Dynamic Slicing Based on Redex Trails[*]

Claudio Ochoa
DSIC, T.U. Valencia
Camino de Vera s/n
46022 Valencia, Spain
cochoa@dsic.upv.es

Josep Silva
DSIC, T.U. Valencia
Camino de Vera s/n
46022 Valencia, Spain
jsilva@dsic.upv.es

Germán Vidal
DSIC, T.U. Valencia
Camino de Vera s/n
46022 Valencia, Spain
gvidal@dsic.upv.es

ABSTRACT

Tracing computations is a widely used methodology for program debugging. Lazy languages, in particular, pose new demands on tracing techniques since following the *actual* trace of a computation is generally useless. Typically, they rely on the construction of a *redex trail*, a graph that describes the reductions of a computation and its relationships. While tracing provides a significant help for locating bugs, the task still remains complex. A well-known debugging technique for imperative programs is based on *dynamic slicing*, a method to find the program statements that influence the computation of a value for a specific program input.

In this work, we introduce a novel technique for dynamic slicing in lazy functional logic languages. Rather than starting from scratch, our technique relies on (a slight extension of) redex trails. We provide a method to compute a *correct* and *minimal* dynamic slice from the redex trail of a computation. A clear advantage of our proposal is that one can enhance existing tracers with slicing capabilities with a modest implementation effort, since the same data structure (the redex trail) can be used for both tracing and slicing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Languages, Theory

Keywords

Lazy functional logic programming, slicing, redex trails

*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, and by MCYT under grants HA2001-59 and HU2003-003.

1. INTRODUCTION

Modern functional logic languages like Curry [10] and Toy [13] combine the most important features of functional *and* logic languages, e.g., lazy evaluation and non-determinism (see, e.g., [8]). Due to the complexity of their operational semantics, following the *actual* trace of a computation is often useless. In the context of (purely) functional lazy languages like Haskell [15], this drawback is overcome by introducing higher-level models that hide the details of lazy evaluation (e.g., Hood [6], Freja [14], Buddha [16], and Hat [18, 21]). Many of these approaches are based on the construction of a *redex trail* [18], 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 to the parent redex that created it.

Recently, Braßel *et al* [4] introduced an instrumented version of an operational semantics for the kernel of lazy functional logic languages that returns, not only the computed values and bindings, but also a trail of the computation. As in the *Augmented Redex Trail* approach [21], this trail can be used to perform tracing and algorithmic debugging, as well as to observe data structures through a computation. Moreover, the correctness of the computed trail is formally proved, which amounts to say that it contains all (and only) the reductions performed in a computation.

While tracing-based debugging provides a powerful tool for inspecting erroneous computations, the task still remains complex. A well-known debugging technique for imperative programs is based on *slicing* [22], a method for decomposing programs by analyzing their data and control flow. Roughly speaking, a *program slice* consists of those program statements which are (potentially) related with the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *program dependence graph* [5] that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards or forwards (from the slicing criterion), which is known as *backward* or *forward* slicing, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. A complete survey on slicing can be found, e.g., in [19].

When debugging programs, we usually start from a particular computation that outputs an incorrect value. In this situation, dynamic slicing may help the programmer to find the location of the bug by extracting a program slice containing only the sentences whose execution influences the incorrect value. Therefore, we are mainly interested in dynamic slicing [11] which only reflects the actual dependences of the erroneous execution, thus producing smaller—more precise—slices than static slicing [19].

To the best of our knowledge, there is no dynamic slicing tool for lazy languages like Haskell or Curry. Therefore, our original motivation was to fill this gap with a formal technique for dynamic slicing in lazy functional *logic* languages (nevertheless, the basic ideas also apply to pure lazy functional languages like Haskell). Rather than starting from scratch, our technique relies on a slight extension of redex trails. Basically, we extend the redex trail model in order to also store the *location*, in the program, of each reduced expression. Then, we compute a dynamic slice by first gathering the *reachable* nodes of the redex trail—according to the slicing criterion—and, then, deleting those program expressions which are not related to the collected nodes.

From our point of view, dynamic slicing may provide a complementary—rather than alternative—approach to tracing. A clear advantage of our approach is that one can enhance existing tracers with slicing capabilities with a modest implementation effort, since the same data structure—the redex trail—is used for both tracing and slicing.

The main contributions of this work can be summarized as follows: (1) We introduce a flexible notion of *slicing criterion* for a lazy functional (logic) language. Previous definitions (e.g., [2, 17]) only considered a *fixed* slicing criterion (i.e., the value computed by the whole program). (2) We formalize the concept of dynamic backward slicing for lazy functional logic programs and show that, in this context, forward slicing can be regarded as a particular case of backward slicing. (3) We define the first (correct and minimal) dynamic slicing technique for the considered programs. (4) Finally, we show how tracing (based on redex trails) and slicing can be combined into a single framework.

This paper is organized as follows. In the next section, we recall the syntax of the flat language. Section 3 presents an informal introduction to tracing and slicing in our setting. Section 4 formalizes an instrumented semantics which builds the trail of a computation including the location of each program expression. Section 5 defines the main concepts involved in dynamic slicing and introduces a method to compute dynamic slices from a redex trail. Section 6 discusses the implementation of a debugging tool which integrates both tracing and slicing. Section 7 includes a comparison to related work and, finally, Section 8 concludes and points out several directions for further research.

2. THE LANGUAGE

In this work, we consider *flat* programs [9], a convenient standard representation for functional logic programs which makes explicit the pattern matching strategy by the use of case expressions. It constitutes the kernel of modern declarative multi-paradigm languages like Curry [8, 10] and Toy [13]. In addition, we assume in the following that flat programs are *normalized*, i.e., *let* constructs are used to ensure that the arguments of functions and constructors are always variables (not necessarily pairwise different). This is essential to express *sharing* without the use of complex graph structures. A simple normalization algorithm can be found in [1]. Basically, this algorithm introduces one new let construct for each non-variable argument of a function or constructor call, e.g., $f(e)$ is transformed into "$let\ x = e\ in\ f(x)$."

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

$$
\begin{array}{lcll}
\mathcal{X} & = & \{x, y, z, \ldots\} & \text{(variables)} \\
\mathcal{C} & = & \{a, b, c, \ldots\} & \text{(constructors)} \\
\mathcal{F} & = & \{f, g, h, \ldots\} & \text{(defined functions)}
\end{array}
$$

Figure 1: Syntax for normalized flat programs

The syntax for normalized flat programs is shown in Figure 1, where $\overline{o_n}$ denotes the *sequence of objects* $o_1, \ldots, o_n$. 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, data constructors, function calls, let bindings where the local variable x is only visible in e_1 and e_2, disjunctions (e.g., to represent set-valued functions), and case expressions. In general, a case expression has the following form (we write $(f)case$ for either $fcase$ or $case$):

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

where x is a variable, $c_1, \ldots, c_k$ are different constructors, and $e_1, \ldots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are locally introduced and bind the variables of e_i. The difference between *case* and *fcase* only shows up when the argument x evaluates (at runtime) to a free variable: *case* suspends whereas *fcase* non-deterministically binds this variable to the pattern in a branch of the case expression.

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 variable or an expression with a constructor at the outermost position. Consequently, the associated operational semantics describes the evaluation of expressions only to head normal form. This is not a restriction since the evaluation to normal form can be reduced to head normal form computations (see, e.g., [9]).

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. In the following, we assume that all extra variables x are explicitly introduced in flat programs by a direct circular let binding of the form "$let\ x = x\ in\ e$". In this paper, we call such variables which are bound to themselves *logical variables.*

3. FROM TRACING TO SLICING

In this section, we present an overview of a debugging tool for lazy functional logic languages that combines tracing and dynamic slicing based on redex trails.

The original *redex trail* is a directed graph which records copies of all values and redexes (*red*ucible *ex*pressions) of a

```
main = printMax (minmax [coin, S Z])

printMin t = fcase t of {Pair x y -> printNat x}
printMax t = fcase t of {Pair x y -> printNat y}

printNat n = fcase n of {Z   -> 0;
                         S m -> 1 + printNat m}

fst t = fcase t of {Pair x y -> x}
snd t = fcase t of {Pair x y -> y}

coin = Z or (S Z)

minmax xs =
  fcase xs of
  {y:ys -> fcase ys of
           {[]   -> Pair y y;
            z:zs -> let m = minmax (z:zs)
                    in Pair (min y (fst m))
                            (max y (snd m))} }

min x y = ite (leq x y) x y
max x y = ite (leq x y) y x

ite x y z = fcase x of {True -> y;
                        False -> z}

leq x y = fcase x of
              {Z   -> False;
               S n -> fcase y of
                       {Z   -> False;
                        S m -> leq n m} }
```

Figure 2: Example program `minmax`

```
main = printMax (minmax [coin, S Z])

printMin t = fcase t of {Pair x y -> printNat x}
printMax t = fcase t of {Pair x y -> printNat y}

printNat n = fcase n of {Z   -> 0;
                         S m -> 1 + printNat m}

fst t = fcase t of {Pair x y -> x}
snd t = fcase t of {Pair x y -> y}

coin = Z or (S Z)

minmax xs =
  fcase xs of
  {y:ys -> fcase ys of
           {[]   ->  Pair y y ;
            z:zs -> let m = minmax (z:zs)
                    in Pair (min y (fst m))
                            (max y (snd m))} }

min x y = ite (leq x y) x y
max x y = ite (leq x y) y x

ite x y z = fcase x of {True -> y;
                        False -> z}

leq x y = fcase x of
              {Z   -> False;
               S n -> fcase y of
                       {Z   -> False;
                        S m -> leq n m} }
```

Figure 3: Slice of program `minmax`

computation, with a backward link from each reduct (and its proper subexpressions) to the parent redex that created it [18]. The ART (*Augmented Redex Trail*) model of the Haskell tracer Hat [21] mainly extends redex trails by also including forward links from every redex to its reduct. Recently, Braßel *et al* [4] introduced a data structure which shares many similarities with the ART model but also includes a special treatment to cope with non-determinism (i.e., disjunctions and flexible case structures). Let us first review this tracing technique by means of an example.

Example 3.1. *Consider the flat program*[1] *shown in Figure 2 (inspired by a similar example in [12]), where data structures are built from:*

```
data Nat = Z | S Nat
data Pairs = Pair Nat Nat
data ListNat = Nil | Cons Nat ListNat
```

As it is common practice in functional languages, we often use "`[]`" and "`:`" as a shorthand for `Nil` *and* `Cons`.

From now on, we assume that computations always start from the distinguished function `main` which has no arguments. The execution of the program above should compute the maximum of the list `[coin, S Z]`, i.e., `S Z`, and thus return `1`. However, it produces—non-deterministically, due to the disjunction in the definition of `coin`—two results, `0` and `1`. Clearly, the first result is erroneous. In order to trace the execution of this program, the considered tracing tool builds the redex trails of both computations.

[1]Here and in the following examples, for the sake of readability, we omit some of the brackets and write function applications as in Curry. Moreover, in this section, we consider programs that are not normalized for clarity.

Figure 4 shows the redex trail corresponding to the first (erroneous) computation (for the time being, the reader can safely ignore the distinction between white and shadowed nodes). In general, redex trails 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 `printMax` in Fig. 4).

Argument arrows: Arguments of both function and constructor calls are denoted by a pointer to the corresponding expression. If the evaluation of an argument is not required in a computation, we have a null pointer for that argument (e.g., the first argument of `Pair` in Fig. 4).

Parent arrows: They are denoted by dashed arrows and point either to the redex from which they result (the inverse of the successor arrow) like, e.g., from `printMax` to `main` in Fig. 4, or to the expression who *demanded* its evaluation, e.g., from `max` to `fcase`.

From the computed trails, the user can inspect the trace of any computation. In our example, the tool initially shows

```
main -> 0
     -> 1
```

to point out that two results were computed. If the user selects the first (erroneous) result, `0`, the following top-level trace of the execution of `main` is shown:

```
0 = main
0 = printMax (Pair _ Z)
0 = printNat Z
0 = 0
```

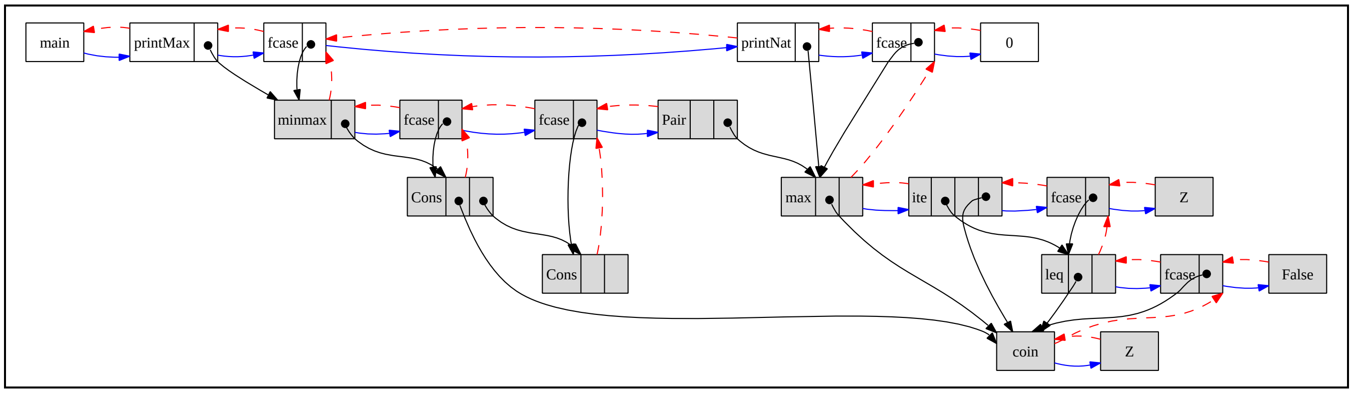

Figure 4: Redex trail for Example 3.1

where "_" denotes an argument whose evaluation is not needed (i.e., a null pointer in the redex trail). 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. Every trace shows a computation from the final result (bottom line) to the initial function call (top line) following the parent arrows of the associated redex trail.

From the trace above, the user can easily see that the argument of `printMax` is incorrect, since the second argument of constructor `Pair`, i.e., the maximum of the input list, should be `(S Z)` rather than `Z`. If the user selects the argument of `printMax`, the following subtrace is shown:

```
0        = printMax (Pair _ Z)
Pair _ Z = minmax   (Z:_)
Pair _ Z = Pair      _  Z
```

At this point, the user can conclude that the *evaluation* of `minmax` (rather than its *definition*!) contains a bug since it returns `Z` (the second element of `Pair`) as the maximum of a list headed by `Z`, ignoring the remaining elements of the list (which were not evaluated in the computation).

Now, the user can either try to figure out the location of the bug by computing further subtraces (which ones?) or by isolating the code fragment which is responsible of the wrong result by *dynamic slicing* and, then, inspecting the computed slice. In principle, *forward* and *backward* slicing could be informally defined in our setting as follows:

Forward slicing: given an entry *val* = *exp* in the trace of a computation, calculate all program constructs that are needed to compute *val* from *exp*. Here, the slicing criterion would be a pair of the form $\langle exp, val\rangle$.

Backward slicing: given an entry *val* = *exp* in the trace of a computation and a *pattern* π—that indicates which part of *val* should be "retained" and which part should be "discarded", like the projections of [17]—calculate all program constructs that are necessary to compute the relevant part of *val*—according to π—from *exp*. Here, the slicing criterion would be a triple $\langle exp, val, \pi\rangle$.

However, if we look at the definitions above, it should be clear that *forward slicing is just a particular case of backward slicing* when the considered pattern indicates that we are interested in the entire result. Therefore, in the following, we focus on dynamic *backward* slicing.

Our proposal for integrating tracing and debugging can be summarized as follows:

- Firstly, the user runs a tracing tool that shows the execution trace of the considered computations.
- When an entry *val* = *exp* of some trace is found erroneous, the user selects this entry (and, optionally, a pattern π for *val* if only part of the result is of interest).
- Now, the original program is shown to the user in such a way that the expressions in the computed slice are clearly distinguishable (e.g., by using a different color).

We consider that patterns for slicing are built from constructor symbols, the special symbol $\bot$ (which denotes a subexpression of the value whose computation is not relevant), and $\top$ (a subexpression which is relevant). Consider, e.g., the following slicing criterion: $\langle$`minmax (Z : _)`, `Pair _ Z`, `Pair` $\bot\,\top\rangle$, associated to the erroneous entry of the previous trace for `minmax`. Intuitively, it determines the extraction of a slice with the expressions that are involved in the computation of the second argument of `Pair` (starting from the call to `minmax`). The computed slice is shown in Figure 3 where expressions that do not belong to the slice are grey-colored.

By inspecting the slice, we can check that `minmax` only calls to function `max` which, in turn, calls to functions `ite`, a standard conditional, and `leq` (for "less or equal"). However, the only case which is not deleted from the definition of `leq` can be read as "`Z` is *not* less than or equal to any natural number", which is clearly wrong. Also, observe that the slice is particularly useful to understand the subtleties of lazy evaluation. For instance, function `minmax` is only called once, with no recursive call to inspect the tail of the input list (as a beginner to lazy languages would expect). This motivates the need for powerful tools for debugging and program understanding in lazy functional (logic) languages.

4. BUILDING THE EXTENDED TRAIL

Braßel *et al* [4] define an instrumented operational semantics for lazy functional logic programs that generates the *redex trail* of a computation. In this section, we extend this instrumented semantics so that the computed trail also stores *program positions*, which are useful to identify precisely the program location of each expression in the trail.

In order to keep the paper self-contained, in the following we present the complete instrumented semantics, also including our extension to compute program positions. The instrumented operational semantics is shown in Figure 5. We distinguish three components in this semantics: the first component (columns *Heap*, *Control*, and *Stack*) defines the standard semantics introduced by Albert *et al* [1], the second component (columns *Graph*, *Ref.*, and *Par.*) is used to build the trail [4], and the third component (columns *Fun* and *Pos*) is used to store *program positions*, i.e., a function name and a position (within this function).

The semantics obeys the following naming conventions:

$$\Gamma, \Delta \in \mathit{Heap} :: \mathcal{X} \to \mathit{Exp} \qquad v \in \mathit{Value} ::= x \mid c(\overline{v_n})$$

A *heap* is a partial mapping from variables to expressions[2] (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 $\Gamma[x] = x$. A *stack* (a list of variable names and case alternatives where the empty stack is denoted by `[]`) is used to represent the current context. A *value* is a constructor-rooted term or a logical variable (w.r.t. the associated heap).

Definition 4.1 (configuration). *A configuration of the semantics is a tuple $\langle \Gamma, e, S, G, r, p, g, w\rangle$ where $\Gamma \in \mathit{Heap}$ is the current heap, $e \in \mathit{Exp}$ is the expression to be evaluated (called the control of the semantics), S is the stack, G is a directed graph (the trail built so far), r and p are references for the current and parent nodes of e, and g and w denote the program position of e.*

First, let us briefly explain the first component of the semantics (columns *Heap*, *Control* and *Stack*).

Rule varcons 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.

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

Rule fun performs a simple function unfolding; here, we assume that the considered program P is a global parameter of the calculus.

In order to reduce a let construct, rule let adds the binding to the heap and proceeds with the evaluation of the main argument of *let*. We rename the local variable with a fresh name in order to avoid variable name clashes.

Rule or *non-deterministically* evaluates a disjunction by either evaluating the first or the second argument.

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 there is no matching pattern, rule select-f returns the special symbol *Fail* (a constructor). 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. Otherwise, if the case expression on the stack is rigid, rule guess-f returns *Fail*.

We now consider the construction of the extended trail. The trail is a directed graph with nodes identified by references[3] that are labeled with expressions. In contrast to [4], our *extended* trails also add to each node a *program position* that uniquely determines its location in the program.

Definition 4.2 (position, program position). *Positions are represented by a sequence of natural numbers, where Λ denotes the empty sequence (i.e., the root position). They are used to address subexpressions of an expression viewed as a tree:*

$e\vert_\Lambda = e$	*for all expression e*
$c(\overline{x_n})\vert_{i.w} = x_i\vert_w$	*if $i \in \{1,\ldots,n\}$*
$f(\overline{x_n})\vert_{i.w} = x_i\vert_w$	*if $i \in \{1,\ldots,n\}$*
$\mathit{let}\ x = e\ \mathit{in}\ e'\vert_{1.w} = e\vert_w$	
$\mathit{let}\ x = e\ \mathit{in}\ e'\vert_{2.w} = e'\vert_w$	
$e_1\ \mathit{or}\ e_2\vert_{i.w} = e_i\vert_w$	*if $i \in \{1,2\}$*
$(f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p_n \to e_n}\}\vert_{1.w} = x\vert_w$	
$(f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p_n \to e_n}\}\vert_{2.i.w} = e_i\vert_w$	*if $i \in \{1,\ldots,n\}$*

Given a program P, we let $\mathcal{P}os(P)$ denote the set of all program positions in P. A program position is a pair $(g, w) \in \mathcal{P}os(P)$ that addresses the subexpression $e|_w$ in the right-hand side of the definition, $g(\overline{x_n}) = e$, of function g in P.

Note that variables in a let construct or patterns in a case expression cannot be addressed (since they will not be considered in the slicing process). While other approaches to dynamic slicing (e.g., [3]) consider some form of *explicit* labeling for each program expression, our program positions can be seen as a form of *implicit* labeling.

In the construction of the extended trail, 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.

- The mapping from nodes to program positions is denoted by $r \mapsto_{(g,w)}$, which means that the label of node r appears at program position (g, w).

- Often, we write $r \overset{p}{\underset{q}{\mapsto}}_{(g,w)} e$ to denote that node r is labeled with expression e, node p is the parent of r, node q is the successor of r, and e appears at program position (g, w). Similarly, we also write $r \overset{p}{\mapsto}_{(g,w)} e$ 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 due to normalization. Hence, these arrows are also called *variable pointers*.

[2]Actually, we consider that each mapping $x \mapsto_{(g,w)} e$ in the heap also contains the program position (g, w) of expression e. Since program positions are not relevant for the standard component of the semantics, we will discuss this point later.

[3]The domain for references is not fixed. One can use, e.g., natural numbers but more complex domains are possible.

Rule	Heap	Control	Stack	Graph	*Ref.*	*Par.*	*Fun*	*Pos*
varcons	$\Gamma[x \mapsto_{(h,w')} t]$	x	S	G	r	p	g	w
	$\Longrightarrow \Gamma[x \mapsto_{(h,w')} t]$	t	S	$G \bowtie (x \rightsquigarrow r)$	r	p	h	w'
varexp	$\Gamma[x \mapsto_{(h,w')} e]$	x	S	G	r	p	g	w
	$\Longrightarrow \Gamma[x \mapsto_{(h,w')} e]$	e	$x : S$	$G \bowtie (x \rightsquigarrow r)$	r	p	h	w'
val	Γ	v	$x : S$	G	r	p	g	w
	$\Longrightarrow \Gamma[x \mapsto_{(g,w)} v]$	v	S	G	r	p	g	w
fun	Γ	$f(\overline{x_n})$	S	G	r	p	g	w
	$\Longrightarrow \Gamma$	$\rho(e)$	S	$G[r \overset{p}{\underset{q}{\mapsto}}_{(g,w)} f(\overline{x_n})]$	q	r	f	Λ
let	Γ	$let\ x = e_1\ in\ e_2$	S	G	r	p	g	w
	$\Longrightarrow \Gamma[y \mapsto_{(g,w.1)} \rho(e_1)]$	$\rho(e_2)$	S	$G[r \overset{p}{\underset{q}{\mapsto}}_{(g,w)} \rho(let\ x = e_1\ in\ e_2)]$	q	r	g	$w.2$
or	Γ	$e_1\ or\ e_2$	S	G	r	p	g	w
	$\Longrightarrow \Gamma$	e_i	S	$G[r \overset{p}{\underset{q}{\mapsto}}_{(g,w)} e_1\ or\ e_2]$	q	r	g	$w.i$
case	Γ	$(f) case\ x\ of\ \{\overline{p_k \to e_k}\}$	S	G	r	p	g	w
	$\Longrightarrow \Gamma$	x	$((f)\{\overline{p_k \to e_k}\}, r) : S$	$G[r \overset{p}{\mapsto}_{(g,w)} (f) case\ x\ of\ \{\overline{p_k \to e_k}\}]$	q	r	g	$w.1$
select	Γ	$c(\overline{y_n})$	$((f)\{\overline{p_k \to e_k}\}, r') : S$	G	r	p	g	w
	$\Longrightarrow \Gamma$	$\rho(e_i)$	S	$G[r \overset{p}{\mapsto}_{(g,w)} c(\overline{y_n}), r' \underset{q}{\mapsto}_{(h,w')}]$	q	r'	h	$w'.2.i$
select-f	Γ	$c(\overline{y_n})$	$((f)\{\overline{p_k \to e_k}\}, r') : S$	G	r	p	g	w
	$\Longrightarrow \Gamma$	$Fail$	S	$G[r \overset{p}{\mapsto}_{(g,w)} c(\overline{y_n}), r' \underset{q}{\mapsto}]$	q	r'	-	-
guess	$\Gamma[y \mapsto_{(g,w)} y]$	y	$(f\{\overline{p_k \to e_k}\}, r') : S$	G	r	p	g	w
	$\Longrightarrow \Gamma[y \mapsto_{(\_,\_)} \rho(p_i), \overline{y_n \mapsto_{(\_,\_)} y_n}]$	$\rho(e_i)$	S	$G[r \overset{p}{\underset{q}{\mapsto}}_{(g,w)} LogVar, q \overset{r'}{\mapsto}_{(\_,\_)} \rho(p_i),$ $y \rightsquigarrow r, r' \underset{s}{\mapsto}_{(h,w')}]$	s	r'	h	$w'.2.i$
guess-f	$\Gamma[y \mapsto_{(g,w)} y]$	y	$(\{\overline{p_k \to e_k}\}, r') : S$	G	r	p	g	w
	$\Longrightarrow \Gamma[y \mapsto_{(g,w)} y]$	$Fail$	S	$G[r \overset{p}{\mapsto}_{(g,w)} LogVar, y \rightsquigarrow r, r' \underset{s}{\mapsto}]$	s	r'	-	-
success-c	Γ	$c(\overline{x_n})$	[]	G	r	p	g	w
	$\Longrightarrow \Gamma$	$\diamond$	[]	$G[r \overset{p}{\mapsto}_{(g,w)} c(\overline{x_n})]$	$\Box$	r	-	-
success-x	$\Gamma[x \mapsto_{(g,w)} x]$	x	[]	G	r	p	g	w
	$\Longrightarrow \Gamma[x \mapsto_{(g,w)} x]$	$\diamond$	[]	$G[r \overset{p}{\mapsto}_{(g,w)} LogVar, x \rightsquigarrow r]$	$\Box$	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:	$\rho = \{x \mapsto y\}$ and y is fresh
or:	$i \in \{1, 2\}$
select:	$i \in \{1, \dots k\}$, $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$
select-f:	$\nexists i \in \{1, \dots, k\}$ such that $p_i = c(\overline{x_n})$
guess:	$i \in \{1, \dots k\}$, $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are fresh

Figure 5: Small-step instrumented semantics

Given a configuration $\langle \Gamma, e, S, G, r, p, g, w\rangle$, the basic idea of the graph construction is to record in every step the current control, e, at the current reference, r, with parent p and program position (g, w). Similarly to the heap, we use $G[r \mapsto e]$ either as a condition on a graph G or as a modification of G. A brief explanation for each rule of the semantics follows:

(varcons and varexp) Since these rules are used to perform a variable lookup in the heap, a new variable pointer for x is added to the graph—if it does not yet contain such a pointer—to denote that the value of x is *needed* in the computation. For this, auxiliary function "$\bowtie$" is introduced:

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

which is used to take care of sharing: if the value of a variable was already demanded, no new variable pointer is added to the graph. In the derived configuration we update the current program position with that of the expression in the heap (h, w') introduced by rule let (see below).

(val) It is only used to update the value of x in the heap. The associated program position for the binding is also updated with the current program position (g, w).

(fun) Whenever this rule is applied, a new node r is added to the graph. This node is labeled with the function call $f(\overline{x_n})$, has parent p, successor q (a fresh reference), and program position (g, w). In the new configuration, r becomes the parent reference and the fresh reference q represents the current reference. Trivially, we reset the program position to (f, Λ) since the expression in the control of the derived configuration is the complete right-hand side of function f.

(let) It proceeds in a similar way as the previous rule by introducing a new node for the (renamed) let expression. As for program positions, the binding $x \mapsto_{(g,w.1)} e_1$ introduced in the heap is labeled with the program position of e_1. This is necessary to recover its program position when rules varcons and varexp move this expression to the control. The program position in the new configuration is updated to $(g, w.2)$ to address the expression e_2 of the let.

(or) Similarly to the previous rule, a new node for the disjunction is introduced in the graph. Clearly, the program position of the derived configuration is $(g, w.i)$ where $i \in \{1, 2\}$ refers to the selected disjunct.

(case) This rule adds to the graph a new node r labeled with a 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, select-f, guess, and guess-f may eventually set the right successor for r. The program position of the derived configuration is updated to $(g, w.1)$ to address the case argument.

(select) This rule adds a new node r labeled with the computed value $c(\overline{y_n})$ for the case argument. It sets p as the parent of r but includes no successor since values are fully evaluated. Reference r' in the stack is used either to set the right successor for the case expression that initiated the subcomputation (the fresh reference q) and to update the program position of the derived configuration, i.e., (h, w').

(select-f) This rule applies when there is no matching branch in the case expression. Since the control of the derived configuration contains the special symbol *Fail* (which does not occur in the program), we set the null program position $(\_, \_)$ as the current program position.

(guess) It proceeds in way similar to rule select. The main difference is that the computed value is a *logical variable*. Here, we add a new node r to the graph labeled with a special symbol $LogVar$, and whose successor is a node q labeled with the selected binding for the logical variable. Observe that patterns (and their local variables) introduced by instantiation do not have an associated program position, i.e., they do not belong to $\mathcal{P}os(P)$.

(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 rule select-f, we set the current program position of the derived configuration to $(\_, \_)$.

(success-c) and (success-x) These rules are only introduced to add a node with the final result of the computation. They apply to configurations in which the stack is empty (thus, there are no pending subcomputations) and the control contains a value. In the derived configuration, we put the special symbol $\diamond$ to denote that no further steps are possible.

Clearly, the instrumented semantics is a conservative extension of the original small-step semantics of [1], since the last five columns of the calculus impose no restriction on the application of the standard component of the semantics (columns *Heap*, *Control*, and *Stack*).

In order to perform computations, we construct an *initial configuration* and (non-deterministically) apply the rules of Figure 5 until a *final configuration* is reached:

Definition 4.3 (initial configuration). *An initial configuration has the form* $\langle \texttt{[]}, \texttt{main}, \texttt{[]}, G_\emptyset, r, \Box, \_, \_\rangle$, *where* $G_\emptyset$ *denotes an empty graph,* r *a reference,* $\Box$ *the null reference, and* $(\_, \_)$ *a null program position (since there is no call to* main *from the right-hand side of any function definition).*

Definition 4.4 (final configuration). *A final configuration has the form:* $\langle \Delta, \diamond, \texttt{[]}, G, \Box, p, \_, \_\rangle$, *where* Δ *is a heap containing the computed bindings and* G *is the extended redex trail of the computation.*

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.

Example 4.5. *Consider again function* leq *of Ex. 3.1, together with the initial call* (leq Z (S Z)). *The normalized flat program is as follows (the program positions of some selected expressions are shown in the right column):*[4]

```
main = let x1 = Z            [let : (main, Λ), Z : (main, 1)]
       in let x2 = S x3               [S x3 : (main, 2.1)]
       in let x3 = Z                   [Z : (main, 2.2.1)]
       in leq x1 x2           [leq x1 x2 : (main, 2.2.2)]

leq x y = fcase x of         [fcase : (leq, Λ), x : (leq, 1)]
           {Z   -> True;           [False : (leq, 2.1)]
            S n -> fcase y of
                    {Z  -> False;
                     S m -> leq n m} }
```

The complete computation with the rules of Fig. 5 is shown in Fig. 6, where we consider natural numbers as references (with initial reference 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.*

[4]Recall that function leq did not appear normalized in Example 3.1 for clarity. Also, the bug is now corrected.

$$\begin{array}{ll}
 & \langle\texttt{[],main,[]},G_\emptyset\texttt{,0,}\Box,\_,\_\rangle \\
\Longrightarrow_{\mathsf{fun}} & \langle\texttt{[],let x1 = Z in let x2 = (S x3) in let x3 = Z in leq x1 x2,[], G[0 }\overset{\Box}{\underset{1}{\mapsto}}_{(\_,\_)}\texttt{ main],1,0,main,}\Lambda\rangle \\
\Longrightarrow_{\mathsf{let}} & \langle\texttt{[x1 }\mapsto_{(\texttt{main},1)}\texttt{ Z], let x2 = (S x3) in let x3 = Z in leq x1 x2,[],} \\
 & \texttt{G[1 }\overset{0}{\underset{2}{\mapsto}}_{(\texttt{main},\Lambda)}\texttt{ let x1 = Z in let x2 = (S x3) in let x3 = Z in leq x1 x2],2,1,main,2}\rangle \\
\Longrightarrow_{\mathsf{let}} & \langle\texttt{[x1 }\mapsto_{(\texttt{main},1)}\texttt{ Z, x2 }\mapsto_{(\texttt{main},2.1)}\texttt{ (S x3)],let x3 = Z in leq x1 x2,[],} \\
 & \texttt{G[2 }\overset{1}{\underset{3}{\mapsto}}_{(\texttt{main},2)}\texttt{ let x2 = (S x3) in let x3 = Z in leq x1 x2],3,2,main,2.2}\rangle \\
\Longrightarrow_{\mathsf{let}} & \langle\texttt{[x1 }\mapsto_{(\texttt{main},1)}\texttt{ Z, x2 }\mapsto_{(\texttt{main},2.1)}\texttt{ (S x3), x3 }\mapsto_{(\texttt{main},2.2.1)}\texttt{ Z],leq x1 x2,[],} \\
 & \texttt{G[3 }\overset{2}{\underset{4}{\mapsto}}_{(\texttt{main},2.2)}\texttt{ let x3 = Z in leq x1 x2],4,3,main,2.2.2}\rangle \\
\Longrightarrow_{\mathsf{fun}} & \langle\texttt{[x1 }\mapsto_{(\texttt{main},1)}\texttt{ Z, x2 }\mapsto_{(\texttt{main},2.1)}\texttt{ (S x3), x3 }\mapsto_{(\texttt{main},2.2.1)}\texttt{ Z],fcase x1 of }\{\texttt{Z}\rightarrow\texttt{True;} \\
 & \texttt{(S n)}\rightarrow\texttt{fcase x2 of }\{\texttt{Z}\rightarrow\texttt{False;(S m)}\rightarrow\texttt{leq n m}\}\}\texttt{,[], G[4 }\overset{3}{\underset{5}{\mapsto}}_{(\texttt{main},2.2.2)}\texttt{ leq x1 x2],5,4,leq,}\Lambda\rangle \\
\Longrightarrow_{\mathsf{case}} & \langle\texttt{[x1 }\mapsto_{(\texttt{main},1)}\texttt{ Z, x2 }\mapsto_{(\texttt{main},2.1)}\texttt{ (S x3), x3 }\mapsto_{(\texttt{main},2.2.1)}\texttt{ Z],x1,} \\
 & \texttt{[(f}\{\texttt{Z}\rightarrow\texttt{True;(S n)}\rightarrow\texttt{fcase x2 of }\{\texttt{Z}\rightarrow\texttt{False;(S m)}\rightarrow\texttt{leq n m}\}\}\texttt{,5)]} \\
 & \texttt{G[5 }\overset{4}{\mapsto}_{(\texttt{leq},\Lambda)}\texttt{ fcase x1 of }\{\texttt{Z}\rightarrow\texttt{True; (S n)}\rightarrow\texttt{fcase x2 of }\{\texttt{Z}\rightarrow\texttt{False;(S m)}\rightarrow\texttt{leq n m}\}\}\texttt{],6,5,leq,1}\rangle \\
\Longrightarrow_{\mathsf{varcons}} & \langle\texttt{[x1 }\mapsto_{(\texttt{main},1)}\texttt{ Z, x2 }\mapsto_{(\texttt{main},2.1)}\texttt{ (S x3), x3 }\mapsto_{(\texttt{main},2.2.1)}\texttt{ Z],Z,} \\
 & \texttt{[(f}\{\texttt{Z}\rightarrow\texttt{True;(S n)}\rightarrow\texttt{fcase x2 of }\{\texttt{Z}\rightarrow\texttt{False;(S m)}\rightarrow\texttt{leq n m}\}\}\texttt{,5)], G[x1 }\rightsquigarrow\texttt{ 6],6,5,main,1}\rangle \\
\Longrightarrow_{\mathsf{select}} & \langle\texttt{[x1 }\mapsto_{(\texttt{main},1)}\texttt{ Z, x2 }\mapsto_{(\texttt{main},2.1)}\texttt{ (S x3),x3 }\mapsto_{(\texttt{main},2.2.1)}\texttt{ Z],True,[], G[6 }\overset{5}{\mapsto}_{(\texttt{main},1)}\texttt{ Z, 5 }\underset{7}{\mapsto}_{(\texttt{leq},\Lambda)}\texttt{],7,5,leq,2.1}\rangle \\
\Longrightarrow_{\mathsf{success}} & \langle\texttt{[x1 }\mapsto_{(\texttt{main},1)}\texttt{ Z, x2 }\mapsto_{(\texttt{main},2.1)}\texttt{ (S x3),x3 }\mapsto_{(\texttt{main},2.2.1)}\texttt{ Z], }\diamond\texttt{,[],G[7 }\overset{5}{\mapsto}_{(\texttt{leq},2.1)}\texttt{ True],}\Box\texttt{,7,}\_,\_\rangle
\end{array}$$

Figure 6: An example computation with the instrumented semantics

The correctness of the original trail is proved in [4]. Now, we state the correctness of the computed program positions.

THEOREM 4.6 (PROGRAM POSITIONS).
Let P be a program and $(C \Longrightarrow^* C')$ a complete derivation in P, where G' is the graph in the final configuration C'. If $(r \mapsto_{(g,w)} e') \in G'$, then either $(g = w = \_)$ or $g(\overline{x_n}) = e \in P$ with $e|_w = e'$ (up to variable renaming).

5. COMPUTING THE SLICE

In this section, we formalize the concept of dynamic backward slice and introduce a method to compute it from the extended redex trail.

A dynamic slice is defined as a set of program positions:

DEFINITION 5.1 (DYNAMIC SLICE).
Let P be a program. A dynamic slice for P is a set $\mathcal{W}$ of program positions such that $\mathcal{W} \subseteq \mathcal{P}os(P)$.

In our approach, we are not interested in producing executable slices but in computing the program positions of the program expressions that influence the slicing criterion. This is enough for our purposes, i.e., for showing the original program with some distinguished expressions.

In the literature of dynamic slicing for imperative programs, the slicing criterion usually identifies a concrete point of the execution history, e.g., $\langle \mathtt{n} = 2, 8^1, \mathtt{x}\rangle$, where $\mathtt{n} = 2$ is the input for the program, 8^1 denotes the *first* occurrence of statement 8 in the execution history, and `x` is the variable we are interested in [19]. In principle, it is not difficult to extend this notion of slicing criterion to a *strict* functional language. For instance, we could define a slicing criterion as a pair $\langle f(\overline{v_n}), \pi\rangle$, where $f(\overline{v_n})$ is a function call that occurs during the execution of the program (whose arguments are fully evaluated since we consider a strict language) and π is a restriction on the output of $f(\overline{v_n})$.

Unfortunately, extending this notion to a *lazy* functional language is not trivial at all. For instance, identifying a function call $f(\overline{e_n})$ of the actual execution is impractical since $\overline{e_n}$ are usually rather complex expressions. As an alternative, if a tracing tool that shows function calls with fully evaluated arguments is available (like the tracing tool of Section 3), we can still consider a slicing criterion of the form $\langle f(\overline{v_n}), \pi\rangle$. In this case, $f(\overline{v_n})$ should be understood as "a function call $f(\overline{e_n})$ whose arguments $\overline{e_n}$ are evaluated to $\overline{v_n}$ in the complete computation". However, this criterion might be ambiguous. For example, we could have `f` $(2 * 3)$ and `f` $(3 * 2)$ within the same computation, where the argument of both calls reduces to 6. How can we distinguish one call from another? We solve this problem similarly to traditional approaches in imperative programming by adding a natural number that indicates the occurrence of the given function call, e.g., $\langle \mathtt{f\ 6,\ 1,}\ \pi\rangle$.

Finally, lazy functional *logic* languages pose a new problem due to non-determinism: the same function call may occur several times in the same computation and, moreover, it can be reduced to different values in a non-deterministic way. Consider, e.g., the following function definitions:

```
main = coin + coin
coin = Z or (S Z)
```

Here, `main` can be non-deterministically reduced to `(S Z)` in two ways: either by reducing the leftmost call to `Z` and the rightmost one to `(S Z)` or vice versa. In order to iden-

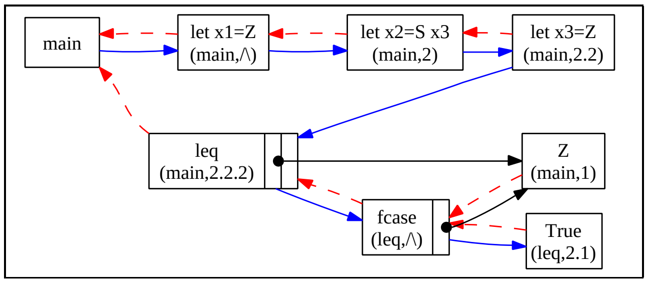

Figure 7: Trail of the computation in Figure 6

tify a concrete occurrence of the given call, we add another element to the slicing criterion: the computed value. Therefore, a slicing criterion is defined as a tuple $\langle f(\overline{v_n}), v, l, \pi\rangle$, where $f(\overline{v_n})$ is a function call with fully evaluated arguments, v is its value (in a particular computation), l is the occurrence of a function call $f(\overline{v_n})$ that outputs v in the considered computation, and π is a pattern that determines the interesting part of the computed value, e.g., $\langle \mathtt{coin}, \mathtt{Z}, 1, \pi\rangle$.

The connection with the tracing tool described in Section 3 should be clear: given a (sub)trace of the form

$$\begin{array}{lcl} val_1 & = & f_1(v_1^1, \ldots, v_m^1) \\ val_2 & = & f_2(v_1^2, \ldots, v_m^2) \\ \ldots & & \\ val_k & = & f_k(v_1^k, \ldots, v_m^k) \end{array}$$

a slicing criterion has the form $\langle f_i(v_1^i, \ldots, v_m^i), val_i, l, \pi\rangle$. Therefore, the user can easily produce a valid slicing criterion from the trace of a computation. Furthermore, traces usually contain a special symbol, "_", to denote that some subexpression is missing because its evaluation was not required in the computation. Consequently, we will also accept expressions containing this special symbol (see below).

Definition 5.2 (slicing criterion). *Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation. A slicing criterion for P w.r.t. $(C_0 \Longrightarrow^* C_m)$ is a tuple $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ such that $f(\overline{pv_n}) \in FCalls$ is a function call whose arguments are fully evaluated, i.e., as much as needed in $(C_0 \Longrightarrow^* C_m)$, $pv \in PValue$ is a partial value, $l > 0$ is a natural number, and $\pi \in Pat$ is a slicing pattern.*

The domains $FCalls$ of fully evaluated calls and $PValue$ of partial values obey the following syntax:

$$FCalls \ ::= \ f(\overline{pv_n}) \qquad pv \in PValue \ ::= \ \_ \mid x \mid c(\overline{pv_k})$$

where $f \in \mathcal{F}$ is a defined function symbol (arity $n \geq 0$), $c \in \mathcal{C}$ is a constructor symbol (arity $k \geq 0$), and "_" is a special symbol to denote any non-evaluated expression.

The domain Pat of slicing patterns is defined as follows:

$$\pi \in Pat \ ::= \ \bot \mid \top \mid c(\overline{\pi_k})$$

where $c \in \mathcal{C}$ is a constructor symbol of arity $k \geq 0$, $\bot$ denotes a subexpression of the value whose computation is not relevant and $\top$ a subexpression which is relevant.[5]

As mentioned in Section 3, a slicing criterion of the form $\langle f(\overline{pv_n}), pv, l, \top\rangle$ actually means that we are interested in the complete evaluation of $f(\overline{pv_n})$, i.e., a *forward* slice.

Given a particular derivation $(C_0 \Longrightarrow^* C_m)$, in order to formalize which is the configuration C_i associated to a slicing criterion, we first need the following abstraction relation:

Definition 5.3 (abstraction). *Let $pv_1, pv_2 \in PValue$ be partial values. Then, we say that pv_1 is an abstraction of pv_2 iff $pv_1 \sqsubseteq pv_2$, where the relation $\sqsubseteq :: (PValue \times PValue \to Bool)$ is defined as follows:*

$$\begin{array}{rcll} \_ & \sqsubseteq & pv & \textit{for all } pv \in PValue \\ x & \sqsubseteq & x & \textit{for all variable } x \in \mathcal{X} \\ c(\overline{pv_n}) & \sqsubseteq & c(\overline{pv'_n}) & \textit{if } pv_i \sqsubseteq pv'_i \textit{ for all } i = 1, \ldots, n \end{array}$$

where $c \in \mathcal{C}$ is a constructor symbol of arity $n \geq 0$.

[5]Our patterns for slicing are similar to the *liveness patterns* used to perform dead code elimination [12].

The relation $\sqsubseteq$ is useful to check whether a partial value is "less defined" than another partial value because of some occurrences of the special symbol "_". Let us illustrate the abstraction relation with some examples:

$$\begin{array}{rclcrcl} c(\_, \_) & \sqsubseteq & c(a, \_) & \qquad & c(a, b) & \not\sqsubseteq & c(a, \_) \\ c(a, \_) & \sqsubseteq & c(a, c(a, b)) & & c(a, \_) & \not\sqsubseteq & c(b, \_) \end{array}$$

where $a, b, c \in \mathcal{C}$ are constructor symbols. If $pv_1 \sqsubseteq pv_2$, we either say that pv_1 is an abstraction of pv_2 or that pv_2 is *more defined* than pv_1.

We can now formally determine the configuration associated to a slicing criterion. In the following, given a configuration of the form $C = \langle \Gamma, e, S, G, r, p, g, w\rangle$, we let $heap(C) = \Gamma$, $control(C) = e$, and $stack(C) = S$.

Definition 5.4 (SC-configuration). *Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation in P. Given a slicing criterion $\langle f(\overline{pv_n}), pv, l, \pi\rangle$, the associated SC-configuration C_i is the l-th occurrence in $(C_0 \Longrightarrow^* C_m)$ of a configuration fulfilling the following conditions:*

1. *$control(C_i) = f(\overline{x_n})$,*
2. *$pv_i \sqsubseteq val(heap(C_m), x_i)$ for all $i = 1, \ldots, n$, and*
3. *$C_i \Longrightarrow^* C_j$, $i < j$, where*
   (a) *$pv \sqsubseteq val(heap(C_m), control(C_j))$,*
   (b) *$stack(C_i) = stack(C_j)$, and*
   (c) *$\not\exists k$ such that $i < k < j$, $stack(C_i) = stack(C_k)$, and $control(C_k)$ is a value.*

Here, function $val :: (Heap \times Exp \to PValue)$ is defined by

$$val(\Gamma, x) = \begin{cases} x & \textit{if } \Gamma[x] = x \\ val(\Gamma, y) & \textit{if } \Gamma[x] = y \textit{ and } x \neq y \\ c(\overline{val(\Gamma, x_r)}) & \textit{if } \Gamma[x] = c(\overline{x_r}) \\ \_ & \textit{otherwise} \end{cases}$$

Intuitively, the SC-configuration associated to an slicing criterion $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ is the l-th configuration, C_i, of a computation that fulfills the following conditions: i) the control of C is $f(\overline{x_n})$, ii) the variables $\overline{x_n}$ are bound in the computation to expressions which are more defined than $\overline{pv_n}$, and iii) the subcomputation that starts from C_i ends with a configuration that contains a value in the control which is more defined than pv. Observe that conditions (3.b) and (3.c) above are only useful to ensure that $C_i \Longrightarrow^* C_j$ is the complete subcomputation from C_i to a configuration whose control contains the value of the control in C_i.

In order to formalize the correctness and minimality of a dynamic slice, we first need to characterize the configurations that *contribute* to the evaluation of a slicing criterion:

Definition 5.5 (contributing configurations). *Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation in P. Let $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ be a slicing criterion and C_i its associated SC-configuration. The set of configurations that contributes to $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ in $(C_0 \Longrightarrow^* C_m)$ is defined by $contr(C_i, stack(C_i), \pi, \{\})$, where auxiliary function contr is shown in Figure 8.*

Roughly speaking, contributing configurations are obtained as follows. First, we collect all configurations in the subcomputation that starts with the SC-configuration. Once we reach the final configuration of the subcomputation, we

$$
\mathit{contr}(C, S, \pi, \{\overline{(x_k, \pi_k)}\}) = \begin{cases}
\{C\} \cup \mathit{contr}_{var}(C', \{\overline{(x_k, \pi_k)}\} \cup \{\overline{(y_n, \top)}\}) & \text{if } \mathit{stack}(C) = S, \mathit{control}(C) = c(\overline{y_n}), \pi = \top, C \Longrightarrow C' \\
\{C\} \cup \mathit{contr}_{var}(C', \{\overline{(x_k, \pi_k)}\} \cup \{\overline{(y_n, \pi_n)}\}) & \text{if } \mathit{stack}(C) = S, \mathit{control}(C) = c(\overline{y_n}), \pi = c(\overline{\pi_n}), C \Longrightarrow C' \\
\mathit{contr}(C', S, \pi, \{\overline{(x_k, \pi_k)}\}) & \text{if } \mathit{control}(C) = x, \mathit{heap}(C)[x] \neq x, \text{ and } C \Longrightarrow C' \\
\{C\} \cup \mathit{contr}(C', S, \pi, \{\overline{(x_k, \pi_k)}\}) & \text{otherwise, where } C \Longrightarrow C'
\end{cases}
$$

$$
\mathit{contr}_{var}(C, \{\overline{(x_m, \pi_m)}\}) = \begin{cases}
\mathit{contr}(C, \mathit{stack}(C), \pi_i, \{\overline{(x_m, \pi_m)}\}) & \text{if } \mathit{control}(C) = x_i,\ i \in \{1, \ldots, m\}, \text{ and } \pi_i \neq \bot \\
\{\} & \text{if } \mathit{control}(C) = \diamond \\
\mathit{contr}_{var}(C', \{\overline{(x_m, \pi_m)}\}) & \text{otherwise, where } C \Longrightarrow C'
\end{cases}
$$

Figure 8: Auxiliary function *contr* to collect contributing configurations

$$
\mathit{Col}(G, r, \pi) = \begin{cases}
\{\} & \text{if } \pi = \bot \\
\{(g, w)\} \ \cup\ \mathit{Col}(G, q, \pi) & \text{if } (r \underset{q}{\mapsto}_{(g,w)} e) \in G,\ \pi \neq \bot, \text{ and } e \text{ is neither constructor-rooted} \\
 & \text{nor a case expression} \\
\{(g, w)\} \ \cup\ \mathit{Col}(G, q, \pi) \ \cup\ \mathit{Col}(G, r', \mathit{hnf}) & \text{if } (r \underset{q}{\mapsto}_{(g,w)} (f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}) \in G,\ \pi \neq \bot, \text{ and } (x \rightsquigarrow r') \in G \\
\{(g, w)\} & \text{if } (r \mapsto_{(g,w)} c(\overline{x_n})) \in G,\ \pi = \mathit{hnf} \\
\{(g, w)\} \ \cup\ \bigcup_{i \in \{1, \ldots, n\}} \mathit{Col}(G, r_i, \top) & \text{if } (r \mapsto_{(g,w)} c(\overline{x_n})) \in G,\ \pi = \top, \text{ and } (x_i \rightsquigarrow r_i) \in G, i = 1, \ldots, n \\
\{(g, w)\} \ \cup\ \bigcup_{i \in \{1, \ldots, n\}} \mathit{Col}(G, r_i, \pi_i) & \text{if } (r \mapsto_{(g,w)} c(\overline{x_n})) \in G,\ \pi = c(\overline{\pi_n}), \text{ and } (x_i \rightsquigarrow r_i) \in G, i = 1, \ldots, n \\
\mathit{Error} & \text{if } (r \mapsto d(\overline{x_m})) \in G,\ \pi = c(\overline{\pi_n}), \text{ and } c \neq d
\end{cases}
$$

Figure 9: Auxiliary function *Col* to collect program positions

also collect those configurations which are needed to compute the inner subterms of the value *according to pattern* π. For this, function contr_{var} ignores the configurations that are not related to the slicing criterion and recursively calls function *contr* to collect the relevant subcomputations.

Let us note that, although we use configurations of the instrumented semantics, only the *standard* component of these configurations is considered in the definitions above.

A dynamic slice is *correct* if the program positions of all contributing configurations belong to the slice. Formally,

Definition 5.6 (correct slice). *Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation. Let $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ be a slicing criterion and C_i its associated SC-configuration. A program slice $\mathcal{W}$ is a correct slice for P w.r.t. $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ iff for all $\langle \Gamma, e, S, G, r, p, g, w\rangle \in \mathit{contr}(C_i, \mathit{stack}(C_i), \pi, \{\})$, there exists a program position $(g, w) \in \mathcal{W}$.*

Analogously, a correct slice is *minimal* if it only contains the program positions of the contributing configurations:

Definition 5.7 (minimal slice). *Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation. Let $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ be a slicing criterion and C_i its associated SC-configuration. A program slice $\mathcal{W}$ is a minimal slice for P w.r.t. $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ iff it is correct and, for all $(g, w) \in \mathcal{W}$, there is a configuration $\langle \Gamma, e, S, G, r, p, g, w\rangle \in \mathit{contr}(C_i, \mathit{stack}(C_i), \pi, \{\})$.*

According to the previous definitions, one could develop a slicing tool by implementing a meta-interpreter for the operational semantics, defining an algorithm to compute the contributing configurations and, finally, extracting the program positions of these configurations. Our aim in this work, however, is to compute the slice from the extended redex trail, since it is already available in existing tracing tools and contains all the necessary information.

From the extended redex trail, the computation of a dynamic backward slice proceeds basically as follows: first, the node associated to the slicing criterion is identified (the *SC-node*, the counterpart of the SC-configuration) and, then, the set of program positions for the nodes which are *reachable* from the SC-node are collected (the counterpart of the contributing configurations).

In order to identify the SC-node, we first need to define an appropriate *ordering* to traverse the nodes of the graph.

Definition 5.8 (graph traversal). *Let G be an extended redex trail and r_{main} the reference associated to the initial function main. The traversal of the graph is given by the function $\mathcal{T}(G, r_{main})$, which is defined by $\mathcal{T}(G, r) =$*

$$
\begin{cases}
r; \mathcal{T}(G, q) & \textit{if } (r \underset{q}{\mapsto} e) \in G \textit{ and } e \textit{ is not a case} \\
r; \mathcal{T}(G, r'); \mathcal{T}(G, q) & \textit{if } (r \underset{q}{\mapsto} (f)\mathit{case}\ x ...) \in G, (x \rightsquigarrow r') \in G \\
r & \textit{otherwise}
\end{cases}
$$

Roughly speaking, we always follow the successor relation unless a case expression is found. In this case, we insert the complete subcomputation before following the successor relation (compare Figure 4). It can easily be shown that the graph traversal follows the same evaluation order of the instrumented semantics in Figure 5.

Definition 5.9 (SC-node). *Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation, where G is the graph in C_m. Given a slicing criterion $\langle f(\overline{pv_n}), pv, l, \pi\rangle$, the associated SC-node is the l-th occurrence in the traversal $\mathcal{T}(G, r_{main})$ of a node r fulfilling the following conditions:*

1. *$(r \mapsto f(\overline{x_n})) \in G$,*

2. *$pv_i \sqsubseteq \mathit{val}_r(G, r_i)$, where $(x \rightsquigarrow r_i) \in G, i = 1, \ldots, n$, and*

3. *$pv \sqsubseteq \mathit{val}_r(G, r)$.*

Here, function val_r is defined by[6]

$$val_r(G,r) = \begin{cases} \_ & \textit{if } r = \Box \\ c(\overline{val_r(G,r_i)}) & \textit{if } (r \mapsto c(\overline{x_i})) \in G \textit{ and} \\ & (x_i \rightsquigarrow r_i) \in G,\ i = 1,\ldots,n \\ val_r(G,q) & \textit{otherwise, where } (r \underset{q}{\mapsto} e) \in G \end{cases}$$

Observe the parallelism with Definition 5.4. Intuitively, the SC-node associated to an slicing criterion $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ is the l-th node, r, of a traversal of the graph that fulfills the following conditions: i) the node r is labeled with $f(\overline{x_n})$, ii) the variables $\overline{x_n}$ point to nodes that evaluate (following the successor chain) to expressions which are more defined than $\overline{pv_n}$, and iii) the path starting at r ends in a node labeled with a value that is more defined than pv.

We now introduce a function to collect the program positions in the nodes that are reachable from the SC-node.

Definition 5.10 (reachable program positions). *Let G be an extended redex trail, $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ a slicing criterion, and r its associated SC-node. The set of reachable program positions of $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ in G is $Col(G, r, \pi)$, where function Col is defined in Figure 9.*

Intuitively, function Col traverses the extended redex trail from the SC-node as follows. First, the program positions of all nodes in the path from the SC-node to its computed value are collected. Once we reach the value of the SC-node, we continue inspecting the computation of values for the variable arguments according to the pattern π. Along the above paths, whenever we found a node labeled with an $(f)case$ expression, we also follow the subcomputation for the case arguments until a head normal form is obtained.

Example 5.11. *Consider the (normalized version) of the program in Example 3.1 and the slicing criterion*

$$\langle \texttt{minmax (Z : \_)},\ \texttt{Pair \_ Z},\ \texttt{1},\ \texttt{Pair} \perp \top\rangle$$

Then, we compute the redex trail of Fig. 4 (note that nodes labeled with let expressions are missing for readability, since they play no relevant role at this point). The associated SC-node is shown in Fig. 10 with a shadowed box, together with all reachable nodes and its associated program positions. The computed slice is the one already shown in Fig. 3.

The following theorem states the correctness and minimality of our dynamic backward slices.

Theorem 5.12 (correctness and minimality). *Let P be a program, $(C_0 \Longrightarrow^* C_m)$ a complete derivation, and G the graph in C_m. Let $\langle f(\overline{pv_n}), pv, l, \pi\rangle$ be a slicing criterion and r its associated SC-node. Then $Col(G, r, \pi)$ is a correct and minimal slice for P w.r.t. $\langle f(\overline{pv_n}), pv, l, \pi\rangle$.*

6. NOTES ON THE IMPLEMENTATION

An implementation of a debugging tool for the declarative multi-paradigm language Curry [10] has been undertaken. It combines both tracing and slicing by extending a previous tracer for Curry programs [4]. The extension of the tracing tool to additionally perform slicing has proceeded as follows. Firstly, we extended the meta-interpreter that generates the trail in order to include program positions. Program positions are added in such a way that they do not interfere with the tracing tool. The extended redex trail is stored in a file that is then loaded in by the viewer (the tool that shows the traces of Section 3). Secondly, we implemented a new function in the viewer that, given a slicing criterion, computes the reachable nodes from the associated SC-node and returns their program positions. We are currently working on the addition of a text editor to show the source program as well as the computed slices.

An interesting extension of this work consists in adapting our developments to *pure* functional languages. For instance, Haskell tracers (e.g., [18, 21]) usually rely on some program transformation to instrument a source program so that its execution—under the standard semantics—generates the associated redex trail. The inclusion of program positions within these approaches can also be done in a similar way as in our approach. Then, the implementation of a dynamic slicer mainly amounts to implement an algorithm to compute reachable nodes in a graph.

7. RELATED WORK

Despite the fact that the usefulness of slicing techniques is known for a long time in the imperative paradigm, there are very few approaches to program slicing in the context of declarative languages. In the following, we review the closest to our work. Reps and Turnidge [17] presents a static backward slicing algorithm for first-order strict functional programs. This work is mainly oriented to perform specialization by computing (static) slices for *part* of a function's output, which is specified by means of a projection. However, it only considers a *fixed* slicing criterion: the output of the whole program. Biswas [2, 3] considers a higher-order strict functional language and develops a technique for dynamic backward slicing based on labeling program expressions and, then, constructing an execution trace. Again, only a fixed slicing criterion is considered (although the extension to more flexible slicing criteria is discussed).

Liu and Stoller [12] define an algorithm for *static* backward slicing in a first-order strict functional language, where their main interest is the removal of dead code. Although the underlying semantics is different (lazy vs. strict), our slicing patterns are inspired by their *liveness patterns*.

Recently, Hallgren [7] presented—as a result of the Programatica project—the development of a slicing tool for Haskell. Although there is little information available yet, it seems to rely on the construction of a graph of functional dependences (i.e., only function names and their dependences are considered). Therefore, in principle, it should produce bigger—less precise—slices than our technique.

The closest approach is the forward slicing technique for lazy functional logic programs introduced by Vidal [20], where a partial evaluator is extended in order to perform slicing. Since partial evaluation naturally supports partial data structures, the distinction between static/dynamic slicing is not necessary (the same algorithm can be applied in both cases). This generality is a clear advantage, but also implies a loss of precision in general (i.e., the computed slices are not generally minimal). Moreover, only forward slicing is considered, which is less useful for debugging.

Therefore, our work can be considered as the first approach to dynamic backward slicing in the context of a modern lazy functional (logic) language like Curry (or Haskell).

[6]For simplicity, in the following we assume that, if there is no r such that $(x \rightsquigarrow r) \in G$, then $(x \rightsquigarrow \Box) \in G$.

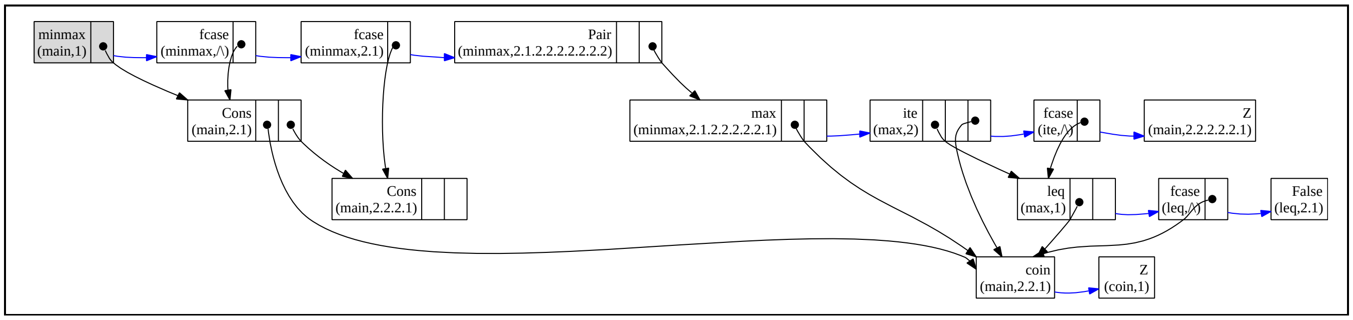

Figure 10: Reachable nodes for program `minmax` w.r.t. $\langle$`minmax (Z : _), Pair _ Z, 1, Pair` $\perp\ \top\rangle$

8. CONCLUSIONS AND FUTURE WORK

In this work, we have presented a new scheme to perform dynamic slicing in lazy functional logic programs which is based on redex trails. For this purpose, we have introduced a non-trivial notion of slicing criterion which is adequate in the context of lazy evaluation. We have also formalized the concept of dynamic backward slicing and have shown that forward slicing can be regarded as a particular case of backward slicing. We have defined the first (correct and minimal) dynamic slicing technique for the considered language and, finally, we have shown how tracing (based on redex trails) and slicing can be combined into a single framework.

There are several interesting topics for future work. On the one hand, we plan to define appropriate extensions in order to cover the additional features of Curry programs (higher-order, constraints, built-in functions, etc). On the other hand, we will investigate the definition of appropriate methods for the *static* slicing of functional logic programs.

9. REFERENCES

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Functional Logic Languages. *Electronic Notes in Theoretical Computer Science*, vol. 76 (Proc. of WFLP'02), 2002.

[2] S.K. Biswas. A Demand-Driven Set-Based Analysis. In *Proc. of ACM Symp. on Principles of Programming Languages*, pages 372–385. ACM Press, 1997.

[3] S.K. Biswas. *Dynamic Slicing in Higher-Order Programming Languages*. PhD thesis, Department of CIS, University of Pennsylvania, 1997.

[4] B. Braßel, M. Hanus, F. Huch, and G. Vidal. A Semantics for Tracing Declarative Multi-Paradigm Programs. In *Proc. of the 6th Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'04)*. ACM Press, 2004. *To appear.*

[5] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

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

[7] T. Hallgren. Haskell Tools from the Programatica Project. In *Proc. of the ACM Workshop on Haskell (Haskell'03)*, pages 103–106. ACM Press, 2003.

[8] 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.

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

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

[11] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[12] Y.A. Liu and S.D. Stoller. Eliminating Dead Code on Recursive Data. *Science of Computer Programming*, 47:221–242, 2003.

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

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

[15] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. CUP, 2003.

[16] B. Pope and L. Naish. A Program Transformation for Debugging Haskell 98. In *Proc. of ACSC 2003*, volume 16 of *Conferences in Research and Practice in Information Technology*, pages 227–236. ACS, 2003.

[17] T. Reps and T. Turnidge. Program Specialization via Program Slicing. In *Partial Evaluation. Dagstuhl Castle, Germany*, pages 409–429. Springer LNCS 1110, 1996.

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

[19] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[20] G. Vidal. Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation. In *Logic-based Program Synthesis and Transformation, LOPSTR'02*, pp. 219-237. Springer LNCS 2664, 2003.

[21] 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.

[22] M.D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.