Electronic Notes in Theoretical Computer Science 76 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume76.html 19 pages

Operational Semantics for Functional Logic Languages ⋆

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

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

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

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

Abstract

In this work we provide a semantic description of functional logic languages covering notions like laziness, sharing, and non-determinism. Such a semantic description is essential, for instance, to have appropriate language definitions in order to reason about programs and check the correctness of implementations. First, we define a "big-step" semantics in natural style to relate expressions and their evaluated results. Since this semantics is not sufficient to reason about the operational aspects of programs, we also define a "small-step" operational semantics covering the main features of functional logic languages. Finally, we demonstrate the equivalence of the "small-step" semantics and the natural semantics.

1 Introduction

This work is motivated by the fact that there does not exist a precise definition of an operational semantics covering all aspects of modern functional logic languages, like laziness and pattern-matching, sharing, logical variables, and non-determinism. For instance, the report on the multi-paradigm language Curry [14] contains a fairly precise operational semantics but covers sharing only informally. The operational semantics of the functional logic language Toy [18] is based on narrowing and sharing but the formal definition is based

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

on a narrowing calculus [10] 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 [4] for a discussion about narrowing strategies and calculi).

In order to define an appropriate basis for reasoning about programs, correctness of implementations, optimizations, or costs of computations, we provide a semantic description covering the important aspects of current functional logic languages. For this purpose, we proceed in two steps. First, we introduce a *natural* semantics which defines the intended results by relating expressions to values. This "big-step" semantics accurately models sharing which is important not only to reason about the space behavior of programs (as in [17]) but also for the correctness of computed results in the presence of non-confluent function definitions [10]. Then, we provide a more implementation-oriented semantics based on the definition of individual computation steps. This final semantics is the formal reference to reason about operational aspects of programs (e.g., to develop appropriate debugging tools). It is also a basis to provide a comprehensive definition of Curry (in contrast to [12,14] which contain only partial definitions). Moreover, one can use it to prove the correctness of implementations by further refinements, as done in [20].

This work is organized as follows. In the next section we introduce some foundations for understanding the subsequent development. Section 3 introduces a semantics for functional logic programs in natural style. This is refined in Section 4 to a semantics describing individual execution steps and the equivalence between both semantics is proven. Section 5 includes a comparison to related work. Finally, Section 6 concludes and points out several directions for further research.

2 Foundations

In this section, we describe the kernel of a modern functional logic language whose execution model combines lazy evaluation with non-determinism and residuation. This model has been introduced in [12] without formalizing the sharing of common subterms. The accurate definition of the latter aspect is the purpose of the subsequent sections.

In this 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
```

(data constructors usually start with upper case letters and function application is denoted by juxtaposition). There are no limitations w.r.t. over-

lapping rules; in particular, one can also have non-confluent rules to define functions that yield more than one result for a given input (these are called *non-deterministic* or *set-valued functions*). For instance, the following function "`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)`" w.r.t. the definition "`double x = x + x`." Similarly to [10], 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 [17]. The goal of this work is to describe the combination of laziness, sharing, and non-determinism in a precise and understandable manner.

In order to provide an understandable 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. Moreover, source programs can easily be translated into this flat form [13]. The syntax for flat programs is as follows:

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

where P denotes a program, D a function definition, p a pattern and $e \in \mathit{Exp}$ an arbitrary expression. 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 $\mathit{Var} = \{x, y, z, \ldots\}$, 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 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* nondeterministically binds this variable to the pattern in a branch of the case expression and proceeds with the appropriate branch. Let bindings are in principle not required for translating source programs but they are convenient to express sharing without the use of complex graph structures (like, e.g., [9,11]). 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:

$$\texttt{and(x,y)} \quad = \quad \textit{case}\ \texttt{x}\ \textit{of}\ \{\ \texttt{True} \to \texttt{y};\ \ \texttt{False} \to \texttt{False}\ \}$$

$$\texttt{choose(x,y)} \quad = \quad \texttt{x}\ \textit{or}\ \texttt{y}$$

Laziness (or neededness) of computations will show up in the description of the behavior of function calls and case expressions. In a function call, parameters are not evaluated but directly passed to the body of the function. In a case expression, the form of the outermost symbol of the case argument is required; therefore, the case argument should be evaluated to *head normal form* (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 normal form or the solving of equations can be reduced to head normal form computations (see [13]). Similarly, the higher-order features of current functional languages can be reduced to first-order definitions by introducing an auxiliary "apply" function [22]. 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 [5]. Indeed, the flat representation for programs constitutes the kernel of modern declarative multi-paradigm languages like Curry [12,14] or Toy [18].

Extra variables are those variables in a rule which do not occur in the left-hand side. Such extra variables are intended to be instantiated by constraints in conditions or right-hand sides. For instance, they are usually introduced in Curry programs by a declaration of the form:

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

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

As Antoy [4] pointed out, the use of extra variables in a functional logic language causes no conceptual problem if these extra variables are renamed whenever a rule is applied. We will model this renaming similar to the renaming of local variables in let bindings. For this purpose, we assume that all extra variables x are explicitly introduced in flat programs by a 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 $\texttt{x} + \texttt{y}$ with logical variables `x` and `y` is represented as *let* $\texttt{x} = \texttt{x}, \texttt{y} = \texttt{y}$ *in* $\texttt{x} + \texttt{y}$. Our representation of logical variables does not exclude the use of other circular data structures, as in *let* $\texttt{x} = \texttt{1} : \texttt{x}$ *in* ... It is interesting to note that circular bindings are also used in implementations of Prolog to represent logical variables [23].

3 A Natural Semantics for Functional Logic Programs

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

Example 3.1 *Consider the following flat program:*

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

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

The definition of our semantics mainly follows the natural semantics defined by Launchbury [17] for the lazy evaluation of functional programs. In this (higher-order) functional semantics, the *let* construct is used for the creation and sharing of *closures* (i.e., functional objects created as the value of lambda expressions). The key idea in Launchbury's natural semantics is to describe the semantics in two 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 separated phases. In the first phase, we apply a normalization process in order to ensure that the arguments of functions and constructors are always variables. These variables will be interpreted as

references to express sharing and need not be pairwise different.

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:*

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

Here, 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 [17], our normalization process does not need to perform "α-conversion" (i.e., a renaming of bound variables in e using completely fresh variables) since our natural semantics already introduces fresh variable names for all bound variables in e, as we will explain in subsequent paragraphs.

For the definition of our semantics, we consider that both the program and the expression to be evaluated have been previously normalized as in Definition 3.2.

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

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

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

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

A *heap* is a partial mapping from variables to expressions (the *empty heap* is denoted by $[\,]$). The value associated to variable x in heap Γ is denoted by $\Gamma[x]$. $\Gamma[x \mapsto e]$ denotes a heap with $\Gamma[x] = e$, i.e., we 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 $\Gamma[x] = x$, i.e., x is not instantiated w.r.t. Γ. A *value* is a constructor-rooted term or a logical variable (w.r.t. the associated heap).

(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. 1. Natural Semantics for Functional Logic Programs

We use judgements of the form "$\Gamma : e \;\Downarrow\; \Delta : v$", which should be interpreted as "the expression e in the context of the heap Γ evaluates to the value v with the (possibly modified) heap Δ". Let us 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 [17], we do not remove the binding for the variable from the heap; this becomes useful to generate fresh variable names easily. [20] 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 the evaluation of a value, we return it without modifying the heap.

(Fun). This rule corresponds to the unfolding of a function call. The result is obtained by reducing the right-hand side of the corresponding rule. 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 1. Given a normalized program P and a normalized expression e (to be evaluated), the *initial configuration* has the form "$[\,] : e$." We say that a derivation is *successful* if it computes a value. The computed *answer* can be extracted from Γ by a simple process of *dereferencing* in order to obtain the values associated to the logical variables in 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 1 does not distinguish between all the above failures. However, they will become observable in the small-step operational semantics.

Figure 2 illustrates the sharing behavior of the semantic description with one of the possible (non-deterministic) derivations for the program and expression of Example 3.3. Note that the heap in the final configuration, $[\texttt{x2} \mapsto \texttt{1}] : \texttt{B0}$, 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.

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

where `sub-proof` has the following form:

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

Fig. 2. Big-Step Semantics of Example 3.3

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 w.r.t. the associated heap. □

4 A Small-Step Semantics

From an operational point of view, an evaluation in the natural semantics builds a *proof* for "$[\,] : e_0 \;\Downarrow\; \Gamma : e_1$" in a bottom-up manner, whereas a computation by using a small-step semantics builds a sequence of states [20]. 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 [20], the context is *extensible* (i.e., if P' is a sub-proof of P, then the context of P' is an extension of the context of P). Thus, the representation of the context is made 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

Heap × *Control* × *Stack*.

The complete small-step semantics is presented in Figure 3. Let us briefly describe the transition rules:

- Rule varcons is perfectly analogous to rule VarCons in the natural semantics.
- In rule varexp, the evaluation of a variable x which is bound to an expression e (which is not a value) proceeds by evaluating e and adding to the stack the reference to variable x. Here, the stack S is a list (the empty stack is denoted by $[\,]$). 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, then rule select is used to select the appropriate branch and continue with the evaluation of this branch. If we reach a logical variable, then rule guess is used to non-deterministically choose one alternative and continue with the evaluation of this branch; moreover, the heap is updated with the binding of the logical variable to the corresponding pattern.

In order to evaluate an expression e, we construct an *initial goal* of the form $([\,], e, [\,])$ and apply the rules of Figure 3. We denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$. A derivation $([\,], e, [\,]) \Longrightarrow^* (\Gamma, e', S)$ is *successful* if e' is in head normal form (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 $([\,], e, [\,]) \Longrightarrow^* (\Delta, v, [\,])$ *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 in [20].

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,

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

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

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. 3. Small-Step Semantics for Functional Logic Programs

derivation holds:

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

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

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

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

$$\begin{array}{lll} & (\Gamma, f(\overline{x_n}), S) & \\ \Longrightarrow & (\Gamma, \rho(e), S) & \text{(by rule fun, with } f(\overline{y_n}) = e \in P \text{ and } \rho = \{\overline{y_n \mapsto x_n}\}) \\ \Longrightarrow^* & (\Delta, v, S) & \text{(by premise and induc. hyp.)} \end{array}$$

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

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

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:

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

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:

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

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

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

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

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.

□

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, then 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

$$\begin{aligned} bal ::= \epsilon \mid & \ \mathsf{varcons} \mid \mathsf{varexp}\ bal\ \mathsf{val} \\ \mid & \ \mathsf{fun}\ bal \mid \mathsf{let}\ bal \mid \mathsf{or}\ bal \\ \mid & \ \mathsf{case}\ bal\ \mathsf{select}\ bal \mid \mathsf{case}\ bal\ \mathsf{guess}\ bal \end{aligned}$$

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* $(\Gamma_0, e_0, S) \implies^* (\Gamma_1, v, S)$ *is balanced, 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.

(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 VarCons.

(varexp *bal* 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 $\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v$.

(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 $(\Gamma_0, \rho(e), S) \implies^* (\Gamma_1, v, S)$ is balanced, we have $\Gamma_0 : \rho(e) \Downarrow \Gamma_1 : v$ by the inductive hypothesis. Then, by applying rule Fun, we obtain $\Gamma_0 : f(\overline{y_n}) \Downarrow \Gamma_1 : v$.

(let *bal*). Then $e_0 = \mathit{let}\ \{\overline{x_k = e_k}\}\ \mathit{in}\ e$, $\rho = \{\overline{x_k \mapsto y_k}\}$ 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 $(\Gamma_0[\overline{y_k \mapsto \rho(e_k)}], \rho(e), S) \implies^* (\Gamma_1, v, S)$ is a balanced trace, we have $\Gamma_0[y_k \mapsto \rho(e_k)] : \rho(e) \Downarrow \Gamma_1 : v$ by the inductive hypothesis. Ap-

plying rule Let to this judgement with the same renaming ρ, we obtain $\Gamma_0 : let\ \{\overline{x_k = e_k}\}\ in\ e \Downarrow \Gamma_1 : v$.

(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).

(case *bal* 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 Related Work

In the field of functional programming, Launchbury [17] 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 semantics is defined in a similar manner, though our language is first-order and it has logical variables and non-determinism. Later, Sestoft

[20] 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, Samsom and Peyton Jones [19] developed the first source-level profiler for a compiled, non-strict, higher-order, purely functional language capable of measuring time and space usage. We could extend our operational semantics with cost information in a similar way in order to develop a profiler for lazy functional logic programs. For this purpose, however, we would first need a *deterministic* version of the semantics which properly models search strategies. Otherwise, we could only compute the cost of *each single derivation* in the search tree, since some computation steps may be shared by more than one derivation. Thus, the definition of a deterministic version of the small-step semantics becomes essential (see [2]).

As for logic programming, [16] and [8] contain operational and denotational descriptions of Prolog with the main emphasis on specifying the backtracking strategy and the "cut" operator. However, laziness and sharing are not covered. The same holds for Börger's descriptions of Prolog's operational semantics (e.g., [6,7]) which consist of various small-step semantics for the different language constructs.

As for functional logic programming, the report on the multi-paradigm language Curry [14] contains a complete operational semantics but covers sharing only informally. The operational semantics of the functional logic language Toy [18] is based on narrowing (with sharing) but the formal definition is based on a narrowing calculus [10] 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 [15], 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, [9,11] 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.

6 Conclusions and Future Work

We presented an operational semantics for functional logic languages based on lazy evaluation with sharing and non-determinism. We developed our semantics in several steps. First, we transformed programs into a normalized form in order to make the pattern matching strategy, common subexpressions, etc. explicit. Then, we defined a natural semantics for these normalized programs covering laziness, sharing and non-determinism. Finally, we presented a corresponding small-step semantics and proved its equivalence with the natural semantics. To the best of our knowledge, this is the first attempt of a rigorous operational description for functional logic languages—including both flexible and rigid case expressions—based on lazy evaluation with sharing and non-determinism.

Our final semantics is an appropriate basis to define concrete functional logic languages. Nevertheless, in order to obtain a complete operational description of a practical language like Curry, one has to add descriptions for modeling search strategies and concurrency, for solving equational constraints, evaluating external functions and higher-order applications. These extensions are orthogonal to the other operational aspects (sharing, non-determinism) and they are the subject of ongoing research (see [2]). Indeed, we are working on the implementation of an interpreter for Curry—based on such an extended operational description—covering all the aforementioned features [2].

The complete operational description could be used, e.g., as a basis to define a cost-augmented semantics in the style of [1,3,19,21], to develop debugging and optimization tools (like partial evaluators), and to check or derive new implementations (like in [20]) for Curry.

References

[1] E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of the 10th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'00)*, pages 103–124. Springer LNCS 2042, 2001.

[2] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Lazy Functional Logic Programs. In *Proc. of Workshop on Reduction Strategies in Rewriting and Programming (WRS'02)*, pages 97–112, 2002.

[3] E. Albert and G. Vidal. Symbolic Profiling of Multi-Paradigm Declarative Languages. In *Proc. of Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'01)*, pages 148–167. Springer LNCS 2372, 2002.

[4] S. Antoy. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206. ACM Press, 2001.

[5] S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the Int'l Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.

[6] E. Börger. 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)*, pages 36–64. Springer LNCS 440, 1990.

[7] E. Börger. A logical operational semantics of full Prolog. Part II: Built-in predicates for database manipulations. In *Proc. of Mathematical Foundations of Computer Science (MFCS'90)*, pages 1–14. Springer LNCS 452, 1990.

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

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

[10] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An Approach to Declarative Programming based on a Rewriting Logic. *Journal of Logic Programming*, 40:47–87, 1999.

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

[12] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM, New York, 1997.

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

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

[15] T. Hortalá-González and E. Ullán. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. of the 5th Int'l Symp. on Functional and Logic Programming (FLOPS 2001)*, pages 216–232. Springer LNCS 2024, 2001.

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

[17] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of the ACM Symp. on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.

[18] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[19] P.M. Sansom and S.L. Peyton-Jones. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, 1997.

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

[21] G. Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 52–62. ACM Press, 2002.

[22] D. H. D. Warren. Higher-Order Extensions to Prolog – Are they needed? In Michie Hayes-Roth and Pao, editors, *Machine Intelligence*, volume 10. Ellis Horwood, 1982.

[23] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical note 309, SRI International, Stanford, 1983.