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 $\lambda \to$ to terms of the untyped lambda-calculus simply by erasing type annotations on lambda-abstractions.

\[
\begin{align*}
\text{erase}(x) &= x \\
\text{erase}(\lambda x : T_1 . t_2) &= \lambda x . \text{erase}(t_2) \\
\text{erase}(t_1 t_2) &= \text{erase}(t_1) \text{erase}(t_2)
\end{align*}
\]

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.
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 \( P \lor Q \) 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 ( P \lor Q )</td>
<td>types ( P \lor 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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logic</th>
<th>Programming languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>propositions ( P \lor Q )</td>
<td>types ( P \lor 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>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.

Derived forms

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

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

Equivalence of the two definitions

Ascription

New syntactic forms

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

\[ t \text{ as } T \quad \text{ascription} \]

New evaluation rules

\[ \frac{v_1 \text{ as } T \rightarrow v_1}{t_1 \rightarrow t'_1} \quad \text{(E-Ascribe)} \]

\[ \frac{t_1 \rightarrow t'_1 \quad t_1 \text{ as } T \rightarrow t'_1 \text{ as } T}{t_1 \text{ as } T \rightarrow t'_1 \text{ as } T} \quad \text{(E-Ascribe1)} \]

New typing rules

\[ \frac{\Gamma \vdash t : T}{\Gamma \vdash t_1 : T} \quad \text{(T-Ascribe)} \]

Ascription as a derived form

\[ t \text{ as } T \overset{\text{def}}{=} (\lambda x: T. x) \, t \]

Let-bindings

New syntactic forms

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

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

New evaluation rules

\[ \frac{\text{let } x = v_1 \text{ in } t_2 \rightarrow [x \mapsto v_1] t_2}{t_1 \rightarrow t'_1} \quad \text{(E-LetV)} \]

\[ \frac{t_1 \rightarrow t'_1 \quad \text{let } x = t_1 \text{ in } t_2 \rightarrow \text{let } x = t'_1 \text{ in } t_2}{t_1 \text{ in } t_2 \rightarrow \text{let } x = t'_1 \text{ in } t_2} \quad \text{(E-Let)} \]

New typing rules

\[ \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad \text{(T-Let)} \]

Pairs, tuples, and records
### Pairs

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>t ::= ...</td>
<td>terms</td>
</tr>
<tr>
<td>{t,t}</td>
<td>pair</td>
</tr>
<tr>
<td>t.1</td>
<td>first projection</td>
</tr>
<tr>
<td>t.2</td>
<td>second projection</td>
</tr>
<tr>
<td>v ::= ...</td>
<td>values</td>
</tr>
<tr>
<td>{v,v}</td>
<td>pair value</td>
</tr>
<tr>
<td>T ::= ...</td>
<td>types</td>
</tr>
<tr>
<td>T1 ×T2</td>
<td>product type</td>
</tr>
</tbody>
</table>

#### Evaluation rules for pairs

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{v_1,v_2}. 1 \rightarrow v_1</td>
<td>(E-PairBeta1)</td>
</tr>
<tr>
<td>{v_1,v_2}. 2 \rightarrow v_2</td>
<td>(E-PairBeta2)</td>
</tr>
<tr>
<td>t_1 \rightarrow t'_1</td>
<td>(E-Proj1)</td>
</tr>
<tr>
<td>t_1.1 \rightarrow t'_1.1</td>
<td></td>
</tr>
<tr>
<td>t_1 \rightarrow t'_1</td>
<td>(E-Proj2)</td>
</tr>
<tr>
<td>t_1.2 \rightarrow t'_1.2</td>
<td></td>
</tr>
<tr>
<td>{t_1,t_2} \rightarrow {t'_1,t_2}</td>
<td>(E-Pair1)</td>
</tr>
<tr>
<td>t_2 \rightarrow t'_2</td>
<td>(E-Pair2)</td>
</tr>
</tbody>
</table>

#### Typing rules for pairs

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\Gamma \vdash t_1 : T_1 \land \Gamma \vdash t_2 : T_2] \rightarrow [\Gamma \vdash {t_1,t_2} : T_1 \times T_2]</td>
<td>(T-Pair)</td>
</tr>
<tr>
<td>[\Gamma \vdash t_1 : T_{11} \times T_{12}] \rightarrow [\Gamma \vdash t_1.1 : T_{11}]</td>
<td>(T-Proj1)</td>
</tr>
<tr>
<td>[\Gamma \vdash t_1 : T_{11} \times T_{12}] \rightarrow [\Gamma \vdash t_1.2 : T_{12}]</td>
<td>(T-Proj2)</td>
</tr>
</tbody>
</table>

### Tuples

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>t ::= ...</td>
<td>terms</td>
</tr>
<tr>
<td>{t_i \mid i \in 1..n}</td>
<td>tuple</td>
</tr>
<tr>
<td>t.i</td>
<td>projection</td>
</tr>
<tr>
<td>v ::= ...</td>
<td>values</td>
</tr>
<tr>
<td>{v_i \mid i \in 1..n}</td>
<td>tuple value</td>
</tr>
<tr>
<td>T ::= ...</td>
<td>types</td>
</tr>
<tr>
<td>{T_i \mid i \in 1..n}</td>
<td>tuple type</td>
</tr>
</tbody>
</table>

#### Evaluation rules for tuples

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{v_i \mid i \in 1..n}. j \rightarrow v_j</td>
<td>(E-ProjTuple)</td>
</tr>
<tr>
<td>t_1 \rightarrow t'_1</td>
<td>(E-Proj)</td>
</tr>
<tr>
<td>t_1.1 \rightarrow t'_1.1</td>
<td></td>
</tr>
<tr>
<td>t_j \rightarrow t'_j</td>
<td>(E-Tuple)</td>
</tr>
<tr>
<td>{v_i \mid i \in 1..n}, j \rightarrow {v_i \mid i \in 1..n}, j</td>
<td>(E-Tuple)</td>
</tr>
</tbody>
</table>

#### Typing rules for tuples

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for each (i) [\Gamma \vdash t_i : T_i] \rightarrow [\Gamma \vdash {t_i \mid i \in 1..n} : {T_i \mid i \in 1..n}]</td>
<td>(T-Tuple)</td>
</tr>
<tr>
<td>[\Gamma \vdash t_1 : {T_i \mid i \in 1..n}] \rightarrow [\Gamma \vdash t_1.i : T_i]</td>
<td>(T-Proj1)</td>
</tr>
</tbody>
</table>
Records

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

Evaluation rules for records

\[ \{l_i = v_i \mid i \in 1..n\} t_j \rightarrow v_j \] (E-ProjRcd)
\[ t_1 \rightarrow t'_1 \]
\[ t_1.l \rightarrow t'_1.l \] (E-Proj)
\[ t_j \rightarrow t'_j \]
\[ \{l_i = v_i \mid i \in 1..j\}, l_j = t'_j, l_k = t_k \mid k \in j+1..n\} \rightarrow \{l_i = v_i \mid i \in 1..j\}, l_j = t'_j, l_k = t_k \mid k \in j+1..n\} \] (E-Rcd)

Typing rules for records

\[ \text{for each } i \quad \Gamma \vdash t_i : T_i \]
\[ \Gamma \vdash \{l_i = t_i \mid i \in 1..n\} : \{l_i : T_i \mid i \in 1..n\} \] (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 – motivating example

PhysicalAddr = \{firstlast:String, addr:String\}
VirtualAddr = \{name:String, email:String\}
Addr = PhysicalAddr + VirtualAddr
inl : “PhysicalAddr → PhysicalAddr+VirtualAddr”
inr : “VirtualAddr → PhysicalAddr+VirtualAddr”

getName = \(\lambda a: \text{Addr}.\)
\[ \text{case } a \text{ of} \]
\[ \text{inl } x \Rightarrow x.\text{firstlast} \]
\[ | \text{inr } y \Rightarrow y.\text{name}; \]

New syntactic forms

\[ t ::= \ldots \text{terms} \]
\[ \text{inl } t \text{ tagging (left)} \]
\[ \text{inr } t \text{ tagging (right)} \]
\[ \text{case } t \text{ of } \text{inl } x \Rightarrow t \mid \text{inr } x \Rightarrow t \text{ case} \]
\[ v ::= \ldots \text{values} \]
\[ \text{inl } v \text{ tagged value (left)} \]
\[ \text{inr } v \text{ tagged value (right)} \]
\[ T ::= \ldots \text{types} \]
\[ T+T \text{ sum type} \]

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

\[
\begin{align*}
\text{case (inl } v_0) \quad & \quad \longrightarrow [x_1 \mapsto v_0]t_1 \quad \text{(E-CaseInl)} \\
\text{of inl } x_1 \mapsto t_1 \mid \text{inr } x_2 \mapsto t_2 \\
\text{case (inr } v_0) \quad & \quad \longrightarrow [x_2 \mapsto v_0]t_2 \quad \text{(E-CaseInr)} \\
\text{of inl } x_1 \mapsto t_1 \mid \text{inr } x_2 \mapsto t_2 \\
\end{align*}
\]

**New typing rules**

\[
\begin{align*}
\Gamma \vdash t_1 : T_1 \\
\Gamma \vdash \text{inl } t_1 \text{ as } T_1 + T_2 \\
\Gamma \vdash t_1 : T_2 \\
\Gamma \vdash \text{inr } t_1 \text{ as } T_1 + T_2 \\
\Gamma \vdash \text{case } t_0 \text{ of inl } x_1 \mapsto t_1 | \text{inr } x_2 \mapsto t_2 : T \\
\end{align*}
\]

**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**

\[
\begin{align*}
t & ::= ... \quad \text{terms} \\
\text{inl } t & \text{ as } T \quad \text{tagging (left)} \\
\text{inr } t & \text{ as } T \quad \text{tagging (right)} \\
v & ::= ... \quad \text{values} \\
\text{inl } v & \text{ as } T \quad \text{tagged value (left)} \\
\text{inr } v & \text{ as } T \quad \text{tagged value (right)}
\end{align*}
\]

Note that as \( T \) 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 \( \text{inl} \) or \( \text{inr} \).

**Evaluation rules ignore annotations:**

\[
\begin{align*}
\text{case (inl } v_0 \text{ as } T_0) \quad & \quad \longrightarrow [x_1 \mapsto v_0]t_1 \\
\text{of inl } x_1 \mapsto t_1 | \text{inr } x_2 \mapsto t_2 \\
\text{case (inr } v_0 \text{ as } T_0) \quad & \quad \longrightarrow [x_2 \mapsto v_0]t_2 \\
\text{of inl } x_1 \mapsto t_1 | \text{inr } x_2 \mapsto t_2 \\
t_1 & \longrightarrow t_1' \\
\text{inl } t_1 \text{ as } T_2 & \longrightarrow \text{inl } t_1' \text{ as } T_2 \\
t_1 & \longrightarrow t_1' \\
\text{inr } t_1 \text{ as } T_2 & \longrightarrow \text{inr } t_1' \text{ as } T_2
\end{align*}
\]
Variants

Just as we generalized binary products to labeled records, we can generalize binary sums to labeled variants.

New syntactic forms

\[
t ::= \ldots \quad <l=t> \text{ as } T \\
\text{case } t \text{ of } <l_i=x_i> \Rightarrow t_i \quad i \in 1..n \\
T ::= \ldots \quad <l_i:T_i> \quad i \in 1..n
\]

New typing rules

\[
\Gamma \vdash t : T \\
\Gamma \vdash t_j : T_j \\
\Gamma \vdash <l_j=t_j> \text{ as } <l_i:T_i> \quad i \in 1..n
\]

Example

\[
\text{Addr} = \langle \text{physical:PhysicalAddr, virtual:VirtualAddr}\rangle; \\
\text{a} = \langle \text{physical=pa}\rangle \text{ as Addr}; \\
\text{getName} = \lambda a:\text{Addr.} \\
\quad \text{case } a \text{ of} \\
\quad \langle \text{physical=x}\rangle \Rightarrow x.\text{firstlast} \\
\quad | <\text{virtual=y}> \Rightarrow y.\text{name};
\]

Options

Just like in OCaml...

\[
\text{OptionalNat} = \langle \text{none:Unit, some:Nat}\rangle; \\
\text{Table} = \text{Nat} \rightarrow \text{OptionalNat}; \\
\text{emptyTable} = \lambda n:\text{Nat.} \quad \langle \text{none=unit}\rangle \text{ as } \text{OptionalNat}; \\
\text{extendTable} = \lambda t:\text{Table.} \quad \lambda m:\text{Nat.} \quad \lambda v:\text{Nat.} \\
\quad \lambda n:\text{Nat.} \\
\quad \quad \text{if equal } n \text{ m then } \langle \text{some=v}\rangle \text{ as } \text{OptionalNat} \\
\quad \quad \text{else } t_n; \\
\quad x = \text{case } t(5) \text{ of} \\
\quad \quad \langle \text{none=u}\rangle \Rightarrow 999 \\
\quad \quad | <\text{some=v}> \Rightarrow v;
\]
Enumerations

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

nextBusinessDay = \( \lambda w : \text{Weekday}. \)
  \cases{ \text{case } w \text{ of } <\text{monday}=x> \Rightarrow <\text{tuesday}=\text{unit}> \text{ as Weekday} \\
   | <\text{tuesday}=x> \Rightarrow <\text{wednesday}=\text{unit}> \text{ as Weekday} \\
   | <\text{wednesday}=x> \Rightarrow <\text{thursday}=\text{unit}> \text{ as Weekday} \\
   | <\text{thursday}=x> \Rightarrow <\text{friday}=\text{unit}> \text{ as Weekday} \\
   | <\text{friday}=x> \Rightarrow <\text{monday}=\text{unit}> \text{ 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

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

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

\[
\text{iseven } 7;
\]

New syntactic forms

\[
t ::= \ldots \quad \text{terms}
  \quad \text{fix } t \quad \text{fixed point of } t
\]

New evaluation rules

\[
\text{fix } (\lambda x: T_1. t_2) \quad \rightarrow [x \mapsto (\text{fix } (\lambda x: T_1. t_2))] t_2 \quad \text{(E-FixBeta)}
\]

\[
t_1 \quad \rightarrow t'_1
\]

\[
\text{fix } t_1 \quad \rightarrow \text{fix } t_1' \quad \text{(E-Fix)}
\]

New typing rules

\[
\Gamma \vdash t : T
\quad \Gamma \vdash t_1 : T_1 \rightarrow T_1
\quad \Gamma \vdash \text{fix } t_1 : T_1 \quad \text{(T-Fix)}
\]
A more convenient form

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

letrec iseven : Nat \rightarrow \text{Bool} =
  \lambda x : \text{Nat}.
  \begin{array}{l}
  \text{if iszero } x \text{ then true} \\
  \text{else if iszero (pred } x) \text{ then false} \\
  \text{else iseven (pred (pred } x))
  \end{array}
\]

in
iseven 7;