



Functional Programming

Lecture 8

Cezary Kaliszyk Jonas Schöpf
Christian Sternagel Vincent van Oostrom

Department of Computer Science

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, 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

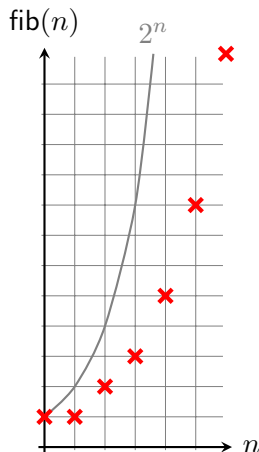
Overview

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

Definition – n th Fibonacci Number

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

Graph



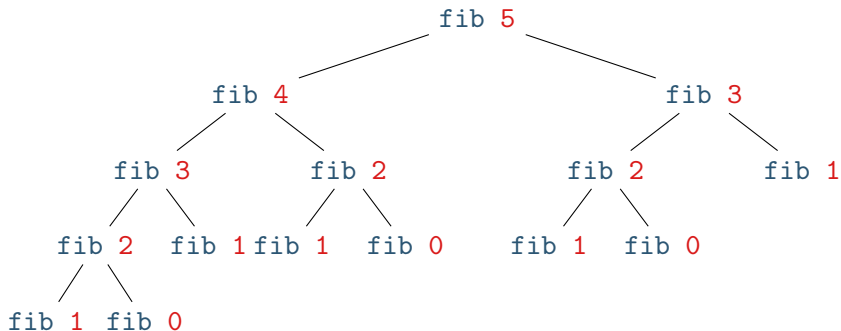
Example

1, 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, 4807526976, 7778742049, 12586269025, 20365011074, 32951280099, 53316291173, ...

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

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

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 = (sum xs, 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)
```


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

Example – Guardedly Recursive

```
map f [] = []
```

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

Accumulating Parameters

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

Example – Tail Recursive

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

Example

```
sumlen' = sl 0 0
```

```
  where
```

```
    sl s l [] = (s, l)
```

```
    sl s l (x:xs) = sl (s + x) (l + 1) xs
```

Exercise

- prove that `sumlen' xs = (sum xs, length xs)` for all `xs`

Problem with `sumlen'` – Memory Leak

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

$$0 + 1 + 2 + \dots + 100000000$$

is stored for computing `sumlen'` [`1..100000000`]

- an unevaluated **thunk** of 100000001 integers (which requires more than 20GB of RAM)

Example – Grouping Consecutive Copies

- specification of `group`: turn given list into list of lists, such that order of elements is preserved and consecutive copies are packed together

- expected behavior

```
group [] = [],
```

```
group [1,2,3] = [[1],[2],[3]],
```

```
group [1,1,2,2] = [[1,1],[2,2]]
```

- implementation

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

```
group [] = []
```

```
group [x] = [[x]]
```

```
group (x:y:ys) | x == y = [x,y] : group ys
                | otherwise = [x] : group (y:ys)
```

Unit Testing

- check program **unit** (for example, single function)
- check specific inputs (like [], [1,2,3], [1,1,2,2])
against expected results (like [], [[1],[2],[3]], [[1,1],[2,2]])

Example – group

```
unitTests :: Bool
unitTests = and [
  group [] == ([] :: [[Int]]),
  group [1,2,3] == [[1],[2],[3]],
  group [1,1,2,2] == [[1,1],[2,2]]
]
```

What is Property-Based Testing?

- also known as **parameterized unit testing** or **property testing**
- define properties that should hold for all possible choices of inputs
- then **try inputs** in search of counterexample

Example – Some Properties of group

- concatenating groups results in original input

```
prop_concat :: Eq a => [a] -> Bool
```

```
prop_concat xs = concat (group xs) == xs
```

- all groups are nonempty

```
prop_nonempty :: Eq a => [a] -> Bool
```

```
prop_nonempty xs = all (not . null) (group xs)
```

Missing

- how to **try inputs**?
- possibilities: enumerate or random-generate

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`
- 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)

Exercise

test `prop_concat` and `prop_nonempty`

Example – More Properties of group

- all elements of a group are equal

```
prop_all_equal :: Eq a => [a] -> Bool
```

```
prop_all_equal xs = all all_equal (group xs)
```

```
  where
```

```
    all_equal (x:y:ys) = x == y && all_equal (y:ys)
```

```
    all_equal _ = True
```

- check that consecutive groups do not contain same elements

```
prop_nonequal :: Eq a => [a] -> Bool
```

```
prop_nonequal xs = nonequal (group xs)
```

```
  where
```

```
    nonequal (x:y:ys) = head x /= head y &&
```

```
      nonequal (y:ys)
```

```
    nonequal _ = True
```

Exercise

test `prop_all_equal` and `prop_nonequal`

Example – Output of Exercise

```
ghci> check prop_all_equal
+++ OK, passed 200 tests.
ghci> check prop_nonequal
*** Failed! Falsifiable (after 4 tests):
[(),(),()]
```

Correct Implementation of group

```
group :: Eq a => [a] -> [[a]]
group [] = []
group [x] = [[x]]
group (x:y:ys) | x == y = (x:zs) : zss
                | otherwise = [x] : zs : zss
  where zs:zss = group (y:ys)
```

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

```
ghci> check (prop_rev_app :: [Int] -> [Int] -> Bool)
```

Examples (cont'd)

- define wrong property

```
prop_wrong :: [a] -> [a] -> Bool
prop_wrong xs ys = xs ++ ys == ys ++ xs
```

- test it

```
ghci> check (prop_wrong :: [Int] -> [Int] -> Bool)
*** 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?

Homework (for January 11th)

1. Read the lecture notes on efficiency, https://wiki.haskell.org/Tail_recursion, and http://en.wikipedia.org/wiki/Tail_recursion#Tail_recursion_modulo_cons
2. Find one function that is tail recursive, one that is guardedly recursive, and one that is neither, in the lecture slides of the earlier weeks. In each case, justify your answer.
3. Use tupling to implement a more efficient version of `initLast xs = (init xs, last xs)` and prove by induction that it coincides with `initLast` on non-empty lists.
4. Give a tail recursive implementation of the factorial function
$$\text{fac } n \mid n \leq 1 = 1$$
$$\mid \text{otherwise} = n * \text{fac } (n - 1)$$
and prove by induction that it coincides with `fac`.

Homework (for January 11th, continued)

5. Use `LeanCheck` to test whether
- ```
sort' xs = foldr insort [] xs
```

```
 where insort x [] = [x]
```

```
 insort x (y:ys)
```

```
 | x < y = x:y:ys
```

```
 | x == y = x:ys
```

```
 | otherwise = y : insort x ys
```

is a correct sorting function.

**Hint:** Implement two functions `sorted :: Ord a => [a] -> Bool` and `count :: Eq a => [a] -> a -> Int` and use them to express the desired property.

6. Give a tail recursive variant of

```
nfold n f x = head $ drop n $ iterate f x
```

and use `LeanCheck` to test whether it coincides with `nfold`.

**Hint:** Import `Test.LeanCheck.Function` to test properties of higher-order functions.

# Optional Programming Exercise (for January 11th)

## Setup:

- (this exercise is worth up to 6 crosses)
- consider propositional formulas  $\phi ::= \perp \mid p \mid \phi \rightarrow \phi$
- additional logical connectives given by  $\neg\phi \stackrel{\text{def}}{=} \phi \rightarrow \perp$ ,  $\phi \vee \psi \stackrel{\text{def}}{=} \neg\phi \rightarrow \psi$ , and  $\phi \wedge \psi \stackrel{\text{def}}{=} \neg(\phi \rightarrow \neg\psi)$
- six **basic inference rules**

$$\begin{array}{ccc} \frac{}{\phi \vdash \phi} \text{ (trivial)} & \frac{\Gamma \vdash \phi}{\psi, \Gamma \vdash \phi} \text{ (assume)} & \frac{\Gamma, \phi, \Delta \vdash \psi \quad \phi \in \Gamma \cup \Delta}{\Gamma, \Delta \vdash \psi} \text{ (discard)} \\ \frac{\Gamma \vdash \neg\neg\phi}{\Gamma \vdash \phi} \text{ (\neg e)} & \frac{\Gamma, \phi, \Delta \vdash \psi}{\Gamma, \Delta \vdash \phi \rightarrow \psi} \text{ (\rightarrow i)} & \frac{\Gamma \vdash \phi \rightarrow \psi \quad \Delta \vdash \phi}{\Gamma, \Delta \vdash \psi} \text{ (\rightarrow e)} \end{array}$$



## Optional Programming Exercise (for January 11th, continued)

- Based on the module `Formula`, implement an ADT `Proof` together with the six basic inference rules represented by the Haskell functions:

```
trivial :: Formula -> Proof
```

```
assume :: Formula -> Proof -> Proof
```

```
discard :: Int -> Proof -> Maybe Proof
```

```
impI :: Int -> Proof -> Maybe Proof
```

```
impE :: Proof -> Proof -> Maybe Proof
```

```
nnegE :: Proof -> Maybe Proof
```

- Furthermore, implement as many of the following derived rules as possible without breaking the abstraction:

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \neg\neg\phi} \text{ (}\neg\neg\text{i)} \quad \frac{\Gamma \vdash \phi \quad \Delta \vdash \psi}{\Gamma, \Delta \vdash \phi \wedge \psi} \text{ (}\wedge\text{i)} \quad \frac{\Gamma \vdash \phi_1 \wedge \phi_2 \quad i \in \{1, 2\}}{\Gamma \vdash \phi_i} \text{ (}\wedge\text{e)}$$

$$\frac{\Gamma \vdash \phi_i \quad i \in \{1, 2\}}{\Gamma \vdash \phi_1 \vee \phi_2} \text{ (}\vee\text{i)} \quad \frac{\Gamma \vdash \phi \vee \psi \quad \phi, \Gamma \vdash \chi \quad \psi, \Gamma \vdash \chi}{\Gamma \vdash \chi} \text{ (}\vee\text{e)}$$

## Example – Peirce's Law

```
peirce :: Formula -> Formula -> Proof
```

```
peirce f g =
```

```
 let Just p0 = trivial (f `Imp` Bot) `impE` trivial f in
```

```
 let Just p1 = impI 0 $ assume (g `Imp` Bot) p0 in
```

```
 let Just p2 = nnegE p1 in
```

```
 let Just p3 = impI 1 p2 in
```

```
 let Just p4 = trivial ((f `Imp` g) `Imp` f) `impE` p3 in
```

```
 let Just p5 = trivial (f `Imp` Bot) `impE` p4 in
```

```
 let Just p6 = discard 0 p5 in
```

```
 let Just p7 = impI 1 p6 in
```

```
 let Just p8 = nnegE p7 in
```

```
 let Just p9 = impI 0 p8 in
```

```
 p9
```

```
> peirce (read "A") (read "B")
```

```
⊢ ((A → B) → A) → A
```