Functional Programming (H)
Lecture 17: Lambda Calculus & Equational Reasoning

Thu 28 Nov 2024 (slides v3)

Simon Fowler and **Jeremy Sing**

last
examinable
lecture

News Update

- Next week - non-examinable (but very exciting) lectures
- Coursework deadline - due Friday 6 December
- Lab tomorrow - focus on Solitaire coursework

- Today - $(FP)^2$ - Functional Programming Fashion Parade - hats and jumpers
- Evasys online questionnaires (?)

Today's learning objectives

- appreciate the differences between **C-like** and **Haskell-like** approaches to program evaluation
- recognize the *benefits* of **referential transparency**
- construct and calculate simple expressions in the **untyped lambda calculus**
- understand the motivation for **static analysis & reasoning** about programs in Haskell
- perform simple **structural induction** for functions that operate on lists

Referential Transparency

What is referential transparency?

- An expression is said to be *referentially transparent* if that expression can be replaced with its corresponding value without modifying the program behaviour.
- i.e. expression *value* is always independent of expression *context*
- cf. pure functions in Python / Java / … / Haskell!
- expressions always evaluate the same way

non-referential transparency

- e.g. side-effecting code in C …

```
int f(int x) {
    static int secret = 0;
    secret++;
    return x+secret;
}

printf("%d\n", f(1) + f(1));
// compare with …
int tmp = f(1);
printf("%d\n", tmp + tmp);
```

hidden preserved state across function calls

prints 5

prints 4

Haskell example of Referentially Transparent Code

- **`let f x = x+1`**
- **`let tmp = (f 1) in print $ tmp + tmp`**
- cf
- **`print $ (f 1) + (f 1)`**

- these evaluate in precisely the same way!
- If we wanted to preserve state across calls to f, we would need to use the State monad.

Using State in Haskell

```
import Control.Monad.State
import Control.Monad.IO.Class (liftIO)

f :: Int -> StateT Int IO ()
f x = do
  secret <- get
  let result = x+secret+1
  liftIO $ putStrLn (show result)
  put (secret+1)

main :: IO()
main = do
  ((), state') <- runStateT (f 1) 0
  ((), state'') <- runStateT (f 1) state'
  return ()
```

hidden preserved state across function calls

`liftIO` to do IO action within State transformer stack

why is Referential Transparency good?

- good for **reasoning** about programs
- whether manually, or automated … whether informally or formally
- good for **parallelism** - no side-effects / inter-thread dependencies
- evaluation is simply ***term rewriting***
  - cf evaluation of lambda terms

Lambda Calculus

Untyped Lambda Calculus

- invented by Alonzo Church in 1930s
- a theoretical model for computation
  - universal computation can be represented in lambda calculus!
- equivalent to Turing machines
  - Church-Turing thesis
- basis of functional programming
- evaluation involves term rewriting

Components (terms) of Lambda Calculus

- **variables:** like *x*, *y*
- **function abstraction:** **λ**x . M
- **function application:** M N

x is a bound variable in M

this is like a function definition

this is a function invocation

Rewrite Rules for Lambda Calculus (1)

- *α conversion is bound-variable renaming*

$$\boldsymbol{\lambda}\, x \,.\, M \xrightarrow{\quad\alpha\quad} \boldsymbol{\lambda}\, y \,.\, M\,[y/x]$$

y is a **fresh** variable name

- example:
- $\boldsymbol{\lambda}\, x \,.\, x \xrightarrow{\alpha} \boldsymbol{\lambda}\, z \,.\, z$

identity function is the same, no matter what name we give to its parameter

Rewrite Rules for Lambda Calculus (2)

- β *reduction is function call evaluation*

$$(\backslash x\ .\ M)\ A \xrightarrow{\beta} M\ [A/x]$$

parameter *x* is replaced by lambda-term *A* (argument value)

- example:
- $(\boldsymbol{\lambda}\, x\ .\ \boldsymbol{\lambda}\, y.\ y\ x)\ (\boldsymbol{\lambda}\, z\ .\ z) \xrightarrow{\beta} \boldsymbol{\lambda}\, y.\ y\ (\ \boldsymbol{\lambda}\, z\ .\ z)$

replace *x* by \z.z in (\y.yx)

Computability

Any computable function can be represented in Lambda calculus

Church numerals - encoding of numbers:

A Church numeral is a function that takes two parameters:
\f -> \x -> …

```
0: \f -> \x-> x
1: \f -> \x -> f x
2: \f -> \x -> f (f x)
n: \f -> \x -> f^n x -- n applications of f
                     -- (iterate f x) !! n
```

repeated function application

Adding Church numerals

- add - take two Church numerals M and N and use M as the 'zero' parameter for N

- **`add = \ M N f x -> N f (M f x)`**

- **`this will evaluate to:`**
- $\underbrace{\texttt{f (f (f …}}_{\text{N applications of f}}\ \underbrace{\texttt{(f … (f x)}}_{\text{M applications of f}}\ \texttt{)))}$

N applications of f, applied to M applications of f, applied to x

Decode Church numerals in Haskell

- **`unchurch :: (a->a) -> a -> a -> Int`**
- **`unchurch n = n (+1) 0`**

Note about lambda calculus

- It's more of a theoretical representation, not actually used for computation in 'the real world'

- It shows the smallest 'programming language' that we can build to represent any computable function - such a programming language only needs function abstraction and function application

- Of course, although the programming language is trivially simple, the corresponding programs will be very verbose.

Extra (non-examinable) info about lambda calculus

- https://www.youtube.com/watch?v=eis11j_iGMs - Intro to Lambda calculus from Computerphile

- https://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf - Textbook on Lambda calculus

Equational Reasoning

program properties

- previously we used *QuickCheck* to do property based testing
- this was **dynamic testing** - use random input values to test specified invariants hold, for a fixed number of tests.
- Now - we want to perform **static verification** - prove properties of functions generally, that hold for all inputs …

A simple example

- Given a function definition:

```
example :: Int -> Int -> Int -> Int
example x y z = x*(y+z)
```

- Prove that example a b c == example a c b

```
example a b c
= a * (b+c)          { example function def }
= a * (c + b)        { symmetry of + }
= example a c b      { example function def }
```

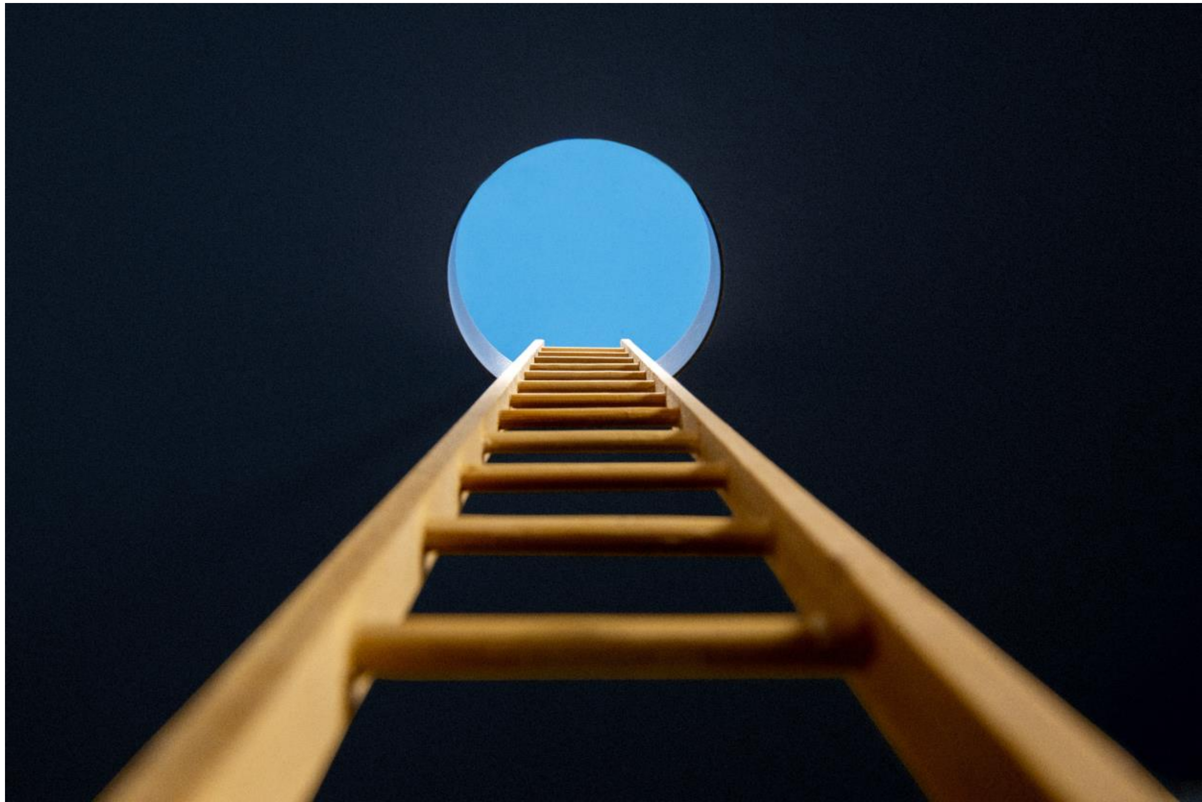
rewrites

justification for rewrites

only using function definition and simple properties of arithmetic

proofs by induction

- do you remember these from Algorithmic Foundations 2 course?
- or from high school maths?



structural induction

- given a property *p(x)* where *x* is a list
- prove *p(* **`[]`** *)*
  - i.e. property holds for empty list (base case)
- prove that *p(* **`xs`** *) -> p(* **`x:xs`** *)*
  - i.e. if property holds for list xs of length n, then we can show it holds for list (x:xs) of length (n+1) (inductive step)
- then by structural induction, *p* holds for arbitrary lists

Some rewrites we might need to make …

-- library definition of (++)

```
[] ++ ys = ys -- base case
(x:xs) ++ ys = x:(xs++ys) -- recursive case
```

-- library definition of length

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

example of structural induction

- We want to prove that, for all lists xs and ys:
  - **`length (xs ++ ys) = (length xs) + (length ys)`**

  - base case, xs is **`[]`**

length ([] ++ ys) = (length []) + (length ys)

{definition of append} {definition of length}

length (ys) = 0 + (length ys)

{arithmetic}

length ys = length ys

true

inductive step

- assume for list xs of length n …
  - **`length (xs ++ ys) = (length xs) + (length ys)`**

  - now, given list (x:xs) of length (n+1)

length ((x:xs) ++ ys) = (length (x:xs)) + (length ys)

{definition of length} {definition of length}

1 + length (xs ++ ys) = (1 + length xs) + (length ys)

{inductive hypothesis} {associativity of (+)}

1 + ((length xs) + (length ys)) = 1 + ((length xs) + (length ys))

true

A second example of structural induction

- We want to prove that, for all lists xs

```
length xs >= 0
```

i.e. length of arbitrary list `xs` is non-negative

base case, xs is []

length [] == 0 { definition of length function}

0 >= 0   BASE CASE is proven

inductive step

- assume for list xs
  - length (xs) >= 0

- Now need to prove that for list x:xs
  - length(x:xs) >= 0

length(x:xs) == 1 + length (xs) {definition of length}

We are assuming length xs >= 0, so

1 + length xs >= 0 { laws of arithmetic}

INDUCTIVE STEP IS PROVEN

Examples to try at home

- list append function `(++)` is associative
- Can also do structural induction over other recursive data types e.g. binary trees
- Further info online: https://www.youtube.com/watch?v=cQ1iziWpt8Q

Takeaways

- **Lambda calculus** is a powerful formalism to model computing
- *'All you need is lambda'*

- **Equational reasoning** enables us to prove program properties

- Next week - non-examinable (but very exciting) lectures