Journal of Universal Computer Science, vol. 13, no. 6 (2007), 854-873
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © J.UCS

Higher-Order Lazy Functional Slicing

Nuno F. Rodrigues
(DI-CCTC, Universidade do Minho, Portugal
nfr@di.uminho.pt)

Luís S. Barbosa
(DI-CCTC, Universidade do Minho, Portugal
lsb@di.uminho.pt)

Abstract: Program slicing is a well known family of techniques intended to identify and isolate code fragments which depend on, or are depended upon, specific program entities. This is particularly useful in the areas of reverse engineering, program understanding, testing and software maintenance. Most slicing methods, and corresponding tools, target either the imperative or the object oriented paradigms, where program slices are computed with respect to a variable or a program statement.

Taking a complementary point of view, this paper focuses on the slicing of higher-order functional programs under a lazy evaluation strategy. A prototype of a Haskell slicer, built as proof-of-concept for these ideas, is also introduced.

Key Words: Program slicing, functional programming, program analysis
Category: D.1.1, I.2.2, I.2.4

1 Introduction

Introduced by Weiser [13, 11, 12] in the late Seventies, program slicing is a family of techniques for isolating parts of a program which depend on or are depended upon a specific computational entity referred to as the *slicing criterion*. In Weiser's view, program slicing is an abstraction exercise that every programmer has gone through, aware of it or not, every time he undertakes source code analysis.

Weiser's original definition has been since then re-worked and expanded several times, leading to the emergence of different methods for defining and computing program slices. Despite this diversity, most of the methods and corresponding tools target either the imperative or the object oriented paradigms, where program slices are computed with respect to a variable or a program statement.

Weiser approach corresponds to what would now be classified as a *backward*, *static* slicing method. A dual concept is that of *forward slicing* introduced by Horwitz et al [3]. In forward slicing one is interested on what depends on or is affected by the entity selected as the slicing criterion. Note that combining the two methods also gives interesting results. In particular the union of a backward to a forward slice for the same criterion n provides a sort of a selective window over the code highlighting the *region* relevant for entity n.

Another duality pops up between *static* and *dynamic* slicing. In the first case only static program information is used, while the second one also considers input values [4, 5] leading frequently, due to the extra information used, to smaller and easier to analyse slices, although with a restricted validity.

Taking a different perspective, this paper focuses on the problem of slicing *functional* programs [1]. Since slicing is a technique intended to be used by programmers while developing, analyzing or transforming source code in a production context, it should target real languages and in a most complete way. Otherwise, such techniques would be useless by failing to stand up to the expectations of their natural clients *i.e.*, programmers and software analysts. Thus, our approach to functional slicing targets an emerging programming paradigm: *functional programs with higher-order constructs sharing a lazy strategy evaluation.* Additionally it will be shown in section 7, that the strict version of the proposed technique can be easily derived from the lazy one, and that the removal of higher-order constructs represents a trivial simplification of the method introduced here.

By the beginning of this work we thought that the development of higher-order lazy functional slicer represented a more or less straightforward engineering problem that could be easily solved by making use of some combination of parsing and syntax tree traversal operations. However, all attempts to build such a tool resorting to a direct implementation of these operations invariantly ended by the discovery of some particular case where the resulting slices did not correspond to the expected correct ones. Even more, by performing minor changes in the implementations in order to correctly cover some special cases, one often ended up introducing new problems or preventing the treatment of other special cases.

Soon, however, we realized the problem complexity had been underestimated from the outset. This lead to the development of a semantic-based approach, providing a suitable level of abstraction, in which the lazy slicing problem could be specified and solved. Furthermore, the formed framework makes it possible to state and verify relevant properties of the slicing process.

The research background of this paper amounts to the development of HaSlicer [1], a functional slicer targeting the functional language HASKELL [1]. Before developing HaSlicer it was decided to pay special attention to high order entities, somehow related to an architectural view over functional systems. Thus, HaSlicer deals with code entities such as modules, data-types and functions, ignoring completely more fine grained entities like functional combinator expressions. Although the success of this decision is attested by the system high level views given by the Functional Dependency Graph visualizer [10], there was still a lack

[1] The tool is available online at http://labdotnet.di.uminho.pt/HaSlicer/HaSlicer.aspx

of proper foundations and techniques for what may be called low level slicing of functional programs. This paper is a step in that direction.

The context for this research is a broader project on *program understanding and re-engineering* of legacy code supported by formal methods. Actually, if forward software engineering can be regarded as an almost lost opportunity for formal methods (with notable exceptions in areas such as safety-critical and dependable computing), reverse engineering looks more and more a promising area for their application, due to the engineering complexity and exponential costs involved. In a situation in which the only quality certificate of the running software artifact still is life-cycle endurance, customers and software producers are little prepared to modify or improve running code. However, faced with so risky a dependence on legacy software, managers are more and more prepared to spend resources to increase confidence on — *i.e.*, the level of understanding of — their code.

The paper is organised as follows. Section introduces the functional language FL, a "sugared" λ-calculus used to express our programs. Section 3 discusses the relationship between slicing and evaluation and justifies the use of a semantic approach to reason about slicing of functional programs. Sections 4 and 5 present two algorithms for performing lazy functional slicing. In section 6 a strict version of the slicing algorithm presented in section 5 is discussed. Section 7 resorts to the semantics presented in sections 5 and 6 to prove that lazy slices are smaller than or equal to their strict counterparts. Finally section 8 concludes and discusses topics for future work.

1.1 Contributions

We formally introduce a dynamic slicing algorithm for higher-order lazy functional languages which, to the best of our knowledge, is a first attempt to address slicing for this kind of programs. It is also shown how the same formal setting can be used to state and prove slicing properties. The whole approach is prototyped in a library developed for Haskell a pure higher-order lazy functional language, as a proof-of-concept[2].

2 Related Work

While we regard this work as a first incursion on higher-order lazy functional slicing, there are a number of related works that should be mentioned.

In [8] Reps and Turnidge provide a static functional slicing algorithm but, in contrast to our approach, theirs target first-order strict functional programs.

[2] The library is available online at http://alfa.di.uminho.pt/∼nfr/Tools/Tools.html

Besides considering a different language class (first-order) and a different evaluation strategy (strict), the authors define slicing criteria by means of projection functions, a strategy that we regard as more rigid when compared to our own approach which resorts to a subexpression labeling mechanism.

In [7] the authors present a strategy to dynamically slice lazy functional languages. Nevertheless, they leave higher-order constructs as a topic for future work, and base their approach on redex trails. This leads to a slicing criterion definition (which consists of a tuple containing a function call with full evaluated arguments, its value in a particular computation, the occurrence of the function call and a pattern indicating the interesting part of the computed value) which is much more complex to use in practice than our own. The latter, by pointing out a specific (sub)expression in the code, represents a more natural way for the analyst to encode the relevant aspects of the code that he/she wants isolated.

Perhaps the work most related to ours is [2], where the author presents an algorithm for dynamic slicing of strict higher-order functional languages followed by a brief adaptation of the algorithm to lazy evaluation. A major difference with the approach proposed in their paper is that, recursive calls must be explicitly declared in the language and there is no treatment of mutual recursive functions which, as pointed out by the author, results in a considerable simplification of the slicing process. Again, we believe that our definition of the slicing criterion is more precise than the one used in [2], which consists of the value computed by the program in question (even though more flexible slicing criteria are briefly discussed).

Finally, it should be emphasized that a slicing criterion, like the one we propose, that permits to choose any (sub)expression of the program under analysis, deeply influences and augments the complexity of the slicing process, specially under a lazy evaluation framework like the one we address. In fact, this aspect is the responsible for the evolution of the slicing algorithm from a one phase process, like the one presented in section 5, to a two phase process where one must first keep track of internal (sub)expression lazy dependencies before calculating the final slicing with respect to the relevant (sub)expressions.

3 The Functional Language

Given that one is not interested on focusing on a single functional language, but rather to come up with a technique that is potentially applicable to all higher-order lazy functional languages, one has decided to introduce a common level functional language which can easily serve several functional programming language implementations.

The process of choosing such a syntax had to fulfill two main requisites. The language could not be excessively broad since this would introduce an unnecessary notational burden in the representation. On the other hand it could not be excessively small because this would make translations from/to real functional languages too complex to achieve.

Values	$z ::=$	$(\lambda x.e)$	
	$\mid$	$(C\ x_1 \cdots x_a)$	$a \geq 0$
Expressions	$e ::=$	z	
	$\mid$	$e\ x$	
	$\mid$	x	
	$\mid$	$\texttt{let}\ x_n = e_n\ \texttt{in}\ e$	$n > 0$
	$\mid$	$\texttt{case}\ e\ \texttt{of}\ \{(C_j\ x_{1j} \cdots x_{aj}\ \texttt{->}\ e_j\}_{j=1}^{n}$	$n > 0,\ a \geq 0$
Programs	$prog ::=$	$x_1 = e_1, \ldots, x_n = e_n$	

Figure 1: The FL syntax

Thus, one had to find a tradeoff between this conditions to make the entire process feasible. Such a tradeoff is captured in language FL where the syntax is presented in figure 1. FL notation is basically a λ-Calculus enriched with `let` and `case` statements. It introduces the domain U of values, the domain E of expressions, the domain P of programs and the domain of V of variables. Note that values are also expressions by the first rule in the definition of expressions.

A very important detail about the language in figure 1 is that functional application cannot occur between two arbitrary functional expressions, but only between an expression and a variable previously defined. In practice this implies that at evaluation time the applied expression must have been previously added to the heap so that it can be used on a functional application. This requisite may seem strange for now, but it is necessary to deal correctly with the semantics upon which we define the slicing process.

It requires, however, some care when converting concrete functional programs to FL. In practice, the translation is achieved by the introduction of a new free variable with a let expression and the subsequent substitution of the expression by the newly introduced variable.

Of course, to treat real functional languages, some other straightforward syntactic translations are in demand. These includes the substitution of `if then else` by `case` expressions with the respective `True` and `False` values or the substitution of `where` constructions by `let`.

Such syntactic transformations have been implemented, as a prove of concept, in a front end functional language (Haskell). Even more, they were implemented isomorphically because by the end of the slicing process, one wants to be able to reconstruct the slice exactly like the original program except by the removal of some sliced expressions.

Finally, we have to uniquely identify the functional expressions and sub-expressions of a program, such that the slicing process refer to these identifiers in order to specify what parts of the program belong to a specific slice. Such identifiers, collected in a set L, are introduced by expression labeling as shown in figure 2, where $a \geq 0$ and $n > 0$.

$$
\begin{array}{llcl}
\text{Values} & z & ::= & (\lambda x : l_1 . e) : l \\
 & & | & (C\ x_1 : l_1 \cdots x_a : l_a) : l \\
\text{Expressions} & e & ::= & z \\
 & & | & e\ (x : l') : l \\
 & & | & x : l \\
 & & | & \texttt{let}\ x_n = e_n : l_n\ \texttt{in}\ e : l \\
 & & | & \texttt{case}\ e\ \texttt{of}\ \{(C_j\ x_{1j} : l_{1j} \cdots x_{aj} : l_{aj}) : l'\ \texttt{->}\ e_j\}_{j=1}^{n} : l \\
\text{Programs} & prog & ::= & x_1 = e_1, \ldots, x_n = e_n
\end{array}
$$

Figure 2: Labeled FL syntax

For the moment, one may look at labels from L as simple unique identifiers of functional expressions. Latter, these labels will be used to capture information about the source language representation of the expression they denote, so that, by the end of the slicing process, one can be able to construct a sliced source code version of the program under analysis.

4 Slicing and Evaluation

Slicing of functional programs is an operation that largely depends on the underlying evaluation strategy for expressions. This can be exemplified in programs where strict evaluation introduces non termination whereas a lazy strategy produces a result. As an example, consider the following functional program.

```
fact :: Int -> Int
fact 0 = 1
fact k = k * fact (k-1)

ssuc :: Int -> Int -> Int
ssuc r y = y + 1

g :: Int -> [Int] -> [Int]
g x z = map (ssuc (fact x)) z
```

If we calculate the slice of the above program w.r.t. expression `g (-3) [1,2]`, taking into account that the program is being evaluated under a strict strategy, the evaluation will never terminate and the slice fails to compute.

On the other hand, under a lazy evaluation strategy, the evaluation is possible because `succ` is not strict over its arguments, and therefore `(fact x)` which introduces non terminating behaviour is not computed. Thus, slicing is now feasible and one would expect to obtain the following slice:

```
ssuc :: Int -> Int -> Int
ssuc r y = y + 1

g :: [Int] -> [Int]
g z = map (ssuc (fact x)) z
```

Note that strictly speaking the computed slice is not executable. Actually this would require definition of function `fact` in order to be interpreted or compiled. This was a deliberate choice because, in a functional framework, if one calculates executable slices (without using any further program transformation), it often happens that such slices take enormous proportions when compared to the original code. Nevertheless, and because the expressions that are sliced away do not interfere with the selected slicing criterion, a program transformation to be used for this case is to substitute the expression in question by some special value of the same type. In HASKELL, for instance, and because types have a cpo structure, one could use the bottom value (usually denoted by $\perp$) of the type in question to signal the superfluous expression. These and other possible code transformations that target the execution of slices are, however, beyond the scope of this paper.

The approach to low level slicing of functional programs proposed in this paper is mainly oriented (but see section 6) to lazy languages. Our motivation was that slicing has never been treated under such an evaluation strategy (combined with higher-order constructs). Moreover, intuition suggests, as in the example above, that lazy slices tend to be smaller than their strict counterparts.

Therefore, our starting point was a lazy semantics for FL introduced by Launchbury in [6], which is presented in figure 3. In this semantics, expression $\Gamma \vdash e \Downarrow \Delta \vdash z$ states that expression e under heap Γ evaluates to value z producing heap Δ as result.

In figure 3 and throughout the paper the following syntactic abbreviations are used: $\hat{z}$ standing for $\alpha-$conversion, $[x_i \mapsto e_i]$ for $[x_1 \mapsto e_1, \ldots, x_i \mapsto e_i]$, $\Gamma[x_i \mapsto e_i]$ to express the update of mapping $[x_i \mapsto e_i]$ in heap Γ and $e[x_i/y_i]$ for the substitution $e[x_1/y_1, \ldots, x_i/y_i]$.

$$\Gamma \vdash \lambda y.e \Downarrow \Gamma \vdash \lambda y.e \qquad \textit{Lamb}$$

$$\Gamma \vdash C\ x_1 \cdots x_a \Downarrow \Gamma \vdash C\ x_1 \cdots x_a \qquad \textit{Con}$$

$$\frac{\Gamma \vdash e \Downarrow \Delta \vdash \lambda y.e' \qquad \Delta \vdash e'[x/y] \Downarrow \Theta \vdash z}{\Gamma \vdash e\ x \Downarrow \Theta \vdash z} \qquad \textit{App}$$

$$\frac{\Gamma \vdash e \Downarrow \Delta \vdash z}{\Gamma[x \mapsto\ e\ ] \vdash x \Downarrow \Delta[x \mapsto\ z\ ] \vdash \hat{z}} \qquad \textit{Var}$$

$$\frac{\Gamma[x_n \mapsto\ e_n] \vdash e \Downarrow \Delta \vdash z}{\Gamma \vdash \texttt{let}\ \{x_n = e_n\}\ \texttt{in}\ e \Downarrow \Delta \vdash z} \qquad \textit{Let}$$

$$\frac{\Gamma \vdash e \Downarrow \Delta \vdash C_k\ x_1 \cdots x_{a_k} \qquad \Delta \vdash e_k[x_i/y_{ik}] \Downarrow \Theta \vdash z}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{C_j\ y_1 \cdots y_{a_j}\ \texttt{->}\ e_j\}_{j=1}^{n} \Downarrow \Theta \vdash z} \qquad \textit{Case}$$

Figure 3: Lazy Semantics

5 Lazy Forward Slicing

We start by analyzing a simplified version of the more general problem of higher-order lazy functional slicing, which we have called *lazy print*. The calculation of this particular kind of slice is completely based on the lazy evaluation coverage of a program, without taking any extra explicit slicing criterion. This means that a *lazy print* calculation amounts to extracting the program fragment that has some influence on the lazy evaluation of an expression within that program. For an example, consider the following trivial functional program.

```
fst :: (a, b) -> a
fst (x, y) = x

sum :: [Int] -> Int
sum []    = 0
sum (h:t) = h + (sum t)
```

```
g :: ([Int], Int) -> Int
g z = sum (fst z)
```

The *lazy print* of this program w.r.t. the evaluation of `g ([], 3)` is

```
fst :: (a, b) -> a
fst (x, ) = x

sum :: [Int] -> Int
sum []    = 0


g :: ([Int], Int) -> Int
g z = sum (fst z)
```

Automating this calculation entails the need to derive an augmented semantics from the lazy semantics presented in figure 3. This extends Launchbury semantics with an extra output value of type set of labels (S), for the evaluation function $\Downarrow$. The purpose of this set S is to collect all the labels from the expressions that constitute the *lazy print* of a given evaluation. Motivated by implementation reasons, instead of using an alpha conversion in the original rule *Var*, we introduce a fresh variable in rule *Let* to avoid variable clashing.

The *lazy print* semantics uses two auxiliary functions, namely $\varphi : \mathrm{E} \times \mathrm{V} \rightarrow \mathbb{P}\ \mathrm{L}$ and $\mathcal{L} : \mathrm{E} \rightarrow \mathbb{P}\ \mathrm{L}$. Function φ collects the labels from all the occurrences of a variable in an expression and function $\mathcal{L}$ returns all the labels in an expression.

The intuition behind this augmented semantics is that it works by collecting all the labels from the expressions as they are being evaluated by the semantic rules. The only exception is rule *Let*, which does not collect all the expression labels immediately. This is explained by the fact that there is not sufficient information available when rule *Let* is applied to decide which variable bindings will be needed in the remainder of the evaluation towards the computation of the final result. A possible solution for this problem is to have a kind of memory associating pending labels and expressions such that, if an expression gets to be used then not only their labels are included in the evaluation labels set, but also the pending labels that were previously registered in the memory.

A straightforward implementation of such a memory mechanism is the heap itself. Thus, by extending the heap from a mapping between variables and expressions to a mapping from variables to pairs of expressions and sets of labels, the semantics becomes able to capture the "pending labels" introduced by the *Let* rule.

A problem is spotted however in slices computed on top of the *lazy print* semantics given in figure 4. As an example, consider the following fragment that

$$\Gamma \vdash (\lambda y : l_1.e) : l \Downarrow_{\{l_1,l\}} \Gamma \vdash (\lambda y : l_1.e) \qquad \textit{Lamb}$$

$$\Gamma \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l' \Downarrow_{\{l'_k,l'\}} \Delta \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l' \qquad \textit{Con}$$
$$\text{where} \quad k \in \{1, \ldots, a\}$$

$$\frac{\Gamma \vdash e \Downarrow_{S_1} \Delta \vdash (\lambda y : l_1.e') : l_2 \qquad \Delta \vdash e'[x/y] \Downarrow_{S_2} \Theta \vdash z}{\Gamma \vdash e\ (x : l') : l \Downarrow_{S_1 \cup S_2 \cup \{l',l\}} \Theta \vdash z} \qquad \textit{App}$$

$$\frac{\Gamma \vdash e \Downarrow_{S_1} \Delta \vdash z}{\Gamma[x \mapsto < e, L >] \vdash x : l \Downarrow_{S_1 \cup L \cup \{l\}} \Delta[x \mapsto < z, \varepsilon >] \vdash z} \qquad \textit{Var}$$

$$\frac{\begin{array}{c}\Gamma[y_n \mapsto < e_n[y_n/x_n], \{l_n\} \cup \varphi(e, x_n) \cup \varphi(e_n, x_n) \cup \mathcal{L}(e_n) >] \vdash \\ e[y_n/x_n] \Downarrow_{S_1} \Delta \vdash z\end{array}}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow_{S_1 \cup \{l\}} \Delta \vdash z}\ y_n \text{ fresh} \qquad \textit{Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \Downarrow_{S_1} \Delta \vdash (C_k\ x_1 : l^\star_1 \cdots x_{a_k} : l^\star_{a_k}) : l^\sharp_k \\ \Delta \vdash e_k[x_i/y_{i_k}] \Downarrow_{S_2} \Theta \vdash z\end{array}}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{(C_j\ y_1 : l'_1 \cdots y_{a_j} : l'_{a_j}) : l^\natural_j\ \texttt{->}\ e_j\}_{j=1}^{n} : l \Downarrow_S \Theta \vdash z} \qquad \textit{Case}$$
$$\text{where} \quad S = S_1 \cup S_2 \cup \{l^\star{}_{n_j} \mid 1 \leq n \leq a\} \cup \{l'{}_{n_j} \mid 1 \leq n \leq a\} \cup \{l^\sharp_k, l^\natural_j, l\}$$

Figure 4: Lazy Print Semantics

calls some complex and very cohesive functions `funcG` and `funcH` which do indeed contribute to the computation of the values in `x` and `y`:

```
f z w = let x = funcG z w
            y = funcH x z
        in (x, y)
```

When computing the *lazy print* of such a program, no matter what values are chosen for `z w`, the returned slice is always

```
f z w =

        (x, y)
```

as if the variables introduced by the `let` expression would have no effect on the result of the overall function, which completely contradicts what one already knew about the behaviour of functions `funcG` and `funcH`.

The reason for such a deviating behaviour induced by the lazy print semantics is explained by the *Con* rule. Because $C\ x_1 : l_1 \cdots x_a : l_a$ expressions are considered primitive in the language, the *Con* rule simply collects the outer labels of such expressions and returns the expression exactly as it was received.

Indeed this explains the odd behaviour of the above example, where function f returns a pair which falls into the $C\ x_1 : l_1 \cdots x_a : l_a$ representation in FL. Therefore, the only semantic rule being applied during the lazy print calculation was the *Con* rule which does not evaluate the constructor (Pair) arguments and their associated expressions. Thus, one may now understand why the only labels that the semantics yields during the evaluation are the ones visible at the time of application of the *Con* rule.

A possible approach to solve this problem of extra laziness induced by the semantics would be to evaluate every data constructor parameter in a strict way. This, however, would throw away most of the lazy motto of the semantics since the evaluation would become strict on every data type.

A much more effective solution is to divide the slicing calculation into two phases. The first phase uses the semantics from figure 4. The second one takes the value and the heap returned by the first phase and passes them to a processor which restates to a semantics similar to the one used in the first phase except for rule *Con* which is substituted by the one from figure 5.

$$\mathit{Con}\quad \frac{\Gamma[x_k \mapsto < e_k, L_k >] \vdash x_k \Downarrow_{S_1} \Delta \vdash z_k}{\Gamma[x_k \mapsto < e_k, L_k >] \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l' \Downarrow_S \Delta \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l'}$$
$$\text{where}\quad k \in \{1, \ldots, a\},\quad S = L_k \cup \{l'_k, l'\} \cup S_1$$

Figure 5: Con Rule for Strict Evaluation of the Result Value

This way, constructor strict evaluation is introduced only over the resulting value, leaving all intermediate values being evaluated as lazy as possible.

6 Lazy Forward Slicing with Slicing Criterion

However, despite the relevance of the *lazy print* in, e.g., program understanding, a further step towards effective slicing techniques for functional programs requires the explicit consideration of slicing criteria. In this section, we present an approach where slicing criteria is specified by sets of program labels.

The slicing process proceeds as in the previous case, except that now one is interested in collecting the program labels affected not only by a given expression, as before, but also by the expressions associated to the labels introduced by the user as a slicing criterion.

A first and straightforward approach to implement a slicer with such a slicing criterion involves taking into account the set of collected labels on both the output and the input of the evaluation function $\Downarrow$. Therefore, the semantic rule for λ-expressions changes to the one presented by the rule of figure 6.

$$\begin{array}{c} \mathit{Lamb} \\ S_i, \Gamma \vdash (\lambda y : l_1.e) : l \Downarrow \Gamma \vdash (\lambda y : l_1.e) : l, S_f \\ \text{where } S_f = S_i \cup \bigcup\{\varphi(e, y) \mid l_1 \in S_i\} \cup \{l \mid l_1 \in S_i\} \end{array}$$

Figure 6: Improved Semantics

This extra rule enables the semantics to evaluate expressions taking into account a set of labels S_i supplied as a slicing criterion and its impact on the resulting slice S_f. Putting it in another way, each rule has to compute the resulting set of labels S_f considering the effect that the input labels in S_i may have in the slice being computed.

Soon, however, it became difficult to specify the semantic rules taking into account the impact of the receiving set of labels. The problem of specifying the rules is related to the fact that in many cases there is not enough information to enable the decision of including a certain label or not.

For instance, in the *App* rule one may not immediately decide whether to include or not label l_1 in the resulting label set. The reason for this is that one has no means of knowing in advance whether a particular expression in the heap will ever become part of the slice. If such an expression is to be included into the slice, sometime along the remainder of the slicing process, then label l_1 will also belong to the slice as well as all the labels that l_1 affects by the evaluation of the first premiss of rule *App*.

In order to overcome this problem, one should look for some independence in the slicing process over the partial slices that are being calculated by each semantic rule. Thus, instead of calculating partial slices on the application of every rule, one computes partial dependencies between labels. This entails the need for a further modification in the rules which are now intended to compute maps of type $\mathrm{L} \rightarrow \mathbb{P}\ \mathrm{L}$, called *lmap*'s, rather than sets, such that all labels in

Lamb

$$\varGamma \vdash (\lambda y : l_1.e) : l \Downarrow_F \varGamma \vdash (\lambda y : l_1.e) : l$$

where $F = [l_1 \mapsto \varphi(e, y) \cup \{l\}]$

Con

$$\varGamma \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l \Downarrow_F \varGamma \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l$$

where $k \in \{1, \ldots, a\}$, $F = [l_k \mapsto l]$

App

$$\frac{\varGamma \vdash e \Downarrow_F \varDelta \vdash (\lambda y : l_1.e') : l_2 \qquad \varDelta \vdash e'[x/y] \Downarrow_G \varTheta \vdash z}{\varGamma \vdash e\ (x : l') : l \Downarrow_H \varTheta \vdash z}$$

where $H = F \oplus G \oplus [l' \mapsto \{l, l_1\}]$

Var

$$\frac{\varGamma \vdash e \Downarrow_F \varDelta \vdash z}{\varGamma[x \mapsto < e, L >] \vdash x : l \Downarrow_G \varDelta[x \mapsto < z, \varepsilon >] \vdash z}$$

where $G = F \oplus [l \mapsto L]$

Let

$$\frac{\begin{array}{c}\varGamma[y_n \mapsto < e_n[y_n/x_n], \{l_n, l\} \cup \varphi(e, x_n) \cup \varphi(e_n, x_n) >] \vdash \\ e[y_n/x_n] \Downarrow_F \varDelta \vdash z\end{array}}{\varGamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow_G \varDelta \vdash z}$$

where $G = F \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$

Case

$$\frac{\varGamma \vdash e \Downarrow_F \varDelta \vdash (C_k\ x_1 : l^\star_1 \cdots x_{a_k} : l^\star_{a_k}) : l^\sharp_k \qquad \varDelta \vdash e_k[x_i/y_{ik}] \Downarrow_G \varTheta \vdash z}{\varGamma \vdash \texttt{case}\ e\ \texttt{of}\ \{(C_j\ y_1 : l'_1 \cdots y_{a_j} : {l'}_{a_j}) : l^\natural_j\ \texttt{->}\ e_j\}_{j=1}^n : l \Downarrow_H \varTheta \vdash z}$$

where $G = F \oplus G \oplus [l^\star_m \mapsto \varphi(e_k, y_m) \cup \{l'_m, l^\natural_k\} | 1 \le m \le a_k] \oplus$
$[l^\natural_k \mapsto \{l\}] \oplus [l'_m \mapsto \varphi(e_k, y_m) \cup \{l^\natural_k\} | 1 \le m \le a_k]$

Figure 7: Higher-Order Slicing Semantics

the codomain depend on the labels in the domain. The resulting semantics is presented in figure 7 where in rule *Let* variable y_n is a fresh variable.

In the sequel the following three operations over *lmap*'s are required: an application operation, resorting to standard finite function application, defined by

$$F(x) = \begin{cases} F\ x & \text{if } x \in dom\ F, \\ \{\} & \text{otherwise.} \end{cases}$$

$$\frac{\mathit{Con}\qquad \Gamma[x_k \mapsto < e_k, L_k >] \vdash x_k \Downarrow_{F_k} \Delta \vdash z_k}{\Gamma[x_k \mapsto < e_k, L_k >] \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l' \Downarrow_G \Delta \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l'}$$
$$\text{where} \quad k \in \{1, \ldots, a\}, \quad G = F_k \oplus [l'_k \mapsto l']$$

Figure 8: Con Rule for Strict Evaluation of the Result Value

a *lmap* multiplication $\oplus$, defined as

$$(F \oplus G)(x) = F(x) \cup G(x)$$

and, finally, a range union operation *urng*, defined as

$$urng\ F = \bigcup_{x \in dom\ F} F(x)$$

Again, this semantics suffers from the problem identified in the *lazy print* specification i.e., the semantics is "too lazy". Once more, to overcome such undesired effect, one introduces a new rule (Fig. 8) to replace the original *Con* rule, and the slicing process is similarly divided into two phases.

By changing the output of the evaluation function from a set to a *lmap* of labels, we no longer have a slice of the program by the end of the evaluation. What is returned, instead, is a *lmap* specifying the different dependencies between the different expressions that form the program under analysis. The desired slice is computed as the transitive closure of such dependencies *lmap*.

Furthermore, splitting the slicing process into a dependencies calculation and the computation of a slice for the set of pertinent labels makes easier the calculation of slices that only differ on the set of pertinent labels. For such cases, one can rely on a common dependencies *lmap* and the whole process amounts to the calculation of the transitive closure for redefined set of labels.

7 Strict Evaluation

Slicing under strict evaluation is certainly easier. A possible semantics, as the one considered in figure 9, can be obtained by a systematic simplification of the one used in the lazy case. Of course, this is not the only possibility. To make comparisons possible between the lazy and strict case, however, we chose to keep specification frameworks as similar as possible, although we are aware that many details in the strict side could have been simplified. For example, strict semantics can always return slices in the form of sets of labels instead of calculating maps capturing dependencies between code entities.

Lamb

$$\mathit{\Gamma} \vdash (\lambda y : l_1.e) : l \Downarrow_F \mathit{\Gamma} \vdash (\lambda y : l_1.e) : l$$

where $F = [l_1 \mapsto \varphi(e, y) \cup \{l\}]$

Con

$$\mathit{\Gamma} \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l \Downarrow_F \mathit{\Gamma} \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l$$

where $k \in \{1, \ldots, a\}$
$F = [l_k \mapsto l]$

App

$$\frac{\mathit{\Gamma} \vdash e \Downarrow_F \mathit{\Delta} \vdash (\lambda y : l_1.e') : l_2 \qquad \mathit{\Delta} \vdash e'[z_1/y] \Downarrow_G \mathit{\Theta} \vdash z}{\mathit{\Gamma}[x \mapsto\ z_1] \vdash e\ (x : l') : l \Downarrow_H \mathit{\Theta} \vdash z}$$

where $H = F \oplus G \oplus [l' \mapsto \{l, l_1\}]$

Var (whnf)

$$\frac{\mathit{\Gamma} \vdash z \Downarrow_F \mathit{\Delta} \vdash z}{\mathit{\Gamma}[x \mapsto\ z] \vdash x : l \Downarrow_G \mathit{\Delta}[x \mapsto\ z\ ] \vdash z}$$

where $G = F$

Let

$$\frac{\mathit{\Gamma} \vdash e_n \Downarrow_F \mathit{\Delta} \vdash z_n \qquad \mathit{\Gamma}[y_n \mapsto z_n] \vdash e[z_n/x_n] \Downarrow_G \mathit{\Delta} \vdash z}{\mathit{\Gamma} \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow_H \mathit{\Delta} \vdash z}\ y_n \text{ fresh}$$

where $H = F \oplus G \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$

Case

$$\frac{\mathit{\Gamma} \vdash e \Downarrow_F \mathit{\Delta} \vdash (C_k\ x_1 : l_1^\star \cdots x_{a_k} : l_{a_k}^\star) : l_k^\sharp \qquad \mathit{\Delta} \vdash e_k[x_i/y_{ik}] \Downarrow_G \mathit{\Theta} \vdash z}{\mathit{\Gamma} \vdash \texttt{case}\ e\ \texttt{of}\ \{(C_j\ y_1 : l_1' \cdots y_{a_j} : l'_{a_j}) : l_j^\natural\ \texttt{->}\ e_j\}_{j=1}^n : l \Downarrow_H \mathit{\Theta} \vdash z}$$

where $G = F \oplus G \oplus [l_m^\star \mapsto \varphi(e_k, y_m) \cup \{l_m', l_k^\natural\} | 1 \leq m \leq a_k] \oplus$
$[l_k^\natural \mapsto \{l\}] \oplus [l_m' \mapsto \varphi(e_k, y_m) \cup \{l_k^\natural\} | 1 \leq m \leq a_k]$

Figure 9: Strict Slicing Semantics

Moreover, in the strict case there is no need to capture pending labels in the heap, since `let` expressions are evaluated as soon as they are found. This leads to a simplification of the heap from a mapping between variables and pairs of expressions and set of labels to a mapping between variables and values.

As for the rules, the *App* and *Let* rules need to be changed, along with some minor adaptation of the rules that deal with the (newly modified) heap.

Another decision taken in the strict slicing semantics specification was to keep value sharing *i.e.*, sharing of values that are stored in the heap. Nevertheless, one can easily derive a slicing semantics without any sharing mechanism, for which case one could probably remove the heap from the semantics.

Finally note that now there is no need to introduce a new *Con* rule to force the evaluation of unevaluated expressions inside result value. Thus, unlike the two previous versions of lazy slicing, strict slicing is accomplished in a single evaluation phase.

8 Some Considerations About the Slicing Processes

All slicing algorithms presented in this paper were introduced as (evaluators of) a specific semantics. Such an approach provides an expressive setting on top of which one may reason formally about slices and slicers. This is illustrated in this section to confirm the intuitive fact that, in general, lazy slices are smaller than strict slices.

In the case of the *lazy print* semantics, such a proof amounts to showing that the set of labels returned by the lazy print is a subset of the set of labels yielded by an hypothetical strict print semantics.

But, since both the higher-order lazy slicing semantics and the strict one do not return sets of labels but maps of dependencies, one has to restate the proof accordingly. This can be achieved in two ways: either including the final transitive closure calculation in the slicing process, or introducing a partial order over the dependency *lmap*'s that respects subset inclusion.

We chose the latter alternative, and introduce the following partial order over *lmap*'s, which is the standard inclusion order on partial functions.

$$F \preceq G \Leftrightarrow dom(F) \subseteq dom(G) \wedge (\forall x \in dom(F).F(x) \subseteq G(x))$$

Now, the property that "lazy slices are smaller than strict slices" is formulated as follows.

If $\Gamma \vdash e \Downarrow_F \Delta \vdash z$ and $\Gamma \vdash e \downdownarrows_G \Theta \vdash z$ then $F \preceq G$

The proof proceeds by induction over the rule-based semantics. First notice that the property is trivially true for all identical rules in both semantics. Such as the cases of rules *Lamb*, *Con* and *Case* for which the resulting *lmap*'s are equal. The remaining cases follows.

Case *App*: Evaluation of expressions under these rules take the following form, according to the evaluation strategy used.

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash (\lambda y : l_1.e') : l_2 \qquad \Delta \vdash e'[x/y] \Downarrow_G \Psi \vdash z}{\Gamma \vdash e\ (x : l') : l \Downarrow_H \Theta \vdash z} \; App$$
$$\text{where} \quad H = F \oplus G \oplus [l' \mapsto \{l, l_1\}]$$

$$\frac{\Gamma \vdash e \Downarrow\!\Downarrow_I \Theta \vdash (\lambda y : l_1.e') : l_2 \qquad \Theta \vdash e'[z_1/y] \Downarrow\!\Downarrow_J \Phi \vdash z}{\Gamma[x \mapsto z_1] \vdash e\ (x : l') : l \Downarrow\!\Downarrow_K \Phi \vdash z} \; App$$
$$\text{where} \quad K = I \oplus J \oplus [l' \mapsto \{l, l_1\}]$$

By induction hypothesis one has that $F \preceq I$. By definition of *Let* rule, which is the only rule that changes the heap, one has that $\mathcal{L}(\Delta) \cup urng\ F = \mathcal{L}(\Theta) \cup urng\ I$, where function $\mathcal{L}$ is overloaded to collect all the labels of the expressions in a heap. It follows that

$$
\begin{array}{ll}
 & \mathcal{L}(\Delta) \cup urng\ F = \mathcal{L}(\Theta) \cup urng\ I \\
\Rightarrow & \quad \{\text{Induction Hypothesis}\} \\
 & \mathcal{L}(\Delta) \setminus \mathcal{L}(\Theta) \subseteq urng\ I \\
\Rightarrow & \quad \{\text{Defintion of } \oplus \text{, noting that every possible label that } G \\
 & \quad \text{may collect from heap } \Delta \text{ is already in } I\} \\
 & G \preceq I \oplus J \\
\Rightarrow & \quad \{\text{Induction Hypothesis}\} \\
 & F \preceq I \wedge G \preceq I \oplus J \\
\Rightarrow & \quad \{\text{Definition of } \oplus\} \\
 & F \oplus G \preceq I \oplus J \\
\Rightarrow & \quad \{\text{Definition of } \oplus\} \\
 & F \oplus G \oplus [l' \mapsto \{l, l_1\}] \preceq I \oplus J \oplus [l' \mapsto \{l, l_1\}] \\
\Rightarrow & \quad \{\text{Defintion of } G \text{ and } H\} \\
 & G \preceq H
\end{array}
$$

Case *Let*: Evaluation of expressions under these rules takes the following format, according to the evaluation strategy used (note that y_n is a fresh variable in both rules).

$$\frac{\Gamma[y_n \mapsto < e_n[y_n/x_n], \{l_n, l\} \cup \varphi(e, x_n) \cup \varphi(e_n, x_n) >] \vdash e[y_n/x_n] \Downarrow_F \Delta \vdash z}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow_G \Delta \vdash z}\ \mathit{Let}$$

where $G = F \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$

$$\frac{\Gamma \vdash e_n \Downarrow\!\Downarrow_H \Theta \vdash z_n \qquad \Gamma[y_n \mapsto z_n] \vdash e[z_n/x_n] \Downarrow\!\Downarrow_I \Phi \vdash z}{\Gamma \vdash \texttt{let}\ \{x_n = e_n : l_n\}\ \texttt{in}\ e : l \Downarrow\!\Downarrow_J \Phi \vdash z}\ \mathit{Let}$$

where $J = H \oplus I \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)]$

By induction hypothesis and because $\mathcal{L}(e_n) \subseteq \mathit{urng}\ H$ one has that $F \preceq H \oplus I$. It follows that

$$\begin{aligned}
& G = F \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)] \\
\Rightarrow & \qquad \{F \preceq H \oplus I\} \\
& G \preceq H \oplus I \oplus [l_n \mapsto \{l\}] \oplus [y \mapsto \varphi(e, x_n) \cup \varphi(e_n, x_n) \mid y \in \mathcal{L}(e_n)] \\
\Rightarrow & \qquad \{\text{Definition of } K\} \\
& G \preceq K
\end{aligned}$$

Case *Var*: Evaluation of expressions under these rules take the following form, according to the evaluation strategy used.

$$\frac{\Gamma \vdash e \Downarrow_F \Delta \vdash z}{\Gamma[x \mapsto < e, L >] \vdash x : l \Downarrow_G \Delta[x \mapsto < z, \varepsilon >] \vdash z}\ \mathit{Var}$$

where $G = F \oplus [l \mapsto L]$

$$\frac{\Gamma \vdash z \Downarrow\!\Downarrow_H \Delta \vdash z}{\Gamma[x \mapsto\ z] \vdash x : l \Downarrow\!\Downarrow_I \Delta[x \mapsto\ z\ ] \vdash z}\ \mathit{Var}$$

where $I = H$

By induction hypothesis one has that $F \preceq H$. Since the only way to add entries to the heap is via the *Let* rule, and because, in strict semantics, such rule increments the dependencies *lmap* with every label from the newly introduced expressions, it follows that increments to the strict evaluation *lmap* will contain every mapping that is pending on the modified higher-order slicing heap. Thus, even though it may happen that at the time of evaluation of the *Var* rule, one may have $I \preceq G$, in the overall evaluation tree the dependency *lmap* for the lazy evaluation is always smaller or equal to the strict evaluation *lmap*.

9 Conclusions and Future Work

This paper introduced a semantic-based approach to low level slicing of functional programs, highlighting a strong relationship between the slicing problem and the underlying evaluation strategy.

Due to space restrictions, we have not been able to expose here a real code example that would highlight the strengths of the presented method. Actually, a realistic application example needs to have at least one large (more than 20 lines) function with several calls to other functions which would also had to be presented in order to achieve an understandable practical example. Nevertheless, we have tested the method against the HASKELL implementation of the semantics presented in section 6, and the interested reader may consult the results at `http://www.di.uminho.pt/~nfr/Results/HoSlicingResults.html`. The need for examples with large function definitions to demonstrate our method capabilities is because in such cases one can point out as a slicing criterion a tag indicating the particular (sub)expression inside the large function definition, and by doing so one can more precisely identify the relevant part of the code that he/she is interested in. Other approaches that rely on slicing criterion defined by a return value cannot achieve slices as precise as ours.

Although the techniques introduced here are oriented to forward slicing, we strongly believe that a correct inversion of the dependencies *lmap*'s, followed by the same transitive closure calculation, will capture the backward cases.

This research adds to our previous work on high level functional slicing *i.e.*, slicing defined over "high level" program entities such as functions, modules, or data-types, as documented in [10]. Reference [9] reports on a completely alternative approach to the slicing problem based on the so called Bird-Meertens calculus. The common context of this research effort is a project on program understanding and re-engineering[3], currently running at Minho University, Portugal.

Acknowledgements

The research reported in this paper is supported by FCT, under contract POSI-ICHS-44304-2002, in the context of the PURe project.

References

1. R. Bird. *Functional Programming Using Haskell.* Series in Computer Science. Prentice-Hall International, 1998.
2. S. K. Biswas. *Dynamic slicing in higher-order programming languages.* PhD thesis, 1997. Supervisor-Carl A. Gunter.
3. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conf. on Programming Usage, Design and Implementation*, pages 35–46. ACM Press, 1988.
4. B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
5. B. Korel and J. Laski. Dynamic slicing of computer programs. *J. Syst. Softw.*, 13(3):187–195, 1990.

[3] http://wiki.di.uminho.pt/twiki/bin/view/PURe

6. J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.
7. C. Ochoa, J. Silva, and G. Vidal. Dynamic slicing based on redex trails. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 123–134, New York, NY, USA, 2004. ACM Press.
8. T. W. Reps and T. Turnidge. Program specialization via program slicing. In *Selected Papers from the International Seminar on Partial Evaluation*, pages 409–429, London, UK, 1996. Springer-Verlag.
9. N. Rodrigues and L. Barbosa. Program slicing by calculation. *Journal of Universal Computer Science*, 12(7):828–848, 2006.
10. N. Rodrigues and L. S. Barbosa. Component identification through program slicing. In L. S. Barbosa and Z. Liu, editors, *Proc. of FACS'05 (2nd Int. Workshop on Formal Approaches to Component Software)*, volume 160, pages 291–304, UNU-IIST, Macau, 2006. Elect. Notes in Theor. Comp. Sci., Elsevier.
11. M. Weiser. *Program Slices: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Methods*. PhD thesis, University of Michigan, An Arbor, 1979.
12. M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.
13. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.