AWESOME PRELUDE

"Liberating Haskell from datatypes!"

Tom Lokhorst, Sebastiaan Visser

February 8, 2010

$$4 + 3 \times 2$$

```
data Expr where
  Con :: Int    → Expr
  Add :: Expr → Expr → Expr
  Mul :: Expr → Expr → Expr
```

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Int} \\
&\mathit{eval}\ (\mathit{Con}\ x) \quad = x \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y \\
&\mathit{eval}\ (\mathit{Mul}\ x\ y) = \mathit{eval}\ x * \mathit{eval}\ y
\end{aligned}
$$

```
import Language.Cil

compile :: Expr -> Assembly
compile e = simpleAssembly (f e)
  where
    f :: Expr -> [MethodDecl]
    f (Con x)   = [ldc_i4 x]
    f (Add x y) = f x ++ f y ++ [add]
    f (Mul x y) = f x ++ f y ++ [mul]
```

$$4 + 2 \times 3$$

$$4 + 2 \times 3$$

```
x :: Expr
x = Add (Con 4)
        (Mul (Con 2)
             (Con 3))
```

```
instance Num Expr where
  fromInteger x = Con (fromIntegral x)
  x + y         = Add x y
  x * y         = Mul x y
```

$$4 + 2 \times 3$$

$$4 + 2 \times 3$$

```
x :: Expr
x = 4 + 2 * 3
```

$$4 + 2 \times 3$$

```
x :: Int
x = 4 + 2 * 3
```

$$4 + 2 \times 3$$

$x :: Num\ a \Rightarrow a$

$x = 4 + 2 * 3$

$$\text{if } 2 + 3 \equiv 5 \text{ then } 1 \text{ else } 0$$

$$\text{if } 2 + 3 \equiv 5 \text{ then } 1 \text{ else } 0$$

```
data Expr where
  Con       :: Int  → Expr
  Add       :: Expr → Expr → Expr
  Mul       :: Expr → Expr → Expr
  ConFalse  :: Expr
  ConTrue   :: Expr
  Eq        :: Expr → Expr → Expr
  If        :: Expr → Expr → Expr → Expr
```

```
eval :: Expr → Either Bool Int
eval (Con x)     = Right x
eval (Add x y)   = let (Right x') = eval x
                       (Right y') = eval y
                   in  Right (x' + y')
eval (Mul x y)   = let (Right x') = eval x
                       (Right y') = eval y
                   in  Right (x' * y')
eval (ConFalse) = Left False
eval (ConTrue)  = Left True
eval (Eq x y)    = Left (eval x == eval y)
eval (If p x y)  = let (Left p') = eval p
                   in  if p'
                       then eval x
                       else  eval y
```

$$\text{if } 2 + 3 \equiv 5 \text{ then } 1 \text{ else } 0$$

$$\text{if } 2 + 3 \equiv 5 \text{ then } 1 \text{ else } 0$$

```
x :: Expr
x = If (Eq (Add (Con 2) (Con 3))
           (Con 5))
       (Con 1)
       (Con 0)
```

$$\text{if } 2 + 3 \equiv 5 \text{ then } 1 \text{ else } 0$$

```
x :: Expr
x = If (Eq (2 + 3) 5) (1) (0)
```

$$\text{if } 2 + 3 \equiv 5 \text{ then } 1 \text{ else } 0$$

```
x :: Expr
x = If (2 + 3 == 5) (1) (0)
```

$$(\texttt{==}) :: Eq\ a \Rightarrow a \to a \to Bool$$

$$(\texttt{==}) :: Eq\ a \Rightarrow a \to a \to Bool$$

$$(\texttt{==}) :: (Eq\ a, BoolLike\ b) \Rightarrow a \to a \to b$$

```
class BoolLike b where
  false :: b
  true  :: b
  bool  :: a → a → b → a
```

```
class BoolLike b where
  false :: b
  true  :: b
  bool  :: a → a → b → a

instance BoolLike Bool where
  false     = False
  true      = True
  bool x y b = if b then y else x
```

```
class BoolLike b where
  false :: b
  true  :: b
  bool  :: a → a → b → a
```

```
instance BoolLike Expr where
  false     = ConFalse
  true      = ConTrue
  bool x y b = If b y x
```

$$
\begin{aligned}
(\&\&) &:: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
(||) &:: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
\mathit{not} &:: \mathit{Bool} \to \mathit{Bool}
\end{aligned}
$$

$$
\begin{aligned}
(\&\&) &:: Bool \to Bool \to Bool \\
(||) &:: Bool \to Bool \to Bool \\
not &:: Bool \to Bool
\end{aligned}
$$

$$(\&\&) :: BoolLike\ b \Rightarrow b \to b \to b$$

$$(||) :: BoolLike\ b \Rightarrow b \to b \to b$$

$$not :: BoolLike\ b \Rightarrow b \to b$$

$$
\begin{aligned}
(\&\&) &:: Bool \to Bool \to Bool \\
(||) &:: Bool \to Bool \to Bool \\
not &:: Bool \to Bool
\end{aligned}
$$

$$
\begin{aligned}
(\&\&) &:: BoolLike\ b \Rightarrow b \to b \to b \\
(\&\&)\ x\ y &= bool\ x\ y\ x
\end{aligned}
$$

$$
\begin{aligned}
(||) &:: BoolLike\ b \Rightarrow b \to b \to b \\
(||)\ x\ y &= bool\ y\ x\ x
\end{aligned}
$$

$$
\begin{aligned}
not &:: BoolLike\ b \Rightarrow b \to b \\
not\ x &= bool\ true\ false\ x
\end{aligned}
$$

```
ghci> :t not
not :: (BoolLike b) => b -> b

ghci> not True
False

ghci> not ConTrue
If ConTrue ConFalse ConTrue
```

```
data Expr where
  Con      :: Int  → Expr
  Add      :: Expr → Expr → Expr
  Mul      :: Expr → Expr → Expr
  ConFalse :: Expr
  ConTrue  :: Expr
  Eq       :: Expr → Expr → Expr
  If       :: Expr → Expr → Expr → Expr
```

```
data Expr a where
  Con      :: Int          → Expr Int
  Add      :: Expr Int     → Expr Int → Expr Int
  Mul      :: Expr Int     → Expr Int → Expr Int
  ConFalse :: Expr Bool
  ConTrue  :: Expr Bool
  Eq       :: Expr Int     → Expr Int → Expr Bool
  If       :: Expr Bool → Expr a    → Expr a → Expr a
```

```
class BoolLike b where
  false :: b
  true  :: b
  bool  :: a -> a -> b -> a

instance BoolLike Expr where
  false     = ConFalse
  true      = ConTrue
  bool x y b = If b y x
```

```
data Bool

class BoolC j where
  false :: j Bool
  true  :: j Bool
  bool  :: j r → j r → j Bool → j r

instance BoolC Expr where
  false      = ConFalse
  true       = ConTrue
  bool x y b = If b y x
```

```
data Bool

class BoolC j where
  false :: j Bool
  true  :: j Bool
  bool  :: j r → j r → j Bool → j r
```

```
data Bool

class BoolC j where
  false :: j Bool
  true  :: j Bool
  bool  :: j r → j r → j Bool → j r


data Maybe a

class MaybeC j where
```

```
data Bool

class BoolC j where
  false :: j Bool
  true  :: j Bool
  bool  :: j r → j r → j Bool → j r


data Maybe a

class MaybeC j where
  nothing :: j (Maybe a)
```

```
data Bool

class BoolC j where
  false :: j Bool
  true  :: j Bool
  bool  :: j r → j r → j Bool → j r


data Maybe a

class MaybeC j where
  nothing :: j (Maybe a)
  just    :: j a → j (Maybe a)
```

```
data Bool

class BoolC j where
  false :: j Bool
  true  :: j Bool
  bool  :: j r → j r → j Bool → j r


data Maybe a

class MaybeC j where
  nothing :: j (Maybe a)
  just    :: j a → j (Maybe a)
  maybe   :: j r → (j a → j r) → j (Maybe a) → j r
```

```
data Bool

class BoolC j where
  false :: j Bool
  true  :: j Bool
  bool  :: j r → j r → j Bool → j r


data Maybe a

class MaybeC j where
  nothing :: j (Maybe a)
  just    :: j a → j (Maybe a)
  maybe   :: j r → (j a → j r) → j (Maybe a) → j r


class ListC j where
```

```
data Bool

class BoolC j where
  false :: j Bool
  true  :: j Bool
  bool  :: j r → j r → j Bool → j r


data Maybe a

class MaybeC j where
  nothing :: j (Maybe a)
  just    :: j a → j (Maybe a)
  maybe   :: j r → (j a → j r) → j (Maybe a) → j r


class ListC j where
  nil :: j [a]
```

```
data Bool

class BoolC j where
  false :: j Bool
  true  :: j Bool
  bool  :: j r → j r → j Bool → j r


data Maybe a

class MaybeC j where
  nothing :: j (Maybe a)
  just    :: j a → j (Maybe a)
  maybe   :: j r → (j a → j r) → j (Maybe a) → j r


class ListC j where
  nil  :: j [a]
  cons :: j a → j [a] → j [a]
```

```
data Bool

class BoolC j where
  false :: j Bool
  true  :: j Bool
  bool  :: j r → j r → j Bool → j r


data Maybe a

class MaybeC j where
  nothing :: j (Maybe a)
  just    :: j a → j (Maybe a)
  maybe   :: j r → (j a → j r) → j (Maybe a) → j r


class ListC j where
  nil  :: j [a]
  cons :: j a → j [a] → j [a]
  list :: j r → (j a → j [a] → j r) → j [a] → j r
```

class *FunC j* **where**

```
class FunC j where
  lam :: (j a → j b) → j (a → b)
```

```
class FunC j where
  lam :: (j a → j b) → j (a → b)

  app :: j (a → b) → j a → j b
```

```
class FunC j where
  lam :: (j a → j b) → j (a → b)
  fix :: (j (a → b) → j (a → b)) → j (a → b)
  app :: j (a → b) → j a → j b
```

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$

$$\begin{aligned} foldr :: (FunC\ j, ListC\ j) &\Rightarrow (j\ a \to j\ b \to j\ b) \to j\ b \to j\ [a] \to j\ b \\ foldr\ f\ b\ xs &= fix\ (\lambda r \to lam\ (list\ b\ (\lambda y\ ys \to f\ y\ (r\ `app`\ ys)))) \\ &\qquad `app`\ xs \end{aligned}$$

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$

$$\begin{aligned}
&foldr :: (FunC\ j, ListC\ j) \Rightarrow (j\ a \to j\ b \to j\ b) \to j\ b \to j\ [a] \to j\ b \\
&foldr\ f\ b\ xs = fix\ (\lambda r \to lam\ (list\ b\ (\lambda y\ ys \to f\ y\ (r\ `app`\ ys))))\\
&\qquad\qquad\qquad\qquad `app`\ xs
\end{aligned}$$

$$\begin{aligned}
&jsFoldr :: (JavaScript\ a \to JavaScript\ b \to JavaScript\ b) \to JavaScript\ b \\
&\qquad\qquad \to JavaScript\ [a] \to JavaScript\ b \\
&jsFoldr = foldr
\end{aligned}$$

```
type Nm = String

data JavaScript a where
  Con    :: Nm                                  -> JavaScript a
  Prim   :: ([Nm] -> Nm) -> [Nm]                -> JavaScript a
  App    :: JavaScript (a -> b) -> JavaScript a -> JavaScript b
  Lam    :: (JavaScript a -> JavaScript b)      -> JavaScript (a -> b)
  Var    :: Nm                                  -> JavaScript a
  Name   :: Nm -> JavaScript a                  -> JavaScript a
```

```
instance BoolC JavaScript where
  -- constructors:
  true  = Con "true"
  false = Con "false"
  -- destructor:
  bool x y z = fun3 "bool"
    (λ[e,t,b] → concat [b,"?",t,"():",e,"()"])
    (lam (const x)) (lam (const y)) z
```

```
instance FunC JavaScript where
  lam f   = Lam f
  app f g = App f g
```

```
instance ListC JavaScript where
  -- constructors:
  nil  = Con "{nil:1}"
  cons = fun2 "cons"
    (λ[x,xs] → concat ["{head:",x,",tail:",xs,"}"])
  -- destructor:
  list b f = fun3 "list"
    (λ[n,c,xs] → concat
      [xs,".nil?",n,":",c,"(",xs,".head)(",xs,".tail)"]
    ) b (lam2 f)
```

```
type a ↦ b = Kleisli IO a b
type Code = String


compiler :: JavaScript a → Code
compiler = runKleisli
    $ (Lambdas.instantiate      :: JavaScript a  ↦ Expression   )
    ∘ (Defs.lift                :: Expression    ↦ Definitions  )
    ∘ (Defs.eliminiateDoubles   :: Definitions   ↦ Definitions  )
    ∘ (FreeVars.annotateDefs    :: Definitions   ↦ DefinitionsFV)
    ∘ (ClosedApplications.lift  :: DefinitionsFV ↦ Definitions  )
    ∘ (Parameters.reindex       :: Definitions   ↦ Definitions  )
    ∘ (CommonDefs.eliminate     :: Definitions   ↦ Definitions  )
    ∘ (Defs.dump                :: Definitions   ↦ Code         )
```

```
test :: Haskell (Num → Num)
test = lam (λx → sum (replicate 3 (2 * 8) ++ replicate 3 8)
                  *  maybe 4 ( * 8) (just (x - 2)))
```

```
test :: Haskell (Num → Num)
test = lam (λx → sum (replicate 3 (2 * 8) ++ replicate 3 8)
                  *  maybe 4 ( * 8) (just (x - 2)))
```

```
ghci> (runHaskell test) 3
576
```

```
test :: JavaScript (Num → Num)
test = lam (λx → sum (replicate 3 (2 * 8) ++ replicate 3 8)
                 *  maybe 4 ( * 8) (just (x - 2)))
```

```
test :: JavaScript (Num → Num)
test = lam (λx → sum (replicate 3 (2 * 8) ++ replicate 3 8)
                  *  maybe 4 ( * 8) (just (x - 2)))
```

```
ghci> Js.compiler test >>=
      writeFile "test.js"
```

JavaScript!

```
var mul = function (v1) { return function (v2) { return v1 * v2; }; }; var fix = function (v1) { return fix
= arguments.callee, v1(function (i) { return fix(v1)(i) }); }; var list = function (v1) { return function
(v2) { return function (v3) { return v3.nil ? v1 : v2(v3.head)(v3.tail); }; }; }; var add = function (v1) {
return function (v2) { return v1 + v2; }; }; var bool = function (v1) { return function (v2) { return
function (v3) { return v3 ? v1(/*force*/) : v2(/*force*/); }; }; }; var cons = function (v1) { return
function (v2) { return { head : v1, tail : v2 }; }; }; var sub = function (v1) { return function (v2) {
return v1 - v2; }; }; var eq = function (v1) { return function (v2) { return v1 == v2; }; }; var maybe =
function (v1) { return function (v2) { return function (v3) { return v3.nothing ? v1 : v2(v3.just); }; }; };
var just = function (v1) { return { just : v1 }; }; var c10_11 = list(0); var c10_12 = function (v1) {
return function (v2) { return c10_11(function (v3) { return function (v4) { return add(v3)(v1(v4)); }; })
(v2); }; }; var c10_13 = fix(c10_12); var c10_14 = function (v1) { return function (v2) { return v1; }; };
var c10_15 = c10_14({ nil : 1 }); var c10_16 = function (v1) { return c10_15(v1); }; var c10_17 = bool(
c10_16); var c10_19 = cons(8); var c10_20 = function (v1) { return function (v2) { return c10_17(function
(v3) { return c10_14(c10_19(v1(sub(v2)(1))))(v3); })(eq(v2)(0)); }; }; var c10_21 = fix(c10_20); var c10_22
= c10_21(3); var c10_23 = list(c10_22); var c10_24 = function (v1) { return function (v2) { return c10_23(
function (v3) { return function (v4) { return cons(v3)(v1(v4)); }; })(v2); }; }; var c10_25 = fix(c10_24);
var c10_31 = mul(2); var c10_32 = c10_31(8); var c10_33 = cons(c10_32); var c10_34 = function (v1) {
return function (v2) { return c10_17(function (v3) { return c10_14(c10_33(v1(sub(v2)(1))))(v3); })(eq(v2)
(0)); }; }; var c10_35 = fix(c10_34); var c10_36 = c10_35(3); var c10_37 = c10_25(c10_36); var c10_38 =
c10_13(c10_37); var c10_39 = mul(c10_38); var c10_40 = maybe(4); var c10_41 = function (v1) { return
mul(v1)(8); }; var c10_42 = c10_40(c10_41); var __main = function (v1) { return c10_39(c10_42(just(sub(v1)
(2)))); };
```

```
alert(__main(3));
```

This prototype

- Abstract away from concrete datatypes.
- Abstract away from functions.
- Replace with type classes.

- Different instances for different computational contexts.
- Functions look similar.
- Types get complicated.

- Plain lazy and purely functional Haskell.
- Purely functional strict JavaScript.
- Functional reactive JavaScript.

Current problems

- Explicit lifting of function application and recursion.
- Type signatures with big contexts.
- No sugar for pattern matching, let bindings, if-then-else.
- Reimplementing the entire Haskell Prelude.
- Lots of manual instances for every datatype and context.

Future work

- Syntactic front-end.
- Additional computational contexts:
  - Strict Haskell.
  - Functional Reactive Haskell.
  - Profiling support.
  - C, Objective-C, C#, etc...
- Generic derivation of instances.
- Improved optimizing compiler.
- Single computation over different contexts.