Clean Warmup – Session 3

Tamás Kozsik
kto@elte.hu
http://kto.web.elte.hu/

Dept. Programming Languages and Compilers
Eötvös Loránd University

CEFP 2009.

Sum up numbers

How to compute?

$x_1 + x_2 + \cdots + x_n$

```
module clean_session_3
import StdEnv

sum1 :: [Int] -> Int       // partial function
sum1 [x] = x
sum1 [x:xs] = x + (sum1 xs)
```

Similar functions

```
prod1 :: [Int] -> Int
prod1 [x] = x
prod1 [x:xs] = x * (prod1 xs)

max1 :: [Int] -> Int
max1 [x] = x
max1 [x:xs] = maximum x (max1 xs)
where maximum x y
        | x < y         = y
        | otherwise     = x

concat1 :: [String] -> String
concat1 [x] = x
concat1 [x:xs] = x +++ (concat1 xs)
```

General solution

The operator is a parameter

x_1 op x_2 op … op x_n

```
foldr1 :: (a->a->a) [a] -> a    // higher-order
foldr1 op [x] = x
foldr1 op [x:xs] = op x (foldr1 op xs)

sum1 = foldr1 (+)
prod1 = foldr1 (*)
concat1 = foldr1 (+++)
```

Partial application

```
sum1 = foldr1 (+)
prod1 = foldr1 (*)
concat1 = foldr1 (+++)

2 + 5       // type is: Int
(+) 2 5     // type is: Int
(+)         // type is: Int Int -> Int
(+) 2       // type is: Int -> Int
```

Exercise

Write the `mymap` function

Apply a function to each element of a list.

```
powersOfTwo = mymap ((^) 2) [0..]
Start = take 10 powersOfTwo
// [1,2,4,8,16,32,64,128,256,512]
```

```
foldr1 :: (a->a->a) [a] -> a
foldr1 op [x] = x
foldr1 op [x:xs] = op x (foldr1 op xs)
```

Solution

```
mymap :: (a->b) [a] -> [b]
mymap _ [] = []
mymap f [x:xs] = [f x : mymap f xs]
```

```
mymap :: (a->b) [a] -> [b]
mymap f xs = [ f x \\ x <- xs ]
```

Left or right?

Right: $x_1 + (x_2 + \cdots + x_n)$

```
foldr1 :: (a->a->a) [a] -> a
foldr1 op [x] = x
foldr1 op [x:xs] = op x (foldr1 op xs)
```

Left: $(x_1 + x_2) + \cdots + x_n$

```
foldl1 :: (a->a->a) [a] -> a
foldl1 op [x] = x
foldl1 op [x,y:ys] = foldl1 op [op x y: ys]
```

Empty sequence?

```
sum :: [Int] -> Int
sum [] = 0
sum [x:xs] = x + (sum xs)

prod :: [Int] -> Int
prod [] = 1
prod [x:xs] = x * (prod xs)

concat :: [String] -> String
concat [] = ""
concat [x:xs] = x +++ (concat xs)
```

First try...

Right: $x_1 + (x_2 + \cdots + x_n)$

```
foldr :: (a->a->a) a [a] -> a
foldr op r [] = r
foldr op r [x:xs] = op x (foldr op r xs)
```

Left: $(x_1 + x_2) + \cdots + x_n$

```
foldl :: (a->a->a) a [a] -> a
foldl op l [] = l
foldl op l [x:xs] = foldl op (op l x) xs
```

```
sum = foldr (+) 0
prod = foldr (*) 1
concat = foldr (+++) ""
```

Defining operators

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

flatten = foldr (++) []

(++) :: [a] [a] -> [a]
(++) [] bs = bs
(++) [a:as] bs = [a : as++bs]
```

Associativity and precedence of operators

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

flatten = foldr (++) []

(++) infixr 0 :: [a] [a] -> [a]
(++) [] bs = bs
(++) [a:as] bs = [a : as++bs]
```

Trickier application of foldr

```
(++) infixr 0 :: [a] [a] -> [a]
(++) [] bs = bs
(++) [a:as] bs = [a : as++bs]
```

More general type for `foldr`

```
foldr :: (a->b->b) b [a] -> b
foldr op r [] = r
foldr op r [x:xs] = op x (foldr op r xs)
```

```
(++) infixr 0 :: [a] [a] -> [a]
(++) as bs = foldr Cons bs as
   where Cons x xs = [x:xs]
```

Using a λ-function

```
(++) infixr 0 :: [a] [a] -> [a]
(++) [] bs = bs
(++) [a:as] bs = [a : as++bs]
```

More general type for `foldr`

```
foldr :: (a->b->b) b [a] -> b
foldr op r [] = r
foldr op r [x:xs] = op x (foldr op r xs)
```

```
(++) infixr 0 :: [a] [a] -> [a]
(++) as bs = foldr (\x xs = [x:xs]) bs as
```

Standard library functions

Right: $x_1 + (x_2 + \cdots + x_n)$

```
foldr :: (a->b->b) b [a] -> b
foldr op r [] = r
foldr op r [x:xs] = op x (foldr op r xs)
```

Left: $(x_1 + x_2) + \cdots + x_n$

```
foldl :: (a->b->a) a [b] -> a
foldl op l [] = l
foldl op l [x:xs] = foldl op (op l x) xs
```

Exercise

- Define the function composition operator `><`
  - Right associative with precedence of 9.
  - The goal is to have: `(f><g) x = f (g x).`
  - Use a λ-function!
- Define a list reversal function

Some help

```
(++) infixr 0 :: [a] [a] -> [a]
(++) as bs = foldr (\x xs = [x:xs]) bs as
```

Solution

```
(><) infixr 9:: (b -> c) (a -> b) a -> c
(><) f g x = f (g x)
```

```
(><) infixr 9:: (b -> c) (a -> b) -> (a -> c)
(><) f g = \x = f (g x)
```

```
(o) infixr 9:: (b -> c) (a -> b) -> (a -> c)
(o) f g = \x = f (g x)
```

```
revs = foldl (\xs x -> [x:xs]) []
```

Definition module

bag.dcl

```
definition module bag
import StdEnv

:: Bag :== [(String,Int)]

emptyBag     :: Bag
insertBag    :: Bag String -> Bag
multiplicity :: Bag String -> Int
removeBag    :: Bag String -> Bag
```

Implementation module

bag.icl

```
implementation module bag
import StdEnv

emptyBag :: Bag
emptyBag = []

insertBag :: Bag String -> Bag
insertBag [] item = [(item,1)]
insertBag [(key,mult):entries] item
 | key==item  = [(key,mult+1):entries]
              = [(key,mult): insertBag entries item]
```

As-patterns

bag.icl

```
implementation module bag
import StdEnv

emptyBag :: Bag
emptyBag = []

insertBag :: Bag String -> Bag
insertBag [] item = [(item,1)]
insertBag [entry=:(key,mult):entries] item
 | key==item   = [(key,mult+1):entries]
               = [entry: insertBag entries item]
```

Main module

use_bag.icl

```
module use_bag
import bag

Start = removeBag
          (insertBag
             (insertBag
                 emptyBag
                 "Rinus")
             "Rinus")
          "Marco"
```

Exercise

Define the type `Stack` in separate module

- Contains `Int`s
- Operations
  - `emptyStack`
  - `push`
  - `pop` (partial function)
  - `top` (partial function)
- Representation: list of ints

Solution

stack.dcl

```
definition module stack
import StdEnv

:: Stack :== [Int]

emptyStack :: Stack
push       :: Stack Int -> Stack
pop        :: Stack -> Stack
top        :: Stack -> Int
```

use_stack.icl

```
module use_stack
import stack
Start = top (push emptyStack 3)
```

stack.icl

```
implementation module stack
import StdEnv

emptyStack :: Stack
emptyStack = []

push :: Stack Int -> Stack
push stack item = [item:stack]

pop :: Stack -> Stack
pop stack = tl stack

top :: Stack -> Int
top stack = hd stack
```

Abstract type

bag.dcl

```
definition module bag
import StdEnv

:: Bag

emptyBag     :: Bag
insertBag    :: Bag String -> Bag
multiplicity :: Bag String -> Int
removeBag    :: Bag String -> Bag
```

Abstract types

bag.icl

```
implementation module bag
import StdEnv

:: Bag :== [(String,Int)]

insertBag :: Bag String -> Bag
insertBag [] item = [(item,1)]
insertBag [(key,mult):entries] item
 | key==item  = [(key,mult+1):entries]
              = [entry: insertBag entries item]
```

Abstract types

use_bag.icl

```
module use_bag
import bag

Start = removeBag
          (insertBag
              (insertBag
                   emptyBag
                   "Rinus")
              "Rinus")
          "Marco"
```

Exercise

Define the type `Stack` in separate module

- Contains `Int`s
- Operations
  - `emptyStack`
  - `push`
  - `pop` (partial function)
  - `top` (partial function)
- Representation: list with sp pointing to the head
- Make it abstract!

Solution

stack.dcl

```
definition module stack
import StdEnv

:: Stack

emptyStack :: Stack
push       :: Stack Int -> Stack
pop        :: Stack -> Stack
top        :: Stack -> Int
```

use_stack.icl

```
module use_stack
import stack
Start = top (push 3 emptyStack)
```

stack.icl

```
implementation module stack
import StdEnv

:: Stack :== [Int]

emptyStack :: Stack
emptyStack = []

push :: Stack Int -> Stack
push stack item = [item:stack]

pop :: Stack -> Stack
pop stack = tl stack

top :: Stack -> Int
top stack = hd stack
```

Parametric polymorphism

bag.dcl

```
definition module bag
import StdEnv
:: Bag a
emptyBag :: Bag a
```

bag.icl

```
implementation module bag
import StdEnv
:: Bag a :== [(a,Int)]
emptyBag :: Bag a
emptyBag = []
```

use_bag.icl

```
module use_bag
import bag
Start = removeBag (insertBag (insertBag emptyBag "Rinus") "Rinus") "Marco"
```

Bounded parametric polymorphism

bag.dcl

```
definition module bag
import StdEnv
:: Bag a
emptyBag      :: Bag a
insertBag     :: (Bag a) a -> (Bag a)    | == a
multiplicity :: (Bag a) a -> Int          | == a
removeBag     :: (Bag a) a -> (Bag a)    | == a
```

bag.icl

```
implementation module bag
import StdEnv
:: Bag a :== [(a,Int)]

emptyBag :: Bag a
emptyBag = []

insertBag :: (Bag a) a -> (Bag a) | == a
insertBag [] item = [(item,1)]
insertBag [(key,mult):entries] item
 | key==item   = [(key,mult+1):entries]
               = [entry: insertBag entries item]
```

Class declaration

```
insertBag :: (Bag a) a -> (Bag a) | == a
insertBag [] item = [(item,1)]
insertBag [(key,mult):entries] item
 | key==item  = [(key,mult+1):entries]
              = [entry: insertBag entries item]
```

Look at this

```
class (==) a :: a a -> Bool
```

Strictness annotations

```
class (==) infix 4 a :: !a !a -> Bool
```

Instance declaration

```
insertBag :: (Bag a) a -> (Bag a) | == a
insertBag [] item = [(item,1)]
insertBag [(key,mult):entries] item
 | key==item  = [(key,mult+1):entries]
              = [entry: insertBag entries item]
```

Class declaration

```
class (==) a :: a a -> Bool
```

Instance declaration

```
:: Bool = True | False
instance (==) Bool where
  (==) :: Bool Bool -> Bool
  (==) True True = True
  (==) False False = True
  (==) _ _ = False
```

This is real...

```
// Bool is built-in
instance (==) Bool where
  (==) :: !Bool !Bool -> Bool
  (==) a b    = code inline {
                    eqB
                }
```

```
Start = insertBag emptyBag True
```

Exercise

Define an instance == for type `Nat`

```
:: Nat = Zero | Succ Nat
```

```
class (==) a :: a a -> Bool
```

Little help

```
:: Bool = True | False
instance (==) Bool where
   (==) :: Bool Bool -> Bool
   (==) True True = True
   (==) False False = True
   (==) _ _ = False
```

Solution

```
:: Nat = Zero | Succ Nat

class (==) a :: a a -> Bool

instance (==) Nat where
   (==) :: Nat Nat -> Bool
   (==) Zero Zero = True
   (==) (Succ n) (Succ m) = (n==m)
   (==) _ _ = False
```

Overloading

```
class (==) a :: a a -> Bool

instance (==) Bool where
   (==) :: Bool Bool -> Bool
   (==) True True = True
   (==) False False = True
   (==) _ _ = False

instance (==) [a]    | (==) a
    where (==) :: [a] [a] -> [a]
          (==) [] [] = True
          (==) [x:xs] [y:ys] = (x==y) && (xs==ys)
          (==) _ _ = False

Start = insertBag emptyBag [[False,False],[True],[False,True]]
```

Encapsulation

```
class Arith a where
  (+) infix 6 :: a a -> a
  (-) infix 6  :: a a -> a
  (*) infix 7  :: a a -> a
  (/) infix 7  :: a a -> a

instance Arith Nat where
  (+) infix 6 :: Nat Nat -> Nat
  (+) Zero m = m
  (+) (Succ n) m = Succ (n + m)

  (*) infix 7 :: Nat Nat -> Nat
  (*) Zero _ = Zero
  (*) (Succ n) m = m + n*m

  // etc.
```

Inheritance

```
class (+) infix 6 :: a a -> a
class (-) infix 6  :: a a -> a
class (*) infix 7  :: a a -> a
class (/) infix 7  :: a a -> a
class Arith a    | +, -, *, / a

instance (+) Nat where
   (+) infix 6 :: Nat Nat -> Nat
   (+) Zero m = m
   (+) (Succ n) m = Succ (n + m)

// etc.
```

Default implementation

```
class (==) a :: a a -> Bool
/
class Eq a  | == a
  where (<>) :: a a -> Bool | Eq a
        (<>) x y = not (x==y)
```

Type constructor classes

```
class Functor f where
    fmap :: (a -> b) (f a) -> (f b)

instance Functor [] where
  fmap :: (a -> b) [a] -> [b]
  fmap f [x:xs] = [f x : fmap f xs]
  fmap f []     = []

:: Tree a = Node a (Tree a) (Tree a)
          | Leaf

instance Functor Tree where
  fmap :: (a -> b) (Tree a) -> (Tree b)
  fmap f (Node val left right)
            = Node (f val) (fmap f left) (fmap f right)
  fmap f Leaf  = Leaf
```

Different kind: * -> * -> *

```
:: Tree2 a b       = Tip a
                   | Bin b (Tree a b) (Tree a b)

class Bifunctor f where
  bmap :: (a1 -> b1) (a2 -> b2) (f a1 a2) -> (f b1 b2)

instance Bifunctor Tree2 where
  bmap :: (a1 -> b1) (a2 -> b2) (Tree2 a1 a2) -> (Tree2 b1 b2)
  bmap f1 f2 (Tip x)       = Tip (f1 x)
  bmap f1 f2 (Bin x l r) = Bin (f2 x) (bmap f1 f2 l) (bmap f1 f2 r)
```

Remember: kind * -> *

```
class Functor f where
    fmap :: (a -> b) (f a) -> (f b)

instance Functor [] where
  fmap :: (a -> b) [a] -> [b]
  fmap f [x:xs] = [f x : fmap f xs]
  fmap f []     = []

:: Tree a = Node a (Tree a) (Tree a) | Leaf

instance Functor Tree where
  fmap :: (a -> b) (Tree a) -> (Tree b)
  fmap f (Node val left right) = Node (f val) (fmap f left) (fmap f right)
  fmap f Leaf  = Leaf
```

Generic definition of map

```
generic gMap a b ::      a            -> b
gMap {|c|}          x                 = x
gMap {|PAIR|}       fx fy (PAIR x y)  = PAIR (fx x) (fy y)
gMap {|EITHER|}     fl fr (LEFT x)    = LEFT (fl x)
gMap {|EITHER|}     fl fr (RIGHT x)   = RIGHT (fr x)
gMap {|CONS|}       fx    (CONS x)    = CONS (fx x)
gMap {|FIELD|}      fx    (FIELD x)   = FIELD (fx x)
```

Classes that are automatically generated for the generic map function given above.

```
class gMap{|*|} t           :: t -> t
class gMap{|*->*|} t        :: (a -> b) (t a) -> t b
class gMap{|*->*->*|} t     :: (a1 -> b1) (a2 -> b2) (t a1 a2) -> t b1 b2
...
```

```
derive gMap [], Tree, Tree2
```

```
fmap :: (a -> b) (f a) -> (f b) | gMap{|*->*|} f
fmap f x y = gMap{|*->*|} f x y

bmap :: (a1 -> b1) (a2 -> b2) (f a1 a2) -> (f b1 b2) | gMap{|*->*->*|} f
bmap f1 f2 x y = gMap{|*->*->*|} f1 f2 x y
```