Unit 7: Tuples

Antoni Diller

26 July 2011

Introduction

Some examples:

```
( "height", 7 ) :: ( [ Char ], Int )
( 10, 30, "time" ) :: ( Int, Int, [ Char ] )
( True, "eat", 8 ) :: ( Bool, [ Char ], Int )
```

Two useful functions defined on tuples are:

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y
```

Tuples can be used inside ZF-expressions:

```
[ (x, y) | x <- [1, 2, 3], y <- "st" ]
```

This evaluates to:

```
[ (1, 's'), (1, 't'), (2, 's'), (2, 't'), (3, 's'), (3, 't') ]
```

Tuples have many uses. One of them is to enable you to define functions that return tuples which contain several pieces of information. For example, a more efficient version of Quicksort can be defined by using the following `split` function:

```
split :: Ord a => a -> [a] -> ([a],[a])
split x [] = ([], [])
split x (y:ys)
  | y < x     = (y:less, greater)
  | otherwise = (less, y:greater)
                where (less, greater) = split x ys
```

Quicksort can now be defined as follows:

```
quick :: Ord a => [a] -> [a]
quick [] = []
quick [x] = [x]
quick (x:xs) = quick less ++ [x] ++ quick greater
               where (less, greater) = split x xs
```

The functions `zip` and `curry`

More information on `zip` can be found in section 4.4 of Bird, *Introduction to Functional Programming using Haskell* (1998). More information on `curry` can be found in chapter 1 of the same book.¡p¿ Some examples:

```
zip [7, 8, 2] "cat" = [(7, 'c'), (8, 'a'), (2, 't')]

zip [5, 1, 3, 4] "on" = [(5, 'o'), (1, 'n')]

unzip [(True, 8), (False, 3), (True, 11) ] =
([True, False, True], [8, 3, 11])
```

To illustrate the usefulness of `zip` consider the following problem: Calculate the scalar product of two vectors. In other words, define a function `sp` which does the following:

```
sp [x1, x2, ..., xn] [y1, y2, ..., yn] =
x1 * y1 + x2 * y2 + ... + xn * yn
```

I will present the solution in several stages. Step 1:

```
zip [x1, x2, ..., xn] [y1, y2, ... , yn] =
[(x1, y1), (x2, y2), ..., (xn, yn)]
```

Let `xs` be `[x1, x2, ..., xn]` Let `ys` be `[y1, y2, ..., yn]` and `times (x, y)` be `x * y`. Then step 2 is:

```
map times (zip xs ys) = [x1 * y1, x2 * y2, ..., xn * yn]
```

Step 3:

```
sum (map times (zip xs ys)) = x1 * y1 + x2 * y2 + ... + xn * yn
```

Thus, the solution is:

```
sp :: Num => [a] -> [a] -> [a]
sp xs ys = sum (map times (zip xs ys))
           where times (x, y) = x * y
```

As another example of the usefulness of `zip` consider the following problem: Decide whether or not a list is in non-decreasing order. In other words, define a function `nondec` which does the following:

```
nondec [x1, x2, ..., xn] =
x1 <= x2 && x2 <= x3 && ... && x(n-1) <= xn
```

I will present the solution in several stages. Step 1: Let `xs` be `[x1, x2, ..., xn]`. Then we have:

```
zip xs (tail xs) = [(x1, x2), (x2, x3), ..., (x(n-1), xn)]
```

Step 2: Define `leq (x, y) = x <= y`. Then we have:

```
map leq (zip xs (tail xs)) =
[x1 <= x2, x2 <= x3, ... , x(n-1) <= xn]
```

Step 3:

```
and (map leq (zip xs (tail xs)) =
x1 <= x2 && x2 <= x3 && ... && x(n-1) <= xn
```

So, the solution of the problem is as follows:

```
nondec :: Ord a => [a] -> Bool
nondec xs = and (map leq (zip xs (tail xs)))
            where leq (x, y) = x <= y
```

Currying

Consider the following two definitions:

```
minof2 :: Ord a => a -> a -> a
minof2 x y
  | x <= y    = x
  | otherwise = y

mint :: Ord a => (a, a) -> a
mint (x, y)
  | x <= y    = x
  | otherwise = y
```

The predefined functions `curry` and `uncurry` are defined as follows:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry g (x, y) = g x y
```

Using `curry` and `uncurry` the functions `minof2` and `mint` can be defined in terms of each other:

```
minof2 = curry mint

mint = uncurry minof2
```

In what follows it will be useful to have uncurried versions of the operators `*` and `<=`. These can be defined in several ways:

```
times (x, y) = x * y
times = uncurry (*)

leq (x, y) = x <= y
leq = uncurry (<=)
```

The function `zipWith` is defined as follows:

```
zipWith op xs ys = map (uncurry op) (zip xs ys)
```

To see why `zipWith` is useful, recall the definitions of the scalar product of two vectors `sp` and the function which determins whether or not a list of numbers is in non-decreasing order `nondec`:

```
sp xs ys = sum (map times (zip xs ys))
           where times (x, y) = x * y

nondec xs = and (map leq (zip xs (tail xs))
            where leq (x, y) = x <= y
```

Using `zipWith` these can be readily defined:

```
sp xs ys = sum (zipWith (*) xs ys)

nondec xs = and (zipWith (<=) xs (tail xs))
```

Using `zipWith` the memoised definition of the Fibonacci numbers can be made even more elegant:

```
fiblist = 0 : 1 : zipWith (+) fiblist (tail fiblist)
```