Lambda Calculus

Edward Z. Yang
Blackboard

e ::= x \mid \lambda x. e \mid e_1 \, e_2

e ::= x
| \text{function}(x) \{ \text{return } e_1 \}
| e_1(e_2)

e ::= x
| \backslash x \to e
| e_1 \, e_2
Somewhere in the ivory tower, someone is cooking.
Still cooking...
Done! And what's left? A very simple language which we like to call the lambda calculus. The language is so simple, it only has ONE language feature: functions. This language will be the subject of our attention today, up in the ivory tower.
As we mentioned earlier, the lambda calculus is the simplest possible programming language which still captures the idea of first-class functions. As such, it is a really good system to study an important syntactic notion that shows up in nearly all languages: binding. The study of capture avoiding substitution is also important in the real world, specifically when one is writing macros or optimizers.

Historically, lambda calculus was important because it was a competing model of computation introduced by Alonzo Church as an alternative to the Turing machine. However, we are not going to talk much about that there.
Another benefit of the simplicity of the lambda calculus is that it is the "base template" for many programming languages: if you want to study a language feature in as pure a context as possible, you might try adding it to the lambda calculus. For example, even in the lambda calculus we can observe the difference between a strict programming language (JavaScript) versus a non-strict language (Haskell).

\[ \land + \text{ evaluation strategy} \]

call-by-value

call-by-name
But the feature that programming language theorists have been most interested in has been type systems, which give structure to the terms of the lambda calculus, and help tame some of the wild and wooly behavior lambda calculus has. We will look at types some more next week, but essentially, if you decide to do programming languages research, the lambda calculus will help you interpret what you are reading in all these papers!

\(\forall + \text{ type system}\)

simply-typed lambda calculus
polymorphic lambda calculus
dependent types
every PL research paper ever
Roadmap

the $\lambda$-calculus

capture-avoiding substitution

evaluation order
Recap

\[ e ::= x \mid \lambda x. e \mid e_1 \, e_2 \]

Example terms:

\[ (\lambda x. (2 + x)) \]

\[ (\lambda x. (2 + x)) \, 5 \quad \Rightarrow \quad 7 \]

\[ (\lambda f. (f \, 3)) \, (\lambda x. (x + 1)) \quad \Rightarrow \quad 4 \]

\( \text{higher order function} \)
Recap: Substitution

\[(\lambda f. \lambda x. f (f x)) (\lambda y. y+1)\]

\[\xrightarrow{\beta} \lambda x. (\lambda y. y+1) (\lambda y. y+1) x\]

\[\xrightarrow{\beta} \lambda x. (\lambda y. y+1) (x+1)\]

\[\xrightarrow{\beta} \lambda x. (x+1) + 1\]
Recap: Closures

\[
((\lambda x. (\lambda y. x)) \ 2) \ 3
\]

\[\xrightarrow{\beta} (\lambda y. 2) \ 3\]

\[\xrightarrow{\beta} 2\]

returned function has \(x\) substituted
Using the \( \lambda \) calculus: Syntax

\[ \lambda x y. e \equiv \lambda x. (\lambda y. e) \]

Left associative application:

\[ f \ x \ y \equiv (f \ x) \ y \neq f \ (x \ y) \]

\[ \lambda x. f \ x \equiv \lambda x. (f \ x) \neq (\lambda x. f) \ x \]

(like Haskell: \( \lambda x \ y \rightarrow e \equiv \lambda x \rightarrow (\lambda y \rightarrow e) \))
Using the λ calculus: Declarations

\[
\begin{align*}
\text{function } f(x) &\equiv \\
&\text{return } x+2; \\
3 &
\end{align*}
\]

\[
\Rightarrow
\]

\[
\lambda f. f(f(3))
\]

\[
\lambda x. x+2
\]

\[
\text{desugar!}
\]

\[
\text{definition of } f
\]

let \( x = e_2 \) in \( e_2 \)  \Rightarrow  \( \lambda x. e_2 \) \( e_1 \)
Bound and Free variables

\((\lambda x. x)\)

Bound Variable (a closed term)

\((\lambda x. y)\)

Free variable (an open term)
Bound and Free variables

α-conversion

\[ (\lambda z . z) \]

name doesn't matter
has no free variables

\[ (\lambda z . y) \]

name matters!
y is a free variable

"I am not a number,
I am a free variable!"
Bound and Free variables

\[
\text{function } (x) \{ \text{return } x + y \}
\]

let \( x = e \) in \( x \)  
Jane hit herself

\[
\int (x + y) \, dx
\]

\[
\forall x. P(x)
\]

\[
\sum_{i} x_i
\]
Bound and Free variables Summary

\[ \text{FV}(x) = \{ x \} \]
\[ \text{FV}(e_1, e_2) = \text{FV}(e_1) \cup \text{FV}(e_2) \]
\[ \text{FV}(\lambda x. e) = \text{FV}(e) \setminus \{ x \} \quad \text{remove } x \text{ from set} \]

\( \alpha \)-conversion: rename bound variables (without capturing free variables)

\( (\lambda x. y) \not\equiv_\alpha (\lambda y. y) \)

\( \alpha \)-equivalence: equality up to \( \alpha \)-conversion
de Bruijn indexes

\((\lambda z. z)\)

name doesn't matter...
de Bruijn indexes

\[(\lambda. \emptyset) \rightarrow \text{so get rid of it!}\]

number of lambdas to count outwards
de Bruijn indexes

\[ \lambda x. \lambda y. x \quad \Rightarrow \quad \lambda. \lambda. 1 \]

\[ (\lambda x. (\lambda y. x) \; x) \quad \Rightarrow \quad \lambda. (\lambda. 1) \; \emptyset \]

*only counts enclosing lambdas*
de Bruijn indexes

\[ \lambda \alpha \lambda y. x \rightarrow \lambda \alpha. 1 \]

structural equivalence

= \alpha-equivalence
Roadmap

the $\lambda$-calculus: binders

capture-avoiding substitution

evaluation order
Substitution is useful

- Evaluation strategy (conceptual, not so great for implementation)

- Optimization/Macros

  can’t run because we don’t know a or b

  \[
  \begin{align*}
  &\text{let } x = a + b \text{ in} \\
  &\hspace{1em} \text{let } a = 7 \text{ in} \\
  &\hspace{2em} x + a
  \end{align*}
  \]

  but would like to inline \( x \)
How do we compute on \( \lambda \)-terms?

This process is called beta-reduction. It operates by taking an application of a lambda and an argument (this application is called a REDEX), and peels away the lambda, substituting all occurrences of its parameter with the expression. The bracket syntax indicates substitution.
Name capture

Recall $let\ x\ =\ e_1\ in\ e_2 \equiv (\lambda x.e_2)\ e_1$

$let\ x\ =\ a\ +\ b\ in$

$let\ a\ =\ 7\ in$  $\Rightarrow$

$x + a$

(let $a = 7$ in $(a+b)+a$

obviously wrong
Name capture

Recall \( \text{let } x = e_1 \text{ in } e_2 \equiv (\lambda x. e_2) \ e_1 \)

\[
\text{let } x = a + b \text{ in } \quad \text{let } s_{796} = 7 \text{ in }
\]
\[
\text{let } a = 7 \text{ in } \quad (a + b) + s_{769}
\]

Some "fresh" new variable
Capture-avoiding substitution

Idea: Rename bound variables \((\alpha\text{-convert them})\) so that they don't capture free variables.
Capture-avoiding substitution

\[ x[x \mapsto e] = e \]
\[ y[x \mapsto e] = y \]
\[ (e_1 e_2)[x \mapsto e] = e_1[x \mapsto e] \ e_2[x \mapsto e] \]
\[ (\lambda x. e_1)[x \mapsto e] = \lambda x. e_1 \]
\[ (\lambda x. e_1)[y \mapsto e] = \lambda x. e_2[y \mapsto e] \text{ if } x \notin \text{FV}(e) \]
\[ (\lambda y. e_1)[x \mapsto e] = \lambda y'. e_1[y \mapsto y'][x \mapsto e] \]

where \( y' \) is fresh
Capture-avoiding substitution

\[ x[x \mapsto e] = e \]
\[ y[x \mapsto e] = y \]
\[ (e_1 e_2)[x \mapsto e] = e_1[x \mapsto e] \ e_2[x \mapsto e] \]
\[ (\lambda x. e_1)[x \mapsto e] = \lambda x. e_1 \]
\[ (\lambda x. e_1)[y \mapsto e] = \lambda x. e_2[y \mapsto e] \text{ if } x \notin \text{FV}(e) \]
\[ (\lambda y. e_1)[x \mapsto e] = \lambda y'. e_1[y \mapsto y'][x \mapsto e] \]

where \[ y' \notin \{x\} \cup \text{FV}(e_1) \cup \text{FV}(e) \]
Summary: Equational theory

\[ \lambda x. e \equiv_\alpha \lambda y. e[x \mapsto y] \quad \text{where } y \notin \text{FV}(e) \]

\[ (\lambda x. e_1) e_2 \equiv_\beta e_1[x \mapsto e_2] \]

\[ \lambda x. e \, x \equiv_\eta e \quad \text{where } x \notin \text{FV}(e) \]
Roadmap

- the Λ-calculus: binders
- capture-avoiding substitution
- evaluation order
\((\lambda x. x) ((\lambda y. y) z)\)
\[(\lambda x. x) \ (\lambda y. y) \ z\]
Does it matter?
Does it matter?

Church-Rosser Theorem:

"If you reduce to a normal form, it doesn't matter what order you do the reductions."
Does it matter?

Church-Rosser Theorem:

"If you reduce to a normal form, it doesn't matter what order you do the reductions."
A curious lambda term called $\Omega$

$$(\forall x. x x) (\forall x. x x)$$
A curious lambda term called $\Omega$

$$(x \ x)[x \mapsto (\lambda x. \ x \ x)]$$
A curious lambda term called $\Omega$

$$(\lambda x. x x) (\lambda x. x x)$$

Deja vu!
\( \Omega \) has no normal form
$(\forall x . y) \Omega$
$(\forall x. y) \Omega 
\xrightarrow{y}$
But WAIT. There is another reduction we can do. That's the reduction of Omega to itself.
ok, evaluation order might be important
Call-by-value (ala JavaScript)

\[ e_1, e_2 \]

\[ \xrightarrow{\beta} (\lambda x. e'_1) e_2 \]

\[ \xrightarrow{\beta} (\lambda x. e'_1) n \]

\[ \xrightarrow{\beta} e'_1 [x \mapsto n] \rightarrow^* \]

\[ \rightarrow^* \]
Call-by-value

\((\lambda x. y) \Omega \rightarrow^\beta (\lambda x. y) \Omega \rightarrow\)
BAM!
Call-by-name  (ala Haskell

\[ e_1 \ e_2 \]

\[ \Rightarrow^*_{\beta} (\lambda x. e'_1) \ e_2 \]

\[ (\text{skip}) - \]

\[ \Rightarrow_{\beta} e'_1 [x_1 \rightarrow e_2] \Rightarrow^*_{\beta} \ldots \]
Call-by-name

\((\text{\textbackslash}lambda x. y) \Omega \rightarrow^\beta Y\)
only do what is absolutely necessary!
Summary

A-term may have many redexes. Evaluation order says which redex to evaluate. Evaluation not guaranteed to find normal form.

CBV: evaluate function & arguments before β-reducing.
CBN: evaluate function, then β-reduce.
Roadmap

the $\Lambda$-calculus: binders
capture-avoiding substitution
evaluation order
Conclusion

\(\lambda\text{-calculus} = \text{Formal System}\)
Conclusion

\[ e ::= \lambda x. e \mid e e \mid e \]

binders show up everywhere!

\[
\begin{align*}
\text{true} &= \lambda x. \lambda y. x \\
\text{false} &= \lambda x. \lambda y. y \\
\text{cond} &= \lambda b. \lambda t. \lambda f. (b \ f \ (t \ f))
\end{align*}
\]

\[
Y = \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x))
\]
Extra topics

- Locally nameless style
- Other evaluation strategies
- Operational semantics