Advanced Functional Programming, 1DL450

Lecture 10, 2012-12-10

Cons T Åhs

Monads revisited

- Monads combine pure functional computation with side effects.
- Sequencing and order of computation are important for the side effects.
- With monads it is much more clear which parts contain side effects/are dependent on order and which are purely functional.
- For I/O, graphics etc it is “obvious” that we need sequential computation.
- It is less obvious how sequentiality is described formally, especially when combined with a lazy language.
- Using the formal description one can use monads to describe more types of computation than just sequential.
- A monad contains (at least) two operators
  - *bind* or >>= used for sequential operation and (optional) binding of name
  - *return* for extracting values from the monad

Monad Axioms

- *Return* acts a "neutral" (or identity; `x + 0 = 0 + x = x`) element of *bind*

```
(return x) >>= f ≡ f x
    m >>= return ≡ m
```

- Binding two functions in succession is the same as binding one function that can be determined from them
  - This is a composition property

```
(m >>= f) >>= g ≡ m >>= (\x -> (f x >>= g))
```

- Optional *zero* value (`x*0 = 0*x = 0`)

```
mzero >>= f ≡ mzero

m >>= (\x -> mzero) ≡ mzero
```

Maybe Monad

- Computations might fail (in general)
- Can we express and propagate failed computations in an easy way?
  - Compare to exceptions; skip rest of computation if one fails

```
data Maybee a = The a | Nil deriving Show

instance Monad Maybee where
  (The x) >>= f = f x
  Nil >>= _ = Nil
  return x = The x

madd x y =
  do
    x1 <- x
    y1 <- y
    return (x1 + y1)

*Main> :type madd
madd :: (Monad m, Num b) => m b -> m b -> m b
*Main> madd (The 2) (The 3)
The 5
*Main> madd (The 2) Nil
Nil
```

Maybe Monad

- Be aware the sequencing/bind “unpacks” values inside monads
- The following two definitions of monadic add are not equivalent
  - observe the very different types!

```
madd x y =
  do
    x1 <- x
    y1 <- y
    return (x1 + y1)

madd2 x y =
  do return (x + y)

*Main> :type madd
madd :: (Monad m, Num b) => m b -> m b -> m b
*Main> :type madd2
madd2 :: (Monad m, Num a) => a -> a -> m a
```

List Monad

- Use a monad for a set of possible results
- Note the it makes sense to read "<-" as "belongs to" or "in"
- Very much like list comprehension

```
instance Monad [] where
  -- single possible result
  return a = [a]
  -- apply to each element and flatten results
  xs >>= f = concat (map f xs)
  -- failure is no results
  fail s = []
```

```
Prelude> do x<-[1,2,3]; y<-[4,5]; return (x*y)
[4,5,8,10,12,15]
Prelude> do x<-[1,2,3]; y<-[4,5]; if x > 1 then return (x*y)
else fail ""
[8,10,12,15]
```

Other Monads

- State monad
  - our previous cell example was a state monad
- Environment Monad
  - compute with values from a (non local) context
- Writer monad
  - perform computations while writing “output” on the side
- See Wikipedia page for more details.