Informatics 1

Functional Programming Lectures 17 and 18

Monday 24 and Tuesday 25 November 2008

Arithmetic

Philip Wadler

University of Edinburgh

The 2008 Informatics 1 Competition

- Prize: A bottle of champagne or book token equivalent
- Sponsored by Galois (galois.com)
- List everyone who worked on the entry
  If you win, do you want Champagne or a book token?
- Deadline: 4pm Friday 28 November 2008
  email to ¡w.b.heijltjes@sms.ed.ac.uk¿
- You may find some inspiration here:

  www.contextfreeart.org

  (Thanks to Aleksandar Krastev for the suggestion.)

Required reading

Haskell: The Craft of Functional Programming, Second Edition,
Simon Thompson, Addison-Wesley, 1999.

Thompson, Chapters 1–3 (pp. 1–52): by Mon 29 Sep 2008.
Thompson, Chapters 4–5 (pp. 53–95): by Mon 6 Oct 2008.
Thompson, Chapters 6–7 (pp. 96–134): by Mon 13 Oct 2008.
Thompson, Chapters 8–9 (pp. 135–166): by Mon 20 Oct 2008.
Thompson, Chapters 10–11 (pp. 167–209): by Mon 3 Nov 2008.
Thompson, Chapters 12–14 (pp. 210–279): by Mon 10 Nov 2008.
Thompson, Chapters 15–17 (pp. 280–382): by Mon 17 Nov 2008.
Thompson, Chapters 18–20 (pp. 338–441): by Mon 24 Nov 2008.

Thompson and other books available in ITO.

Part I

Arithmetic over Naturals

Naturals

```
data Nat = Z | S Nat
```

Values

`Z` stands for `0` — zero

`S n` stands for `n+1` — successor

Arithmetic

```
(+) :: Nat -> Nat -> Nat
m + Z       =  m
m + (S n)   =  S (m + n)

(*) :: Nat -> Nat -> Nat
m * Z       =  Z
m * (S n)   =  (m * n) + m

(^) :: Nat -> Nat -> Nat
m ^ Z       =  S Z
m ^ (S n)   =  (m ^ n) * m
```

An example of addition

```
  3 + 2
=
  (S (S (S Z))) + (S (S Z))
=
  S ((S (S (S Z))) + (S Z))
=
  S (S ((S (S (S Z))) + Z))
=
  S (S (S (S (S Z))))
```

An example of multiplication

```
  3 * 2
=
  (S (S (S Z))) * (S (S Z))
=
  ((S (S (S Z))) * (S Z)) + (S (S (S Z)))
=
  (((S (S (S Z))) * Z) + (S (S (S Z)))) + (S (S (S Z)))
=
  (Z + (S (S (S Z)))) + (S (S (S Z)))
=
  S (S (S (S (S (S Z)))))
```

In Haskell notation

```
(+) :: Int -> Int -> Int
m + 0        =  m
m + (n+1)    =  (m + n)+1

(*) :: Int -> Int -> Int
m * 0        =  0
m * (n+1)    =  (m * n) + m

(^) :: Int -> Int -> Int
m ^ 0        =  1
m ^ (n+1)    =  (m ^ n) * m
```

Type classes

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a

instance Num Int where
  m + 0         =  m
  m + (n+1)     =  (m + n)+1

  m ^ 0         =  1
  m ^ (n+1)     =  (m ^ n) * m

(^) :: (Num a, Integral b) => a -> b -> a
x ^ 0         =  1
x ^ (n+1)     =  (x ^ n) * x
```

Part II

Arithmetic over Types

Tuples

```
data Pair a b = Pair a b
```

Type

`(a,b)` stands for `Pair a b`

Values

`(x,y)` stands for `Pair x y`

Arithmetic

If there are m values `x :: a`,
and n values `y :: b`,
then there are $m \times n$ values `(x,y) :: (a,b)`.

Set theory

`(a,b)` is the *cartesian product* of `a` and `b`.

Tuples

```
data Bool     =  False | True
data Colour   =  Red | Green | Blue
```

Arithmetic

There are $2 \times 3 = 6$ values of type `(Bool, Colour)`.

```
(False, Red)
(False, Green)
(False, Blue)
(True,  Red)
(True,  Green)
(True,  Blue)
```

Unit

```
data Unit = Unit
```

Type

`()` stands for `Unit`

Values

`()` stands for `Unit`

Arithmetic

There is 1 value `()` of type `()`.

Set theory

`()` is a *singleton set*.

Unit

```
data Colour  =  Red | Green | Blue
```

Arithmetic

There are $1 \times 3 = 3$ values of type `((), Colour)`.

```
((), Red)
((), Green)
((), Blue)
```

Either

```
data Either a b  =  Left a | Right b
```

Type

`Either a b`

Values

`Left x`
`Right y`

Arithmetic

If there are m values `x :: a`,
and n values `y :: b`,
then there are $m + n$ values `Left x`, `Right y :: Either a b`.

Set theory

`Either a b` is the *disjoint union* of `a` and `b`.

Either

```
data Bool = False | True
data Colour = Red | Green | Blue
```

Arithmetic

There are $2 + 3 = 5$ values of type `Either Bool Colour`.

```
Left False
Left True
Right Red
Right Green
Right Blue
```

Empty

```
data Empty
```

Type

`Empty`

Values

(there are none!)

Arithmetic

There are 0 values of type `Empty`.

Set theory

`Empty` is the *empty set*.

Empty

```
data Colour = Red | Green | Blue
```

Arithmetic

There are $0 + 3 = 3$ values of type `Either Empty Colour`.

```
Right Red
Right Green
Right Blue
```

(there are no values `Left x`!)

Booleans

data Bool = False — True

Correspondence

`Either () ()` corresponds to `Bool`
`Left ()` corresponds to `False`
`Right ()` corresponds to `True`

Arithmetic

There are two values `False, True :: Bool`.

$$1 + 1 = 2$$

Maybe

```
data Maybe a  =  Nothing | Just a
```

Correspondence

`Either Unit a` correspond to `Maybe a`

`Left ()` corresponds to `Nothing`

`Right x` corresponds to `Just x`

Arithmetic

If there are m values `x :: a`,

then there are $m + 1$ values `Nothing, Just x :: Maybe a`.

A use of Maybe

Comprehension

```
lookup :: a -> [(a,b)] -> Maybe b
lookup x xys  =  f [ y' | (x',y') <- xys, x == x' ]
  where
  f []       =  Nothing
  f (y:ys)   =  Just y
```

Recursion

```
lookup :: a -> [(a,b)] -> Maybe b
lookup x []                    =  Nothing
lookup x ((x',y'):xys)
  | x == x'                    =  Just x
  | otherwise                  =  lookup x xys
```

Lists

```
data List a = Nil | Cons a (List a)
```

Type

`[a]` stands for `List a`

Values

`[]` stands for `Nil`

`x:xs` stands for `Cons x xs`

Correspondence

`Either () (a, List a)` corresponds to `List a`

`Left ()` corresponds to `[]`

`Right (x,xs)` corresponds to `x:xs`

Naturals

```
data Nat = Z | S Nat
```

Type

`Int` (often) stands for `Nat`

Values

`0` stands for `Z` — zero

`n+1` stands for `S n` — successor

Correspondence

`Either () Nat` corresponds to `Nat`

`Left ()` corresponds to `0`

`Right n` corresponds to `n+1`

Functions

The one data type that is not an algebraic type!

Type

```
a -> b
```

Values

```
\x -> y
```

where `x :: a` and `y :: b`

Arithmetic

If there are m values `x :: a`
and n values `y :: b`
then there are n^m functions `\x -> y :: a -> b`.

Representing functions

Sometimes we represent a function with list of pairs.

```
type Fun a b = [(a,b)]

nilFun :: a -> b
nilFun x  =  undefined

consFun :: (Eq a) => (a,b) -> (a -> b) -> (a -> b)
consFun (x,y) f x' | x == x'     =  y
                   | otherwise   =  f x'

convert :: (Eq a) => Fun a b -> (a -> b)
convert xys x  =  foldr consFun nilFun xys
```

Observe

```
convert []              =  nilFun
convert ((x,y):xys)     =  consFun (x,y) (convert xys)
```

Representing functions

Remarkably, we have `convert == lookUp`

```
lookUp :: (Eq a) => Fun a b -> a -> b
lookUp xys x  =  the [ y | (x',y) <- xys, x == x' ]
  where
  the [x]  =  x
```

Part III

Arithmetic over Lists

Arithmetic over lists

```
(++) :: [a] -> [a] -> [a]
[] ++ ys        =  ys
(x:xs) ++ ys    =  x : (xs ++ ys)

(**) :: [a] -> [b] -> [(a,b)]
xs ** ys        =  [(x,y) | x <- xs, y <- ys]

(^^) :: [b] -> [a] -> [[(a,b)]]
ys ^^ []        =  [[]]
ys ^^ (x:xs)    =  [(x,y):e | y <- ys, e <- ys^^xs]
```

Arithmetic over lists, revisited

```
(+++) :: [a] -> [b] -> [Either a b]
[] +++ ys        =  map Right ys
(x:xs) +++ ys    =  Left x : (xs +++ ys)

(***) :: [a] -> [b] -> [(a,b)]
[] *** ys        = []
(x:xs) *** ys    = map f (ys +++ (xs *** ys))
  where
  f (Left y)     =  (x,y)
  f (Right p)    =  p

(^^^) :: [b] -> [a] -> [a -> b]
ys ^^^ []        =  [ nilFun ]
ys ^^^ (x:xs)    =  map g (ys *** (ys ^^^ xs))
  where
  g (y,e)        =  consFun (x,y) e
```