University of Auckland

415.360

Functional Programming

Christian Collberg

March 22, 1996

Function Examples

The `remdups` Function I

- Define a function to remove duplicate adjacent elements from a list:

```
remdups [1]           ⇒ [1]
remdups [1,2,1]       ⇒ [1,2,1]
remdups [1,2,1,1,2]   ⇒ [1,2,2]
remdups [1,1,1,2]     ⇒ [1,2,1]
```

- We have to consider three cases:
  1. The first two elements of the list are identical. Remove one of them, then remove duplicates from the rest of the list.
  2. The first two elements are different. Keep them and remove duplicates from the rest of the list.
  3. There are fewer that two elements in the list. Keep them.

The `remdups` Function II

Algorithm in English:

Case 1: Let the first two elements of the list be `x` and `y`. Let `x==y`. Example: `L==[1,1,2,3], x==y==1`. Discard `x`. Recursively remove duplicates from the remaining list `L=[1,2,3]`.

Case 2: The first two elements of the list `(x,y)` are different (`x/=y`). Example: `L==[1,2,2,3], x==1, y==2`. Append `x` onto the result of removing duplicates from the list `L'=[2,2,3]` from which `x` has been removed.

Case 3: The list has 0 or 1 element. Return it.

The `remdups` Function III

Simulation:

```
remdups [1,2,2] ⇒            ⇐ case 2,x=1,y=2,xs=[2]
  1:(remdups 2:[2]) ≡
  1:(remdups [2,2]) ⇒        ⇐ case 1,x=y=2,xs=[ ]
    1:(remdups 2:[ ]) ≡
    1:(remdups [2]) ⇒        ⇐ case 3,xs=[2]
      1:[2] ⇒
        [1,2]
```

The `remdups` Function IV

Algorithm in Gofer:

```
remdups ::  [Int] -> [Int]
remdups x:y:xs =
   if x == y then
      remdups y:xs       ⇐ case 1
   else
      x :  remdups y:xs ⇐ case 2
remdups xs = xs          ⇐ case 3
```

case 1: First two elements identical.

case 2: First two elements different.

case 3: Less than 2 elements left.

- Note that `x:y:xs` matches any list with 2 or more elements.

Slide 7–4

Gofer Context Predicates I

- Obviously `remdups` should work for any list, not just lists of `Ints`. Removing duplicates from a list of strings is no different from removing duplicates from a list of integers.
- However, there's a complication. In order to remove duplicates from a list, we must be able to compare list elements for equality.
- The polymorphic type

  ```
  [a] -> [a]
  ```

  is therefore a bit too general, since it would allow any type, even one for which equality is not defined.

Slide 7–5

Gofer Context Predicates II

- Gofer uses **context predicates** to restrict polymorphic types:

  ```
  remdups ::  Eq [a] => [a] -> [a]
  ```

  Now, `remdups` may only be applied to list of elements where the element type has `==` and `\=` defined.
- `Eq` is called a **type class**. `Ord` is another useful type class. It is used to restrict the polymorphic type of a function to types for which the relational operators (`<, <=, >, >=`) have been defined.

Slide 7–6

Gofer Guards I

- There's another bit of syntax that is useful when defining Gofer functions, called **guards**:

  ```
  fun_name fun_args
     | guard₁        = expr₁
     | guard₂        = expr₂
           ...
     | otherwise     = exprₙ
  ```

- This is equivalent to:

  ```
  fun_name fun_args
     if guard₁ then
        expr₁
     else if guard₂ then
        expr₂
     else if ···
     else exprₙ
  ```

Slide 7–7

Gofer Guards II

- The syntax in B&W is different:

```
fun_name fun_args
    = expr_1,          if guard_1
          ...
    = expr_n,          otherwise
```

fact with guards:

```
fact ::  Int -> Int
fact n
   | n==0      = 1
   | otherwise = n * fact (n-1)
```

remdups with guards:

```
remdups ::  Eq [a] => [a] -> [a]
remdups (x:y:xs)
   | x==y      = remdups (y:xs)
   | x /= y    = x :  remdups (y:xs)
remdups xs = xs
```

Slide 7–8

The append Function I

- We want to define a function that appends two lists together:

```
append [1,2] [3,4]⇒ [1,2,3,4]
append [ ] [1,2]  ⇒ [1,2]
```

  Only use `cons` (":") and recursion.
- Remember that `cons` creates a new list from an element `x` and a list `xs`, such that `x` is the first element and `xs` the last elements of the list:

```
5 :  [1,2]          ⇒ [5,1,2]
```

"Algorithm" for `append xs ys`:

1. Take `xs` apart and use `cons` to put the elements together to make a new list.
2. Again use `cons` to make `ys` the tail of this new list.

Slide 7–9

The append Function II

Simulation:

```
append [1,2,3] [4,5] ⇒
 1:  (append [2,3] [4,5]) ⇒
   1:  (2:  (append [3] [4,5])) ⇒
     1:  (2:  (3:  (append [ ] [4,5]))) ⇒
       1:  (2:  (3:  [4,5])) ⇒
     1:  (2:  [3,4,5]) ⇒
   1:  [2,3,4,5] ⇒
 [1,2,3,4,5]
```

Slide 7–10

The append Function III

Simulation:

```
append [1,2,3] [4,5] ⇒
  1:  (append [2,3] [4,5]) ⇒
    1:  (2:  (append [3] [4,5])) ⇒
      1:  (2:  (3:  (append [ ] [4,5]))) ⇒
        1:  (2:  (3:  [4,5])) ⇒
      1:  (2:  [3,4,5]) ⇒
    1:  [2,3,4,5] ⇒
  [1,2,3,4,5]
```

- Note how we take the first argument apart when going into the recursion, and how it is put together when returning back up.
- Notice also that the second argument to `append` is never traversed. It is simply "tacked on" (using `cons`) to the end of the new list when the bottom of the recursion has been reached.

Slide 7–11

The append Function IV

Algorithm in Gofer:

```
append ::  [a] -> [a] -> [a]
append [ ] xs = xs
append (x:xs) ys = x :  append xs ys
```

++ as append:

- It is more convenient to define `append` as an infix operator. `++` is predefined in the standard prelude.

```
infixr 5 ++
(++) ::  [a] -> [a] -> [a]
[ ]++xs = xs
(x:xs) ++ ys = x :  (xs ++ ys)
```

Slide 7–12

Local definitions I (a) – where

- In Pascal we can **nest** declarations, i.e. declarations can be made **local** to a particular procedure:

  ```
  function P (···) :  ···
     function X (···) :  ···
     ....
  begin ··· X(···) ··· end.
  ```

  The local function `X` can only be accessed from within `P`. This is an important way to break a complicated routine into manageable chunks. We also **hide** the definition of `X` from routines other than `P`.
- Gofer has a `where`-clause that works much the same way as a local Pascal function or variable.

Slide 7–13

Local definitions I (b) – where

- The `where`-clause follows **after** a function body:

  ```
  fun_name fun_args =
     <fun_body>
                 where
                    decl1
                    decl2
                    ...
                    decln
  ```
- A declaration $\texttt{decl}_i$ is like any global function definition.
- Note that a constant declaration $\boxed{\texttt{id = expr}}$ is allowed since it is seen as a constant 0-argument function.

Slide 7–14

Local definitions I (c) – where

```
deriv f x =
   (f(x+dx) - f x)/dx
         where dx = 0.0001

sqrt x = newton f x
         where f y = y^2 - x

g ::  Int -> Int
g n | (n `mod` 3) == x  = x
|     (n `mod` 3) == y  = y
|     (n `mod` 3) == z  = z
                  where x = 0
                        y = 1
                        z = 2
```

- Note that the **scope** (area of visibility) of a `where`-clause is the entire right-hand side of the function definition.

Slide 7–15

Local definitions II (a) – `let`

- An other, less flexible way, of introducing a local definition, is the `let`-clause.
- The syntax of a `let`-clause:

```
let
   <local_definitions>
in
   <expression>
```

- Note that the scope of the `let`-clause is only one expression, whereas the `where` clause can span over several.

Slide 7–16

Local definitions II (b) – `let`

```
f ::  [Int] -> [Int]
f [ ] = [ ]
f xs =
   let
      square a = a * a
      one = 1
      (y:ys) = xs
   in
      (square y + one) :  f ys
```

Simulation:

```
f [1,2] ⇒
   (square 1 + one) :  f [2] ⇒
   2 :  f [2] ⇒
      2 :  ((square 2 + one) :  f [])) ⇒
      2 :  (5 :  f []) ⇒
         2 :  (5 :  []) ⇒
      2 :  [5] ⇒
   [2,5]
```

Slide 7–17

Rational Arithmetic I

- Build a package implementing rational arithmetic. See B&W pp. 34–35, 38–39.

Arithmetic Laws:

$$\frac{a}{b}+\frac{c}{d} = \frac{ad+bc}{bd} \qquad \frac{a}{b}-\frac{c}{d} = \frac{ad-bc}{bd}$$

$$\frac{a}{b}*\frac{c}{d} = \frac{ac}{bd} \qquad \frac{a}{b}/\frac{c}{d} = \frac{ad}{bc}$$

Examples:

$$\frac{5}{4}+\frac{6}{7} = \frac{5*7+4*6}{4*7}=\frac{59}{28}$$

$$\frac{5}{4}*\frac{6}{7} = \frac{5*6}{4*7}=\frac{15}{14}$$

Slide 7–18

Rational Arithmetic II

- There is more than one way to represent the same rational number:

$$\frac{1}{7}=\frac{-1}{-7}=\frac{3}{21}=\frac{168}{1176}$$

  We would like to represent each rational number $\frac{a}{b}$ in the simplest way, called the **normal** form, such that a and b are **relatively prime**. Hence, $\frac{168}{1176}$ would always be represented as $\frac{1}{7}$.
- Two numbers a and b are relatively prime if a and b have no common divisor. 9 and 16 are relatively prime, but 9 and 15 aren't (they both have the common divisor 3).
- 0 is always represented by $\frac{0}{1}$.

Slide 7–19

Rational Arithmetic III

- We represent a rational number as a tuple of the numerator and the denominator:
  `type Rat = (Int, Int)`
- We normalize a rational number by dividing the numerator and denominator by their **greatest common divisor**.

```
normRat ::  Rat -> Rat
normRat (_,0) = error("Invalid!\n")
normRat (0,_) = (0,1)
normRat (x,y) = (a/d,b/d)
        where    a = (signum y) * x
                 d = abs y
                 d = gcd a b
normRat (-168,1176) ⇒ (-1,7)
```

Slide 7–20

Rational Arithmetic IV

The `signum` Function:

- `signum x` (from the standard prelude) returns -1 if `x` is negative, 0 if `x` is 0, and 1 if `x` is positive.

```
signum ::  (Num a, Ord a) => a -> Int
signum n |  n == 0   = 0
         |  n > 0    = 1
         |  n < 0    = -1
```

The `gcd` Function:

```
gcd ::  Int -> Int -> Int
gcd x y = gcd' (abs x) (abs y)
    where
       gcd' x 0 =  x
       gcd' x y =  gcd' y (rem x y)

gcd 78 42      ⇒ 6
```

Slide 7–21

Rational Arithmetic V

Arithmetic:

```
negRat ::  Rat -> Rat
negRat (a,b) = normRat (-a,b)

addRat, SubRat, mulRat, divRat ::
   Rat -> Rat -> Rat
addRat (a,b) (c,d) =
   normRat (a*d + c*b, b*d)
subRat (a,b) (c,d) =
   normRat (a*d - c*b, b*d)
mulRat (a,b) (c,d) =
   normRat (a*c, b*d)
divRat (a,b) (c,d) =
   normRat (a*d, b*c)
```

Examples:

```
?  addRat (4,5) (5,6)
   49/24
```

Slide 7–22

Rational Arithmetic VI

Relational Comparison:

```
eqRat ::  Rat -> Rat -> Bool
eqRat (a,0) (c,d) = err
eqRat (a,b) (c,0) = err
eqRat (a,b) (c,d) = (a*d == b*c)
   where err = error ("Invalid!")
```

Examples:

```
?  eqRat (4, 0) (4, 1)
   Invalid!
?  eqRat (4, 1) (4, 0)
   Invalid!
?  eqRat (4, 0) (4, 0)
   Invalid!
?  eqRat (4, 5) (4, 5)
   True
?  eqRat (4, 5) (4, 6)
   False
```

Slide 7–23

List Element Selection

- `!!n` returns the n:th element of a list.

Examples:

```
[2,5,8,3,9,5,7] !!3     ⇒ 8
[2,5] !!3               ⇒ ERROR
[[1],[2,3],[4]] !!2!!1  ⇒ 2
```

Gofer:

```
infixl 9 !!

(!!)  ::  [a] -> Int -> a
(x:_) !!  0       = x
(_:xs) !!  (n+1)  = xs !!  n
```

Slide 7–24

The `zip` Function

- `zip` takes two lists `xs` and `ys` and returns a list `zs` of pairs drawn from `xs` and `ys`. `xs` and `ys` are combined like the two parts of a zipper.
- Extra elements from different length lists are discarded.

Examples:

```
zip [1,2] ['a','b'] ⇒
   [(1,'a'),(2,'b')]
zip [1,2,3] ['a','b'] ⇒
   [(1,'a'),(2,'b')]
```

Gofer:

```
zip ::  [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys)
   = (x,y) :  zip xs ys
zip _ _ = [ ]
```

Slide 7–25

Breaking Lists I – `take`

- `take n xs` (from the standard prelude) takes a number n and a list, and returns the first n elements of the list.

Examples:

```
take 3 ['a','b','c','d','e'] ⇒
   ['a','b','c']
take 3 ['a','b'] ⇒ ['a','b']
```

Gofer:

```
take ::  Int -> [a] -> [a]
take 0 _              = [ ]
take _ [ ]            = [ ]
take (n+1) (x:xs)     = x :  take n xs
```

Slide 7–26

Breaking Lists II – `drop`

- `drop n xs` (from the standard prelude) takes a number n and a list, and returns the remaining elements when the first n have been removed.

Examples:

```
drop 3 ['a','b','c','d','e'] ⇒
   ['d','e']
drop 3 ['a','b'] ⇒ [ ]
```

Gofer:

```
drop ::  Int -> [a] -> [a]
drop 0 xs             = xs
drop _ [ ]            = [ ]
drop (n+1) (x:xs)     = drop n xs
```

Slide 7–27

Example – Making Change I

- Write a function `change` that computes the optimal (smallest) set of coins to make up a certain amount.

Defining available (UK) coins:

```
type Coin = Int
coins ::  [Coin]
coins = reverse (
          sort [1,2,5,10,20,50,100])
```

Example:

```
?  change 23
   [20,2,1]
?  coins
   [100,50,20,10,5,2,1]
?  all_change 4
   [[2,2],[2,1,1],[1,2,1],
   [1,1,2],[1,1,1,1]]
```

Slide 7–28

Example – Making Change II

- `all_change` returns all the possible ways of combining coins to make a certain amount.
- `all_change` returns shortest list first. Hence `change` becomes simple:

```
change amount =
  head (all_change amount)
```

- `all_change` returns all possible (decreasing sequences) of change for the given amount.

```
all_change ::  Int -> [[Coin]]
all_change 0 = [[]]
all_change amount = [ c:cs |
      c<-coins, amount>=c,
      cs<-all_change (amount - c) ]
```

Slide 7–29

Example – Making Change III

- `all_change` works by recursion from within a list comprehension. To make change for an amount `amount` we
  1. Find the largest coin `c` $\leq$ `amount`: `c<-coins,amount>=c`.
  2. Find how much we now have left to make change for: `amount - c`.
  3. Compute all the ways to make change from the new amount: `cs<-all_change (amount - c)`
  4. Combine `c` and `cs`: c:cs.

```
all_change ::  Int -> [[Coin]]
all_change 0 = [[]]
all_change amount = [ c:cs |
      c<-coins, amount>=c,
      cs<-all_change (amount - c) ]
```

Slide 7–30

Example – Making Change IV

- If there is more than one coin `c` $\leq$ `amount`, then `c<-coins,amount>=c` will produce all of them. Each such coin will then be combined with all possible ways to make change from `amount - c`.
- `coins` returns the available coins in reverse order. Hence `all_change` will try larger coins first, and return shorter lists first.

```
all_change ::  Int -> [[Coin]]
all_change 0 = [[]]
all_change amount = [ c:cs |
      c<-coins, amount>=c,
      cs<-all_change (amount - c) ]
```

Slide 7–31

Summary

- B&W reading list:
  1. Guards: 11, 27
  2. Local definitions: 11,22
  3. Rational arithmetic: 34–35, 38–39
  4. List indexing: 59
  5. Zip: 57, 112
  6. Take and drop: 55–56, 114
- `where` and `let` introduce local definitions. They enhance the readability of long function definitions.
- **otherwise** is an abbreviation for `True`.

Slide 7–32

Homework I

- Define a function `split xs` that takes alist of pairs, makes two lists, one from the first elements of the pair and the other from the second pair elements, and returns the two lists as a pair.

Examples:

```
split [(1,"a"),(2,"b"),(3,"c")]
   ⇒ ([1,2,3],["a","b","c"])

split [(1,True),(2,False),(3,False)]
   ⇒ ([1,2,3],[True,False,False])
```

Slide 7–33

Homework II (a)

- We model vectors as triples of floating point numbers:

  `type Vector = (Float, Float, Float)`

  Define functions `add'v`, `scale'v`, `dot'v` (dot product), and `cross'v` (cross product) according to the definitions below:

$$
\begin{aligned}
(a_1,a_2,a_3)+(b_1,b_2,b_3) &= (a_1+b_1,a_2+b_2,a_3+b_3)\\
k(a_1,a_2,a_3) &= (ka_1,ka_2,ka_3)\\
(a_1,a_2,a_3)\cdot(b_1,b_2,b_3) &= a_1b_1+a_2b_2+a_3b_3\\
(a_1,a_2,a_3)\times(b_1,b_2,b_3) &= (a_2b_3-b_2a_3,a_3b_1-a_1b_2,a_1b_2-a_2b_1)
\end{aligned}
$$

Slide 7–34

Homework II (b)

1. Now model 3×3 matrices as triples of `Vector`.
2. Define a function `scale'm` that scales a matrix `m` by a float `s`, i.e. multiplies all elements by `s`.
3. Define a function `add'm` that adds two matrices `a` and `b` together to form a new matrix `c`, i.e. $c_{i,j} = a_{i,j} + b_{i,j}$.
4. Define a function `transpose'm m` that turns the rows of a matrix `m` into columns, and vice versa, i.e. $t_{i,j} = m_{j,i}$.

Slide 7–35