefix(p),xs)
```

For instance, the solutions for the constraint

```
commonPrefix(p,["abc", "abda", "abab"])
```

are the instantiations `""`, `"a"`, or `"ab"` for the variable `p`.

3 Small-Step Operational Semantics

In this section, we recall the small-step operational semantics for lazy functional logic programs presented in [2]. This semantics covers notions like laziness, sharing, and non-determinism. It 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. These variables will be interpreted as references to express sharing and need not be pairwise different.

Definition 3.1 The normalization of an expression e flattens all the arguments of function (or constructor) applications by means of the mapping e^* which is inductively defined as follows:

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

Here, φ denotes either a constructor or a function symbol. The extension of this normalization process to programs is straightforward.

Normalization introduces one new *let* construct for each non-variable argument of a function (or constructor) application. Clearly, this could be modified in order to produce one single *let* with bindings for all non-variable arguments. For the definition of the small-step semantics, it is assumed that both the program and the expression to be evaluated have been previously normalized.

The state transition semantics can be found in Figure 2. The rules obey the following naming conventions:

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

A *heap* is a partial mapping from variables to expressions. The *empty heap* is denoted by $[\,]$. The value associated to variable x in heap Γ is denoted by $\Gamma[x]$. $\Gamma[x \mapsto e]$ denotes a heap with $\Gamma[x] = e$, i.e., we will use this notation either as a condition on a heap Γ or as a modification of Γ. A logical variable

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 varcons: t is constructor-rooted

varexp: e is not constructor-rooted and $e \neq x$

val: v is constructor-rooted or a variable with $\Gamma[v] = v$

fun: $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$

let: $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ are 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 $\overline{y_n}$ are fresh

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

x is represented in a heap Γ by a circular binding of the form $\Gamma[x] = x$, i.e., x is not instantiated w.r.t. Γ. A *value* is a constructor-rooted term or a logical variable (w.r.t. the associated heap).

The small-step semantics is defined on *states* (or *goals*) of the form (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. For stacks $S \in \mathit{Stack}$, we use the same notation as for lists. *Goal* denotes the domain $\mathit{Heap} \times \mathit{Control} \times \mathit{Stack}$.

Let us briefly describe the transition rules of the small-step semantics:

varcons: This rule is used to evaluate a variable which is bound (in the heap) to a constructor-rooted term. In this case, it simply returns this term.

varexp: It is used to evaluate a variable x which is bound to an expression e (which is neither constructor-rooted nor a logical variable). It proceeds by evaluating e and, then, adding to the stack the reference to variable x.

val: When a variable x is found on top of the stack, this rule updates the heap with the binding $x \mapsto v$ once a value v is computed.

fun: It performs the unfolding of a function application. Here, the program P is a global parameter of the calculus.

let: This rule is used to reduce a *let* construct by adding the associated (renamed) bindings to the heap and by continuing with the evaluation of the main argument of *let*.

or: It is used to non-deterministically evaluate an *or* expression by either evaluating the first argument or the second argument.

case: This rule initiates the evaluation of a case expression by evaluating the case argument and pushing the alternatives $(f)\{\overline{p_k \to e_k}\}$ on the stack.

select: If we reach a constructor-rooted term, then this rule is used to select the appropriate branch of the case expression and continue with the evaluation of this branch.

guess: If we reach a logical variable, then this rule is used to choose one alternative non-deterministically and continue with the evaluation of the selected 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.

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 in head normal form (the computed *value*) and S is the empty stack. The computed *answer* can be extracted from Γ by dereferencing the variables of the initial goal e.

Example 3.2 Consider the program composed of function "`addB`", defined

$$
\begin{array}{ll}
 & ([\,], \mathit{let}\ \mathtt{x1} = \mathtt{bit}\ \mathit{in}\ \mathtt{foo(x1)}, [\,]) \\
\overset{\mathsf{let}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{bit}], \mathtt{foo(x2)}, [\,]) \\
\overset{\mathsf{fun}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{bit}], \mathtt{addB(x2, x2)}, [\,]) \\
\overset{\mathsf{fun}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{bit}], \mathit{case}\ \mathtt{x2}\ \mathit{of}\ \{\mathtt{0} \to \mathtt{x2};\ \mathtt{1} \to \mathit{case}\ \mathtt{x2}\ \mathit{of}\ \{\mathtt{0} \to \mathtt{1};\ \mathtt{1} \to \mathtt{BO}\}\}, [\,]) \\
\overset{\mathsf{case}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{bit}], \mathtt{x2}, [\{\mathtt{0} \to \mathtt{x2};\ \mathtt{1} \to \mathit{case}\ \mathtt{x2}\ \mathit{of}\ \{\mathtt{0} \to \mathtt{1};\ \mathtt{1} \to \mathtt{BO}\}\}]) \\
\overset{\mathsf{varexp}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{bit}], \mathtt{bit}, [\mathtt{x2}, \{\mathtt{0} \to \mathtt{x2};\ \mathtt{1} \to \mathit{case}\ \mathtt{x2}\ \mathit{of}\ \{\mathtt{0} \to \mathtt{1};\ \mathtt{1} \to \mathtt{BO}\}\}]) \\
\overset{\mathsf{fun}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{bit}], \mathtt{0}\ \mathit{or}\ \mathtt{1}, [\mathtt{x2}, \{\mathtt{0} \to \mathtt{x2};\ \mathtt{1} \to \mathit{case}\ \mathtt{x2}\ \mathit{of}\ \{\mathtt{0} \to \mathtt{1};\ \mathtt{1} \to \mathtt{BO}\}\}]) \\
\overset{\mathsf{or}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{bit}], \mathtt{1}, [\mathtt{x2}, \{\mathtt{0} \to \mathtt{x2};\ \mathtt{1} \to \mathit{case}\ \mathtt{x2}\ \mathit{of}\ \{\mathtt{0} \to \mathtt{1};\ \mathtt{1} \to \mathtt{BO}\}\}]) \\
\overset{\mathsf{val}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{1}], \mathtt{1}, [\{\mathtt{0} \to \mathtt{x2};\ \mathtt{1} \to \mathit{case}\ \mathtt{x2}\ \mathit{of}\ \{\mathtt{0} \to \mathtt{1};\ \mathtt{1} \to \mathtt{BO}\}\}]) \\
\overset{\mathsf{select}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{1}], \mathit{case}\ \mathtt{x2}\ \mathit{of}\ \{\mathtt{0} \to \mathtt{1};\ \mathtt{1} \to \mathtt{BO}\}, [\,]) \\
\overset{\mathsf{case}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{1}], \mathtt{x2}, [\{\mathtt{0} \to \mathtt{1};\ \mathtt{1} \to \mathtt{BO}\}]) \\
\overset{\mathsf{varcons}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{1}], \mathtt{1}, [\{\mathtt{0} \to \mathtt{1};\ \mathtt{1} \to \mathtt{BO}\}]) \\
\overset{\mathsf{select}}{\Longrightarrow} & ([\mathtt{x2} \mapsto \mathtt{1}], \mathtt{BO}, [\,])
\end{array}
$$

Fig. 3. Evaluation of "*let* `x1` = `bit` *in* `foo(x1)`"

in Section 2, and the following functions:

$$
\begin{array}{lcl}
\mathtt{foo(x)} & = & \mathtt{addB(x, x)} \\
\mathtt{bit} & = & \mathtt{0}\ \mathit{or}\ \mathtt{1}
\end{array}
$$

Figure 3 details the computation steps performed in one of the possible (non-deterministic) derivations for the above program and the goal "`foo(bit)`" (which is written as "*let* `x1` = `bit` *in* `foo(x1)`" in normalized form). In each step, we indicate the transition rule which has been applied.

The example is borrowed from [2] where it is used to illustrate the sharing behavior of a big-step semantics. Similarly, we can also observe the effect of sharing in the small-step semantics where the shared subterm "`bit`" is evaluated only once. Note that the heap in the final goal, $[\mathtt{x2} \mapsto \mathtt{1}]$, does not contain a binding for the variable `x1` of the initial expression (due to the renaming of local variables). This corresponds to the fact that the computed answer is the empty substitution.

4 Language Extensions

We described so far an operational semantics for the kernel of a functional logic language. In this section, we extend the small-step operational semantics in order to cover extensions like integer and floating point numbers, external functions, predefined constraints (unification), and higher-order functions.

4.1 Equality

An important feature of logic languages is their ability to perform constraint solving in an efficient way. For equational constraints between terms, this is achieved by unification, where equations between variables are solved by binding these variables (instead of instantiating them to all possible values). Similarly, functional logic languages offer equational constraints between expressions containing defined functions. Since such functions can denote infinite terms, equational constraints are interpreted as strict equalities [9,19]: an *equational constraint* $e_1 \texttt{=:=} e_2$ is satisfiable if both arguments e_1 and e_2 can be reduced to unifiable constructor terms (i.e., expressions without occurrences of defined functions). Usually, this is implemented by a recursive evaluation of arguments to head normal form followed by the comparison of both arguments with a possible instantiation of logical variables.

In order to provide a generic definition of the above operational behavior, we need a way to evaluate arbitrary expressions to head normal form. In the basic language of Figure 1, the only way to enforce the evaluation of an expression to head normal form is the use of case expressions. This causes difficulties for large sets (or even infinite sets of constructors like numbers, see below). Therefore, we introduce a new predefined function $\mathit{hnf}(e_1, e_2)$ which first evaluates the argument e_1 to head normal form before it returns e_2 as result. In order to formally specify this behavior in our small-step operational semantics, we first perform the evaluation of the current expression e_1 and push a *hnf* context containing e_2 on the stack. This element is popped off the stack when the first element is in head normal form. Thus, the operational semantics of *hnf* is formally defined by the following rules:

Rule		*Heap*	*Control*	*Stack*
hnf_1		Γ	$\mathit{hnf}(x_1, x_2)$	S
	$\Longrightarrow$	Γ	x_1	$\mathit{hnf}(x_2) : S$
hnf_2		Γ	v	$\mathit{hnf}(x) : S$
	$\Longrightarrow$	Γ	x	S

where v is a constructor-rooted term or a variable y with $\Gamma[y] = y$.

With the use of function *hnf*, arbitrary expressions can be evaluated to head normal form. This fact is exploited in the following definition of the strict equality (note that this definition needs to be normalized as any other program rule):

$$x_1 \texttt{=:=} x_2 \;=\; \mathit{hnf}(x_1, \mathit{hnf}(x_2, \mathit{prim\_constrEq}(x_1, x_2)))$$

This definition ensures that x_1 and x_2 are reduced to head normal form, i.e., a constructor-rooted term or a logical variable. The primitive function *prim_constrEq* recursively descends its two arguments and restarts the small-

step operational semantics for subexpressions by putting new expressions into the control. In the case of a successful unification, it yields a modified heap and the result `Success` (an internal constructor to represent the successful solving of a constraint).

The precise definition of the behavior of *prim_constrEq* causes a new complication due to unification. Since logical variables are not always instantiated to constructor-rooted terms (as in rule guess) but can also be bound to other logical variables, chains of bindings might occur in the heap. For instance, if we unify variable x to y and then later unify y with constant 0, then x is not directly bound to 0 and we have a heap Γ with $\Gamma[x] = y$ and $\Gamma[y] = 0$. This property requires the *dereferencing* of heap variables before we access them. We express this by a function Γ^* which is defined as follows:

$$\Gamma^*(x) = \begin{cases} \Gamma^*(y) & \text{if } \Gamma[x] = y \text{ and } x \neq y \\ \Gamma[x] & \text{otherwise} \end{cases}$$

Note that $\Gamma^*(x) = y$ implies that y is a logical variable (i.e., $\Gamma[y] = y$). In the following rules, we will apply Γ^* only to variables x which were already evaluated to head normal form, i.e., $\Gamma^*(x)$ is always a value. Now, we can define the small-step semantics of *prim_constrEq* as follows:

Rule	*Heap*	*Control*	*Stack*
$\mathsf{constrEq}_1$	Γ	$prim\_constrEq(x, y)$	S
	$\Longrightarrow \Gamma[x' \mapsto y']$	`Success`	S
$\mathsf{constrEq}_2$	Γ	$prim\_constrEq(x, y)$	S
	$\Longrightarrow \Gamma[x' \mapsto c(\overline{x_n}), \overline{x_n \mapsto x_n}]$	$(x_1 \texttt{=:=} y_1 \texttt{\&>} \ldots \texttt{\&>} x_n \texttt{=:=} y_n)^*$	S
$\mathsf{constrEq}_3$	Γ	$prim\_constrEq(x, y)$	S
	$\Longrightarrow \Gamma[y' \mapsto c(\overline{y_n}), \overline{y_n \mapsto y_n}]$	$(x_1 \texttt{=:=} y_1 \texttt{\&>} \ldots \texttt{\&>} x_n \texttt{=:=} y_n)^*$	S
$\mathsf{constrEq}_4$	Γ	$prim\_constrEq(x, y)$	S
	$\Longrightarrow \Gamma$	$(x_1 \texttt{=:=} y_1 \texttt{\&>} \ldots \texttt{\&>} x_n \texttt{=:=} y_n)^*$	S

where in $\mathsf{constrEq}_1$: $\Gamma^*(x) = x'$ and $\Gamma^*(y) = y'$

$\mathsf{constrEq}_2$: $\Gamma^*(x) = x'$, $\Gamma^*(y) = c(\overline{y_n})$, and $\overline{x_n}$ are fresh

$\mathsf{constrEq}_3$: $\Gamma^*(x) = c(\overline{x_n})$, $\Gamma^*(y) = y'$, and $\overline{y_n}$ are fresh

$\mathsf{constrEq}_4$: $\Gamma^*(x) = c(\overline{x_n})$ and $\Gamma^*(y) = c(\overline{y_n})$

In the rules above, equational constraints are solved in an incremental way by an interleaved lazy evaluation of expressions and binding of variables to

constructor terms. In particular, when both arguments of the equational constraint, x and y, are bound in the heap to logical variables, x' and y', rule $\mathsf{constrEq}_1$ returns `Success` and updates the heap by binding x' to y'. In rule $\mathsf{constrEq}_2$, variable x is bound to a logical variable x' but variable y is bound to a constructor application $c(\overline{y_n})$. In this case, we bind x' to a constructor application of the form $c(\overline{x_n})$, where $\overline{x_n}$ are fresh variable names, and constraint equality is checked for the constructor arguments. Since the number of arguments which must be compared recursively depends on the arity of constructor c, we put a new expression (in normalized form) containing the sequential conjunction operator "`&>`" on the control. Here, we consider an empty conjunction ($n = 0$) as equivalent to `Success`. The operator "`&>`" on constraints is defined as follows:

$$x_1 \texttt{ \&> } x_2 \;=\; \mathit{case}\ x_1\ \mathit{of}\ \{\texttt{Success} \to x_2\}$$

Rule $\mathsf{constrEq}_3$ proceeds in a similar manner. Finally, if both arguments, x and y, are bound to the same constructor application, rule $\mathsf{constrEq}_4$ continues with the comparison of the constructor arguments (without modifying the heap).

For the sake of simplicity, we have omitted the occur check in rules $\mathsf{constrEq}_2$ and $\mathsf{constrEq}_3$. For instance, in rule $\mathsf{constrEq}_2$ the occur check must ensure that variable x' does not occur in the value represented by y (if x' and y are different). Here, the value represented by y is the part of the expression recursively referred to by y (according to the current heap) without considering function applications (see [14, Appendix D.4] for more details).

We can also define the Boolean test equality function "`==`" for testing the strict equality of two expressions in a similar way. In contrast to "`=:=`", function "`==`" is only defined on ground constructor terms (i.e., it suspends in the presence of logical variables) and returns `True` (resp. `False`) if both terms are identical (resp. different). Function "`==`" can be defined as follows:

$$x_1 \texttt{ == } x_2 \;=\; \mathit{hnf}(x_1, \mathit{hnf}(x_2, \mathit{prim\_boolEq}(x_1, x_2)))$$

where *prim_boolEq* recursively checks its two arguments for equality:

Rule		*Heap*	*Control*	*Stack*
boolEq_1		Γ	$\mathit{prim\_boolEq}(x, y)$	S
	$\Longrightarrow$	Γ	$(x_1 \texttt{ == } y_1 \texttt{ \&\& } \ldots \texttt{ \&\& } x_n \texttt{ == } y_n)^*$	S
boolEq_2		Γ	$\mathit{prim\_boolEq}(x, y)$	S
	$\Longrightarrow$	Γ	`False`	S

where in boolEq_1: $\Gamma^*(x) = c(\overline{x_n})$ and $\Gamma^*(y) = c(\overline{y_n})$

boolEq_2: $\Gamma^*(x) = c(\ldots), \Gamma^*(y) = d(\ldots)$, and $c \neq d$

In rule boolEq_1 the operator `&&` denotes the Boolean conjunction, which is

defined as follows:

$$x_1 \,\texttt{\&\&}\, x_2 \;=\; \mathit{case}\ x_1\ \mathit{of}\ \{\texttt{True} \to x_2;\ \texttt{False} \to \texttt{False}\}$$

Furthermore, we consider an empty conjunction ($n = 0$) as equivalent to `True`.

4.2 External Functions

Every realistic programming language must support a number of functions that are not implemented in the same programming language. Let us consider, for instance, arithmetic operators which are used to perform computations on numbers. Conceptually, the infinite set of integers or floating point numbers can be interpreted as an infinite set of constants (0-ary constructors). In the following, we will call these constants *literals*. Literals can occur everywhere in programs, including the patterns of case expressions. For instance, we could also interpret arithmetic functions computing with integers (e.g., addition on integers) as defined by an infinite set of program rules. Since case expressions have only a finite number of branches, we cannot represent such an infinite set in our kernel language. This requires an extension of the language in order to include externally defined functions, i.e., functions which are not explicitly defined by program rules. Such functions are called *external functions*.

In a naive approach, one could try to extend our operational semantics to cover external functions with a generic rule like

$$\langle \Gamma,\ F(\overline{e_n}),\ S\rangle \quad\Longrightarrow\quad \langle \Gamma,\ F_{\mathcal{A}}(\overline{e_n}),\ S\rangle$$

where the semantics of each predefined function F is represented by means of an interpretation $F_{\mathcal{A}}$. However, in general, this is not sufficient since the arguments of F are expressions that need to be evaluated to literals before we interpret them with $F_{\mathcal{A}}$. Thus, we must ensure that these expressions are evaluated to literals before the function $F_{\mathcal{A}}$ is applied.

Similarly to equational constraints, we use the primitive *hnf* to solve this problem. For example, we define the addition of two integers with the use of the external function *prim_+* by the rule

$$x1 + x2 \;=\; \mathit{hnf}(x1, \mathit{hnf}(x2, \mathit{prim\_+}(x1, x2))$$

Since the primitive function *prim_+* is always applied to arguments which are already evaluated to literals (or logical variables, see below), we define its small-step semantics as follows:

Rule		*Heap*	*Control*	*Stack*
prim_+		Γ	$\mathit{prim\_+}(x, y)$	S
	$\Longrightarrow$	Γ	$l_1 +_{\mathcal{A}} l_2$	S

where $\Gamma^*(x) = l_1$, $\Gamma^*(y) = l_2$, l_1, l_2 are literals, and $+_{\mathcal{A}}$ denotes the arithmetic sum. Note that this definition implies that the evaluation of *prim_+* suspends (there is no successor in $\Longrightarrow$) if one of the arguments is a logical variable.

The definition of rules for the remaining primitive functions of a language like Curry could easily be done in a similar manner.

4.3 Higher-Order Features

According to the syntax of Figure 1, flat programs are restricted to first-order. In principle, this is sufficient since it is well known (e.g., [25]) that the higher-order features of typical functional (logic) languages can be translated into applications of a distinguished function *apply* which can be defined by a set of first-order rules. For instance, an expression like "$(f\ a)\ b$" can be translated into $apply(apply(f,a),b)$ where the definition of *apply* contains the following rules for the binary function f:

$$\begin{aligned} apply(f,x) &= f(x) \\ apply(f(x),y) &= f(x,y) \end{aligned}$$

In order to avoid the generation of these rules for all functions of the program, we provide a definition of *apply* based on a primitive function *prim_apply* which assumes that the first argument is in head normal form; note that only the first argument of *apply* must be evaluated to head normal form. Thus, we define *apply* by the following rule:

$$apply(x_1,x_2) = hnf(x_1, prim\_apply(x_1,x_2))$$

The small-step semantics is then extended as follows:

Rule		*Heap*	*Control*	*Stack*
apply		Γ	$prim\_apply(x,y)$	S
	$\Longrightarrow$	Γ	$\varphi(\overline{x_k},y)$	S

where $\Gamma^*(x) = \varphi(\overline{x_k})$ and either φ is a constructor symbol or $\varphi(\overline{y_n}) = e \in P$ with $k < n$. For user-defined functions, the condition $k < n$ is necessary since "over-applications" are possible in higher-order languages, as the following example shows (for clarity, the program is not normalized):

```
f(x)   = g(x)
g(x,y) = 42
h      = apply(apply(f,1),2)
```

In the definition of function `h`, it may seem that `f` is applied to two arguments. However, this is an over-application and rule `fun` must directly unfold function `f` once `f` is applied to one argument. For constructors, a similar condition on the arity of φ is not necessary since the type system of the source language should avoid over-applications of constructors.

Note that our definition requires a partial application like `and(True)` to be considered as a constructor-rooted term. This means that functions with

Rule	*Heap*	*Control*	*Stack*		$(Heap \times Control \times Stack)^*$
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 $\overline{y_{n\,i}}$ are fresh variables

Fig. 4. Deterministic Small-Step Semantics

missing arguments are considered as constructor-rooted terms. However, these constructors are "hidden" and only defined for the purpose of the operational semantics, i.e., they do not appear in patterns.

5 A Deterministic Operational Semantics

The semantics presented so far is still non-deterministic. In actual declarative multi-paradigm languages, this non-determinism is implemented by some search strategy. For tracing or profiling, it is necessary to model search strategies as well. For instance, consider the computation of *costs* associated to a program execution. In this case, by considering an *instrumented* non-deterministic semantics, we could only compute the cost of *each single derivation* in the search tree. However, we could not calculate the cost of a computation path within the search tree, since some computation steps may be shared by more than one derivation. Thus, it becomes essential to provide a deterministic version of the semantics which properly models search strategies. For this purpose, 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 2. Thus, the deterministic semantics consists of all the rules presented so far except for the rules or and guess which are replaced by the deterministic version of Figure 4. The only 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 still be explored. For example, a (left-to-right) depth-first search strategy ($\circ_d$) and a breadth-first search strategy ($\circ_b$) can easily 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 $\longrightarrow \subseteq Goal^* \times Goal^*$ satisfying

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

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. In this case, the first goal in the sequence has the form $(\Gamma, v, [\,])$, where v is the computed value. Furthermore, the computed answer can be extracted from Γ by dereferencing the variables of the initial expression e_0.

Suspension. Then, the expression of the first goal in the sequence should be either a rigid case expression with a logical variable in the argument position or a primitive function applied to some logical variable. Note that, in the latter case, not all primitive functions suspend on logical variables, e.g., *prim_constrEq* performs unification in this case. This situation represents a suspended goal and will be discussed in more detail in the next section.

Fail. Here, the first goal of the sequence should be either a case expression whose argument does not match any of the patterns or the application of a primitive function which does not succeed, e.g., *prim_constrEq* applied to values with different outermost constructors.

No more goals: This situation occurs when all the goals in the sequence have already been explored.

In order to distinguish the different possibilities, 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*. For expressions e that are not primitive function applications (i.e., $e \neq prim\_f(\overline{x_n})$), it is defined as follows:

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

For primitive functions, it is defined by using a function *primType* representing their behavior:

$$type(\Gamma, prim\_f(\overline{x_n}), S) = primType(\Gamma, prim\_f(\overline{x_n}), S)$$

The function *primType* represents the behavior of any primitive function. In particular, $primType(\Gamma, prim\_f(\overline{x_n}), S) = COMP$ iff $(\Gamma, prim\_f(\overline{x_n}), S) \implies G$ for some G. For instance, for the external function *prim_+*, it is defined as

follows:

$$primType(\Gamma, prim\_+(x,y), S) = \begin{cases} SUSP & \text{if } \Gamma^*(x) = z \text{ or } \Gamma^*(y) = z \\ COMP & \text{otherwise} \end{cases}$$

where z is a logical variable. Similar definitions can be provided for the remaining primitive functions. In particular, for constraint equality, suspension is not a possible behavior. Moreover, constraint equality may fail when it is applied to different constructors:

$$primType(\Gamma, prim\_constrEq(x,y), S) = \begin{cases} FAIL & \text{if } \Gamma^*(x) = c(\overline{y_n}),\ \Gamma^*(y) = d(\overline{z_m}), \text{ and } c \neq d \\ COMP & \text{otherwise} \end{cases}$$

With the use of function *type*, we can now define the complete evaluation of an expression as follows:

$$\text{(Eval)}\ \frac{g \implies G}{g\ G' \overset{COMP}{\longrightarrow} G \circ G'} \qquad \text{(Discard)}\ \frac{g \not\implies}{g\ G' \overset{type(g)}{\longrightarrow} G'} \qquad (g \in Goal \text{ and } G, G' \in Goal^*)$$

The (decidable) condition $g \not\implies$ of rule Discard means that none of the rules for $\implies$ matches. In this case, $\longrightarrow$ does not perform a *COMP* step as the following lemma states:[3]

Lemma 5.1 *If $g_0 \longrightarrow^* g\ G'$ and $g \not\implies$, then $type(g) \neq COMP$.*

Proof. Case analysis on $g = (\Gamma, e, S)$:

- e is a value. We distinguish two cases:

(i) $e = c(\overline{x_n})$:
If $S = x : S'$, then val is applicable and $g \implies G$.
If $S = (f)\{\overline{p_k \longrightarrow e_k}\} : S'$, then either $e = c(\overline{y_n})$ or $e = x$ with $\Gamma[x] = x$. In the first case, rule select is applicable or $type(g) = FAIL$; in the second case, we can either apply rule guess or $type(g) = SUSP$.
If $S = [\,]$, then no rule is applicable and $type(g) = SUCC$.

(ii) $e = x$:
If $S \neq [\,]$ and $S \neq \{\overline{p_k \longrightarrow e_k}\} : S'$, then either rules varcons, varexp or guess are applicable, $type(g) = SUCC$, or $type(g) = FAIL$.

- $e = prim\_f(\overline{x_n})$. The function *primType* yields the type *COMP* exactly in the same cases where $\implies$ yields a successor.
- e is any other expression. Then, for all possible expressions, there exists a rule applicable independently of Γ and S.

□

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

$$\text{(Eval)}\ \frac{(\Gamma, e, S) \implies (\Gamma_1, e_1, S_1) \ldots (\Gamma_n, e_n, S_n)}{(\Gamma, T \oplus (e, S)) : G \overset{COMP}{\longrightarrow} (\Gamma_1, T \oplus (e_1, S_1)) \ldots (\Gamma_n, T \oplus (e_n, S_n)) \circ G}$$

$$\text{(Fork)}\ \frac{-}{(\Gamma, T \oplus (e_1 \,\&\, e_2, S)) : G \overset{COMP}{\longrightarrow} (\Gamma, T \oplus (e_1, S) \oplus (e_2, S)) \circ G}$$

$$\text{(Succ}_1\text{)}\ \frac{type(\Gamma, e, S) = SUCC}{(\Gamma, T \oplus (e, S)) : G \overset{COMP}{\longrightarrow} (\Gamma, T) \circ G} \qquad \text{(Succ}_2\text{)}\ \frac{-}{(\Gamma, \emptyset) : G \overset{SUCC}{\longrightarrow} G}$$

$$\text{(Fail)}\ \frac{type(\Gamma, e, S) = FAIL}{(\Gamma, T \oplus (e, S)) : G \overset{FAIL}{\longrightarrow} G} \qquad \text{(Deadlock)}\ \frac{\forall (e, S) \in T : type(\Gamma, e, S) = SUSP}{(\Gamma, T) : G \overset{SUSP}{\longrightarrow} G}$$

Fig. 5. Concurrent Semantics for Multi-Paradigm Programs

The relation $\longrightarrow$ contains all information of a computation. One can easily extract the part of interest from the (possibly infinite) derivation. For example, the set of all solutions can be defined in the following way:

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

6 Adding Concurrency

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

In this section, we show how our deterministic semantics $\longrightarrow$ can be extended naturally to model concurrency. For simplicity, we restrict the considered concurrent programs by requiring that the initial expression is always a constraint (i.e., `main` is of type `Success`).

For the formalization of concurrency (see Figure 5), we extend the expressions and stacks in the goals to sets of expressions and stacks, i.e., $Goal = Heap \times \mathcal{P}(Control \times Stack)$. Each element of $\mathcal{P}(Control \times Stack)$ represents a thread and these threads can perform actions non-deterministically (which is the idea of an interleaving semantics). As an abbreviation for the disjoint union $T \uplus \{(e, S)\}$ we write $T \oplus (e, S)$. New threads are created with the concurrent conjunction operator "`&`" by adding the new thread to the set (Fork). 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.

In our concurrent operational semantics, the following possibilities for dis-

carding a goal are distinguished:

FAIL A goal fails if one of its threads fails.

SUCC A goal is a solution if all threads terminate successfully.

SUSP A goal represents a deadlock situation if all threads suspend.

The concurrent semantics is *indeterministic* (in other words, 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 from other threads. Then, a step with $\Longrightarrow$ is again possible for the awoken process. A goal is only discarded in one of the three cases discussed above. Note that there is only a non-deterministic choice possible between rules Eval, Fork, Succ_1, and Fail. In the application of rules Succ_2 and Deadlock, there is no alternative successor.

The rule Eval allows computation steps in an arbitrary thread of the first goal. If such a step is don't-know non-deterministic, i.e., it 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 [16] and Oz [22] and could easily be specified in our framework, too.

We conjecture that $\longrightarrow$ is confluent (up to variable renaming) for fair search strategies, like breadth-first search. The reason is that the heap can only be extended and logical variables can only be bound to one value. If the variable bindings of different threads in the shared heap clash, then this will happen in any scheduling policy due to the absence of committed choice construct.

7 Implementation

Our semantic description does not only provide the theoretical foundation to reason about actual multi-paradigm functional logic programs but it can also be used as a basis to implement abstract machines, 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 work.

The interpreter is written in Haskell. Thus, it can easily be 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. The implementation also includes a garbage collector on the heap to be able 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, its performance is comparable to other meta-interpreters. In particular, it is faster than previous meta-interpreters for Curry (e.g., [3]) 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 Related Work

We have presented an operational semantics for declarative multi-paradigm languages covering features like laziness, sharing, non-determinism, higher-order functions, equational constraints, and external functions. Furthermore, we have extended this semantic description in several ways. First, we have provided a deterministic version of the operational semantics which makes the search strategy explicit. This is especially important for the development of programming tools related to the operational aspects of a language, like profilers and tracers. Then, we have refined our small-step semantics in order to consider concurrent programming features with a distribution of the computation tasks. Therefore, the developed semantics provides an appropriate foundation to model declarative multi-paradigm languages like Curry [14].

Let us compare our work with some other related approaches. Operational semantics for functional logic programs with sharing can be found in [10] and [15]. However, [10] does not model the pattern-matching strategy used in real implementations, and [15] does not model search strategies and concurrency (but allows partial applications in case patterns). Sestoft [23] proposes similar descriptions for purely lazy functional languages where logic programming features and concurrency are not covered. [17] and [7] contain operational and denotational descriptions of Prolog with the main emphasis on covering the backtracking strategy and the "cut" operator. Although our modeling of search strategies by the use of goal sequences has some similarities with their description, laziness, sharing, and concurrency are not covered there. Podelski and Smolka [20] define 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.

For future work, we plan to enhance this operational semantics with the computation of cost information (which is useful, e.g., for profiling [4,21] or for formally checking the improvement achieved by program optimizations [1,24]). Furthermore, it could be interesting to use our operational semantics as a basis to develop debugging and optimization tools (like partial evaluators

[3]), and to check or derive new implementations (like in [23]) 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. Operational Semantics for Functional Logic Languages. In *Proc. of Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP'02)*. Electronic Notes in Theoretical Computer Science vol. 76, 2002.

[3] 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.

[4] 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.

[5] 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.

[6] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[7] S.K. Debray and P. Mishra. Denotational and Operational Semantics for Prolog. *Journal of Logic Programming (5)*, pages 61–91, 1988.

[8] R. Echahed and J. Janodet. Admissible Graph Rewriting and Narrowing. In *Proc. of the 1998 Joint Int'l Conference and Symposium on Logic Programming (JICSLP'98)*, pages 325–340. MIT Press, 1998.

[9] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.

[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] A. Habel and D. Plump. Term Graph Narrowing. *Mathematical Structures in Computer Science*, 6(6):649–676, 1996.

[12] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM, New York, 1997.

[13] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[14] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.

[15] 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.

[16] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proc. of the Int'l Logic Programming Symposium (ILPS'91)*, pages 167–183. MIT Press, 1991.

[17] N.D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. In S. Tärnlund, editor, *Proc. of 2nd Int'l Conf. on Logic Programming (ICLP'84)*, pages 281–288, 1984.

[18] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of the Int'l Conf. on Rewrite Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[19] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.

[20] A. Podelski and G. Smolka. Operational Semantics of Constraint Logic Programs with Coroutining. In *Proc. of the Twelfth International Conference on Logic Programming (ICLP'95)*, pages 449–463. MIT Press, 1995.

[21] 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.

[22] C. Schulte and G. Smolka. Encapsulated Search for Higher-Order Concurrent Constraint Programming. In *Proc. of the Int'l Logic Programming Symposium (ILPS'94)*, pages 505–520. MIT Press, 1994.

[23] P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[24] 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.

[25] 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.