



Functional Programming

Week 13 – Lambda Calculus, Summary

René Thiemann

Jonathan Bodemann

James Fox

Joshua Ocker

Daniel Rainer

Daniel Ranalter

Christian Sternagel

Last Lecture

- cyclic definitions, e.g., `fibs = 0 : 1 : zipWith (+) fibs (tail fibs)`
- abstract data types
 - specify type of operations and behavior
 - hide implementation details (via suitable module export-lists)
 - example: queues
 - used to implement breadth-first-search in trees
 - basic implementation was simple, n operations require $\sim \frac{1}{2}n^2$ evaluation steps
 - improved implementation represents queues as two lists, n operations require $\sim 2n$ eval. steps

λ -Calculus

A Glimpse of λ -Calculus

- λ -calculus works on λ -terms, which is either a λ -abstraction, a variable, or an application
- no types, no data type definitions, no function definitions, no built-in arithmetic, . . .
- only one evaluation mechanism: β -reduction

replace $(\lambda x \rightarrow s) t$ by $s[x/t]$

where $s[x/t]$ is the term s where the variable x is substituted by t

- sufficiently strong to encode functional programs

Booleans in λ -Calculus

- encode Booleans as λ -terms, i.e., implement `Bool` as abstract data type
 - internal construction of provided operations
 - `Bool`: $a \rightarrow a \rightarrow a$
 - `True`: $\lambda x y \rightarrow x$
 - `False`: $\lambda x y \rightarrow y$
 - `if-then-else`: $\lambda c t e \rightarrow c t e$
 - satisfied axioms
 - `(if True then t else e) = t`:
$$\begin{aligned} & (\lambda c t e \rightarrow c t e) (\lambda x y \rightarrow x) t e \\ &= (\lambda t e \rightarrow (\lambda x y \rightarrow x) t e) t e \\ &= (\lambda e \rightarrow (\lambda x y \rightarrow x) t e) e \\ &= (\lambda x y \rightarrow x) t e \\ &= (\lambda y \rightarrow t) e \\ &= t \end{aligned}$$
 - `(if False then t else e) = e`: similar

Booleans in λ -Calculus, continued

- so far, we have λ -terms that encode `True`, `False`, and `if-then-else`
- other Boolean functions can easily be encoded

- `b && c = if b then c else False`
- `b || c = if b then True else c`
- `not b = if b then False else True`

- example: computation of `False && True`:

```
False && True           -- unfold encoding of &&
= if False then True else False -- unfold encoding of ite, False, True
= (\ c t e -> c t e) (\ x y -> y) (\ x y -> x) (\ x y -> y)
  -- the line above is the lambda-term that is evaluated
= (\ t e -> (\ x y -> y) t e) (\ x y -> x) (\ x y -> y)
= (\ e -> (\ x y -> y) (\ x y -> x) e) (\ x y -> y)
= (\ x y -> y) (\ x y -> x) (\ x y -> y)
= (\ y -> y) (\ x y -> y)
= \ x y -> y           -- representation of False
```

Pairs in λ -Calculus

- pairs can be encoded similarly to Booleans
- we need three operations: `(x, y)`, `fst`, `snd`
 - encoding of pairs is not typable in Haskell
 - encoding of `(x, y)`: `\ c -> if c then x else y`
 - encoding of `fst`: `\ p -> p True`
 - encoding of `snd`: `\ p -> p False`

- soundness, e.g., `snd (x, y) = y`

```
snd (x, y)                                -- expand snd and (x, y)
= (\ p -> p False) (\ c -> if c then x else y)      -- beta
= (\ c -> if c then x else y) False                -- beta
= if False then x else y                          -- soundness of ite
= y
```

- using pairs, we can model tuples and lists

Church Numerals

- also natural numbers can be represented in λ -calculus
- Church numerals: n is encoded as $\lambda f x \rightarrow f (f \dots (f x) \dots)$ with n applications of f
- encoding type of natural numbers: $(a \rightarrow a) \rightarrow a \rightarrow a$
- examples
 - zero: $\lambda f x \rightarrow x$
 - one: $\lambda f x \rightarrow f x$
 - two: $\lambda f x \rightarrow f (f x)$
 - test on zero: $\lambda n \rightarrow n (\lambda b \rightarrow \text{False}) \text{ True}$
 - successor: $\lambda n f x \rightarrow f (n f x)$
 - addition: $\lambda n m f x \rightarrow n f (m f x)$
 - multiplication: $\lambda n m f x \rightarrow n (m f) x$
 - predecessor: possible, but more difficult

Recursion

- for defining general recursion, one can use the Y-combinator:

```
Y = \ f -> (\ x -> f (x x)) (\ x -> f (x x))
```

- important property: $Y\ g$ reduces to $g\ (Y\ g)$, i.e., $Y\ g$ is a fixpoint of g : $g\ (Y\ g) = Y\ g$

- recursive functions can be written as **fixpoints** of non-recursive functions

```
add x y = if x == 0 then y else add (x-1) (y+1)
```

```
-- add is fixpoint of the non-recursive function addNR
```

```
-- equality: addNR add = add
```

```
addNR a x y = if x == 0 then y else a (x-1) (y+1)
```

- encoding of above addition function in λ -calculus
 - encode non-recursive function `addNR` as λ -term `t` similarly to previous slides
 - encode `add` as fixpoint: `add = fixpoint of addNR = Y t`

Summary of Course

What You Should Have Learned

- definition of types and functions
 - type definitions via `type`, `newtype`, and `data`
 - specify functions in various forms: pattern matching, recursion, combination of predefined (higher-order) function, list comprehensions, ...
- understanding of types
 - parametric polymorphism and type classes
 - ability to infer most general types for simple definitions
- I/O in Haskell, do-notation, compilation with `ghc`
- definition and advantages of modules and abstract data types
- evaluation strategies, in particular Haskell's lazy evaluation
- basic knowledge of predefined types and functions within Prelude
 - types `Int`, `Integer`, `Double`, `[a]`, `Maybe a`, `Either a b`, `String`, `Char`, `Bool`, tuple
 - type classes for numbers, `Show`, `Read`, `Eq`, `Ord`
 - arithmetic and Boolean functions and operators
 - functions involving lists and strings
 - I/O: primitives for reading and writing (also into files)

What You Did Not Learn in This Course

- type inference algorithms
- compilation of functional programs
- static analysis and optimization of functional programs
- debugging and verification of functional programs
- concurrency
- more functional programming techniques (monads, functors, continuations, ...)