Computational Logic

Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 8

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, implementing a type checker, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

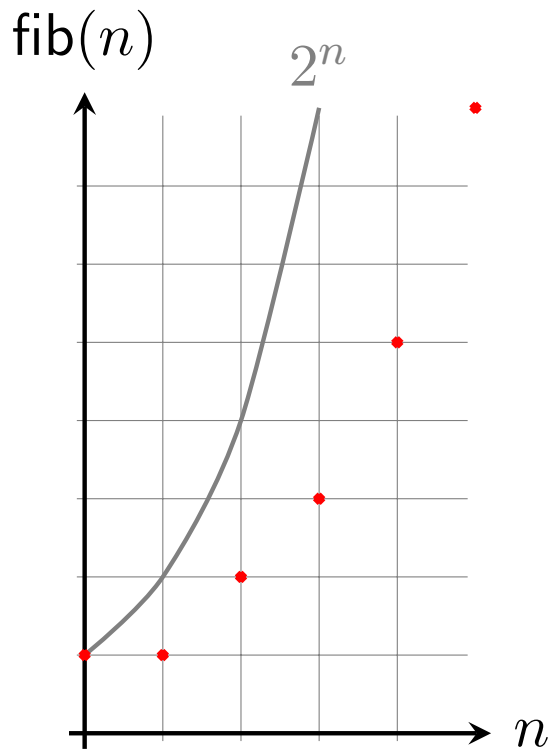
Overview

- Efficiency – Fibonacci Numbers
- Tupling
- Tail Recursion and Guarded Recursion
- Property-Based Testing with LeanCheck

Definition – n-th Fibonacci Number

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

Graph



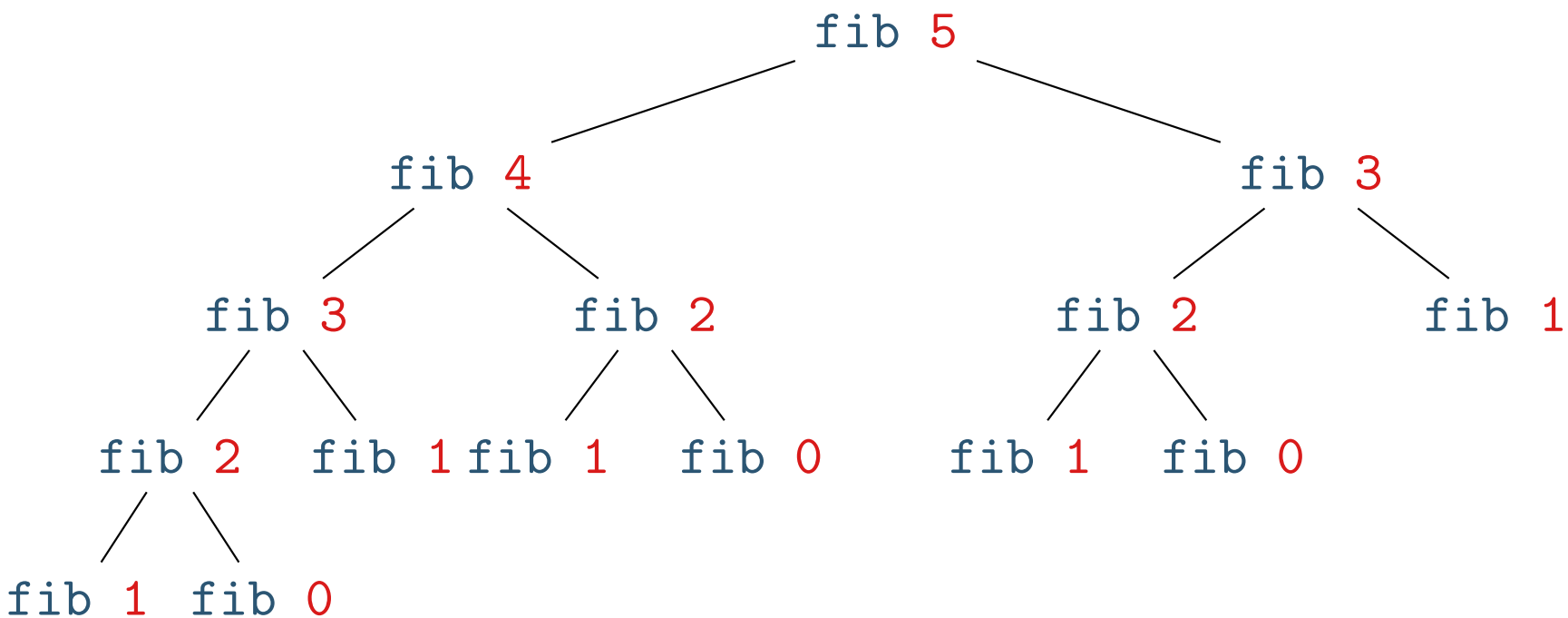
Example

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)
```

Example



Combining 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
- 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) = (s + x, l + 1)
      where
        (s, l) = sumlen xs
```

- one traversal of `xs` suffices

Exercise

show `sumlen` $xs = (\texttt{sum}\ xs, \texttt{length}\ xs)$ by induction over xs

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 – 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
```

Guarded Recursion

- every recursive call is inside ("guarded by") a data constructor
- also known as "tail recursion modulo cons"
- more important than tail recursion in Haskell
- allows the result to be consumed lazily (laziness will be discussed in final lecture)

Example – Guardedly Recursive

```
map f [] = []
map f (x:xs) = f x : map f xs
```

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
```

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.,

  $$0 + 1 + 2 + \cdots + 100000000$$

  is stored for computing `sumlen' [1..100000000]`
- an unevaluated thunk of 100000001 integers (which requires more than 20GB of RAM)

Installing LeanCheck with Cabal

- Cabal is system for building, packaging, and installing Haskell libraries and programs
- update package list
  ```
  $ cabal update
  ```
- install latest Cabal libraries
  ```
  $ cabal install Cabal
  ```
- install LeanCheck
  ```
  $ cabal install leancheck
  ```
- now we can use LeanCheck
  ```
  import Test.LeanCheck
  ```
- (tested on `zid-gpl.uibk.ac.at`)

Enumerative Property-Based Testing

- properties are Haskell functions with result type `Bool`
- idea is that property should return `True` for all possible inputs
- properties are tested on enumeration of values of argument type

Useful LeanCheck Functions

- `holds n p` – test property `p` on `n` enumerated inputs (return `True` or `False`)
- `witnesses n p` – like `holds` but instead of `Bool` return list of values on which `p` holds (length of result is `n` iff `holds n p` is `True`)
- `check p` – check property `p` and generate report as IO action
- `c ==> p` – implication, test `p` only if condition `c` holds
- `list :: a` – (potentially infinite) enumeration of values of type `a` (these are the values used for testing)

Examples

- define property

```
prop_rev_app xs ys =
  reverse (xs ++ ys) == reverse ys ++ reverse xs
```

- test property

```
ghci> check prop_rev_app
+++ OK, passed 200 tests.
```

- which values did we actually test?

```
ghci> witnesses 200 prop_rev_app
[["[]","[]"],["[]","[()]"],["[()]","[]"],
 ["[]","[(),()]"],["[()]","[()]"], ...]
```

- without type annotations, type variables default to unit type `()`
- better

```
prop_rev_app :: [Int] -> [Int] -> Bool
prop_rev_app xs ys =
  reverse (xs ++ ys) == reverse ys ++ reverse xs
```

Examples (cont'd)

- define a wrong property

```
prop_app_commute_wrong :: [Int] -> [Int] -> Bool
prop_app_commute_wrong xs ys = xs ++ ys == ys ++ xs
```

- test it

```
ghci> check prop_app_commute_wrong
*** Failed! Falsifiable (after 14 tests):
[0] [1]
```

- which values did we test?

```
ghci> take 14 $ list :: [([Int],[Int])]
[([],[]),([],[0]),([0],[]),([],[0,0]),([],[1]),
([0],[0]),([0,0],[]),([1],[]),([],[0,0,0]),
([],[0,1]),([],[1,0]),([],[-1]),([0],[0,0]),([0],[1])]
```

- a conditional property

```
prop_take_neq i xs =
  i < length xs ==> take i xs /= xs
```

- does it hold?

Exercises (for December 15th)

1. Read the lecture notes on efficiency, 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. Consider the function
```
range m n | m > n      = []
          | otherwise = m : range (m + 1) n
```
Give a tail recursive variant of `range`.
4. Use LeanCheck to test whether

$$\texttt{range}\ m\ (n-1)\ \texttt{++}\ [n] = \texttt{range}\ m\ n$$

holds. If not, can you add a condition, so that it does?
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`.