CS2104 Programming Languages

- Haskell
  - Hello world
  - Functions
  - Types
    - Primitive types
    - Algebraic types
  - Type conversion
  - Finite and infinite lists
  - Local binder
  - Layout rule
  - List comprehension
    - Translations to HOF
  - Arrays
  - Functor
    - `Maybe` as a functor
    - `[]` as a functor
    - `IO` as a functor
    - Function as a functor
  - Applicatives `<*>`
  - Monads
    - List as monads
  - `>>=` vs `>>`
  - Return statement
  - Do-comprehension
    - Translation between `>>=` and `do`
    - Translation between `>>` and `do`
    - List comprehension and do comprehension
  - Monadic IO
- Prolog
  - Operators
    - Tests
    - Arithmetic
  - Facts
  - Query on facts

    - Horn clauses
    - Unification
    - List manipulation
    - Negation as failure
    - Cut operator
    - Finite constraint solver
        - Arithmetic constraints
        - Labelling
- OCaml
    - Types
    - Records
    - Class
        - Class inheritance
        - Class signature and type
        - Subtyping via row polymorphism
        - Virtual class and method
    - Modules
        - Module signature
        - Functors

Haskell

- Strongly-typed with polymorphism and generic types
- Higher-order functions
- Pure functions (output depends solely on the input; no side effects; no updates to global variables)
- Lazy (only evaluates what is needed)
- Expressive Type classes (Monads, Arrows, etc.)

Hello world

```
putStrLn "Hello World!"
putStrLn :: String -> IO()
```

A pure function which takes as input a `String`, and returns an `IO` monad.

Functions

```haskell
-- named function
let inc1 x = x + 1

-- lambda function
let inc2 = \ x -> x + 1

let inc3 = (+ 1)
```

All 3 lines above are different ways of writing a pure function which takes an integer argument `x`, and returns a new integer.

Types

Every expression has a type; use ascription (`e :: t`) to force an expression `e` to have a given type `t`

```haskell
let tuple = ("hello", 2, 3.5)
-- `:t tuple` will be inferred as ([Char], Num, Fractional)
```

- Type of `tuple` will be inferred without ascriptions (with the most general type)

```haskell
let tuple = ("hello", 2, 3.5) :: (String, Int, Double)
```

- Type can be forced to a more specific type by using ascriptions

---

The `void` type is just the empty tuple `()`.

```haskell
:t ()
-- returns () :: ()
```

Primitive types

Primitive types in Haskell are **boxed types**. Unboxed types are built-in but seldom used (except when implementing the compiler).

> Boxed/unboxed types are identical to object/primitive types in Java

Advantages of boxed types:

- Supports polymorphism
- Supports lazy evaluation

Algebraic types

A more systematic way to capture union types:

```
let data Data = I Int | F Float | S String deriving Show

-- then, we can build objects of different values:
let v1 = I 3;
let v2 = F 4.5;
let v3 = S "CS2104";
```

- The `deriving` keyword can be thought of like the `extends` keyword in Java, where the type `Data` will also inherit the methods in the type `Show` (which is used for printing to stdout)

Type conversion

```
fact :: Int -> Integer
fact 0 = 1
fact n = (fromIntegral n) * fact(n - 1)
```

- The `fromIntegral` is necessary to convert `n` of type `Int` to `Integer`

Finite and infinite lists

The following will all return an infinite list of `1`s:

```
xs :: [Int]
-- using recursive binding:
xs = 1:xs
-- using list range notation:
xs = [1,1..]
-- using list comprehension:
xs = [ys | ys <- [1,1..]]
```

Local binder

```
foo =
  let greeting = "hello" in
    print (greeting ++ " world")
-- prints "hello world"
```

- `let` is similar to javascript's `let`, and binds the variable to the current local block scope `foo`

- `in` goes along with `let` to name one or more local expressions in a pure function, and is what is returned to the caller

```
let quad x =
  let add x y = x + y
    double x = add x x
  in (double x) + (double x)
```

- `add` and `double` are local functions
- `quad` is a global declaration

What are "binders" in programming language?

- A way to attach *identifiers* to *entities*
  - Identifiers: the name we give the entity
  - Entity: data, objects, operations, functions, etc.

Why do you need binders?

- Organise identifiers/entities into scopes and to group computations
- Basically to give structure to code

How are binders normally implemented?

- In the lexing phase, lexemes are identified based on patterns

Layout rule

- Indentations matter (like Python/Ruby)
- `<space>` is an infix operator that is left associative (with high precedence)
  - Use brackets, or the `$` symbol to override to prefix behaviour

```
a b c
-- means ((a b) c)
foo n + 1
-- actually means (foo n) + 1
foo $ n + 1
-- now means foo (n + 1)
foo(n + 1)
-- also means foo (n + 1)
```

What is the "layout rule" syntactic mechanism in programming language?

- The use of indentation/whitespaces to construct a language into logical blocks

Why is this called a two-dimentional syntax mechanism?

- Both *rows* and *columns* are taken into consideration

Why would you want to design layout rule mechamism for a new PL?

- Do away with the need for `{}` and `;`
- Reduce verbosity and cleaner syntax

How are layout rules normally implemented?

- **Lexers**: classify symbols (lexemes) into meaningful tokens
- **Parsers**: classify tokens into an AST

What other PL uses layout rule mechanism? Can you compare and contrast their features?

Haskell:

- Same indentation: starts a new clause in the same block
- Greater indentation: continues a current clause
- Less indentation: start of a new block

List comprehension

Similar to set comprehension in Mathematics.

For a set that contains the first ten even natural numbers:

$$S = \{2 \cdot x | x \in \mathbb{N}, x \leq 10\}$$

```
[x * 2 | x <- [1..10]]
>> [2, 4, 6, ..., 18, 20]

-- similar to map
map (\ x -> x * 2) [1..10]
>> [2, 4, 6, ..., 18, 20]
```

Using filters:

```
usingFilter xs = [ if x < 10 then x else x * 10 | x <- xs, odd x]
usingFilter [1..20]
>> [1,3,5,7,9,110,130,150,170,190]
```

- `x <- xs` is the list generator
- `odd x` is the filter

- `if x < 10 then x else x * 10` is then mapped to each element in the list

Translations to HOF

```
[f x | x <- xs]
map (\ x -> f x) xs
------------------
[f x | x <- xs, x > 5]
map (\ x -> f x) (filter (\ x -> x > 5) xs)
------------------
-- concatMap is similar to JS flatmap
[(x, y) | x <- xs, y <- ys]
concatMap (\ x -> map (\ y -> (x, y)) ys) xs
```

Arrays

```
:t array
>> array :: Ix i => (i, i) -> [(i, e)] -> Array i e
```

- `(i, i)`: the bounds of the array give in `(low, high)` inclusive
- `(i, e)`: where `i` is the index, and `e` the element at that index

```
-- generate an array of 2s of length 5:
array (1,5) [(i,2) | i <- [1..5]]
>> array (1,5) [(1,2),(2,2),(3,2),(4,2),(5,2)]

-- access at index (follows value of `i` above):
arr ! 5
>> 2
```

Functor

A functor is **a container of any generic type `A`** that, when subjected to a function that **maps from `A` to any generic type `B`**, **yields the same container of type `B`**.

In haskell, a `Functor` is a typeclass, and any `Functor` type needs to have a concrete implementation of `fmap`. In JavaScript, a `Functor` is any object that has a concrete implementation of the `map` function. For example, a JavaScript array is a `Functor` because the `Array` class has a well defined `map` function.

```
-- when executing `:i Functor`
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

Some types of common functors in Haskell include:

- `Maybe`
- `[]`
- `IO`
- Any function

`Maybe` as a functor

```
-- in the source code for the Maybe type:
instance  Functor Maybe  where
  fmap _ Nothing       = Nothing
  fmap f (Just a)      = Just (f a)

fmap (+1) (Just 3)  -- Maybe is a functor
>> Just 4
```

`[]` as a functor

```
instance Functor [] where
  fmap = map

map (+1) [1, 2, 3]
>> [2, 3, 4]

fmap (+1) [1, 2, 3] -- [] is a functor
>> [2, 3, 4]

map (+1) (Just 3)   -- Maybe is not a list
>> Error!
```

Essentially, `map` is just a specialised version of `fmap` which only works on lists.

`IO` as a functor

```
-- in the source code for the IO type:
instance  Functor IO where
  fmap f x = x >>= (pure . f)
```

Function as a functor

```haskell
-- in the source code for the ((->) r) type:
instance Functor ((->) r) where
  fmap = (.)

plusFive = fmap (+3) (+2) -- similar to `(.) (+3) (+2)`
plusFive 1
>> 6
```

`fmap` of 2 functions is a composition of those 2 functions

Applicatives `<*>`

The sequential application `<*>` takes a wrapped function and a wrapped value. The output is the result of unwrapping the function and value, applying the function to the value, and then wrapping the result.

```haskell
(<*>) :: f (a -> b) -> f a -> f b

fmap (Just (+2)) (Just 5)
>> Error!
(Just (+2)) <*> (Just 5)
>> Just 7
```

Monads

Monads apply a *function which returns a wrapped value* to a *wrapped value*. It is essentially the functional way of achieving polymorphism, because the wrapped type of the function can be different from the wrapped type of the input value.

In JavaScript, a **Monad is a Functor** which *also* implements `flatMap` (in addition to implementing `map`).

Implementation:

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

-- laws of Monad class
(return a) >>= k = k a
m >>= return = m
-- associativity (order of application does not matter for monads)
-- k and h are monads
(m >>= (\a -> (k a) >>= (\b -> h b)) = m >>= (\a -> k a) >>= (\b -> h b)
```

Suppose `half` is a function that only works on even numbers:

```
-- half returns a wrapped value (a Maybe type to be exact)
half x = if even x then
           Just (div x 2)
         else
           Nothing
```

If we pass a wrapped value:

```
half (Just 4)
>> Error!
```

We can use the bind operator `>>=` to apply `half` to the wrapped value `Just 4`:

```
Just 4 >>= half
>> Just 2
Just 5 >>= half
>> Nothing
```

We can also chain it (like UNIX pipes):

```
Just 20 >>= half >>= half >>= half
>> Nothing
```

List as monads

Using the bind operator `>>=` on a list `xs` to a function `f` is equivalent to using a `concatMap` (or `map` followed by `concat`):

```
(xs >>= f) = (concatMap f xs) = (concat (map f xs))

list_of_list = [[1], [2], [3]]
return_self = \xs -> xs

concatMap return_self list_of_list
>> [1, 2, 3]
list_of_list >>= return_self
>> [1, 2, 3]
```

`>>=` vs `>>`

```haskell
-- types: in ghci `:i >>=` and `:i >>`
(>>=) :: m a -> (a -> m b) -> m b
(>>) :: m a -> m b -> m b
```

- `m >>= k`: run `m`, and then run function `k` on the result of `m`
- `m >> n`: run `m`, and then run `n`, ignoring the result of `m`
- `a >> b` is equivalent to `a >>= \_ -> b`

Return statement

In haskell, `return` is an example of **type inference**. Since Haskell is lazily evaluated, the value of an expression and hence **its type isn't computed** until it really needs to be. This `return` is **totally different** from the `return` statements in other languages like C/Java/JS.

`return`: function which takes an argument and returns a `Monad` with that type in it.

```haskell
-- in this inc function, it takes a number `a` and returns a Monad `m a`
inc :: (Monad m, Num a) => a -> m a
inc num = return (num + 1)

-- since the list `[1,2,3]` is a Monad, and `inc` is a function which returns a Monad:
[1,2,3] >>= inc
>> [2,3,4]

-- since `Just 5` is a Monad, and `inc` is a function which returns a Monad:
Just 5 >>= inc
>> Just 6
```

Do-comprehension

Translation between `>>=` and `do`

```haskell
-- `getChar` and `putChar` are both I/O operations in Haskell
echo :: IO ()
echo = do c <- getChar
          putChar c

-- similar to using `<<=`:
echo :: IO ()
echo = getChar >>= (\c -> putChar c)
```

1. User character input will be stored in `c`
2. Then variable `c` is printed to I/O

For a list (using previous example):

```
-- this bind operation:
[[1], [2], [3]] >>= \xs -> xs

-- can be translated to:
flattened_list = do
  xs <- [[1], [2], [3]]
  xs

-- both results in:
[1, 2, 3]
```

Translation between `>>` and `do`

```
printHello :: IO ()
printHello = do putStr "Hello, "
                putStrLn "world!"

-- is similar to using `>>`:
printHello = putStr "Hello, " >> putStrLn "world!"

-- is also similar to using `>>=`:
printHello = putStr "Hello, " >>= (\_ -> putStrLn "world!")
```

List comprehension and do comprehension

- List is an instance of monad
- List comprehension is an instance of do-comprehension

This list comprehension:

```
[(x,y) | x <- xs, test x, y <- ys]
```

is similar to this do-comprehension:

```
filter test = \ x -> if test x then return a else empty

do
  x <- xs
  filter test
  y <- ys
  return (a, b)
```

Monadic IO

```
getLine :: IO String
getLine = do c <- getChar
             if c == '\n' then
               return ""
             else
               do l <- getLine
                  return (c:l)
```

Prolog

- **Untyped**
- **Atoms**: start with lower-case letters are constants
  - eg. `cat`, `john` , `5`, `-1`
- **Variables**: start with upper-case letter or underscore
  - eg. `X`, `Y2`, `_var`, `_1`
- **Compound term**: composed of an atom called a "functor" and a number of "arguments", which are again terms

Operators

Tests

- `atom(X)`: succeed if `X` is an atom (or an empty list)
- `atomic(X)`: succeed if `X` is an atom or number
- `number(X)`: succeed if `X` is a number
- `integer(X)`: succeed if `X` is an integer
- `float(X)`: succeed if `X` is a real number
- `var(X)`: succeed if `X` is unbound (a non-instantiated variable)
- `nonvar(X)`: succeed if `X` is bound
- `X == Y`: succeed if `X` and `Y` are identical (but do not unify them)
- `X \== Y`: succeed if `X` and `Y` are not identical

Arithmetic

- `X is E`: evaluate `E` and unify the result with `X`
- `X + Y`: when evaluated, yields the sum of `X` and `Y`
- `X - Y`: when evaluated, yields the difference of `X` and `Y`
- `X * Y`: when evaluated, yields the product of `X` and `Y`

- `X / Y`: when evaluated, yields the quotient of `X` and `Y`
- `X mod Y`: when evaluated, yields the remainder of `X` divided by `Y`
- `X =:= Y`: evaluate `X` and `Y` and compare them for equality
- `X =\= Y`: evaluate `X` and `Y` and succeed if they are not equal

Facts

We can provide facts as relations. Ends with a period `.` like so:

- `male(john).`: a fact which says that *John is male*
- `father(john, mark).`: a fact which says that *Mark's father is John*

Query on facts

- `?- father(X, mark).`: *who is the father of Mark?*

Horn clauses

`pred(...) :- pred1(), pred2(), ... predn().` has the logical meaning of `pred1() AND pred2() AND ... AND predn()` implies `pred(...)`.

- `sibling(X,Y) :- parent(Z,X), parent(Z,Y), X\==Y.`: `X` and `Y` are siblings (where they are not the same person)

Also supports recursive clauses:

- `ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).`

Unification

Computing a substitution for the *variables* so that two *terms* can be made equal:

- `a = X`: success, `X = a`
- `n(a, Y) = n(X, p(a))`: success, `X = a` and `Y = p(a)`
- `a = b`: fail, both `a` and `b` are atoms, not variables
- `n(a, X) = n(X, p(a))`: fail, `X` cannot be two different things simultaneously

List manipulation

Lists are denoted by square bracket `[]` with its elements separated by comma `,`: `[mary, [], n(A), [1,2,3], X]`.

The **append** relation can be executed in different ways:

- Appending lists: `?- append([1,2,3], [4,5], Z).` returns `Z = [1,2,3,4,5]`
- Computing the difference: `?- append([1,2,3], Y, [1,2,3,4,5]).` returns `Y = [4,5]`
- Splitting a list: `?- append(X,Y,[1,2]).` returns `X=[], Y=[1,2]; X=[1], Y=[2]; X=[1,2], Y=[].`

Negation as failure

- Only something which **can be proven is true**
- Anything which **cannot be proven is assumed to be false**

Thus, we can use this clause: `female(X) :- not(male(X)).`, which says if a person cannot be proven to be male, we shall assume the person is female.

Warning: negation as failure is sound only if the given clauses are complete.

```
# Assume we have the following rules:
male(kevin).
male(tom).
human(mary).
female(X) :- not(male(X)).

# query for all males:
?- male(X).
X = kevin ;
X = tom.

# query for all females:
?- female(X).
false.

# query if mary females:
?- female(mary).
true.
```

- When querying if `mary` is `female`, we cannot prove that `mary` is a male, and as such `male(mary)` is assumed to be `false.`
- However querying for `female(X)` will not work because `X` must be known

Cut operator

Similar to an `if-then-else` statement.

```
pred :- X1, !, X2, X3, X4.
pred :- Y1, Y2.
```

- If `X1` is true, then proceed with checking `X2, X3, X4` only
- If `X1` is false, backtrack to next statement `Y1, Y2`

Finite constraint solver

Via the `clpfd` library, imported using `:- use_module(library(clpfd)).` at the top of a prolog file.

Arithmetic constraints

- `Expr1 #+ Expr2`: `Expr1` equals `Expr2`
- `Expr1 #\= Expr2`: `Expr1` is not equal to `Expr2`
- `Expr1 #>= Expr2`: `Expr1` is greater than or equal to `Expr2`
- `Expr1 #=< Expr2`: `Expr1` is less than or equal to `Expr2`
- `Expr1 #> Expr2`: `Expr1` is greater than `Expr2`
- `Expr1 #< Expr2`: `Expr1` is less than `Expr2`

In contrast to the primitive operators like `in` and `=:=`, the constraints can be **used in any direction**, and **do not require the `Expr` expressions to be instantiated**.

Labelling

- Purpose is to enumerate and try all possible values (instead of just giving range)

```
range(X) :-
    X#>0,
    X#<100.

?- range(X). -- gives the constraint only:
X in 1..99.

?- range(X), label([X]). -- enumerate all possible values of X:
X = 1 ;
X = 2 ;
...
X = 98 ;
X = 99 ;
```

OCaml

Types

- **`unit`**: consists of exactly one value, written as `()`; used to represent type of expressions that return no value (ie. are evaluated for side-effects only)

- `ref`: akin to memory location (C-style pointer); dereference the `ref` using the prefix `!` operator

Records

User defined type, similar to `struct` in C.

```
(* record type must first be declared: *)
type account = {
  first_name: string;
  last_name: string;
  mutable balance: int;
} ;;

(* then initialise a variable with the record type: *)
let acc = {
  first_name = "John";
  last_name = "Doe";
  balance = 5
} ;;

(* access the variables using dot notation: *)
Printf.printf "Name: %s %s\n" acc.first_name acc.last_name ;; (* Name: John Doe *)
Printf.printf "Balance: %d\n" acc.balance ;; (* Balance: 5 *)

(* mutate a mutable variable: *)
acc.balance <- acc.balance * 2 ;;
Printf.printf "Balance: %d\n" acc.balance ;; (* Balance: 10 *)
```

Class

```
class counter init =
  object (name)
    val mutable x = init
    method inc = x <- x + 1
    method get = x
    method set y = x <- y
    method print = () = Printf.printf "Count: %d" (name # get)
  end
;;
```

- `counter` is a factory of the object
- `init` is the constructor parameter
- `name` is the explicit name of the current object; analogous to `this` in Java
- All fields are immutable by default (use `mutable` to make them mutable)

- All fields (like `x`) are private and not accessible, and must be called through the `get` / `set` methods
- Setting an object variable is similar to setting a record variable

```
let p = new counter 0 ;;
let q = new counter 5 ;;

p # print ;; (* Count: 0 *)
p # inc ;;
p # inc ;;
p # print ;; (* Count: 2 *)

q # print ;; (* Count: 5 *)
q # set 10 ;;
q # print ;; (* Count: 10 *)
```

Class inheritance

Inherit fields and methods of superclass, and support method overriding.

```
class counter_step init step =
  object (s)
    inherit counter init
    method inc = x <- x + step (* override counter's inc method *)
  end
;;

let r = new counter_step 0 4 ;;

r # print ;; (* Count: 0 *)
r # inc ;;
r # inc ;;
r # print ;; (* Count: 8 *)
```

Class signature and type

Based on structure of the **set of visible methods** (since fields are always private). Thus, using `counter` example above:

```
counter : <get : 'a; inc : unit; set : 'a -> unit; print : unit>
```

- `'a` refers to a generic type
- Two classes are of the same type if they are structurally equivalent to each other based on their visible methods

Subtyping via row polymorphism

```
foo : < get : 'b; .. > -> 'b
```

- `foo` will unify with any type with a `get` method in its class type
- The matched type need not be structurally equivalent

Virtual class and method

Similar to an `abstract` class, a class can be defined with some undefined/`virtual` methods:

```
class virtual ['a] buffer_eq init =
  object (this)
    val mutable value :'a = init
    method get = value
    method virtual eq : 'a buffer_eq -> bool
    method neq b = not(this # eq b)
  end
;;

class ['a] buffer init =
  object (this)
    inherit ['a] buffer_eq init
    (* concrete classes must give concrete implementation of eq: *)
    method eq that = this # get = that # get
    method set n = value <- n
  end
;;

let b = new buffer 5    (* ok *)
let c = new buffer_eq 5 (* error; virtual classes cannot be instantiated *)
```

- Objects of virtual classes **cannot be instantiated**
- Concrete subclasses should give definitions for the virtual methods

Modules

Used to group *types*, *values*, *functions*, *exceptions*, and other *modules* together.

An example of a functional stack implemented using OCaml lists:

```
module FunctionalStack = struct
  type number = Int of int | Float of float
  let init () : number list = []
  let is_empty xs = (xs = [])
  let push xs x = x :: xs
  let peek = function
    | [] -> failwith "Empty"
    | x::_ -> x
  let pop = function
    | [] -> failwith "Empty"
    | _::xs -> xs
  let number_to_string = function
    | Int x -> string_of_int x
    | Float x -> Float.to_string x
  let to_string xs = String.concat ", " (List.map number_to_string xs)
  let print xs = Printf.printf "[%s]" (to_string xs)
end ;;
```

Use dot notation to access the definitions within the module:

```
let stack = FunctionalStack.init () ;;
let stack = FunctionalStack.push stack (Int 1) ;;
let stack = FunctionalStack.push stack (Float 4.5) ;;
let stack = FunctionalStack.push stack (Int 9) ;;
FunctionalStack.print stack ;; (* [9, 4.5, 1] *)
```

Use `open` to expose the definitions within the module (ie. no need to use dot notation):

```
open FunctionalStack ;;
let stack = init () ;;
let stack = push stack (Int 1) ;;
let stack = push stack (Float 4.5) ;;
let stack = push stack (Int 9) ;;
print stack ;; (* [9, 4.5, 1] *)
```

Module signature

Each module has a type signature that may be explicitly declared. For example, we can create the following type signature:

```
module type STACK = sig
  type number = Int of int | Float of float
  val init : unit -> number list
  val is_empty : number list -> bool
  val push : number list -> number -> number list
  val peek : number list -> number
  val pop : number list -> number list
  val print : number list -> unit
end
```

And associate the original `FunctionalStack` with the `STACK` signature like so:

```
module FunctionalStack : STACK = struct
  (* ... same code as above ... *)
end
```

Notice that `STACK` does not have definitions for `number_to_string` and `to_string`. As such, **information hiding** can be implemented (ie. make some fields **private and not callable**).

```
open FunctionalStack ;;
let stack = init () ;;
number_to_string (Int 5) ;; (* error: unbound value number_to_string *)
to_string stack ;;          (* error: unbound value to_string *)
print stack ;;              (* ok! STACK has definition for print *)
```

Functors

A function from structures to structures. It is OCaml's way of creating parametrised structures.

As an example, we first create a module type `NUMBER_TYPE`:

```
module type NUMBER_TYPE = sig
  type t
  val value : t ref
  val inc : unit -> unit
  val print : unit -> unit
end ;;
```

Then, we define the `Number` module, a functor which takes as input another structure of type `NUMBER_TYPE`:

```
module Number =
  functor (Elt: NUMBER_TYPE) ->
  struct
    type t = Elt.t
    let value : t ref = Elt.value
    let inc = Elt.inc
    let print = Elt.print
  end
;;
```

Then, define the `Integer` and `Float` modules:

```
module Integer : NUMBER_TYPE = struct
  type t = int
  let value : t ref = ref 0
  let inc () = value := !value + 1
  let print () = Printf.printf "value: %s\n" (string_of_int !value)
end ;;

module Float : NUMBER_TYPE = struct
  type t = float
  let value : t ref = ref 0.0
  let inc () = value := Float.add !value 0.1
  let print () = Printf.printf "value: %s\n" (Float.to_string !value)
end ;;
```

Since `Number` is a functor, we can create specific modules like so:

```
module INumber = Number(Integer) ;;
module FNumber = Number(Float) ;;

(* usage: *)
INumber.print () ;; (* value: 0 *)
FNumber.print () ;; (* value: 0. *)

INumber.inc () ;;
INumber.inc () ;;
FNumber.inc () ;;

INumber.print () ;; (* value: 2 *)
FNumber.print () ;; (* value: 0.1 *)
```