Any Questions?
Plan

“We have the technology…”

- In this lecture and the next, we’re going to cover some simple extensions of the typed-lambda calculus (TAPL Chapter 11).
  1. Products, records
  2. Sums, variants
  3. Recursion
- We’re skipping Chapters 10 and 12.
Erasure and Typability
Erasure

We can transform terms in $\lambda \rightarrow$ to terms of the untyped lambda-calculus simply by erasing type annotations on lambda-abstractions.

\[
\begin{align*}
erase(x) &= x \\
erase(\lambda x: T_1. \ t_2) &= \lambda x. \ erase(t_2) \\
erase(t_1 \ t_2) &= erase(t_1) \ erase(t_2)
\end{align*}
\]
Conversely, an untyped $\lambda$-term $m$ is said to be typable if there is some term $t$ in the simply typed lambda-calculus, some type $T$, and some context $\Gamma$ such that $\text{erase}(t) = m$ and $\Gamma \vdash t : T$.

This process is called type reconstruction or type inference.
Typability

Conversely, an untyped $\lambda$-term $m$ is said to be typable if there is some term $t$ in the simply typed lambda-calculus, some type $T$, and some context $\Gamma$ such that $\text{erase}(t) = m$ and $\Gamma \vdash t : T$.

This process is called type reconstruction or type inference.

Example: Is the term

$$\lambda x. \ x \ x$$

typable?
Intro vs. elim forms

An *introduction form* for a given type gives us a way of *constructing* elements of this type.

An *elimination form* for a type gives us a way of *using* elements of this type.
The Curry-Howard Correspondence

In constructive logics, a proof of $P$ must provide evidence for $P$.

- “law of the excluded middle” — $P \lor \neg P$ — not recognized.

A proof of $P \land Q$ is a pair of evidence for $P$ and evidence for $Q$.

A proof of $P \supset Q$ is a procedure for transforming evidence for $P$ into evidence for $Q$. 
## Propositions as Types

<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 $P \times Q$</td>
</tr>
<tr>
<td>proof of proposition $P$</td>
<td>term $t$ of type $P$</td>
</tr>
<tr>
<td>proposition $P$ is provable</td>
<td>type $P$ is inhabited (by some term)</td>
</tr>
<tr>
<td></td>
<td>evaluation</td>
</tr>
</tbody>
</table>
Propositions as Types

<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 $P \wedge Q$</td>
<td>type $P \times Q$</td>
</tr>
<tr>
<td>proof of proposition $P$</td>
<td>term $t$ of type $P$</td>
</tr>
<tr>
<td>proposition $P$ is provable</td>
<td>type $P$ is inhabited (by some term)</td>
</tr>
<tr>
<td>proof simplification</td>
<td>(a.k.a. “cut elimination”)</td>
</tr>
</tbody>
</table>
On to real programming languages...
Base types

Up to now, we’ve formulated “base types” (e.g. Nat) by adding them to the syntax of types, extending the syntax of terms with associated constants (zero) and operators (succ, etc.) and adding appropriate typing and evaluation rules. We can do this for as many base types as we like.

For more theoretical discussions (as opposed to programming) we can often ignore the term-level inhabitants of base types, and just treat these types as uninterpreted constants.
E.g., suppose B and C are some base types. Then we can ask (without knowing anything more about B or C) whether there are any types S and T such that the term

\[(\lambda f:S. \lambda g:T. f \ g) \ (\lambda x:B. \ x)\]

is well typed.
The **Unit type**

\[ t ::= ... \quad \text{terms} \]

\[ \text{unit} \quad \text{constant unit} \]

\[ v ::= ... \quad \text{values} \]

\[ \text{unit} \quad \text{constant unit} \]

\[ T ::= ... \quad \text{types} \]

\[ \text{Unit} \quad \text{unit type} \]

*New typing rules*

\[ \Gamma \vdash t : T \]

\[ \Gamma \vdash \text{unit} : \text{Unit} \quad (T\text{-UNIT}) \]
Sequencing

\[
t ::= \ldots \quad t_1; t_2
\]
Sequencing

\[ t ::= \ldots \]
\[ t_1; t_2 \]

\[
\frac{t_1 \rightarrow t_1'}{t_1; t_2 \rightarrow t_1'; t_2} \quad \text{(E-SEQ)}
\]

\[
\text{unit}; t_2 \rightarrow t_2 \quad \text{(E-SEQNEXT)}
\]

\[
\frac{\Gamma \vdash t_1 : \text{Unit} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1; t_2 : T_2} \quad \text{(T-SEQ)}
\]
Derived forms

- Syntactic sugar
- Internal language vs. external (surface) language
Sequencing as a derived form

\[ t_1; t_2 \overset{\text{def}}{=} (\lambda x: \text{Unit}. t_2) \ t_1 \]

where \( x \notin \text{FV}(t_2) \)
Equivalence of the two definitions

[board]
Ascription

New syntactic forms

\[ t ::= \ldots \]

\[ t \text{ as } T \]

New evaluation rules

\[ v_1 \text{ as } T \rightarrow v_1 \]  

\[ t_1 \rightarrow t_1' \]

\[ t_1 \text{ as } T \rightarrow t_1' \text{ as } T \]

New typing rules

\[ \Gamma \vdash t_1 : T \]

\[ \Gamma \vdash t_1 \text{ as } T : T \]
Ascription as a derived form

\[ t \text{ as } T ^ \text{def} (\lambda x : T. \ x) \ t \]
Let-bindings

New syntactic forms

\[ t ::= \ldots \]

\[ \text{let } x = t \text{ in } t \]

New evaluation rules

\[ \text{let } x = v_1 \text{ in } t_2 \rightarrow [x \mapsto v_1]t_2 \]  \hspace{1cm} (E-LETV)

\[ t_1 \rightarrow t'_1 \]

\[ \frac{t_1 \rightarrow t'_1}{\text{let } x = t_1 \text{ in } t_2 \rightarrow \text{let } x = t'_1 \text{ in } t_2} \]  \hspace{1cm} (E-LET)

New typing rules

\[ \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \]  \hspace{1cm} (T-LET)
Pairs, tuples, and records
Pairs

t ::= ... terms
   {t, t} pair
   t.1 first projection
   t.2 second projection

v ::= ... values
   {v, v} pair value

T ::= ... types
   T_1 \times T_2 product type
Evaluation rules for pairs

\[
\{v_1, v_2\}.1 \rightarrow v_1 \quad \text{(E-PairBeta1)} \\
\{v_1, v_2\}.2 \rightarrow v_2 \quad \text{(E-PairBeta2)} \\
\]

\[
t_1 \rightarrow t'_1 \\
\frac{t_1.1 \rightarrow t'_1.1}{(E-Proj1)} \\
\]

\[
t_1 \rightarrow t'_1 \\
\frac{t_1.2 \rightarrow t'_1.2}{(E-Proj2)} \\
\]

\[
t_1 \rightarrow t'_1 \\
\frac{\{t_1, t_2\} \rightarrow \{t'_1, t_2\}}{(E-Pair1)} \\
\]

\[
t_2 \rightarrow t'_2 \\
\frac{\{v_1, t_2\} \rightarrow \{v_1, t'_2\}}{(E-Pair2)} \\
\]
Typing rules for pairs

\[ \Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \]
\[ \Gamma \vdash \{t_1, t_2\} : T_1 \times T_2 \]
\[ (T\text{-PAIR}) \]

\[ \Gamma \vdash t_1 : T_{11} \times T_{12} \]
\[ \Gamma \vdash t_1.1 : T_{11} \]
\[ (T\text{-PROJ1}) \]

\[ \Gamma \vdash t_1 : T_{11} \times T_{12} \]
\[ \Gamma \vdash t_1.2 : T_{12} \]
\[ (T\text{-PROJ2}) \]
Tuples

t ::= \ldots 
  \{t_i \mid i \in 1..n\} 
  t.i 

t ::=

v ::= \ldots 
  \{v_i \mid i \in 1..n\} 

v ::=

t ::= \ldots 
  \{T_i \mid i \in 1..n\} 

T ::= \ldots 

terms 
tuple 
projection 
values 
tuple value 
types 
tuple type
Evaluation rules for tuples

\[
\{v_i \mid i \in 1..n\}.j \rightarrow v_j \quad \text{(E-PROJTUPLE)}
\]

\[
t_1 \rightarrow t'_1 \\
\frac{t_1}{t_1.i} \rightarrow t'_1.i
\quad \text{(E-PROJ)}
\]

\[
t_j \rightarrow t'_j \\
\frac{\{v_i \mid i \in 1..j-1, t_j, t_k \mid k \in j+1..n\}}{\{v_i \mid i \in 1..j-1, t'_j, t_k \mid k \in j+1..n\}}
\quad \text{(E-TUPLE)}
\]
Typing rules for tuples

for each $i$ $\Gamma \vdash t_i : T_i$

\[ \Gamma \vdash \{ t_i \}_{i \in 1..n} : \{ T_i \}_{i \in 1..n} \]  

(T-TUPLE)

\[ \Gamma \vdash t_1 : \{ T_i \}_{i \in 1..n} \]

\[ \Gamma \vdash t_1.j : T_j \]  

(T-PROJ)
Records

\[ t ::= \ldots \{ l_i := t_i \mid i \in 1..n \} \]
\[ t.l \]

terms

\[ v ::= \ldots \{ l_i := v_i \mid i \in 1..n \} \]

values

\[ T ::= \ldots \{ l_i : T_i \mid i \in 1..n \} \]

types

record

projection

record value

type of records
Evaluation rules for records

\[ \{l_i = v_i \mid i \in 1..n\}.l_j \rightarrow v_j \quad \text{(E-PROJRCD)} \]

\[ t_1 \rightarrow t'_1 \]
\[ \frac{t_1.l \rightarrow t'_1.l}{t_1.l \rightarrow t'_1.l} \quad \text{(E-PROJ)} \]

\[ t_j \rightarrow t'_j \]
\[ \frac{t_j \rightarrow t'_j}{\{l_i = v_i \mid i \in 1..j-1\}, l_j = t_j, l_k = t_k \mid k \in j+1..n\} \rightarrow \{l_i = v_i \mid i \in 1..j-1\}, l_j = t'_j, l_k = t_k \mid k \in j+1..n\}} \quad \text{(E-RCD)} \]
Typing rules for records

\[ \Gamma \vdash t_i : T_i \quad \text{for each } i \]

\[ \Gamma \vdash \{ l_i = t_i \mid i \in 1..n \} : \{ l_i : T_i \mid i \in 1..n \} \]

\[ (T-RCD) \]

\[ \Gamma \vdash t_1 : \{ l_i : T_i \mid i \in 1..n \} \]

\[ \Gamma \vdash t_1.l_j : T_j \]

\[ (T-PROJ) \]
Sums and variants
Sums – motivating example

PhysicalAddr = {firstlast: String, addr: String}
VirtualAddr  = {name: String, email: String}
Addr          = PhysicalAddr + VirtualAddr

\[ \text{inl} : \text{"PhysicalAddr} \rightarrow \text{PhysicalAddr}+\text{VirtualAddr"} \]
\[ \text{inr} : \text{"VirtualAddr} \rightarrow \text{PhysicalAddr}+\text{VirtualAddr"} \]

getName = \lambda a: Addr.
         case a of
            inl x \Rightarrow x.firstlast
            | inr y \Rightarrow y.name;
New syntactic forms

\[ t ::= \ldots \quad \text{terms} \]
\[ \text{inl } t \quad \text{tagging (left)} \]
\[ \text{inr } t \quad \text{tagging (right)} \]
\[ \text{case } t \text{ of } \text{inl } x \Rightarrow t \mid \text{inr } x \Rightarrow t \quad \text{case} \]

\[ v ::= \ldots \quad \text{values} \]
\[ \text{inl } v \quad \text{tagged value (left)} \]
\[ \text{inr } v \quad \text{tagged value (right)} \]

\[ T ::= \ldots \quad \text{types} \]
\[ T + T \quad \text{sum type} \]

\[ T_1 + T_2 \] is a disjoint union of \( T_1 \) and \( T_2 \) (the tags \text{inl} and \text{inr} ensure disjointness)
New evaluation rules

\[
\text{case (inl } v_0) \quad \rightarrow [x_1 \mapsto v_0] t_1 \quad \text{(E-CASEINL)}
\]

\[
\text{case (inr } v_0) \quad \rightarrow [x_2 \mapsto v_0] t_2 \quad \text{(E-CASEINR)}
\]

\[
t_0 \rightarrow t'_0
\]

\[
\frac{
\begin{array}{l}
\text{case } t_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2 \\
\rightarrow \text{case } t'_0 \text{ of inl } x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2
\end{array}
}{
\text{(E-CASE)}
}
\]

\[
\text{inl } t_1 \rightarrow \text{inl } t'_1
\]

\[
\text{(E-INL)}
\]

\[
\text{inr } t_1 \rightarrow \text{inr } t'_1
\]

\[
\text{(E-INR)}
\]
New typing rules

\[ \Gamma \vdash t : T \]

\[ \begin{align*}
\Gamma \vdash t_1 : T_1 \\
\Gamma \vdash \text{inl } t_1 : T_1 + T_2
\end{align*} \]

\[ (T\text{-INL}) \]

\[ \begin{align*}
\Gamma \vdash t_1 : T_2 \\
\Gamma \vdash \text{inr } t_1 : T_1 + T_2
\end{align*} \]

\[ (T\text{-INR}) \]

\[ \begin{align*}
\Gamma \vdash t_0 : T_1 + T_2 \\
\Gamma, x_1 : T_1 \vdash t_1 : T \\
\Gamma, x_2 : T_2 \vdash t_2 : T
\end{align*} \]

\[ \Gamma \vdash \text{case } t_0 \text{ of inl } x_1 \Rightarrow t_1 | \text{inr } x_2 \Rightarrow t_2 : T \]

\[ (T\text{-CASE}) \]
Sums and Uniqueness of Types

Problem:

If $t$ has type $T$, then $\text{inl } t$ has type $T+U$ for every $U$.

I.e., we’ve lost uniqueness of types.

Possible solutions:

- “Infer” $U$ as needed during typechecking
- Give constructors different names and only allow each name to appear in one sum type (requires generalization to “variants,” which we’ll see next) — OCaml’s solution
- Annotate each $\text{inl}$ and $\text{inr}$ with the intended sum type. For simplicity, let’s choose the third.
New syntactic forms

\[ t ::= \ldots \]
\[ \text{terms} \]
\[ \text{inl } t \text{ as } T \]
\[ \text{tagging (left)} \]
\[ \text{inr } t \text{ as } T \]
\[ \text{tagging (right)} \]

\[ v ::= \ldots \]
\[ \text{values} \]
\[ \text{inl } v \text{ as } T \]
\[ \text{tagged value (left)} \]
\[ \text{inr } v \text{ as } T \]
\[ \text{tagged value (right)} \]

Note that \textit{as } T \textit{ here is not the ascription operator that we saw before — i.e., not a separate syntactic form: in essence, there is an ascription “built into” every use of } \textbf{inl} \textit{ or } \textbf{inr}. \]
New typing rules

\[
\Gamma \vdash t : T \\
\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \text{inl } t_1 \text{ as } T_1 + T_2 : T_1 + T_2} \quad \text{(T-INL)}
\]

\[
\frac{\Gamma \vdash t_1 : T_2}{\Gamma \vdash \text{inr } t_1 \text{ as } T_1 + T_2 : T_1 + T_2} \quad \text{(T-INR)}
\]
Evaluation rules ignore annotations:

\[
\begin{align*}
\text{case (inl } v_0 \text{ as } T_0) & \rightarrow t_1 | \text{ inr } x_2 & \Rightarrow t_2 \\
& \rightarrow [x_1 \mapsto v_0]t_1 \\
\text{(E-CASEINL)}
\end{align*}
\]

\[
\begin{align*}
\text{case (inr } v_0 \text{ as } T_0) & \rightarrow t_1 | \text{ inl } x_2 & \Rightarrow t_2 \\
& \rightarrow [x_2 \mapsto v_0]t_2 \\
\text{(E-CASEINR)}
\end{align*}
\]

\[
\begin{align*}
\text{t}_1 & \rightarrow t'_1 \\
\text{inl } t_1 \text{ as } T_2 & \rightarrow \text{inl } t'_1 \text{ as } T_2 \\
\text{(E-INL)}
\end{align*}
\]

\[
\begin{align*}
\text{t}_1 & \rightarrow t'_1 \\
\text{inr } t_1 \text{ as } T_2 & \rightarrow \text{inr } t'_1 \text{ as } T_2 \\
\text{(E-INR)}
\end{align*}
\]
Variants

Just as we generalized binary products to labeled records, we can generalize binary sums to labeled *variants*. 
New syntactic forms

\[
\begin{align*}
t & ::= \ldots \\
  & <l=t> \text{ as } T \\
  & \text{case } t \text{ of } <l_i=x_i> \Rightarrow t_{i \in 1..n} \\
\end{align*}
\]

\[
\begin{align*}
T & ::= \ldots \\
  & <l_i:T_i>_{i \in 1..n} \\
\end{align*}
\]
New evaluation rules

\[
\text{case } \langle l_j=v_j \rangle \text{ as } T \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i \quad i \in 1..n \\
\quad \to [x_j \mapsto v_j] t_j
\]  
\hspace{2cm} (E-CASEVARIANT)

\[
\frac{t_0 \to t_0'}{\text{case } t_0 \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i \quad i \in 1..n} \\
\quad \to \text{case } t_0' \text{ of } \langle l_i=x_i \rangle \Rightarrow t_i \quad i \in 1..n
\]  
\hspace{2cm} (E-CASE)

\[
\frac{t_i \to t_i'}{\langle l_i=t_i \rangle \text{ as } T \to \langle l_i=t_i' \rangle \text{ as } T}
\]  
\hspace{2cm} (E-VARIANT)
New typing rules

\[ \Gamma \vdash t : T \]

\[ \Gamma \vdash t_j : T_j \]
\[ \Gamma \vdash \llbracket j \mapsto t_j \rrbracket \text{ as } \langle l_i : T_i \mid i \in 1..n \rangle : \langle l_i : T_i \mid i \in 1..n \rangle \]

\[ (T\text{-VARIANT}) \]

\[ \Gamma \vdash t_0 : \langle l_i : T_i \mid i \in 1..n \rangle \]
for each \( i \) \[ \Gamma, x_i : T_i \vdash t_i : T \]
\[ \Gamma \vdash \text{case } t_0 \text{ of } \llbracket l_j \mapsto x_j \rrbracket \Rightarrow t_i \mid i \in 1..n : T \]

\[ (T\text{-CASE}) \]
Example

Addr = <physical:PhysicalAddr, virtual:VirtualAddr>;

a = <physical=pa> as Addr;

getName = \(\lambda a:\text{Addr.}
\begin{align*}
\text{case } a \text{ of} \\
& \quad <\text{physical}=x> \Rightarrow x.\text{firstlast} \\
| & \quad <\text{virtual}=y> \Rightarrow y.\text{name};
\end{align*}
\)
Options

Just like in OCaml...

OptionalNat = <none:Unit, some:Nat>;

Table = Nat → OptionalNat;

emptyTable = \n:Nat. <none=unit> as OptionalNat;

extendTable = \\
  \t:Table. \m:Nat. \v:Nat. \\
    \n:Nat. \\
      if equal n m then <some=v> as OptionalNat \\
      else t n;

x = case t(5) of \\
    <none=u> ⇒ 999 \\
| <some=v> ⇒ v;
Enumerations

Weekday = <monday:Unit, tuesday:Unit, wednesday:Unit, thursday:Unit, friday:Unit>;

nextBusinessDay = \( w : \text{Weekday} \rightarrow \text{Weekday} \)
  case w of
    <monday=x> \Rightarrow <tuesday=unit> as Weekday
    | <tuesday=x> \Rightarrow <wednesday=unit> as Weekday
    | <wednesday=x> \Rightarrow <thursday=unit> as Weekday
    | <thursday=x> \Rightarrow <friday=unit> as Weekday
    | <friday=x> \Rightarrow <monday=unit> as Weekday;
Recursion
Recursion in $\lambda \rightarrow$

- In $\lambda \rightarrow$, all programs terminate. (Cf. Chapter 12.)
- Hence, untyped terms like $\text{omega}$ and $\text{fix}$ are not typable.
- But we can *extend* the system with a (typed) fixed-point operator...
Example

\[ ff = \lambda i : \text{Nat} \rightarrow \text{Bool}. \]
\[ \quad \lambda x : \text{Nat}. \]
\[ \quad \quad \text{if iszero } x \text{ then true } \]
\[ \quad \quad \text{else if iszero (pred } x \text{) then false } \]
\[ \quad \quad \text{else ie (pred (pred } x \text{))}; \]

\[ \text{iseven} = \text{fix } ff; \]

\[ \text{iseven 7}; \]
New syntactic forms

\[ t ::= \ldots \]

Fix \( t \)

\[ \text{terms} \]

fixed point of \( t \)

New evaluation rules

\[
\begin{align*}
\text{Fix } (\lambda x:T_1.t_2) & \quad \rightarrow [x \mapsto (\text{Fix } (\lambda x:T_1.t_2))]t_2 \\
(\text{E-FixBeta})
\end{align*}
\]

\[
\begin{align*}
t_1 & \rightarrow t'_1 \\
\hline
\text{Fix } t_1 & \rightarrow \text{Fix } t'_1 \\
(\text{E-Fix})
\end{align*}
\]
New typing rules

\[ \Gamma \vdash t : T \]

\[ \Gamma \vdash t_1 : T_1 \rightarrow T_1 \]

\[ \Gamma \vdash \text{fix } t_1 : T_1 \]

\((\text{T-Fix})\)
A more convenient form

\[ \text{letrec } x : T_1 = t_1 \text{ in } t_2 \overset{\text{def}}{=} \text{let } x = \text{fix } (\lambda x : T_1 . t_1) \text{ in } t_2 \]

\[ \text{letrec iseven : Nat} \rightarrow \text{Bool } = \]
\[ \lambda x : \text{Nat}. \]
\[ \text{if iszero } x \text{ then true} \]
\[ \text{else if iszero (pred } x) \text{ then false} \]
\[ \text{else iseven (pred (pred } x)) \]
\[ \text{in} \]
\[ \text{iseven 7}; \]