COMP4075/G54RFP: Lecture 13 & 14

Functional Programming with Structured Graphs

Henrik Nilsson

University of Nottingham, UK

Overview (1)

While pure languages are excellent for expressing computation over tree-like structures . . .

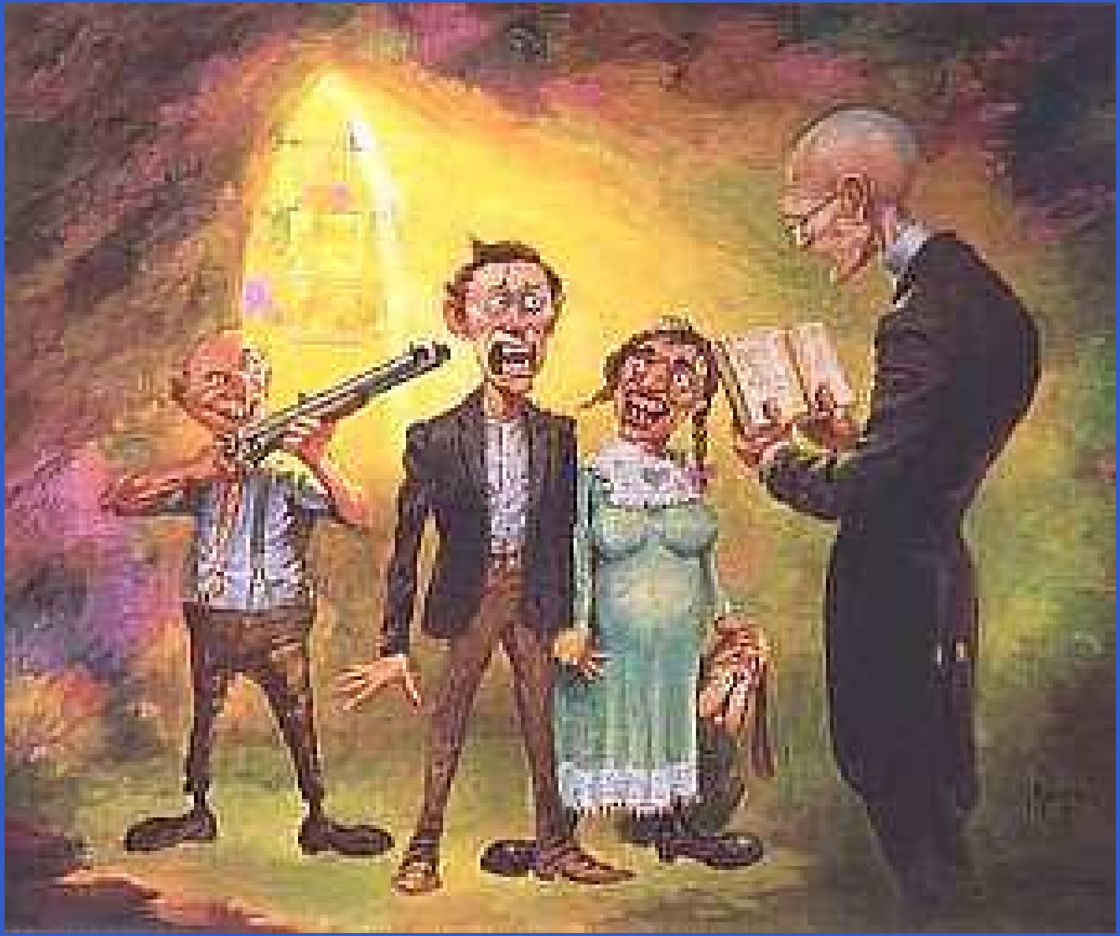
Overview (1)

While pure languages are excellent for expressing computation over tree-like structures . . .

. . . you could even argue it's a match made in heaven . . .

Overview (2)

…pure languages and graphs is more of a shotgun wedding:



Overview (3)

Summary of approaches:

Overview (3)

Summary of approaches:

- Resort to an essentially imperative formulation; e.g.:
  - Monads for structure and possibly performance (King and Launchbury 1994)
  - Clever tricks exploiting lazy evaluation (Johnsson 1998)

Overview (3)

Summary of approaches:

- Resort to an essentially imperative formulation; e.g.:
  - Monads for structure and possibly performance (King and Launchbury 1994)
  - Clever tricks exploiting lazy evaluation (Johnsson 1998)
- Implicitly exploiting the implementation-level graph structure of lazy evaluation
  - Limited applicability and fragile (Hughes 1985)

Overview (4)

- Explicitly exploiting the lazy evaluation graph structure
  - Impure, fragile (Gill 2009)

Overview (4)

- Explicitly exploiting the lazy evaluation graph structure
  - Impure, fragile (Gill 2009)
- Inductive graphs
  - Elegant, but imperative features needed in library implementation to realise standard asymptotic time complexity (Erwig 2001)
  - Foundation of the package FGL (Functional Graph Lipbrary) which is current.

Overview (5)

Oliveira & Cook (2012) propose a novel approach for ***structured graphs***:

Overview (5)

Oliveira & Cook (2012) propose a novel approach for ***structured graphs***:

- Key idea: account for sharing and cycles using ***parametric higher-order abstract syntax*** (PHOAS) (Chlipala 2008)

Overview (5)

Oliveira & Cook (2012) propose a novel approach for ***structured graphs***:

- Key idea: account for sharing and cycles using ***parametric higher-order abstract syntax*** (PHOAS) (Chlipala 2008)
- Similar ideas have been explored in the past (e.g.: Fegaras & Sheard 1996; Ghani, Hamana, Uustalu, Vene 2006), but none is as flexible or easy to use.

Overview (6)

- Good fit for functional programming (e.g. Haskell, Agda):

Overview (6)

- Good fit for functional programming (e.g. Haskell, Agda):
  - graphs can be seen as extension of algebraic data types

Overview (6)

- Good fit for functional programming (e.g. Haskell, Agda):
  - graphs can be seen as extension of algebraic data types
  - amenable to conventional functional programming and reasoning techniques (e.g., folds, induction)

Overview (6)

- Good fit for functional programming (e.g. Haskell, Agda):
  - graphs can be seen as extension of algebraic data types
  - amenable to conventional functional programming and reasoning techniques (e.g., folds, induction)
  - relatively light-weight; does not assume too exotic language features (rank 2 types)

Structured Graphs?

So what ***is*** a structured graph, then?

Structured Graphs?

So what ***is*** a structured graph, then?

Oliveira & Cook:

> *Structured graphs can be viewed as an extension of conventional algebraic datatypes that allow explicit definition and manipulation of cycles or sharing by using recursive binders and variables to explicitly represent possible sharing points.*

Structured Graphs? Take 2

A structured graph is a directed graph where:

Structured Graphs? Take 2

A structured graph is a directed graph where:

- the nodes are grouped into a ***hierarchy of regions***;

Structured Graphs? Take 2

A structured graph is a directed graph where:

- the nodes are grouped into a ***hierarchy of regions***;
- one or more designated ***named nodes*** in a region are the only possible targets for ***back-edges*** and ***cross-edges*** from nodes ***within*** that region (and its sub-regions).

Structured Graphs? Take 2

A structured graph is a directed graph where:

- the nodes are grouped into a ***hierarchy of regions***;
- one or more designated ***named nodes*** in a region are the only possible targets for ***back-edges*** and ***cross-edges*** from nodes ***within*** that region (and its sub-regions).

Think of ***scope*** in programming language terms, which is where PHOAS enters the picture, leveraging the host language to enforce the above constraints and facilitate the manipulation of such graphs.

Higher-order Abstract Syntax (HOAS)

Conventional representation of λ-terms:

$$\begin{array}{l}\mathbf{data}\ \mathit{Term} = \\ \quad\mid \mathit{Var}\ \mathit{Id} \\ \quad\mid \mathit{Lam}\ \mathit{Id}\ \mathit{Term} \\ \quad\mid \mathit{App}\ \mathit{Term}\ \mathit{Term}\end{array}$$

Higher-order Abstract Syntax (HOAS)

Conventional representation of λ-terms:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Term} = \\
&\qquad \mid\ \mathit{Var}\ \mathit{Id} \\
&\qquad \mid\ \mathit{Lam}\ \mathit{Id}\ \mathit{Term} \\
&\qquad \mid\ \mathit{App}\ \mathit{Term}\ \mathit{Term}
\end{aligned}
$$

HOAS representation of λ-terms:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Term} = \\
&\qquad \ \mathit{Lam}\ (\mathit{Term} \rightarrow \mathit{Term}) \\
&\qquad \mid\ \mathit{App}\ \mathit{Term}\ \mathit{Term}
\end{aligned}
$$

Parametric HOAS

HOAS representation of λ-terms:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Term} = \\
&\qquad \mathit{Lam}\ (\mathit{Term} \to \mathit{Term}) \\
&\quad\ \mid \mathit{App}\ \mathit{Term}\ \mathit{Term}
\end{aligned}
$$

Parametric HOAS

HOAS representation of λ-terms:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Term} = \\
&\qquad \mathit{Lam}\ (\mathit{Term} \to \mathit{Term}) \\
&\quad\ \mid \mathit{App}\ \mathit{Term}\ \mathit{Term}
\end{aligned}
$$

PHOAS representation of λ-terms:

$$
\begin{aligned}
&\textbf{data}\ \mathit{PTerm}\ a = \\
&\qquad \mathit{Var}\ a \\
&\quad\ \mid \mathit{Lam}\ (a \to \mathit{PTerm}\ a) \\
&\quad\ \mid \mathit{App}\ (\mathit{PTerm}\ a)\ (\mathit{PTerm}\ a) \\
&\textbf{newtype}\ \mathit{Term} = \downarrow \{\uparrow :: \forall a\ .\ \mathit{PTerm}\ a\}
\end{aligned}
$$

Advantages of PHOAS

- Well-scopedness guaranteed (parametricity)
- No explicit environments
- Easy to define operations; in particular, HOAS often necessitates a function ***reify***: the inverse of the operation being defined.

Recursive PHOAS Binders (1)

Recursive binders can easily be added and given a fixed-point semantics. E.g., evaluation of λ-terms:

$$\mathbf{data}\ PTerm\ a = Mu_1\ (a \to PTerm\ a) \mid \ldots$$

$$peval :: PTerm\ Value \to Value$$

$$\ldots$$

$$peval\ (Mu_1\ f) = fix\ (peval \circ f)$$

Recursive PHOAS Binders (1)

Recursive binders can easily be added and given a fixed-point semantics. E.g., evaluation of λ-terms:

$$\mathbf{data}\ PTerm\ a = Mu_1\ (a \to PTerm\ a) \mid \ldots$$

$$peval :: PTerm\ Value \to Value$$

$$\ldots$$

$$peval\ (Mu_1\ f) = \mathit{fix}\ (peval \circ f)$$

(Intuition: When applied to a $Value$, f returns a $PTerm\ Value$ representing the the body of f with the $Value$ substituted for the function argument; evaluation of that term yields the $Value$ we applied f to in the first place; i.e. the ***fixed point***.)

Recursive PHOAS Binders (2)

Or a $\mathtt{letrec}$-like construct:

$$\mathbf{data}\ PTerm\ a = Mu_2\ ([\,a\,] \to [\,PTerm\ a\,])\ |\ \ldots$$

$$peval :: PTerm\ Value \to Value$$

$$\ldots$$

$$peval\ (Mu_2\ f) = head\ \$\ fix\ (map\ peval \circ f)$$

Recursive PHOAS Binders (2)

Or a $\mathbf{letrec}$-like construct:

$$\mathbf{data}\ PTerm\ a = Mu_2\ ([\,a\,] \to [\,PTerm\ a\,])\ |\ \ldots$$

$$peval :: PTerm\ Value \to Value$$

$$\ldots$$

$$peval\ (Mu_2\ f) = head\ \$\ fix\ (map\ peval \circ f)$$

- Note that the guarantee of well-formedness has been (subjectively) weakened. E.g.:

$$Mu_2\ (\lambda xs \to \ldots\ Var\ (xs\ !!\ n)\ \ldots)$$

Recursive PHOAS Binders (2)

Or a letrec-like construct:

$$\mathbf{data}\ PTerm\ a = Mu_2\ ([a] \to [PTerm\ a]) \mid \ldots$$

$$peval :: PTerm\ Value \to Value$$

$$\ldots$$

$$peval\ (Mu_2\ f) = head\ \$\ fix\ (map\ peval \circ f)$$

- Note that the guarantee of well-formedness has been (subjectively) weakened. E.g.:

$$Mu_2\ (\lambda xs \to \ldots\ Var\ (xs\ !!\ n) \ldots)$$

- Length-indexed vectors could help.

Cyclic Streams

$$
\begin{aligned}
&\textbf{data}\ \mathit{PStream}\ a\ v = \\
&\qquad \mathit{Var}\ v \\
&\qquad \mid \mathit{Mu}\ (v \to \mathit{PStream}\ a\ v) \\
&\qquad \mid \mathit{Cons}\ a\ (\mathit{PStream}\ a\ v) \\
&\textbf{newtype}\ \mathit{Stream}\ a = \downarrow \{\uparrow :: \forall v\ .\ \mathit{PStream}\ a\ v\}
\end{aligned}
$$

Finitely representable cyclic streams if inductive interpretation chosen.

Cyclic Streams

$$\begin{array}{l}\textbf{data}\ \mathit{PStream}\ a\ v = \\ \qquad \mathit{Var}\ v \\ \quad\ \mid \mathit{Mu}\ (v \to \mathit{PStream}\ a\ v) \\ \quad\ \mid \mathit{Cons}\ a\ (\mathit{PStream}\ a\ v)\end{array}$$

$$\textbf{newtype}\ \mathit{Stream}\ a = \downarrow \{\uparrow :: \forall v\ .\ \mathit{PStream}\ a\ v\}$$

Finitely representable cyclic streams if inductive interpretation chosen. Example:

$$s_1 = \downarrow\ (\mathit{Cons}\ 1\ (\mathit{Mu}\ (\lambda x \to (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Var}\ x))))))$$

represents the stream $s_1 = 1 : \boxed{2 : 3 : \bullet}$

Fold on Cyclic Streams (1)

$$
\begin{array}{l}
\mathit{sfold} :: (a \to b \to b) \to b \to \mathit{Stream}\ a \to b \\
\mathit{sfold}\ f\ \ b\ \ s = \mathit{sfAux}\ (\uparrow\ \ s) \\
\quad \textbf{where} \\
\qquad \begin{array}{ll}
\mathit{sfAux}\ (\mathit{Var}\ v) & = v \\
\mathit{sfAux}\ (\mathit{Mu}\ g) & = \mathit{sfAux}\ (g\ \ b) \\
\mathit{sfAux}\ (\mathit{Cons}\ x\ \mathit{xs}) & = f\ \ x\ (\mathit{sfAux}\ \mathit{xs})
\end{array} \\
\mathit{selems} :: \mathit{Stream}\ a \to [\,a\,] \\
\mathit{selems} = \mathit{sfold}\ (:)\ [\,]
\end{array}
$$

Fold on Cyclic Streams (1)

$$
\begin{array}{l}
sfold :: (a \to b \to b) \to b \to Stream\ a \to b \\
sfold\ f\ b\ s = sfAux\ (\uparrow\ s) \\
\quad \textbf{where} \\
\qquad \begin{array}{ll}
sfAux\ (Var\ v) & = v \\
sfAux\ (Mu\ g) & = sfAux\ (g\ b) \\
sfAux\ (Cons\ x\ xs) & = f\ x\ (sfAux\ xs)
\end{array} \\
selems :: Stream\ a \to [a] \\
selems = sfold\ (:)\ [\,]
\end{array}
$$

Example:

$$selems\ s_1 \Rightarrow [1, 2, 3]$$

Fold on Cyclic Streams (2)

Another possibility, using g at type $() \to PStream\ a\ ()$:

$$
\begin{aligned}
&sfold :: (a \to b \to b) \to b \to Stream\ a \to b\\
&sfold\ f\ b\ s = sfAux\ (\uparrow\ s)\\
&\quad \textbf{where}\\
&\qquad sfAux\ (Var\ \_) \qquad\ \ = b\\
&\qquad sfAux\ (Mu\ g) \qquad\ \ = sfAux\ (g\ ())\\
&\qquad sfAux\ (Cons\ x\ xs) = f\ x\ (sfAux\ xs)
\end{aligned}
$$

Cyclic Fold on Cyclic Streams

$$scfold :: (a \to b \to b) \to Stream\ a \to b$$
$$scfold\ f\ s = csfAux\ (\uparrow\ s)$$

where

$$\begin{aligned} csfAux\ (Var\ v) &= v \\ csfAux\ (Mu\ g) &= \mathit{fix}\ (csfAux \circ g) \\ csfAux\ (Cons\ x\ xs) &= f\ x\ (csfAux\ xs) \end{aligned}$$

$$toList :: Stream\ a \to [a]$$
$$toList = scfold\ (:)$$

Cyclic Fold on Cyclic Streams

$$
\begin{array}{l}
scfold :: (a \to b \to b) \to Stream\ a \to b \\
scfold\ f\ s = csfAux\ (\uparrow\ s) \\
\quad \mathbf{where} \\
\qquad \begin{array}{ll}
csfAux\ (Var\ v) & = v \\
csfAux\ (Mu\ g) & = \mathit{fix}\ (csfAux \circ g) \\
csfAux\ (Cons\ x\ xs) & = f\ x\ (csfAux\ xs)
\end{array} \\
toList :: Stream\ a \to [a] \\
toList = scfold\ (:)
\end{array}
$$

Example: (Because $\mathit{fix}\ (\lambda x \to 2 : 3 : x) = [2, 3, 2, 3, \ldots)$

$$scfold\ s_1 \Rightarrow [1, 2, 3, 2, 3, 2, 3, \ldots$$

Sharing-preserving Transformation

$$
\begin{array}{l}
smap :: (a \to b) \to Stream\ a \to Stream\ b \\
smap\ f\ s = \downarrow\ (smAux\ (\uparrow\ s)) \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
smAux\ (Var\ x) & = & Var\ x \\
smAux\ (Mu\ g) & = & Mu\ (smAux \circ g) \\
smAux\ (Cons\ x\ xs) & = & Cons\ (f\ x)\ (smAux\ xs)
\end{array}
\end{array}
$$

Sharing-preserving Transformation

$$
\begin{array}{l}
smap :: (a \rightarrow b) \rightarrow Stream\ a \rightarrow Stream\ b \\
smap\ f\ s = \downarrow\ (smAux\ (\uparrow\ s)) \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
smAux\ (Var\ x) & = & Var\ x \\
smAux\ (Mu\ g) & = & Mu\ (smAux \circ g) \\
smAux\ (Cons\ x\ xs) & = & Cons\ (f\ x)\ (smAux\ xs)
\end{array}
\end{array}
$$

Note that standard map on a list that happens to be represented by a cyclic heap structure in a lazy functional language (like $ones = 1 : ones$) will lose the cyclic structure (unless memoization is used).

Tail of a Cyclic Stream (1)

$$
\begin{array}{l}
stail :: Stream\ a \rightarrow Stream\ a \\
stail\ s = \downarrow\ (pjoin\ (ptail\ (\uparrow\ s))) \\
\quad \textbf{where} \\
\qquad ptail\ (Cons\ x\ xs) = xs \\
\qquad ptail\ (Mu\ g) \qquad\quad = Mu\ (\lambda x \rightarrow \\
\qquad\quad \textbf{let}\ phead\ (Mu\ g) \qquad\quad = phead\ (g\ x) \\
\qquad\qquad\ \ phead\ (Cons\ y\ ys) = y \\
\qquad\quad \textbf{in} \\
\qquad\qquad ptail\ (g\ (Cons\ (phead\ (g\ x))\ x)))
\end{array}
$$

Here, g is used at type
$PStream\ a\ v \rightarrow PStream\ a\ (PStream\ a\ v)$

Tail of a Cyclic Stream (2)

$pjoin$ is a monadic-like $join$ operation.

$$
\begin{aligned}
&pjoin :: PStream\ a\ (PStream\ a\ v) \rightarrow PStream\ a\ v\\
&pjoin\ (Var\ x) \qquad\quad = x\\
&pjoin\ (Mu\ f) \qquad\quad = Mu\ (pjoin \circ f \circ Var)\\
&pjoin\ (Cons\ x\ xs) = Cons\ x\ (pjoin\ xs)
\end{aligned}
$$

Structural Equality

The nub of the algorithm:

$$
\begin{array}{llll}
peq :: Eq\ a \Rightarrow Int \rightarrow PStream\ a\ Int \rightarrow PStream\ a\ Int \rightarrow Bool \\
peq\ n\ (Var\ n_1) & (Var\ n_2) & = n_1 \equiv n_2 \\
peq\ n\ (Mu\ f) & (Mu\ g) & = peq\ (n+1)\ (f\ n)\ (g\ n) \\
peq\ n\ (Cons\ x\ xs) & (Cons\ y\ ys) & = x \equiv y \wedge peq\ n\ xs\ ys \\
peq\ \_\ \_ & \_ & = False
\end{array}
$$

Generic Structured Graphs

- Parametrize on a functor describing the node structure.
- Employ multi-binder to allow cross-edges in addition to back-edges.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Rec}\ f\ \ v = \\
&\qquad \mathit{Var}\ \ v \\
&\quad\ \mid \mathit{Mu}\ ([v] \to [f\ \ (\mathit{Rec}\ f\ \ a)]) \\
&\quad\ \mid \mathit{In}\ (f\ \ (\mathit{Rec}\ f\ \ a)) \\
&\mathbf{newtype}\ \mathit{Graph}\ f = \downarrow\ \{\uparrow :: \forall v\ .\ \mathit{Rec}\ f\ \ v\}
\end{aligned}
$$

Cyclic Trees in Terms of Graphs

$$\textbf{data}\ \mathit{TreeF}\ a\ r = \mathit{Empty} \mid \mathit{Fork}\ a\ r\ r$$
$$\quad \textbf{deriving}\ (\mathit{Functor}, \mathit{Foldable}, \mathit{Traversable})$$

$$\textbf{type}\ \mathit{Tree}\ a = \mathit{Graph}\ (\mathit{TreeF}\ a)$$

Example:

$$\begin{array}{l}
\mathit{tree} = \downarrow\ (\mathit{Mu}\ (\lambda(\sim(t_1 : t_2 : t_3 : _)) \to [\\
\quad \mathit{Fork}\ 1\ (\mathit{In}\ (\mathit{Fork}\ 4\ (\mathit{Var}\ t_2)\ (\mathit{In}\ \mathit{Empty})))\\
\qquad\qquad (\mathit{Var}\ t_3),\\
\quad \mathit{Fork}\ 2\ (\mathit{Var}\ t_1)\ (\mathit{Var}\ t_3),\\
\quad \mathit{Fork}\ 3\ (\mathit{Var}\ t_2)\ (\mathit{Var}\ t_1)]))
\end{array}$$

Some Generic Graph Folds

$$
\begin{aligned}
&\mathit{fold} :: \mathit{Functor}\ f\ (f\ \ a \to a) \to a \to \mathit{Graph}\ f \to a\\
&\mathit{cfold} :: \mathit{Functor}\ f\ (f\ \ a \to a) \to \mathit{Graph}\ f \to a\\
&\mathit{sfold} :: (\mathit{Eq}\ \ a, \mathit{Functor}\ f) \Rightarrow\\
&\qquad\qquad (f\ \ a \to a) \to a \to \mathit{Graph}\ f \to a
\end{aligned}
$$

sfold uses a fixed-point operator that iterates the function until convergence (assuming monotonicity).

An Application: Liveness (1)

A variable v is ***live*** at point p if there exists an execution path from p to a use of v along which v is not updated.

```
1   i := m;
2   n := 1;
3   while (i < 10) do begin
4       n := n * p;
5       i := i + 1
6   end
7   return n;
```

Which of `i`, `m`, `n`, `p` are live immediately before lines 1, 3, 7?

An Application: Liveness (2)

Define a suitable dataflow graph:

$$\mathbf{data}\ \mathit{Expr} = \mathit{Lit}\ \mathit{Int} \mid \mathit{Use}\ \mathit{Id} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \ldots$$

$$\mathit{uses} :: \mathit{Expr} \to [\mathit{Id}]$$

$$\mathit{uses} = \ldots$$

$$\begin{aligned}
&\mathbf{data}\ \mathit{CodeF}\ a = \\
&\qquad \mathit{Return}\ \mathit{Id} \\
&\quad \mid \mathit{Assign}\ \mathit{Id}\ \mathit{Expr}\ a \\
&\quad \mid \mathit{IfZ}\ \mathit{Id}\ a\ a \\
&\quad \mathbf{deriving}\ (\mathit{Functor}, \mathit{Foldable}, \mathit{Traversable})
\end{aligned}$$

An Application: Liveness (3)

Define the analysis algebra:

$$
\begin{aligned}
&\mathit{liveF} :: \mathit{CodeF}\ [\mathit{Id}] \to [\mathit{Id}] \\
&\mathit{liveF}\ (\mathit{Return}\ v) \qquad\quad = [v] \\
&\mathit{liveF}\ (\mathit{Assign}\ v\ e\ l) = \mathit{uses}\ e \cup (l \setminus [v]) \\
&\mathit{liveF}\ (\mathit{IfZ}\ v\ l_1\ l_2) \quad\ = [v] \cup l_1 \cup l_2
\end{aligned}
$$

Finally, define the liveness analysis as an sfold:

$$
\begin{aligned}
&\mathit{live} :: \mathit{Graph}\ \mathit{CodeF} \to [\mathit{Id}] \\
&\mathit{live} = \mathit{sfold}\ \mathit{liveF}\ [\,]
\end{aligned}
$$

(Returns what's live at whatever block is first.)

Conclusions

- Oliveira’s and Cook’s method works well for applications where the graph structure is preserved or where computation is by folding over a graph.
- Structure-changing operations is possible, but much more involved (see paper).

References (1)

- Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. ICFP'08, 2008.
- Martin Erwig. Inductive graphs and functional graph algorithms. Journal of Functional Programming 11(5), 2001.
- Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). POPL'96, 1996.

References (2)

- Neil Ghani, Makoto Hamana, Tarmo Uustalu, & Varmo Vene. Representing cyclic structures as nested datatypes. TFP'06, 2006.
- Andy Gill. Type-safe observable sharing in Haskell. Haskell'09, 2009.
- John Hughes. Lazy memo-functions. FPCA'85, 1985.
- Thomas Johnsson. Efficient graph algorithms using lazy monolithic arrays. Journal of Functional Programming 8(4), 1998

References (3)

- David J. King & John Launchbury. Lazy depth-first search and linear graph algorithms in Haskell. Glasgow Workshop on Functional Programming, 1994.
- Bruno Oliveira & William Cook. Functional Programming with Structured Graphs. Draft, March 2012.