UNTYPED LAMBDA CALCULUS (II)
RECALL: CALL-BY-VALUE O.S.

- Basic rule

\[
(\lambda x.e) \, v \rightarrow e \, [v/x]
\]

- Search rules:

\[
\frac{e_1 \rightarrow e_1'}{e_1 \, e_2 \rightarrow e_1' \, e_2}
\]

\[
\frac{e_2 \rightarrow e_2'}{v \, e_2 \rightarrow v \, e_2''}
\]
CALL-BY-VALUE EVALUATION EXAMPLE

\((\langle x. x \rangle \langle y. y \rangle)\) 
\rightarrow x x [\langle y. y \rangle / x] 
= (\langle y. y \rangle \langle y. y \rangle) 
\rightarrow y [\langle y. y \rangle / y] 
= \langle y. y \rangle
In other words, it is simple to write non-terminating computations in the lambda calculus.

what else can we do?
WE CAN DO EVERYTHING

- The lambda calculus can be used as an “assembly language”

- We can show how to compile useful, high-level operations and language features into the lambda calculus
  - Result = adding high-level operations is convenient for programmers, but not a computational necessity
    - Concrete syntax vs. abstract syntax
    - “Syntactic sugar”
  - Result = make your compiler intermediate language simpler
BOOLEANS

- we can encode booleans
- we will represent “true” and “false” as functions named “trú” and “fls”
- how do we define these functions?
- think about how “true” and “false” can be used
- they can be used by a testing function:
  - “test b then else” returns “then” if b is true and returns “else” if b is false
  - the only thing the implementation of test is going to be able to do with b is to apply it
  - the functions “trú” and “fls” must distinguish themselves when they are applied
BOOLEANS

\( \text{tru} = \lambda t. \lambda f. \ t \quad \text{fls} = \lambda t. \lambda f. \ f \)
\( \text{test} = \lambda x. \lambda \text{then}. \lambda \text{else}. \ x \ \text{then then else} \)

- E.g. (underlined are redexes):
  \( \text{test} \ \text{tru} \ a \ b \)
  \( = (\lambda x. \lambda \text{then}. \lambda \text{else}. \ x \ \text{then then else}) \ \text{tru} \ a \ b \)
  \( \rightarrow (\lambda \text{then}. \lambda \text{else}. \ \text{tru then then else}) \ a \ b \)
  \( \rightarrow (\lambda \text{else}. \ \text{tru} \ a \ \text{else}) \ b \)
  \( \rightarrow \ \text{tru} \ a \ b \)
  \( = (\lambda t. \lambda f. \ t) \ a \ b \)
  \( \rightarrow (\lambda f. \ a) \ b \)
  \( \rightarrow a \quad \text{Try apply test to fls??} \)
BOOLEANS

tru = \t. \f. t  fls = \t. \f. f
and = \b. \c. b c fls

and tru tru
→* tru tru fls
→* tru

(→* stands for multi-step evaluation)
BOOLEANS

tru = \t.\f. t       fls = \t.\f. f
and = \b.\c. b c fls

and fls tru
\* fls tru fls
\* fls
PAIRS

pair = \f.\s.\b. b f s (*pair is a constructor: pair x y*)
fst = \p. p tru
snd = \p. p fls

fst (pair v w)
= fst ((\f.\s.\b. b f s) v w)
→ fst ((\s.\b. b v s) w)
→ fst (\b. b v w)
= (\p. p tru) (\b. b v w)
→(\b. b v w) tru
→ tru v w
→* v
AND WE CAN GO ON...

- numbers
- arithmetic expressions (+, -, *, ...)
- lists, trees and datatypes
- exceptions, loops, ...
- ...
- the general trick:
  - values will be functions – construct these functions so that they return the appropriate information when called by an operation
SIMPLY-TYPED LAMBDA CALCULUS
SIMPLY TYPED LAMBDA-CALCULUS

- Goal: construct a similar system of language that combines the pure lambda-calculus with the basic types such as bool and num.
- A new type rule:
  \( \lambda x. e : \rightarrow \) (arrow type)
- Set of simple types over the type bool is
  \[ t ::= \text{bool} \]
  \[ | \quad t \rightarrow t \]
- Note: type constructor \( \rightarrow \) is right associative:
  - \( t_1 \rightarrow t_2 \rightarrow t_3 = t_1 \rightarrow (t_2 \rightarrow t_3) \)
**Syntax (I)**

\[
e ::= \begin{array}{l}
  x \\
  \text{true} \\
  \text{false} \\
  \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
  \lambda x : t . e \\
  e_1 \; e_2
\end{array}
\]

\text{expressions:}
\begin{align*}
\text{(variable)} \\
\text{(true value)} \\
\text{(false value)} \\
\text{(conditional)} \\
\text{(abstraction)} \\
\text{(application)}
\end{align*}

\[
v ::= \begin{array}{l}
  \text{true} \\
  \text{false} \\
  \lambda x : t . e
\end{array}
\]

\text{values:}
\begin{align*}
\text{(true value)} \\
\text{(false value)} \\
\text{(abstraction value)}
\end{align*}
Syntax (II)

\[ t ::= \]
\[ \text{types:} \]
\[ \text{bool (base boolean type)} \]
\[ \mid t \rightarrow t \quad \text{(type of functions)} \]

\[ \Gamma ::= \]
\[ \text{contexts:} \]
\[ . \text{ (empty context)} \]
\[ \mid \Gamma, x: t \quad \text{(variable binding)} \]
**Typing Rules**

The type system of a language consists of a set of inductive definitions with judgment form:

\[ \Gamma \vdash e : t \]

- \( \Gamma \) (sometimes written as \( G \)) is a typing context (type map) which is a set of hypothesis of the form \( x : t \)
- \( x \) is the variable name appearing in \( e \)
- \( t \) is a type that’s bound to \( x \)
- If the current typing context is \( \Gamma \), then expression \( e \) has type \( t \).
- This judgment is known as *hypothetical judgment* (\( \Gamma \) is the hypothesis).
EVALUATION (O.S.)

[e → e']

\[
\frac{e_1 \rightarrow e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow \text{if } e_1' \text{ then } e_2 \text{ else } e_3} \quad (E\text{-if0})
\]

\[
\frac{}{\text{if } true \text{ then } e_2 \text{ else } e_3 \rightarrow e_2} \quad (E\text{-if1})
\]

\[
\frac{}{\text{if } false \text{ then } e_2 \text{ else } e_3 \rightarrow e_3} \quad (E\text{-if2})
\]

\[
\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2} \quad (E\text{-App1})
\]

\[
\frac{e_2 \rightarrow e_2'}{\nu_1 e_2 \rightarrow \nu_1 e_2'} \quad (E\text{-App2})
\]

\[
(\lambda x : t.e) \nu \rightarrow e[\nu / x] \quad (E\text{-AppAbs})
\]
**Typing**

\[ \Gamma \vdash e : t \]

\[
\frac{x : t \in \Gamma}{\Gamma \vdash \lnot x : t} \quad \text{(T-Var)}
\]

\[
\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad \text{(T-True)}
\]

\[
\frac{}{\Gamma \vdash \text{false} : \text{bool}} \quad \text{(T-False)}
\]

\[
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t} \quad \text{(T-If)}
\]

\[
\frac{\Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \lambda x : t_1 . e_2 : t_1 \to t_2} \quad \text{(T-Abs)}
\]

\[
\frac{\Gamma \vdash e_1 : t_{11} \to t_{12} \quad \Gamma \vdash e_2 : t_{11}}{\Gamma \vdash e_1 \ e_2 : t_{12}} \quad \text{(T-App)}
\]
Properties of Simply-Typed Lambda Calculus

Lemma 1 (Uniqueness of Typing). For every typing context \( \Gamma \) and expression \( e \), there exists at most one \( t \) such that \( \Gamma \vdash e : t \).

*(note: we don’t consider sub-typing here)*

Proof:
By induction on the derivation of \( \Gamma \vdash e : t \).

Case \( t \)-var: since there’s at most one \((x, t)\) pair in \( \Gamma \), \( x \) has either no type or one type \( t \). Case proved.

Case \( t \)-true and \( t \)-false: obviously true.

Case \( t \)-if:

\[
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}
\]

(1) \( t \) is unique

(By I.H.)
PROPERTIES OF SIMPLY-TYPED LAMBDA CALCULUS

Case t-abs:

\[ \frac{\Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \lambda x : t_1. e_2 : t_1 \rightarrow t_2} \]

(1) \( t_2 \) is unique  
(By I.H.)

(2) \( \Gamma \) contains just one \((x, t)\) pair so \( t_1 \) is unique  
(By (1) and assumption of t-abs)

(3) \( t_1 \rightarrow t_2 \) is unique  
(By (2) and t-abs)

Case t-app:

\[ \frac{\Gamma \vdash e_1 : t_1 \\
\Gamma \vdash e_2 : t_{11}}{\Gamma \vdash e_1 \ e_2 : t_{12}} \]

(1) \( e_1 \) and \( e_2 \) satisfies Lemma 1  
(By I.H.)

(2) There's at most one instance of \( t_{11} \)  
(By (1))

(3) \( t_{12} \) is unique, too  
(By (2))
Properties of Simply-Typed Lambda Calculus

Lemma 2 (Inversion for Typing).

- If $\Gamma \vdash x : t$ then $x : t \in \Gamma$
- If $\Gamma \vdash (\lambda x : t_1.e) : t$ then there is a $t_2$ such that $t = t_1 \rightarrow t_2$ and $\Gamma, x : t_1 \vdash e : t_2$
- If $\Gamma \vdash e_1 e_2 : t$ then there is a $t'$ such that $\Gamma \vdash e_1 : t' \rightarrow t$ and $\Gamma \vdash e_2 : t'$

Proof:
From the definition of the typing rules, there is only one rule for each type of expression, hence the result.

- **Well-typeness:** An expression $e$ in the language $L$ is said to be *well-typed*, if there exists some type $t$, such that $e : t$. 


Properties of Simply-Typed Lambda Calculus

Canonical Forms Lemma
(Idea: Given a type, want to know something about the shape of the value)
If \( \vdash v : t \) then
- If \( t = \text{bool} \) then \( v = \text{true} \) or \( v = \text{false} \);
- If \( t = t_1 \rightarrow t_2 \) then \( v = \lambda x : t_1. e \)

Proof:
By inspection of the typing rules.
Properties of Simply-Typed Lambda Calculus

Exchange Lemma
If G, x:t1, y:t2, G' | - e:t,
then G, y:t2, x:t1, G' | - e:t.

Proof by induction on derivation of
G, y:t, x:t, G' | - e:t
(Practice!)

Weakening Lemma
If G | - e:t then G, x:t' | - e:t (provided x not in Dom(G))
(Practice!)
**Type Safety**

- Safety = Progress + Preservation

- Progress: A well-type term is not stuck (either it is a value or it can take a step according to the evaluation rules)

- Preservation: If a well-typed term (with type $t$) takes a step of evaluation, then the resulting term is also well typed with type $t$.

- **Type-checking**: the process of verifying type safety of a program (or a term).
PROGRESS THEOREM

Suppose e is a closed and well-typed term (that is e : t for some t). Then either e is a value or else there is some e’ for which e \rightarrow e’.

Proof: By induction on the derivation of typing: [\Gamma \vdash e : t]
Case T-Var: doesn’t occur because e is closed.
Case T-True, T-False, T-Abs: immediate since these are values.
Case T-App:
(1) e_1 is a value or can take one step evaluation. Likewise for e_2. \hspace{1cm} (By I.H.)
(2) If e_1 can take a step, then E-App1 can apply to (e_1 e_2). \hspace{1cm} (By (1))
(3) If e_2 can take a step, then E-App2 can apply to (e_1 e_2) \hspace{1cm} (By (1))
(4) If both e_1 and e_2 are values, then e_1 must be an abstraction, therefore E-AppAbs can apply to (e_1 e_2) \hspace{1cm} (By (1) and canonical forms v)
(5) Hence (e_1 e_2) can always take a step forward. \hspace{1cm} (By (2,3,4))
PROGRESS THEOREM (CONT’D)

Case T-if:

1. e1 can either take a step or is a value (By I.H.)
2. Subcase 1: e1 can take a step (By I.H.)
   1. if e1 then e2 else e3 can take a step (By E-if0)
3. Subcase 2: e1 is a value (By I.H.)
   1. If e1 = true, if e1 then e2 else e3 \rightarrow e2 (By E-if1)
   2. If e1 = false, if e1 then e2 else e3 \rightarrow e3 (By E-if2)
4. In both subcases, e can take a step. Case proved.
**Preservation Theorem**

- If $G |- e : t$ and $e \rightarrow e'$, then $G |- e' : t$.

**Proof:** By induction on the derivation of $G |- e : t$.

Case $T$-$\text{Var}$, $T$-$\text{Abs}$, $T$-$\text{True}$, $T$-$\text{False}$:

Case doesn’t apply because variable or values can’t take one step evaluation.

Case $T$-$\text{If}$: $e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3$.

If $e \rightarrow e'$ there are two subcases cases:

Subcase 1: $e_1$ is not a value.

1. $e_1 : \text{bool}$ (By assumption)
2. $e_1 \rightarrow e_1'$ and $e_1' : \text{bool}$ (By IH)
3. $G |- e' : t$ (By $T$-$\text{If}$ and (2))

Subcase 2: $e_1$ is a value, i.e. either true or false.

4. $e \rightarrow e_2$ or $e \rightarrow e_3$ and $e' : t$ ($e' = e_2$ or $e_3$) (By $E$-$\text{If1}$, $E$-$\text{If2}$ and IH)

Case proved.
**Preservation Theorem (Cont’d)**

Case T-App: $e = e_1 e_2$. Need to prove, $G |- e' : t_{12}$

(5) Since $e_1 : t_{11} \rightarrow t_{12}$, $e_1$ is an abstraction. (By T-Abs)

There are two subcases for $e_2$.

Subcase 1: $e_2$ is a value. Let’s call it $v$.

(6) $e = \backslash x . e'' v$, and

\[
\begin{align*}
G & |- x : t_{11}, \\
G, x : t_{11} & |- e'' : t_{12}, \\
G & |- v : t_{11}
\end{align*}
\]

(7) $\backslash x. e'' v \rightarrow e'' [v / x]$ (By (5) and inversion of T-Abs)

(8) $G |- e''[v / x] : t_{12}$. (By (7) and substitution lemma)

(9) $G |- e' : t_{12}$ (By (8) & assumption)

Subcase 2: $e_2$ is not a value.

(10) Suppose $e_2 \rightarrow e_2'$. Then $e \rightarrow e_1 e_2'$, i.e., $e' = e_1 e_2'$. (By E-App2)

(11) $G |- e_2' : t_{11}$, therefore $G |- e_1 e_2' : t_{12}$. (By I.H., T-App)

(12) $G |- e' : t_{12}$. (By (11))

Case proved.

QED.
**Substitution Lemma**

If $G, x : t' \vdash e : t$, and $G \vdash v : t'$, then $G \vdash e[v/x] : t$.

Proof left as an exercise.
**Curry-Howard Correspondence**

- A.k.a *Curry-Howard Isomorphism*
- Connection between type theory and logic

<table>
<thead>
<tr>
<th>Logic</th>
<th>Programming Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositions</td>
<td>Types</td>
</tr>
<tr>
<td>Proposition $P \supset Q$</td>
<td>Type $P \rightarrow Q$</td>
</tr>
<tr>
<td>Proposition $P \land Q$</td>
<td>Type $P \times Q$ (product/pair type)</td>
</tr>
<tr>
<td>Proof of proposition $P$</td>
<td>Expression $e$ of type $P$</td>
</tr>
<tr>
<td>Proposition $P$ is provable</td>
<td>Type $P$ is inhabited (by some expression)</td>
</tr>
</tbody>
</table>