Reasoning in Haskell

Greg Price (price)

2008 Jan 31

Reasoning in Haskell: Hindley-Milner Types

- C: `void qsort(void *xs, int, int, int cmp(void*, void*))`
- Haskell: `qsort :: (t -> t -> Int) -> [t] -> [t]` (∀t)
- H-M type allows type variables, universally quantified (with ∀)
- automatic *type inference*—never need declare a variable's type!
- lexical distinction by caps: `variable`, `Fixed`

```
map :: (a -> b) -> [a] -> [b]
all :: (a -> Bool) -> [a] -> Bool
takeWhile :: -- ?
(.) :: (b -> c) -> (a -> b) -> (a -> b)
($) :: -- ?
```

Reasoning in Haskell: IO Actions

- Haskell *pure*, evaluation no side effects, always same
- so `putStr "hello" :: IO ()` returns an "`IO` action"
- some actions produce values: `getLine :: IO String`
- composite actions: `do { x <- action1; action2; action3 x }`
- ultimately execute an action by calling it `main`

```
do { name <- getLine; putStr $ "Hi, " ++ name ++ "\n" }
-- or reformat as
do name <- getLine
   putStr $ "Hi, " ++ name ++ "\n"

-- complete program:
#!/usr/bin/runhaskell
main = putStr "hello world\n"
```

Reasoning in Haskell: More Actions

- `readTVar v :: STM a` an STM action
- execute STM actions using `atomically :: STM a -> IO a`
- keep an `s` as state: `get :: State s s`, `put :: s -> State s ()`
- run a `State s` action with `evalState :: State s a -> s -> a`
- typical language interpreter in Haskell: interpret statements as actions

```
atomically (do x <- readTVar v
               y <- readTVar u
               return (x-y)    ) :: IO Int
```

Reasoning in Haskell: Monads

- *monad*: a genre of actions, like `IO` or `STM` or `State Int`
- must have
  - a way to make trivial actions: `return :: a -> MyMonad a`
  - a way to chain actions: `do { ... }`
  - to be useful, primitives and a way to carry out actions
- `do {x <- act1; act2 x}` is sugar for `act1 >>= (\x -> act2 x)`
- so `return :: a -> m a`, `(>>=) :: m a -> (a -> m b) -> m b`
- algebraic laws: `(return x) >>= f` ≡ `f x`, others
- inspiration, name, laws come from category theory

Reasoning in Haskell: Type Classes

- *type class*: an interface or constraint on a type, like "is a monad"
- `class (..constraints..) => MyClass t where ..methods..`
- similarly `instance` to implement the interface

```
-- in Prelude:
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

-- in Control.Monad.State:
newtype State s a = State { runState :: s -> (a, s) }
instance Monad (State s) where
  return x = State $ \s -> (x, s)
  m >>= f  = State $ \s -> let (x, s') = runState m s
                           in runState (f x) s'
```

Reasoning in Haskell: Type Classes II

- classics `Eq`, `Ord`, `Num`; stringifying `Read`, `Show`
- Monad instances are *higher-kind types* like `IO :: * -> *`
- literal `1` means `fromInteger 1 :: Num a => a` (as we noticed)
- instances may rely on constraints, posing a logic problem

```
> :t 3
3 :: (Num t) => t
> :i fromInteger
class (Eq a, Show a) => Num a where
  ...
  fromInteger :: Integer -> a

instance (Eq a, Eq b) => Eq (a, b) where
  (a, b) == (a', b')  =  (a == a') && (b == b')
> ("hello", (False, 'c')) == ("hello", (False, 'c'))
True
```

Reasoning in Haskell: Algebraic (& your own!) Data Types

- ADT: `data MyType tyarg = Branch1 | Branch2 Int [tyarg]`
- take apart with `case exp of pat1 -> body1 ; pat2 -> body2`
- or write a case as "equations"
- `if` is sugar for `case .. of True -> .. ; False -> ..`
- can give names to members, making gettors

```
data Bool = True | False              -- built in
data List a = Nil | Cons a (List a)   -- but sugar is nicer

last :: [a] -> a
last [] = error "oops"
last [x] = x
last (x:xs) = last xs

data Queue a = Queue {hd :: Int, tl :: Int, a :: Array Int a}
> :t hd
hd :: forall a. Queue a -> Int
```