A Semantics for Tracing Declarative Multi-Paradigm Programs*

B. Brassel M. Hanus F. Huch
Institut für Informatik
CAU Kiel, Olshausenstr. 40
D-24098 Kiel, Germany
{bbr,mh,fhu}@informatik.uni-kiel.de

G. Vidal
DSIC, Technical University of Valencia
Camino de Vera s/n, E-46022 Valencia, Spain
Phone: +34.96.3877350 / Fax: +34.96.3877359
gvidal@dsic.upv.es

ABSTRACT

We introduce the theoretical basis for tracing lazy functional logic computations in a declarative multi-paradigm language like Curry. Tracing computations is a difficult task due to the subtleties of the underlying operational semantics which combines laziness and non-determinism. In this work, we define an instrumented operational semantics that generates not only the computed values and bindings but also an appropriate data structure—a sort of *redex trail*—which can be used to trace computations at an adequate level of abstraction. In contrast to previous approaches, which rely solely on a transformation to instrument source programs, the formal definition of a *tracing* semantics improves the understanding of the tracing process. Furthermore, it allows us to formally prove the correctness of the computed trail. A prototype implementation of a tracer based on this semantics demonstrates the usefulness of our approach.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.6 [**Programming Techniques**]: Logic Programming; D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms

Languages, Theory

Keywords

Functional Logic Programming, Tracing, Semantics

*This work has been partially supported by CICYT TIC 2001-2705, by Generalitat Valenciana CTIDIA/2002/205 and GRUPOS03/025, by EU-India Economic Cross Cultural Prog. ALA/95/23/2003/077-054, by MCYT HA2001-59 and HU2003-003, and by the DFG under grant Ha 2457/1-2.
In Proc. of the 6th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'04), pp. 179–190, Verona, Italy, August 24–26, 2004.

1. INTRODUCTION

Declarative languages offer many facilities for reasoning about the *extensional* properties of programs—mainly, to study the correctness of program analyses and transformations. However, *intensional* or operational properties are much harder to analyze. Features like laziness and non-determinism make modern functional logic languages (e.g., Curry [9, 11] and Toy [13]) more powerful but also operationally more complex. For instance, from the programmer's point of view, following the *actual* trace of a computation is almost useless when debugging Curry programs.

In lazy functional languages like Haskell [16], several works have advocated the construction of *declarative* traces that hide the details of lazy evaluation. The main such approaches are: Freja [15], which is based on the algorithmic debugging technique from logic programming [17], Hat [18], which enables the exploration of a computation backwards starting at the program output or error message, and Hood [7], which allows the programmer to observe the data structures at given program points. Recently, Hat has been improved in such a way that it covers all previous three approaches thanks to the construction of an extended trail [19]: the *augmented redex trail* (ART).

In general, these approaches to debugging are based on some program transformation. For instance, Hat's ART is defined (indirectly) through the transformation that enables its creation: the source program is first instrumented and, then, executed to create the trail. Therefore, it is not easy to understand how the ART of a computation should be constructed (e.g., by hand), it remains unclear which assumptions about the operational semantics are made and, most importantly, there are no correctness results for the transformation [5]. In contrast, providing a direct, semantics-based definition of ARTs would improve the understanding of the tracing process and allow to prove properties like "every reduction occurring in a computation can be observed in the trail". Chitil [5] presents a first approach to such a direct definition by presenting an augmented small-step operational semantics for a core of Haskell that generate ARTs. However, this work does not consider correctness issues.

In this paper, we share the aims of [5] but consider a lazy functional *logic* language. More precisely, we define an instrumented version of a small-step operational semantics for a core of Curry [1] that returns not only the computed values and bindings but also a trail of the computation (which is similar to an ART but also includes logical variables).

Similarly to [19], this trail can be used to perform tracing and algorithmic debugging, as well as to observe data structures through a computation. Furthermore, we state the correctness of the computed trail, which amounts to say that it contains all (and only) the reductions performed in a computation. To the best of our knowledge, this is the first correctness result for redex trails in the literature. The usefulness of our approach is tested by means of a prototype implementation of a Curry tracer based on the ideas presented in this work.

This paper is organized as follows. In the next section, we recall some foundations for understanding the subsequent developments. Section 3 introduces our model for tracing functional logic computations and illustrates it with some examples. Section 4 formalizes an instrumented semantics which builds the trail of a computation. The correctness of the instrumented semantics is then shown in Section 5. Section 6 describes an implementation of a Curry tracer based on the instrumented semantics. Section 7 includes a comparison to related work and, finally, Section 8 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. 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

```
and True  y = y
and False y = False
```

where data constructors usually start with upper case letters and function application is denoted by juxtaposition. There are no limitations w.r.t. overlapping rules; in particular, one can also have non-confluent rules to define functions that yield more than one result for a given input (these are called *non-deterministic operations*). For instance, the operation "`choose`" non-deterministically returns one of its arguments:

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

Similarly to [8], we follow the "call-time choice" semantics where all descendants of a subterm are reduced to the same value in a derivation, e.g., the expression

```
double (choose 1 2)
```

w.r.t. the definition

```
double x = x + x
```

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 [12].

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. Moreover, source programs can be easily (and automatically) translated into this flat form [10]. The syntax for flat programs is shown in Figure 1, where we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$.

Program			
P	::=	$D_1 \ldots D_m$	
Definition			
D	::=	$f(x_1, \ldots, x_n) = e$	
Expression			
e	::=	x	(variable)
	|	$c(e_1, \ldots, e_n)$	(constructor call)
	|	$f(e_1, \ldots, e_n)$	(function call)
	|	*case* e *of* $\{\overline{p_n \to e_n}\}$	(rigid case)
	|	*fcase* e *of* $\{\overline{p_n \to e_n}\}$	(flexible case)
	|	e_1 *or* e_2	(disjunction)
	|	*let* $\overline{x_n = e_n}$ *in* e	(let binding)
Pattern			
p	::=	$c(x_1, \ldots, x_n)$	

Figure 1: Syntax for flat programs

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 $e \in Exp$ composed by variables $\{x, y, z, \ldots\} \in Var$, data constructors (e.g., a, b, c,...), function calls (e.g., f, g, h,...), case expressions, disjunctions (e.g., to represent non-deterministic operations), 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 $e, e_1, \ldots, e_k$ 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 evaluates to a free variable: *case* suspends the computation 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 (like, e.g., [6]).

As an example of the flat representation, we show the translation of functions "`and`" and "`choose`" into flat form. Here and in the following examples, for the sake of readability, we omit some of the brackets and write function applications as in Curry:

```
and x y      = fcase x of { True  -> y;
                            False -> False }
choose x y   = x or 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 outermost symbol of the case argument is required. Therefore, the case argument should be evaluated to *head normal form* (i.e., a free variable or an expression with a constructor at the outermost position).

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 flexible case expressions. For

[1] We write $(f)case$ for either *fcase* or *case*.

instance, in Curry programs, they are usually introduced by a declaration of the form:

`let x free in` *expression*

As Antoy [2] 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, 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` *expression*

Throughout this paper, we call such variables which refer to themselves *logical variables*. For instance, an expression `x + y` with logical variables `x` and `y` is represented as `let x = x, y = y in x + y`.

The flat representation of programs constitutes the kernel of modern declarative multi-paradigm languages like Curry [9, 11] or Toy [13].

3. A MODEL FOR TRACING

In this section, we informally describe our tracing model for lazy functional logic programs; we postpone the formal definition of trails to the next section.

The original redex trail structure of [18] is a directed graph which records copies of all values and redexes (*red*ucible *ex*pressions) of a computation, with a backward link from each reduct (and each proper subexpression contained within it) to the parent redex that created it. The ART (Augmented Redex Trail) model mainly extends the original redex trails by also including forward links from every redex to its reduct (see [19] for a detailed description). Thanks to this extension, the same data structure can be used to perform—when defining an appropriate *viewer*—tracing [18] and algorithmic debugging [15] as well as to print Hood's observations [7].

In order to trace lazy functional *logic* programs, we consider a data structure which shares many similarities with ARTs but also includes a special treatment to record the binding of logical variables.

We illustrate our approach by the following flat program:

EXAMPLE 3.1.

```
selfAnd x   = and x x
and x y     = fcase x of { False -> False;
                           True  -> y }
choose x y = x or y
main        = selfAnd (choose False True)
```

From now on, we assume that computations always start from the distinguished function `main` which has no arguments. The execution of the program above produces (non-deterministically) two trails, one associated to the computed value `False` and another one associated to the computed value `True`. Figure 2 shows the redex trails corresponding to these two computations.

In general, our graphs contain three types of arrows:

- *Successor arrows:* There is a successor arrow, denoted by a solid arrow, from each redex to its reduct (e.g., from `main` to `selfAnd` in Fig. 2).

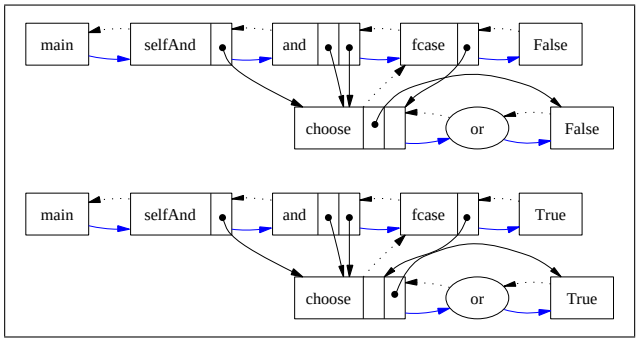

Figure 2: Redex trails for Example 3.1

- *Argument arrows:* Arguments of function and constructor calls are denoted by a pointer to the corresponding expression. Note that, when the evaluation of an argument is not required in a computation, we have a null pointer for that argument (e.g., the second argument of `choose` in the first graph of Fig. 2).

- *Parent arrows:* They are denoted by dotted arrows and point either to the redex from which it results (thus, the inverse of the corresponding successor arrow) like, e.g., from `selfAnd` to `main` in Fig. 2, or to the expression who *demanded* its evaluation, e.g., from `choose` to `fcase`.

Note that our notion of parent is different from that in [18, 19]. In these works, a parent arrow points to the redex that *introduced* a particular expression (i.e., every subexpression in the right-hand side of a rule points to the redex which called to this function). In contrast, we have a parent arrow from an expression to the expression that first *demanded* its evaluation.

From the computed trails, we developed a tool that navigates through them and allows the user to inspect the trace of any given computation. In particular, since the execution of `main` gives rise (non-deterministically) to either `False` and `True`, our tool initially shows

```
main -> False
     -> True
```

to point out that two results were computed. If the user selects the first result, `False`, then the following trace is shown:

```
False = main
False = selfAnd False
False = and     False False
False = False
```

This is the top-level trace of the execution of `main`. Each row shows a pair "*val* = *exp*" where *exp* is an expression and *val* is its value. Note that function arguments appear *fully evaluated*—i.e., as much as needed in the complete computation—in order to ease the understanding of the trace. Basically, the fully evaluated arguments are retrieved by following forwards both the argument and successor arrows. Every trace shows a computation from the final result (bottom line) to the initial function call (top line) following the parent arrows.

In our tracing tool, the user can also select any argument of an expression in order to see the associated trace. For

instance, if we select the first argument of "`and`", the tool shows the following trace:

```
False = and    False False
False = choose False _
False = False
```

Here, we denote by "`_`" an argument whose evaluation is not needed (i.e., a null pointer); in this particular case, because function `choose` returns the first argument and ignores the second one.

Consider now the following example where function `main` contains a logical variable:

EXAMPLE 3.2.

```
data Person = Christine | Antony | Monica
            | John | Susan | Peter

mother c = fcase c of { John  -> Christine;
                        Susan -> Monica;
                        Peter -> Monica }

father c = fcase c of { John  -> Antony;
                        Susan -> John;
                        Peter -> John }

parent x = (father x) or (mother x)

grandfather c = father (parent c)

main = let x = x in grandfather x
```

Here, the execution of function `main` produces six trails, associated with the following (non-deterministic) results:

```
main -> Fail
     -> Antony
     -> Antony
     -> Fail
     -> Fail
     -> Fail
```

Note that failing derivations are also shown, which is particularly useful for debugging.

Figure 3 shows the trail of the first result `Antony`. If the user selects it, the following top-level trace is shown:

```
Antony = main
Antony = grandfather x/Susan
Antony = father      John
Antony = Antony
```

The expression "`x/Susan`" means that the argument was originally a logical variable which was bound to `Susan` in the selected computation. Now, if we select the argument of `grandfather`, we get the following trace:

```
Antony = grandfather x/Susan
Antony = father      John
John   = parent      x/Susan
John   = father      x/Susan
Susan  = x/          Susan
```

From this trace, one can easily see that the logical variable was bound to `Susan` during the evaluation of function `father`, and then propagated backwards until the initial call to `grandfather`.

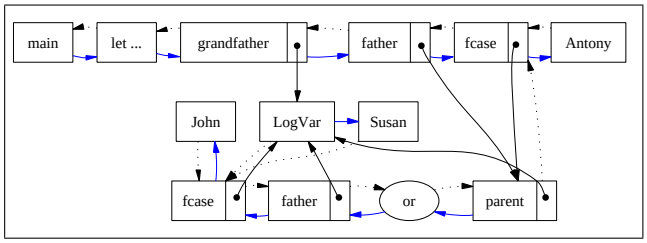

Figure 3: A redex trail for Example 3.2

4. A TRACING SEMANTICS

This section introduces an instrumented version of the small-step operational semantics of [1] which additionally builds the trail of a computation. In order to cope with sharing, the original small-step semantics requires programs to be *normalized*. Normalization ensures that the arguments of function and constructor calls are always variables (not necessarily pairwise different). These variables will be interpreted as references to express sharing.

DEFINITION 4.1 (NORMALIZATION [1]). *The normalization of an expression e flattens all the arguments of function (or constructor) calls by means of the mapping e^* which is inductively defined as follows:*

$$
\begin{array}{rcl}
x^* & = & x \\
\varphi(\overline{x_n})^* & = & \varphi(\overline{x_n}) \\
\varphi(\overline{x_{i-1}}, \overline{e_m})^* & = & \textit{let } x_i = e_1^* \textit{ in} \\
 & & \quad \varphi(\overline{x_i}, e_2, \ldots, e_m)^* \\
 & & \textit{where } e_1 \notin \mathit{Var},\ x_i \textit{ is fresh} \\
(\textit{let } \overline{x_k = e_k} \textit{ in } e)^* & = & \textit{let } \overline{x_k = e_k}^* \textit{ in } e^* \\
(e_1 \textit{ or } e_2)^* & = & e_1{}^* \textit{ or } e_2{}^* \\
((f)\textit{case } x \textit{ of } \{\overline{p_k \to e_k}\})^* & = & (f)\textit{case } x \textit{ of } \{\overline{p_k \mapsto e_k{}^*}\} \\
((f)\textit{case } e \textit{ of } \{\overline{p_k \to e_k}\})^* & = & \textit{let } x = e^* \textit{ in} \\
 & & \quad (f)\textit{case } x \textit{ of } \{\overline{p_k \mapsto e_k{}^*}\} \\
 & & \textit{where } e \notin \mathit{Var},\ x \textit{ is fresh}
\end{array}
$$

Here, φ denotes a constructor or function symbol. The extension of normalization to flat programs is straightforward, i.e., each rule $f(\overline{x_n}) = e$ is transformed into $f(\overline{x_n}) = e^*$.

For simplicity, the normalization mapping above 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 the following, we consider only *normalized flat programs*.

For instance, the normalization of function `main` in Example 3.1 is as follows:

```
main = let x = False, y = True, z = choose x y
       in  selfAnd z
```

The instrumented operational semantics is shown in Figure 4. It obeys 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

Rule	*Heap*	*Control*	*Stack*	*Graph*	*Ref.*	*Par.*
varcons	$\Gamma[x \mapsto t]$	x	S	G	r	p
	$\Longrightarrow \Gamma[x \mapsto t]$	t	S	$G \bowtie (x \rightsquigarrow r)$	r	p
varexp	$\Gamma[x \mapsto e]$	x	S	G	r	p
	$\Longrightarrow \Gamma[x \mapsto e]$	e	$x : S$	$G \bowtie (x \rightsquigarrow r)$	r	p
val	Γ	v	$x : S$	G	r	p
	$\Longrightarrow \Gamma[x \mapsto v]$	v	S	G	r	p
fun	Γ	$f(\overline{x_n})$	S	G	r	p
	$\Longrightarrow \Gamma$	$\rho(e)$	S	$G[r \underset{q}{\overset{p}{\mapsto}} f(\overline{x_n})]$	q	r
let	Γ	$let\ \overline{x_k = e_k}\ in\ e$	S	G	r	p
	$\Longrightarrow \Gamma[\overline{y_k \mapsto \rho(e_k)}]$	$\rho(e)$	S	$G[r \underset{q}{\overset{p}{\mapsto}} \rho(let\ \overline{x_k = e_k}\ in\ e)]$	q	r
or	Γ	$e_1\ or\ e_2$	S	G	r	p
	$\Longrightarrow \Gamma$	e_i	S	$G[r \underset{q}{\overset{p}{\mapsto}} e_1\ or\ e_2]$	q	r
case	Γ	$(f)case\ x\ of\ \{\overline{p_k \to e_k}\}$	S	G	r	p
	$\Longrightarrow \Gamma$	x	$((f)\{\overline{p_k \to e_k}\}, r) : S$	$G[r \overset{p}{\mapsto} (f)case\ x\ of\ \{\overline{p_k \to e_k}\}]$	q	r
select	Γ	$c(\overline{y_n})$	$((f)\{\overline{p_k \to e_k}\}, r') : S$	G	r	p
	$\Longrightarrow \Gamma$	$\rho(e_i)$	S	$G[r \overset{p}{\mapsto} c(\overline{y_n}), r' \underset{q}{\mapsto}]$	q	r'
guess	$\Gamma[y \mapsto y]$	y	$(f\{\overline{p_k \to e_k}\}, r') : S$	G	r	p
	$\Longrightarrow \Gamma[y \mapsto \rho(p_i), \overline{y_n \mapsto y_n}]$	$\rho(e_i)$	S	$G[r \underset{q}{\overset{p}{\mapsto}} LogVar, q \overset{r'}{\mapsto} \rho(p_i),$ $y \rightsquigarrow r, r' \underset{s}{\mapsto}]$	s	r'

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

Figure 4: Small-Step Tracing Semantics

Γ, a logical variable x is represented by a circular binding of the form $\Gamma[x] = x$. A *value* is a constructor-rooted term or a logical variable (w.r.t. the associated heap).

A *configuration* of the semantics is a tuple (Γ, e, S, G, r, p), where Γ is the current heap, e is the expression to be evaluated (often called the *control* of the small-step semantics), S is the stack (a list of variable names and case alternatives where the empty stack is denoted by `[]`) which represents the current context, G is a directed graph (the trail built so far), and r, p are references for the *current* and *parent* nodes of the expression in the control (see below).

For the time being, we ignore the last three columns (which are responsible of the construction of the trail) and focus on the *standard* component of the transition rules (columns *Heap*, *Control* and *Stack*):

(varcons) This rule is used to evaluate a variable x which is bound to a constructor-rooted term t in the heap. Trivially, it returns t as a result of the evaluation.

(varexp and val) In order to evaluate a variable x that is bound to an expression e (which is not a value), this rule starts a subcomputation for e and adds the variable x to the stack. If a value v is eventually computed and there is a variable x on top of the stack, rule val updates the heap with the binding $x \mapsto v$.

(fun) This rule performs a simple function unfolding (program P is a global parameter of the calculus).

(let) In order to reduce a let construct, this rule adds the bindings to the heap and proceeds 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.

(case, select, and guess) 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 (together with the current graph reference r, see below). If we reach a constructor-rooted term, then rule select is applied to select the appropriate branch and continue with the evaluation of this branch. If we reach a logical variable and the case expression on the stack is flexible (i.e., of the form $f\{\overline{p_k \to e_k}\}$), 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.

Now, we discuss the construction of the trail (columns *Graph*, *Ref.*, and *Par.*). Similarly to the ART model, our trail is a directed graph with nodes identified by references[2] that are labeled with expressions. We adopt the following conventions:

- We write $r \mapsto e$ to denote that the node with reference r is labeled with expression e.
- Successor arrows are denoted by $r \underset{q}{\mapsto}$ which means that node q is the successor of node r.
- Analogously, parent arrows are denoted by $r \overset{p}{\mapsto}$ which means that node p is the parent of node r.
- Often, we write $r \underset{q}{\overset{p}{\mapsto}} e$ to denote that node r is labeled with expression e, node p is the parent of r, and node q is the successor of r. Similarly, we also write $r \overset{p}{\mapsto} e$ when the successor node is yet unknown (e.g., in rule case) or if there is no successor (e.g., in rule select).
- Argument arrows are denoted by $x \rightsquigarrow r$ which means that variable x points to node r. This is safe in our context since only variables can appear as arguments of function and constructor calls. Hence, these arrows are also called *variable pointers*.

In general, given a configuration (Γ, e, S, G, r, p), G denotes the graph built so far (not yet including the current expression e), r represents a fresh reference to store the current expression e in the control (with some exceptions, see below), and p denotes the parent of r. The basic idea of the graph construction is to record the actual control at the actual reference in every step. A brief explanation for each rule of the semantics follows:

(varcons and varexp) These rules are used to perform a variable lookup in the heap. If one of these rules is applied, it means that the evaluation of variable x is needed in the computation and, thus, a variable pointer for x should be added to the current graph G if it does not yet contain such a pointer . For this purpose, we introduce function $\bowtie$ which is defined as follows:

$$G \bowtie (x \rightsquigarrow r) = \begin{cases} G[x \rightsquigarrow r] & \text{if } \nexists r'.\ (x \rightsquigarrow r') \in G \\ G & \text{otherwise} \end{cases}$$

Intuitively, function $\bowtie$ is used to take care of sharing: if the value of a given variable has already been demanded in the computation, no new variable pointer is added to the graph.

(val) This rule is only used to update the heap; therefore, the current graph is not modified.

(fun) Whenever this rule is applied, node r (the value in column *Ref.*) is added to the graph. The node is labeled with the function call $f(\overline{x_n})$ and has parent p (the value in column *Par.*) and successor q (a fresh reference). In the new configuration, r becomes the parent reference (*Par.*) and the fresh reference q represents the current reference (*Ref.*).

[2]The domain for references is not fixed. For instance, we can use natural numbers as references but more complex domains are also possible.

(let and or) They proceed in a similar way as the previous rule by introducing the node for either the (renamed) let expression or the disjunction.

(case) This rule initiates the evaluation of the case expression. Therefore, it adds node r to the graph which is labeled with the case expression. We set p as the parent of r but include no successor since it will not be known until the case argument is evaluated to head normal form. For this reason, reference r is also stored in the stack (together with the case alternatives) so that rules select and guess may eventually set the right successor for r.

(select) This rule adds node r to the graph which is labeled with the computed value $c(\overline{y_n})$. It sets p as the parent of r but includes no successor since values are fully evaluated. Reference r' (stored in the stack) is used to set the right successor for the case expression that initiated the subcomputation: the fresh reference q. Note that, in the derived configuration, we have r' as parent reference—the case expression—rather than r.

(guess) This rule proceeds in a similar way as the previous one. The main difference is that the computed value is a *logical variable*. Here, we add node r to the graph which is labeled with a special symbol $LogVar$, and whose successor is a new node q which is labeled with the selected binding for the logical variable.

Clearly, the instrumented semantics is a conservative extension of the original small-step semantics of [1], since the last three columns of the calculus impose no restriction on the application of the standard component of the semantics (columns *Heap*, *Control*, and *Stack*). This is a trivial but an essential property of our tracing semantics.

In order to perform computations, we construct an *initial configuration* and (non-deterministically) apply the rules of Figure 4.

Definition 4.2 (initial configuration).
An initial configuration has the form: $\langle$`[]`, `main`, `[]`, $G_\emptyset, r, \square\rangle$, *where* $G_\emptyset$ *denotes an empty graph,* r *is a reference and* $\square$ *denotes the null reference.*

In principle, *successful* derivations end with a configuration which contains a value (a constructor-rooted term or a logical variable) in the control and an empty stack. In this case, the computed *answer* can easily be extracted from the final heap by dereferencing the logical variables introduced through the computation. Unfortunately, in such a configuration, the computed value will not appear in the associated graph (since expressions are added to the graph *after* evaluation). On the other hand, we are interested not only in the trace of successful derivations but also in the trace of *failing* derivations, since this information may be relevant for debugging.

In order to overcome these drawbacks, we extend the semantics with four new rules, depicted in Figure 5. These rules proceed as follows:

(select-f) This rule is the counterpart of rule select when there is no matching branch in the case expression. In contrast to select, the control of the derived configuration contains the special symbol *Fail* (a constructor).

Rule	*Heap*	*Control*	*Stack*	*Graph*	*Ref.*	*Par.*
select-f	Γ	$c(\overline{y_n})$	$((f)\{\overline{p_k \to e_k}\}, r') : S$	G	r	p
	$\Longrightarrow \Gamma$	$Fail$	S	$G[r \overset{p}{\mapsto} c(\overline{y_n}), r' \underset{q}{\mapsto}]$	q	r'
guess-f	$\Gamma[y \mapsto y]$	y	$(\{\overline{p_k \to e_k}\}, r') : S$	G	r	p
	$\Longrightarrow \Gamma[y \mapsto y]$	$Fail$	S	$G[r \overset{p}{\mapsto} LogVar, y \rightsquigarrow r, r' \underset{s}{\mapsto}]$	s	r'
success-c	Γ	$c(\overline{x_n})$	[]	G	r	p
	$\Longrightarrow \Gamma$	$\diamond$	[]	$G[r \overset{p}{\mapsto} c(\overline{x_n})]$	$\square$	r
success-x	$\Gamma[x \mapsto x]$	x	[]	G	r	p
	$\Longrightarrow \Gamma[x \mapsto x]$	$\diamond$	[]	$G[r \overset{p}{\mapsto} LogVar, x \rightsquigarrow r]$	$\square$	r

where in select-f: $\nexists i \in \{1, \ldots, k\}$ such that $p_i = c(\overline{x_n})$

Figure 5: Extensions to the Tracing Semantics

(guess-f) It applies when we have a logical variable in the control and (the alternatives of) a *rigid* case expression on top of the stack. Similarly to the previous rule, we put the special symbol *Fail* in the control of the derived configuration. Note that we make no distinction between failures and *suspensions* since concurrency issues are not considered in this paper.

(success-c) and (success-x) They apply to configurations in which the stack is empty (i.e., there are no pending subcomputations) and the control contains a value. In either case, a new node r—labeled with the computed value—is added. In the derived configuration, we put the special symbol $\diamond$ into the control (with current reference $\square$) to denote that no further steps are possible.

Now, a final configuration is defined as follows:

Definition 4.3 (final configuration).
A final configuration has the form: $\langle \Delta, \diamond, [], G, \square, p \rangle$, where Δ is a heap, G is a graph and p is a reference.

We denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$. A derivation $C \Longrightarrow^* C'$ is *complete* if C is an initial configuration and C' is a final configuration. In general, derivations can be either complete or *incomplete*, e.g., if the user stops the derivation before it finishes (the only possibility when the derivation is infinite).

We illustrate the tracing semantics with a simple example. Consider the following program:

```
mother x = fcase x of { John  -> Christine;
                        Peter -> Monica }

father x = fcase x of { Peter -> John }

main = let x = x, y = father x in mother y
```

The complete computation with the rules of Figures 4 and 5 is shown in Figure 6. Here, we consider natural numbers as references (thus, the initial reference is 0). For clarity, each computation step is labeled with the applied rule and, in each configuration, `G` denotes the graph of the previous configuration. The computed trail is depicted in Figure 7.

Similarly to the original small-step semantics of [1], our tracing semantics is *non-deterministic*, i.e., it computes a different trail—a graph—for each non-deterministic computation from the initial configuration. In practice, however, it is more convenient to build a single graph that comprises all possible non-deterministic paths (see Section 6).

5. CORRECTNESS

In this section, we formally prove the correctness of our tracing semantics. Essentially, we prove the following basic properties for the instrumented computations:

1. For each "relevant" (see below) expression in a computation, the graph stores a node which is labeled with such an expression (Proposition 5.2).
2. Whenever an expression is reduced, we have the corresponding successor and parent arrows in the graph (Proposition 5.5).
3. For each variable whose value is *demanded* in a computation, we have an associated variable pointer in the graph (Proposition 5.8).
4. Whenever a case expression demands the evaluation of its argument, we have a parent arrow in the graph from such an argument to the case expression that demanded its evaluation (Propositions 5.9 and 5.10).

The following definition formalizes the notion of *relevant* configuration. Roughly speaking, a configuration is *relevant* when its control contains an expression that will be stored in the graph in the next computation step.

Definition 5.1 (relevant configuration).
A configuration $\langle \Delta, e, S, G, r, p \rangle$ is relevant iff one of the following conditions hold:

- *e is a value (w.r.t. Δ, i.e., a constructor call or a variable x with $\Delta[x] = x$) and the stack S is not headed by a variable, or*
- *e is a function call, a let expression, a disjunction, or a case expression.*

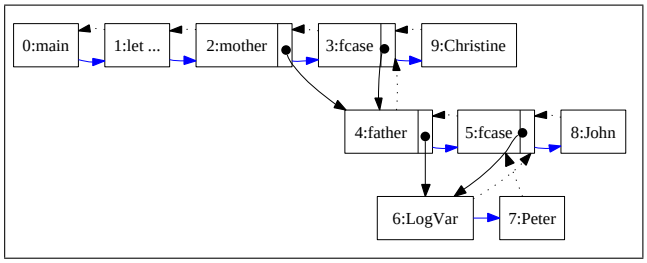

Figure 7: Trail of the computation in Figure 6

$\langle$[],main,[],$G_\emptyset$,0,$\Box\rangle$

$\Longrightarrow_{\mathsf{fun}}$ $\langle$[],let x=x,y=father x in mother y, [0 $\overset{\Box}{\underset{1}{\mapsto}}$ main],1,0$\rangle$

$\Longrightarrow_{\mathsf{let}}$ $\langle$[x $\mapsto$x,y$\mapsto$father x],mother y,[], G[1 $\overset{0}{\underset{2}{\mapsto}}$let x=x,y=father x in mother y],2,1$\rangle$

$\Longrightarrow_{\mathsf{fun}}$ $\langle$[x $\mapsto$x,y$\mapsto$father x],fcase y of {John $\rightarrow$ Christine, Peter $\rightarrow$ Monica} ,[], G[2 $\overset{1}{\underset{3}{\mapsto}}$ mother y],3,2$\rangle$

$\Longrightarrow_{\mathsf{case}}$ $\langle$[x $\mapsto$x,y$\mapsto$father x],y,[(f{John $\rightarrow$ Christine, Peter $\rightarrow$ Monica},3)],
G[3 $\overset{2}{\mapsto}$ fcase y of {John $\rightarrow$ Christine, Peter $\rightarrow$ Monica}],4,3$\rangle$

$\Longrightarrow_{\mathsf{varexp}}$ $\langle$[x $\mapsto$x,y$\mapsto$father x],father x,[y,(f{John $\rightarrow$ Christine, Peter $\rightarrow$ Monica},3)],G[y $\rightsquigarrow$ 4],4,3$\rangle$

$\Longrightarrow_{\mathsf{fun}}$ $\langle$[x $\mapsto$x,y$\mapsto$father x],fcase x of {Peter $\rightarrow$ John}, [y,(f{John $\rightarrow$ Christine, Peter $\rightarrow$ Monica},3)],
G[4 $\overset{3}{\underset{5}{\mapsto}}$ father x],5,4$\rangle$

$\Longrightarrow_{\mathsf{case}}$ $\langle$[x $\mapsto$x,y$\mapsto$father x],x, [(f{Peter $\rightarrow$ John},5),y,(f{John $\rightarrow$ Christine, Peter $\rightarrow$ Monica},3)],
G[5 $\overset{4}{\mapsto}$ fcase x of {Peter $\rightarrow$ John}],6,5$\rangle$

$\Longrightarrow_{\mathsf{guess}}$ $\langle$[x $\mapsto$Peter,y$\mapsto$father x],John, [y,(f{John $\rightarrow$ Christine, Peter $\rightarrow$ Monica},3)],
G[6 $\overset{5}{\underset{7}{\mapsto}}$ LogVar, 7 $\overset{5}{\mapsto}$ Peter, x $\rightsquigarrow$ 6, 5 $\underset{8}{\mapsto}$],8,5$\rangle$

$\Longrightarrow_{\mathsf{val}}$ $\langle$[x $\mapsto$Peter,y$\mapsto$John],John, [(f{John $\rightarrow$ Christine, Peter $\rightarrow$ Monica},3)],G,8,5$\rangle$

$\Longrightarrow_{\mathsf{select}}$ $\langle$[x $\mapsto$Peter,y$\mapsto$John],Christine, [], G[8 $\overset{5}{\mapsto}$ John, 3 $\underset{9}{\mapsto}$],9,3$\rangle$

$\Longrightarrow_{\mathsf{success-c}}$ $\langle$[x $\mapsto$Peter,y$\mapsto$John],$\diamond$, [], G[9 $\overset{3}{\mapsto}$ Christine],$\Box$,9$\rangle$

Figure 6: An Example of Tracing Computation

An expression is *relevant* if it appears in the control of a relevant configuration.

In the following, given a configuration $C = \langle \Delta, e, S, G, r, p\rangle$, we define $heap(C) = \Delta$, $control(C) = e$, $stack(C) = S$, and $graph(C) = G$. The following result states that, given a relevant configuration $\langle \Delta, e, S, G, r, p\rangle$, e will be added—in the next step—to G with reference r and parent p.

PROPOSITION 5.2. *Let $(C_0 \Longrightarrow^* C_n)$, $n > 0$, be a derivation. There is a relevant configuration $C_i = \langle \Delta, e, S, G, r, p\rangle$, $0 \leq i < n$, iff either $G_{i+1} = G[r \overset{p}{\mapsto} e]$ or $G_{i+1} = G[r \overset{p}{\mapsto} LogVar]$, where $G_{i+1} = graph(C_{i+1})$.*

PROOF. ($\Rightarrow$) We make a case distinction on expression e. If e is a function call, a let expression, a disjunction, or a case expression, then the claim follows by applying rule fun, let, or, or case, respectively.

If e is a constructor call, we consider two possibilities:

- If the stack is headed by a case construct, the claim follows by applying rule select or select-f.
- If the stack is empty, the claim follows by the application of rule success-c.

If e is a logical variable, i.e., $e = y$ with $\Delta[y] = y$, we consider the following two possibilities:

- If the stack is headed by a case construct, the claim follows by applying rule guess or guess-f.
- If the stack is empty, the claim follows by the application of rule success-x.

($\Leftarrow$) This direction is straightforward, since new $\mapsto$-arrows are only introduced in the current graph by the application of rules fun, let, or, case, select, select-f, guess, guess-f, success-c, or success-x, and in all these cases the considered configuration is relevant according to Definition 5.1. $\square$

Now, we introduce the notion of *successor derivation*. This notion will become useful to prove the correctness of successor arrows in the graph, i.e., property (2) above.

DEFINITION 5.3 (SUCCESSOR DERIVATION). *Let $\mathcal{D} : (C_0 \Longrightarrow^* C_n)$, $n > 0$, be a derivation. Let $C_i = \langle \Delta, e, S, G, r, p\rangle$, $0 \leq i < n$, be a relevant configuration such that e is a function call, a let expression, a disjunction, or a case expression. Then, $C_i \Longrightarrow^* C_j$, $i < j \leq n$, is a successor subderivation of $\mathcal{D}$ iff*

- *C_j is relevant,*
- *$stack(C_i) = stack(C_j)$, and*
- *there is no relevant configuration C_k, $i < k < j$, such that $stack(C_i) = stack(C_k)$.*

Intuitively, a successor derivation represents a single reduction step (including the associated subcomputations, if any). The following lemma states a useful property.

LEMMA 5.4. *Let $\mathcal{D} : (C_0 \Longrightarrow^* C_n)$, $n > 0$, be a derivation. There exists a successor subderivation*

$$C_i = \langle \Delta, e, S, G, r, p\rangle \Longrightarrow^* C_j = \langle \Delta', e', S', G', r', p'\rangle$$

$0 \leq i < j < n$, in $\mathcal{D}$, iff $(r \overset{p}{\underset{r'}{\mapsto}} e) \in G_j$ and $p' = r$.

PROOF. ($\Rightarrow$) We prove the claim by a case distinction on the expression e.

If $e = f(\overline{x_m})$ is a function call, rule fun is applied:

$$\langle \Delta, e, S, G, r, p\rangle \Longrightarrow \langle \Delta, \rho(e''), S, G[r \overset{p}{\underset{r'}{\mapsto}} e], r', r\rangle$$

where $f(\overline{y_m}) = e'' \in P$, $\rho = \{\overline{y_m \mapsto x_m}\}$, and r' is a fresh reference. If the new configuration is relevant (i.e., $C_{i+1} = C_j$), then the claim follows. Otherwise, only rules varcons, varexp, and val can be applied. Since these rules do not change the current and parent references, C_j is eventually reached and the claim follows.

If e is a let expression or a disjunction, then the proof is perfectly analogous to the previous case.

Finally, if $e = (f)case\ x\ of\ \{\overline{p_k \to e_k}\}$, rule case is applied:

$$\langle \Delta, (f)case\ x\ of\ \{\overline{p_k \to e_k}\}, S, G, r, p\rangle \\ \Longrightarrow\ \langle \Delta, x, ((f)\{\overline{p_k \to e_k}\}, r) : S, G^*, q, r\rangle$$

where $G^* = G[r \overset{p}{\mapsto} (f)case\ x\ of\ \{\overline{p_k \to e_k}\}]$ and q is a fresh reference. Now, the successor subderivation must contain a subcomputation of the form:

$$\langle \Delta, x, ((f)\{\overline{p_k \to e_k}\}, r) : S, G^*, q, r\rangle \\ \Longrightarrow^*\ \langle \Delta'', v, ((f)\{\overline{p_k \to e_k}\}, r) : S, G'', r'', p''\rangle$$

where v is a value (a constructor call or a logical variable). Then, by applying one of the rules select, select-f, guess, or guess-f, we get a new configuration of the form $\langle \Delta^*, e^*, S, G', r', r\rangle$ such that $(r \underset{r'}{\mapsto}) \in G'$. If the new configuration is already relevant, then the proof is done. Otherwise, only rules varcons, varexp, and val can be applied. Since these rules do not change the current and parent references, C_j is eventually reached and the claim follows.

($\Leftarrow$) We prove the claim by a case distinction on the expression e.

If $e = f(\overline{x_m})$ is a function call, there must be a (relevant) configuration $C_i = \langle \Delta, e, S, G, r, p\rangle$ with $0 < i < n$, such that the application of rule fun introduced $(r \overset{p}{\underset{r'}{\mapsto}} e)$ in G. If the derived configuration, $\langle \Delta, \rho(e'), S, G[r \overset{p}{\underset{r'}{\mapsto}} e], r', r\rangle$, is relevant, the claim follows since

$$C_i \Longrightarrow \langle \Delta, \rho(e'), S, G[r \overset{p}{\underset{r'}{\mapsto}} e], r', r\rangle$$

is a successor subderivation of $\mathcal{D}$. Otherwise, only rules varcons, varexp, and val can be applied. Since these rules do not change the current and parent references, C_j is eventually reached and the claim follows.

If e is a let expression or a disjunction, then the proof is perfectly analogous to the previous case.

Finally, if $e = (f)case\ x\ of\ \{\overline{p_k \to e_k}\}$, then there must be a (relevant) configuration of the form

$$C_i = \langle \Delta, e, S, G, r, p\rangle$$

with $0 < i < n$, such that the application of rule case introduced $(r \overset{p}{\mapsto} e)$ in G. Now, since $(r \underset{r'}{\mapsto})$ also belongs to G', we know that a subcomputation of the following form exists in $\mathcal{D}$:

$$\langle \Delta, x, ((f)\{\overline{p_k \to e_k}\}, r) : S, G^*, q, r\rangle \\ \Longrightarrow^*\ \langle \Delta'', v, ((f)\{\overline{p_k \to e_k}\}, r) : S, G'', r'', p''\rangle$$

where v is a value (a constructor call or a logical variable). Then, by applying one of the rules select, select-f, guess, or guess-f, we get a new configuration of the form $\langle \Delta^*, e^*, S, G', r', r\rangle$ such that $(r \underset{r'}{\mapsto}) \in G'$. If the derived configuration is relevant, the claim follows since

$$C_i \Longrightarrow^* \langle \Delta^*, e^*, S, G', r', r\rangle$$

is a successor subderivation of $\mathcal{D}$. Otherwise, only rules varcons, varexp, and val can be applied. Since these rules do not change the current and parent references, C_j is eventually reached and the claim follows. □

The following result states property (2) above: for each successor subderivation in a computation, the semantics of Figures 4 and 5 adds a corresponding successor arrow to the graph, and vice versa.

Proposition 5.5. *Let $\mathcal{D} : (C_0 \Longrightarrow^* C_n)$, $n > 0$, be a derivation. There exists a successor subderivation*

$$C_i = \langle \Delta, e, S, G, r, p\rangle \Longrightarrow^* \langle \Delta', e', S', G', r', p'\rangle = C_j$$

$0 \le i < j < n$, in $\mathcal{D}$, iff $(r \overset{p}{\underset{r'}{\mapsto}} e) \in graph(C_{j+1})$ and $(r' \overset{r}{\underset{q}{\mapsto}} e') \in graph(C_{j+1})$.

Proof. By Lemma 5.4, there is a successor subderivation

$$C_i = \langle \Delta, e, S, G, r, p\rangle \Longrightarrow^* \langle \Delta', e', S', G', r', p'\rangle = C_j$$

$0 \le i < j < n$, in $\mathcal{D}$, iff $(r \overset{p}{\underset{r'}{\mapsto}} e) \in G'$. Furthermore, by Proposition 5.2, $\langle \Delta', e', S', G', r', p'\rangle$ is a relevant configuration iff $(r' \overset{r}{\underset{q}{\mapsto}} e') \in graph(C_{j+1})$, for some reference q. The fact that no rule of the calculus removes information from the graph concludes the proof. □

So far, we have proved the correctness of successor arcs as well as some of the parent arcs (namely, those parent arcs which are the inverse of a successor arc). Now, we consider property (3) above: for each argument x of a function and constructor call, the graph includes a variable pointer $x \rightsquigarrow r$ that points to the reference of the corresponding argument *whenever this argument is needed in the computation.*

Definition 5.6 (lookup configuration).
Let $(C_0 \Longrightarrow^* C_n)$, $n > 0$, be a derivation. A configuration C_i, $0 \le i \le n$, is a lookup configuration of $\mathcal{D}$ iff $control(C_i) = x$ and there exists no configuration C_j with $control(C_j) = x$ and $j < i$.

When a lookup configuration $\langle \Delta, x, S, G, r, p\rangle$ appears in a computation, a variable pointer $x \rightsquigarrow r$ should be added to the graph. Moreover, reference r should store the *dereferenced value* of heap variable x. We express this by a function Δ^* which is defined as follows:

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

Note that $\Delta^*(x) = y$ implies that y is a logical variable (i.e., $\Delta[y] = y$).

The following auxiliary lemma states a useful property:

Lemma 5.7. *Let $\mathcal{D} : (C_0 \Longrightarrow^* C_n)$, $n > 0$, be a derivation. $C_i = \langle \Delta, x, S, G, r, p\rangle$, $0 < i < n$, is a lookup configuration in $\mathcal{D}$ iff $\nexists r'$ such that $(x \rightsquigarrow r') \in G$.*

Proof. The claim follows trivially by definition of lookup configuration. □

Now, we formally state property (3) above:

Proposition 5.8.
Let $\mathcal{D} : (C_0 \Longrightarrow^* C_n)$, $n > 0$, be a derivation. There exists a lookup configuration $C_i = \langle \Delta, x, S, G, r, p\rangle$, $0 < i < n$, in $\mathcal{D}$, iff $(x \rightsquigarrow r) \in graph(C_{j+1})$ and $r \overset{p}{\mapsto} \Delta^*(x) \in graph(C_{j+1})$, where C_j, $i \le j < n$, is a relevant configuration and there is no relevant configuration C_k with $i < k < j$.

Proof. ($\Rightarrow$) We consider the following possibilities.

If $\Delta[x] = x$ is a logical variable (and, thus, $\Delta^*(x) = LogVar$), then there exist two possibilities:

- If C_i is a relevant configuration (i.e., the current stack S is not headed by a variable), then $C_i = C_j$ and one of the following cases apply. If S is empty, the claim follows by applying rule success-x. If S is headed by a case expression, the claim follows by applying either rule guess or guess-f.
- Otherwise, there is a variable on top of the stack S. In this case, rule val applies. Since this rule does not change neither the control nor the current and parent references, the claim follows by applying rule val as much as needed until we get the relevant configuration C_j, since then the previous case applies.

If $\Delta[x] = c(\overline{x_m})$ is a constructor call (and, thus, $\Delta^*(x) = c(\overline{x_m})$), rule varcons applies:

$$C_i \Longrightarrow \langle \Delta, c(\overline{x_m}), S, G \bowtie (x \rightsquigarrow r), r, p\rangle$$

By Lemma 5.7, there is no reference r' such that $(x \rightarrow r') \in G$. Therefore, $G \bowtie (x \rightsquigarrow r) = G[x \rightsquigarrow r]$. If the derived configuration, C_{i+1} is relevant (i.e., $C_{i+1} = C_j$), then the claim follows by Proposition 5.2. Otherwise, only rule val can be applied. Similarly to the previous case, since this rule does not change neither the control nor the current and parent references, the claim follows by applying rule val as much as needed until we get the relevant configuration C_j, since then the claim follows by Proposition 5.2.

If $\Delta[x] = e$, where e is a function call, a let expression, a disjunction, or a case expression, then rule varexp applies:

$$C_i \Longrightarrow \langle \Delta, e, x : S, G \bowtie (x \rightsquigarrow r), r, p\rangle$$

By Lemma 5.7, $G \bowtie (x \rightsquigarrow r) = G[x \rightsquigarrow r]$. Finally, since the derived configuration C_{i+1} is relevant (i.e., $C_{i+1} = C_j$), the claim follows by Proposition 5.2.

Otherwise, $\Delta[x] = y$ with $x \neq y$. In this case, rule varexp applies:

$$C_i \Longrightarrow \langle \Delta, y, x : S, G \bowtie (x \rightsquigarrow r), r, p\rangle = C_{i+1}$$

By Lemma 5.7, $G \bowtie (x \rightsquigarrow r) = G[x \rightsquigarrow r]$. Now, we prove $(r \overset{p}{\mapsto} \Delta^*(y)) \in C_{j+1}$ by induction on the number l of (recursive) calls to $\Delta^*()$ performed by $\Delta^*(y)$ (i.e., on the length of the variable chain from y to $\Delta^*(y)$ in Δ):

($l = 1$) Then, $\Delta^*(y) = \Delta[y] = e$. If e is a logical variable (i.e., $e = y$), rule val applies:

$$C_{i+1} \Longrightarrow \langle \Delta[x \mapsto y], y, S, G[x \rightsquigarrow r], r, p\rangle = C_{i+2}$$

If C_{i+2} is a relevant configuration (i.e., $C_{i+2} = C_j$), the claim follows by Proposition 5.2. Otherwise, only rule val can be applied. Since this rule does not change neither the control nor the current and parent references, the claim follows by applying rule val as much as needed until we get the relevant configuration C_j, since then the claim follows by Proposition 5.2.

If $e = c(\overline{x_m})$ is a constructor call, rule varcons applies:

$$C_{i+1} \Longrightarrow \langle \Delta[x \mapsto y], c(\overline{x_m}), x : S, G[x \rightsquigarrow r] \bowtie (y \rightsquigarrow r), r, p\rangle = C_{i+2}$$

and, then, rule val:

$$C_{i+2} \Longrightarrow \langle \Delta[x \mapsto c(\overline{x_m})], c(\overline{x_m}), S, G[x \rightsquigarrow r] \bowtie (y \rightsquigarrow r), r, p\rangle = C_{i+3}$$

As in the previous case, if C_{i+3} is a relevant configuration (i.e., $C_{i+3} = C_j$), then the claim follows by Proposition 5.2. Otherwise, only rule val can be applied. Since this rule does not change neither the control nor the current and parent references, the claim follows by applying rule val as much as needed until we get the relevant configuration C_j, since then the claim follows by Proposition 5.2.

Finally, if e is a function call, a let expression, a disjunction, or a case expression, rule varexp applies:

$$C_{i+1} \Longrightarrow \langle \Delta, e, x : S, G[x \rightsquigarrow r], r, p\rangle = C_{i+2}$$

Since C_{i+2} is a relevant configuration (i.e., $C_{i+2} = C_j$), then the claim follows by Proposition 5.2.

($l > 1$) In this case, $\Delta^*(y) = \Delta^*(z)$ with $\Delta(y) = z$ and $y \neq z$. Therefore, rule varexp applies:

$$C_{i+1} \Longrightarrow \langle \Delta[x \mapsto y, y \mapsto z], z, y : x : S, G[x \rightsquigarrow r] \bowtie (y \rightsquigarrow r), r, p\rangle$$

Finally, since the number of (recursive) calls to $\Delta^*()$ in $\Delta^*(z)$ is strictly lesser than in $\Delta^*(y)$, the claim follows by induction.

($\Leftarrow$) This direction can be proved easily. If $(x \rightsquigarrow r) \in \mathit{graph}(C_{j+1})$, then there must be a configuration, C_i, of the form $\langle \Delta, x, S, G, r, p\rangle$, $0 < i < n$, in $\mathcal{D}$, such that there is no r' with $(x \rightsquigarrow r') \in G$. Therefore, C_i is a lookup configuration. Furthermore, since $(r \overset{p}{\mapsto} \Delta^*(x)) \in \mathit{graph}(C_{j+1})$, there must be a relevant configuration, C_j, of the form $\langle \Delta', \Delta^*(x), S', G', r, p\rangle$, and the claim follows. □

Finally, we consider the last property, i.e., we prove that, whenever a case expression demands the evaluation of its argument, a parent arrow is added to the graph from such an argument to the case expression that demanded its evaluation (and vice versa).

PROPOSITION 5.9. *Let $\mathcal{D} : (C_0 \Longrightarrow^* C_n)$, $n > 0$, be a derivation. There exists a configuration*

$$C_i = \langle \Delta, (f)\,\mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \rightarrow e_k}\}, S, G, r, p\rangle, \quad 0 < i < n$$

iff $(r \mapsto (f)\,\mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \rightarrow e_k}\}) \in \mathit{graph}(C_{i+1})$ and $(q \overset{r}{\mapsto} \Delta^*(x)) \in \mathit{graph}(C_{j+1})$, where C_j, $i < j < n$, is a relevant configuration and there is no relevant configuration C_k with $i < k < j$.

PROOF. By Proposition 5.2, there exists a (relevant) configuration

$$C_i = \langle \Delta, (f)\,\mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \rightarrow e_k}\}, S, G, r, p\rangle, \quad 0 < i < n$$

iff $(r \mapsto (f)\,\mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \rightarrow e_k}\}) \in \mathit{graph}(C_{i+1})$. Moreover, by applying rule case, we get the following reduction step:

$$C_i \Longrightarrow \langle \Delta, x, ((f)\{\overline{p_k \rightarrow e_k}\}, r) : S, G[r \overset{p}{\mapsto} (f)\,\mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \rightarrow e_k}\}, q, r\rangle = C_{i+1}$$

If C_{i+1} is a lookup configuration, then the claim follows by Proposition 5.8. Otherwise, only rules varcons, guess, guess-f, or success-x can be applied, and the claim follows straightforwardly (with $C_{i+1} = C_j$). □

There exists an additional situation in which a parent arrow is added to the graph: when a logical variable is bound by a flexible case expression. The next lemma proves the correctness of such a case:

PROPOSITION 5.10. *Let* $\mathcal{D} : (C_0 \Longrightarrow^* C_n)$, $n > 0$, *be a derivation. There exists a configuration* $C_i = \langle \Delta[y \mapsto y], y, (f\{\overline{p_k \to e_k}\}, r') : S, G, r, p\rangle$, $0 < i < n$, *iff*

- $(r' \mapsto \mathit{fcase}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}) \in G$,
- $(r \underset{q}{\mapsto} \mathit{LogVar}) \in \mathit{graph}(C_{i+1})$, *and*
- $(q \overset{r'}{\mapsto} \rho(p_i)) \in \mathit{graph}(C_{i+1})$.

where $i \in \{1, \ldots, k\}$, $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \to y_n}\}$, *and* $\overline{y_n}$ *are fresh.*

PROOF. In order to have an element $(f\{\overline{p_k \to e_k}\}, r')$ on top of the stack, a configuration with $\mathit{fcase}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}$ in the control should be previously reduced. By Proposition 5.2, there exists a configuration

$$C_j = \langle \Gamma, \mathit{fcase}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}, S, G', r', p'\rangle$$

in $\mathcal{D}$, with $0 < j < i$, iff $(r' \mapsto \mathit{fcase}\ x\ \mathit{of}\ \{p_k \to e_k\}) \in G$. Now, the claim follows trivially by applying rule guess:

$$C_i \Longrightarrow \langle \Delta[y \mapsto \rho(p_i), \overline{y_n \mapsto y_n}], \rho(e_i), S, G[r \underset{q}{\mapsto} \mathit{LogVar}, q \overset{r'}{\mapsto} \rho(p_i)], s, r'\rangle = C_{i+1}$$

□

6. IMPLEMENTATION ISSUES

We have prototypically implemented the presented graph construction by extending an interpreter for flat programs. In contrast to the semantics presented in this paper, our interpreter uses a fixed search strategy (e.g., depth-first search or breadth-first search) to deal with non-determinism similarly to [1]. As a consequence, we do not generate a graph for each non-deterministic computation. Instead, we generate a single graph containing the graphs of all evaluations executed by the search strategy, in the following called the *unified graph*.

In order to distinguish the different evaluations in the unified graph, we add *path* information to every node. Initially, the computation starts with the empty path. Whenever a branching is performed by rules or and guess, the subsequent computations are distinguished by extended paths. As an example, consider the following flat program which is similar to Example 3.1 but without sharing:

EXAMPLE 6.1.

```
and x y    = fcase x of { False -> False;
                          True  -> y }
choose x y = x or y
main       = and (choose False True)
                 (choose False True)
```

The unified graph for this example is presented in Figure 8. If a node is computed on a non-empty path, then the path (a list of numbers) is added to the label. The paths grow to the left; hence, the paths `[1,2]` and `[2,2]` are extensions of path `[2]`. This unified graph represents three graphs, related to the paths `[1]`, `[1,2]`, and `[2,2]`. These three graphs can be computed from the unified graph by considering only nodes with a path suffix. For instance, the node labeled with `main` belongs to all graphs, the node labeled with `[2]:choose` belongs to the graphs `[1,2]` and `[2,2]` and the node labeled with `[1,2]:False` only belongs to the graph `[1,2]`.

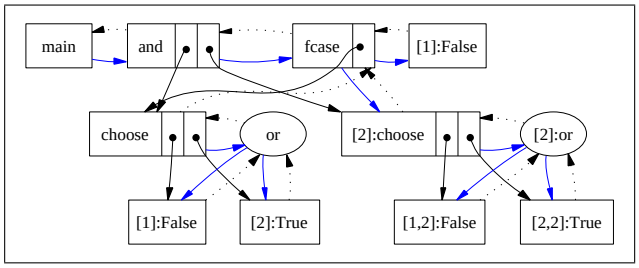

Figure 8: Unified Graph of Example 6.1

Generating a unified graph instead of a separate graph for each computation has two advantages. Firstly, large parts of the different graphs are identical (e.g., all nodes labeled with the empty path belong to all graphs). Hence, storing a unified graph needs less space. Secondly, in the viewer tool, it is not sufficient to present only a single graph to detect errors related to non-determinism. Rather, different results of a computation have to be presented to the programmer. Furthermore, the information about structures that are identical for two non-deterministic branches can be of great help for debugging, too. It is much easier to obtain these results in the unified graph.

We have also prototypically implemented viewing tools for the constructed redex trails. One of these is the text-based view which was briefly sketched in Section 3. We have also implemented a web-based view providing hyperlinks to navigate through the presented information.

Bringing redex trailing to the world of functional logic programming also entailed special requirements for the viewer. In order to enable the search for bugs in programs using logical language features, we designed a new view on the redex trail. This view is independent from the ones described in [19] and will be described in detail in future works.

7. RELATED WORK

The necessity for presenting the execution trace for debugging in a way different from the concrete execution is known for a long time in lazy functional languages [14]. This is due to the fact that the control flow of a demand-driven evaluation is very different from the program text. Thus, in order to relate the concrete program execution to the intended (declarative) semantics of a program, it is necessary to provide different views of the concrete sequence of execution steps. These are usually generated by reconstructing appropriate views from a stored execution trace. For instance, Freja [15] shows function calls with fully evaluated arguments and results (at least as far as the computation has demanded their evaluation) in order to apply algorithmic debugging techniques from logic programming [17]. A similar view of function calls is provided in Hat [18] that presents a computation "backwards" from the results to the initial calls. Furthermore, Hood [7] supports the observation of data structures and functional objects at distinguished points in a program specified by the programmer. All these approaches stores information about the concrete program execution before presenting these different views to the programmer. [19] presents a "unified" trace model by storing appropriate execution trace information from which the views of Freja, Hat, or Hood can be generated.

All these proposed techniques are related to functional

languages since logic languages are usually based on a strict evaluation strategy that makes it easier to relate concrete trace results with the program text [4]. Thus, the extension of these techniques to modern functional logic languages based on non-strict evaluation (i.e., the addition of non-determinism and logical variables) is an open problem addressed in this paper. A recent exception is [3] where an extension of Hood to functional logic programs is presented but without a formal justification. Chitil [5] presents a first attempt to formally justify trace-oriented debugging methods for non-strict languages by augmenting a small-step operational semantics. However, [5] considers purely functional languages and does not provide correctness results. Therefore, our work can be considered as the first approach to formally justify trace-oriented debugging methods for non-strict functional (logic) languages.

8. CONCLUSIONS AND FUTURE WORK

In this work we have presented an instrumented operational semantics to trace functional logic computations. In addition to the normal output of the computed values and bindings, this semantics yields a data structure representing a redex trail. This redex trail is very useful for debugging purposes. The main contribution of this work is twofold: (1) the extension of redex trailing for functional logic languages and (2) the first known formal definition which allows formal reasoning about the tracing process. Using this definition we were able to formally proof essential correctness properties of the computed redex trails. In consequence, this work provides a first foundation for techniques formerly used intuitively by stating program transformations. Furthermore, we have described prototypical implementations of both the operational semantics and viewers for the redex trails and discussed some issues of these implementations.

There are several points for future work. As this paper is mainly concerned with foundations, a detailed description of applications and usability of this approach is desirable. Moreover, the current implementation of the trail construction is based on an interpreter of the operational semantics. In order to make the approach useful for larger programs, we want to construct the trail by transforming the original program. Work on such a program transformation is already quite advanced and further improvements on the efficiency of the tools are planned. Finally, the debugging tools will hopefully be soon in a state of practicality. We will then encourage a broader usage of these tools and expect some feedback which enables further improvement of the different views on the redex trails.

Acknowledgments

We would like to thank Santiago Escobar for many helpful comments and discussions on an earlier version of this paper.

9. REFERENCES

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. To appear in *Journal of Symbolic Computation*, 2004.

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

[3] B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing Functional Logic Computations. In *Proc. of the Sixth Int'l Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pages 193–208. Springer LNCS 3057, 2004.

[4] L. Byrd. Understanding the control flow of prolog programs. In *Proc. of the Workshop on Logic Programming*, Debrecen, 1980.

[5] O. Chitil. A Semantics for Tracing. In *13th Int'l Workshop on Implementation of Functional Languages (IFL 2001)*, pages 249–254. Ericsson CSL, 2001.

[6] R. Echahed and J. Janodet. Admissible Graph Rewriting and Narrowing. In *Proc. of the 1998 Joint Int'l Conf. and Symp. on Logic Programming*, pages 325–340. MIT Press, 1998.

[7] A. Gill. Debugging Haskell by Observing Intermediate Data Structures. In *Proc. of the 4th Haskell Workshop*. University of Nottingham, 2000.

[8] 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.

[9] 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, 1997.

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

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

[12] 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.

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

[14] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

[15] H. Nilsson and J. Sparud. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering*, 4(2):121–150, 1997.

[16] S. Peyton Jones, editor. *Haskell 98 Language and Libraries : The Revised Report*. Cambridge University Press, 2003.

[17] E.Y. Shapiro. *Algorithmic Programming Debugging*. MIT Press, 1983.

[18] J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of the 9th Int'l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS 1292, 1997.

[19] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-View Tracing for Haskell: a New Hat. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. Universiteit Utrecht UU-CS-2001-23, 2001.