Types in CLEAN

CLEAN

Góbi Attila

2009. május 21.

Outline

Type aliases

```
:: Name :== String
```

- Documentary nature
  $\longrightarrow$ improves readablity
- Like typedef in C++

Excercise – Synonyms

Write a new `icl` file, containing

- type synonyms `Length` for integer! Create a Start function to test it.

Solution – Synonyms

```
:: Length :== Int
```

Tuples

```
:: Point2D :== (Int,Int)
:: Point3D :== (Int,Int,Int)
```

Using tuples

```
mirror :: Point2D -> Point2D
mirror (x,y) = (y, -x)
```

Tuples elements of different types

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

```
insertBag :: Bag String -> Bag
insertBag [] string = [ (string, 1) ]
insertBag [(item,mult):xs] string
    | item == string = [(item, mult+1):xs]
    | otherwise      = [(item,mult):(insertBat xs string)]
```

Excercise – Tuples

Write a new `icl` file, containing

- The previously declared Bag type.
- Write a function with the following signature:

  ```
  multiplicity :: Bag String -> Int
  ```

  returns the multiplicity of the given string or zero.
- A Start function to test it:

  ```
  Start = multiplicity [("a", 1), ("b",2)] "q"
  ```

Solution – Tuples

```
multiplicity :: Bag String -> Int
multiplicity []                _       = 0
multiplicity [(item,mult):xs] string
    | item == string = mult
    | otherwise      = multiplicity xs string
```

Example

Declaration

```
:: Point = { x :: Int
           , y :: Int
           }
```

Value

```
p1 :: Point
p1 = { x = 1
     , y = 1
     }
```

Referencing

```
x1 :: Int
x1 = p1.x
```

Comparison with tuple

With record

```
:: Point = { x :: Int
           , y :: Int
           }

norm1 :: Point -> Int
norm1 point = point.x + point.y
```

With tuple

```
:: Point :== (Int, Int)

norm1 :: Point -> Int
norm1 (x,y) = x + y
```

Records can be ‘updated’

```
projectX :: Point -> Point
projectX point = { point & x = 0 }
```

- No need to refer to other fields’ values
- Makes the usage of records flexible (fields can be added later without having to rewrite all functions)

Exercise – Records

Define new records for pieces of different shapes:

- `Square`
- and `Circle`

Use type `Length`!

Solution

```
:: Square = { side :: Length }
:: Circle = { radius :: Length }
```

Example – Direction

```
:: Direction = Left | Right

mirror :: Direction -> Direction
mirror Left = Right
mirror Right = Left
```

Example – Point

```
:: Point = Point Int Int

Point :: Point -> Int
norm1 Point x y = x + y
```

With tuple

```
:: Point :== (Int, Int)

norm1 :: Point -> Int
norm1 (x,y) = x + y
```

Example – Nat

```
:: Nat = Succ Nat | Zero

One = Succ Zero
Two = Succ One

pred :: Nat -> Nat
pred (Succ n) = n

double :: Nat -> Nat
// 2*(1+n) = 1+1+(2*n)
double (Succ n) = Succ (Succ (double n))
double Zero     = 0
```

Composite matching

```
:: Nat = Succ Nat | Zero

even :: Nat -> Bool
even Zero                = True
even (Succ Zero)         = False
even (Succ (Succ n))     = even n
```

Example – BiNat

```
:: BiNat = One | Double BiNat |
    DoublePlusOne BiNat
six = Double ( DoublePlusOne One )

succ One                  = Double One
succ (Double n)           = DoublePlusOne n
// 1 + (2n+1) = 2*(n+1)
succ (DoublePlusOne n) = Double (succ n)
```

Parametric

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

height :: Tree a -> Nat
height Leaf       = 1
height Node l r = max (height l) (height r)
```

Notation:

- Type constructor: `Tree`
- Data constructor: `Node`

Preorder traversal

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

preorder :: Tree a -> [a]
preorder Leaf       = []
preorder Node l r = [a] ++ (preorder l)
                        ++ (preorder r)
```

Algebraic Types – Excercise

- Write an algebraic type `Color` which contains 4 different colours!
- Write a function that can calculate the number of the leaves in a given tree!
- Write a postorder traversal on the type Tree and save it a list.

Algebraic Types – Solution

```
:: Color = Black | White | Red | Yellow

length :: Tree a -> Int
length Leaf       = 1
length Node l r = (length l) + (length r)
```

Algebraic Types – Solution (cont)

```
preorder :: Tree a -> [a]
preorder Leaf       = []
preorder Node l r = (preorder l) ++
                    (preorder r) ++ [a]
```

Benefits of unique types

- Normally destructive updates can't be done in Clean, because it violates referential transparency.
- Sometimes a destructive update is needed:
  - user interactions
  - writing files or on screen
  - etc.
- We can't refer to a uniqness type twice, so we can safely update it.
- When a unique variable is updated, it is destroyed and a new one created.
- Memory can be reused.

Input/Output

When we read a file it has a side-effect of changing the current position in the file. The next read from the file will return different value, so it violates referential transparency.
If the read function returns a new file (with changed current position) we can make the original file unique.

```
write12
  # f  = open("test.txt")
  # f2 = fwritec 'a' f
  = 1
```

Arrays

- Contiguous blocks in memory
- Increases efficiency (constant time element selection)
- Increases risk of run-time errors

Lazy vs. Strict Arrays

- Lazy: elements are evaluated only when directly referred at

```
:: LazyArray a :== {a}
```

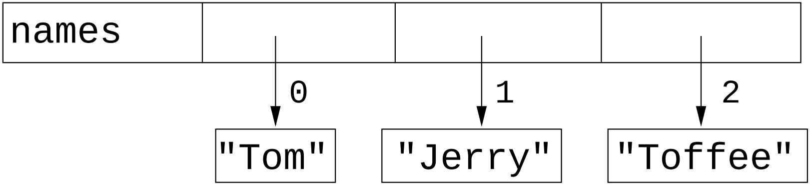
- Strict: elements are evaluated when the array is being referred at

```
:: StrictArray a :== {!a}
```

Boxed arrays

Only pointers to elements are stored in array itself

```
:: BoxedArray a :== {a}
names :: BoxedArray
names = {"Tom", "Jerry", "Toffee"}
```



Unboxed Arrays

Elements are stored directed in array itself

```
:: UnboxedArray a :== {#a}
numbers :: UnboxedArray
numbers = {1, 2, 3, 5, 7, 11}
```

numbers	1	2	3	5	7	11

Array elements

Fields can be referred at directly by their indices (first index is 0):

```
thirdElement array = array.[2]
```

Array comprehension

```
{ element \\ element <-: array }
```

Unique arrays can be updated:

```
uniqueArray :: *{Int} -> *{Int}
uniqueArray x = { x & [4] = 3, [3] = 4 }
```

Excercise – Arrays

- Create an array `EmptyArray`, the size of 6 and contains only zeros!
- Write a function, `addAmount`, which takes an array, the amount of paint needed, and index, that indicates which colour's amount should be modified and updates the array respectively!

Solution – Arrays

```
EmptyArray = { 0.0 \\ i <- [0..NumOfColors-1] }

addAmount array amount index
    # (value, array) = uselect array index
    = { array & [index] = value + amount }

// A bad solution. Two references to array
addAmount array amount index
    = { array & [index] = amount + array.[index] }
```

Declaration

The type's interface is placed in a separate definition module; user of the type sees only this module

stack.dcl

```
definition module stack

:: Stack a

Push  :: a (Stack a) -> Stack a
Pop   :: (Stack a) -> Stack a
top   :: (Stack a) -> a
Empty :: Stack a
```

Implementation

Representation of type and implementation of functions is hidden from user of the module (hence the name abstract)

stack.icl

```
implementation module stack

:: Stack a :== [a]

Push    e   s   = [ e : s ]
Pop   [ e : s ] =   s
top   [ e : s ] =   e
Empty           = []
```

Usage

Representation and implementation can be changed without affecting the modules using this type

```
module stack_user

import stack

Start = top (Push 1 Empty)
```

Abstract Types – Excercise

- Define and implement an abstract type `Queue` for queueing any types (a queue is a first in first out data-structure)! It has to have the following functions:
  - `EmptyQueue`
  - `insertItem`
  - `getFirstItem`
  - `removeFirstItem`
- Write a `Start` expression to test the new queue!

Abstract Types – Solution

```
definition module queue

::Queue a

EmptyQueue :: Queue a
insertItem :: (Queue a) a -> (Queue a)
getFirstItem :: (Queue a) -> a
removeFirstItem :: (Queue a) -> (Queue a)
```

Abstract Types – Solution

```
implementation module queue

import StdList

:: Queue a :== [a]

EmptyQueue = []
insertItem queue item = queue ++ [item]
getFirstItem queue = hd queue
removeFirstItem queue = tl queue
```

Abstract Types – Solution (cont)

```
import queue

Start = ( getFirstItem
          ( insertItem
             ( insertItem
                ( insertItem EmptyQueue 8 )
             2 )
          3 )
        )
```

‘Ad hoc’ polymorphism: we need the same set of functions for different types, but implementation depends on type

pretty printing

```
class PrettyPrint a
where
    format :: a -> String
    concat :: a a -> String
    concat a1 a2 = (format a1) ++ ",
" ++ (format a2)
```

- We have to instantiate the class for all types we'd like to use it for
- Instantiation can differ from type to type

Instatiation for Int

```
instance PrettyPrint Int
where
    format :: Int -> String
    format i = "Integer: " ++ toChar i
```

Instantiation for Point

```
instance PrettyPrint Point
where
    format :: Point -> String
    format p = "Point: (x: " ++ (toChar p.x)
                   ++ ", y: " ++ (toChar p.y)
                   ++ ")"
```

- Member `concat` was derived from `format`
- Usage: user has to indicate that an instantiation of a particular class is needed

```
printList :: [a] -> String | PrettyPrint a
printList [ x : xs ] = (format x)
                       ++ printList xs
```

Classes – Excercise

- Write a type class, `Measures a` that has to functions:
  - `circumference`
  - `surface`!
- Instantiate the class to both types of pieces!
- Add a field `price` to each type of pieces! Does it affect the functions you've just implemented?

Classes – Solution

```
class Measures a
where
   circumference :: a -> Real
   surface :: a -> Real

instance Measures Square
where
   circumference square = fromInt (square.side * 4)
   surface square = fromInt (square.side * square.side)
```

Classes – Solution (cont)

```
instance Measures Circle
where
   circumference circle
      = fromInt (2 * circle.radius) * 3.14
   surface circle
      = 3.14 * fromInt (circle.radius * circle.radius)

:: Square = { side :: Length
            , s_color :: Color, s_price :: Int
            }

:: Circle = { radius :: Length
            , c_color :: Color, c_price :: Int
            }
```