COMP2111 Week 9
Term 1, 2022
Lambda Calculus and Higher-Order Logic
Admin announcements

**myExperience**  
The myExperience survey is up. Link on the website. Please take a moment to fill it in, it helps a lot! :}

**Final exam**  
The exam is a 24-hour take-home exam. Tuesday May 10th 8AM–Wednesday May 11th 8AM. When the time starts, you will receive the exam questions in your UNSW inbox. Submission via pdf through give. You can ask questions on Ed during the exam (make your threads initially private). I sleep at night, so try to ask early. The intended workload is <4 hours; the 24 hours are just to give you flexibility. We can’t stop you from using more than 4 hours...
Admin announcements

myExperience  ● The myExperience survey is up. Link on the website. Please take a moment to fill it in, it helps a lot! :)

Final exam  ● The exam is a 24-hour take-home exam.
Admin announcements

**myExperience**
- The myExperience survey is up. Link on the website. Please take a moment to fill it in, it helps a lot! :)

**Final exam**
- The exam is a 24-hour take-home exam.
- Tuesday May 10th 8AM–Wednesday May 11th 8AM.
Admin announcements

**myExperience**  
The myExperience survey is up. Link on the website. Please take a moment to fill it in, it helps a lot! :)

**Final exam**  
The exam is a 24-hour take-home exam.  
Tuesday May 10th 8AM–Wednesday May 11th 8AM.  
When the time starts, you will receive the exam questions in your UNSW inbox.
**Admin announcements**

**myExperience**
- The myExperience survey is up. Link on the website. Please take a moment to fill it in, it helps a lot! :)

**Final exam**
- The exam is a 24-hour take-home exam.
- Tuesday May 10th 8AM–Wednesday May 11th 8AM.
- When the time starts, you will receive the exam questions in your UNSW inbox.
- Submission via pdf through give.
- You can ask questions on Ed during the exam (make your threads initially private). I sleep at night, so try to ask early.
Admin announcements

**myExperience**
- The myExperience survey is up. Link on the website. Please take a moment to fill it in, it helps a lot! :)

**Final exam**
- The exam is a 24-hour take-home exam.
- Tuesday May 10th 8AM–Wednesday May 11th 8AM.
- When the time starts, you will receive the exam questions in your UNSW inbox.
- Submission via pdf through give.
- You can ask questions on Ed during the exam (make your threads initially private). I sleep at night, so try to ask early.
Admin announcements

**myExperience**
- The myExperience survey is up. Link on the website. Please take a moment to fill it in, it helps a lot! :)

**Final exam**
- The exam is a 24-hour take-home exam.
- Tuesday May 10th 8AM–Wednesday May 11th 8AM.
- When the time starts, you will receive the exam questions in your UNSW inbox.
- Submission via pdf through give.
- You can ask questions on Ed during the exam (make your threads initially private). I sleep at night, so try to ask early.
- The intended workload is < 4 hours; the 24 hours are just to give you flexibility. We can’t stop you from using more than 4 hours...
Detour: program evaluation

```c
int f (int n) {
    if n = 0 then 1 else 0
}
```

Let’s evaluate this function step by step:

\[ f(0) \rightarrow \]

Detour: program evaluation

```c
int f (int n) {
    if n = 0 then 1 else 0
}
```

Let’s evaluate this function step by step:

\[
\begin{align*}
    f(0) & \quad \rightarrow \quad (\text{function application}) \\
    \text{if } 0 = 0 \text{ then } 1 \text{ else } 0 & \quad \rightarrow \\
\end{align*}
\]

*Function application*: substitute the argument into the body.
Detour: program evaluation

```c
int f (int n) {
    if n == 0 then 1 else 0
}
```

Let’s evaluate this function step by step:

$f(0) \quad \rightarrow \quad \text{(function application)}$

if $0 = 0$ then 1 else 0 $\quad \rightarrow \quad \text{(equality comparison)}$

if true then 1 else 0 $\quad \rightarrow \quad$

*Function application*: substitute the argument into the body.
Detour: program evaluation

```c
int f (int n) {
    if n == 0 then 1 else 0
}
```

Let’s evaluate this function step by step:

- \( f(0) \)
- if 0 = 0 then 1 else 0 \( \rightarrow \) (equality comparison)
- if true then 1 else 0 \( \rightarrow \) (if true)
- 1

*Function application*: substitute the argument into the body.
Detour: program evaluation

The λ-calculus formalises this kind of step-by-step evaluation of program expressions...
Detour: program evaluation

The λ-calculus formalises this kind of step-by-step evaluation of program expressions...

...for a tiny language where the only operation is function application. Nonetheless, λ-calculus is Turing complete. How can that be?
\(\lambda\)-calculus

Alonzo Church

- lived 1903–1995
- supervised people like Alan Turing, Stephen Kleene
- famous for Church-Turing thesis, lambda calculus, first undecidability results
- invented \(\lambda\) calculus in 1930’s
\textbf{\(\lambda\)-calculus}

\textbf{Alonzo Church}

- lived 1903–1995
- supervised people like Alan Turing, Stephen Kleene
- famous for Church-Turing thesis, lambda calculus,
  first undecidability results
- invented \(\lambda\) calculus in 1930's

\textbf{\(\lambda\)-calculus}

- originally meant as foundation of mathematics
- important applications in theoretical computer science
- foundation of computability and functional programming
Basic intuition:

instead of \( f(x) = x + 5 \)
write \( f = \lambda x. x + 5 \)
untyped $\lambda$-calculus

Basic intuition:

\[ f(x) = x + 5 \]

Instead of

Write

\[ f = \lambda x. x + 5 \]

- a term
Untyped λ-calculus

Basic intuition:

instead of \( f(x) = x + 5 \)
write \( f = \lambda x. x + 5 \)

\( \lambda x. x + 5 \)

- a term
- a nameless function
untyped $\lambda$-calculus

Basic intuition:

instead of $f(x) = x + 5$
write $f = \lambda x. \ x + 5$

$\lambda x. \ x + 5$

- a term
- a nameless function
- that adds 5 to its parameter
Function Application

For applying arguments to functions

instead of \( f(a) \)
write \( f\ a \)

Example:

\((\lambda x. x + 5) a\) evaluates to \((a + 5)\)
Function Application

For applying arguments to functions

instead of \( f(a) \)
write \( f \ a \)

Example: \( (\lambda x. \ x + 5) \ a \)
Function Application

For applying arguments to functions

instead of \( f(a) \)
write \( f \ a \)

Example: \( (\lambda x. \ x + 5) \ a \)

Evaluating: in \( (\lambda x. \ t) \ a \) replace \( x \) by \( a \) in \( t \)
(computation!)
Function Application

For applying arguments to functions

instead of $f(a)$
write $f\ a$

Example: $(\lambda x. x + 5)\ a$

Evaluating: in $(\lambda x. t)\ a$ replace $x$ by $a$ in $t$
(computation!)

Example: $(\lambda x. x + 5)\ (a + b)$ evaluates to $(a + b) + 5$
That’s it!
That’s it!

Now Formally
Terms: \[ t ::= v \mid c \mid (t \ t) \mid (\lambda x. \ t) \]

\[ v, x \in V, \quad c \in C, \quad V, C \text{ sets of names} \]
Syntax

Terms: \[ t ::= v \mid c \mid (t \ t) \mid (\lambda x. \ t) \]

\( v, x \in V, \quad c \in C, \quad V, C \) sets of names

- \( v, x \) variables
- \( c \) constants
- \( (t \ t) \) application
- \( (\lambda x. \ t) \) abstraction
Conventions

- leave out parentheses where possible
- list variables instead of multiple $\lambda$

Example: instead of $(\lambda y. (\lambda x. (x y)))$ write $\lambda y \ x. \ x \ y$
Conventions

- leave out parentheses where possible
- list variables instead of multiple \( \lambda \)

**Example:** instead of \((\lambda y. (\lambda x. (x y))))\) write \(\lambda y \ x. \ x \ y\)

**Rules:**
- list variables: \(\lambda x. (\lambda y. t) = \lambda x y. t\)
- application binds to the left: \(x \ y \ z = (x \ y) \ z \neq x \ (y \ z)\)
- abstraction binds to the right:
  \(\lambda x. x \ y = \lambda x. (x \ y) \neq (\lambda x. x) \ y\)
- leave out outermost parentheses
Getting used to the Syntax

Example:
\[ \lambda x \; y \; z. \; x \; z \; (y \; z) = \]
Getting used to the Syntax

Example:
\[ \lambda x \ y \ z. \ x \ z \ (y \ z) = \]
\[ \lambda x \ y \ z. \ (x \ z) \ (y \ z) = \]
Getting used to the Syntax

Example:
\[ \lambda x \ y \ z. \ x \ z \ (y \ z) = \]
\[ \lambda x \ y \ z. \ (x \ z) \ (y \ z) = \]
\[ \lambda x \ y \ z. \ ((x \ z) \ (y \ z)) = \]
Getting used to the Syntax

Example:
\[ \lambda x \ y \ z. \ x \ z \ (y \ z) = \]
\[ \lambda x \ y \ z. \ (x \ z) \ (y \ z) = \]
\[ \lambda x \ y \ z. \ ((x \ z) \ (y \ z)) = \]
\[ \lambda x. \ \lambda y. \ \lambda z. \ ((x \ z) \ (y \ z)) = \]
Getting used to the Syntax

Example:
\[ \lambda x \ y \ z. \ x \ z \ (y \ z) = \]
\[ \lambda x \ y \ z. \ (x \ z) \ (y \ z) = \]
\[ \lambda x \ y \ z. \ ((x \ z) \ (y \ z)) = \]
\[ \lambda x. \ \lambda y. \ \lambda z. \ ((x \ z) \ (y \ z)) = \]
\[ (\lambda x. \ (\lambda y. \ (\lambda z. \ ((x \ z) \ (y \ z))))) \]
Encoding numbers

Remember Peano arithmetic?

\[
\begin{align*}
0 & \equiv Z \\
1 & \equiv S(Z) \\
2 & \equiv S(S(Z))
\end{align*}
\]

In the \(\lambda\)-calculus we have Church numerals:

\[
\begin{align*}
0 & \equiv \lambda f \cdot x. \ x \\
1 & \equiv \lambda f \cdot x. \ f \ x \\
2 & \equiv \lambda f \cdot x. \ f(f \ x)
\end{align*}
\]
Encoding numbers

0 ≡ λf. x. x
1 ≡ λf. x. f x
2 ≡ λf. x. f(f x)

We encode a number \( n \) as a function...
Encoding numbers

0 \equiv \lambda f \ x. \ x
1 \equiv \lambda f \ x. \ f \ x
2 \equiv \lambda f \ x. \ f(f \ x)

- We encode a number \( n \) as a function...
- ..that accepts two arguments, \( f \) and \( x \)...
We encode a number $n$ as a function...
..that accepts two arguments, $f$ and $x$...
..and returns the result of applying $f$ to $x$ $n$ times.
Encoding numbers

0 \equiv \lambda f \, x. \, x
1 \equiv \lambda f \, x. \, f \, x
2 \equiv \lambda f \, x. \, f(f \, x)

Here’s the successor function (given $m$ return $m + 1$):

$$\lambda m. \, \lambda f \, x. \, f \, (m \, f \, x)$$
Encoding numbers

\[\begin{align*}
0 & \equiv \lambda f \ x. \ x \\
1 & \equiv \lambda f \ x. \ f \ x \\
2 & \equiv \lambda f \ x. \ f(f \ x)
\end{align*}\]

Here’s addition (given \(m\) and \(n\) return \(m + 1\)):

\[\lambda m \ n. \ \lambda f \ x. \ m \ f \ (n \ f \ x)\]
We encode booleans as binary functions. \texttt{true} returns the first argument, \texttt{false} returns the second argument.
We encode booleans as binary functions. `true` returns the first argument, `false` returns the second argument.
We encode booleans as binary functions. \texttt{true} returns the first argument, \texttt{false} returns the second argument.
 Encoding booleans

true ≡ λx y. x
false ≡ λx y. y
¬b ≡ λb. b false true

if z then x else y ≡

We encode booleans as binary functions. true returns the first argument, false returns the second argument.
We encode booleans as binary functions. \texttt{true} returns the first argument, \texttt{false} returns the second argument.
Encoding booleans

\[
\begin{align*}
\text{true} & \equiv \lambda x\ y.\ x \\
\text{false} & \equiv \lambda x\ y.\ y \\
\neg b & \equiv \lambda b.\ b\ \text{false}\ \text{true} \\
\text{if } z \text{ then } x \text{ else } y & \equiv \lambda z\ x\ y.\ z\ x\ y \\
isZero(n) & \equiv \\
\end{align*}
\]

We encode booleans as binary functions. true returns the first argument, false returns the second argument.
Encoding booleans

true ≡ λx y. x
false ≡ λx y. y
¬b ≡ λb. b false true
if z then x else y ≡ λz x y. z x y
isZero(n) ≡ λn. n (λx. false) true

We encode booleans as binary functions. true returns the first argument, false returns the second argument.
Detour revisited

\text{int } f (\text{int } n) \{ \\
\quad \text{if } n = 0 \text{ then } 1 \text{ else } 0 \\
\}

This program can be encoded in \(\lambda\)-calculus as follows:

\lambda n. \\
\quad (n (\lambda x. (\lambda x y. y)) (\lambda x y. x)) \\
\quad (\lambda f x. f x) \\
\quad (\lambda f x. x)
Detour revisited

```c
int f (int n) {
    if n = 0 then 1 else 0
}
```

This program can be encoded in $\lambda$-calculus as follows:

$$
\lambda n. \\
(n (\lambda x. (\lambda x y. y)) (\lambda x y. x)) \text{ if } isZero(n) \text{ then } \\
(\lambda f x. f x) \\
(\lambda f x. x)
$$
Detour revisited

int f (int n) {
    if n = 0 then 1 else 0
}

This program can be encoded in $\lambda$-calculus as follows:

$$
\lambda n. \\
(n (\lambda x. (\lambda x y. y)) (\lambda x y. x)) \text{ if isZero}(n) \text{ then } \\
(\lambda f x. f x) \text{ else } \\
(\lambda f x. x) 
$$
Detour revisited

int f (int n) {
    if n = 0 then 1 else 0
}

This program can be encoded in λ-calculus as follows:

λn.
    (n (λx. (λx y. y)) (λx y. x)) if isZero(n) then
     (λf x. f x) 1
    (λf x. x) else 0
Computation

**Intuition:** replace parameter by argument
   this is called $\beta$-reduction

**Example**

$$(\lambda x \ y. \ f \ (y \ x)) \ 5 \ (\lambda x. \ x) \rightarrow_\beta$$
**Computation**

**Intuition:** replace parameter by argument
this is called $\beta$-reduction

**Example**

$$(\lambda x \ y. \ f \ (y \ x)) \ 5 \ (\lambda x. \ x) \Rightarrow_{\beta}$$

$$(\lambda y. \ f \ (y \ 5)) \ (\lambda x. \ x) \Rightarrow_{\beta}$$
Computation

Intuition: replace parameter by argument
this is called $\beta$-reduction

Example

$$(\lambda x \ y \ . \ f \ (y \ x)) \ 5 \ (\lambda x . \ x) \rightarrow_\beta$$
$$(\lambda y . \ f \ (y \ 5)) \ (\lambda x . \ x) \rightarrow_\beta$$
$$f \ ((\lambda x . \ x) \ 5) \rightarrow_\beta$$
Computation

**Intuition:** replace parameter by argument
this is called $\beta$-reduction

**Example**

\[
\begin{align*}
(\lambda x \ y. \ f \ (y \ x)) \ 5 & \rightarrow_{\beta} (\lambda x. \ x) \\
(\lambda y. \ f \ (y \ 5)) \ (\lambda x. \ x) & \rightarrow_{\beta} \\
f \ ((\lambda x. \ x) \ 5) & \rightarrow_{\beta} \\
f \ 5 & \rightarrow_{\beta}
\end{align*}
\]
Defining Computation

$\beta$ reduction:

\[
\begin{align*}
(\lambda x. s) t & \rightarrow_\beta s[x \leftarrow t] \\
\lambda x. s & \rightarrow_\beta \lambda x. s'
\end{align*}
\]

\[
\begin{align*}
t & \rightarrow_\beta t' \\
(s \; t) & \rightarrow_\beta (s \; t') \\
\end{align*}
\]

\[
\begin{align*}
s & \rightarrow_\beta s' \\
\lambda x. s & \rightarrow_\beta \lambda x. s'
\end{align*}
\]

Still to do: define $s[x \leftarrow t]$
Defining Computation

$\beta$ reduction:

\[
\frac{(\lambda x. s) t \reduce{\beta} s[x \leftarrow t]}{(\lambda x. s) t \reduce{\beta} s[x \leftarrow t]}
\]

\[
\frac{(s t) \reduce{\beta} (s' t)}{(s t) \reduce{\beta} (s' t)}
\]

\[
\frac{t \reduce{\beta} t'}{(s t) \reduce{\beta} (s t')}
\]

\[
\frac{t \reduce{\beta} t'}{(s t) \reduce{\beta} (s t')}
\]

\[
\frac{s \reduce{\beta} s'}{(s t) \reduce{\beta} (s' t)}
\]

\[
\frac{s \reduce{\beta} s'}{(s t) \reduce{\beta} (s' t)}
\]

\[
\frac{\lambda x. s \reduce{\beta} \lambda x. s'}{\lambda x. s \reduce{\beta} \lambda x. s'}
\]

Still to do: define $s[x \leftarrow t]$
Defining Substitution

Easy concept. Small problem: variable capture.

**Example:** \((\lambda x. x z)[z \leftarrow x]\)

We do not want: \((\lambda x. x x)[z \leftarrow x]\)

So, solution is: rename bound variables.
Defining Substitution

Easy concept. Small problem: variable capture.

Example: \((\lambda x. x \ z)[z \leftarrow x]\)

We do not want: \((\lambda x. x \ x)\) as result.

What do we want?
Defining Substitution

Easy concept. Small problem: variable capture.

Example: \((\lambda x. x z)[z \leftarrow x]\)

We do not want: \((\lambda x. x x)\) as result.

What do we want?

In \((\lambda y. y z) [z \leftarrow x]\) = \((\lambda y. y x)\) there would be no problem.

So, solution is: rename bound variables.
Free Variables

Bound variables: in \((\lambda x. t)\), \(x\) is a bound variable.
Free Variables

Bound variables: in $(\lambda x. t)$, $x$ is a bound variable.

Free variables $FV$ of a term:

$FV(x) = \{x\}$
$FV(c) = \{\}$
$FV(s \ t) = FV(s) \cup FV(t)$
$FV(\lambda x. t) = FV(t) \setminus \{x\}$

Example: $FV(\lambda x. (\lambda y. (\lambda x. x) y) y x)$
Free Variables

Bound variables: in \((\lambda x. \ t)\), \(x\) is a bound variable.

Free variables \(FV\) of a term:

- \(FV(x) = \{x\}\)
- \(FV(c) = \{\}\)
- \(FV(st) = FV(s) \cup FV(t)\)
- \(FV(\lambda x. \ t) = FV(t) \setminus \{x\}\)

Example: \(FV(\lambda x. (\lambda y. (\lambda x. x) y) \ y \ x) = \{y\}\)
**Free Variables**

**Bound variables:** in \((\lambda x. \ t)\), \(x\) is a bound variable.

**Free variables** \(FV\) of a term:

\[
\begin{align*}
FV \ (x) & = \{x\} \\
FV \ (c) & = \{} \\
FV \ (s \ t) & = FV(s) \cup FV(t) \\
FV \ (\lambda x. \ t) & = FV(t) \setminus \{x\}
\end{align*}
\]

**Example:** \(FV(\ \lambda x. \ (\lambda y. \ (\lambda x. \ x) \ y) \ y \ x \ ) = \{y\}\)

Term \(t\) is called **closed** if \(FV(t) = \{}\)
Free Variables

Bound variables: in \((\lambda x. \ t)\), \(x\) is a bound variable.

Free variables \(FV\) of a term:

\[
\begin{align*}
FV\ (x) & = \{x\} \\
FV\ (c) & = \{\} \\
FV\ (s\ t) & = FV(s) \cup FV(t) \\
FV\ (\lambda x.\ t) & = FV(t) \setminus \{x\}
\end{align*}
\]

Example: \(FV(\ \lambda x.\ (\lambda y.\ (\lambda x.\ x)\ y)\ y\ x)\ = \{y\}\)

Term \(t\) is called closed if \(FV(t) = \{\}\)

The substitution example, \((\lambda x.\ x\ z)[z \leftarrow x]\), is problematic because the bound variable \(x\) is a free variable of the replacement term “\(x\)”.

Substitution

\[
x [x \leftarrow t] = t \\
y [x \leftarrow t] = y \quad \text{if } x \neq y \\
c [x \leftarrow t] = c \\
(s_1 \ s_2) [x \leftarrow t] =
\]
Substitution

\[ x \ [x \leftarrow t] = t \]
\[ y \ [x \leftarrow t] = y \quad \text{if} \ x \neq y \]
\[ c \ [x \leftarrow t] = c \]
\[ (s_1 \ s_2) \ [x \leftarrow t] = (s_1[x \leftarrow t] \ s_2[x \leftarrow t]) \]
\[ (\lambda x. \ s) \ [x \leftarrow t] = \]
**Substitution**

\[
x [x \leftarrow t] = t
y [x \leftarrow t] = y \quad \text{if } x \neq y
c [x \leftarrow t] = c
\]

\[
(s_1 \; s_2) [x \leftarrow t] = (s_1[x \leftarrow t] \; s_2[x \leftarrow t])
\]

\[
(\lambda x. \; s) [x \leftarrow t] = (\lambda x. \; s)
(\lambda y. \; s) [x \leftarrow t] =
\]
Substitution

\[ x[x \leftarrow t] = t \]
\[ y[x \leftarrow t] = y \quad \text{if } x \neq y \]
\[ c[x \leftarrow t] = c \]

\[(s_1 s_2)[x \leftarrow t] = (s_1[x \leftarrow t] s_2[x \leftarrow t]) \]

\[(\lambda x. s)[x \leftarrow t] = (\lambda x. s) \quad \text{if } x \neq y \text{ and } y \notin FV(t) \]
\[(\lambda y. s)[x \leftarrow t] = (\lambda y. s[x \leftarrow t]) \]
\[(\lambda y. s)[x \leftarrow t] = \]
Substitution

\[
\begin{align*}
x \ [x \leftarrow t] & = t \\
y \ [x \leftarrow t] & = y & \text{if } x \neq y \\
c \ [x \leftarrow t] & = c \\
(s_1 \ s_2) \ [x \leftarrow t] & = (s_1[x \leftarrow t] \ s_2[x \leftarrow t]) \\
(\lambda x. \ s) \ [x \leftarrow t] & = (\lambda x. \ s) & \text{if } x \neq y \text{ and } y \notin FV(t) \\
(\lambda y. \ s) \ [x \leftarrow t] & = (\lambda y. \ s[x \leftarrow t]) & \text{if } x \neq y \\
(\lambda y. \ s) \ [x \leftarrow t] & = (\lambda z. \ s[y \leftarrow z][x \leftarrow t]) & \text{if } x \neq y \text{ and } z \notin FV(t) \cup FV(s)
\end{align*}
\]
Substitution Example

\[(x \ (\lambda x. \ x) \ (\lambda y. \ z \ x))[x \leftarrow y]\]
Substitution Example

\[
(x (\lambda x. x) (\lambda y. z x))[x \leftarrow y] =
(x[x \leftarrow y]) (((\lambda x. x)[x \leftarrow y]) ((\lambda y. z x)[x \leftarrow y])
\]
Substitution Example

\[
(x \ (\lambda x. \ x) \ (\lambda y. \ z \ x))[x ← y] \\
= \ (x[x ← y]) \ (\ (\lambda x. \ x)[x ← y]) \ (\ (\lambda y. \ z \ x)[x ← y]) \\
= \ y \ (\lambda x. \ x) \ (\lambda y'. \ z \ y)
\]
**$\alpha$ Conversion**

Bound names are irrelevant:
$\lambda x. \ x$ and $\lambda y. \ y$ denote the same function.

**$\alpha$ conversion:**
$s =_{\alpha} t$ means $s = t$ up to renaming of bound variables.
$\alpha$ Conversion

Bound names are irrelevant:
$\lambda x. \ x$ and $\lambda y. \ y$ denote the same function.

$\alpha$ conversion:
$s =_\alpha t$ means $s = t$ up to renaming of bound variables.

Formally:

\[
\frac{y \notin FV(t)}{(\lambda x. \ t) \to_\alpha (\lambda y. \ t[x \leftarrow y])}
\]

\[
\frac{s \to_\alpha s'}{(s \ t) \to_\alpha (s' \ t)}
\]

\[
\frac{t \to_\alpha t'}{(s \ t) \to_\alpha (s' \ t')}
\]

\[
\frac{s \to_\alpha s'}{\lambda x. \ s \to_\alpha (\lambda x. \ s')}
\]
α Conversion

Bound names are irrelevant:
λx. x and λy. y denote the same function.

α conversion:
s =α t means s = t up to renaming of bound variables.

Formally:

\[
\begin{align*}
&\quad y \notin FV(t) \\
\frac{y \notin FV(t)}{(\lambda x. \ t) \longrightarrow_\alpha (\lambda y. \ t[x \leftarrow y])} \\
&\quad s \longrightarrow_\alpha s' \\
\frac{(s \ t) \longrightarrow_\alpha (s' \ t)}{t \longrightarrow_\alpha t'} \\
\frac{(s \ t) \longrightarrow_\alpha (s \ t')}{(s \ t) \longrightarrow_\alpha (s \ t')}
\end{align*}
\]

s =α t iff s \longrightarrow^*_\alpha t

(\longrightarrow^*_\alpha = reflexive, transitive closure of \longrightarrow_\alpha = multiple steps)
α Conversion

Examples:
\[ x (\lambda x\ y.\ x\ y) \]
\( \alpha \) Conversion

Examples:

\[ x \ (\lambda x \ y. \ x \ y) \]
\[ =_{\alpha} \ x \ (\lambda y \ x. \ y \ x) \]
α Conversion

Examples:

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

\[ =_\alpha x (\lambda y \ x. \ y \ x) \]

\[ =_\alpha x (\lambda z \ y. \ z \ y) \]
\( \alpha \) Conversion

Examples:

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

\[ \equiv_\alpha \ x \ (\lambda y \ x . \ y \ x) \]

\[ \equiv_\alpha \ x \ (\lambda z \ y . \ z \ y) \]

\[ \not\equiv_\alpha \ z \ (\lambda z \ y . \ z \ y) \]
\textbf{\(\alpha\) Conversion}

\textbf{Examples:}

\begin{align*}
& x \ (\lambda x \ y. \ x \ y) \\
\equiv_{\alpha} & \ x \ (\lambda y \ x. \ y \ x) \\
\equiv_{\alpha} & \ x \ (\lambda z \ y. \ z \ y) \\
\not\equiv_{\alpha} & \ z \ (\lambda z \ y. \ z \ y) \\
\not\equiv_{\alpha} & \ z \ (\lambda x \ x. \ x \ x)
\end{align*}
We have defined $\beta$ reduction: $\rightarrow^\beta$

Some notation and concepts:

- **$\beta$ conversion**: $s =^\beta t$ iff $\exists n. s \rightarrow^\beta n \land t \rightarrow^\beta n$
Back to $\beta$

We have defined $\beta$ reduction: $\rightarrow^\beta$

Some notation and concepts:

- $\beta$ conversion: $s =^\beta t$ iff $\exists n. s \rightarrow^*_\beta n \land t \rightarrow^*_\beta n$
- $t$ is reducible if there is an $s$ such that $t \rightarrow^\beta s$
Back to $\beta$

We have defined $\beta$ reduction: $\rightarrow^\beta$

Some notation and concepts:

- $\beta$ **conversion**: $s =^\beta t$ iff $\exists n. s \rightarrow^\beta n \land t \rightarrow^\beta n$
- $t$ is **reducible** if there is an $s$ such that $t \rightarrow^\beta s$
- $(\lambda x. s) t$ is called a **redex** (reducible expression)
We have defined $\beta$ reduction: $\rightarrow_\beta$

Some notation and concepts:

- $\beta$ conversion: $s =_\beta t$ iff $\exists n. s \rightarrow_\beta^n n \land t \rightarrow_\beta^n n$
- $t$ is reducible if there is an $s$ such that $t \rightarrow_\beta s$
- $(\lambda x. s) t$ is called a redex (reducible expression)
- $t$ is reducible iff it contains a redex
Back to $\beta$

We have defined $\beta$ reduction: $\rightarrow_{\beta}$

Some notation and concepts:

- $\beta$ conversion: $s =_{\beta} t$ iff $\exists n. s \rightarrow_{\beta}^* n \land t \rightarrow_{\beta}^* n$
- $t$ is **reducible** if there is an $s$ such that $t \rightarrow_{\beta} s$
- $(\lambda x. s) t$ is called a **redex** (reducible expression)
- $t$ is reducible iff it contains a redex
- if it is not reducible, $t$ is in **normal form**
Does every $\lambda$ term have a normal form?

Example:

$$(\lambda x. x x) (\lambda x. x x) \xrightarrow{\beta}$$
Does every $\lambda$ term have a normal form?

Example:

$$(\lambda x. x x) (\lambda x. x x) \rightarrow^\beta$$

$$(\lambda x. x x) (\lambda x. x x) \rightarrow^\beta$$
Does every \( \lambda \) term have a normal form?

No!

Example:

\[
(\lambda x. x x) (\lambda x. x x) \rightarrow_{\beta} \\
(\lambda x. x x) (\lambda x. x x) \rightarrow_{\beta} \\
(\lambda x. x x) (\lambda x. x x) \rightarrow_{\beta} \ldots
\]
Does every $\lambda$ term have a normal form?

No!

Example:

$$(\lambda x. x \; x) \; (\lambda x. x \; x) \rightarrow^\beta$$

$$(\lambda x. x \; x) \; (\lambda x. x \; x) \rightarrow^\beta$$

$$(\lambda x. x \; x) \; (\lambda x. x \; x) \rightarrow^\beta \ldots$$

(but: $$(\lambda y. y) \; (\lambda x. x \; x) \; (\lambda x. x \; x) \rightarrow^\beta \; (\lambda y. y)$$)
Does every $\lambda$ term have a normal form?

No!

Example:

$$(\lambda x. x x) (\lambda x. x x) \rightarrow_\beta$$
$$(\lambda x. x x) (\lambda x. x x) \rightarrow_\beta$$
$$(\lambda x. x x) (\lambda x. x x) \rightarrow_\beta \ldots$$

(but: $$(\lambda y. y) ((\lambda x. x x) (\lambda x. x x)) \rightarrow_\beta \lambda y. y$$)

$\lambda$ calculus is not terminating
\[ \beta \text{ reduction is confluent} \]

Confluence: 
\[ s \rightarrow^{*}_{\beta} x \land s \rightarrow^{*}_{\beta} y \implies \exists t. x \rightarrow^{*}_{\beta} t \land y \rightarrow^{*}_{\beta} t \]
\[ \beta \text{ reduction is confluent} \]

Confluence:  
\[ s \overset{\beta}{\rightarrow}^* x \land s \overset{\beta}{\rightarrow}^* y \implies \exists t. \ x \overset{\beta}{\rightarrow}^* t \land y \overset{\beta}{\rightarrow}^* t \]

Order of reduction does not matter for result  
Normal forms in \( \lambda \) calculus are unique
\( \beta \) reduction is confluent

Example:

\[
(\lambda x \ y. \ y) \ ((\lambda x. \ x \ x) \ a)
\]

\[
(\lambda x \ y. \ y) \ ((\lambda x. \ x \ x) \ a)
\]
\[ \beta \text{ reduction is confluent} \]

Example:

\[
(\lambda x \ y \ y) \ ( (\lambda x \ x \ x) \ a) \xrightarrow{\beta} (\lambda x \ y \ y) \ (a \ a)
\]

\[
(\lambda x \ y \ y) \ ( (\lambda x \ x \ x) \ a) \xrightarrow{\beta} \lambda y \ y
\]
\( \beta \) reduction is confluent

Example:

\[
(\lambda x \ y. \ y) \ ((\lambda x. \ x \ x) \ a) \rightarrow_\beta (\lambda x \ y. \ y) \ (a \ a) \rightarrow_\beta \lambda y. \ y
\]

\[
(\lambda x \ y. \ y) \ ((\lambda x. \ x \ x) \ a) \rightarrow_\beta \lambda y. \ y
\]
So, what can you do with \( \lambda \) calculus?

\( \lambda \) calculus is very expressive, you can encode:

- logic, set theory
- turing machines, functional programs, etc.

Examples:
So, what can you do with $\lambda$ calculus?

$\lambda$ calculus is very expressive, you can encode:

- logic, set theory
- turing machines, functional programs, etc.

Examples:

true $\equiv \lambda x. y. x$
false $\equiv \lambda x. y. y$
if $\equiv \lambda z. x y. z x y$
So, what can you do with $\lambda$ calculus?

$\lambda$ calculus is very expressive, you can encode:

- logic, set theory
- turing machines, functional programs, etc.

Examples:

true $\equiv \lambda x \ y. \ x$ \hspace{1cm} if true $x \ y \mathrel{\longrightarrow}^* x$

false $\equiv \lambda x \ y. \ y$ \hspace{1cm} if false $x \ y \mathrel{\longrightarrow}^* y$

if $\equiv \lambda z \ x \ y. \ z \ x \ y$
So, what can you do with \( \lambda \) calculus?

\( \lambda \) calculus is very expressive, you can encode:

- logic, set theory
- turing machines, functional programs, etc.

**Examples:**

\[
\begin{align*}
\text{true} & \equiv \lambda x \ y. \ x \\
\text{false} & \equiv \lambda x \ y. \ y \\
\text{if} & \equiv \lambda z \ x \ y. \ z \ x \ y
\end{align*}
\]

if \( \text{true} \ x \ y \rightarrow^* \beta \ x \)

if \( \text{false} \ x \ y \rightarrow^* \beta \ y \)

Now, not, and, or, etc is easy:
So, what can you do with λ calculus?

λ calculus is very expressive, you can encode:
- logic, set theory
- turing machines, functional programs, etc.

Examples:

true \equiv \lambda x y. x  
false \equiv \lambda x y. y  
if \equiv \lambda z x y. z x y

if \text{true} \ x \ y \rightarrow^* \ x  
if \text{false} \ x \ y \rightarrow^* \ y

Now, not, and, or, etc is easy:

not \equiv \lambda x. \text{if} \ x \text{false} \text{true}  
and \equiv \lambda x y. \text{if} \ x \ y \text{false}  
or \equiv \lambda x y. \text{if} \ x \text{true} \ y
Fix Points

$$(\lambda x. f \ (x \ x \ f)) \ (\lambda x. f \ (x \ x \ f)) \ t \xrightarrow{\beta}$$
Fix Points

$$((\lambda x. f . (x \times f)) \ (\lambda x. f . (x \times f))) \ t \ \longrightarrow_\beta$$

$$((\lambda f. f \ ((\lambda x. f . (x \times f)) \ (\lambda x. f . (x \times f)))) \ (\lambda x. f . (x \times f))) \ t \ \longrightarrow_\beta$$
(\lambda x \ f. \ f \ (x \ f)) \ (\lambda x \ f. \ f \ (x \ f)) \ t \rightarrow^\beta \\
(\lambda f. \ f \ ((\lambda x \ f. \ f \ (x \ f)) \ (\lambda x \ f. \ f \ (x \ f)) \ f)) \ t \rightarrow^\beta \\
t \ ((\lambda x \ f. \ f \ (x \ f)) \ (\lambda x \ f. \ f \ (x \ f)) \ t)
Fix Points

$((\lambda x. f. (x \times f)) \ (\lambda x. f. (x \times f))) \ t \rightarrow_{\beta} \beta$

$((\lambda f. f. (\lambda x. f. (x \times f)) \ (\lambda x. f. (x \times f)) \ f)) \ t \rightarrow_{\beta} \beta$

$t \ ((\lambda x. f. (x \times f)) \ (\lambda x. f. (x \times f)) \ t)$

$\mu = ((\lambda x. f. (x \times f)) \ (\lambda x. f. (x \times f)))$

$\mu \ t \rightarrow_{\beta} \beta \ t \ (\mu \ t) \rightarrow_{\beta} \beta \ t \ (t \ (\mu \ t)) \rightarrow_{\beta} \beta \ t \ (t \ (t \ (\mu \ t))) \rightarrow_{\beta} \beta \ ...$
Fix Points

\[(\lambda x. f \ (x \ x \ f)) \ (\lambda x. f \ (x \ x \ f)) \quad t \rightarrow^{\beta}
\]

\[(\lambda f. f \ ((\lambda x. f \ (x \ x \ f)) \ (\lambda x. f \ (x \ x \ f)) \ f)) \quad t \rightarrow^{\beta}
\]

\[t \ ((\lambda x. f \ (x \ x \ f)) \ (\lambda x. f \ (x \ x \ f)) \ t)\]

\[\mu = (\lambda x f. f \ (x \ x \ f)) \ (\lambda x f. f \ (x \ x \ f))\]

\[\mu \ t \rightarrow^{\beta} t \ (\mu \ t) \rightarrow^{\beta} t \ (t \ (\mu \ t)) \rightarrow^{\beta} t \ (t \ (t \ (\mu \ t))) \rightarrow^{\beta} \ldots\]

\[(\lambda x f. f \ (x \ x \ f)) \ (\lambda x f. f \ (x \ x \ f))\] is Turing’s fix point operator
Nice, but ...

As a mathematical foundation, (untyped) $\lambda$ does not work. It is inconsistent.

Russell (1901): Paradox

$R \equiv \{X | X \not\in X\}$

Church (1930): $\lambda$ calculus as logic, true, false, $\land$, ... as $\lambda$ terms

Russell’s paradox in $\lambda$-calculus: you can write $R \equiv \lambda x.\neg(x x)$ and get $(R R) = \beta \neg(R R)$ because $(R R) = (\lambda x.\neg(x x)) R \rightarrow \beta \neg(R R)$.
As a mathematical foundation, (untyped) \( \lambda \) does not work. It is inconsistent.

- **Russell** (1901): Paradox \( R \equiv \{ X | X \notin X \} \)
- **Church** (1930): \( \lambda \) calculus as logic, true, false, \( \land \), ... as \( \lambda \) terms

**Russell’s paradox in \( \lambda \)-calculus:**
As a mathematical foundation, (untyped) $\lambda$ does not work. It is inconsistent.

- **Russell** (1901): Paradox $R \equiv \{ X \mid X \notin X \}$
- **Church** (1930): $\lambda$ calculus as logic, true, false, $\land$, ... as $\lambda$ terms

Russell’s paradox in $\lambda$-calculus:
you can write $R \equiv \lambda x. \not \left( x \ x \right)$
Nice, but ...

As a mathematical foundation, (untyped) $\lambda$ does not work. It is inconsistent.

- **Russell** (1901): Paradox $R \equiv \{X | X \notin X\}$
- **Church** (1930): $\lambda$ calculus as logic, true, false, $\land$, ... as $\lambda$ terms

**Russell’s paradox in $\lambda$-calculus:**
- you can write $R \equiv \lambda x. \text{not} (x \ x)$
- and get $(R \ R) \beta \text{not} (R \ R)$
As a mathematical foundation, (untyped) $\lambda$ does not work. It is inconsistent.

- **Russell** (1901): Paradox $R \equiv \{X | X \notin X\}$
- **Church** (1930): $\lambda$ calculus as logic, true, false, $\land$, ... as $\lambda$ terms

**Russell’s paradox in $\lambda$-calculus:**
- you can write $R \equiv \lambda x. \text{not} \ (x \ x)$
- and get $(R \ R) =_{\beta} \text{not} \ (R \ R)$
- because $(R \ R) = (\lambda x. \text{not} \ (x \ x)) \ R \rightarrow_{\beta} \text{not} \ (R \ R)$
Summary of what we learned so far

- λ calculus syntax
- free variables, substitution
- α conversion
- β reduction
- β reduction is confluent
- λ calculus is very expressive (Turing complete)
- λ calculus is inconsistent (as a logic)
Exercise

Reduce \((\lambda x. \ y \ (\lambda v. \ x \ v)) \ (\lambda y. \ v \ y)\) to \(\beta\) normal form.
\( \lambda \) calculus is inconsistent

Can find term \( R \) such that \( R \ R \ \beta \equiv \text{not}(R \ R) \)
\[ \text{\textlambda calculus is inconsistent} \]

Can find term \( R \) such that \( R \; R \; =_{\beta} \; \text{not}(R \; R) \)

**Solution**: rule out ill-formed terms by using types.

(Church 1940)
Introducing types

Idea: assign a type to each “sensible” λ term.

Examples:
Introducing types

Idea: assign a type to each “sensible” \( \lambda \) term.

Examples:

- for term \( t \) has type \( \alpha \) write \( t :: \alpha \)
Introducing types

Idea: assign a type to each “sensible” λ term.

Examples:

- for term $t$ has type $\alpha$ write $t :: \alpha$
- if $x$ has type $\alpha$ then $\lambda x. x$ is a function from $\alpha$ to $\alpha$
  Write: $(\lambda x. x) :: \alpha \Rightarrow \alpha$
Introducing types

Idea: assign a type to each “sensible” $\lambda$ term.

Examples:

- for term $t$ has type $\alpha$ write $t :: \alpha$
- if $x$ has type $\alpha$ then $\lambda x. x$ is a function from $\alpha$ to $\alpha$
  Write: $(\lambda x. x) :: \alpha \Rightarrow \alpha$
- for $s$ $t$ to be sensible:
  $s$ must be a function
  $t$ must be right type for parameter
- If $s :: \alpha \Rightarrow \beta$ and $t :: \alpha$ then $(s \ t) :: \beta$
That’s it!
That’s it!
Now Formally
Syntax for $\lambda \rightarrow$

**Terms:**

$t ::= v \mid c \mid (t \ t) \mid (\lambda x. \ t)$

$v, x \in V, \ c \in C, \ V, C$ sets of names

**Types:**

$\tau ::= b \mid \nu \mid \tau \Rightarrow \tau$

$b \in \{\text{bool, int, ...}\}$ base types

$\nu \in \{\alpha, \beta, ...\}$ type variables

$\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)$
Syntax for $\lambda \to$

**Terms:**

\[ t ::= v \mid c \mid (t \ t) \mid (\lambda x. \ t) \]

\[ v, x \in V, \quad c \in C, \quad V, C \text{ sets of names} \]

**Types:**

\[ \tau ::= b \mid \nu \mid \tau \Rightarrow \tau \]

\[ b \in \{\text{bool, int, ...}\} \text{ base types} \]

\[ \nu \in \{\alpha, \beta, ...\} \text{ type variables} \]

\[ \alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma) \]

**Context $\Gamma$:**

$\Gamma$: function from variable and constant names to types.
Syntax for $\lambda \rightarrow$

**Terms:**
$$t ::= v \mid c \mid (t \ t) \mid (\lambda x. \ t)$$

$v, x \in V, \ c \in C, \ V, C$ sets of names

**Types:**
$$\tau ::= b \mid \nu \mid \tau \Rightarrow \tau$$

$b \in \{\text{bool, int, ...}\}$ base types

$\nu \in \{\alpha, \beta, \ldots\}$ type variables

$$\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)$$

**Context $\Gamma$:**

$\Gamma$: function from variable and constant names to types.

**Term $t$ has type $\tau$ in context $\Gamma$:**
$$\Gamma \vdash t :: \tau$$
Examples

Γ ⊢ (λx. x) :: α ⇒ α
Examples

Γ ⊢ (λx. x) :: α ⇒ α

[y ← int] ⊢ y :: int
Examples

\[ \Gamma \vdash (\lambda x. x) :: \alpha \Rightarrow \alpha \]

\[ [y \leftarrow \text{int}] \vdash y :: \text{int} \]

\[ [z \leftarrow \text{bool}] \vdash (\lambda y. y) \, z :: \text{bool} \]
Examples

\[ \Gamma \vdash (\lambda x. \ x) :: \alpha \Rightarrow \alpha \]

\[ [y \leftarrow \text{int}] \vdash y :: \text{int} \]

\[ [z \leftarrow \text{bool}] \vdash (\lambda y. \ y) \ z :: \text{bool} \]

\[ [] \vdash \lambda f \ x. \ f \ x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta \]
Examples

\[ \Gamma \vdash (\lambda x. \, x) :: \alpha \Rightarrow \alpha \]

\[ [y \leftarrow \text{int}] \vdash y :: \text{int} \]

\[ [z \leftarrow \text{bool}] \vdash (\lambda y. \, y) \, z :: \text{bool} \]

\[ [] \vdash \lambda f \, x. \, f \, x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta \]

A term \( t \) is **well typed** or **type correct** if there are \( \Gamma \) and \( \tau \) such that \( \Gamma \vdash t :: \tau \)
Type Checking Rules

Variables: \[ \Gamma \vdash x :: \Gamma(x) \]
Type Checking Rules

Variables:
\[ \Gamma \vdash x :: \Gamma(x) \]

Application:
\[ \Gamma \vdash (t_1 \ t_2) :: \tau \]
Type Checking Rules

Variables:

\[ \Gamma \vdash x :: \Gamma(x) \]

Application:

\[ \begin{align*} 
\Gamma \vdash t_1 :: \tau_2 & \Rightarrow \tau \\
\Gamma \vdash t_2 :: \tau_2 \\
\Gamma \vdash (t_1 \ t_2) :: \tau 
\end{align*} \]
Type Checking Rules

Variables: \[ \Gamma \vdash x :: \Gamma(x) \]

Application: \[ \Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2 \]
\[ \Gamma \vdash (t_1 \ t_2) :: \tau \]

Abstraction: \[ \Gamma \vdash (\lambda x. \ t) :: \tau_x \Rightarrow \tau \]
Type Checking Rules

Variables: \( \Gamma \vdash x :: \Gamma(x) \)

Application: \( \Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2 \)
\( \Gamma \vdash (t_1 \ t_2) :: \tau \)

Abstraction: \( \Gamma \vdash [x \leftarrow \tau_x] \vdash t :: \tau \)
\( \Gamma \vdash (\lambda x. \ t) :: \tau_x \Rightarrow \tau \)
What about $\beta$ reduction?
What about \( \beta \) reduction?

Definition of \( \beta \) reduction stays the same.
What about $\beta$ reduction?

**Definition of $\beta$ reduction stays the same.**

**Fact:** Well typed terms stay well typed after $\beta$ reduction

**Formally:** $\Gamma \vdash s :: \tau \land s \rightarrow_\beta t \implies \Gamma \vdash t :: \tau$
What about $\beta$ reduction?

Definition of $\beta$ reduction stays the same.

**Fact:** Well typed terms stay well typed after $\beta$ reduction

Formally: \[ \Gamma \vdash s :: \tau \land s \xrightarrow{\beta} t \implies \Gamma \vdash t :: \tau \]

This property is called **subject reduction**
Exercise

Derive the normal form by $\beta$-reducing the following term:

$$(\lambda f \ x. \ f \ x)(\lambda y. \ y) \ z$$
What about termination?

$\beta$ is decidable.

To decide if $s = \beta t$, reduce $s$ and $t$ to normal form (always exists, because $\rightarrow^\ast$ terminates), and compare result.
What about termination?

\[ \beta \text{ reduction in } \lambda \rightarrow \text{ always terminates.} \]

(Alan Turing, 1942)
What about termination?

\[ \beta \text{ reduction in } \lambda \rightarrow \text{ always terminates.} \]

(Alan Turing, 1942)

\[ \equiv_{\beta} \text{ is decidable} \]

To decide if \( s \equiv_{\beta} t \), reduce \( s \) and \( t \) to normal form (always exists, because \( \rightarrow_{\beta} \) terminates), and compare result.
What about termination?

\[ \beta \text{ reduction in } \lambda \overset{\rightarrow}{\rightarrow} \text{ always terminates.} \]

(Alan Turing, 1942)

- \(=_{\beta}\) is decidable
  To decide if \(s =_{\beta} t\), reduce \(s\) and \(t\) to normal form (always exists, because \(\rightarrow^{\beta}\) terminates), and compare result.

- \(=_{\alpha\beta}\) is decidable
We have learned so far...

- Simply typed lambda calculus: $\lambda \to$
- Typing rules for $\lambda \to$, type variables, type contexts
- $\beta$-reduction in $\lambda \to$ satisfies subject reduction
- $\beta$-reduction in $\lambda \to$ always terminates
Defining Higher Order Logic
What is Higher Order Logic?

- Propositional Logic:
  - no quantifiers
  - all variables have type bool
What is Higher Order Logic?

- **Propositional Logic:**
  - no quantifiers
  - all variables have type bool

- **First Order Logic:**
  - quantification over values, but not over functions and predicates,
  - terms and formulas syntactically distinct
What is Higher Order Logic?

- **Propositional Logic:**
  - no quantifiers
  - all variables have type bool

- **First Order Logic:**
  - quantification over values, but not over functions and predicates,
  - terms and formulas syntactically distinct

- **Higher Order Logic:**
  - quantification over everything, including predicates
  - consistency by types
  - formula = term of type bool
  - definition built on $\lambda \rightarrow$ with certain default types and constants
Defining Higher Order Logic

Default types:
Default types:

bool
Defining Higher Order Logic

Default types:

```
  bool  _  ⇒  _
```
Defining Higher Order Logic

Default types:

bool _ ⇒ _ ind
Defining Higher Order Logic

Default types:

bool \rightarrow \_ \Rightarrow \_ \quad \text{ind}

- bool
- \Rightarrow \text{sometimes called fun}
Defining Higher Order Logic

Default types:

\[ \text{bool} \quad \_ \quad \Rightarrow \quad \_ \quad \text{ind} \]

- bool
- \[ \Rightarrow \] sometimes called \textit{fun}

Default Constants:
Defining Higher Order Logic

Default types:

\[ \text{bool} \quad \Rightarrow \quad \text{ind} \]

- bool
- \( \Rightarrow \) sometimes called \emph{fun}

Default Constants:

\[ = \quad :: \quad \alpha \Rightarrow \alpha \Rightarrow \text{bool} \]
Problem: Define syntax for binders like ∀ and ∃
Problem: Define syntax for binders like $\forall$ and $\exists$

One approach: $\forall :: \text{var} \Rightarrow \text{term} \Rightarrow \text{bool}$

Drawback: need to think about substitution, $\alpha$ conversion again.
Problem: Define syntax for binders like ∀ and ∃

One approach: ∀ :: var ⇒ term ⇒ bool

Drawback: need to think about substitution, α conversion again.

But: Already have binder, substitution, α conversion in host language
Higher Order Abstract Syntax

**Problem:** Define syntax for binders like $\forall$ and $\exists$

**One approach:** $\forall :: var \Rightarrow term \Rightarrow bool$

**Drawback:** need to think about substitution, $\alpha$ conversion again.

**But:** Already have binder, substitution, $\alpha$ conversion in host language

$$\lambda$$

**So:** Use $\lambda$ to encode all other binders.
Higher Order Abstract Syntax

Example:

\[
\text{ALL} :: (\alpha \Rightarrow \text{bool}) \Rightarrow \text{bool}
\]

HOAS \hspace{1cm} \text{usual syntax}
Higher Order Abstract Syntax

Example:

\[ \text{ALL} :: (\alpha \Rightarrow \text{bool}) \Rightarrow \text{bool} \]

HOAS                     usual syntax

\[ \text{ALL} \ (\lambda x. \ x = 2) \]
Higher Order Abstract Syntax

Example:

\[ \text{ALL} :: (\alpha \Rightarrow \text{bool}) \Rightarrow \text{bool} \]

<table>
<thead>
<tr>
<th>HOAS</th>
<th>usual syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ALL} \ (\lambda x. x = 2) )</td>
<td>( \forall x. x = 2 )</td>
</tr>
</tbody>
</table>
Example:

\[ \text{ALL} :: (\alpha \rightarrow \text{bool}) \rightarrow \text{bool} \]

\begin{align*}
\text{HOAS} & \quad \text{usual syntax} \\
\text{ALL (} \lambda x. x = 2 \text{)} & \quad \forall x. x = 2 \\
\text{ALL } P & \\
\end{align*}
Higher Order Abstract Syntax

Example:

\[ \text{ALL} :: (\alpha \Rightarrow \text{bool}) \Rightarrow \text{bool} \]

<table>
<thead>
<tr>
<th>HOAS</th>
<th>usual syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL (\lambda x. x = 2)</td>
<td>\forall x. x = 2</td>
</tr>
<tr>
<td>ALL P</td>
<td>\forall x. P x</td>
</tr>
</tbody>
</table>
Back to HOL

Base: \( \text{bool, } \Rightarrow, \text{ind} \)

And the rest is

\( \text{True} \equiv (\lambda x. x) = (\lambda x. x) \)

\( P \land Q \equiv \lambda p q. ((\lambda f. f p q) = (\lambda f. f \text{True True})) \)

\( P \rightarrow Q \equiv \lambda p q. ((p \land q) = p) \)

\( \text{All } P \equiv P = (\lambda x. \text{True}) \)

\( \text{Ex } P \equiv \forall Q. ((\forall x. P x \rightarrow Q) \rightarrow Q) \)

\( \text{False} \equiv \forall P. P \)

\( \neg P \equiv P \rightarrow \text{False} \)

\( P \lor Q \equiv \forall R. (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R \)
Back to HOL

Base: \( \text{bool, } \Rightarrow, \text{ ind} \)

And the rest is definitions:

- True \( \equiv \)
- \( P \land Q \) \( \equiv \)
- \( P \Rightarrow Q \) \( \equiv \)
- All \( P \) \( \equiv \)
- Ex \( P \) \( \equiv \)
- False \( \equiv \)
- \( \neg P \) \( \equiv \)
- \( P \lor Q \) \( \equiv \)
Back to HOL

Base: \( \text{bool, } \Rightarrow, \text{ind} \quad = \quad \)

And the rest is definitions:

- True \( \equiv (\lambda x. x) = (\lambda x. x) \)
- \( P \land Q \equiv \)
- \( P \rightarrow Q \equiv \)
- All \( P \equiv \)
- Ex \( P \equiv \)
- False \( \equiv \)
- \( \neg P \equiv \)
- \( P \lor Q \equiv \)
Back to HOL

Base: \( \text{bool, } \Rightarrow, \text{ind} \) =

And the rest is definitions:

\[
\begin{align*}
\text{True} & \equiv (\lambda x. x) = (\lambda x. x) \\
P \land Q & \equiv \lambda p q. ((\lambda f. f p q) = (\lambda f. f \text{ True True})) \\
P \rightarrow Q & \equiv \\
\text{All } P & \equiv \\
\text{Ex } P & \equiv \\
\text{False} & \equiv \\
\neg P & \equiv \\
P \lor Q & \equiv
\end{align*}
\]
Back to HOL

Base: \textit{bool}, \Rightarrow, \textit{ind}

And the rest is definitions:

- \text{True} \equiv (\lambda x. x) = (\lambda x. x)
- \mathcal{P} \land \mathcal{Q} \equiv \lambda p q. ((\lambda f. f p q) = (\lambda f. f \text{ True} \text{ True}))
- \mathcal{P} \rightarrow \mathcal{Q} \equiv \lambda p q. ((p \land q) = p)
- \forall \mathcal{P} \equiv 
- \exists \mathcal{P} \equiv 
- \text{False} \equiv 
- \neg \mathcal{P} \equiv 
- \mathcal{P} \lor \mathcal{Q} \equiv
Back to HOL

Base: \( \text{bool, } \Rightarrow, \; \text{ind} \)

And the rest is definitions:

- **True** \( \equiv \) \((\lambda x. \; x) = (\lambda x. \; x)\)
- **\( P \land Q \)** \( \equiv \) \(\lambda p \; q. \; ((\lambda f. \; f \; p \; q) = (\lambda f. \; f \; \text{True} \; \text{True}))\)
- **\( P \rightarrow Q \)** \( \equiv \) \(\lambda p \; q. \; ((p \land q) = p)\)
- **All \( P \)** \( \equiv \) \( P = (\lambda x. \; \text{True})\)
- **Ex \( P \)** \( \equiv \) \( \forall Q. \; (\forall x. \; P \; x \rightarrow Q) \rightarrow Q\)
- **False** \( \equiv \) \( \forall P. \; P\)
- **\( \neg P \)** \( \equiv \) \( P \rightarrow \text{False}\)
- **\( P \lor Q \)** \( \equiv \) \( \forall R. \; (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R\)
In conclusion

That was HOL from scratch! But we skipped some steps.

For example, what did we assume about $=$?

Anyhow, the resulting logic is consistent. (How do we know that?)
We have learned so far ... 

- HOL
- Higher Order Abstract Syntax
- Defining HOL

More on HOL in COMP4161.