CS558 Programming Languages

Fall 2015

Lecture 4b

CAPTURING ANOTHER PATTERN OF ABSTRACTION

Consider the following problems:

Sum a list of integers:

```
def sum (l:List[Int]) : Int = l match {
  case Nil => 0
  case h::t => h + sum(t)
  }
```

Multiply a list of integers:

```
def prod (l:List[Int]) : Int = l match {
  case Nil => 1
  case h::t => h * prod(t)
  }
```

The Pattern Continues...

Calculate the length of a list (of any type):

```
def len[A](l:List[A]) : Int = l match {
  case Nil => 0
  case _::t => 1 + len(t)
  }
```

Copy a list (of any type):

```
def copy[A](l:List[A]) : List[A] = l match {
  case Nil => Nil
  case h::t => h::copy(t)
  }
```

Query: How does `copy` differ from the identity function `(x => x)` ?

FOLDS

We can **abstract** over the common inductive pattern displayed by these examples:

```
def foldr[A,B] (c: (A,B) => B, n:B) (l:List[A]) : B = l match {
  case Nil => n
  case h::t => c (h,foldr(c,n)(t))
  }

val sum = foldr[Int,Int] ((x,y) => x+y,0) _
val prod = foldr[Int,Int] (_*_,1) _
def len[A] = foldr[A,Int] ((_,y) => 1+y,0) _
def copy[A] = foldr[A,List[A]] (_::_,Nil) _
```

Function `foldr` computes a value working from the tail of the list to the head (from **r**ight to left). Argument `n` is the value to return for the empty list. Argument `c` is the function to apply to each element and the previously computed result.

The `foldr` function is Curried to make it convenient to partially apply it.

FOLDS (2)

We can view `foldr (`c`,`n`) (`l`)` as replacing each `::` constructor in l with x and the `Nil` constructor with n. For example:

```
l = x1 :: (x2 :: (... :: (xn :: Nil)))
foldr (_+_,0) (l) =
    x1 + (x2 + (... (xn + 0)))
```

It is also possible to define a `foldl` that accumulates a value from the left; sometimes this will be more efficient.

In some languages, `fold` is called `reduce`, because we "reduce" a list of values to a single value. A similar ideas appears in "map-reduce" frameworks for organizing massively distributed computations.