Lightweight Program Specialization via Dynamic Slicing *

Claudio Ochoa
DIA, T.U. Madrid
28660 Boadilla del Monte
Madrid, Spain
cochoa@clip.dia.fi.upm.es

Josep Silva Germán Vidal
DSIC, T.U. Valencia
Camino de Vera S/N
E-46022 Valencia, Spain
{jsilva,gvidal}@dsic.upv.es

Abstract

Program slicing is a well-known technique that extracts from a program those statements which are relevant to a particular criterion. While *static* slicing does not consider any input data, *dynamic* slices are computed from a particular program execution. Thus, dynamic slicers are usually easier to design and implement.

In this work, we present a program specialization technique for lazy functional logic programming which is based on dynamic slicing. Our method exploits the code size reduction capabilities of slicing in order to produce a version of the original program specialized w.r.t. a given criterion. We also introduce some simple, post-processing transformations that allow us to further simplify the specialized program. The kind of specialization performed by our approach cannot be achieved with other related techniques like partial evaluation.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—operational semantics

General Terms algorithms, performance

Keywords Lazy functional logic programming, dynamic slicing, program specialization

1. Introduction

Program slicing, originally introduced by Weiser [22] in the context of imperative programming, is a general method for extracting the program sentences that potentially affect (or are affected by) some criterion (e.g., a program point, a variable, a procedure, etc), usually referred to as a *slicing criterion*. Program slices are often computed from a *program dependence graph* [4, 11] 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.

Program slicing was first proposed as a debugging technique to allow a better understanding of the portion of code which revealed an error. Since then, it has been successfully applied to a wide variety of software engineering tasks, like program understanding, debugging, testing, differencing, specialization, merging, etc. More detailed information on slicing for imperative programs can be found in the surveys of Harman and Hierons [9] and Tip [19]. 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., [5, 13, 15, 16, 17, 18, 21]).

Among them, [16] presents a specialization method for *strict* first-order functional programs based on *static* slicing. Basically, given a program P and a projection function π, [16] extracts a program that behaves like $\pi(P)$ (roughly, by symbolically pushing π backwards through the body of P). For instance, this technique can be used to extract a program slice for computing the number of lines in a string from a more general program that returns a tuple with both the number of lines and the number of characters of the string; this is the example that illustrates the backward slicing technique of Reps and Turnidge [16].

Our aim in this work is the definition of a *lightweight* approach to program specialization for *lazy* functional logic programming which is based on dynamic slicing. In contrast to [16], we consider dynamic slicing as 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 [10, 20]). A drawback of our approach is that the computed slices are semantically equivalent to the original program only w.r.t. the input data considered for slicing. However, as we will see later, the results obtained by dynamic slicing are usually general enough to produce useful specializations. Furthermore, one can also identify a *representative* set of slicing criteria (with different input data) and extract the *union* of the corresponding dynamic slices. Similarly to [16], the kind of specialization performed by our approach cannot be achieved with other related techniques like partial evaluation.

This paper is organized as follows. In the next section, we recall a simple lazy functional logic language. Section 3 formalizes an instrumented semantics which also stores the location of each reduced expression. Section 4 presents the main concepts involved in our approach to program specialization based on dynamic slicing. A complete example which illustrates our technique is shown in Section 5. Finally, Section 6 concludes and points out several directions for further research.

* This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2004-00231 and HU 2003-0003, by the *Generalitat Valenciana* GRUPOS03/025, and by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054.

WCFLP'05 September 29, 2005, Tallinn, Estonia.

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

Figure 1. Syntax for normalized flat programs

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 [14]. 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 [12], 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 (denoted $x, y, z, \ldots \in \mathcal{X}$), data constructors (denoted $a, b, c, \ldots \in \mathcal{C}$), function calls (denoted $f, g, h, \ldots \in \mathcal{F}$), 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 free variable x is explicitly introduced by a let binding of the form "$let\ x = x\ in\ e$".

3. Instrumented Semantics

In this section, we present an instrumented version of the small-step operational semantics for lazy functional logic programs of [1] that also computes *program positions*. This additional information will become useful to identify the location, in the program, of each reduced expression.

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 (column *Pos*) is used to store *program positions*, i.e., a list of pairs (function name, position within this function) associated to the current expression in the control. The semantics obeys the following naming conventions:

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

A *heap* is a partial mapping from variables to expressions. Each mapping $x \mapsto_{(g,w)} e$ is also 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 free 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 free variable (w.r.t. the associated heap).

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

DEFINITION 3.1 (position, program position).
Positions are represented by a sequence of natural numbers, where Λ *denotes the empty sequence (i.e., the root position). They are used to address subexpressions of an expression viewed as a tree:*

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

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

Note that variables in a let construct or patterns in a case expression are not addressed by program positions since they will not be considered in the slicing process. While other approaches to dynamic slicing, e.g., [2], consider some form of *explicit* labeling for each program expression, our program positions can be seen as a form of *implicit* labeling.

The instrumented operational semantics is shown in Fig. 2. Two extensions are necessary w.r.t. the standard operational semantics. As mentioned before, we consider that each mapping $x \mapsto_{(g,w)} e$ in the heap also contains the program position (g, w) of expression e. On the other hand, we could add a new component, (g, w), to each state $\langle \Gamma, e, S \rangle$ of the standard semantics, so that (g, w) denotes the program position of the current expression e in the control. However, this would not be appropriate from an implementation point of view. As we will discuss later, our slicing tool is implemented on top on an existing tracer based on *redex trails* [3]. In this model, *variables* are not stored in redex trails but we still need to collect their program positions. Our solution consists in adding to each state of the standard semantics a *list* of program positions $\mathcal{P}$ so that the program position in the head of list $\mathcal{P}$ denotes the program position of the current expression in the control and, if the tail of $\mathcal{P}$ is not empty, it also contains the program positions of the (chain of) variables whose evaluation was required in order to reach the current expression in the control. This approach, though more complex, allows us to easily implement our slicer—and, thus, the program specialization technique proposed in this paper—by adding the list of program positions to the nodes of the redex trail. Therefore, a configuration of the instrumented semantics is defined as follows:

Rule	*Heap*	*Control*	*Stack*	*Pos*	
varcons	$\langle\ \Gamma[x \mapsto_{(g,w)} t],$	$x,$	$S,$	$\mathcal{P}\ \rangle$	where t is constructor-rooted
	$\Longrightarrow \langle\ \Gamma[x \mapsto_{(g,w)} t],$	$t,$	$S,$	$(g,w) : \mathcal{P}\ \rangle$	
varexp	$\langle\ \Gamma[x \mapsto_{(g,w)} e],$	$x,$	$S,$	$\mathcal{P}\ \rangle$	where e is not constructor-rooted
	$\Longrightarrow \langle\ \Gamma[x \mapsto_{(g,w)} e],$	$e,$	$x : S,$	$(g,w) : \mathcal{P}\ \rangle$	and $e \neq x$
val	$\langle\ \Gamma,$	$v,$	$x : S,$	$(g,w) : \mathcal{P}\ \rangle$	where v is a value
	$\Longrightarrow \langle\ \Gamma[x \mapsto_{(g,w)} v],$	$v,$	$S,$	$(g,w) : \mathcal{P}\ \rangle$	
fun	$\langle\ \Gamma,$	$f(\overline{x_n}),$	$S,$	$\mathcal{P}\ \rangle$	where $f(\overline{y_n}) = e \in P$
	$\Longrightarrow \langle\ \Gamma,$	$\rho(e),$	$S,$	$[(f,\Lambda)]\ \rangle$	and $\rho = \{\overline{y_n \mapsto x_n}\}$
let	$\langle\ \Gamma,$	$let\ x = e_1\ in\ e_2,$	$S,$	$\mathcal{P}\ \rangle$	where $\rho = \{x \mapsto y\}$, y is fresh,
	$\Longrightarrow \langle\ \Gamma[y \mapsto_{(g,w.1)} \rho(e_1)],$	$\rho(e_2),$	$S,$	$[(g,w.2)]\ \rangle$	and $\mathcal{P} = (g,w) : \mathcal{P}'$
or	$\langle\ \Gamma,$	$e_1\ or\ e_2,$	$S,$	$\mathcal{P}\ \rangle$	where $i \in \{1,2\}$
	$\Longrightarrow \langle\ \Gamma,$	$e_i,$	$S,$	$[(g,w.i)]\ \rangle$	and $\mathcal{P} = (g,w) : \mathcal{P}'$
case	$\langle\ \Gamma,$	$(f)case\ x\ of\ \{\overline{p_k \rightarrow e_k}\},$	$S,$	$\mathcal{P}\ \rangle$	where $\mathcal{P} = (g,w) : \mathcal{P}'$
	$\Longrightarrow \langle\ \Gamma,$	$x,$	$((f)\{\overline{p_k \rightarrow e_k}\}, \mathcal{P}) : S,$	$[(g,w.1)]\ \rangle$	
select	$\langle\ \Gamma,$	$c(\overline{y_n}),$	$((f)\{\overline{p_k \rightarrow e_k}\}, \mathcal{P}) : S,$	$\mathcal{P}''\ \rangle$	where $i \in \{1,\ldots k\}$, $p_i = c(\overline{x_n})$,
	$\Longrightarrow \langle\ \Gamma,$	$\rho(e_i),$	$S,$	$[(g,w.2.i)]\ \rangle$	$\rho = \{\overline{x_n \mapsto y_n}\}$, and $\mathcal{P} = (g,w) : \mathcal{P}'$
guess	$\langle\ \Gamma[y \mapsto_{(h,w')} y],$	$y,$	$(f\{\overline{p_k \rightarrow e_k}\}, \mathcal{P}) : S,$	$\mathcal{P}''\ \rangle$	where $i \in \{1,\ldots k\}$, $p_i = c(\overline{x_n})$,
	$\Longrightarrow \langle\ \Gamma[y \mapsto_{(\_,\_)} \rho(p_i),$	$\rho(e_i),$	$S,$	$[(g,w.2.i)]\ \rangle$	$\rho = \{\overline{x_n \mapsto y_n}\}$, $\overline{y_n}$ are fresh
	$\overline{y_n \mapsto_{(\_,\_)} y_n}],$				variables, and $\mathcal{P} = (g,w) : \mathcal{P}'$

Figure 2. Small-step instrumented semantics

DEFINITION 3.2 (configuration).
A configuration of the semantics is a tuple $\langle \Gamma, e, S, \mathcal{P}\rangle$ where $\Gamma \in$ Heap is the current heap, $e \in$ Exp is the expression to be evaluated (called the control), S is the stack, and $\mathcal{P}$ denotes a list of program positions.

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 this rule, we update the list of program positions to $(g,w) : \mathcal{P}$, i.e., the program position of the expression in the control followed by the program positions of all variables (if any) whose evaluation was demanded for computing it (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. Consider, e.g., a configuration with heap $\Gamma = [\mathtt{x} \mapsto_{(\mathtt{g}',\mathtt{w}')} \mathtt{y}, \mathtt{y} \mapsto_{(\mathtt{g},\mathtt{w})} (\mathtt{s} : \mathtt{ss})]$, control `x`, and program positions $\mathcal{P}$. Then rules varexp and varcons apply, in this order, so that the list of program positions in the derived configuration is $(\mathtt{g},\mathtt{w}) : (\mathtt{g}',\mathtt{w}') : \mathcal{P}$.

(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 $[(f,\Lambda)]$ 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 reset 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 associated list of program positions in 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 $(f)\{\overline{p_k \rightarrow e_k}\}$ (together with the current list of program positions) 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 free variable is produced and the case expression on the stack is flexible (i.e., of the form $f\{\overline{p_k \rightarrow e_k}\}$), 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 free variable to the corresponding pattern. In both rules, select and guess, the list of program positions 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 column 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. 2 until a *final configuration* is reached.

DEFINITION 3.3 (initial configuration).
An initial configuration has the form $\langle$`[]`, `main`, `[]`, `[]`$\rangle$, where `main` *is a distinguished function (with no arguments) that is used to start the computation.*

DEFINITION 3.4 (final configuration).
A final configuration has the form: $\langle \Delta, v,$ `[]`$, \mathcal{P}\rangle$, where v is a value, Δ is a heap containing the computed bindings, and $\mathcal{P}$ is a list of program positions.

We denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$. A derivation $C \Longrightarrow^* C'$ is *complete* if C' is a final configuration.

The reason why we need to store a list of positions instead of a single one, can be better explained by means of a simple example. Consider the following program where the distinguished function `main` calls to the function `f`, and this function just passes its argument to another function `g`.

```
main = let x=1 in f x

f x = g x

g x = x
```

The execution of this programs with our semantics yields the following configurations:

$\langle$[], main, [], []$\rangle$
$\Longrightarrow \langle$[], let x=1 in f x, [], [(main,Λ)]$\rangle$ (fun)
$\Longrightarrow \langle$[y $\mapsto_{(main,1)}$ 1], f y, [], [(main,2)]$\rangle$ (let)
$\Longrightarrow \langle$[y $\mapsto_{(main,1)}$ 1], g y, [], [(f,Λ)]$\rangle$ (fun)
$\Longrightarrow \langle$[y $\mapsto_{(main,1)}$ 1], y, [], [(g,Λ)]$\rangle$ (fun)
$\Longrightarrow \langle$[y $\mapsto_{(main,1)}$ 1], 1, [], [(main,1),(g,Λ)]$\rangle$ (varcons)

Note here that a list of positions is needed to express the fact that the value in the control of the semantics is related to both *(main,1)* and *(g,Λ)*.

4. Program Specialization

In this section, we introduce our technique to perform program specialization via the computation of executable dynamic slices. Following [15], a dynamic slice is originally defined as a set of program positions:

DEFINITION 4.1 (dynamic slice). *Let P be a program. A dynamic slice for P is a set $\mathcal{W}$ of program positions such that $\mathcal{W} \subseteq \mathcal{P}os(P)$.*

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 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 order to keep the technical developments simple, we introduce in this work a restricted notion of slicing criterion. A more flexible definition can be found in [15].

DEFINITION 4.2 (slicing criterion). *A slicing criterion is given by a* slicing pattern *π that denotes the interesting parts of the program's output (i.e., of the evaluation of a distinguished function* main*).*

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.

Roughly speaking, a dynamic slice should contain the program positions in those configurations of a derivation that ends with the computation of a value which *matches* the slicing criterion. We now formally define the auxiliary function *match*:

DEFINITION 4.3 (*match*). *Let v be a value (a constructor-rooted term or a free variable) and π a slicing pattern. Then, function $match(v, \pi)$ is defined as follows:*

$$match(v,\pi) = \begin{cases} \{\,\} & \text{if } v \in \mathcal{X} \text{ and } \pi = \top \\ \{\overline{(x_n, \top)}\} & \text{if } v = c(\overline{x_n}) \text{ and } \pi = \top \\ \{\overline{(x_n, \pi_n)}\} & \text{if } v = c(\overline{x_n}) \text{ and } \pi = c(\overline{\pi_n}) \\ Fail & \text{otherwise} \end{cases}$$

According to the instrumented semantics of Fig. 2, not all expressions in the control of a configuration start a new list of program positions. The following definition, adapted from [3] to include program positions, introduces the notion of *relevant configuration*, formalizing when a configuration contains an expression that initializes a list of program positions.

DEFINITION 4.4 (relevant configuration).
A configuration $\langle \Gamma, e, S, \mathcal{P} \rangle$ is relevant iff one of the following conditions hold:

- *e is a value (according to Γ, i.e., a constructor call or a variable x with $\Gamma[x] = x$) and the stack S is not headed by a variable, or*
- *e is a function call, a let expression, a disjunction, or a case expression.*

Given a derivation $\mathcal{D} : (C_0 \Longrightarrow^* C_n)$, $\mathcal{R}el(\mathcal{D})$ denotes the set of relevant configurations in $\mathcal{D}$.

Now, we can already identify those configurations which *contribute* to the dynamic slice:

DEFINITION 4.5 (contributing configurations).
Let P be a program and π be a slicing criterion. The set of contributing configurations for P w.r.t. π is given by $\mathcal{C}on(P, \langle$[], main, [], []$\rangle, \pi)$, where auxiliary function $\mathcal{C}on$ is defined as follows:

$$\mathcal{C}on(P, C, \pi) = \mathcal{R}el(\mathcal{D}) \ \cup \bigcup_{i \in \{1,\ldots,n\}} \mathcal{C}on(P, C_i, \pi_i)$$

where $\mathcal{D} = (C \Longrightarrow^* C')$ is a complete derivation in P, $C' = \langle \Gamma, v,$ [], $\mathcal{P} \rangle$, $match(v, \pi) = \{\overline{(x_n, \pi_n)}\} \neq Fail$, and $C_i = \langle \Gamma, x_i,$ [], [] $\rangle$ for all $i = 1, \ldots, n$.

Note that in Def. 4.5 new subcomputations are started for each argument of the slicing criteria. A dynamic slice can easily be obtained from the set of contributing configurations as follows:

DEFINITION 4.6 (dynamic slice).
Let P be a program and π be a slicing criterion. The dynamic slice of P w.r.t. π, $\mathcal{S}lice(P, \pi)$, is defined as the set of program positions

$$\{\ (g, w) \mid \ \langle \Gamma, e, S, \mathcal{P} \rangle \in \mathcal{C}on(P, \langle \texttt{[]}, \texttt{main}, \texttt{[]}, \texttt{[]} \rangle, \pi) \text{ and } (g, w) \text{ belongs to list } \mathcal{P} \ \}$$

As mentioned before, the notion of dynamic slice as a set of program positions is not enough for program specialization. Therefore, we introduce a method to extract an *executable* program slice from the computed program positions.

DEFINITION 4.7 (executable slice). *Let P be a program and π a slicing criterion. The associated executable slice, $\mathcal{ES}lice(P, \pi)$, is obtained as follows:*

$$\mathcal{ES}lice(P,\pi) = \{ f(\overline{x_n}) = [\![e]\!]^{\Lambda}_{\mathcal{Q}} \mid \ (f, \Lambda) \in \mathcal{S}lice(P,\pi) \text{ and } \mathcal{Q} = \{p \mid (f,p) \in \mathcal{S}lice(P,\pi)\} \}$$

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

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

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 free variable (i.e., like the anonymous variable of Prolog). This is very easy to implement in Curry by just adding a local declaration to each function with some fresh variables. For instance,

$$\texttt{lineCharCount str} \;=\; \texttt{lcc str ? Z}$$

is replaced by

$$\texttt{lineCharCount str} \;=\; \texttt{let x = x in lcc str x Z}$$

Moreover, there are several post-processing simplifications that can easily be added in order to reduce the program size and improve its readability:

(let simplification)

$$let\ x = ?\ in\ e \implies \rho(e) \quad \text{where } \rho = \{x \mapsto ?\}$$

(or simplification)

$$\begin{array}{lll} e_1\ or\ e_2 & \Longrightarrow & e_1 \quad \text{if } e_2 = ? \\ e_1\ or\ e_2 & \Longrightarrow & e_2 \quad \text{if } e_1 = ? \end{array}$$

(case simplification)

$$(f)case\ x\ of\ \{\overline{p_k \to e_k}\} \implies (f)case\ x\ of\ \{\overline{p'_m \to e'_m}\}$$

where $\{\overline{p'_m \to e'_m}\} = \{p_i \to e_i \mid e_i \neq ?,\ i = 1, \ldots, k\}$

(argument deletion)

if a function or constructor is always called with ? at some argument position, these arguments can be deleted from the program.

Now, the specialization process should be clear:

1. First, the user introduces a function `main` in the source program P which identifies the desired computation.
2. Then, the user also provides the slicing criterion π that determines the part of the output of function `main` that she is interested in.
3. Now, an executable slice, $\mathcal{E}Slice(P, \pi)$ is built.
4. Finally, some post-processing simplifications are performed in order to further reduce the computed slice.

It is worthwhile to note that our specialized programs *are not* more efficient than the original programs. In contrast to *partial evaluation* (which aims at improving program performance), our goal is to obtain a program which is specialized to produce only *part* of the results of the original program. The next section illustrates how our specialization technique proceeds.

5. The Method in Practice

In this section, we present an example that illustrates the kind of specialization that can be achieved with our method.

Similarly to Reps and Turnidge [16], we use an example inspired by a functional version of the well-known Unix word-count utility, and specialize it to *only* count characters. Consider the following program:[1]

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

data String = Nil | Cons Letter String

printLineCount (Pair x _) = printNat x
printCharCount (Pair _ x) = printNat x

printNat x = fcase x of
               {Z -> 0;
             (S m) -> let n = printNat m
                      in 1 + n}

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))}

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 } }
```

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

This program 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 `ite` is a simple conditional while `eq` is an equality test on letters. Functions `printLineCount` and `printCharCount` convert numbers in Peano's notation to natural numbers by means of the function `printNat`.

For example, the execution of "`lineCharCount` $[\texttt{A}, \texttt{A}, \texttt{B}, \texttt{CR}]$" returns the following result:

$$\texttt{Pair (S Z) (S (S (S (S Z))))}$$

whilst the execution of

$$\texttt{printCharCount (Pair (S Z) (S (S (S (S Z)))))}$$

returns 4. For the sake of readability, we are representing strings as lists of numbers without the *lets* needed to build such lists, according to our flat language.

Now, our aim is the specialization of this program in order to extract those statements which are needed to compute only *the number of characters* in a string. For this, we add for instance the following definition of `main` to the original program:

```
main = let lc = lineCharCount [A,A,B,CR]
       in printCharCount lc
```

Then, we compute an executable slice w.r.t. the slicing criterion "`Pair` $\bot$ $\top$", in order to discard the computations associated to the first argument of `Pair`. The computed slice is as follows:

```
main = let lc = lineCharCount [A,A,B,CR]
       in printCharCount lc

printCharCount (Pair _ x) = printNat x
```

[1] For readability, we omit some of the brackets and write function applications as in Curry. Furthermore, we consider programs that are not normalized for clarity.

```
printNat x = fcase x of
               {Z -> 0;
             (S m) -> let n = printNat m
                      in 1 + n}

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)) }

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

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

Note that we use here "?" instead of "$let\ x = x\ in$" for a better understanding. Finally, by applying the post-processing simplifications shown above, we get the following program:

```
main = let lc = lineCharCount [A,A,B,CR]
       in printCharCount lc

printCharCount (Pair x) = printNat x

printNat x = fcase x of
               {Z -> 0;
             (S m) -> let n = printNat m
                      in 1 + n}

lineCharCount str = lcc str Z

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

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

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

As can be seen, despite the fact that we have produced a *dynamic* slice for a particular computation, the specialized function `lineCharCount` can still be used to count the number of characters in any arbitrary string.

Clearly, there are cases in which computing a dynamic slice gives a too restricted specialized program. In these cases, the user can provide a *representative* set of slicing criteria—covering all the interesting computations—so that the executable slice is produced from the union of the associated dynamic slices (i.e., sets of program positions).

5.1 Implementation Issues

An implementation of this specialization technique has been undertaken by extending an existing slicing tool for Curry programs [15].

Rather than starting from scratch, the technique of [15] relies on a slight extension of redex trails. Basically, they extend the redex trail model of [3] in order to also store the *location*, in the program, of each reduced expression. A dynamic slice is then computed by first gathering the set of nodes, in the redex trail, which are *reachable* from the slicing criterion and, then, by deleting (or hiding) those expressions of the original program whose location does not appear in the set of collected nodes. Basically, a dynamic slice is given by a set of program positions which are used to highlight the parts of the source program that belong to the slice (which is appropriate for debugging).

Now, the implemented slicer has been extended in order to produce an *executable slice* rather than a set of program positions. Then, the application of the post-processing simplifications will produce the desired specialized program.

6. Concluding Remarks

In this paper, we have presented a lightweight approach to the 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.

This work can be seen as an extension of the dynamic slicing technique of [15]. The main contributions of our work, in comparison to [15], are the following: first, we have introduced a simplified notion of dynamic slicing (the construction in [15] is much more complex since they consider a generic notion of slicing criterion); we have introduced an appropriate definition of *executable* slice ([15] only considered sets of program positions as slices); we have identified several post-processing simplifications that can be used to further reduce the size of these executable slices; and, finally, we have illustrated the use of dynamic slicing for program specialization through a representative example.

As a future work, we plan to extend the current method in order to perform more aggressive simplifications. Another promising topic for future work is the definition of a program specialization technique 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] S.K. Biswas. *Dynamic Slicing in Higher-Order Programming Languages*. PhD thesis, Department of CIS, University of Pennsylvania, 1997.

[3] 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.

[4] 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.

[5] V. Gouranton. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. In *Proc. of SAS'98*, pages 115–133, 1998.

[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 R.M. Hierons. An Overview of Program Slicing. *Software Focus*, 2(3):85–92, 2001.

[10] T. Hoffner, M. Kamkar, and P. Fritzson. Evaluation of Program Slicing Tools. In *Proc. of AADEBUG'95*, pages 51–69. IRISA-CNRS, 1995.

[11] 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.

[12] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. of POPL'93*, pages 144–154. ACM Press, 1993.

[13] 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.

[14] 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.

[15] C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of PEPM'04*, pages 123–134. ACM Press, 2004.

[16] 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.

[17] S. Schoenig and M. Ducasse. A Backward Slicing Algorithm for Prolog. In *Proc. of SAS'96*, pages 317–331. Springer LNCS 1145, 1996.

[18] G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and Dynamic Slicing of Constraint Logic Programs. *J. Automated Software Engineering*, 9(1):41–65, 2002.

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

[20] G.A. Venkatesh. Experimental Results from Dynamic Slicing of C Programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–217, 1995.

[21] 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.

[22] M.D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.