Lazy Functional Slicing

CIC'06

Nuno Rodrigues
nfr@di.uminho.pt

Agenda

- What is Program Slicing?
- Functional Slicing
  - High Level Functional Slicing
  - Low Level Functional Slicing
- Conclusions and Future Work

What is Program Slicing ?

- The original concept was introduced by Weiser 1979.
- “A program slice S is a ***reduced***, ***executable*** *program* obtained from a program P by **removing statements**, such that S **replicates part of the behaviour** of P”
- Other notions of program slices have been purposed. Mainly because different applications require different properties of slices.
- Program slicing consists in isolating a specific part of a program using some choice criterion

Example – criterion(10, product)

```
1:  read(n)
2:  i :=1 ;
3:  sum := 0;
4:  product := 1;
5:  while i <= n
    {
6:      sum := sum + i;
7:      product := product * i;
8:      i := i + 1;
    }
9:  write(sum);
10: write(product);
```

```
1:  read(n)
2:  i :=1 ;
3:  sum := 0;
4:  product := 1;
5:  while i <= n
    {
6:      sum := sum + i;
7:      product := product * i;
8:      i := i + 1;
    }
9:  write(sum);
10: write(product);
```

Applications of Program Slicing

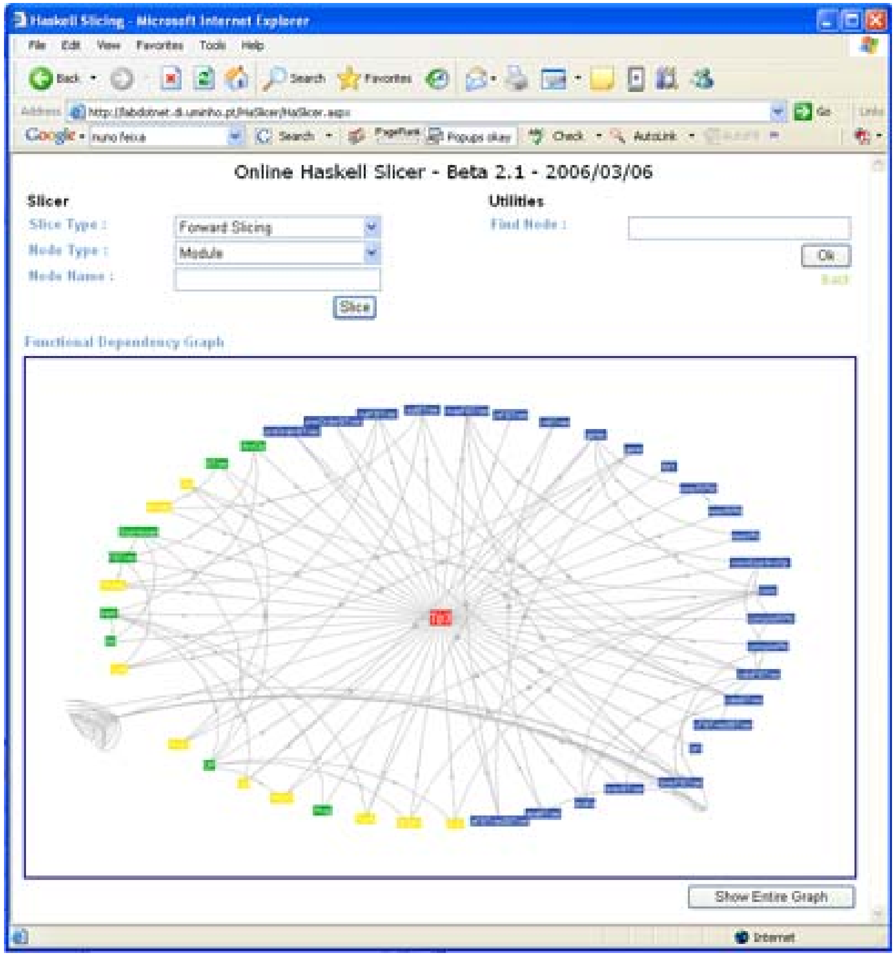
- Debugging
- Maintenance
- Parallelization
- Model Checking
- Security Analysis
- Reverse engineering
- Program Comprehension
- Program Restructuring, Refactoring
- …

Functional Slicing Problems

- Non trivial control flow
  - Depends largely on the values being evaluated
  - In a complete static analysis, the CFG is enormous
- Polymorphism
  - Data dependencies may not be explicit
- High Order functions
  - Function dependencies may not be explicit
- Functional programs are not oriented to code line. Thus, the approach taken by most of the existing slicing techniques doesn’t apply.
- Depends on the evaluation strategy

High Level Functional Slicing

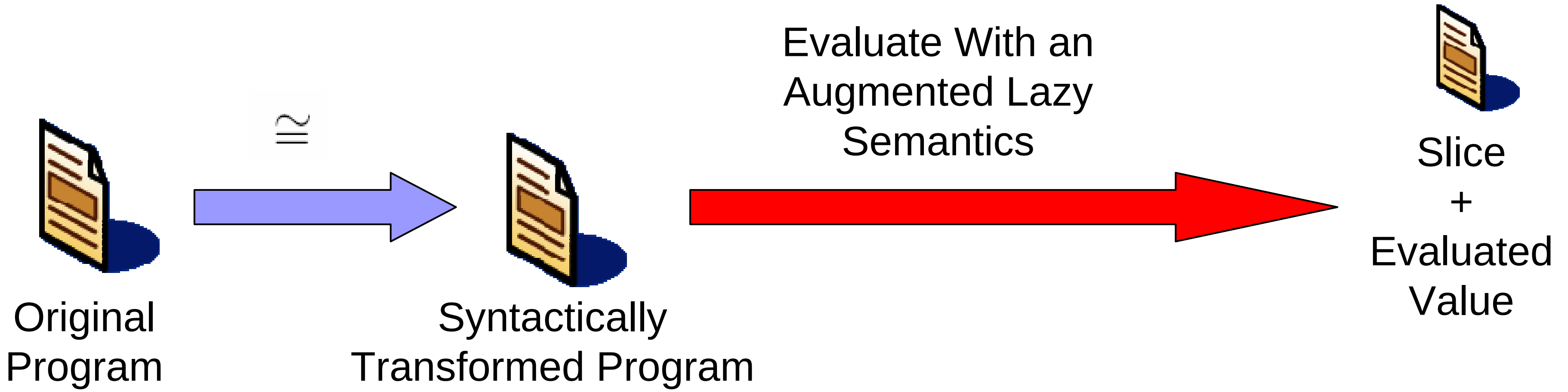
- HaSlicer
  - Fully functional Haskell slicer
  - Based on FDG
  - Slices high order entities
    - Modules
    - Data Types
    - Functions
  - Several applications
    - Component Discovery
  - Doesn’t go to the statement/expression level



http://labdotnet.di.uminho.pt/HaSlicer/HaSlicer.aspx

Low Level Functional Slicing

- Given that slicing depends on the evaluation strategy, we choose lazy evaluation



Syntactic Transformation

Values	$z ::= (\lambda x : l_1 . e) : \breve{l}$	
	$\mid\ (C\ x_1 : l_1 \cdots x_a : l_a) : \breve{l}$	$a \geq 0$
Expressions	$e ::= z$	
	$\mid\ e\ (x : l') : \breve{l}$	
	$\mid\ x : l$	
	$\mid\ \texttt{let}\ x_n = e_n : l_n\ \texttt{in}\ e : \breve{l}$	$n > 0$
	$\mid\ \texttt{case}\ e\ \texttt{of}\ \{(C_j\ x_{1j} : l_{1j} \cdots x_{aj} : l_{aj}) : \breve{l}' \texttt{->}\ e_j\}_{j=1}^{n} : \breve{l}$	$n > 0,\ a \geq 0$
Programs	$prog ::= x_1 = e_1, \ldots, x_n = e_n$	

- Functional application involves an expression and a variable
  - There are no expression-expression applications
- *If then else* statements are substituted by case expressions
- Every expression is tagged
  - Place in code
  - Kind of transformation

The Lazy Semantics

- John Launchbury

$$\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textit{Lambda}$$

$$\frac{\Gamma : e \Downarrow \Delta : z}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto z) : \hat{z}} \qquad \textit{Variable}$$

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

$$\frac{(\Gamma, x_1 \mapsto e_1 \ \cdots\ x_n \mapsto e_n) : e \Downarrow \Delta : z}{\Gamma : let\ x_1 = e_1 \ \cdots\ x_n = e_n\ in\ e \Downarrow \Delta : z} \qquad \textit{Let}$$

$$\Gamma : c\ x_1 \cdots x_n \Downarrow \Gamma : c\ x_1 \cdots x_n \qquad \textit{Constructors}$$

$$\frac{\Gamma : e \Downarrow \Delta : c_k\ x_1 \cdots x_{m_k} \qquad \Delta : e_k[x_i/y_i]_{i=1}^{m_k} \Downarrow \Theta : z}{\Gamma : case\ e\ of\ \{c_i\ y_1 \cdots y_{m_i} \to e_i\}_{i=1}^{n} \Downarrow \Theta : z} \qquad \textit{Case}$$

Lazy Slicing Without Criterion

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

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

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

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

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

Lazy Slicing WSC (continued)

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

$$\frac{\Gamma \vdash e \Downarrow_{S_1} \Delta \vdash (C_k\ x_1 : l_1^\star \cdots x_{a_k} : l_{a_k}^\star) : l_k^\sharp \qquad \Delta \vdash e_k[x_i/y_{ik}] \Downarrow_{S_2} \Theta \vdash z}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{(C_j\ y_1 : l_1' \cdots y_{a_j} : l'_{a_j}) : l_j^\natural\ \texttt{->}\ e_j\}_{j=1}^n : l \Downarrow_S \Theta \vdash z} \quad \mathit{Case}$$

where $S = S_1 \cup S_2 \cup \{l^\star{}_{n_j} \mid 1 \le n \le a\} \cup \{l'{}_{n_j} \mid 1 \le n \le a\} \cup \{l_k^\sharp, l_j^\natural, l\}$

The evaluation is too lazy!!!

```
let x = complexF y z w
     y = complexG w k z
in (x, y)
```

Slice

```
let x = complexF y z w
     y = complexG w k z
in (x, y)
```

- So we need an extra rule to put it to work

$$\frac{\Gamma[x_k \mapsto < e_k, L_k >] \vdash x_k \Downarrow_{S_1} \Delta \vdash z_k}{\Gamma[x_k \mapsto < e_k, L_k >] \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l' \Downarrow_S \Delta \vdash (C\ x_1 : l'_1 \cdots x_a : l'_a) : l'}\ Con$$

$$\text{where} \quad k \in \{1, \ldots, a\}$$
$$S = L_k \cup \{l'_k, l'\} \cup S_1$$

Remarks about Lazy Slicing WSC

- It slices not only the program code, but also the values passed to it.
  - Good to inspect what and how much values are being consumed by the program
  - Good to inspect some special case values behaviours: empty list, Nothing, negative integers, etc…
- Slices tend to be rather big.
  - Specially with programs developed by experienced functional programmers and with a wide range of values
- Possibly interesting for education….

Example

```
foo x y z = fst (len (app x y), snd z)

len k = case k of
           []      -> Z
           (y:ys) -> Succ (len ys)

app m n = case m of
             []      -> n
             (z:zs) -> z : (app zs n)

fst (p, q) = p

snd (x, y) = y

p =(0, 1)
a = []
b = [1,2,3]

*> foo a b p
```

Example

```
foo x y z = fst (len (app x y), snd z)

len k = case k of
            []      -> Z
            (y:ys) -> Succ (len ys)

app m n = case m of
              []      -> n
              (z:zs) -> z : (app zs n)

fst (p, q) = p

snd (x, y) = y

p =(0, 1)
a = []
b = [1,2,3]

*> foo a b p
```

Lazy Slicing With Slicing Criterion

$$S_i, \Gamma \vdash (\lambda y : l_1.e) : l \Downarrow \Gamma \vdash (\lambda y : l_1.e) : l, S_f \qquad Lamb$$

$$\text{where } S_f = S_i \cup \bigcup\{\varphi(e,y) \mid l_1 \in S_i\} \cup \{l \mid l_1 \in S_i\}$$

$$S_i, \Gamma[x \mapsto < e_k, L_k >] \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l \Downarrow \Delta[x \mapsto < e_k, L_k \cup \{l_k, l\} >] \vdash (C\ x_1 : l_1 \cdots x_a : l_a) : l, S_f \quad Con$$

$$\begin{aligned} \text{where} \quad k &\in \{1, \ldots, a\} \\ S_f &= S_i \cup \{l \mid \exists p \in \{1, \ldots, a\}\ .\ l_p \in S_i\} \end{aligned}$$

$$\frac{S_i, \Gamma \vdash e \Downarrow \Delta \vdash (\lambda y : l_1.e') : l_2, S_{\lambda f} \qquad S'_i, \Delta \vdash e'[x/y] \Downarrow \Theta \vdash z, S_{zf}}{S_i, \Gamma \vdash e\ (x : l') : l \Downarrow \Theta \vdash z, S_f} \quad App$$

$$\begin{aligned} \text{where} \quad S'_i &= S_{\lambda f} \cup \bigcup\{\varphi(e', y) \cup \{l_1, l_2\} \mid l' \in S_i\} \\ S_f &= S_{zf} \cup \{l \mid l' \in S_z f\} \end{aligned}$$

Calculate a Slicing Function

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

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

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

$$\begin{array}{ll} \text{where} & k \in \{1, \ldots, a\} \\ & F = [l_k \mapsto l] \end{array}$$

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

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

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

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

Calculating the slice

- Given a Slicing Criterion x = {sc} and a Slicing function *F*, the slice can be calculated by

$$\mu x \,.\, F\, x \cup x$$

- More slices can be easily calculated reusing slicing function *F*

Example

```
foo x y z = fst (len (app x y), snd z)

len k = case k of
           []     -> Z
           (y:ys) -> Succ (len ys)

app m n = case m of
             []     -> n
             (z:zs) -> z : (app zs n)

fst (p, q) = p

snd (x, y) = y

p =(0, 1)
a = []
b = [1,2,3]

*> foo a b p
```

Example

```
foo x y z = fst (len (app x y), snd z)

len k = case k of
           []     -> Z
           (y:ys) -> Succ (len ys)

app m n = case m of
            []     -> n
            (z:zs) -> z : (app zs n)

fst (p, q) = p

snd (x, y) = y

p =(0, 1)
a = []
b = [1,2,3]

*> foo a b p
```

Conclusions and Future Work

- The methods are implemented and working with good results for small tests
- Backward Slicing
  - Just invert the slicing function
- Static Slicing
  - No ideas, yet.
- Implement an automatic translator for the syntactic transformation
- Prove that the slicing process doesn’t introduce non-termination (or not)