Fine Control of Demand in Haskell

Bill Harrison, Tim Sheard, & James Hook

Introducing "Phugs"

Programatica Hugs:

1. First Haskell implementation to meet the rigorous, internationally-recognized WSHI* standard

2. Uses Programatica front-end `pfe`

3. Ex:

```
module Phugs where
   fac n  = if n==0 then 1 else n*(fac (n-1))
   odd n  = if n==0 then False else (even (n-1))
   even n = if n==0 then True else (odd (n-1))
   topLevel = odd (fac 3)
```

---

* "World's Slowest Haskell Implementation"

Fine Control of Demand in Haskell

1. Functional languages typically have mixed evaluation—i.e., neither completely lazy nor completely eager
   (a) `if-then-else` in ML/Scheme, etc.
   (b) Features such as pattern-matching, guards, etc., in Haskell
2. "Textbook" language semantics tend to model pure, (i.e., not mixed) evaluation strategies
3. Question: But just how do the semantics of real languages with messy, mixed evaluation relate to these textbook examples?

Goals

Previous denotational approaches to Haskell compile away "hard stuff" (nested patterns,...) into simpler language

1. Resulting simplified language is susceptible to standard denotational description
2. However, such semantics
   (a) are non-compositional
   (b) involve semantically-tricky fresh variable generation ; these require specialized semantic setting beyond usual CPO semantics

Want to use standard techniques from denotational semantics to produce a semantics for all of Haskell98

1. Features described here cover "fine control of demand"
2. Not addressing overloading or the `IO` monad today

“Fine Control of Demand?”

```
data Tree = T Tree Tree | S Tree | R Tree | L

1. (\ (T (S x) (R y)) -> L) (T L (R L)) ---> ⊥

2. (\ ~(T (S x) (R y)) -> L) (T L (R L)) ---> L

3. (\ ~(T (S x) (R y)) -> x) (T L (R L)) ---> ⊥

4. (\ ~(T ~(S x) (R y)) -> y) (T L (R L)) ---> L

5. (\ ~(T (S x) ~(R y)) -> y) (T L (R L)) ---> ⊥
```

Methodology

Initially, we constructed a number of elegant, compelling categorical semantics on paper:

$$
\begin{array}{ccc}
\dfrac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (x \texttt{ -> } e) : \sigma \to \mathsf{M}\tau} & \begin{array}{c} = \\ = \end{array} & \begin{array}{l} h \\ \mathsf{curry}(\eta_A \circ h) \text{ where } A = \widehat{\tau} \end{array} \\
\dfrac{\Gamma \vdash (p \texttt{ -> } e) : \sigma \to \mathsf{M}\tau}{\Gamma \vdash (\mathsf{S}\, p \texttt{ -> } e) : \mathsf{S}\sigma \to \mathsf{M}\tau} & \begin{array}{c} = \\ = \end{array} & \begin{array}{l} \mathsf{curry}(k) \\ \mathsf{curry}(k \diamond (\mathsf{id}_\Gamma \times \mathsf{S}^{-1})) \end{array} \\
\dfrac{\Gamma \vdash (p \texttt{ -> } e) : \sigma \to \mathsf{M}\tau}{\Gamma \vdash (\texttt{\~{}}\mathsf{S}\, p \texttt{ -> } e) : \mathsf{S}\sigma \to \mathsf{M}\tau} & \begin{array}{c} = \\ = \end{array} & \begin{array}{l} \mathsf{curry}(k) \\ \mathsf{curry}(k \circ \nu \circ \mathsf{S}^{-1}) \end{array} \\
 & \vdots &
\end{array}
$$

The idea: extend standard CCC translations of λ-calculus to Haskell.

Methodology (cont'd)

... unfortunately these "pencil and paper" semantics were also:

$$
\begin{array}{lcl}
\Gamma, x : \sigma \vdash e : \tau & = & h \\
\hline
\Gamma \vdash (x \;\texttt{->}\; e) : \sigma \to \mathsf{M}\tau & = & \mathsf{curry}(\eta_A \circ h) \text{ where } A = \widehat{\tau} \\[1ex]
\Gamma \vdash (p \;\texttt{->}\; e) : \sigma \to \mathsf{M}\tau & = & \mathsf{curry}(k) \\
\hline
\Gamma \vdash (\mathsf{S}\, p \;\texttt{->}\; e) : \mathsf{S}\sigma \to \mathsf{M}\tau & = & \mathsf{curry}(k \diamond (\mathsf{id}_\Gamma \times \mathsf{S}^{-1})) \\[1ex]
\Gamma \vdash (p \;\texttt{->}\; e) : \sigma \to \mathsf{M}\tau & = & \mathsf{curry}(k) \\
\hline
\Gamma \vdash (\texttt{\~{}}\mathsf{S}\, p \;\texttt{->}\; e) : \mathsf{S}\sigma \to \mathsf{M}\tau & = & \mathsf{curry}(k \circ \nu \circ \mathsf{S}^{-1}) \\
 & \vdots &
\end{array}
$$

WRONG!

Although compelling, etc., these attempts failed. Why?

The interaction of the

Upshot: automated approach was, if not strictly necessary, then certainly extremely useful

Overview

1. Semantic setting
2. Patterns
3. Expressions
4. Bodies (i.e., guarded expressions)
5. Declarations
6. Mutual recursion, let binding, and where clauses
7. Summation

Calculational Semantics for Haskell

Scalar, function, and structured data values ; Environments bind names to values

```
data V = Z Integer  | FV (V -> V) | Tagged Name [V]
type Env = Name -> V
```

Meanings for expressions, patterns, bodies, and declarations

```
mE  :: E -> Env -> V           mB  :: B -> Env -> Maybe V
mP  :: P -> V -> Maybe [V]     mD  :: D -> Env -> V
```

Semantic Setting

Rather than giving an explicitly categorical or domain-theoretic treatment, assume existence of certain basic semantic operators:

– Function composition (diagrammatic)

$(\ggg) :: (a \to b) \to (b \to c) \to a \to c$

$f \ggg g = g \circ f$

– Function application

$\mathrm{app} :: V \to V \to V$

$\mathrm{app}\ (FV\ f)\ x = f\ x$

– Currying

$\mathrm{sharp} :: \mathrm{Int} \to [V] \to (V \to V) \to V$

$\mathrm{sharp}\ 0\ \mathrm{vs}\ \mathrm{beta} = \mathrm{beta}\ (\mathrm{tuple}\ \mathrm{vs})$

$\mathrm{sharp}\ n\ \mathrm{vs}\ \mathrm{beta} =$

$\quad FV\ (\lambda\ v\ .\ \mathrm{sharp}\ (n\text{-}1)\ (\mathrm{vs}{+}{+}[v])\ \mathrm{beta})$

– Domains are pointed

$\mathrm{bottom} :: a$

$\mathrm{bottom} = \mathrm{undefined}$

– Semantic "seq"

$\mathrm{semseq} :: V \to V \to V$

$$\begin{array}{lll} \mathrm{semseq}\ x\ y = \mathrm{case}\ x\ \mathrm{of} & (Z\ \_) & \to y\ ; \\ & (FV\ \_) & \to y\ ; \\ & (\mathrm{Tagged}\ \_\ \_) & \to y \end{array}$$

– Least fixed points exist

$\mathrm{fix} :: (a \to a) \to a$

$\mathrm{fix}\ f = f\ (\mathrm{fix}\ f)$

Semantic Setting (cont'd)

Semantic operators for pattern-matching:

– Purification: the "run" of Maybe monad

purify :: Maybe a $\rightarrow$ a
purify (Just x) $=$ x
purify Nothing $=$ bottom

– Kleisli composition (diagrammatic)

$(\diamond)$ :: (a $\rightarrow$ Maybe b) $\rightarrow$ (b $\rightarrow$ Maybe c) $\rightarrow$ a $\rightarrow$ Maybe c
f $\diamond$ g $= \lambda$ x . (f x) $\star$ g

– Alternation: "app (fatbar m1 m2) v" similar to "case v of { m1 ; m2 }"

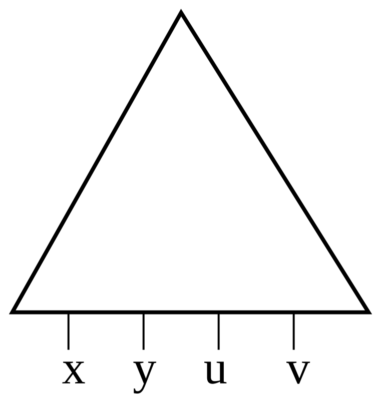
($\|$) :: (a $\rightarrow$ Maybe b) $\rightarrow$ (a $\rightarrow$ Maybe b) $\rightarrow$ (a $\rightarrow$ Maybe b)
f $\|$ g $= \lambda$ x . (f x) ‘fb‘ (g x)
where fb :: Maybe a $\rightarrow$ Maybe a $\rightarrow$ Maybe a
Nothing ‘fb‘ y $=$ y
(Just v) ‘fb‘ y $=$ (Just v)

Nested Pattern Language P

```
data P
  = Pconst Integer           --- 5
  | Pvar Name                --- x
  | Ptuple [P]               --- (p1,p2)
  | Pcondata Name [P]        --- data T1 = C1 t1 t2; {C1 p1 p1} = e
  | Pnewdata Name P          --- newtype T2 = C2 t1;  {C2 p1} = e
  | Pwildcard                --- _
  | Ptilde P                 --- ~ p
```

N.b., this abstract syntax is representative of Haskell's patterns, but not exhaustive.

The "fringe of a pattern" are its variables in order of occurrence.
E.g., [x,y,u,v]



($\sim$) shifts matching from binding to evaluation

Consider the following Haskell expressions for datatype

```
data Tree = T Tree Tree | R Tree | L
```

Here, match failure occurs at binding-time of `x` & `y`:

```
case (T L L) of { (T (R x) y)   -> y }    ---fails
```

Match failure occurs (if at all) at evaluation-time of `x` & `y`:

```
case (T L L) of { (T ~(R x) y) -> y }     ---produces L
case (T L L) of { (T ~(R x) y) -> x }     ---fails
```

Binding-time match failure is modelled by `Nothing`, and evaluation-time match failure by binding `x` to `bottom`

Semantics of `P`: `(mP :: V -> Maybe [V])`

```
mP  :: P -> V -> Maybe [V]
mP (Pvar x) v                          = Just [v]
mP (Pconst i) (Z j)                    = if i==j then Just [] else Nothing
mP Pwildcard v                         = Just []
mP (Pnewdata n p) v                    = mP p v
mP (Pcondata n ps) (Tagged t vs)       = if n==t then
                                              stuple (map mP ps) vs
                                           else Nothing

stuple :: [V -> Maybe [V]] -> [V] -> Maybe [V]
stuple [] []          = Just []
stuple (q:qs) (v:vs) = do { v' <- q v ; vs' <- stuple qs vs ; Just (v'++vs') }
```

Semantics of `P` (cont'd)

```
mP  :: P -> V -> Maybe [V]
mP (Ptilde p) v
   = Just(case mP p v of { Nothing -> replicate lp bottom
                         ; Just z -> z })
             where lp = length (fringe p)
```

Why does this work? If `(mP p v)` is `Nothing` (i.e., a binding-time match failure), then it is converted into a (potential) evaluation-time match failure. That is,

`(mP p v)==Nothing` $\Leftrightarrow$ `(mP (~p) v)==Just[bottom,...,bottom]`

Semantics of E: Simple Expressions

```
mE :: E -> Env -> V
mE (Var n) rho          = rho n
mE (Const i) rho        = (Z i)
mE (TupleExp es) rho    =
   tuple $ map (\e-> mE e rho) es
mE (Cond e0 e1 e2) rho =
   ifV (mE e0 rho) (mE e1 rho) (mE e2 rho)
mE Undefined rho        = bottom
```

```
ifV :: V -> a -> a -> a
ifV (Tagged "True" []) x y = x
ifV (Tagged "False" []) x y = y

tuple :: [V] -> V
tuple [v] = v
tuple vs = Tagged "tuple" vs
```

N.b., tuples are treated as Tagged values.

Semantics of E: Application and Abstraction

```
mE :: E -> Env -> V
mE (App e1 e2) rho = app (mE e1 rho) (mE e2 rho)
mE (Abs [p] e) rho = FV $ lam p e rho
mE (Abs ps e) rho  = sharp (length ps) [] (lam (ptuple ps) e rho)
        where ptuple :: [P] -> P
              ptuple [p] = p
              ptuple ps = Pcondata "tuple" ps

lam :: P -> E -> Env -> V -> V
lam p e rho = (mP p <> ((\vs -> mE e (extend rho xs vs)) >>> Just)) >>> purify
     where xs = fringe p
```

Subtlety: $(\backslash p_1\, p_2\text{->}\, e)$ is lazier than $(\backslash p_1\text{->}(\backslash p_2\text{->}\, e))$

The Haskell98 report [section 3.3] states:

$$\backslash p_1 \ldots p_n\text{->}\, e = \backslash x_1 \ldots x_n\text{->}\, \mathtt{case}\ (x_1, \ldots, x_n)\ \mathtt{of}\ (p_1, \ldots, p_n)\ \text{->}\ e$$

Guarded Expresssions (aka "bodies" B)

Guarded expressions occur within cases:

```
case e of { p | g1->e1 ... gn->en where { decls } ; <rest> }
```

where g1,...,gn are boolean expressions.

The semantics for B is then:

```
mB :: B -> Env -> Maybe V

mB (Normal e) rho = Just (mE e rho)

mB (Guarded [(g1,e1),...,(gn,en)]) rho =
          ifV (mE g1 rho) (Just (mE e1 rho))
                            ...
               (ifV (mE gn rho) (Just (mE en rho)) Nothing)
```

Case Expressions

The AST for case expressions has the form:

```
Case e [(p1,b1,ds1),...,(pn,bn,dsn)]
```

where `pi`, `bi`, and `dsi` are patterns, bodies, and where-clause bindings, respectively.

The semantics of a "match" `(p,b,ds)` is defined by a function:

```
match :: Env -> (P, B, [D]) -> V -> Maybe V
match rho (p,b,ds) = mP p <> ( vs -> mwhere (extend rho xs vs) b ds)
        where xs = fringe p
```

Here, `(mwhere rho b ds)` is the meaning of "`b where ds`", and `(mwhere rho b [])` is simply `(mB b rho)`.

Case Expressions (cont'd)

```
mE :: E -> Env -> V
mE (Case e ml) rho = mcase rho ml (mE e rho)

mcase :: Env -> [(P,B,[D])] -> V -> V
mcase rho ml = (fatbarL $ map (match rho) ml) >>> purify
        where fatbarL :: [V -> Maybe V] -> V -> Maybe V
              fatbarL ms = foldr fatbar (\ _ -> Nothing) ms
```

Note that unfolding (mcase rho [m1,...,mn]) has the form:

```
( (match rho m1) `fatbar`
                    ...
  (match rho mn) `fatbar` (\ _ -> Nothing)) )  >>> purify
```

If the `Nothing` branch is reached, then the `purify` will convert the resulting match failure into `bottom`. This occurs when the branches of a case have been exhausted.

seq, strict and newtype constructors

```
mE :: E -> Env -> V

 -- Miscellaneous Functions
mE (Seq e1 e2) rho    = semseq (mE e1 rho) (mE e2 rho)

 -- Strict and Lazy Constructor Applications
mE (ConApp n el) rho  = evalL el rho n []
  where
   evalL :: [(E,LS)] -> Env -> Name -> [V] -> V
   evalL [] rho n vs             = Tagged n vs
   evalL ((e,Strict):es) rho n vs =
               semseq (mE e rho) (evalL es rho n (vs ++ [mE e rho]))
   evalL ((e,Lazy):es) rho n vs = evalL es rho n (vs ++ [mE e rho])

-- New type constructor applications
mE (NewApp n e) rho   = mE e rho
```

Multi-line function & pattern declarations

AST for function declarations: `(Fun Name [([P],B,[D])])`

```
nth :: Int -> [a] -> a
nth 0 (x:xs) = x
     where foobar = 89
nth i (x:xs) = nth (i-1) xs
```

Haskell98 Report[Section 4.4.3] defines these by translation into a single `case` expression:

The general binding form for functions is semantically equivalent to the equation (i.e. simple pattern binding):

$$x = \backslash \mathtt{x}_1 \ldots \mathtt{x}_\mathtt{k} \texttt{->}\ \texttt{case}\ (\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{k})\ \ \texttt{of}\ \ \begin{array}{l} (p_{11}, \ldots, p_{1k})\ \ match_1 \\ \qquad\qquad \vdots \\ (p_{m1}, \ldots, p_{mk})\ \ match_m \end{array}$$

where the "x_i" are new identifiers.

Use `sharp` and `mcase` as in case expressions

Multi-line function & pattern declarations (cont'd)

Compare the translation:

$$x = \backslash\mathtt{x}_1 \ldots \mathtt{x}_k \texttt{->}\ \mathtt{case}\ (\mathtt{x}_1, \ldots, \mathtt{x}_k)\ \ \mathtt{of}\ \ \begin{array}{c} (p_{11}, \ldots, p_{1k})\ \ match_1 \\ \vdots \\ (p_{m1}, \ldots, p_{mk})\ \ match_m \end{array}$$

with the semantics:

```
mD :: D -> Env -> V
mD (Fun f cs) rho  = sharp k [] body
   where
      body = mcase rho (map (\(ps,b,ds) -> (ptuple ps, b,ds)) cs)
      k  = length ((\(pl,_,_)->pl) (head cs))

mD (Val p b ds) rho = purify (mwhere rho b ds)
```

Using `sharp` eliminates the need for name generation here

Approach to mutually-recursive let-binding

Mutual recursion and recursive `let` achieved by combining standard techniques in the following scheme:

$$\texttt{let}\ \{\ p_1 = e_1\ ;\ \ldots\ ;\ p_n = e_n\ \}\ \texttt{in}\ e =$$

$$(\texttt{\textbackslash}\ \texttt{\~{}}(\texttt{\~{}p}_1, \ldots, \texttt{\~{}p}_n)\ \texttt{->}\ \texttt{e})\ (\texttt{fix}\ (\texttt{\textbackslash}\ \texttt{\~{}}(\texttt{\~{}p}_1, \ldots, \texttt{\~{}p}_n)\ \texttt{->}\ (\texttt{e}_1, \ldots, \texttt{e}_n)))$$

1. Recursion resolved with explicit `fix`,
2. Pattern-matching makes abstractions less-than-lazy—mysterious appearances of $(\sim)$ to recover laziness,
3. Both `letbind` and `mwhere` are defined using the scheme above.

Comparing the semantics to Hugs

```
e1 = seq ((\ (Just x) y -> x) Nothing) 3
e2 = seq ((\ (Just x) -> (\ y -> x)) Nothing) 3
e3 = (\ ~(x, Just y) -> x) (0, Nothing)
```

```
e4 = case 1 of
      x | x==z -> (case 1 of w | False -> 33)
           where z = 1
      y -> 101
```

```
e5 = case 1 of
          x | x==z -> (case 1 of w | True -> 33)
                where z = 2
          y -> 101
```

```
e6 = let  fac 0 = 1
          fac n = n * (fac (n-1))
     in fac 3
```

```
Semantics> mE e1 rho0              Hugs> e1
    3                                  3
Semantics> mE e2 rho0              Hugs> e2
    Program error: {undefined}         Program error: {e2_v2550 Maybe_Nothing}
Semantics> mE e3 rho0              Hugs> e3
    Program error: {undefined}         Program error: {e3_v2558 (Num_fromInt instNum_v35 0,Maybe_Nothi
Semantics> mE e4 rho0              Hugs> e4
    Program error: {undefined}         Program error: {e4_v2562 (Num_fromInt instNum_v35 1)}
Semantics> mE e5 rho0              Hugs> e5
    101                                101
Semantics> mE e6 rho0              Hugs> e6
    6                                  6
```

Conclusions

1. Semantics is compositional,
2. And have not used fresh name generation,
3. To do:
   (a) Finish proving that semantics validates the Haskell98 Report translations
   (b) Add overloading