

**Background**

- Developed in 1930’s by Alonzo Church
  - studied in logic and computer science

- Test bed for programming languages
  - Simple, Powerful, Extensible

“Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus” (Landin ‘66)

---

**Syntax**

- Three kinds of expressions (terms):
  - $e ::= x$ Variables
  - $\lambda x.e$ Functions ($\lambda$-abstraction)
  - $e_1 e_2$ Application

- $\lambda x.e$ is a one-argument function with body $e$
- $e_1 e_2$ is a function application

- Application associates to the left:
  - $x y z$ means $(x y) z$

- Abstraction extends to the right as far as possible:
  - $\lambda x. \lambda y. z$ means $\lambda x.(\lambda y.(z))$

---

**Examples of Lambda Expressions**

- The identity function:
  - $\lambda x. x$

- A function that given an argument $y$ discards it and returns the identity function:
  - $\lambda y. (\lambda x. x) y$

- A function that given a function $f$ invokes it on the identity function:
  - $f (\lambda x. x)$
Free and Bound Variables

- Variable “shadowing”
  - Different occurrences of var may refer to different values

- E.g., in ML: let x = E in x + (let x = E' in x) + x

- In lambda calculus: \( \lambda x. (\lambda x. x) x \)

Renaming Bound Variables

\( \alpha \)-renaming:

- \( \lambda \)-terms after renaming bound var occurrences
- Considered identical to original

Example: \( \lambda x. x \) is identical to \( \lambda y. y \) and to \( \lambda z. z \)

Rename bound variables so names unique

- E.g., write \( \lambda x. x (\lambda y. y) x \) instead of \( \lambda x. x (\lambda x. x) x \)

- Easy to see the scope of bindings

Substitution

\( [E'/x] E \) : Substitution of \( E' \) for \( x \) in \( E \)

1. Uniquely rename bound vars in \( E \) and \( E' \)
2. Do textual substitution of \( E' \) for \( x \) in \( E \)

Example: \( [y (\lambda x. x)/x] \lambda y. (\lambda x. x) y x \)

1. After renaming: \( [y (\lambda v. v)/x] \lambda z. (\lambda u. u) z x \)
2. After substitution: \( \lambda z. (\lambda u. u) z (y (\lambda v. v)) \)

Informal Semantics

The evaluation of \( (\lambda x. e) e' \)

1. binds \( x \) to \( e' \)
2. evaluates \( e \) with the new binding
3. yields the result of this evaluation

Like \( \text{let } x = e' \text{ in } e \)

Example: \( (\lambda f. f (f e)) g \) evaluates to \( g (g e) \)
Another View of Reduction

The application becomes:

\(\lambda x. x\) \(x\) \(x\) \(e\) \(e'\) \(e'\) \(e'\) \(e\)

Terms can grow substantially by reduction

Examples of Evaluation

- The identity function:
  \((\lambda x. x) E\)
  \(\Rightarrow [E / x] x\)
  \(\Rightarrow E\)

- Another example with the identity:
  \((\lambda f. f (\lambda x. x)) (\lambda x. x)\)
  \(\Rightarrow [\lambda x. x / f] f (\lambda x. x)\)
  \(\Rightarrow [(\lambda x. x) / f] f (\lambda y. y)\)
  \(\Rightarrow (\lambda x. x) (\lambda y. y)\)
  \(\Rightarrow (\lambda y. y) (\lambda x. x)\)
  \(\Rightarrow \lambda y. y\)

Examples of Evaluation

- \((\lambda x. x x)(\lambda y. y y)\)
  \(\Rightarrow [\lambda y. y y / x] x x\)
  \(\Rightarrow (\lambda y. y y)(\lambda y. y y)\)
  \(\Rightarrow (\lambda x. x x)(\lambda y. y y)\)
  \(\Rightarrow \ldots\)
  A non-terminating evaluation!

Next

“Programming” with the \(\lambda\)-Calculus
Programming with the λ-calculus

How does the λ-calculus relate to “real” programming languages?

- Bools / If-then-else?
- Records
- Integers?
- Recursion?
- Functions (well, those we have ...)

Encoding Booleans in λ-calculus

Q: What can we do with a boolean?
A: Make a binary choice

Q: So, how can you view this as a “function”?
A: Bool = fun that takes two choices, returns one
- true =def λx. λy. x
- false =def λx. λy. y
- if E1 then E2 else E3 =def E1 E2 E3

Example: “if true then u else v” is
(λx. λy. x) u v ®β (λy. u) v ®β u

Boolean Operations

Boolean operations: not
Function takes b:

returns function takes x,y:

not =def λb. (λx. λy. b y x)

Boolean operations: or
Function takes b1, b2:

returns function takes x,y:

or =def λb1. λb2. (λx. λy. b1 x (b2 x y))

Programming with the λ-calculus

How does the λ-calculus relate to “real” programming languages?

- Bools / If-then-else?
- Records
- Integers?
- Recursion?
- Functions (well, those we have ...)
Encoding Pairs (and so, Records)

Q: What can we do with a pair?
A: We can select one of its elements

Pair: function takes a bool, returns the left or the right element

\[
\text{mkpair } e_1 e_2 = \lambda b. b e_1 e_2
\]

Note: “pair” encoded as \(\lambda\)-abstraction, “waiting” for bool

\[
\text{fst } p = \text{def } p \text{ true}
\]
\[
\text{snd } p = \text{def } p \text{ false}
\]

Ex: \(\text{fst (mkpair } x y) \mapsto (\text{mkpair } x y) \text{ true} \mapsto \text{true } x y \mapsto x\)

Programming with the \(\lambda\)-calculus

How does the \(\lambda\)-calculus relate to “real” programming languages?

• Bools / If-then-else?
• Records
• Integers?
• Recursion?
• Functions (well, those we have …)

Encoding Natural Numbers

Q: What can we do with a natural number?
A: Iterate a number of times over some function

Nat: function that takes fun \(f\), starting value \(s\):
returns: \(f\) applied to \(s\) a number of times

\[
0 = \text{def } \lambda f. \lambda s. s
\]
\[
1 = \text{def } \lambda f. \lambda s. f s
\]
\[
2 = \text{def } \lambda f. \lambda s. f (f s)
\]

M

Called Church numerals, unary representation

Note: \((n f s)\) : apply \(f\) to \(s\) “\(n\)” times, i.e. \(f^n(s)\)

Operating on Natural Numbers

• Testing equality with 0
  \(\text{iszero } n = \text{def } n (\lambda b. \text{false}) \text{ true}\)
  \(\text{iszero } n = \text{def } \lambda n. (n (\lambda b. \text{false}) \text{ true})\)
• The successor function
  \(\text{succ } n = \text{def } \lambda f. \lambda s. f (n f s)\)
  \(\text{succ } n = \text{def } \lambda f. \lambda s. f (n f s)\)
• Addition
  \(\text{add } n_1 n_2 = \text{def } \lambda n. n_1 n_2 n\) \text{ succ } n\)
• Multiplication
  \(\text{mult } n_1 n_2 n = \text{def } n_1 (n_2 n) 0\)

Ex: Computing with Naturals

What is the result of \texttt{add 0}?

\texttt{(\lambda n_1. \lambda n_2. \ n_1 \ succ \ n_2) \ 0}
\implies \texttt{\lambda n_2. \ 0 \ succ \ n_2}
\implies \texttt{\lambda n_2. \ (\lambda f. \ \lambda s. \ s) \ succ \ n_2}
\implies \texttt{\lambda x. \ x}

Programming with the \(\lambda\)-calculus

How does the \(\lambda\)-calculus relate to “real” programming languages?

• Bools / If-then-else?
• Records
• Integers?
• Recursion?
• Functions (well, those we have …)

Encoding Recursion

• Write a function \texttt{find} that:
  takes “predicate” \(P\), “natural” \(n\) returns:
    smallest natural
  larger than \(n\)
  satisfying \(P\)

\begin{align*}
\texttt{find} \ P \ n & = \text{if } P \ n \ \text{then } n \ \text{else } \texttt{find} \ P \ (\texttt{succ} \ n) \\
\texttt{find} & \text{ satisfies the equation:} \\
\texttt{F} & \text{ is a fixpoint of } \texttt{F}\end{align*}

• Define: \(F = \lambda f. \lambda p. \lambda n. \left[ p \ n \right] \ n \ \left[ f (\texttt{succ} \ n) \right] \)

• A fixpoint of \(F\) is an \(x\) s.t. \(x = F \ x\)

• \texttt{find} is a fixpoint of \(F\)!

- as \texttt{\texttt{find} \ P \ n = \texttt{F} \ \texttt{find} \ P \ n}
- so \texttt{\texttt{find} = \texttt{F} \ \texttt{find}}
The Y-Combinator

Define: \( Y = \text{def} \lambda F. (\lambda y. F(y y)) (\lambda x. F(x x)) \)

- Called the fixpoint combinator as...
  - \( Y F \)
    - \( = (\lambda y. F(y y)) (\lambda x. F(x x)) \)
    - \( = F ((\lambda x. F(x x))(\lambda z. F(z z))) \)
    - \( = F (Y F) \)
  - i.e. \( Y F = Y F \)

- Can get fixpoint for any \( \lambda \)-calculus function

Whoa!

Define: \( F = \lambda f. \lambda p. \lambda n. (p n) n (f p (\text{succ} n)) \)

and: \( \text{find} = Y F \)

What's going on?

\( \text{find} p n \)
  - \( = Y F p n \)
  - \( = F (Y F) p n \)
  - \( = F \text{find} p n \)
  - \( = (p n) n (\text{find} p (\text{succ} n)) \)

Many other fixpoint combinators

Including those that work for CBV

Including Klop's Combinator:

\( Y_k = \text{def} L \)

where:

\( L = \text{def} (\text{this is a fixpoint combinator}) \)

Programming with the \( \lambda \)-calculus

How does the \( \lambda \)-calculus relate to “real” programming languages?

- Bools / If-then-else?
- Records
- Integers?
- Recursion?
- Functions (well, those we have ...)
That's all folks!

- Hope 130 taught you something …
  - … many ways of computational thinking

- Good luck for final
  - On Monday
  - Review Session: Sun 5-7, CSE 4140

- Want more?
  - CSE 230 (Winter 2012)