A Lightweight Approach to Program Specialization[⋆]

Claudio Ochoa, Josep Silva, and Germán Vidal

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 debugging, testing, differencing, specialization, and merging. In this work, we present a lightweight approach to program specialization of lazy functional logic programs which is based on dynamic slicing. The kind of specialization performed by our approach cannot be achieved with other related techniques like partial evaluation.

1 Introduction

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 [24]—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, merging). Although it is not so popular in the declarative programming community, several slicing techniques for declarative programs have also been developed during the last decade (see, e.g., [4, 14, 16–18, 20, 23]).

Basically, 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* [3, 12] 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* slices, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. More detailed information on program slicing can be found in [10, 21].

In this work, we propose a lightweight approach to program specialization in the context of *lazy* functional logic languages which is based on dynamic slicing. Modern functional logic languages like Curry [6, 8] and Toy [15] combine the most important features of functional *and* logic languages, e.g., lazy evaluation and non-determinism (see [5] for a survey). Our aim in this paper is similar to that

⋆ This work has been partially supported by CICYT TIC 2001-2705-C03-01, by the Generalitat Valenciana GRUPOS03/025, by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054, and by the MCYT HU2003-0003.

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

Fig. 1. Syntax for normalized flat programs

of Reps and Turnidge [17], who designed a program specialization method for *strict* functional programs based on *static* slicing. However, in contrast to [17], 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 dynamic slices can be considerably smaller than static slices [11, 22]). Informally speaking, specialization proceeds as follows: first, the user identifies a *representative* set of slicing criteria; then, dynamic slicing is used to compute a program slice for each slicing criterion; finally, the specialized program is built from the union of the computed slices. This is a lightweight approach since dynamic slicing is simpler than static slicing (as in [17]). Furthermore, a dynamic slicing technique is already available for the considered language [16]. Unfortunately, [16] 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 [16]). Therefore, we also introduce an appropriate technique to extract executable slices from a set of program positions.

This paper is organized as follows. In the next section, we recall 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

We consider *flat* programs [7] in this work, 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 [6, 8] and Toy [15]. In addition, we assume that flat programs are *normalized*, i.e., *let* constructs are used to ensure that the arguments of functions and constructors are always variables. As in [13], 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 "$let\ x = e\ in\ f(x)$."

The syntax of normalized flat programs is shown in Fig. 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 $e \in Exp$ 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:

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

where $(f)case$ stands for either *fcase* or *case*, 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* shows up when the argument x evaluates to a free variable: *case* suspends whereas *fcase* non-deterministically binds this variable to the pattern in a branch of the case expression.

Extra variables are those variables in a rule which do not occur in the left-hand side. They are intended to be instantiated by flexible case expressions. We assume that each extra variable x is explicitly introduced by a let binding of the form "$let\ x = x\ in\ e$". We also call such variables *logical variables.*

3 Overview of the Specialization Process

In this section, we present an informal overview of the specialization process based on dynamic slicing. Similarly to Reps and Turnidge [17], we use an example inspired in a functional version of the well-known Unix word-count utility, and specialize it to *only* count characters. This is a kind of specialization that cannot be achieved by standard partial evaluation [17]. Consider the program shown in Fig. 2 where data structures are built from:[1]

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

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

The program in Fig. 2 includes a function, `lineCharCount`, to count the number of lines and characters in a string (by using two accumulators, `lc` and `cc`, to build up the line and character counts as it travels down the string `str`). Function `printLineCount` (resp. `printCharCount`) is used to only print the number of lines (resp. characters) in a string. Function `ite` is a simple conditional while `eq`—not shown in Fig. 2—is an equality test on letters. For example, the execution of "`lineCharCount [letter, CR]`" returns the following three results:

```
Pair (S Z)     (S (S Z))     if letter reduces to A
Pair (S Z)     (S (S Z))     if letter reduces to B
Pair (S (S Z)) (S (S Z))     if letter reduces to CR
```

since function `letter` non-deterministically outputs `A`, `B` or `CR`.

[1] In the examples, for 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.

```
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}
```

Fig. 2. Example program `lineCharCount`

Now, our aim is the specialization of this program in order to extract those statements which are needed to compute *the number of characters* in a string. For this, we first consider the selection of appropriate slicing criteria and, then, the extraction of the corresponding slice (the *specialized* program).

From Tracing to Slicing. 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 trace, and `x` is the variable we are interested in (see [21]).

In our functional logic setting, slicing criteria can also be determined by inspecting the traces of the relevant program executions. For this purpose, we consider a tracing tool [2] which relies on the computation of *redex trails* [19]. In order to use this tracer, a distinguished function `main` (with no arguments) should be added to start the computation. Since we are only interested in counting characters, an appropriate definition of `main` could be as follows:

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

Here, the tracer initially shows:

```
main -> 2
     -> 2
     -> 2
```

which means that there are three non-deterministic computations starting from `main` and, moreover, all of them give the same result, 2, since `printCharCount` only demands the computation of the number of characters in the input list.

The user can now select any of these values and the system will show a top-level trace of the associated computation. For example, the trace associated to the first computation—where `letter` is evaluated to `A`—is as follows:

```
2 = main
2 = printCharCount (Pair _ (S (S Z)))
2 = printNat       (S (S Z))
2 = (+)            1 1
2 = 2
```

where "_" denotes an argument whose evaluation is not needed in the computation. Each row shows a pair "$val = exp$" where exp is an expression and val is its value. Note that function arguments appear *fully evaluated w.r.t. the considered computation*[2] in order to ease the understanding of the trace. Now, we select the argument of `printCharCount` in order to see the corresponding subtrace:

```
2               = printCharCount (Pair _ (S (S Z)))
Pair _ (S (S Z)) = lineCharCount  [A,CR]
Pair _ (S (S Z)) = lcc            [A,CR]  _ Z
Pair _ (S (S Z)) = ite            False  _  (Pair _ (S (S Z)))
Pair _ (S (S Z)) = lcc            [CR]  _  (S Z)
Pair _ (S (S Z)) = ite            True  (Pair _ (S (S Z)))  _
Pair _ (S (S Z)) = lcc            []  _  (S (S Z))
Pair _ (S (S Z)) = Pair           _  (S (S Z))
```

From this subtrace, we can already identify the interested call to `lineCharCount`. However, this information is not generally enough to determine a slicing criterion. Following [16], a slicing criterion is defined as a tuple of the form $\langle f(\overline{v_n}), v, l, \pi \rangle$, where $f(\overline{v_n})$ is a function call with fully evaluated arguments, v is its value (within 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 (see below). The connection with the tracing tool should be clear: given a (sub)trace of the form

$$val_1 = f_1(v_1^1, \ldots, v_m^1)$$
$$\ldots$$
$$val_k = f_k(v_1^k, \ldots, v_m^k)$$

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 determine a slicing criterion from the trace of a computation. For example,

$$\langle \texttt{lineCharCount [A,CR], Pair \_ (S (S Z)), 1, Pair} \perp \top \rangle$$

is a valid slicing criterion w.r.t. the considered computation. In the slicing pattern (the fourth component), $\top$ means that the computation of the corresponding subexpression is of interest while $\perp$ means that it can be ignored.

Specialization based on slicing. Given a slicing criterion, [16] presents a dynamic slicing technique that extracts the associated program slice from the *redex trail* of the computation. Redex trails [19] are directed graphs which record 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. Note that the same data structure—the redex trail—is used for both tracing [2] and slicing [16]. For instance, Fig. 3 shows the redex trail of the previous computation for `main`. Basically, 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 `printCharCount` in Fig. 3).

Argument arrows: Arguments of function and constructor calls are denoted by a pointer to the corresponding subexpression. If the evaluation of an argument

[2] I.e., as much as needed in the *complete* computation under consideration.

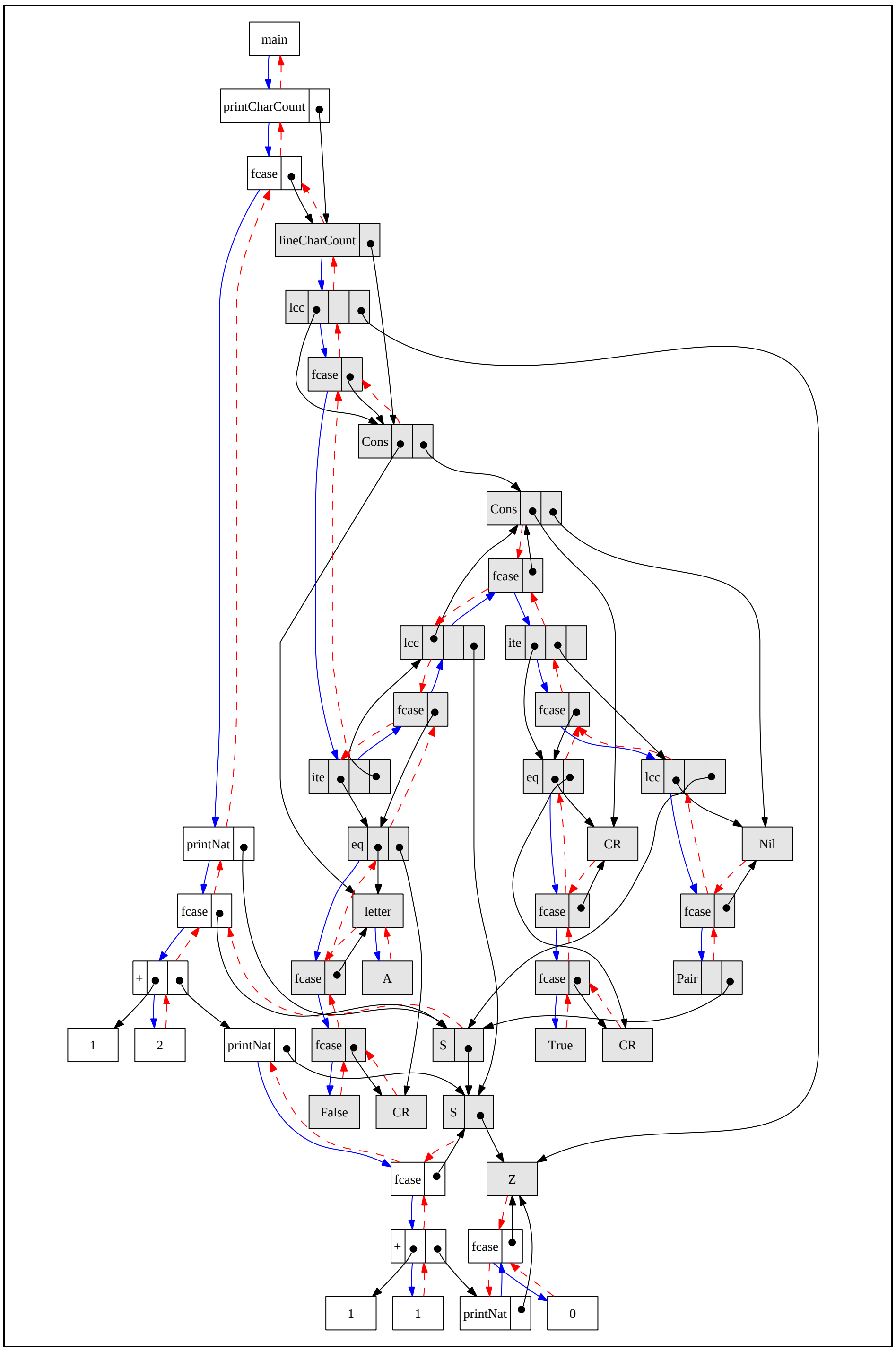

Fig. 3. Redex trail for program `lineCharCount`

```
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 ?

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

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

Fig. 4. Specialization of program `lineCharCount`

is not required in a computation, we have a null pointer (e.g., the second argument of `lcc` in Fig. 3).

Parent arrows: They are denoted by dashed arrows and point either to the redex from which a expression results—i.e., the inverse of the successor arrow—like, e.g., from `printCharCount` to `main` in Fig. 3, or to the expression who *demanded* its evaluation, e.g., from `lineCharCount` to `fcase`.

In order to extract a program slice from the computed redex trail, one should first identify the node of the graph which is associated to the slicing criterion. In our example, the node associated to the slicing criterion

$\langle$`lineCharCount` [`A`, `CR`], `Pair _ (S (S Z))`, `1`, `Pair` $\perp$ $\top\rangle$

is the topmost shadowed node of Fig. 3, since it records the first call to function `lineCharCount` that evaluates to `(Pair _ (S (S Z)))` and whose argument evaluates to the list `[A,CR]`. Then, we determine the nodes which are reachable from this node according to the slicing pattern (`Pair` $\perp$ $\top$). The set of reachable nodes are also shadowed in Fig. 3. Finally, the set of *program positions* (i.e., the locations in the source program, see Def. 1) of the expressions in the shadowed nodes is returned as the computed slice.

This notion of slice is sufficient for debugging, where we are interested in highlighting those expressions which are related to the slicing criterion. In order to specialize programs, however, we should produce *executable* program slices. For this purpose, we introduce an algorithm that extracts a program slice from the original program and the computed set of program positions. For instance, the program slice associated to the shadowed nodes of Fig. 3 is shown in Fig. 4, where the distinguished symbol "?" is used to point out that some subexpression is missing due to the slicing process.

Clearly, producing a specialized program for a concrete computation is not always appropriate. For instance, in the slice above, function `letter` can only be evaluated to `A` but not to `B` or `CR`. To overcome this problem, [17] considers *static* slicing where no input data are provided and, thus, the extracted slices preserve the semantics of the original program for any input data. Our lightweight approach is based on dynamic slicing and, thus, a *representative* set of slicing criteria—covering the interesting computations—should be determined in order to obtain program slices which are general enough.

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

Fig. 5. Small-step instrumented semantics

4 Instrumented Semantics

Now, we present an instrumented version of the small-step operational semantics for functional logic programs of [1] that also computes *program positions.*

The instrumented operational semantics is shown in Fig. 5. We distinguish two components in this semantics: the first component (columns *Heap*, *Control*, and *Stack*) defines the standard semantics introduced in [1]; the second component (columns *Fun* and *Pos*) is used to store *program positions* [16], i.e., a function name and a position (within this function) for the current expression in the control. The 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).

A configuration of the semantics is a tuple $\langle \Gamma, e, S, g, w\rangle$ where $\Gamma \in \mathit{Heap}$ is the current heap, $e \in \mathit{Exp}$ is the expression to be evaluated (called the control), 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 1 (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:*

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

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.

Note that not all subexpressions are addressed by program positions (this is the case, e.g., of the patterns in a case expression). Indeed, only those expressions which are relevant for the slicing process need to be addressed. A brief explanation for each rule of the extended semantics follows:

(varcons) It 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 the derived configuration, the current program position is updated with the position of the expression in the heap, (h, w'), which was previously introduced by rules val or let (see below).

(varexp and val) 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 variable x to the stack. As in rule varcons, the derived configuration is updated with the program position of the expression in the heap. Once a value is eventually computed, rule val updates the value of x in the heap. The associated program position for the binding is then updated with the program position (g, w) of the computed value.

(fun) This rule performs a simple unfolding; we assume that the considered program P is a global parameter of the calculus. Trivially, the program position is reset 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 expression. The local variable is renamed with a fresh name to avoid variable name clashes. 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 the program position of the binding in rules varcons and varexp. The program position in the new configuration is updated to $(g, w.2)$ in order to address the expression e_2 of the *let* construct.

(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 $(f)\{\overline{p_k \to e_k}\}$ (together with the current program position) on top of the stack. If a constructor-rooted term is produced, rule select is used to select the appropriate branch and continue with the evaluation of this branch. If a logical variable is produced and the case expression on the stack is flexible (i.e., of the form $f\{\overline{p_k \to e_k}\}$), rule guess non-deterministically chooses one alternative and continues 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.

Trivially, the instrumented semantics is a conservative extension of the original small-step semantics of [1], 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 Fig. 5 until a *final configuration* is reached. 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). 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 functions `eq` and `letter` of program `lineCharCount` again, together with the initial call (`eq letter CR`). The normalized flat program is shown in Fig. 6, where the program positions of some selected expressions are displayed in the right column. A complete computation—the one in which the non-deterministic function `letter` is evaluated to `A`—with the rules of Fig. 5 is shown in Fig. 7, where each computation step is labeled with the applied rule.

5 Computing Specialized Programs

We now formalize the computation of specialized programs. Following [16], a dynamic slice is originally defined as a set of program positions:

Definition 2 (dynamic slice). *Let P be a program. A dynamic slice for P is a set $\mathcal{W}$ of program positions such that $\mathcal{W} \subseteq \mathcal{P}os(P)$.*

Although we are interested in computing executable slices, the above notion of slice is still useful in order to easily compute the differences between several

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

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),x:(eq,1))]
         {A  -> fcase y of              [fcase:(eq,2.1),y:(eq,2.1.1)]
                {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 {A -> False; B -> True; CR -> False}
          CR -> fcase y of {A -> False; B -> False; CR -> True} }
```

Fig. 6. Functions `letter` and `eq` with program positions

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. In particular, specialized programs are computed w.r.t. a set of relevant slicing criteria and, thus, they will be obtained from the original program and the *union* of the computed sets of program positions (see below).

As discussed in Section 3, an appropriate slicing criterion in our setting is given by a tuple $\langle f(\overline{pv_n}), pv, l, \pi\rangle$. Formally,

Definition 3 (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 w.r.t. $(C_0 \Longrightarrow^* C_m)$, $pv \in PValue$ is a partial value, $l > 0$ is a natural number, and $\pi \in Pat$ is a slicing pattern. The domains FCalls (fully evaluated calls) and PValue (partial values) obey the following syntax:*

$$FCalls \ ::= \ f(pv_1, \ldots, pv_n) \qquad pv \in PValue \ ::= \ \_ \mid x \mid c(pv_1, \ldots, pv_k)$$

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:

$$\pi \in Pat \ ::= \ \bot \mid \top \mid c(\pi_1, \ldots, \pi_k)$$

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

In order to compute a slice, we should first identify the configuration that is associated to a slicing criterion, which we call *SC-configuration.* 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: (i) the control of C_i is $f(\overline{x_n})$, (ii) the variables $\overline{x_n}$ are bound in the considered computation to expressions which are "more defined" than $\overline{pv_n}$ (i.e., which are equal to $\overline{pv_n}$ except that some occurrences of "_" are replaced by an arbitrary expression), 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. The formal definition can be found in [16].

Now, we should determine those configurations that *contribute* to the evaluation of a slicing criterion. Roughly speaking, these contributing configurations

```
           ⟨[], 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⟩
⟹val       ⟨[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. 7. An example computation with the instrumented semantics

are obtained as follows: First, we collect all configurations in the subcomputation from the SC-configuration to a configuration with a value in the control. 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. A formal algorithm can be found in [16].

Given a slicing criterion, dynamic slices can be trivially obtained from the program positions of the SC-configuration as well as the contributing configurations [16]. Now, we introduce a novel method to extract an *executable* program slice from the computed program positions.

Definition 4 (executable slice). *Let P be a program and $\mathcal{W}$ an associated dynamic slice. The corresponding executable slice, $P_{\mathcal{W}}$, is obtained as follows:*

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

where $[\![e]\!]^{p}_{\mathcal{Q}} = ?$ if $p \notin \mathcal{Q}$ and, otherwise (i.e., $p \in \mathcal{Q}$), it is defined inductively as follows:

$$\begin{aligned}
[\![x]\!]^{p}_{\mathcal{Q}} &= x \\
[\![\varphi(x_1,\ldots,x_n)]\!]^{p}_{\mathcal{Q}} &= \varphi(x_1,\ldots,x_n) \\
[\![\mathit{let}\ x = e'\ \mathit{in}\ e]\!]^{p}_{\mathcal{Q}} &= \mathit{let}\ x = [\![e']\!]^{p.1}_{\mathcal{Q}}\ \mathit{in}\ [\![e]\!]^{p.2}_{\mathcal{Q}} \\
[\![e_1\ \mathit{or}\ e_2]\!]^{p}_{\mathcal{Q}} &= [\![e_1]\!]^{p.1}_{\mathcal{Q}}\ \mathit{or}\ [\![e_2]\!]^{p.2}_{\mathcal{Q}} \\
[\![(f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}]\!]^{p}_{\mathcal{Q}} &= (f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to [\![e_k]\!]^{p.2.k}_{\mathcal{Q}}}\}
\end{aligned}$$

where $\varphi \in (\mathcal{C} \cup \mathcal{F})$ is either a constructor or a defined function symbol.

The slice computed in this way would be trivially executable by considering "?" as a fresh 0-ary constructor symbol (since it is not used in the considered computation). However, from an implementation point of view, this is not a good decision since we should also extend all program types in order to include "?". Fortunately, in our functional *logic* setting, there is a simpler solution: we consider "?" as a fresh existential variable (i.e., like the anonymous variable of 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
```

Now, the specialization process should be clear. First, we compute dynamic slices—sets of program positions—for the selected slicing criteria. Then, an executable slice is built for the union of the computed dynamic slices. Moreover, there are some post-processing simplifications that can be added in order to reduce the program size and improve its readability:

$$\begin{array}{rcl} let\ \ x = ?\ \ in\ e & \Longrightarrow & \rho(e) \qquad \text{if } \rho = \{x \mapsto ?\} \\ (f)case\ x\ of\ \{\overline{p_k \to e_k}\} & \Longrightarrow & (f)case\ x\ of\ \{\overline{p'_m \to e'_m}\} \\ & & \quad \text{if } \{\overline{p'_m \to e'_m}\} = \{p_i \to e_i \mid e_i \neq ?,\ i = 1, \ldots, k\} \end{array}$$

More powerful post-processing simplifications could be added but they require a form of *amorphous* slicing [9], which is out of the scope of this work.

In [16], we prove that the computed slices are correct and minimal. From these results, it would be easy to prove that our executable slices (according to Def. 4) are also correct and minimal specializations of the original program w.r.t. the considered slicing criteria.

6 Conclusions and Future Work

In this work, we have presented a lightweight approach to program specialization of lazy functional logic programs which is based on dynamic slicing. Our method obtains a kind of specialization that cannot be achieved with other related techniques like partial evaluation.

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

2. B. Braßel, M. Hanus, F. Huch, and G. Vidal. A Semantics for Tracing Declarative Multi-Paradigm Programs. In *Proc. of PPDP'04*, pages 179–190. ACM Press, 2004.
3. J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM TOPLAS*, 9(3):319–349, 1987.
4. V. Gouranton. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. In *Proc. of SAS'98*, pages 115–133, 1998.
5. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
6. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of POPL'97*, pages 80–93. ACM, 1997.
7. M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
8. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~curry/`.
9. M. Harman and S. Danicic. Amorphous Program Slicing. In *Proc. of the 5th Int'l Workshop on Program Comprehension.* IEEE Computer Society Press, 1997.
10. M. Harman and R.M. Hierons. An Overview of Program Slicing. *Software Focus*, 2(3):85–92, 2001.
11. T. Hoffner, M. Kamkar, and P. Fritzson. Evaluation of Program Slicing Tools. In *Proc. of AADEBUG'95*, pages 51–69. IRISA-CNRS, 1995.
12. D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimization. In *Proc. of POPL'81*, pages 207–218. ACM, 1981.
13. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of POPL'93*, pages 144–154. ACM Press, 1993.
14. 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.
15. 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.
16. C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of PEPM'04*, pages 123–134. ACM Press, 2004.
17. 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.
18. S. Schoenig and M. Ducasse. A Backward Slicing Algorithm for Prolog. In *Proc. of SAS'96*, pages 317–331. Springer LNCS 1145, 1996.
19. J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of PLILP'97*, pages 291–308. Springer LNCS 1292, 1997.
20. G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.
21. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
22. G.A. Venkatesh. Experimental Results from Dynamic Slicing of C Programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–217, 1995.
23. G. Vidal. Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation. In *Proc. of LOPSTR 2002*, pages 219–237. Springer LNCS 2664, 2003.
24. M.D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, The University of Michigan, 1979.