Operational Semantics for Declarative Multi-Paradigm Languages ⋆

Elvira Albert

DSIP, Complutense University of Madrid, E-28040 Madrid, Spain

Michael Hanus * Frank Huch

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany

Javier Oliver Germán Vidal

DSIC, Technical University of Valencia, E-46022 Valencia, Spain

Abstract

Declarative multi-paradigm languages combine the most important features of functional, logic and concurrent programming. The computational model of such integrated languages is usually based on a combination of two different operational principles: narrowing and residuation. This work is motivated by the fact that a precise definition of an operational semantics including all aspects of modern multi-paradigm languages like laziness, sharing, non-determinism, equational constraints, external functions, concurrency, etc. does not exist. Therefore, in this article, we present the first rigorous operational description covering all the aforementioned features in a precise and understandable manner. We develop our operational semantics in several steps. First, we define a natural (big-step) semantics covering laziness, sharing and non-determinism. We also present an equivalent small-step semantics which additionally includes a number of practical features like equational constraints and external functions. Then, we introduce a deterministic version of the small-step semantics which makes the search strategy explicit; this is essential for profiling, tracing, debugging, etc. Finally, the deterministic semantics is extended in order to cover the concurrent facilities of modern declarative multi-paradigm languages. The developed semantics provides an appropriate foundation to model actual declarative multi-paradigm languages like Curry. The complete operational semantics has been implemented and used for various programming tools.

Key words: Functional logic programming, operational semantics

1 Introduction

Declarative multi-paradigm languages combine the most important features of functional programming (nested expressions, higher-order functions, efficient demand-driven computations, polymorphism), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent computations with inter-thread synchronization and communication on logical variables). The computational model of such integrated languages is usually based on a seamless combination of two different operational principles: narrowing and residuation (see Hanus (1994) for a survey). *Narrowing* (Slagle, 1974) allows the instantiation of variables in expressions and then applies reduction steps to the function calls of the instantiated expressions. This instantiation is usually computed by unifying a subterm of the expression with the left-hand side of some program rule. On the other hand, *residuation* (Aït-Kaci et al., 1987) is based on the idea of delaying function calls until they are ready for a deterministic evaluation. Residuation preserves the deterministic nature of functions and naturally supports concurrent computations by employing dynamic scheduling.

This work is motivated by the fact that there does not exist a precise definition of an operational semantics covering all aspects of modern declarative multi-paradigm languages. For instance, the report on the multi-paradigm language Curry (Hanus, 2003) contains a fairly precise operational semantics but covers sharing only informally. The operational semantics of the functional logic language Toy (López-Fraguas and Sánchez-Hernández, 1999) is based on narrowing and sharing but the formal definition is based on a narrowing calculus (González-Moreno et al., 1999) which does not include a particular pattern-matching strategy. However, the latter becomes important, e.g., if one wants to reason about costs of computations (see Antoy (2001) for a discussion about narrowing strategies and calculi). The definition of a precise operational semantics for these languages is not an easy task since one must cover many different notions like sharing, logical variables, search strategies, concurrency, etc., as well as the interactions among them.

⋆ A preliminary version of this article appeared in the Proceedings of WFLP'02 (Albert et al., 2002b) and WRS'02 (Albert et al., 2002a). This work has been partially supported by CICYT TIC 2001-2705-C03-01, by the Generalitat Valenciana under grants CTIDIA/2002/205 and GRUPOS03/025, by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054, by the MCYT under grants HA2001-0059 and HU2003-0003, and by the DFG under grant Ha 2457/1-2.
∗ Corresponding author.

Email addresses: `elvira@sip.ucm.es` (Elvira Albert), `mh@informatik.uni-kiel.de` (Michael Hanus), `fhu@informatik.uni-kiel.de` (Frank Huch), `fjoliver@dsic.upv.es` (Javier Oliver), `gvidal@dsic.upv.es` (Germán Vidal).

The definition of a rigorous operational semantics covering all aspects of actual multi-paradigm languages is a difficult but important task, not only for reasoning about programs and correctness of implementations but also for the development of implementation-oriented analyses and tools (like profilers, tracers, debuggers, partial evaluators, etc). Well-known semantics for functional programs, like the natural semantics of Launchbury (1993) and the small-step semantics of Sestoft (1997), are not appropriate for our purposes since they do not cover logical variables and non-determinism, two important features of multi-paradigm languages. Furthermore, the extension of these semantics is not trivial since logical variables are *values* in our context and, thus, several concepts have to be reformulated. Also, non-determinism introduces new challenges in order to model *search strategies*, a key ingredient of multi-paradigm languages which does not appear in functional languages.

On the other hand, current semantics for functional logic languages, like the small-step semantics of Hortalá-González and Ullán (2001), do not consider the combination of narrowing and residuation (the basis of the language Curry) and, moreover, do not provide a high-level description (like a natural semantics) which is more suitable for stating a variety of properties. In order to achieve these goals and overcome existing limitations, we develop our operational semantics in several steps:

(1) First, we introduce a *natural* semantics which defines the intended results by relating expressions to values. This "big-step" semantics accurately models sharing which is important not only to reason about the space behavior of programs (as in Launchbury, 1993) but also for the correctness of computed results in the presence of non-confluent function definitions (see González-Moreno et al., 1999).
(2) Then, we provide a more implementation-oriented semantics based on the definition of individual computation steps. This "small-step" semantics is the formal reference to reason about operational aspects of programs. We formally prove the equivalence between the small-step semantics and the previous natural semantics.
(3) In order to obtain a complete operational semantics of a practical multi-paradigm language, like Curry (Hanus, 2003) or Toy (López-Fraguas and Sánchez-Hernández, 1999), one has to add descriptions for solving equational constraints, higher-order features, and evaluating external functions. These extensions are orthogonal to the other operational aspects (sharing, laziness, non-determinism). For this purpose, we properly extend our small-step operational semantics in order to cover all these features in a precise and understandable manner. Therefore, we are able to deal with practical features like integer and floating point numbers, external functions (e.g., arithmetic operators), predefined constraints (unification), and higher-order functions. We also discuss how these extensions can be modeled in the previous natural semantics.

(4) 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., Albert et al., 2001; Albert and Vidal, 2002; Sansom and Peyton-Jones, 1997; Vidal, 2004). This is useful to formally quantify the improvements achieved by a concrete program optimization and to compare different search strategies. Note that this approach is not possible by considering a non-deterministic semantics since such a semantics cannot properly describe the computation paths associated to a particular search strategy.
(5) Finally, we consider the use of threads to model concurrent computations and extend the previous semantics accordingly. Thus, we obtain a complete semantics which supports all aspects of modern multi-paradigm languages.

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 (Hanus, 2003). This semantics has been implemented as an interpreter that can be used to test language extensions, to check program optimizations, or to derive programming tools by designing instrumented versions.

This article is organized as follows. In the next section, we introduce some foundations for understanding the subsequent development. Section 3 introduces a semantics for multi-paradigm functional logic programs in natural style. This is refined in Section 4 to a small-step semantics describing individual execution steps; the equivalence between both semantics is formally proved. The small-step semantics is then extended in Section 5 to cover the practical features of declarative multi-paradigm languages. Section 6 presents a deterministic version of the semantics and Section 7 adds concurrency so that the final semantics covers all the important features. In Section 8, we describe an implementation of our semantics. Section 9 includes a comparison to related work. Finally, Section 10 concludes and points out several directions for further research.

2 Foundations

In this section, we describe the kernel of a multi-paradigm functional logic language whose execution model combines lazy evaluation with non-determinism. This model has been introduced by Hanus (1997) without formalizing the sharing of common subterms.

In this paper, all programs are untyped. Although a sophisticated type system is a well-known feature of modern functional and functional logic languages, we do not consider types in programs for the following reasons. The main use of a type system is to detect inconsistencies in a program at compile time. In this case, they have no influence on the run-time behavior of a program (e.g., Damas and Milner, 1982; Wadler and Blott, 1989). Depending on the type system, it is also reasonable to include types at run time in order to reduce the search space of non-deterministic computations. This has been mainly studied for logic programs (e.g., Hanus, 1991; Huber and Varsek, 1987; Smolka, 1988), but most functional logic languages—like Curry (Hanus, 2003), Escher (Lloyd, 1999) and Toy (López-Fraguas and Sánchez-Hernández, 1999)—use types only at compile time. Therefore, we ignore types for the sake of simplicity since we are interested in specifying the run-time behavior of programs. Nevertheless, our framework can be extended to many-sorted or parametrically polymorphic programs in a straightforward way. The inclusion of run-time type information in order to reduce the search space is an orthogonal issue.

In our context, a program is a set of function definitions where 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 following rules:

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

where data constructors usually start with upper case letters and function application is denoted by juxtaposition. There are no limitations with respect to 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 example, the following function "`choose`" non-deterministically returns one of its two arguments:

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

A subtle question is the meaning of nested applications containing such functions, e.g., the set of possible values of "`double (choose 1 2)`" with respect to the definition "`double x = x + x`". Similarly to González-Moreno et al. (1999), we follow 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 (Launchbury, 1993). A main goal of this

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

where P denotes a program, D a function definition, p a pattern and $e \in \mathit{Exp}$ an arbitrary expression.

Fig. 1. Syntax for flat programs

work is to describe the combination of laziness, sharing, and non-determinism in a precise and understandable manner.

2.1 The Flat Language

In order to provide a simple operational description, we assume that source programs are translated into a "flat" form, which is a convenient standard representation for functional logic programs. The main advantage of the flat form is the explicit representation of the pattern matching strategy by the use of case expressions which is important for the operational reading. For instance, consider the function definition

```
or False False  =  False
or x     True   =  True
```

and a non-terminating function $\perp$. From this definition, the evaluation of a function call like `(or` $\perp$ `True)` is not obvious. For instance, Haskell (Peyton-Jones, 2003) does not terminate on this call since its strict left-to-right evaluation strategy causes the non-terminating evaluation of $\perp$. On the other hand, Curry (Hanus, 2003) returns the normal form `True` since it evaluates inductive arguments (here: the second argument) first.

Such different behaviors are made explicit by the use of case expressions in the flat language. Moreover, source programs can be easily translated into this flat form (see Hanus and Prehofer, 1999). Different narrowing strategies can be represented by translations into differently structured case expressions.

The syntax for flat programs is shown in Figure 1. 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 $Var = \{x, y, z, \ldots\}$, data constructors (e.g., a, b, c,...), function calls (e.g., f, g, h,...), 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 following form: [1]

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

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* only shows up when the argument e is a free variable: *case* suspends whereas *fcase* non-deterministically binds this variable to the pattern in a branch of the case expression. Let bindings are in principle not required for translating source programs but they are convenient to express sharing without the use of complex graph structures (see, e.g., Echahed and Janodet, 1998; Habel and Plump, 1996). Operationally, let bindings introduce new structures in memory that are updated after evaluation, which is essential for lazy computations.

As an example of the flat representation, we show the translation of functions "`and`" and "`choose`" into flat form:

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

Laziness 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 body of the function. In a case expression, the outermost symbol of the case argument is required. Therefore, the case argument should be evaluated to *head normal form* (Barendregt, 1984), i.e., a variable or an expression with a constructor at the outermost position. Consequently, our operational semantics will describe the evaluation of expressions only to head normal form. This is not a restriction since the evaluation to

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

normal form or the solving of equations can be reduced to head normal form computations (see Hanus and Prehofer, 1999). Similarly, the higher-order features of current functional languages can be reduced to first-order definitions (see below). 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 (Antoy and Hanus, 2000). Indeed, the flat representation of programs constitutes the kernel of modern declarative multi-paradigm languages like Curry (Hanus, 1997, 2003) or Toy (López-Fraguas and Sánchez-Hernández, 1999).

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 by flexible case expressions in right-hand sides. For instance, in Curry programs, they are usually introduced by a declaration of the form:

```
let x free in ...
```

As Antoy (2001) 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 similarly 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 direct circular let binding of the form "$let\ x = x\ in\ e$". Throughout this paper, we call such variables which are bound to themselves *logical variables*. For instance, an expression $\mathtt{x} + \mathtt{y}$ with logical variables x and y is represented as "$let\ \mathtt{x} = \mathtt{x}, \mathtt{y} = \mathtt{y}\ in\ \mathtt{x} + \mathtt{y}$". Our representation of logical variables does not exclude the use of other circular data structures, as in "$let\ \ \mathtt{x} = \mathtt{1} : \mathtt{x}\ \ in\ \ldots$". It is interesting to note that circular bindings are also used in implementations of Prolog to represent logical variables (Warren, 1983).

2.2 Additional Language Features

The multi-paradigm language Curry also includes a number of practical features which we describe in this section. In particular, Curry extends the optimal evaluation strategy of (Antoy et al., 2000) 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* (Giovannetti et al., 1991; Moreno-Navarro and Rodríguez-Artalejo, 1992). Operationally, an equational constraint of the form 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 anonymous function definitions by lambda abstractions. In our (first-order) flat representation, higher-order functions are translated into applications of an auxiliary function *apply* (Warren, 1982). This distinguished function can easily be defined by means of ordinary program rules (see the discussion in Section 5.3). 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 (Hanus and Prehofer, 1999). 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.

We 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`):

$$\begin{array}{lcll} \mathtt{conc(xs,ys)} & = & \mathit{fcase}\ \mathtt{xs}\ \mathit{of}\ \{\ \mathtt{[]} & \to \mathtt{ys};\\ & & \quad (\mathtt{z:zs}) & \to \mathtt{z:conc(zs,ys)}\ \} \end{array}$$

The use of a flexible case implies that `conc` acts as a flexible function which can be used to solve equations over functional expressions. For instance, 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:

$$\mathtt{prefix(p,xs)} \;=\; \mathit{let}\ \mathtt{s=s}\ \mathit{in}\ \mathtt{conc(p,s)\ =:=\ xs}$$

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`:

$$\begin{array}{lcll} \mathtt{satisfyAll(c,zs)} & = & \mathit{case}\ \mathtt{zs}\ \mathit{of}\ \{\ \mathtt{[]} & \to \mathtt{Success};\\ & & \quad (\mathtt{x:xs}) & \to \mathit{apply}(\mathtt{c,x})\\ & & & \quad\ \mathtt{\&\ satisfyAll(c,xs)}\ \} \end{array}$$

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):

$$\texttt{commonPrefix(p,xs)} \quad = \quad \texttt{satisfyAll(prefix(p),xs)}$$

For instance, the solutions for the constraint

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

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

3 A Natural Semantics

In this section, we introduce a natural (big-step) semantics for multi-paradigm functional logic programs which is in the midway between a (simple) denotational semantics and a (complex) operational semantics for a concrete abstract machine. Our semantics is non-deterministic and models sharing accurately. This is achieved by using the *let* construct which can be interpreted as a reference to subcomputations that are only evaluated when required. We illustrate the effect of sharing by means of an example.

Example 3.1 *Consider the following flat program:*

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

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

The definition of our semantics mainly follows the natural semantics defined by Launchbury (1993) 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 stages: a "normalization" process—which consists in converting the λ-calculus into a form where the creation and sharing of closures is made explicit—followed by the definition of a simple semantics at the level of closures. Similarly, we also describe our (first-order) semantics for functional logic programs in two separate phases. In the first phase, we apply a normalization process in order to ensure that the arguments of functions and constructors are always variables (not necessarily pairwise different). These variables will be interpreted as references to express sharing.

3.1 Normalization

In this section, we describe the normalization process for flat programs.

Definition 3.2 (normalization) *The normalization of an expression e flattens all the arguments of function (or constructor) calls by means of the mapping e^* which is defined inductively as follows:*

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

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. 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. In contrast to Launchbury (1993), our normalization process does not need to perform "α-conversion" (i.e., a renaming of bound variables using completely fresh variables) since our natural semantics already introduces fresh variable names for all bound variables, as we will see later.

Example 3.3 *Consider the program and goal of Example 3.1 again. Their normalization yields the program unchanged and the following goal:*

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

(VarCons) $\Gamma[x \mapsto t] : x \Downarrow \Gamma[x \mapsto t] : t$ where t is constructor-rooted

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

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

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

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

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

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

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

Fig. 2. Natural Semantics for Functional Logic Programs

3.2 Semantics of Normalized Programs

In the following, we assume that both the program and the expression to be evaluated have been normalized as in Definition 3.2.

The state transition semantics is defined in Figure 2. Our rules obey the following naming conventions:

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

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 Γ' with $\Gamma'[x] = e$ and $\Gamma'[y] = \Gamma[y]$ for all $x \neq y$. We use this notation either as a condition on a heap Γ or as a modification of Γ. In a heap Γ, a logical variable x is represented by a circular binding of the form $\Gamma[x] = x$. A *value* is a constructor-rooted term (i.e., a term whose outermost function symbol is a constructor symbol) or a logical variable (with respect to the associated heap).

We use judgements of the form "$\Gamma : e \Downarrow \Delta : v$" which are interpreted as "the expression e in the context of the heap Γ evaluates to the value v with the (possibly modified) heap Δ", according to the rules of Figure 2. We briefly explain the rules of our semantics:

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

(VarExp) This rule achieves the effect of sharing. If the variable to be evaluated is bound to some expression in the heap, then the expression is evaluated and the heap is updated with the computed value; finally, we return this value as the result. In contrast to Launchbury (1993), we do not remove the binding for the variable from the heap; this becomes useful to generate fresh variable names easily. Sestoft (1997) solves this problem by introducing a variant of Launchbury's relation which is labeled with the names of the already used variables. The only disadvantage of our approach is that *black holes* (a detectably self-dependent infinite loop) are not detected at the semantical level. However, this does not affect the natural semantics since black holes have no value.

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

(Fun) This rule corresponds to the unfolding of a function call. The result is obtained by reducing the right-hand side of the corresponding rule. We assume that the considered program P is a global parameter of the calculus.

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

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

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

(Guess) This rule corresponds to the evaluation of a flexible case expression whose argument reduces to a logical variable. 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, we update the heap with the (renamed) logical variables of the pattern.

A proof of a judgement corresponds to a derivation sequence using the rules of Figure 2. Given a normalized program P and a normalized expression e (to be evaluated), the *initial configuration* has the form "$[\,] : e$". If the judgement "$[\,] : e \Downarrow \Gamma : v$" holds, then the computed *answer* can be extracted from the final heap Γ by a simple process of *dereferencing* in order to obtain the

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

where `sub-proof` has the following form:

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

Fig. 3. Big-Step Semantics of Example 3.3

values associated to the logical variables of the initial expression e. If we try to construct a proof, then this may *fail* because of two different situations: there may be no finite proof that a reduction is valid—which corresponds to an infinite loop—or there may be no rule which applies in a (sub-part) of the proof. In the latter case, we have two possibilities: either rule Select is not applicable because there is no matching branch or rule Guess cannot be applied because a logical variable has been obtained as the argument of a rigid case expression. The natural semantics of Figure 2 does not distinguish between all the above failures. However, they will become observable in the small-step operational semantics.

Figure 3 illustrates the sharing behavior of the semantic description with one of the possible (non-deterministic) derivations for the program and expression of Example 3.3. Note that the heap in the final configuration, $[\mathtt{x2} \mapsto \mathtt{1}] : \mathtt{B0}$, does not contain bindings for the variable `x1` of the initial expression (due to the renaming of local variables in let expressions). This corresponds to the fact that the computed answer is the empty substitution.

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

Lemma 3.4 *If* $\Gamma : e \Downarrow \Delta : v$*, then either* v *is rooted by a constructor symbol or it is a logical variable in* Δ *(i.e.,* $\Delta[v] = v$*).*

Proof. It is an easy consequence of the fact that the non-recursive rules of the natural semantics (i.e., VarCons and Val) can only return a constructor-rooted term or a logical variable with respect to the associated heap. □

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. 4. Small-Step Semantics for Functional Logic Programs

4 A Small-Step Operational Semantics

From an operational point of view, an evaluation in the natural semantics builds a proof for "$[\,] : e_0 \Downarrow \Gamma : e_1$" in a bottom-up manner whereas a computation by using a small-step semantics builds a sequence of states (Sestoft, 1997). 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 rule VarExp, 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. Similarly to Sestoft (1997), the context is *extensible* (i.e., if Q' is a sub-proof of Q, then the context of Q' is an extension of the context of Q). Thus, the context is represented by a *stack*.

A configuration "$\Gamma : e$" of the big-step semantics 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 $\mathit{Heap} \times \mathit{Control} \times \mathit{Stack}$.

The complete small-step semantics is presented in Figure 4 which also shows the kind of elements stored in the stack. Formally, the stack is a list (the empty stack is denoted by $[\,]$) which contains two kinds of elements: *variables*, which are pushed on the stack when their values are required, and *case expressions* (abbreviated as $(f)\{\overline{p_k \to e_k}\}$, where the optional "$f$" indicates that the case expression is flexible), which are stored in the stack while their arguments are being evaluated to head normal form. We 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 that is bound to an expression e (which is not a value) proceeds by evaluating e and adding to the stack the reference to x. If a value v is eventually computed and there is a variable x on top of the stack, rule val updates the heap with $x \mapsto v$. In the big-step semantics, this situation corresponds to the application of rule VarExp.
- 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, rule select is used to select the appropriate branch and continue with the evaluation of this branch. If we reach a logical variable, 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 new binding for the logical variable.

In order to evaluate an expression e, we construct an *initial goal* of the form $([\,], e, [\,])$ and apply the rules of Figure 4. 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 (i.e., the computed *value*) and S is the empty stack. Similarly to the big-step semantics, the computed *answer* can easily be extracted from Γ by dereferencing the variables of the initial goal. The equivalence of the small-step semantics and the natural semantics is stated in the following theorem.

Theorem 4.1 *Let $e \in Exp$ be an expression and v a constructor-rooted term or a logical variable in heap Δ. Then,* $([\,], e, [\,]) \Longrightarrow^* (\Delta, v, [\,])$ *if and only if* $[\,] : e \Downarrow \Delta : v$.

In order to prove this theorem, we first need some auxiliary results. Our proof technique is an extension of the proof scheme of Sestoft (1997).

The following lemma shows that our small-step semantics can simulate derivations by the natural semantics.

Lemma 4.2 (completeness) *If* $\Gamma : e \Downarrow \Delta : v$ *then* $(\Gamma, e, S) \Longrightarrow^* (\Delta, v, S)$.

Proof. We prove it by induction on the structure of the derivation $\Gamma : e \Downarrow \Delta : v$. We distinguish the following cases:

(VarCons) Then, $\Gamma[x \mapsto t] : x \Downarrow \Gamma[x \mapsto t] : t$. Trivially,

$$(\Gamma[x \mapsto t], x, S) \Longrightarrow (\Gamma[x \mapsto t], t, S) \qquad \text{(by rule } \mathsf{varcons}\text{)}$$

(VarExp) We have $\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v$. Then, the following derivation holds:

$$\begin{array}{lll} & (\Gamma[x \mapsto e], x, S) & \\ \Longrightarrow & (\Gamma[x \mapsto e], e, x : S) & \text{(by rule } \mathsf{varexp}\text{)} \\ \Longrightarrow^* & (\Delta, v, x : S) & \text{(by premise and ind. hypothesis)} \\ \Longrightarrow & (\Delta[x \mapsto v], v, S) & \text{(by rule } \mathsf{val}\text{)} \end{array}$$

(Val) We have $\Gamma : v \Downarrow \Gamma : v$. In this case,

$$(\Gamma, v, S) \Longrightarrow^* (\Gamma, v, S) \qquad \text{(by considering an empty sequence)}$$

(Fun) We have $\Gamma : f(\overline{x_n}) \Downarrow \Delta : v$. Then, the following derivation holds:

$$\begin{array}{lll} & (\Gamma, f(\overline{x_n}), S) & \\ \Longrightarrow & (\Gamma, \rho(e), S) & \text{(by rule } \mathsf{fun}\text{)} \\ \Longrightarrow^* & (\Delta, v, S) & \text{(by premise and ind. hypothesis)} \end{array}$$

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

(Let) We have $\Gamma : let\ \{\overline{x_k = e_k}\}\ in\ e \Downarrow \Delta : v$. Now, the following derivation holds:

$$
\begin{array}{lll}
& (\Gamma, let\ \{\overline{x_k = e_k}\}\ in\ e, S) & \\
\Longrightarrow & (\Gamma[\overline{y_k \mapsto \rho(e_k)}], \rho(e), S) & \text{(by rule } \mathsf{let}\text{)} \\
\Longrightarrow^* & (\Delta, v, S) & \text{(by premise and ind. hypothesis)}
\end{array}
$$

with $\rho = \{\overline{x_k \mapsto y_k}\}$. Furthermore, we assume that $\overline{y_k}$ are the same fresh variables used in rule Let which is always possible since both derivations can use the same variables in corresponding steps.

(Or) We have $\Gamma : e_1\ or\ e_2 \Downarrow \Delta : v$. Then, the following derivation holds:

$$
\begin{array}{lll}
& (\Gamma, e_1\ or\ e_2, S) & \\
\Longrightarrow & (\Gamma, e_i, S) & \text{(by rule } \mathsf{or}, i \in \{1, 2\}\text{)} \\
\Longrightarrow^* & (\Delta, v, S) & \text{(by premise and ind. hypothesis)}
\end{array}
$$

Furthermore, we assume that e_i is the same argument selected in the premise of rule Or.

(Select) We have $\Gamma : (f)case\ e\ of\ \{\overline{p_k \to e_k}\} \Downarrow \Theta : v$. Then, the following derivation holds:

$$
\begin{array}{lll}
& (\Gamma, (f)case\ e\ of\ \{\overline{p_k \to e_k}\}, S) & \\
\Longrightarrow & (\Gamma, e, (f)\{\overline{p_k \to e_k}\} : S) & \text{(by rule } \mathsf{case}\text{)} \\
\Longrightarrow^* & (\Delta, c(\overline{y_n}), (f)\{\overline{p_k \to e_k}\} : S) & \text{(by left premise and ind. hyp.)} \\
\Longrightarrow & (\Delta, \rho(e_i), S) & \text{(by rule } \mathsf{select}\text{)} \\
\Longrightarrow^* & (\Theta, v, S) & \text{(by right premise and ind. hyp.)}
\end{array}
$$

where $p_i = c(\overline{x_n})$, and $\rho = \{\overline{x_n \mapsto y_n}\}$.

(Guess) We have $\Gamma : fcase\ e\ of\ \{\overline{p_k \to e_k}\} \Downarrow \Theta : v$. Then, the following derivation holds:

$$
\begin{array}{lll}
& (\Gamma, fcase\ e\ of\ \{\overline{p_k \to e_k}\}, S) & \\
\Longrightarrow & (\Gamma, e, f\{\overline{p_k \to e_k}\} : S) & \text{(by rule } \mathsf{case}\text{)} \\
\Longrightarrow^* & (\Delta, x, f\{\overline{p_k \to e_k}\} : S) & \text{(by left premise and ind. hyp.)} \\
\Longrightarrow & (\Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}], \rho(e_i), S) & \text{(by Lemma 3.4 and rule } \mathsf{guess}\text{)} \\
\Longrightarrow^* & (\Theta, v, S) & \text{(by right premise and ind. hyp.)}
\end{array}
$$

where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$ and $\overline{y_n}$ are the same fresh variables selected in rule Guess.

$\square$

In order to show the soundness of the small-step semantics, i.e., that it computes no more results than the natural (big-step) semantics, we introduce the concept of *balanced* computations.

Definition 4.3 (balanced computation)
A computation $(\Gamma, e, S) \Longrightarrow^* (\Delta, e', S)$ *is balanced if the initial and final stacks are the same and every intermediate stack extends the initial one.*

In particular, every successful computation $([], e, []) \Longrightarrow^* (\Gamma, v, [])$ is balanced.

Definition 4.4 (trace, balanced trace) *The trace of a computation is the sequence of transition rules used in the computation. A balanced trace is the trace of a balanced computation.*

There are several possibilities for a trace to be balanced. Clearly, the empty trace is balanced. Now, consider nonempty traces and an arbitrary initial stack S. Nonempty balanced traces must start with any of the following rules: varcons, varexp, fun, let, or, and case. The remaining rules cannot produce a nonempty balanced trace since they would produce an intermediate stack which does not extend the initial stack S.

A trace that begins with varcons can only contain this single transition, since it produces an intermediate stack S and an expression t which should be a constructor-rooted term. The only rules that could be applied are val and select, but both rules would remove an element from the stack which contradicts the balancedness of the trace.

If the trace begins with varexp, producing an intermediate stack of the form $x : S$, then rule val must be eventually applied in order to restore the initial stack to S. In this case, the derived expression is constructor-rooted and, thus, only rules val and select could be applied. However, since they would remove an element from the stack, this contradicts the balancedness of the computation; hence, the trace must have the form (varexp *bal* val), where *bal* stands for arbitrary balanced traces.

A trace that begins with fun is balanced whenever the subtrace after fun is balanced. Thus, it must have the form (fun *bal*). Similarly, the traces (let *bal*) and (or *bal*) are balanced.

If the trace begins with case, an intermediate stack of the form $(f)\{\overline{p_k \to e_k}\} : S$ is produced. The initial stack must be restored by applying either rule select or guess. Such balanced traces must have the form (case *bal* select *bal*) and (case *bal* guess *bal*), respectively.

In summary, all balanced traces can be derived from the grammar

$bal ::= \epsilon \mid \mathsf{varcons} \mid \mathsf{varexp}\ bal\ \mathsf{val}$

$\quad\quad \mid \mathsf{fun}\ bal \mid \mathsf{let}\ bal \mid \mathsf{or}\ bal$

$\quad\quad \mid \mathsf{case}\ bal\ \mathsf{select}\ bal \mid \mathsf{case}\ bal\ \mathsf{guess}\ bal$

where ϵ denotes the empty trace. Each balanced trace corresponds to one of the rules in the big-step semantics. The following lemma formalizes the proof of this statement.

Lemma 4.5 (soundness) *If* $(\Gamma_0, e_0, S) \Longrightarrow^* (\Gamma_1, v, S)$ *is balanced and* v *is constructor-rooted or a logical variable, then* $\Gamma_0 : e_0 \Downarrow \Gamma_1 : v$.

Proof. The proof is done by induction on the structure of balanced traces following the grammar above.

(ϵ) Then e_0 must be a constructor-rooted term or a logical variable. Thus, the proof follows by applying rule Val.

($\mathsf{varcons}$) Then $e_0 = x$ and $\Gamma_0 = \Gamma[x \mapsto t]$. Thus, (Γ_1, t, S) is the derived state, where $\Gamma_1 = \Gamma[x \mapsto t]$. Now, the proof follows by applying rule $\mathsf{VarCons}$.

($\mathsf{varexp}\ bal\ \mathsf{val}$) Then $e_0 = x$ and $\Gamma_0 = \Gamma[x \mapsto e]$ (where e is not constructor-rooted nor a logical variable). The state after applying rule varexp must be $(\Gamma[x \mapsto e], e, x : S)$, and the state before applying rule val must have the form $(\Delta, v, y : S')$. Since the trace between these states is balanced, we have $y = x$, $S' = S$, and $\Gamma[x \mapsto e] : e \Downarrow \Delta : v$ by the inductive hypothesis. The state after applying rule val must be $(\Delta[x \mapsto v], v, S)$, where $\Gamma_1 = \Delta[x \mapsto v]$. Therefore, using rule VarExp, we have $\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v$.

($\mathsf{fun}\ bal$) Then $e_0 = f(\overline{y_n})$, where $f(\overline{x_n}) = e \in P$ and $\rho = \{\overline{x_n \mapsto y_n}\}$. The state after applying rule fun must be $(\Gamma_0, \rho(e), S)$. Since $(\Gamma_0, \rho(e), S) \Longrightarrow^* (\Gamma_1, v, S)$ is balanced, we have $\Gamma_0 : \rho(e) \Downarrow \Gamma_1 : v$ by the inductive hypothesis. Then, by applying rule Fun, we obtain $\Gamma_0 : f(\overline{y_n}) \Downarrow \Gamma_1 : v$.

($\mathsf{let}\ bal$) Then $e_0 = let\ \{\overline{x_k = e_k}\}\ in\ e$, $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ are fresh variables. The state after applying rule let must be $(\Gamma_0[\overline{y_k \mapsto \rho(e_k)}], \rho(e), S)$. Since $(\Gamma_0[\overline{y_k \mapsto \rho(e_k)}], \rho(e), S) \Longrightarrow^* (\Gamma_1, v, S)$ is a balanced trace, we have $\Gamma_0[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \Downarrow \Gamma_1 : v$ by the inductive hypothesis. Applying rule Let to this judgement with the same renaming ρ, we obtain $\Gamma_0 : let\ \{\overline{x_k = e_k}\}\ in\ e \Downarrow \Gamma_1 : v$.

($\mathsf{or}\ bal$) Then $e_0 = e_1\ or\ e_2$. The state after applying rule or must be (Γ_0, e_i, S), with $i \in \{1, 2\}$. Since $(\Gamma_0, e_i, S) \Longrightarrow^* (\Gamma_1, v, S)$ is balanced, we have $\Gamma_0 : e_i \Downarrow \Gamma_1 : v$ by the inductive hypothesis. Then, the proof follows by applying rule Or, $\Gamma_0 : e_1\ or\ e_2 \Downarrow \Gamma_1 : v$ (selecting the same argument as in the application of rule or).

($\mathsf{case}\ bal\ \mathsf{select}\ bal$) Then $e_0 = (f)case\ e\ of\ \{\overline{p_k \mapsto e_k}\}$. The state after applying rule case must be $(\Gamma_0, e, (f)\{\overline{p_k \to e_k}\} : S)$, and the state before applying rule select must have the form $(\Delta, c(\overline{y_n}), (f)\{\overline{p_k \to e_k}\} : S)$. Since the trace between these states is balanced, we have $\Gamma_0 : e \Downarrow \Delta : c(\overline{y_n})$ by the inductive hypothesis. Now, the state after applying rule select must be $(\Delta, \rho(e_i), S)$, where $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$. Since the trace from $(\Delta, \rho(e_i), S)$ to (Γ_1, v, S) is also balanced, we have $\Delta : \rho(e_i) \Downarrow \Gamma_1 : v$ by the inductive hypothesis. Finally, the proof follows by applying rule Select, $\Gamma_0 : (f)case\ e\ of\ \{\overline{p_k \to e_k}\} \Downarrow \Gamma_1 : v$.

(case *bal* guess *bal*) Then $e_0 = fcase\ e\ of\ \{\overline{p_k \mapsto e_k}\}$. The state after applying rule case must be $(\Gamma_0, e, f\{\overline{p_k \to e_k}\} : S)$, and the state before applying rule guess must have the form $(\Delta[x \mapsto x], x, f\{\overline{p_k \to e_k}\} : S)$. Since the trace between these states is balanced, we have $\Gamma_0 : e \Downarrow \Delta[x \mapsto x] : x$ by the inductive hypothesis. Now, the state after applying rule guess must be $(\Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}], \rho(e_i), S)$, where $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are the same fresh variables selected in the application of rule guess. Since the trace from $(\Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}], \rho(e_i), S)$ to (Γ_1, v, S) is also balanced, we have $\Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}], \rho(e_i) \Downarrow \Gamma_1 : v$ by the inductive hypothesis. Finally, the proof follows by applying rule Guess, $\Gamma_0 : fcase\ e\ of\ \{\overline{p_k \to e_k}\} \Downarrow \Gamma_1 : v$.

□

Now, we can proceed with the proof of Theorem 4.1.

Proof. The "if" part follows directly from Lemma 4.2. The "only if" part is a consequence of Lemma 4.5 and the fact that any computation of the form $([\,], e, [\,]) \Longrightarrow^* (\Delta, v, [\,])$ is balanced. □

5 Language Extensions

So far, we described an operational semantics for the kernel of a multi-paradigm functional logic language. In this section, we extend the small-step operational semantics in order to cover typical extensions of modern multi-paradigm languages like integer and floating point numbers, external functions, predefined constraints (unification), and higher-order functions. We also show how to extend the natural semantics to deal with these features.

5.1 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, one has to be careful when defining the meaning of equality. We interpret equational constraints as strict equalities as it is common practice in functional logic programming (Antoy et al., 2000; Giovannetti et al., 1991; Hanus, 2003; Moreno-Navarro and Rodríguez-Artalejo, 1992): 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 e_1 and e_2 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.[2] 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 an *hnf* context containing e_2 on the stack. This element is popped from 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

[2] In Haskell (Peyton-Jones, 2003) (and similarly in Curry), there exists a related predefined function "`seq`" to force the evaluation of an expression to a value. It is mainly used to improve performance by avoiding unneeded laziness (e.g., when defining strict arguments within some data type declarations). However, `seq` is different from *hnf* since `seq` is not defined on logical variables, i.e., it suspends if the first argument is a logical variable, whereas *hnf* does not suspend since a logical variable is a head normal form in our context.

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

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

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

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

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

Fig. 5. Small-Step Semantics of *prim_constrEq*

rule to provide sharing):

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

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. Then, 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 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:

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

Rule		Heap	Control	Stack
boolEq$_1$		Γ	$prim\_boolEq(x, y)$	S
	$\Longrightarrow$	Γ	$(x_1 \texttt{==} y_1 \texttt{\&\&} \ldots \texttt{\&\&} x_n \texttt{==} y_n)^*$	S
boolEq$_2$		Γ	$prim\_boolEq(x, y)$	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 $c \neq d$

Fig. 6. Small-Step Semantics of *prim_boolEq*

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* by the rules of Figure 5. In these rules, 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 constrEq$_1$ returns `Success` and updates the heap by binding x' to y'. In rule 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:

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

Rule constrEq$_3$ proceeds in a similar manner. Finally, if both arguments, x and y, are bound to the same constructor application, rule 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 constrEq$_2$ and constrEq$_3$. For instance, in rule constrEq$_2$, the occur check should 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 referenced by y (according to the current heap) without considering applications of defined functions (see Hanus, 2003, 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` or `False` if both terms are identical or different, respectively. 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, as defined in Figure 6. 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`.

5.2 External Functions

Every realistic programming language must support some functions that are not implemented in the same programming language. For instance, consider 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

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

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

$$x_1 + x_2 \;=\; \mathit{hnf}(x_1, \mathit{hnf}(x_2, \mathit{prim\_+}(x_1, x_2)))$$

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.

Often, one can assume that external functions are executed only if all arguments are evaluated to literals. Since there are a few exceptions to this rule, we adopt the following general scheme: given an external function f of arity n, we define it by the rule

$$f(x_1, \ldots, x_n) \;=\; \mathit{hnf}(x_{j_1}, \mathit{hnf}(x_{j_2}, \ldots, \mathit{hnf}(x_{j_m}, \mathit{prim\_f}(x_1, \ldots, x_n)) \ldots))$$

where the set $\{j_1, \ldots, j_m\}$ denotes the positions of the arguments whose evaluation is required by the primitive function $\mathit{prim\_f}$.

In this way, the definition of the rules for the primitive functions of a realistic language like Curry can be easily done.

5.3 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., Warren, 1982) 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 $\mathit{apply}(\mathit{apply}(f, a), b)$ where the definition of *apply* contains the following rules for the binary function f (this technique is used, e.g., in the implementation described by Antoy and Hanus, 2000):

$$\begin{array}{lcl} \mathit{apply}(f, x) & = & f(x) \\ \mathit{apply}(f(x), y) & = & f(x, y) \end{array}$$

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 the second argument of *apply* does not need to be evaluated to head normal form. Thus, we define *apply* by the following rule:

$$\mathit{apply}(x_1, x_2) \;=\; \mathit{hnf}(x_1, \mathit{prim\_apply}(x_1, x_2))$$

The small-step semantics is then extended as follows:

Rule		*Heap*	*Control*	*Stack*
apply		Γ	$\mathit{prim\_apply}(x, y)$	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):

$$\begin{array}{lcl} \texttt{f(x)} & = & \texttt{g(x)} \\ \texttt{g(x,y)} & = & \texttt{42} \\ \texttt{h} & = & \mathit{apply}\texttt{(}\mathit{apply}\texttt{(f,1),2)} \end{array}$$

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 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.

The natural semantics of Figure 2 can also be augmented in order to cope with the language extensions considered so far. In this section, we show the counterpart of the previous rules in the context of the big-step semantics.

5.4.1 Equality

To cover equality, we first need to define the semantics of function *hnf*:

$$(\mathsf{HNF}) \quad \frac{\Gamma : x_1 \Downarrow \Delta : v_1 \qquad \Delta : x_2 \Downarrow \Theta : v_2}{\Gamma : \mathit{hnf}(x_1, x_2) \Downarrow \Theta : v_2}$$

where v_i, $i \in \{1, 2\}$, is a constructor-rooted term or a variable y with $\Delta[y] = y$.

Now, analogously to Section 5.1, we can define the semantics of constraint equalities (*prim_constrEq*) and Boolean equalities (*prim_boolEq*) as follows. The semantics of *prim_constrEq* can be given by the following rules:

$$(\mathsf{ConstrEq}_1) \quad \Gamma : \mathit{prim\_constrEq}(x, y) \Downarrow \Gamma[x' \mapsto y'] : \texttt{Success}$$

$$\text{where } \Gamma^*(x) = x' \text{ and } \Gamma^*(y) = y'$$

$$(\mathsf{ConstrEq}_2) \quad \frac{\Gamma[x' \mapsto c(\overline{x_n}), \overline{x_n \mapsto x_n}] : (x_1 \texttt{=:=} y_1 \;\texttt{\&>} \ldots \texttt{\&>}\; x_n \texttt{=:=} y_n)^* \Downarrow \Delta : v}{\Gamma : \mathit{prim\_constrEq}(x, y) \Downarrow \Delta : v}$$

$$\text{where } \Gamma^*(x) = x',\ \Gamma^*(y) = c(\overline{y_n}),\ \text{and } \overline{x_n} \text{ are fresh}$$

$$(\mathsf{ConstrEq}_3) \quad \frac{\Gamma[y' \mapsto c(\overline{y_n}), \overline{y_n \mapsto y_n}] : (x_1 \texttt{=:=} y_1 \;\texttt{\&>} \ldots \texttt{\&>}\; x_n \texttt{=:=} y_n)^* \Downarrow \Delta : v}{\Gamma : \mathit{prim\_constrEq}(x, y) \Downarrow \Delta : v}$$

$$\text{where } \Gamma^*(x) = c(\overline{x_n}),\ \Gamma^*(y) = y',\ \text{and } \overline{y_n} \text{ are fresh}$$

$$(\mathsf{ConstrEq}_4) \quad \frac{\Gamma : (x_1 \texttt{=:=} y_1 \;\texttt{\&>} \ldots \texttt{\&>}\; x_n \texttt{=:=} y_n)^* \Downarrow \Delta : v}{\Gamma : \mathit{prim\_constrEq}(x, y) \Downarrow \Delta : v}$$

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

Similarly, we define the semantics of *prim_boolEq* by the following two rules:

$$(\mathsf{BoolEq}_1) \quad \frac{\Gamma : (x_1 \texttt{==} y_1 \;\texttt{\&\&} \ldots \texttt{\&\&}\; x_n \texttt{==} y_n)^* \Downarrow \Delta : v}{\Gamma : \mathit{prim\_boolEq}(x, y) \Downarrow \Delta : v}$$

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

$$(\mathsf{BoolEq}_2)\quad \Gamma : \mathit{prim\_boolEq}(x,y) \ \Downarrow\ \Gamma : \mathtt{False}$$
$$\text{where } \Gamma^*(x) = c(\ldots), \Gamma^*(y) = d(\ldots), \text{ and } c \neq d$$

5.4.2 External Functions

Here, we assume the same considerations of Section 5.2, i.e., each primitive function is defined in terms of function *hnf* and an associated external function. For instance, the big-step semantics of the external function *prim_+* is given by the following simple rule:

$$(\mathsf{Prim\_+})\quad \Gamma : \mathit{prim\_+}(x,y) \ \Downarrow\ \Gamma : l_1 +_{\mathcal{A}} l_2$$

where $\Gamma^*(x) = l_1$, $\Gamma^*(y) = l_2$, l_1, l_2 are literals, and $+_{\mathcal{A}}$ denotes the arithmetic sum.

5.4.3 Higher-Order Features

In this case, it suffices to provide the big-step semantics of the distinguished function *apply*:

$$(\mathsf{Apply})\quad \Gamma : \mathit{prim\_apply}(x,y) \ \Downarrow\ \Gamma : \varphi(\overline{x_k}, y)$$

where $\Gamma^*(x) = \varphi(\overline{x_k})$ and either φ is a constructor symbol or $\varphi(\overline{y_n}) = e \in P$ with $k < n$.

The equivalence between the extended big-step semantics and the corresponding small-step semantics can be proved as an easy extension of the proof of Theorem 4.1.

6 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

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

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

Fig. 7. Deterministic Small-Step Semantics

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 4. 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 shown in Figure 7. The only difference is that, in the deterministic version, 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 with respect to a search function "$\circ$" can be defined as the smallest relation $\longrightarrow \;\; \subseteq \;\; Goal^* \times Goal^*$ satisfying

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

The computation 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. Here, 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 is either a rigid case expression with a logical variable in the argument position or a primitive function applied to some logical variable (note that 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 is 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 which are not primitive function applications (i.e., $e \neq \mathit{prim\_f}(\overline{x_n})$), it is defined as follows:

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

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

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

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

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

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 fails when it is applied to different constructors:

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

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

$$\textsf{(Eval)}\ \frac{g \Longrightarrow G}{g\ G' \overset{COMP}{\longrightarrow} G \circ G'} \qquad \textsf{(Discard)}\ \frac{g \not\Longrightarrow}{g\ G' \overset{type(g)}{\longrightarrow} G'} \qquad \begin{array}{c}(g \in Goal \text{ and}\\ G, G' \in Goal^*)\end{array}$$

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 a *COMP* step as the following lemma states:[3]

Lemma 6.1 *If $g_0 \longrightarrow^* g\ G'$ and $g \not\Longrightarrow$, then $type(g) \neq COMP$.*

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

- e is a value. We distinguish the following cases:

(1) $e = c(\overline{x_n})$:
If $S = x : S'$, then val is applicable and $g \Longrightarrow G$.
If $S = (f)\{\overline{p_k \longrightarrow e_k}\} : S'$, then either rule select is applicable (if there is a branch for constructor c) or $type(g) = FAIL$.
If $S = hnf(x) : S'$, then rule hnf_2 is applicable and $g \Longrightarrow G$.
If $S = [\,]$, then no rule is applicable and $type(g) = SUCC$.

(2) $e = x$ and $\Gamma[x] \neq x$:
Then either rule varcons (if $\Gamma[x]$ is a value) or rule varexp is applicable.

(3) $e = x$ and $\Gamma[x] = x$:
If $S = x : S'$, then val is applicable and $g \Longrightarrow G$.
If $S = f\{\overline{p_k \longrightarrow e_k}\} : S'$, we can apply rule guess.
If $S = \{\overline{p_k \longrightarrow e_k}\} : S'$, then $type(g) = SUSP$.
If $S = hnf(y) : S'$, then rule hnf_2 is applicable and $g \Longrightarrow G$.
If $S = [\,]$, then no rule is applicable and $type(g) = SUCC$.

- $e = prim\_f(\overline{x_n})$. By our requirement on *primType* above, $primType(\Gamma, e, S)$ yields the type *COMP* exactly in the same cases where $\Longrightarrow$ has a successor.
- e is any other expression. Then, for all possible expressions, there exists an applicable rule independently of Γ and S.

□

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:

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

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

7 Adding Concurrency

Hitherto, our semantics only covers narrowing. Additionally, modern declarative multi-paradigm languages like Curry support residuation, a technique for a seamless integration of rigid—in most cases predefined—functions into non-deterministic, search-based implementations of functional logic languages. Our semantics already includes basic support for the integration of residuation: if a rigid case or a predefined function is applied to a logical variable, then $\Longrightarrow$ provides no successor, i.e., the goal suspends (the function *type* yields *SUSP*). So far, our semantics makes no real difference between *FAIL* and *SUSP*. By means of residuation, though, a computation may suspend until a logical variable is bound by another computation.

For the implementation of residuation, modern declarative multi-paradigm languages like Curry support concurrency. The combination of concurrency and residuation makes multi-threading with communication on shared logical variables possible. For concurrency, the simplest semantics is *interleaving* which is usually defined at the level of a small-step operational semantics. A definition of residuation and concurrency at the level of the big-step semantics would be possible as well. However, this would result in a mixture of the different kinds of non-determinism resulting from narrowing and concurrency. The first non-determinism has to be determined by some kind of search, while for the latter some kind of scheduling algorithm is usually chosen. Alternatively, one could determine one of these non-determinisms explicitly in the big-step semantics. For concurrency, this could be done by considering all possible results like in denotational semantics for concurrent languages (e.g., Debbabi and Bolignano, 1997) which is technically complex. Determining the don't-know non-determinism of narrowing at the level of the big-step semantics is also difficult and we solved it at the level of the deterministic small-step semantics. Hence, an integration of these concepts at the level of the deterministic small-step semantics seems to be appropriate and comprehensible.

Our deterministic semantics can be naturally extended 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 8), 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

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

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

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

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

Fig. 8. Concurrent Semantics for Multi-Paradigm Programs

with each other by means of variable bindings in this global heap.

In our concurrent operational semantics, the following possibilities for discarding 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*: we can non-deterministically select one thread and ignore the remaining ones. Indeterminism is similar to *don't-care* non-determinism in logic programming (Lloyd, 1987), e.g., literals in a goal are selected in a don't-care nondeterministic way; this contrasts with *don't-know* non-determinism which is used in the selection of program clauses, where all possibilities should be considered. 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 $\Longrightarrow$-step 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. There is no alternative successor for the application of rules Succ_2 and Deadlock.

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 (Janson and Haridi, 1991) and Oz (Schulte and Smolka, 1994) and could easily be specified in our framework, as well.

8 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 small-step semantics described in the preceding sections where each derivation rule is almost literally translated into program code.

The interpreter is written in Haskell. 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. 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 more than 22000 reductions per second on a 2.0 GHz Linux-PC (AMD Athlon XP 2600 with 256 KB cache). For logic programs involving search, more than 3200 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., Albert et al., 2002c; Hanus and Koj, 2001) due to an improved handling of variable sharing.

Our implementation can serve as a starting point to develop further tools like program optimizers based on partial evaluators, tracing tools, etc. Actually, this interpreter has been used to generate run-time information (redex trails) to trace computations at an adequate abstraction level (Braßel et al., 2004b).

9 Related Work

In the field of functional programming, Launchbury (1993) defined the first operational semantics for purely lazy functional languages which provides an accurate model for sharing. It is separated into two stages: the first stage is a static conversion of the λ-calculus into a form where the creation and sharing of closures is explicit; the semantics is then defined at the level of closures. Our natural semantics is defined in a similar manner, though our language is first-order and it has logical variables and non-determinism. Later, Sestoft (1997) developed an abstract machine for the λ-calculus with lazy evaluation starting from Launchbury's natural semantics, where lazy evaluation means non-strict evaluation with sharing of argument evaluation, i.e., call-by-need. Similarly, we have defined a small-step semantics for functional logic programs with sharing from the previous natural semantics. Our small-step semantics can be seen as an extension of Sestoft's abstract machine to consider also logical variables and non-determinism. Starting from Sestoft's semantics, Sansom and Peyton-Jones (1997) developed the first source-level profiler for a compiled, non-strict, higher-order, purely functional language capable of measuring time and space usage. One can extend our operational semantics with cost information in a similar way in order to develop a profiler for multi-paradigm functional logic programs, as done in Braßel et al. (2004a).

As for logic programming, Jones and Mycroft (1984) and Debray and Mishra (1988) propose 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. The same holds for Börger's descriptions of Prolog's operational semantics (e.g., Börger, 1990a,b) which consist of various small-step semantics for the different language constructs. On the other hand, Börger and Rosenzweig's description of the operational semantics of full Prolog (Börger and Rosenzweig, 1995a) is based on the use of *evolving algebras* (Gurevich, 2000). Intuitively, evolving algebras are abstract machines—hence, also known as Abstract State Machines (ASM)—used mainly for the formal specification of semantics in a rigorous mathematical framework. Its application to a functional logic context requires the definition of an abstract model of functional logic programs from which one can derive stepwise refinements of the model. The description has been carried out for an innermost narrowing semantics (Börger et al., 1994) but, as its authors state, the adaption to a lazy semantics would involve a more difficult kind of control and substantial modifications. We believe that, for lazy evaluation, this alternative approach could be used to derive a small-step semantics which is equivalent but lower-level than the small-step semantics considered in this paper, so that it could be used to prove the correctness of a Curry compiler. Indeed, this is the approach taken by Börger and Rosenzweig (1995b) in order to provide a mathematical analysis of the WAM for executing Prolog and a correctness proof for a general compilation scheme of Prolog. Finally, Podelski and Smolka (1995) 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 are not contained in their semantics. The latter aspects are also not covered by other semantic foundations of concurrent logic programming (e.g., Haridi et al., 1992; Saraswat et al., 1991; Smolka, 1994). Similarly, concurrent extensions of functional languages (e.g., Armstrong et al., 1996; Chakravarty et al., 1998; Peyton Jones et al., 1996;

Panangaden and Reppy, 1997) do not cover search and constraint solving.

As for functional logic programming, the report on the multi-paradigm language Curry (Hanus, 2003) contains a complete operational semantics but covers sharing only informally. The operational semantics of the functional logic language Toy (López-Fraguas and Sánchez-Hernández, 1999) is based on narrowing (with sharing) but the formal definition is based on a narrowing calculus (González-Moreno et al., 1999) which does not consider a particular pattern-matching strategy. However, the latter becomes important, e.g., if one wants to reason about costs of computations. The approach of Hortalá-González and Ullán (2001), the closest to our work, contains an operational semantics for a lazy narrowing strategy which considers sharing, non-deterministic functions, and allows partial applications in patterns. However, they do not consider the distinction between flexible and rigid case expressions, which is necessary for defining an operational semantics combining narrowing and residuation (as in Curry). Furthermore, we presented two characterizations of our operational semantics: a high-level description in natural style and a more detailed small-step semantics, and formally proved their equivalence. Finally, Echahed and Janodet (1998) and Habel and Plump (1996) present graph narrowing relations by extending graph rewriting with some form of unification. Graph narrowing requires a complex machinery to represent and manipulate graphs. Nevertheless, for the purpose of modeling sharing, our approach based on the use of let bindings is sufficient.

10 Conclusions and Future Work

To the best of our knowledge, this is the first attempt of a rigorous operational description for multi-paradigm functional logic languages including features like laziness, sharing, non-determinism, higher-order functions, equational constraints, external functions, and concurrency. We developed our semantics in a stepwise manner: starting from a simple natural semantics covering only laziness, sharing and non-determinism to a detailed operational semantics which is deterministic and models concurrent computations. The natural semantics, as well as its small-step version, can be useful to reason about programs, to prove the correctness of program transformations, to check the appropriateness of language implementations, etc. The deterministic semantics is especially important for the development of programming tools related to the operational aspects of a language, like profilers, debuggers and tracers. The complete semantics provides an appropriate foundation to model realistic multi-paradigm languages like Curry (Hanus, 2003).

In particular, we are currently working on the definition of the theoretical foundations for *tracing* lazy functional logic computations in Curry (Braßel et al., 2004b). For this purpose, we have defined a conservative extension of the small-step semantics defined in this article that outputs not only the computed value and bindings but also an appropriate data structure—a sort of *redex trail* (Wallace et al., 2001)—which can be used to trace computations at an adequate level of abstraction. This approach shows the usefulness and practicality of our operational semantics.

Furthermore, we have also enhanced the operational semantics with the computation of cost information (Braßel et al., 2004a). This is useful, e.g., for profiling (Albert and Vidal, 2002; Sansom and Peyton-Jones, 1997) and for formally checking the improvement achieved by program optimizations (Albert et al., 2001; Vidal, 2004).

For future work, we want to use this operational semantics for the formal development of further programming tools. In particular, it could be interesting to use it as a basis to develop optimization tools, e.g., partial evaluators (Albert et al., 2002c; Albert and Vidal, 2001), and to check or derive new implementations (like Sestoft, 1997) for Curry. From a more theoretical point of view, it might be interesting to formally prove the confluence of the concurrent semantics (up to variable renaming) for fair search strategies, like breadth-first search. Indeed, we conjecture that it is confluent since 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 a committed choice construct.

Acknowledgements

We gratefully acknowledge the anonymous referees as well as the participants of WFLP 2002 and WRS 2002 for many helpful comments and suggestions on preliminary versions of this paper. We also thank the anonymous referees of this journal for their useful remarks and suggestions, which helped to improve our paper.

References

Aït-Kaci, H., Lincoln, P., Nasr, R., 1987. Le Fun: Logic, equations, and Functions. In: Proc. of the 4th IEEE Int'l Symp. on Logic Programming. IEEE Press, New York, pp. 17–23.

Albert, E., Antoy, S., Vidal, G., 2001. 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 2000). Springer LNCS 2042, pp. 103–124.

Albert, E., Hanus, M., Huch, F., Oliver, J., Vidal, G., 2002a. An operational semantics for declarative multi-paradigm languages. In: Proc. of the Int'l Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002). Vol. 70(6) of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers.

Albert, E., Hanus, M., Huch, F., Oliver, J., Vidal, G., 2002b. Operational semantics for functional logic languages. In: Proc. of the Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2002). Vol. 76 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers.

Albert, E., Hanus, M., Vidal, G., 2002c. A practical partial evaluation scheme for multi-paradigm declarative languages. Journal of Functional and Logic Programming 2002 (1).

Albert, E., Vidal, G., 2001. The narrowing-driven approach to functional logic program specialization. New Generation Computing 20 (1), 3–26.

Albert, E., Vidal, G., 2002. Symbolic profiling of multi-paradigm declarative languages. In: Proc. of the Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR 2001). Springer LNCS 2372, pp. 148–167.

Antoy, S., 2001. Constructor-based conditional narrowing. In: Proc. of the 3rd Int'l ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001). ACM Press, pp. 199–206.

Antoy, S., Echahed, R., Hanus, M., 2000. A needed narrowing strategy. Journal of the ACM 47 (4), 776–822.

Antoy, S., Hanus, M., 2000. Compiling multi-paradigm declarative programs into Prolog. In: Proc. of the Int'l Workshop on Frontiers of Combining Systems (FroCoS 2000). Springer LNCS 1794, pp. 171–185.

Armstrong, J., Williams, M., Wikstrom, C., Virding, R., 1996. Concurrent Programming in Erlang. Prentice Hall.

Barendregt, H., 1984. The Lambda Calculus—Its syntax and semantics. Elsevier.

Börger, E., 1990a. A logical operational semantics of full Prolog. Part I: Selection core and control. In: Proc. of the 3rd Int'l Workshop on Computer Science Logic (CSL'89). Springer LNCS 440, pp. 36–64.

Börger, E., 1990b. A logical operational semantics of full Prolog. Part II: Built-in predicates for database manipulations. In: Proc. of Mathematical Foundations of Computer Science. Springer LNCS 452, pp. 1–14.

Börger, E., López-Fraguas, F., Rodríguez-Artalejo, M., 1994. A model for mathematical analysis of functional logic languages and their implementations. In: Proc. of the IFIP 13th World Computer Congress (IFIP'94). IFIP-transactions A-51, pp. 410–415.

Börger, E., Rosenzweig, D., 1995a. A mathematical definition of full Prolog. Science of Computer Programming 24 (3), 249–286.

Börger, E., Rosenzweig, D., 1995b. The WAM—Definition and compiler correctness. In: Logic Programming: Formal Methods and Practical Applications. Elsevier Science Publishers, pp. 20–90.

Braßel, B., Hanus, M., Huch, F., Silva, J., Vidal, G., 2004a. Run-time profiling of functional logic programs. In: Pre-Proceedings of the Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2004).

Braßel, B., Hanus, M., Huch, F., Vidal, G., 2004b. A semantics for tracing declarative multi-paradigm programs. In: Proc. of the 9th ACM SIGPLAN Int'l Conference on Principles and Practice of Declarative Programming (PPDP'04). ACM Press.

Chakravarty, M., Guo, Y., Köhler, M., Lock, H., 1998. Goffin - Higher-order functions meet concurrent constraints. Science of Computer Programming 30 (1-2), 157–199.

Damas, L., Milner, R., 1982. Principal type-schemes for functional programs. In: Proc. of the 9th Annual Symposium on Principles of Programming Languages (POPL'82). ACM Press, pp. 207–212.

Debbabi, M., Bolignano, D., 1997. A semantic theory for ML higher-order concurrency primitives. In: Nielson, F. (Ed.), ML with Concurrency: Design, Analysis, Implementation and Application. Monographs in Computer Science. Springer, pp. 145–184.

Debray, S., Mishra, P., 1988. Denotational and operational semantics for Prolog. Journal of Logic Programming (5), 61–91.

Echahed, R., Janodet, J., 1998. Admissible graph rewriting and narrowing. In: Proc. of the 1998 Joint Int'l Conference and Symposium on Logic Programming (JICSLP'98). MIT Press, pp. 325–340.

Giovannetti, E., Levi, G., Moiso, C., Palamidessi, C., 1991. Kernel Leaf: A logic plus functional language. Journal of Computer and System Sciences 42, 363–377.

González-Moreno, J., Hortalá-González, M., López-Fraguas, F., Rodríguez-Artalejo, M., 1999. An approach to declarative programming based on a rewriting logic. Journal of Logic Programming 40, 47–87.

Gurevich, Y., 2000. Sequential abstract state machines capture sequential algorithms. ACM Transactions on Computational Logic 1 (1), 77–111.

Habel, A., Plump, D., 1996. Term graph narrowing. Mathematical Structures in Computer Science 6 (6), 649–676.

Hanus, M., 1991. Parametric order-sorted types in logic programming. In: Proc. of the TAPSOFT'91. Springer LNCS 494, pp. 181–200.

Hanus, M., 1994. The integration of functions into logic programming: From theory to practice. Journal of Logic Programming 19&20, 583–628.

Hanus, M., 1997. A unified computation model for functional and logic programming. In: Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97). ACM, New York, pp. 80–93.

Hanus, M., 2003. Curry: An integrated functional logic language. Available at: `http://www.informatik.uni-kiel.de/~curry/`.

Hanus, M., Koj, J., 2001. An integrated development environment for declarative multi-paradigm programming. In: Proc. of the Int'l Workshop on Logic Programming Environments (WLPE 2001). Paphos (Cyprus), pp. 1–14, also available from the Computing Research Repository (CoRR) at

http://arXiv.org/abs/cs.PL/0111039.
Hanus, M., Prehofer, C., 1999. Higher-order narrowing with definitional trees. Journal of Functional Programming 9 (1), 33–75.
Haridi, S., Janson, S., Palamidessi, C., 1992. Structural operational semantics for AKL. Journal of Future Generation Computer Systems 8, 409–421.
Hortalá-González, T., Ullán, E., 2001. 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). Springer LNCS 2024, pp. 216–232.
Huber, M., Varsek, I., 1987. Extended prolog with order-sorted resolution. In: Proc. of the 4th IEEE Int'l Symposium on Logic Programming. San Francisco, pp. 34–43.
Janson, S., Haridi, S., 1991. Programming paradigms of the Andorra Kernel Language. In: Proc. of the Int'l Logic Programming Symposium. MIT Press, pp. 167–183.
Jones, N., Mycroft, A., 1984. Stepwise development of operational and denotational semantics for Prolog. In: Proc. of the 2nd Int'l Conf. on Logic Programming (ICLP'84). pp. 281–288.
Launchbury, J., 1993. A natural semantics for lazy evaluation. In: Proc. of the ACM Symp. on Principles of Programming Languages (POPL'93). ACM Press, pp. 144–154.
Lloyd, J., 1987. Foundations of Logic Programming. Springer-Verlag, Berlin, second edition.
Lloyd, J., 1999. Programming in an integrated functional and logic language. Journal of Functional and Logic Programming (3), 1–49.
López-Fraguas, F., Sánchez-Hernández, J., 1999. TOY: A multiparadigm declarative system. In: Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99). Springer LNCS 1631, pp. 244–247.
Moreno-Navarro, J., Rodríguez-Artalejo, M., 1992. Logic programming with functions and predicates: The language Babel. Journal of Logic Programming 12 (3), 191–224.
Panangaden, P., Reppy, J., 1997. The essence of concurrent ML. In: Nielson, F. (Ed.), ML with Concurrency: Design, Analysis, Implementation, and Application. Monographs in Computer Science. Springer, pp. 5–29.
Peyton-Jones, S. (Ed.), 2003. Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press.
Peyton Jones, S., Gordon, A., Finne, S., 1996. Concurrent Haskell. In: Proc. of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96). ACM Press, pp. 295–308.
Podelski, A., Smolka, G., 1995. Operational semantics of constraint logic programs with coroutining. In: Proc. of the 12th Int'l Conference on Logic Programming (ICLP'95). MIT Press, pp. 449–463.
Sansom, P., Peyton-Jones, S., 1997. Formally based profiling for higher-order functional languages. ACM Transactions on Programming Languages and Systems 19 (2), 334–385.
Saraswat, V., Rinard, M., Panangaden, P., 1991. Semantic foundation of con-

current constraint programming. In: Proc. of 18th ACM Symp. on Principles of Programming Languages (POPL'91). ACM, New York, pp. 333–352.

Schulte, C., Smolka, G., 1994. Encapsulated search for higher-order concurrent constraint programming. In: Proc. of the 1994 Int'l Logic Programming Symposium. MIT Press, pp. 505–520.

Sestoft, P., 1997. Deriving a lazy abstract machine. Journal of Functional Programming 7 (3), 231–264.

Slagle, J., 1974. Automated theorem-proving for theories with simplifiers, commutativity and associativity. Journal of the ACM 21 (4), 622–642.

Smolka, G., 1988. Logic programming with polymorphically order-sorted types. In: Proc. of the 1st Int'l Workshop on Algebraic and Logic Programming. Springer LNCS 343, pp. 53–70.

Smolka, G., 1994. A foundation for higher-order concurrent constraint programming. In: Proc. of the 1st Int'l Conference on Constraints in Computational Logics. Springer LNCS 845, pp. 50–72.

Vidal, G., 2004. Cost-augmented partial evaluation of functional logic programs. Higher-Order and Symbolic Computation 17 (1-2), 7–46.

Wadler, P., Blott, S., 1989. How to make ad-hoc polymorphism less ad hoc. In: Proc. of the 16th ACM Symp. on Principles of Programming Languages (POPL'89). ACM Press, pp. 60–76.

Wallace, M., Chitil, O., Brehm, T., Runciman, C., 2001. Multiple-view tracing for Haskell: a new Hat. In: Proc. of the 2001 ACM SIGPLAN Haskell Workshop. Universiteit Utrecht UU-CS-2001-23.

Warren, D., 1983. An abstract Prolog instruction set. Technical note 309, SRI International, Stanford.

Warren, D. H. D., 1982. Higher-order extensions to Prolog – Are they needed? In: Hayes-Roth, M., Pao (Eds.), Machine Intelligence. Vol. 10. Ellis Horwood.