Program Specialization Based on Dynamic Slicing

Claudio Ochoa[1], Josep Silva[1], and Germán Vidal[1]

DSIC, Tech. University of Valencia, Camino de Vera s/n, E-46022 Valencia, Spain.
{cochoa,jsilva,gvidal}@dsic.upv.es

Abstract. Within the imperative programming paradigm, program slicing has been widely used as a basis to solve many software engineering problems, like program understanding, debugging, testing, differencing, specialization, and merging. In this work, we present a lightweight approach to program slicing in lazy functional logic languages and discuss its potential applications in the context of pervasive systems where resources are limited. In particular, we show how program slicing can be used to achieve a form of program specialization that cannot be achieved with other, related techniques like partial evaluation.

1 Introduction

Essentially, program slicing is a method for decomposing programs by analyzing their data and control flow. It was first proposed as a debugging tool to allow a better understanding of the portion of code which revealed an error. Since this concept was originally introduced by Weiser [36, 37]—in the context of imperative programs—it has been successfully applied to a wide variety of software engineering tasks (e.g., program understanding, debugging, testing, differencing, specialization, and merging). Surprisingly, there are very few approaches to program slicing in the context of *declarative* programming; some notable exceptions are, e.g., [10, 24, 27–29, 31, 35].

Intuitively 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* [9, 22] that makes explicit both the data and control dependences for each operation in a program. Program dependences can be traversed backwards and forwards—from the slicing criterion—giving rise to so-called *backward* and *forward* slicing, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. *Quasi* static slicing was the first attempt to define a hybrid method ranging between static and dynamic slicing [33]. It becomes useful when only the value of some parameters is known. All approaches to slicing mentioned so far are *syntax preserving*, i.e., they are mainly obtained from the original program by statement deletion. In contrast, *amorphous* slicing [17] exploits different program transformations in order to simplify the program while preserving its semantics w.r.t. the slicing criterion. More detailed information on program slicing can be found in the surveys of Harman and Hierons [18] and Tip [32].

In this work, we propose a lightweight approach to program slicing in the context of *lazy* functional logic languages like Curry [13, 15] and Toy [26]. These languages combine the most important features of functional *and* logic languages, e.g., lazy evaluation and non-determinism (see [12] for a survey). In particular, we propose a novel method for *program specialization* which is based on dynamic slicing. Our aim is similar to that of Reps and Turnidge [28], who designed a program specialization method for *strict* functional programs based on *static* slicing. In contrast, we consider *lazy* functional logic programs and our technique is based on *dynamic* slicing. We consider dynamic slicing since it is simpler and more accurate than static slicing (i.e., it has been shown that the dynamic slices can be considerably smaller than static slices [19, 34]). Furthermore, a dynamic slicing technique for the considered programs is already available, which allows us to reuse previous developments.

Our specialization method proceeds basically as follows. First, the user identifies a representative set of *slicing criteria*. Then, by using the dynamic slicing technique of [27], we obtain the locations, in the program, of the expressions whose evaluation is needed w.r.t. the considered slicing criteria. We note that [27] defines a program slice as a set of *program positions* rather than as an executable program, since this is sufficient for debugging, the aim of [27]. Therefore, we introduce an algorithm to extract an executable program slice—the specialized program—from the computed program positions.

$$
\begin{array}{rrll}
 & P & ::= D_1 \ldots D_m & \\
 & D & ::= f(x_1,\ldots,x_n) = e & \\
Exp \ni & e & ::= x & \text{(variable)} \\
 & & \mid\ c(x_1,\ldots,x_n) & \text{(constructor call)} \\
 & & \mid\ f(x_1,\ldots,x_n) & \text{(function call)} \\
 & & \mid\ let\ x = e_1\ in\ e_2 & \text{(let binding)} \\
 & & \mid\ e_1\ or\ e_2 & \text{(disjunction)} \\
 & & \mid\ case\ x\ of\ \{\overline{p_n \to e_n}\} & \text{(rigid case)} \\
 & & \mid\ fcase\ x\ of\ \{\overline{p_n \to e_n}\} & \text{(flexible case)} \\
 & p & ::= c(x_1,\ldots,x_n) & \text{(flat pattern)}
\end{array}
\qquad
\begin{array}{rcll}
\mathcal{X} & = & \{x,y,z,\ldots\} & \text{(variables)} \\
\mathcal{C} & = & \{a,b,c,\ldots\} & \text{(constructors)} \\
\mathcal{F} & = & \{f,g,h,\ldots\} & \text{(defined functions)}
\end{array}
$$

Fig. 1. Syntax for normalized flat programs

Although dynamic slicing has been mainly applied to debugging, its potential applications extends well beyond this technique. Apart from specialization, the aim of our work, other potential applications include carrying out dependence based software testing [8, 21], measuring module cohesion for purpose of code restructuring [11], and guiding the development of performance enhancing transformations based upon estimation of criticality of instructions [38]. Another interesting application of dynamic slicing is the detection and removal of *dead code* (see, e.g., [20, 25, 28]). In the literature, *dead code* usually refers either to pieces of code that never get executed or to code that gets executed without making any contribution to the final output [1]. Under a lazy evaluation model, expressions are only evaluated if their results are actually needed [4]. Therefore, in this setting, dead code *only* refers to code that is never executed (noted also in [6]).

Our approach is simple and easy to implement since dynamic slicing is already available for the considered language [27]. The main novelty in this work is the use of dynamic slicing for program specialization (rather than debugging), which can be useful in the context of pervasive systems in order to produce smaller programs. We also introduce an appropriate definition to extract executable slices from a set of program positions.

This paper is organized as follows. In the next section, we recall the syntax of a simple lazy functional logic language. Section 3 presents an informal introduction to program specialization by dynamic slicing. Section 4 formalizes an instrumented semantics which also stores the location of each reduced expression. Section 5 defines the main concepts involved in dynamic slicing and, finally, Section 6 concludes and points out several directions for further research.

2 The Language

In this work, we consider *flat* programs [14], a convenient standard representation for functional logic programs which makes explicit the pattern matching strategy by the use of case expressions. The flat representation constitutes the kernel of modern declarative multi-paradigm languages like Curry [13, 15] and Toy [26]. 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). As in [23], this is essential to express *sharing* without the use of complex graph structures. A simple normalization algorithm can be found in [2]. 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 "$let\ x = e\ in\ f(x)$."

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 form:[1]

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

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

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 corresponding variables of the subexpression 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* [5] (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., [14]).

The higher-order features of current functional languages can be reduced to first-order definitions by considering the distinguished function *apply*. For instance, an expression like "$(f\ x)\ y$" can be translated to $(f@x)@y$ where the definition of "@" contains the following rules for the binary function f (this is used, e.g., in the implementation of [3]):

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

In order to avoid the generation of these rules for all program functions, an explicit definition of "@" based on the primitive functions *apply* and *partcall* is usually considered [16]. For example, the previous expression "$(f\ x)\ y$" is translated to "$\mathit{apply}(\mathit{apply}(\mathit{partcall}(f), x), y)$" and then reduced by the rules of the operational semantics. However, the tracing technique of [7] only considers *first-order* flat programs. Therefore, we restrict our developments to first-order flat programs too. Nevertheless, extending our technique to flat programs with *partcall* and *apply* would not be difficult.

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 "$\mathit{let}\ x = x\ \mathit{in}\ e$". In this paper, we call such variables which are bound to themselves *logical variables.*

3 An Overview of the Specialization Process

In this section, we present an informal overview of the specialization process based on dynamic backward slicing. Similarly to [28], we use an example inspired in a functional version of the well-known Unix word-count utility, and specialize it to count *only* characters. This is a kind of specialization that cannot be achieved by standard partial evaluation. Consider the program shown in Figure 2 where data structures are built from:[2]

```
data Nat   = Z | S Nat            data Letter = A | B | CR
data Pairs = Pair Nat Nat         data String = Nil | Cons Letter String
```

As it is common practice in functional languages, we often use "`[]`" and "`:`" as a shorthand for `Nil` and `Cons`. Observe that we consider a maximally simplified alphabet for simplicity.

Figure 2 includes a function, `lineCharCount`, to count the number of lines and characters in a string (by using two accumulating parameters, `lc` and `cc`, to build up the line and character counts as it travels down the string `str`). Function `ite` is a simple conditional while `eq` is an equality test on letters. If we look at function `main`,[3] it should be clear that only the number of characters is computed. In general, however, the execution of the function call

```
lineCharCount [letter,CR]
```

[2] In the 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.

[3] We assume that computations always start from the distinguished function `main` which has no arguments.

```
main = printCharCount (lineCharCount [letter,CR])

printLineCount t = fcase t of {Pair x y -> printNat x}
printCharCount t = fcase t of {Pair x y -> printNat y}

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

lineCharCount str = lcc str Z Z

lcc str lc cc = fcase str of {[]     -> Pair lc cc;
                              (s:ss) -> ite (eq s CR) (lcc ss (S lc) (S cc))
                                                      (lcc ss lc (S cc))    }
letter = A or B or CR

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

eq x y = fcase x of {A  -> fcase y of {A  -> True;  B  -> False; CR -> False};
                     B  -> fcase y of {A  -> False; B  -> True;  CR -> False};
                     CR -> fcase y of {A  -> False; B  -> False; CR -> True } }
```

Fig. 2. Example program `lineCharCount`

returns the following three results (due to the non-deterministic function `letter`):

(`Pair 1 2`)	if `letter` reduces to `A`
(`Pair 1 2`)	if `letter` reduces to `B`
(`Pair 2 2`)	if `letter` reduces to `CR`

Now, our aim is the specialization of this program in order to extract those statements which are actually needed to compute the number of characters in a string. For this purpose, we consider the following set of slicing criteria:

$\langle$`lineCharCount` [`A`, `CR`], `Pair 1 2`, 1, `Pair` $\perp$ $\top\rangle$
$\langle$`lineCharCount` [`B`, `CR`], `Pair 1 2`, 1, `Pair` $\perp$ $\top\rangle$
$\langle$`lineCharCount` [`CR`, `CR`], `Pair 2 2`, 1, `Pair` $\perp$ $\top\rangle$

Following [27], the first component of each slicing criterion is a function call whose arguments appear fully evaluated, i.e., as much as needed in the considered computation. The second and third components are required in order to uniquely identify a concrete computation by providing a result of the function call as well as the occurrence of this result (since a function call could produce, non-deterministically, the same value several times). Finally, the fourth component is a *slicing pattern* which is used to denote the parts of the computed value we are interested in: $\top$ means that the computation of the corresponding subexpression is of interest while $\perp$ means that it can be ignored. We note that our patterns for slicing are similar to the *liveness patterns* used by Liu and Stoller [25] to perform dead code elimination.

Despite the apparent complexity of our notion of slicing criterion, all components are easily accessible from a tracing tool like the one presented in [7]. Indeed, the dynamic slicing technique of [27] is built on top of this tracer and relies on the computation of a *redex trail* [30], 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. For instance, Figure 3 shows the redex trail of the first computation—starting from `main`—where function `letter` is reduced to `A`, i.e., the computation identified by the first slicing criterion. Here, the nodes of the trail which are related to the pattern of this slicing criterion are shadowed. Then, a backward slice w.r.t. the first criterion must include all the expressions of the original program which appear in a shadowed node. For this purpose, [27] extends the redex trail model in order to also store the program position (i.e., the location in the source program) of each expression in a node of the graph.

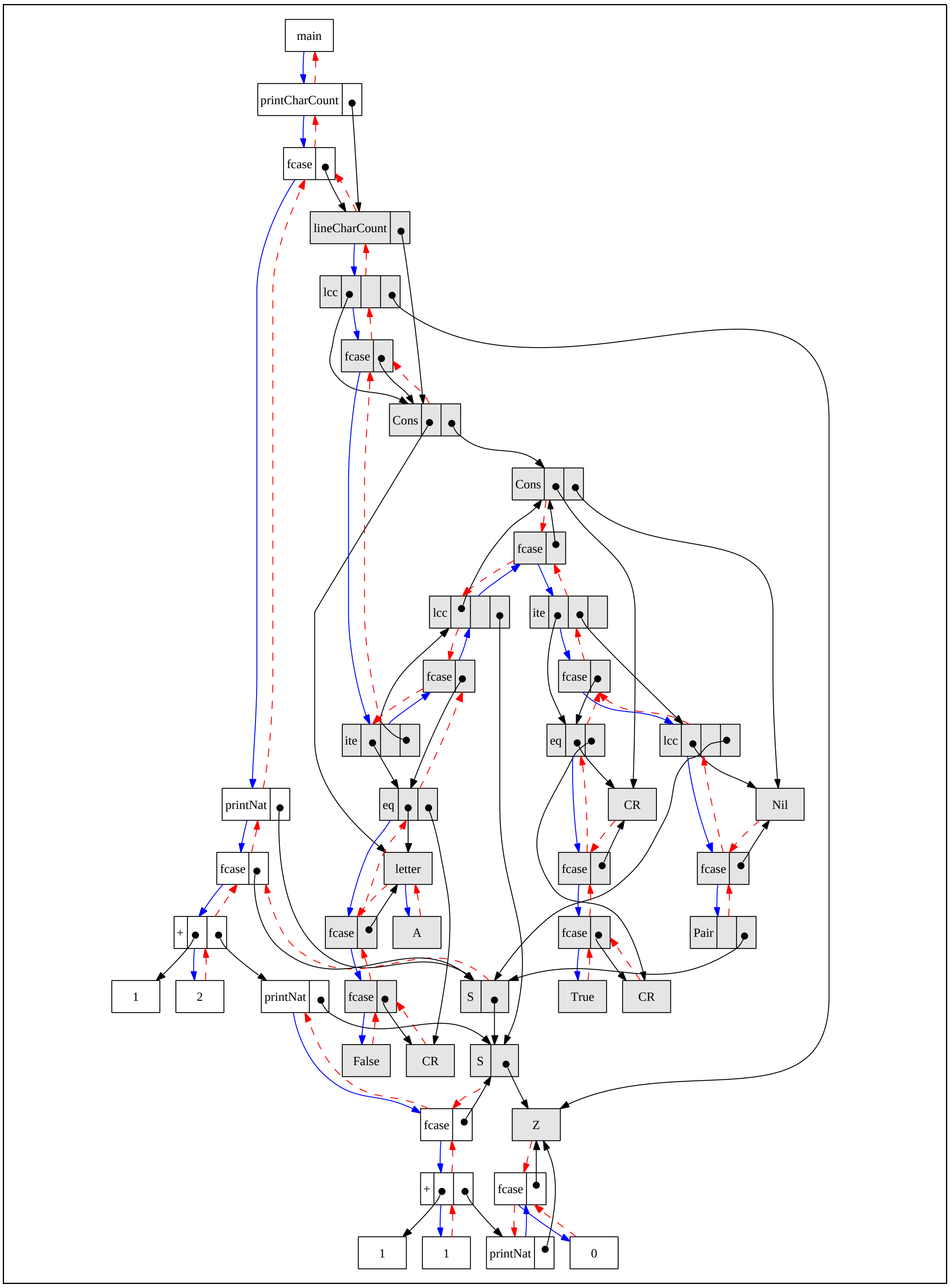

Fig. 3. Redex trail for program `lineCharCount`

```
lineCharCount str = lcc str ? Z

lcc str lc cc = fcase str of {[]     -> Pair ? cc;
                              (s:ss) -> ite (eq s CR) (lcc ss ? (S cc))
                                                      (lcc ss ? (S cc))    }
letter = A or B or CR

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

eq x y = fcase x of {A  -> fcase y of {A  -> True;  B  -> False; CR -> False};
                     B  -> fcase y of {A  -> False; B  -> True;  CR -> False};
                     CR -> fcase y of {A  -> False; B  -> False; CR -> True } }
```

Fig. 4. Specialization of program `lineCharCount`

Clearly, producing a specialized program for a particular computation is not always appropriate. Reps and Turnidge [28] considers *static* slicing and, thus, no input data are provided, only the slicing pattern. Here, we use dynamic slicing but consider a *representative* set of slicing criteria covering the interesting computations. Basically, we proceed as follows:

– For each slicing criterion, we compute the set of program positions that are related to this slicing criterion. The tracing tool of [7] can be used to help the user to easily identify the interesting slicing criteria, while the technique of [27] can be used to compute the sets of program positions, i.e., the *slices* in the terminology of [27].
– Then, we extract an executable slice from the original program and the union of the program positions related to all slicing criteria. Essentially, functions that were not used in any computation are completely deleted. Case branches that were not selected in any computation are also deleted. For the remaining expressions, we follow a simple rule: if their position belongs to the computed set, it appears in the slice; otherwise, it is replaced by a new constant symbol "?". The meaning of this symbol is not relevant since it will never be executed in the considered computations (it is only used to point out that some subexpression is missing due to the slicing process). We will discuss this point further in Section 5.

The final program that results from slicing `lineCharCount` w.r.t. the three slicing criteria discussed above is shown in Figure 4. Observe that two further simplifications are obvious: deleting the second argument of function `lcc` and replacing the call to

`ite (eq s CR) (lcc ss ? (S cc)) (lcc ss ? (S cc))`

by (`lcc ss ? (S cc)`), since both branches of the conditional statement are identical. This is, however, not allowed in traditional approaches to slicing where the structure of the original program should be preserved through the transformation. However, if amorphous slicing is considered—which is subject of ongoing work—the specialized program could be reduced to only three functions:

```
lineCharCount str = lcc str Z

lcc str cc = fcase str of {[]     -> Pair ? cc;
                           (s:ss) -> lcc ss (S cc)}
letter = A or B or CR
```

The next sections formalize the notion of dynamic slice—as a set of program positions—as well as the extraction of executable slices.

4 Extending the Semantics with Program Positions

In this section, we present an extension of the (small-step) operational semantics for lazy functional logic programs [2] in order to also compute *program positions*. This information will become useful to identify the location, in the program, of each reduced expression in a computation.

Rule	*Heap*	*Control*	*Stack*	*Fun*	*Pos*
varcons	$\Gamma[x \mapsto_{(h,w')} t]$	x	S	g	w
	$\Longrightarrow \Gamma[x \mapsto_{(h,w')} t]$	t	S	h	w'
varexp	$\Gamma[x \mapsto_{(h,w')} e]$	x	S	g	w
	$\Longrightarrow \Gamma[x \mapsto_{(h,w')} e]$	e	$x : S$	h	w'
val	Γ	v	$x : S$	g	w
	$\Longrightarrow \Gamma[x \mapsto_{(g,w)} v]$	v	S	g	w
fun	Γ	$f(\overline{x_n})$	S	g	w
	$\Longrightarrow \Gamma$	$\rho(e)$	S	f	Λ
let	Γ	$let\ x = e_1\ in\ e_2$	S	g	w
	$\Longrightarrow \Gamma[y \mapsto_{(g,w.1)} \rho(e_1)]$	$\rho(e_2)$	S	g	$w.2$
or	Γ	$e_1\ or\ e_2$	S	g	w
	$\Longrightarrow \Gamma$	e_i	S	g	$w.i$
case	Γ	$(f)case\ x\ of\ \{\overline{p_k \to e_k}\}$	S	g	w
	$\Longrightarrow \Gamma$	x	$((f)\{\overline{p_k \to e_k}\}, g, w) : S$	g	$w.1$
select	Γ	$c(\overline{y_n})$	$((f)\{\overline{p_k \to e_k}\}, h, w') : S$	g	w
	$\Longrightarrow \Gamma$	$\rho(e_i)$	S	h	$w'.2.i$
guess	$\Gamma[y \mapsto_{(g,w)} y]$	y	$(f\{\overline{p_k \to e_k}\}, h, w') : S$	g	w
	$\Longrightarrow \Gamma[y \mapsto_{(\_,\_)} \rho(p_i), \overline{y_n \mapsto_{(\_,\_)} y_n}]$	$\rho(e_i)$	S	h	$w'.2.i$

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:	$i \in \{1, \ldots k\},\ p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$
guess:	$i \in \{1, \ldots k\},\ p_i = c(\overline{x_n}),\ \rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are fresh

Fig. 5. Small-step instrumented semantics

In order to keep the paper self-contained, in the following we present the instrumented semantics, including also our extension to compute program positions. The instrumented operational semantics is shown in Figure 5. We distinguish two components in this semantics: the first component (columns *Heap*, *Control*, and *Stack*) defines the standard semantics introduced in [2]; the second component (columns *Fun* and *Pos*) is used to store *program positions*, i.e., a function name and a position (within this function) for the current expression in the control. Our semantics obeys the following naming conventions:

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

A *heap* is a partial mapping from variables to expressions. Each mapping $x \mapsto_{(g,w)} e$ is labeled with the program position (g, w) of expression e (see below). The *empty heap* is denoted by `[]`. The value associated to variable x in heap Γ is denoted by $\Gamma[x]$. $\Gamma[x \mapsto_{(g,w)} e]$ 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 1 (configuration). *A configuration of the semantics is a tuple $\langle \Gamma, e, S, g, w \rangle$ where $\Gamma \in$ Heap is the current heap, $e \in$ Exp is the expression to be evaluated (called the control of the semantics), S is the stack, and g and w denote the program position of expression e.*

Program positions uniquely determine the location, in the program, of each reduced expression. This notion is formalized as follows:

Definition 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:

$$\begin{array}{rlrl}
e|_{\Lambda} = e & (e \in Exp) & let\ x = e\ in\ e'|_{1.w} = e|_w & \\
c(\overline{x_n})|_{i.w} = x_i|_w & (i \in \{1,\ldots,n\}) & let\ x = e\ in\ e'|_{2.w} = e'|_w & \\
f(\overline{x_n})|_{i.w} = x_i|_w & (i \in \{1,\ldots,n\}) & (f)case\ x\ of\ \{\overline{p_n \to e_n}\}|_{1.w} = x|_w & \\
e_1\ or\ e_2|_{i.w} = e_i|_w & (i \in \{1,2\}) & (f)case\ x\ of\ \{\overline{p_n \to e_n}\}|_{2.i.w} = e_i|_w & (i \in \{1,\ldots,n\})
\end{array}$$

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.

Let us note that not all subexpressions are addressed by program positions (this is the case, e.g., of the variables in a let construct or the patterns in a case expression). Only those expressions which are relevant for the slicing process can be addressed.

A brief explanation for each rule of the extended semantics follows:

(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. Also, in the derived configuration, we update the current program position with that of the expression in the heap, (h,w'), which was previously introduced by rules val or let (see below).

(varexp) 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 to the stack the variable x. As in rule varcons, the derived configuration is updated with the program position of the expression in the heap.

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

(fun) This rule performs a simple function unfolding; here, we assume that the considered program P is a global parameter of the calculus. Trivially, we reset the program position to (f,Λ) since the control of the derived configuration is the complete right-hand side of function f.

(let) In order to reduce a let construct, this rule 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. 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 in rules varcons and varexp to recover the program position of the binding. The program position in the new configuration is updated to $(g, w.2)$ to address the expression e_2 of the *let*.

(or) This rule *non-deterministically* evaluates a disjunction by either evaluating the first or the second argument. Clearly, the program position of the derived configuration is $(g, w.i)$ where $i \in \{1,2\}$ refers to the selected disjunct.

(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 program position. 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. In both rules, select and guess, the program position of the case expression (stored in the stack by rule case) is used to properly set the program position of the derived configuration.

Clearly, the instrumented semantics is a conservative extension of the original small-step semantics of [2], since the last two 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 3 (initial configuration).
An initial configuration has the form $\langle$ `[]`, `main`, `[]`, _, _ $\rangle$, where (_, _) denotes a null program position (since there is no call to `main` *from the right-hand side of any function definition).*

Definition 4 (final configuration).
A final configuration has the form: $\langle \Delta, v, [\,], g, w\rangle$, where v is a value, Δ is a heap containing the computed bindings, and (g, w) is a program position.

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.

As an example, consider again functions `eq` and `letter` of program `lineCharCount`, together with the initial call "`eq letter CR`". The normalized flat program is as follows (the program positions of some selected expressions are shown in the right column):[4]

```
main = let x1 = letter                        [let : (main,Λ), letter : (main,1)]
       in let x2 = CR                               [let : (main,2), CR : (main,2.1)]
       in eq x1 x2                                          [eq x1 x2 : (main,2.2)]

letter = A or (B or CR)          [A : (letter,1), B : (letter,2.1), CR : (letter,2.2)]

eq x y = fcase x of                                               [fcase : (eq,Λ)]
              {A   -> fcase y of                                 [fcase : (eq,2.1)]
                         {A  -> True;                        [True : (eq,2.1.2.1)]
                          B  -> False;                      [False : (eq,2.1.2.2)]
                          CR -> False}                      [False : (eq,2.1.2.3)]
               B   -> fcase y of                                 [fcase : (eq,2.2)]
                         {A  -> False;                      [False : (eq,2.2.2.1)]
                          B  -> True;                        [True : (eq,2.2.2.2)]
                          CR -> False}                      [False : (eq,2.2.2.3)]
               CR -> fcase y of                                  [fcase : (eq,2.3)]
                         {A  -> False;                      [False : (eq,2.3.2.1)]
                          B  -> False;                      [False : (eq,2.3.2.2)]
                          CR -> True}}                       [True : (eq,2.3.2.3)]
```

A complete computation—the one which evaluates `letter` to `A`—with the rules of Figure 5 is shown in Figure 6. For clarity, each computation step is labeled with the applied rule.

The next theorem states the correctness of the computed program positions.

Theorem 1 (program positions).
Let P be a program and $(C_0 \Longrightarrow^* C_k)$, $k > 0$, a complete derivation in P. For each configuration $C_i = \langle \Gamma, e', S, g, w\rangle$, either $(g = w = \_)$ or $g(\overline{x_n}) = e \in P$ with $e|_w = e'$ (up to variable renaming).

5 Extracting an Executable Slice

In this section, we formalize the concept of dynamic backward slice based on the computations performed by the extended operational semantics. Following [27], a dynamic slice is defined as a set of program positions:

Definition 5 (dynamic slice).
Let P be a program. A dynamic slice for P is a set $\mathcal{W}$ of program positions such that $\mathcal{W} \subseteq \mathcal{P}os(P)$.

The above notion of dynamic slice is useful in order to easily compute the differences between several slices—i.e., the intersection of the corresponding program positions—as well as the merging of slices—i.e., the union of the corresponding program positions. Furthermore, executable slices for several slicing criteria can easily be obtained from the original program and the computed set of program positions, as we will see later.

As discussed in Section 3, an appropriate slicing criterion in our setting is given by a tuple $\langle f(\overline{pv_n}), pv, l, \pi\rangle$, where $f(\overline{pv_n})$ is a function call whose arguments are evaluated to $\overline{pv_n}$ in the considered computation, pv is the output of this function call, l is the occurrence of a function call $f(\overline{v_n})$ that outputs v in the considered computation (which is necessary due to non-deterministic functions), and π is a slicing pattern that determines the interesting part of the computed value.

[4] Recall that these functions did not appear normalized in program `lineCharCount`.

```
         ⟨[], main, [], _, _⟩
⟹fun     ⟨[], let x1 = letter in let x2 = CR in eq x1 x2, [], main, Λ⟩
⟹let     ⟨[x1 ↦(main,1) letter], let x2 = CR in eq x1 x2, [], main, 2⟩
⟹let     ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], eq x1 x2, [], main, 2.2⟩
⟹fun     ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], fcase x1 of {A → fcase ...;
         B → fcase ...; CR → fcase ...}, [], eq, Λ⟩
⟹case    ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], x1,
         [(f{A → fcase ...; B → fcase ...; CR → fcase ...}, (eq,Λ))], eq, 1⟩
⟹varexp  ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], letter,
         [x1,(f{A → fcase ...; B → fcase ...; CR → fcase ...}, (eq,Λ))], main, 1⟩
⟹fun     ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], A or (B or CR),
         [x1,(f{A → fcase ...; B → fcase ...; CR → fcase ...}, (eq,Λ))], letter, Λ⟩
⟹or      ⟨[x1 ↦(main,1) letter, x2 ↦(main,2.1) CR], A,
         [x1,(f{A → fcase ...; B → fcase ...; CR → fcase ...}, (eq,Λ))], letter, 1⟩
⟹varexp  ⟨[x1 ↦(letter,1) A, x2 ↦(main,2.1) CR], A,
         [(f{A → fcase ...; B→ fcase ...; CR → fcase ...}, (eq,Λ))], letter, 1⟩
⟹select  ⟨[x1 ↦(letter,1) A, x2 ↦(main,2.1) CR], fcase x2 of {A → True; B → False; CR → False},
         [], eq, 2.1⟩
⟹case    ⟨[x1 ↦(letter,1) A, x2 ↦(main,2.1) CR], x2,
         [(f{A → True; B → False; CR → False}, (eq,2.1))], eq, 2.1.1 ⟩
⟹varcons ⟨[x1 ↦(letter,1) A, x2 ↦(main,2.1) CR], CR,
         [(f{A → True; B → False; CR → False}, (eq,2.1))], main, 2.1 ⟩
⟹select  ⟨[x1 ↦(letter,1) A, x2 ↦(main,2.1) CR], False,  [], eq, 2.1.2.3⟩
```

Fig. 6. An example computation with the instrumented semantics

Definition 6 (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.

As discussed in [27], 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 7 (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 $\sqsubseteq :: (PValue \times PValue \rightarrow Bool)$ is defined as follows:

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

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

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:

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

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 $C = \langle \Gamma, e, S, g, w\rangle$, we let $heap(C) = \Gamma$, $control(C) = e$, and $stack(C) = S$.

Definition 8 (SC-configuration).
Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation in P. Given a slicing criterion of the form $\langle f(\overline{pv_n}), pv, l, \pi\rangle$, 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) *$\nexists k$ such that $i < k < j$, $stack(C_i) = stack(C_k)$, and $control(C_k)$ is a value.*

Here, function $val :: (Heap \times Exp \rightarrow PValue)$ 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 a 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:

- the control of C is $f(\overline{x_n})$,
- the variables $\overline{x_n}$ are bound in the computation to expressions which are more defined than $\overline{pv_n}$, and
- 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 result of the evaluation 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 9 (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 7.

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

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

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

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

Fig. 7. Auxiliary function *contr* to collect contributing configurations

Definition 10 (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 $\langle \Gamma, e, S, g, w\rangle \in contr(C_i, stack(C_i), \pi, \{\})$, 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 11 (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 exists a configuration $\langle \Gamma, e, S, g, w\rangle \in contr(C_i, stack(C_i), \pi, \{\})$.

In [27], we present a dynamic slicing technique—which is based on redex trails—that computes correct and minimal slices. Since the aim of [27] is to extend tracing-based debugging, the computed program positions are sufficient to highlighted the relevant expressions of the original program which are related to a particular slicing criterion. In the following, we introduce a method to extract an *executable* program slice from the computed program positions.

Definition 12 (executable slice).
Let P be a program and $\mathcal{W}$ an associated slice. The corresponding executable slice, $P_{\mathcal{W}}$, is obtained as follows:

$$P_{\mathcal{W}} = \{ f(\overline{x_n}) = [\![e]\!]^{\Lambda}_{\mathcal{Q}} \mid (f,\Lambda) \in \mathcal{W} \textit{ and } \mathcal{Q} = \{p \mid (f,p) \in \mathcal{W}\}\ \}$$

Auxiliary function $[\![\ ]\!]^p_{\mathcal{Q}}$ is defined inductively as follows:

$$\begin{array}{ll} [\![x]\!]^p_{\mathcal{Q}} & = x \\ [\![\varphi(x_1,\ldots,x_n)]\!]^p_{\mathcal{Q}} & = \begin{cases} \varphi(x_1,\ldots,x_n) & \textit{if } p\in\mathcal{Q} \\ ? & \textit{if } p\notin\mathcal{Q} \end{cases} \qquad\qquad \textit{where } \varphi \in (\mathcal{C}\cup\mathcal{F}) \\ [\![\textit{let } x = e' \textit{ in } e]\!]^p_{\mathcal{Q}} & = \begin{cases} \textit{let } x = [\![e']\!]^{p.1}_{\mathcal{Q}} \textit{ in } [\![e]\!]^{p.2}_{\mathcal{Q}} & \textit{if } p\in\mathcal{Q} \\ ? & \textit{if } p\notin\mathcal{Q} \end{cases} \\ [\![e_1 \textit{ or } e_2]\!]^p_{\mathcal{Q}} & = \begin{cases} [\![e_1]\!]^{p.1}_{\mathcal{Q}} \textit{ or } [\![e_2]\!]^{p.2}_{\mathcal{Q}} & \textit{if } p\in\mathcal{Q} \\ ? & \textit{if } p\notin\mathcal{Q} \end{cases} \\ [\![(f)\textit{case } x \textit{ of } \{\overline{p_k \to e_k}\}]\!]^p_{\mathcal{Q}} & = \begin{cases} (f)\textit{case } x \textit{ of } \{\overline{p_k \to [\![e_k]\!]^{p.2.k}_{\mathcal{Q}}}\} & \textit{if } p\in\mathcal{Q} \\ ? & \textit{if } p\notin\mathcal{Q} \end{cases} \end{array}$$

The slice computed in this way is trivially executable by considering ? as a fresh 0-ary constructor symbol. However, from an implementation point of view, we should also extend all program types in order to also include ?. While this is mandatory in a pure functional language, we have a simpler solution in our setting: consider ? as a fresh (existential) variable (i.e., like the anonymous variable in Prolog). This is very easy to implement in Curry by just adding a `where`-declaration to each function with some fresh variables. For instance,

```
lineCharCount str  =  lcc str ? Z
```

is replaced by

```
lineCharCount str  =  lcc str x Z  where x free
```

On the other hand, there are some post-processing simplifications that can be added in order to reduce the program size and improve its readability:

$$\begin{array}{rl} \textit{let}\ x = ?\ \textit{in}\ e \Longrightarrow \rho(e) & \text{where } \rho = \{x \mapsto ?\} \\ (f)\textit{case}\ x\ \textit{of}\ \{\overline{p_k \to e_k}\} \Longrightarrow (f)\textit{case}\ x\ \textit{of}\ \{\overline{p'_m \to e'_m}\} & \text{where } \{\overline{p'_m \to e'_m}\} = \{p_i \to e_i \mid e_i \neq ?\} \end{array}$$

More powerful post-processing simplifications could be added—like the ones discussed in Section 3—but they require a form of *amorphous* slicing, which is subject of ongoing work.

6 Conclusions and Future Work

In this work, we have presented a lightweight approach to program specialization which is based on dynamic slicing. Our method achieves a kind of specialization that cannot be achieved with other, related techniques like partial evaluation. Our approach is simple and easy to implement since a dynamic slicing technique already exists for the considered programs.

As a future work, we plan to extend the current method in order to perform more aggressive simplifications (i.e., a form of amorphous slicing). This extended form of slicing can be very useful in the context of pervasive systems where resources are limited and, thus, code size should be reduced as much as possible. Another promising topic for future work is the definition of program specialization based on *static* slicing rather than on *dynamic* slicing. In particular, it would be interesting to establish the strengths and weaknesses of each approach.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.
2. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Functional Logic Languages. In *Proc. of the Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, volume 76 of *Electronic Notes in Theoretical Computer Science*, 2002.
3. S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the Int'l Workshop on Frontiers of Combining Systems*, pages 171–185. Springer LNCS 1794, 2000.
4. Z.M. Ariola, J. Maraist, M. Odersky, M. Felleisen, and P. Wadler. A Call-By-Need Lambda Calculus. In *Proc. of the 22nd ACM Symp. on Principles of Programming Languages (POPL'95)*, pages 233–246. ACM Press, 1995.
5. H.P. Barendregt. *The Lambda Calculus—Its Syntax and Semantics.* Elsevier, 1984.
6. S.K. Biswas. A Demand-Driven Set-Based Analysis. In *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 372–385. ACM Press, 1997.
7. 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.* Available from URL: `www.dsic.upv.es/users/elp/german/papers.html`.
8. E. Duesterwald, R. Gupta, and M.L. Soffa. Rigorous Data Flow Testing through Output Influences. In *Proc. of the 2nd Irvine Software Symposium*, pages 131–145. UC Irvine, CA, 1992.
9. 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.

10. V. Gouranton. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. In *Proc. of the Int'l Static Analysis Symposium (SAS'98)*, pages 115–133, 1998.
11. R. Gupta and P. Rao. Program Execution Based Module Cohesion Measurement. In *Proc. of the 16th IEEE Int'l Conf. on Automated Software Engineering (ASE'01)*, pages 144–153. IEEE Press, 2001.
12. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
13. 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.
14. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
15. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~curry/`.
16. M. Hanus (ed.), S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.6: The Portland Aachen Kiel Curry System—User Manual. Technical report, U. Kiel, Germany, 2004.
17. M. Harman and S. Danicic. Amorphous Program Slicing. In *Proc. of the 5th Int'l Workshop on Program Comprehension*. IEEE Computer Society Press, 1997.
18. M. Harman and R.M. Hierons. An Overview of Program Slicing. *Software Focus*, 2(3):85–92, 2001.
19. T. Hoffner, M. Kamkar, and P. Fritzson. Evaluation of Program Slicing Tools. In *In Proc. of the 2nd Int'l Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, pages 51–69. IRISA-CNRS, 1995.
20. S.B. Jones and D. Le Métayer. Compile-Time Garbage Collection by Sharing Analysis. In *Proc. Int'l Conf. on Functional Programming Languages and Computer Architecture*, pages 54–74. ACM Press, 1989.
21. M. Kamkar. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Linöping University, 1993.
22. D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimization. In *Proc. of the 8th Symp. on the Principles of Programming Languages (POPL'81), SIGPLAN Notices*, pages 207–218, 1981.
23. 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.
24. M. Leuschel and M.H. Sørensen. Redundant Argument Filtering of Logic Programs. In *Proc. of LOPSTR'96*, pages 83–103. LNCS 1207 83–103, 1996.
25. Y.A. Liu and S.D. Stoller. Eliminating Dead Code on Recursive Data. *Science of Computer Programming*, 47:221–242, 2003.
26. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
27. C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of the Int'l Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'04)*. ACM Press, 2004. To appear. Available from URL: `www.dsic.upv.es/users/elp/german/papers.html`.
28. T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, pages 409–429. Springer LNCS 1110, 1996.
29. S. Schoenig and M. Ducasse. A Backward Slicing Algorithm for Prolog. In *Proc. of the Int'l Static Analysis Symposium (SAS'96)*, pages 317–331. Springer LNCS 1145, 1996.
30. 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.
31. G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.
32. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
33. G.A. Venkatesh. The Semantic Approach to Program Slicing. *SIGPLAN Notices*, 26(6):107–119, 1991. *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'91)*.
34. G.A. Venkatesh. Experimental Results from Dynamic Slicing of C Programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–217, 1995.
35. G. Vidal. Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation. In *Logic-based Program Synthesis and Transformation (revised and selected papers from the 12th Int'l Workshop LOPSTR 2002)*, pages 219–237. Springer LNCS 2664, 2003.
36. M.D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.
37. M.D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
38. C.B. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proc. of the ACM/IEEE 27th Int'l Symposium on Computer Architecture*, 2000.