The λ-Calculus is Practical, Actually

Background

The λ-calculus is a theory of computation and was created by Alonzo Church in the 1930s.

> *an alternative to Turing machine. Turing machine read/write to memory whereas lambda-calculus all about functions*
>
> *could think of as a "programming language", but, really, different foundation for computation*
>
> *by universality theorem, can simulate eachother. that's what lambda-calculus interpreters are doing*

Contents

- Quickstart
- Named Bindings
- Algebraic Data Types
- Self-Reference
- Type Classes
- Unexpected Application

> *goal of the talk: Church's vanilla lambda-calculus is surprisingly practical/usable as-is*

Quickstart

> *familiarize with syntax*

```
// JavaScript
// prettier-ignore
(x => y => x(y))(z => z(z));
```

```
# Python
(lambda x: lambda y: x(y))(lambda z: z(z))
```

```
; λ-calculus
(\x. \y. x y) (\z. z z)
```

> *all functions take one argument; currying to take more*
>
> *justaposition means function application*

$(\lambda xy.xy)(\lambda z.zz)$

above is what one might see in the literature. won't be using because:

- *multi-letter variables would cause ambiguity*
- *LaTeX-formatting everything is tedious*

the notation we're using is not completely *made up… it's kind of Haskell-ish*

Named Bindings

naming useful because allows for encapsulation/abstraction: using a simple name to refer to something complex

below, all four blocks achieve the same goal

also, `B` and `C` are placeholders for actual expressions

```
-- Haskell
let a = B in C
let id = (\x -> x) in id 5
```

```
# Python
a = B; C
id = lambda x: x; id(5)
```

```
# Python
(lambda a: C) B
(lambda id: id(5)) (lambda x: x)
```

```
; λ-calculus
(\a. C) B
(\id. id 5) (\x. x)
```

Algebraic Data Types

Do yourself a huge favor and go learn about the Curry–Howard correspondence.

fundamental; product types and sum types are the ands and ors of logic

allow us to build types like arrays and integers out of thin air

---

Product Types

```
// Rust
struct Pair<T, U>(T, U);
let some_pair = Pair(1, 2);
let Pair(fst, snd) = some_pair;
// ... ...
```

```
; λ-calculus
(\pair.
  (\fst.
    (\snd.
      ; ...
    ) (\pair. pair (\fst. \snd. snd))
  ) (\pair. pair (\fst. \snd. fst))
) (\fst. \snd. \pair. pair fst snd)
```

we use named bindings for `pair`*,* `fst`*,* `snd`*, as discussed before*

the only logical thing to do with a product type is destructure it; hence:

product types are functions that call their single parameter with several arguments to emulate destructuring

---

Sum Types

```
// Rust
enum Bool { True, False }
let some_bool = Bool::True;
match some_bool {
  Bool::True => // ...
  Bool::False => // ...
}
```

```
; λ-calculus
(\true.
  (\false.
    (\and.
      ; ...
    ) (\lhs. \rhs. lhs rhs lhs)
  ) (\true. \false. false)
) (\true. \false. true)
```

we use named bindings for `true`*,* `false`*,* `and`*, as discussed before*

the only logical thing to do with a sum type is case analysis; hence:

sum types are functions that call one of their several parameters to emulate case analysis

---

Recursive Types

```
// Rust
enum Nat { Succ(Nat), Zero }
use Nat::*;
let three = Succ(Succ(Succ(Zero)));
```

```
; λ-calculus
(\succ.
  (\zero.
    (\pred.
      ; ...
    ) (\nat. \succ. \zero. nat (\x. x) zero)
  ) (\succ. \zero. zero)
) (\nat. \succ. \zero. succ nat)
```

we will be using Scott encoding instead of Church encoding

recursive because `Nat` may contain a `Nat`. I classify them separately because this is where Scott and Church encoding differ

important to understand that the `Succ` variant contains the predecessor

Rust useless but lambda-calculus useful because no numbers in lambda-calculus

---

An Alternative Syntax

```
; λ-calculus
(\succ.
  (\zero.
    (\pred.
      ; ...
    ) (\nat. \succ. \zero. nat (\x. x) zero)
  ) (\succ. \zero. zero)
) (\nat. \succ. \zero. succ nat)
```

indentation and parens can get unwieldy

```
; λ-calculus in Polish notation
.\succ \zero zero \zero
.\nat \succ \zero .nat succ \succ
.\nat \succ \zero .zero .\x x nat \pred
; ...
```

here's a neat idea: why not do lambda-calculus in Polish notation

- `\` *is lambda-abstraction*
- `.` *is application (operands reversed)*

Self-Reference

self-reference useful because necessary condition for recursion

```
# Python
def fact(n):
  return 1 if n == 0 else n * fact(n - 1)
fact
```

the lambda-calculus has no primitive for self-reference. however, can be emulated using a fixed-point combinator

$f(\text{fix } f) = \text{fix } f$

$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

```
; λ-calculus
(\y.
  (\fact.
    ; ...
  ) (y (\fact. \n. n (\p. mul n (fact p)) (succ zero)))
) (\f. (\x. f (x x)) (\x. f (x x)))
```

above we assume `mul`, `succ`, `zero` are defined

y combinator only works with normal-order reduction aka lazy eval. it is normalizing, aka will never loop forever if halting is acheiveable

intuitively, if we give `f` the factorial function, it will return us the factorial function. that's why we need its fixed point, and that's why its fixed point is the factorial function

---

Self-Reference is Powerful

```
-- Haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
fib n = fibs!!n
```

self-reference is not only about recursion

a self-referential list of infinite length?? that's cool

$G(x) = 0 + x + xG(x) + x^2G(x) \Rightarrow G(x) = x/(1 - x - x^2)$

quick aside: this is how you derive the generating function for the Fibonacci sequence

- *`0 :` is $0 +$*
- *`1 :` is $x +$*
- *`fibs` is $x^2G(x)$ (offset by factor of x^2 because `zipWith` begins at index 2)*

- `tail fibs` is $x^2G(x)/x = xG(x)$

then, factor out $G(x)$ and you get the generating function

```
; λ-calculus
(\i. \y.
  (\pair. \fst. \snd.
    (\succ. \zero. \pred.
      (\add.
        (\zipwith.
          (\at.
            (\fibs.
              (\fib.
                fib (succ (succ (succ (succ (succ zero)))))
              ) (at fibs)
            ) (y (\fibs. pair zero (pair (succ zero) (zipwith add (snd fibs)
fibs))))
          ) (y (\at. \l. \n. l (\fst. \snd. n (at snd) fst)))
        ) (y (\zipwith. \op. \l1. \l2. l1 (\fst1. \snd1. l2 (\fst2. \snd2. pair
(op fst1 fst2) (zipwith op snd1 snd2)))))
      ) (y (\add. \lhs. \rhs. rhs (add (succ lhs)) lhs))
    ) (\nat. \succ. \zero. succ nat) (\succ. \zero. zero) (\nat. \succ. \zero.
nat i zero)
  ) (\fst. \snd. \pair. pair fst snd) (\pair. pair (\fst. \snd. fst)) (\pair.
pair (\fst. \snd. snd))
) (\x. x) (\f. (\x. f (x x)) (\x. f (x x)))
```

turns out this also works in the lambda-calculus!

the above computs fib(5)

Type Classes

Consider the following type classes.

```
-- Haskell
class Eq a where
    -- "eq"
    (==) :: a -> a -> Bool

class Eq a => Ord a where
    -- "leq"
    (<=) :: a -> a -> Bool
```

nothing to do with OOP classes. more similar to OOP interfaces (but better)

notice that `Ord` is a subclass of `Eq`; anything orderable must be equatable

Type Class Constraints

```
-- Haskell
pairEq :: (Eq f, Eq s) => (f, s) -> (f, s) -> Bool
pairEq (lhs_fst, lhs_snd) (rhs_fst, rhs_snd) =
  lhs_fst == rhs_fst && lhs_snd == rhs_snd
```

```
// Rust
fn pair_eq<F: Eq, S: Eq>((lhs_fst, lhs_snd): (F, S), (rhs_fst, rhs_snd): (F, S))
-> bool {
  lhs_fst == rhs_fst && lhs_snd == rhs_snd
}
```

```
; λ-calculus
\fst_eq. \snd_eq. \lhs_pair. \rhs_pair.
  lhs_pair (\lhs_fst. \lhs_snd.
    rhs_pair (\rhs_fst. \rhs_snd.
      and (fst_eq lhs_fst rhs_fst) (snd_eq lhs_snd rhs_snd)))
```

can emulate type class constraints by manually passing in the vtable

neat thing: using algebraic data types we can do boolean algebra on type class constraints

---

Type Class Inheritance

```
-- Haskell
instance Ord a => Eq a where
    lhs == rhs = (lhs <= rhs) && (rhs <= lhs)
```

side note: this works because <= is an antisymmetric relation

```
// Rust
impl<T: Ord> PartialEq for T {
  fn eq(&self, other: &Self) -> bool {
    self <= other && other <= self
  }
}
```

```
; λ-calculus
\leq. \lhs. \rhs. and (leq lhs rhs) (leq rhs lhs)
```

can emulate type class inheritance by defining a function that maps the subclass vtable to the superclass vtable

---

SSA is λ-Calculus

```
define i32 @pow(i32 %x, i32 %y) {
start:
  br label %loop_start

loop_start:
  %i.0 = phi i32 [0, %start], [%i.new, %loop]
  %r.0 = phi i32 [1, %start], [%r.new, %loop]
  %done = icmp eq i32 %i.0, %y
  br i1 %done, label %exit, label %loop

loop:
  %r.new = mul i32 %r.0, %x
  %i.new = add i32 %i.0, 1
  br label %loop_start

exit:
  ret i32 %r.0
}
```

all seems a bit detached from reality? there's this IR used in procedural languages called SSA. turns out that SSA is just another name for the lambda-calculus

above is LLVM IR. branch instructions are function application and phi nodes are lambda-abstraction

for more info check out paper below

Andrew W. Appel. *SSA is Functional Programming*

the best way to compile procedural programs is to first convert them to purely functional programs (drop the mic)