Theory of Programming and Types
Exercise 3

Robert Hensing
3361063

June 19, 2015

Contents

1 Exercise 1, datatype kinds

```haskell
data Tree a b   = Tip a | Branch (Tree a b) b (Tree a b)
data GList f a = GNil | GCons a (GList f (f a))
data Bush a     = Bush a (GList Bush (Bush a))
data HFix f a  = HIn {hout :: f (HFix f) a}
data Exists b where
  Exists :: a → (a → b) → Exists b
data Exp where
  Bool   :: Bool → Exp
  Int    :: Int → Exp
  GT     :: Exp → Exp → Exp
  Add    :: Exp → Exp → Exp   -- I have added this
  IsZero :: Exp → Exp
  Succ   :: Exp → Exp
  If     :: Exp → Exp → Exp → Exp
```

These are the kinds of the types. The type aliases only provide a place to put the kind signatures.

$$
\begin{array}{l}
\textbf{type}\ \mathit{TreeK}\ \ \ = (\mathit{Tree}\ \ :: * \to * \to *)\\
\textbf{type}\ \mathit{GListK}\ = (\mathit{GList} :: (* \to *) \to * \to *)\\
\textbf{type}\ \mathit{BushK}\ \ = (\mathit{Bush}\ :: * \to *)\\
\textbf{type}\ \mathit{HFixK}\ \ = (\mathit{HFix}\ :: ((* \to *) \to * \to *) \to * \to *)\\
\textbf{type}\ \mathit{ExistsK} = (\mathit{Exists} :: * \to *)\\
\textbf{type}\ \mathit{ExpK}\ \ \ \ = (\mathit{Exp}\ \ \ :: *)
\end{array}
$$

2 Exercise 2

2.1 Exercise 2a, eval Exp

The eval function evaluates Exps:

$$
\begin{array}{l}
\mathit{eval} :: \mathit{Exp} \to \mathit{Maybe}\ (\mathit{Either}\ \mathit{Int}\ \mathit{Bool})\\
\mathit{eval}\ (\mathit{GT}\ a\ b)\ \ = \mathit{asBool}\ \langle\$\rangle\ ((>)\ \ \ \langle\$\rangle\ \mathit{evalInt}\ a\ \langle*\rangle\ \mathit{evalInt}\ b)\\
\mathit{eval}\ (\mathit{Add}\ a\ b)\ = \mathit{asInt}\ \ \ \langle\$\rangle\ ((+)\ \ \ \langle\$\rangle\ \mathit{evalInt}\ a\ \langle*\rangle\ \mathit{evalInt}\ b)\\
\mathit{eval}\ (\mathit{If}\ c\ t\ e)\ \ = \qquad\qquad \mathit{bool}\ \ \ \ \langle\$\rangle\ \mathit{eval}\ e\ \langle*\rangle\ \mathit{eval}\ t\ \langle*\rangle\ \mathit{evalBool}\ c\\
\mathit{eval}\ (\mathit{Bool}\ b)\ \ \ = \mathit{return}\ (\mathit{asBool}\ b)\\
\mathit{eval}\ (\mathit{Int}\ i)\ \ \ \ \ \ = \mathit{return}\ (\mathit{asInt}\ \ \ i)\\
\mathit{eval}\ (\mathit{IsZero}\ n) = \mathit{asBool}\ \langle\$\rangle\ ((\equiv 0)\ \langle\$\rangle\ \mathit{evalInt}\ n)\\
\mathit{eval}\ (\mathit{Succ}\ n)\ \ \ = \mathit{asInt}\ \ \ \langle\$\rangle\ ((+1)\ \ \langle\$\rangle\ \mathit{evalInt}\ n)\\
\\
\mathit{evalInt} :: \mathit{Exp} \to \mathit{Maybe}\ \mathit{Int}\\
\mathit{evalInt}\ e = \mathit{eval}\ e \ggg\!= \mathit{safeLeft}\\
\\
\mathit{evalBool} :: \mathit{Exp} \to \mathit{Maybe}\ \mathit{Bool}\\
\mathit{evalBool}\ e = \mathit{eval}\ e \ggg\!= \mathit{safeRight}\\
\\
\mathit{asInt} = \mathit{Left}\\
\mathit{asBool} = \mathit{Right}\\
\\
\mathit{safeLeft} :: \mathit{Either}\ a\ b \to \mathit{Maybe}\ a\\
\mathit{safeLeft}\ (\mathit{Left}\ a) = \mathit{Just}\ a\\
\mathit{safeLeft}\ \_\ \ \ \ \ \ \ \ \ \ = \mathit{Nothing}\\
\\
\mathit{safeRight} :: \mathit{Either}\ a\ b \to \mathit{Maybe}\ b\\
\mathit{safeRight}\ (\mathit{Right}\ b) = \mathit{Just}\ b\\
\mathit{safeRight}\ \_\ \ \ \ \ \ \ \ \ \ \ \ = \mathit{Nothing}
\end{array}
$$

2.2 Exercise 2b, Fix and Exp'

We define Exp' isomorphically to Exp using Fix by defining ExpF

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{Fix}\ f = \mathit{In}\ \{\mathit{out} :: f\ (\mathit{Fix}\ f)\}\\
\textbf{type}\ \mathit{Exp}'\ \ \ \ \ \ \ \ = \mathit{Fix}\ \mathit{ExpF}\\
\\
\textbf{data}\ \mathit{ExpF}\ e\ \textbf{where}\\
\quad \mathit{Bool}'\ \ \ :: \mathit{Bool} \to \qquad \mathit{ExpF}\ e
\end{array}
$$

```
Int'    :: Int →                ExpF e
GT'     :: e → e →              ExpF e
Add'    :: e → e →              ExpF e
IsZero' :: e →                  ExpF e
Succ'   :: e →                  ExpF e
If'     :: e → e → e →          ExpF e
```

2.3 Exercise 2c, functor and algebra

```
instance Functor ExpF where
  fmap f (Bool' b)     = Bool'   b
  fmap f (Int' i)      = Int'    i
  fmap f (GT' a b)     = GT'     (f a) (f b)
  fmap f (Add' a b)    = Add'    (f a) (f b)
  fmap f (IsZero' n)   = IsZero' (f n)
  fmap f (Succ' n)     = Succ'   (f n)
  fmap f (If' c t e)   = If'     (f c) (f t) (f e)

fold :: Functor f ⇒ (f a → a) → Fix f → a
fold f = f ∘ fmap (fold f) ∘ out
  where out = ⊥

eval' :: Exp' → Maybe (Either Int Bool)
eval' = fold evalAlg
  where
    evalAlg :: ExpF (Maybe (Either Int Bool)) → Maybe (Either Int Bool)
    evalAlg (Bool' b)    = Just (Right b)
    evalAlg (Int' i)     = Just (Left i)
    evalAlg (GT' a b)    = asBool ⟨$⟩ ((>) ⟨$⟩ getInt a ⟨*⟩ getInt b)
    evalAlg (Add' a b)   = asInt ⟨$⟩ (  (+) ⟨$⟩ getInt a ⟨*⟩ getInt b)
    evalAlg (IsZero' n)  = asBool ⟨$⟩ ((0 ≡) ⟨$⟩ getInt n)
    evalAlg (Succ' n)    = asInt ⟨$⟩ ((1+) ⟨$⟩ getInt n)
    evalAlg (If' c t e)  =                bool ⟨$⟩ e ⟨*⟩ t ⟨*⟩ getBool c
    getBool = (≫= safeRight)
    getInt  = (≫= safeLeft)
```

2.4 Exercise 2d, well-typed language

```
type ExpT' = HFix ExpTF
data ExpTF e t where
  PureTF   :: a →                       ExpTF e a
  GTTF     :: e Int → e Int →           ExpTF e Bool
  AddTF    :: e Int → e Int →           ExpTF e Int
  IsZeroTF :: e Int →                   ExpTF e Bool
  SuccTF   :: e Int →                   ExpTF e Int
  IfTF     :: e Bool → e a → e a →      ExpTF e a
```

This expression can be evaluated but would be ill-typed

$$example = If\ (Bool\ True)\ (Succ\ (Int\ 12))\ (Bool\ False)$$

An attempt to define the ill-typed expression in the well-typed representation:

$$IfTF\ (PureTF\ True)\ (PureTF\ (12 :: Int))\ (PureTF\ False)$$

The Haskell type checker will complain that it can not match Int with Bool, as expected.

2.5 Exercise 2e, HFunctor

The assignment defines the following:

$$\begin{array}{l}
\textbf{class}\ HFunctor\ f\ \textbf{where} \\
\quad hfmap :: (forall\ b \circ g\ b \rightarrow h\ b) \rightarrow f\ g\ a \rightarrow f\ h\ a \\
\\
hfold :: HFunctor\ f \Rightarrow (forall\ b \circ f\ r\ b \rightarrow r\ b) \rightarrow HFix\ f\ a \rightarrow r\ a \\
hfold\ f = f \circ hfmap\ (hfold\ f) \circ hout \\
\\
\quad \text{-- newtype Id a = Id unId :: a} \\
\textbf{type}\ Id = Identity \quad \text{-- I would rather re-use Identity} \\
unId = runIdentity \\
\\
evalT' :: ExpT'\ a \rightarrow a \\
evalT' = unId \circ hfold\ evalAlgT \\
\\
evalAlgT :: ExpTF\ Id\ a \rightarrow Id\ a
\end{array}$$

These definitions can be used to implement the evaluation function:

$$\begin{array}{l}
\textbf{instance}\ HFunctor\ ExpTF\ \textbf{where} \\
\quad hfmap\ f\ (PureTF\ v) \quad\ = PureTF\ v \\
\quad hfmap\ f\ (GTTF\ a\ b) \ \ = GTTF\ (f\ a)\ (f\ b) \\
\quad hfmap\ f\ (AddTF\ a\ b) = AddTF\ (f\ a)\ (f\ b) \\
\quad hfmap\ f\ (IsZeroTF\ a) = IsZeroTF\ (f\ a) \\
\quad hfmap\ f\ (SuccTF\ a) \quad = SuccTF\ (f\ a) \\
\quad hfmap\ f\ (IfTF\ c\ t\ e) \quad = IfTF\ (f\ c)\ (f\ t)\ (f\ e)
\end{array}$$

The identity data type is an applicative functor, which is useful.

$$\begin{array}{l}
evalAlgT\ (PureTF\ v) \quad\ = pure\ v \\
evalAlgT\ (GTTF\ a\ b) \ \ = (>) \quad \langle\$\rangle\ a\ \langle*\rangle\ b \\
evalAlgT\ (AddTF\ a\ b) = (+) \quad \langle\$\rangle\ a\ \langle*\rangle\ b \\
evalAlgT\ (IsZeroTF\ a) = (0 \equiv)\ \langle\$\rangle\ a \\
evalAlgT\ (SuccTF\ a) \quad = (1+) \ \ \langle\$\rangle\ a \\
evalAlgT\ (IfTF\ c\ t\ e) \quad = bool \quad \langle\$\rangle\ e\ \langle*\rangle\ t\ \langle*\rangle\ c
\end{array}$$

3 Exercise 3, children

This is the user-visible function:

$$\begin{array}{l}
children :: (R.Regular\ r, Children\ (PF\ r)) \Rightarrow r \to [r] \\
children = children' \circ R.from
\end{array}$$

$$\begin{array}{l}
\textbf{class}\ Children\ f\ \textbf{where} \\
\quad children' :: f\ a \to [a]
\end{array}$$

A recursive position has a single child

$$\begin{array}{l}
\textbf{instance}\ Children\ I\ \textbf{where} \\
\quad children'\ (I\ x) = [x]
\end{array}$$

The unit and constants have no children

$$\begin{array}{l}
\textbf{instance}\ Children\ U\ \textbf{where} \\
\quad children'\ _ = [\,]
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ Children\ (K\ f)\ \textbf{where} \\
\quad children'\ _ = [\,]
\end{array}$$

In case of a sum, use whichever we get

$$\begin{array}{l}
\textbf{instance}\ (Children\ l, Children\ r) \Rightarrow Children\ (l :+: r)\ \textbf{where} \\
\quad children'\ (L\ l) = children'\ l \\
\quad children'\ (R\ r) = children'\ r
\end{array}$$

In case of a product, give both

$$\begin{array}{l}
\textbf{instance}\ (Children\ l, Children\ r) \Rightarrow Children\ (l :*: r)\ \textbf{where} \\
\quad children'\ (l :*: r) = children'\ l +\!\!+\ children'\ r
\end{array}$$

Ignore any constructor or selector names

$$\begin{array}{l}
\textbf{instance}\ (Children\ f) \Rightarrow Children\ (C\ c\ f)\ \textbf{where} \\
\quad children'\ (C\ a) = children'\ a
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ (Children\ f) \Rightarrow Children\ (S\ c\ f)\ \textbf{where} \\
\quad children'\ (S\ a) = children'\ a
\end{array}$$

$$\begin{array}{l}
\$\,(R.deriveAll\ \text{''}\ Exp\ \texttt{"PFExp"}) \\
\textbf{type instance}\ PF\ Exp = PFExp
\end{array}$$

`If` in `Exp` has three children

$$length\ (children\ example) \equiv 3$$

4 Exercise 4, parents

The user-visible function:

$$\begin{array}{l}
parents :: (R.Regular\ r, Parents\ (PF\ r), Children\ (PF\ r)) \Rightarrow r \to [r] \\
parents\ x = includeself \circ parents'\ parents \circ R.from\ \$\ x \\
\quad \textbf{where}\ includeself = \textbf{if}\ null\ (children\ x) \\
\qquad \textbf{then}\ id \\
\qquad \textbf{else}\ (x{:})
\end{array}$$

Because `parents'` needs access to the `parents` function and the proper `Parents` context is lost, we need to pass it explicitly.

$$\textbf{class}\ \mathit{Parents}\ f\ \textbf{where}$$
$$\quad \mathit{parents}' :: (a \to [\,a\,]) \to f\ \ a \to [\,a\,]$$

Constant and unit have no children, so we don't need to look for parents (children with children) in them.

$$\textbf{instance}\ \mathit{Parents}\ I\ \textbf{where}$$
$$\quad \mathit{parents}'\ p\ (I\ \ x) = p\ x$$

$$\textbf{instance}\ \mathit{Parents}\ U\ \textbf{where}$$
$$\quad \mathit{parents}'\ p\ \_ = [\,]$$

$$\textbf{instance}\ \mathit{Parents}\ (K\ f)\ \textbf{where}$$
$$\quad \mathit{parents}'\ p\ \_ = [\,]$$

In case of a sum, use whichever we get

$$\textbf{instance}\ (\mathit{Parents}\ l, \mathit{Parents}\ r) \Rightarrow \mathit{Parents}\ (l :+: r)\ \textbf{where}$$
$$\quad \mathit{parents}'\ p\ (L\ l) = \mathit{parents}'\ p\ l$$
$$\quad \mathit{parents}'\ p\ (R\ r) = \mathit{parents}'\ p\ r$$

In case of a product, give both

$$\textbf{instance}\ (\mathit{Parents}\ l, \mathit{Parents}\ r) \Rightarrow \mathit{Parents}\ (l :*: r)\ \textbf{where}$$
$$\quad \mathit{parents}'\ p\ (l :*: r) = \mathit{parents}'\ p\ l \mathbin{+\!\!+} \mathit{parents}'\ p\ r$$

Ignore any constructor or selector names

$$\textbf{instance}\ (\mathit{Parents}\ f) \Rightarrow \mathit{Parents}\ (C\ \ c\ f)\ \textbf{where}$$
$$\quad \mathit{parents}'\ p\ (C\ \ a) = \mathit{parents}'\ p\ a$$

$$\textbf{instance}\ (\mathit{Parents}\ f) \Rightarrow \mathit{Parents}\ (S\ \ c\ f)\ \textbf{where}$$
$$\quad \mathit{parents}'\ p\ (S\ \ a) = \mathit{parents}'\ p\ a$$