Functional Programming

WS 2010/11

Christian Sternagel (VO)
Friedrich Neurauter (PS) Ulrich Kastlunger (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

November 24, 2010

Today's Topics

- Efficiency - Fibonacci Numbers
- Tupling
- Tail Recursion

Efficiency - Fibonacci Numbers

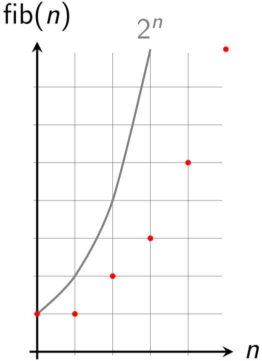
Definition - *n*-th Fibonacci Number

$$\mathrm{fib}(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{otherwise} \end{cases}$$

Definition - n-th Fibonacci Number

$$\text{fib}(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

Graph



Example

1

Example

$1, 1$

Example

1, 1, 2

Example

1, 1, 2, 3

Example

1, 1, 2, 3, 5

Example

1, 1, 2, 3, 5, 8

Example

1, 1, 2, 3, 5, 8, 13

Example

1, 1, 2, 3, 5, 8, 13, 21

Example

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181 ,6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817, 39088169, 63245986, 102334155, 165580141, 267914296, 433494437, 701408733, 1134903170, 1836311903, 2971215073, ...

Haskell

- definition

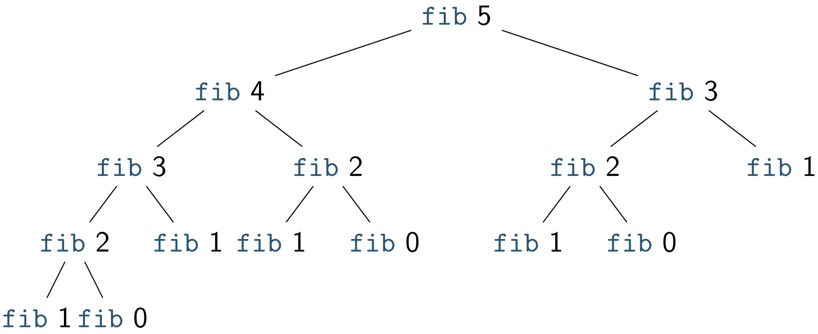
```
fib n | n <= 1     =  1
      | otherwise  =  fib (n-1) + fib (n-2)
```

Haskell

- definition

```haskell
fib n | n <= 1     =  1
      | otherwise  =  fib (n-1) + fib (n-2)
```

- example



Tupling

Combining Several Results

- use tuples to return more than one result
- make results available as return values instead of recomputing them

Combining Several Results

- use tuples to return more than one result
- make results available as return values instead of recomputing them

Fibonacci Numbers - Alternative Definition

- definition

```
fib' = snd . fibpair
  where fibpair n | n <= 0    = (0,1)
                  | otherwise = (f2,f1+f2)
          where (f1,f2) = fibpair (n-1)
```

- this function is linear in *n*
- since every recursive call reduces *n* by one

Combining Several Results

- use tuples to return more than one result
- make results available as return values instead of recomputing them

Fibonacci Numbers - Alternative Definition

- definition

```
fib' = snd . fibpair
  where fibpair n | n <= 0    = (0,1)
                  | otherwise = (f2,f1+f2)
          where (f1,f2) = fibpair (n-1)
```

- this function is linear in *n*
- since every recursive call reduces *n* by one

Exercise - `fibpair` computes `fib`

$$\texttt{fibpair}\ (n+1) = (\texttt{fib}\ n, \texttt{fib}\ (n+1))$$

Example - List Average

- goal: compute average of integer list

Example - List Average

- goal: compute average of integer list
- 1st approach:

```
average xs = sum xs `div` length xs
```

Example - List Average

- goal: compute average of integer list
- 1st approach:

```
average xs = sum xs `div` length xs
```

- two traversals of `xs`

Example - List Average

- goal: compute average of integer list
- 1st approach:

```
average xs = sum xs `div` length xs
```

- two traversals of `xs`
- combined function

```
average' xs = if l /= 0 then s/l
                        else 0
  where (s,l)         = sumlen xs
        sumlen []     = (0,0)
        sumlen (x:xs) = (sum + x,len + 1)
          where (sum,len) = sumlen xs
```

Example - List Average

- goal: compute average of integer list
- 1st approach:

```
average xs = sum xs `div` length xs
```

- two traversals of `xs`
- combined function

```
average' xs = if l /= 0 then s/l
                        else 0
  where (s,l)         = sumlen xs
        sumlen []     = (0,0)
        sumlen (x:xs) = (sum + x,len + 1)
          where (sum,len) = sumlen xs
```

- one traversal of `xs` suffices

Exercise

- show `sumlen` $xs = ($`sum` $xs,$ `length` $xs)$ by induction over xs

Tail Recursion

Recursion vs. Tail Recursion

- a function calling itself is recursive

Recursion vs. Tail Recursion

- a function calling itself is recursive
- functions that mutually call each other are mutually recursive

Recursion vs. Tail Recursion

- a function calling itself is recursive
- functions that mutually call each other are mutually recursive
- a special kind of recursion is tail recursion

Recursion vs. Tail Recursion

- a function calling itself is recursive
- functions that mutually call each other are mutually recursive
- a special kind of recursion is tail recursion
- a function is tail recursive, if the last action in the function body is the recursive call

Example - Recursive (but not Tail Recursive)

```
length  []    = 0
length (x:xs) = 1 + length xs
```

Example - Recursive (but not Tail Recursive)

```
length   []    = 0
length (x:xs) = 1 + length xs
```

Example - Mutually Recursive (and Tail Recursive)

```
even n | n <= 0    = True
       | otherwise = odd (n-1)
odd n  | n <= 0    = False
       | otherwise = even (n-1)
```

Example - Recursive (but not Tail Recursive)

```
length   []    = 0
length (x:xs) = 1 + length xs
```

Example - Mutually Recursive (and Tail Recursive)

```
even n | n <= 0     = True
       | otherwise = odd (n-1)
odd n  | n <= 0     = False
       | otherwise = even (n-1)
```

Example - Tail Recursive

```
reverse = rev []
  where rev acc   []    = acc
        rev acc (x:xs) = rev (x:acc) xs
```

Accumulating Parameters

- idea: make function tail recursive

Accumulating Parameters

- idea: make function tail recursive
- provide intermediate results as additional input

Accumulating Parameters

- idea: make function tail recursive
- provide intermediate results as additional input
- why? (tail recursive functions can be transformed into space-efficient loops automatically)

Example

```
sumlen' = sl 0 0
  where sl s l   []   = (s,l)
        sl s l (x:xs) = sl (s+x) (l+1) xs
```

Example

```
sumlen' = sl 0 0
  where sl s l    []    = (s,l)
        sl s l (x:xs) = sl (s+x) (l+1) xs
```

Exercise

- show `sumlen` *xs* = `sumlen'` *xs* by induction over *xs*

Problem

- lazy evaluation
- hence `s+x` and `l+1` are only evaluated when result of `sumlen'` is used
- results in huge memory consumption
- e.g.,

  $$0 + 1 + 2 + \cdots + 1000000$$

  is stored for computing `sumlen' [1..1000000]`
- a thunk of 1000001 integers (about 8MB)

Exercises (for December 3rd)

1. Read `http://www.haskell.org/haskellwiki/Tail_recursion` and `http://en.wikipedia.org/wiki/Tail_recursion#Tail_recursion_modulo_cons`
2. Find a function in the lecture slides of the previous weeks that is not tail recursive. Justify your answer.
3. Give a tail recursive implementation of `range`.
4. Use induction to prove that the function from Exercise 3 indeed computes `range`.
5. Use tupling to implement a more efficient version of `splitAt n xs = (take n xs, drop n xs)`
6. Use induction to prove that the function from Exercise 5 computes `splitAt`.