WS 2023/2024

Functional Programming

Week 13 – Lambda Calculus, Summary, Previous Exam

René Thiemann James Fox Lukas Hofbauer Christian Sternagel Tobias Niederbrunner

Department of Computer Science

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 `(\ x -> 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 -> a -> a`
    - `True`: `\ x y -> x`
    - `False`: `\ x y -> y`
    - `if-then-else`: `\ c t e -> c t e`
  - satisfied axioms
    - `(if True then t else e) = t`:

```
  (\ c t e -> c t e) (\ x y -> x) t e
= (\ t e -> (\ x y -> x) t e) t e
= (\ e -> (\ x y -> x) t e) e
= (\ x y -> x) t e
= (\ y -> t) e
= t
```

    - `(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 `\ f x -> f (f ... (f x) ...)` with n applications of `f`
- encoding type of natural numbers: `(a -> a) -> a -> a`
- examples
  - zero: `\ f x -> x`
  - one: `\ f x -> f x`
  - two: `\ f x -> f (f x)`
  - test on zero: `\ n -> n (\ b -> False) True`
  - successor: `\ n f x -> f (n f x)`
  - addition: `\ n m f x -> n f (m f x)`
  - multiplication: `\ n m f x -> 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, . . .)