Week 10

Today  $\lambda$-calculus.

Friday  Wrapup and exam prep.
Admin announcements

myExperience

The myExperience survey is up. Link on the website. Please take a moment to fill it in, it helps a lot! :)
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.
- **Wednesday May 3rd 8AM–Friday May 4th 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.
Wednesday May 3rd 8AM–Friday May 4th 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.
- Wednesday May 3rd 8AM–Friday May 4th 8AM.
- When the time starts, you will receive the exam questions in your UNSW inbox.
- Submission via pdf through give.
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.
- Wednesday May 3rd 8AM–Friday May 4th 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. 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.
- Wednesday May 3rd 8AM–Friday May 4th 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. 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

```cpp
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) return 1;
    return 0;
}
```

Let’s evaluate this function step by step:

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

*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:

- \( f(0) \) → (function application)
- if 0 = 0 then 1 else 0 → (equality comparison)
- if true then 1 else 0 →

*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:

- $f(0) \rightarrow$ (function application)
  - 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 $\lambda$-calculus formalises this kind of step-by-step evaluation of program expressions...
Detour: program evaluation

The $\lambda$-calculus formalises this kind of step-by-step evaluation of program expressions...

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

Basic intuition:

Instead of $f(x) = x + 5$

Write $f = \lambda x. x + 5$

- A term
Basic intuition:

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

\( \lambda x. x + 5 \)

- a term
- a nameless function
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 \( f \ a \)
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
Syntax

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

\( v, x \in V, \ c \in C, \ V, C \) 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?

0 ≡ Z
1 ≡ S(Z)
2 ≡ S(S(Z))

In the λ-calculus we have Church numerals:

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

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

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

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

\(\text{We encode a number } n \text{ a function...}\)

\(\text{..that accepts two arguments, } f \text{ and } x...\)
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 \( n \) a function...
- ..that accepts two arguments, \( f \) and \( x \)...
- ..and returns the result of applying \( f \) to \( x \) \( n \) times.
Encoding numbers

0 ≡ λf x. x
1 ≡ λf x. f x
2 ≡ λf 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

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

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

$$\lambda m n. \lambda f x. m f (n f x)$$
Encoding booleans

\[
\text{true} \equiv \lambda x \ y. \ x \\
\text{false} \equiv \lambda x \ y. \ y
\]

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

true \equiv \lambda x \ y. \ x
false \equiv \lambda x \ y. \ y
\neg b \equiv \lambda b. \ b \ false \ true
if \ z \ then \ x \ else \ y \equiv

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 ≡ λz x y. z x 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

true \equiv \lambda x \ y \ . \ x
false \equiv \lambda x \ y \ . \ y
¬b \equiv \lambda b. \ b \ false \ true
if \ z \ then \ x \ else \ y \equiv \lambda z \ x \ y. \ z \ x \ y
isZero(n) \equiv \lambda n. \ n \ (\lambda x. \ false) \ true

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

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

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

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

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

This program can be encoded in λ-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

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

$$
\lambda n. \\
(n (\lambda x. (\lambda x y. y)) (\lambda x y. x)) \text{ if } \text{isZero}(n) \text{ then } \\\n(\lambda f x. f x) \text{ if } n = 0 \text{ then } 1 \text{ else } 0 \\\n(\lambda f x. x) 
$$
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:

\[\lambda n.\]
\[(n \,(\lambda x. \,(\lambda x \, y. \, y)) \,(\lambda x \, y. \, x)) \quad \text{if isZero}(n) \text{ then} \]
\[(\lambda f \, x. \, f \, x) \quad 1 \]
\[(\lambda f \, x. \, x) \quad \text{else} \, 0\]
Computation

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

Example

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

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

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

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

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

**Example**

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

\[ \begin{align*}
\beta \text{ reduction:} & \\
(\lambda x. s) \ t & \rightarrow_\beta s[x \leftarrow t] \\
\underline{\hspace{2cm}} & \\
t & \rightarrow_\beta t' \\
\underline{\hspace{2cm}} & \\
(s \ t) & \rightarrow_\beta (s \ t') \\
\underline{\hspace{2cm}} & \\
\lambda x. s & \rightarrow_\beta \lambda x. s' \\
\underline{\hspace{2cm}} & \\
s & \rightarrow_\beta s' \\
\underline{\hspace{2cm}} & \\
(s \ t) & \rightarrow_\beta (s' \ t) \\
\underline{\hspace{2cm}} & \\
\lambda x. s & \rightarrow_\beta \lambda x. s'
\end{align*} \]
Defining Computation

\[ \beta \text{ reduction:} \]

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

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

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

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:

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

Example: \(FV(\lambda x. (\lambda y. (\lambda x. \, x) \, y) \, y \, x) = \{y\}\)
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:
\[
FV (x) = \{x\} \\
FV (c) = \{} \\
FV (s \ t) = FV(s) \cup FV(t) \\
FV (\lambda x. \ t) = FV(t) \setminus \{x\}
\]

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

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

\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] & =
\end{align*}
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

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

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

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

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

\[
(x \ (\lambda x. \ x) \ (\lambda y. \ z \ x))[x \leftarrow y] \\
= \ (x[x \leftarrow y]) \ (\((\lambda x. \ x)[x \leftarrow y]) \ (\((\lambda y. \ z \ x)[x \leftarrow y])) \\
= \ y \ (\lambda x. \ x) \ (\lambda y'. \ z \ y)
\]
α 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.
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:

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

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:
\[
\begin{align*}
  &y \notin FV(t) \\
  \quad (\lambda x. \ t) \rightarrow_{\alpha} (\lambda y. \ t[x \leftarrow y])
\end{align*}
\]

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

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

\[
\begin{align*}
  &s \rightarrow_{\alpha} s' \\
  \quad (\lambda x. \ s) \rightarrow_{\alpha} (\lambda x. \ s')
\end{align*}
\]

\[
\begin{align*}
  s =_{\alpha} t \quad \text{iff} \quad s \rightarrow^{*}_{\alpha} t
\end{align*}
\]

\( \rightarrow^{*}_{\alpha} = \) transitive, reflexive closure of \( \rightarrow_{\alpha} = \) multiple steps)
\( \alpha \) Conversion

Examples:

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

Examples:

\[
x (\lambda x \ y \cdot x \ y) =_\alpha \ x (\lambda y \ x \cdot y \ x)
\]
\[\alpha\text{ Conversion}\]

Examples:
\[x \ (\lambda x \ y. \ x \ y) =_\alpha x \ (\lambda y \ x. \ y \ x) =_\alpha x \ (\lambda z \ y. \ z \ y)\]
**α** Conversion

Examples:

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

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

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

\[ \not= _\alpha \ z \ (\lambda z \ y. \ z \ y) \]
\[ \alpha \text{ Conversion} \]

**Examples:**

\[
\begin{align*}
    x \ (\lambda x \ y \ . \ x \ y) &= \alpha \ x \ (\lambda y \ x \ . \ y \ x) \\
    &= \alpha \ x \ (\lambda z \ y \ . \ z \ y) \\
    \neq \alpha \ z \ (\lambda z \ y \ . \ z \ y) \\
    \neq \alpha \ z \ (\lambda x \ x \ . \ x \ x)
\end{align*}
\]
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 $t \rightarrow_\beta s$
  - $t$ is called a redex (reducible expression)
  - $t$ is reducible iff it contains a redex
  - if it is not reducible, $t$ is in normal form
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 $t \rightarrow^\beta s$
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: $\longrightarrow_\beta$

Some notation and concepts:

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

Some notation and concepts:

- $\beta$ conversion: $s =^\beta t$ iff $\exists n. s \rightarrow^* n \land t \rightarrow^* n$
- $t$ 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: $$(\lambda x \ y. y) \ ((\lambda x. x \ x) \ (\lambda x. x \ x)) \rightarrow_{\beta} \ \lambda y. \ y$$)
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: $(\lambda x y. y) ((\lambda x. x x) (\lambda x. x x)) \rightarrow^\beta \lambda y. y$)

$\lambda$ calculus is not terminating
\( \beta \) reduction is confluent

**Confluence:**

\[
s \xrightarrow{\beta}^* x \land s \xrightarrow{\beta}^* y \iff \exists t. \ x \xrightarrow{\beta}^* t \land y \xrightarrow{\beta}^* t
\]
\( \beta \) reduction is confluent

Confluence: \( s \rightarrow^* x \land s \rightarrow^* y \overset{\exists t.}{\Rightarrow} \exists t. \ x \rightarrow^* t \land y \rightarrow^* t \)

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

Example:

$$(\lambda x \ y. \ y) \ ((\lambda x. \ x \ x) \ a)$$
$$(\lambda x \ y. \ y) \ ((\lambda x. \ x \ x) \ a)$$
$\beta$ reduction is confluent

Example:

$(\lambda x \ y \ y) \ ((\lambda x \ x \ x) \ a) \to_\beta (\lambda x \ y \ y) \ (a \ a)$

$(\lambda x \ y \ y) \ ((\lambda x \ x \ x) \ a) \to_\beta \lambda y. \ y$
\( \beta \) reduction is confluent

Example:

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

\[
(\lambda x \ y. \ y) \ ((\lambda x. \ x \ x) \ a) \xrightarrow{\beta} \lambda y. \ y
\]
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 $\equiv \lambda x \ y. \ x$  
if true $x \ y \rightarrow^\ast x$

false $\equiv \lambda x \ y. \ y$  
if false $x \ y \rightarrow^\ast y$

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:**

- $\text{true} \equiv \lambda x \, y. \, x$ \hspace{1cm} if $\text{true} \, x \, y \xrightarrow{\beta} x$
- $\text{false} \equiv \lambda x \, y. \, y$ \hspace{1cm} if $\text{false} \, x \, y \xrightarrow{\beta} y$
- $\text{if} \equiv \lambda z \, x \, y. \, z \, x \, y$

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

\[
\text{true} \equiv \lambda x \ y. \ x \quad \text{if} \quad \text{true} \ x \ y \rightarrow^* x
\]

\[
\text{false} \equiv \lambda x \ y. \ y \quad \text{if} \quad \text{false} \ x \ y \rightarrow^* y
\]

\[
\text{if} \quad \equiv \lambda z \ x \ y. \ z \ x \ y
\]

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

\[
\text{not} \equiv \lambda x. \ \text{if} \ x \ \text{false} \ \text{true}
\]

\[
\text{and} \equiv \lambda x \ y. \ \text{if} \ x \ y \ \text{false}
\]

\[
\text{or} \quad \equiv \lambda x \ y. \ \text{if} \ x \ \text{true} \ y
\]
Fix Points

\[(\lambda x. f \, (x \, x \, f)) \, (\lambda x. f \, (x \, x \, f)) \quad t \rightarrow_{\beta}
\]
Fix Points

\[ (\lambda x \ f \ . \ f \ (x \ x \ f)) \ (\lambda x \ f \ . \ f \ (x \ x \ f)) \ t \xrightarrow{\beta} \]

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

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

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

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

\[
(\lambda x \ f. \ f \ (x \ x \ f)) \ (\lambda x \ f. \ f \ (x \ x \ f)) \ t \xrightarrow{\beta}
\]

\[
(\lambda f. \ f \ ((\lambda x \ f. \ f \ (x \ x \ f)) \ (\lambda x \ f. \ f \ (x \ x \ f)) \ f)) \ t \xrightarrow{\beta}
\]

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

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

\[
\mu \ t \xrightarrow{\beta} t \ (\mu \ t) \xrightarrow{\beta} t \ (t \ (\mu \ t)) \xrightarrow{\beta} t \ (t \ (t \ (\mu \ t))) \xrightarrow{\beta} \ldots
\]
Fix Points

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

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

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

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

\[\mu \ t \xrightarrow{\beta} t \ (\mu \ t) \xrightarrow{\beta} t \ (t \ (\mu \ t)) \xrightarrow{\beta} t \ (t \ (t \ (\mu \ t))) \xrightarrow{\beta} \ldots \]

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

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

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:
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, $\land$, $\cdots$ as $\lambda$ terms

**Russell’s paradox in $\lambda$-calculus:**
you can write $R \equiv \lambda x. \text{not } (x \ x)$
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 $(R \ R) \beta \text{not} (R \ R)$
Nice, but ...

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 $R \equiv \lambda x. \text{not}(x \ x)$

And get $(R \ R) \mapsto_{\beta} \text{not}(R \ R)$

Because $(R \ R) = (\lambda x. \text{not}(x \ x)) \ R \mapsto_{\beta} \text{not}(R \ R)$
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 \( R \ R \ \Rightarrow^{\beta} \ \text{not}(R \ R) \)
\[ \lambda \text{ calculus is inconsistent} \]

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

**Solution**: rule out ill-formed terms by using types.
(Church 1940)
Introducing types

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

Examples:
Introducing types

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

Examples:

- for \( t \) has type \( \alpha \) write \( t :: \alpha \)
**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: \( (\lambda x. x) :: \alpha \Rightarrow \alpha \)
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: $t ::= \nu \mid c \mid (t \ t) \mid (\lambda x. \ t)$
$v, x \in V, \ c \in C, \ V, C$ 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

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

**Terms:**
$$ t ::= \nu \ | \ c \ | \ (t \ t) \ | \ (\lambda x. \ t) $$
$v, x \in V, \ c \in C, \ V, C$ sets of names

**Types:**
$$ \tau ::= b \ | \ \nu \ | \ \tau \Rightarrow \tau $$
$b \in \{\text{bool, int, ...}\}$ 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.
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:**

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

$b \in \{\text{bool, int,} \ldots\}$ 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

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

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

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

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

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

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

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

Application: \[ \begin{array}{l}
\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \\
\Gamma \vdash t_2 :: \tau_2 \\
\hline
\Gamma \vdash (t_1 \ t_2) :: \tau 
\end{array} \]

Abstraction: \[ \begin{array}{l}
\Gamma[x \leftarrow \tau_x] \vdash t :: \tau \\
\hline
\Gamma \vdash (\lambda x. \ t) :: \tau_x \Rightarrow \tau 
\end{array} \]
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: $\Gamma \vdash s :: \tau \land s \rightarrow_\beta t \implies \Gamma \vdash t :: \tau$

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?

\[ \beta \]

\( \alpha \beta \) is decidable
What about termination?

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

(Alan Turing, 1942)
What about termination?

$\beta$ reduction in $\lambda \rightarrow$ 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.
What about termination?

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

(Alan Turing, 1942)

- \( \equiv_\beta \) 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.

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

- Simply typed lambda calculus: $\lambda \rightarrow$
- Typing rules for $\lambda \rightarrow$, type variables, type contexts
- $\beta$-reduction in $\lambda \rightarrow$ satisfies subject reduction
- $\beta$-reduction in $\lambda \rightarrow$ 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:
Defining Higher Order Logic

Default types:

bool
Defining Higher Order Logic

Default types:

bool ⇒ _ ⇒ _
Defining Higher Order Logic

Default types:

\[ \text{bool} \quad _, \Rightarrow \quad _ \quad \text{ind} \]
Defining Higher Order Logic

Default types:

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

- bool

\[ \Rightarrow \text{sometimes called } fun \]
Defining Higher Order Logic

Default types:

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

- \text{bool}
- \Rightarrow \quad \text{sometimes called fun}

Default Constants:
Defining Higher Order Logic

Default types:

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

- bool
- \Rightarrow \text{sometimes called fun}

Default Constants:

\[ = \quad :: \quad \alpha \Rightarrow \alpha \Rightarrow \text{bool} \]
Problem: Define syntax for binders like $\forall$ and $\exists$
Higher Order Abstract Syntax

**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 $\forall$ and $\exists$

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

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

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

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

Example:

ALL :: (α ⇒ bool) ⇒ bool

HOAS_usual syntax
Higher Order Abstract Syntax

Example:

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

HOAS \hspace{1cm} \text{usual syntax}

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

Example:

ALL :: (α ⇒ bool) ⇒ bool

HOAS                      usual syntax

ALL (λx. x = 2)            ∀x. x = 2
Higher Order Abstract Syntax

Example:

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

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

Example:

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

\begin{align*}
\text{HOAS} & \quad \text{usual syntax} \\
\text{ALL (} \lambda x. x = 2) & \quad \forall x. x = 2 \\
\text{ALL } P & \quad \forall x. P \ x
\end{align*}
Back to HOL

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

And the rest is
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: bool, ⇒, ind =

And the rest is definitions:

True ≡ (λx. x) = (λx. x)
P ∧ Q ≡
P → Q ≡
All P ≡
Ex P ≡
False ≡
¬P ≡
P ∨ Q ≡
Back to HOL

Base: \begin{align*}
& \text{bool, } \Rightarrow, \text{ ind} = \\
\end{align*}

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: \( bool, \Rightarrow, ind \)

And the rest is definitions:

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

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

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

All \( P \equiv \)

Ex \( P \equiv \)

False \equiv \)

\( \neg P \equiv \)

\( P \lor Q \equiv \)
Back to HOL

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

And the rest is definitions:

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