Let’s add simple types to λ-calculus...

Two syntactic changes from untyped λ-calculus:

- Require function arguments to be explicitly typed.
- Add a primitive type and value (e.g., unit).

\[
e ::= () \mid v \mid \lambda v: \tau . e \mid e_1 e_2
\]
\[
\tau ::= \text{unit} \mid \tau_1 \rightarrow \tau_2
\]

Now we need a static semantics:

\[
\Gamma : v \rightarrow \tau \quad \text{(typing contexts)}
\]
\[
\Gamma \vdash e : \tau \quad \text{(typing judgments)}
\]
Typing Rules

\[ \Gamma \vdash () : \text{unit} \]

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

\[ \Gamma \vdash \lambda v : \tau_1 . e : \tau_1 \rightarrow \tau_2 \]

\[ \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \]

\[ \Gamma \vdash e_1 e_2 : \tau_2 \]
Operational Semantics

Operational semantics are unchanged:

\[
\frac{e_1 \rightarrow_1 e'_1}{e_1 e_2 \rightarrow_1 e'_1 e_2}
\]

\[
(\lambda v: \tau. e_1) e_2 \rightarrow_1 e_1[e_2/v]^{(\beta\text{-reduction})}
\]

Called *simply-typed λ-calculus* (\(\rightarrow\))
More simply-typed $\lambda$-calculus

More simple types and operations commonly included in $\lambda_\to$:

\[
e ::= () \mid v \mid \lambda v:\tau.e \mid e_1 e_2
\]

\[
\mid n \mid e_1 \text{ aop } e_2
\]

\[
\mid \text{true} \mid \text{false} \mid e_1 \text{ bop } e_2
\]

\[
\mid e_1 \text{ cmp } e_2
\]

\[
\mid (e_1, e_2) \mid \pi_1 e \mid \pi_2 e
\]

\[
\mid \text{in}^{\tau_1+\tau_2}_1 e \mid \text{in}^{\tau_1+\tau_2}_2 e
\]

\[
\mid (\text{case } e \text{ of } \text{in}_1(v_1) \to e_1 \mid \text{in}_2(v_2) \to e_2)\]

\[
\tau ::= \text{unit} \mid \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \text{void}
\]

\]
Pairs

Pairs are like in OCaml:

- \((e_1, e_2)\) constructs a pair of values (any types)
- \(\pi_1\) extracts ("projects") the first value of a pair (like `fst` in OCaml)
- \(\pi_2\) projects second value (like `snd`)
- Pairs have type \(\tau_1 \times \tau_2\) (like \(\tau_1 \ast \tau_2\) in OCaml)

**Statics:**

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\frac{}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \\
\Gamma \vdash e : \tau_1 \times \tau_2 \\
\frac{}{i \in \{1, 2\} \quad \Gamma \vdash \pi_i e : \tau_i}
\]

**Large-step:**

\[
e_1 \Downarrow u_1 \quad e_2 \Downarrow u_2 \\
\frac{}{(e_1, e_2) \Downarrow (u_1, u_2)} \\
e \Downarrow (u_1, u_2) \\
\frac{}{i \in \{1, 2\} \quad \pi_i e \Downarrow u_i}
\]
Injections

Injections are like OCaml variant types:

- \( \text{in}_{1}^{\tau_{1} + \tau_{2}}(e) \) and \( \text{in}_{2}^{\tau_{1} + \tau_{2}}(e) \) are like writing \( \text{Constructor1}(e) \) and \( \text{Constructor2}(e) \) in OCaml, with the following type definition:

  \[
  \text{type t1_plus_t2 = Constructor1 of } \tau_{1} \mid \text{Constructor2 of } \tau_{2}
  \]

- Destruct injections with \( \text{case } e \text{ of in}_{1}(v_{1}) \rightarrow e_{1} \mid \text{in}_{2}(v_{2}) \rightarrow e_{2} \) does not really a limitation; just nest them (e.g., \( \tau_{1} + (\tau_{2} + (\tau_{3} + \cdots)) \)).

Statics:

\[
\begin{align*}
\Gamma \vdash e : \tau_{i} & \quad i \in \{1, 2\} \\
\Gamma \vdash \text{in}_{i}^{\tau_{1} + \tau_{2}} e : \tau_{1} + \tau_{2} \\
\Gamma \vdash e : \tau_{1} + \tau_{2} & \quad \Gamma[v_{1} \mapsto \tau_{1}] \vdash e_{1} : \tau \quad \Gamma[v_{2} \mapsto \tau_{2}] \vdash e_{2} : \tau \\
\Gamma \vdash (\text{case } e \text{ of in}_{1}(v_{1}) \rightarrow e_{1} \mid \text{in}_{2}(v_{2}) \rightarrow e_{2}) : \tau
\end{align*}
\]

Large-step:

\[
\begin{align*}
\text{if } e \Downarrow u \quad i \in \{1, 2\} & \quad \text{then } \text{in}_{i}^{\tau_{1} + \tau_{2}} e \Downarrow \text{in}_{i} u \\
\text{if } e \Downarrow u \quad i \in \{1, 2\} & \quad \text{then } e_{i}[u/v_{i}] \Downarrow u' \\
\text{(case } e \text{ of in}_{1}(v_{1}) \rightarrow e_{1} \mid \text{in}_{2}(v_{2}) \rightarrow e_{2}) \Downarrow u' & \quad \text{if } e \Downarrow u \quad i \in \{1, 2\}
\end{align*}
\]
Void type

\[ \tau ::= \text{unit} \mid \text{int} \mid \text{bool} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \text{void} \]

Catalog of simple types:

- () is the only value of type \text{unit}
- integers have type \text{int}
- booleans have type \text{bool}
- functions have type \( \tau_1 \to \tau_2 \)
- pairs have type \( \tau_1 \times \tau_2 \)
- injections have type \( \tau_1 + \tau_2 \)
- nothing has type \text{void}

Why would we want a valueless type like \text{void}?

One reason: Create opaque (uncallable) functions for encoding purposes.

Example: \( \lambda x: \text{void}.x \) is uncallable
Can encode Church numerals without risking expansion (e.g., \( \lambda x: \text{void}.x = 0_{\mathbb{N}}, (\text{false}, 0_{\mathbb{N}}) = 1_{\mathbb{N}}, \text{etc.} \))
Challenge: Can you write an infinite loop in $\lambda \to$?

First attempt: $(\lambda x:?.xx)(\lambda x:?.xx)$

But we need to fill in the types in order to have a legal term for $\lambda \to$. (And the term must be well-typed according to the static semantics!)
So we need types $\tau$ and $\tau'$ for which we can complete the following derivation:

$$\bot \vdash \lambda x:\tau.xx : \tau \to \tau'$$
Challenge: Can you write an infinite loop in \( \lambda \rightarrow \)?

First attempt: \((\lambda x : ? . xx)(\lambda x : ? . xx)\)

But we need to fill in the types in order to have a legal term for \( \lambda \rightarrow \). (And the term must be well-typed according to the static semantics!)

So we need types \( \tau \) and \( \tau' \) for which we can complete the following derivation:

\[
\begin{array}{c}
\frac{
\frac{
\frac{
\{ (x, \tau) \} \vdash xx : \tau'}{\bot \vdash \lambda x : \tau. xx : \tau \rightarrow \tau'}
}{\{ (x, \tau) \} \vdash xx : \tau'}
}{\bot \vdash \lambda x : \tau. xx : \tau \rightarrow \tau'}
\end{array}
\]

Conclusion: \( \tau = \tau \rightarrow \tau \) for some \( \tau' \).

Impossible! (\( \tau \) can't be bigger than itself!)
Strong Normalization

Challenge: Can you write an infinite loop in $\lambda \rightarrow$?

First attempt: $(\lambda x:?.xx)(\lambda x:?.xx)$

But we need to fill in the types in order to have a legal term for $\lambda \rightarrow$. (And the term must be well-typed according to the static semantics!)

So we need types $\tau$ and $\tau'$ for which we can complete the following derivation:

$$
\begin{array}{ccc}
\{ (x, \tau) \} \vdash x : \tau \rightarrow \tau' & \{ (x, \tau) \} \vdash x : \tau \\
\hline
\{ (x, \tau) \} \vdash xx : \tau' \\
\hline
\bot \vdash \lambda x:\tau.xx : \tau \rightarrow \tau'
\end{array}
$$

Conclusion: $\tau = \tau \rightarrow \tau'$ for some $\tau'$. Impossible! ($\tau$ can't be bigger than itself!)
Challenge: Can you write an infinite loop in $\lambda \to$?

First attempt: $(\lambda x : ? . xx)(\lambda x : ? . xx)$

But we need to fill in the types in order to have a legal term for $\lambda \to$. (And the term must be well-typed according to the static semantics!)

So we need types $\tau$ and $\tau'$ for which we can complete the following derivation:

$$
\begin{align*}
\{ (x, \tau) \} & \vdash x : \tau \to \tau' \\
\{ (x, \tau) \} & \vdash x : \tau \\
\{ (x, \tau) \} & \vdash xx : \tau' \\
\bot & \vdash \lambda x : \tau . xx : \tau \to \tau'
\end{align*}
$$

Conclusion: $\tau = \tau \to \tau'$ for some $\tau'$. 

Challenge: Can you write an infinite loop in $\lambda \rightarrow$?

First attempt: $(\lambda x:?.xx)(\lambda x:?.xx)$

But we need to fill in the types in order to have a legal term for $\lambda \rightarrow$. (And the term must be well-typed according to the static semantics!)

So we need types $\tau$ and $\tau'$ for which we can complete the following derivation:

$$
\frac{
\{ (x, \tau) \} \vdash x : \tau \rightarrow \tau' \\
\{ (x, \tau) \} \vdash x : \tau 
}{
\{ (x, \tau) \} \vdash xx : \tau' 
\frac{
\bot \vdash \lambda x:\tau.xx : \tau \rightarrow \tau'
}{
\tau = \tau \rightarrow \tau' \text{ for some } \tau'.
\text{Impossible! (}\tau\text{ can’t be bigger than itself!)}
}$$
Strong Normalization

Weird facts:

- It’s impossible to write a non-terminating loop in $\lambda \rightarrow$.
  - Full proof involves finding a *normal form* to which every term (eventually) reduces.
  - Languages with this property are called **strongly normalizing**.
- $\lambda \rightarrow$ is not Turing-complete.
  - How did merely adding some types lose so much power...?

How to fix?

One solution: Add a primitive `fix` operator...
Fixpoint Operator

Fixpoint operator \( \text{fix} \) acts like the Y-combinator:

\[
\begin{align*}
\text{Statics:} & \quad \Gamma \vdash e : (\tau \rightarrow \tau') \rightarrow (\tau \rightarrow \tau') \\
& \quad \Gamma \vdash \text{fix}(e) : \tau \rightarrow \tau'
\end{align*}
\]

\[
\begin{align*}
\text{Large-step:} & \quad e \Downarrow \lambda v:\tau.e_0 \\
& \quad e_0[\text{fix}(e)/v] \Downarrow u \\
& \quad \text{fix}(e) \Downarrow u
\end{align*}
\]

(Basis for \texttt{let rec} in OCaml)

Convention: From now on when we refer to “simply-typed \( \lambda \)-calculus (\( \lambda \rightarrow \))”, we will assume it includes all of the aforementioned operators \textbf{but not fix}. To add \texttt{fix}, we will say “simply-typed \( \lambda \)-calculus with fixpoints.”
Non-simple types

Extending $\lambda \rightarrow$ to non-simple types:

1. **parametric polymorphism ($\lambda_2$, also called System F)**
   - OCaml includes parametric polymorphism but not full System F.
   - Supported by Haskell and OCaml with recursive types extension

2. **parametrically polymorphic datatypes ($\lambda_\omega$)**
   - OCaml example: `type 'a tree = Empty | Node of ('a * 'a tree * 'a tree)`

3. **dependent types ($\lambda_\Pi$)**
   - not available in OCaml or Haskell
   - Recommended language: Gallina (Coq)

In this class we will only study formalisms for System F.
The $\lambda$-cube

\[
\begin{array}{c}
\lambda_\omega & \longrightarrow & \lambda_C \\
\lambda_2 & \longrightarrow & \lambda_\Pi \\
\lambda_\omega & \longrightarrow & \lambda_\Pi \omega \\
\lambda & \longrightarrow & \lambda_\Pi \\
\end{array}
\]
Intro to System F

Polymorphic abstractions are functions from types to terms:

\[(\Lambda \alpha.e)[\tau] \rightarrow_1 e[\tau/\alpha]\]
Polymorphic Function Examples

Example #1: Polymorphic identity function $\Lambda \alpha. \lambda x: \alpha. x$

$$(\Lambda \alpha. \lambda x: \alpha. x)[int](3) \rightarrow_1 (\lambda x: int. x)(3) \rightarrow_1 3$$

$$(\Lambda \alpha. \lambda x: \alpha. x)[bool](\text{false}) \rightarrow_1 (\lambda x: bool. x)(\text{false}) \rightarrow_1 \text{false}$$

Example #2: Polymorphic application function $\Lambda \alpha. \Lambda \beta. \lambda f: \alpha \rightarrow \beta. \lambda x: \alpha. f x$

$$(\Lambda \alpha. \Lambda \beta. \lambda f: \alpha \rightarrow \beta. \lambda x: \alpha. f x)[int][bool]((>)1)(3)$$

$\rightarrow_1 (\Lambda \beta. \lambda f: int \rightarrow \beta. \lambda x: int. f x)[bool]((>)1)(3)$$

$\rightarrow_1 (\lambda f: int \rightarrow bool. \lambda x: int. f x)((>)1)(3)$$

$\rightarrow_1 (\lambda x: int.((>)1x))(3)$$

$\rightarrow_1 (>13)$$

$\rightarrow_1 \text{false}$
Static Semantics of System F

\[ \Gamma \vdash e : \tau \quad \Gamma \vdash e : \forall \alpha.\tau' \]

\[ \Gamma \vdash \Lambda\alpha.e : \forall \alpha.\tau \quad \Gamma \vdash e[\tau] : \tau'[\tau/\alpha] \]

Example #1: Polymorphic identity function

\[(\Lambda\alpha.\lambda x:\alpha.x) \quad : \forall \alpha.(\alpha \to \alpha)\]

\[(\Lambda\alpha.\lambda x:\alpha.x)[int] \quad : int \to int\]

\[(\Lambda\alpha.\lambda x:\alpha.x)[int]3 \quad : int\]

Example #2: Polymorphic application function

\[(\Lambda\alpha.\Lambda\beta.\lambda f:\alpha\to\beta.\lambda x:\alpha.fx) \quad : \forall \alpha.\forall \beta.(\alpha \to \beta) \to \alpha \to \beta\]

\[(\Lambda\alpha.\Lambda\beta.\lambda f:\alpha\to\beta.\lambda x:\alpha fx)[int] \quad : \forall \beta.(\text{int} \to \beta) \to \text{int} \to \beta\]

\[(\Lambda\alpha.\Lambda\beta.\lambda f:\alpha\to\beta.\lambda x:\alpha fx)[int][bool] \quad : (\text{int} \to \text{bool}) \to \text{int} \to \text{bool}\]

\[(\Lambda\alpha.\Lambda\beta.\lambda f:\alpha\to\beta.\lambda x:\alpha fx)[int][bool][(\text{>}1)] \quad : \text{int} \to \text{bool}\]

\[(\Lambda\alpha.\Lambda\beta.\lambda f:\alpha\to\beta.\lambda x:\alpha fx)[int][bool]((\text{>}1)(3)) \quad : \text{bool}\]
**Type Inhabitation**

**Definition (type inhabitation):** A type $\tau$ is said to be inhabited if there exists a term $e$ having type $\tau$.

**Q:** Which System F types are not inhabited?
Definition (type inhabitation): A type $\tau$ is said to be inhabited if there exists a term $e$ having type $\tau$.

Q: Which System F types are not inhabited?

Are there any besides $void$?
Type Inhabitation

**Definition (type inhabitation):** A type $\tau$ is said to be *inhabited* if there exists a term $e$ having type $\tau$.

**Q:** Which System F types are not inhabited?

Are there any besides *void*?

Are there any that don’t have *void* in them at all?
Void Type

Convention: Since we don’t need `void` in System F to get an uninhabited type, from now on in System F, `void` is just an alias for $\forall \alpha. \alpha$:

\[
void = \forall \alpha. \alpha
\]
Exercise: Define an algorithm \( \mathcal{I} : \tau \rightarrow \{ T, F \} \) that decides whether any System F type \( \tau \) is inhabited.

\[
\begin{align*}
\mathcal{I}(\text{int}) &= T \\
\mathcal{I}(\text{bool}) &= T \\
\mathcal{I}(\text{unit}) &= ? \\
\mathcal{I}(\tau_1 \times \tau_2) &= ? \\
\mathcal{I}(\tau_1 + \tau_2) &= ? \\
\mathcal{I}(\tau_1 \rightarrow \tau_2) &= ? \\
\mathcal{I}(\forall \alpha. \tau) &= ?
\end{align*}
\]
Exercise: Define an algorithm $I : \tau \rightarrow \{T, F\}$ that decides whether any System F type $\tau$ is inhabited.

\[
\begin{align*}
I(int) &= T \\
I(bool) &= T \\
I(unit) &= T \\
I(\tau_1 \times \tau_2) &= I(\tau_1) \land I(\tau_2) \\
I(\tau_1 + \tau_2) &= I(\tau_1) \lor I(\tau_2) \\
I(\tau_1 \rightarrow \tau_2) &= I(\tau_1) \Rightarrow I(\tau_2) \\
I(\forall \alpha.\tau) &= \forall \alpha : bool, I(\tau)
\end{align*}
\]

*Implication $\Rightarrow$ here refers to intuitionistic implication, not classical implication from classical propositional logic. But in this class I will not give any problems for which the difference matters.*
Curry-Howard Isomorphism: The observation that there is a direct correspondence between the logic of computation (programs, types, etc.) and the logic of mathematics (proofs, propositions, etc.).

- Discovered by William Howard (U. Chicago, 1969) building upon work by Haskell Curry (Penn State, 1934)

- **propositions-as-types**: The operators of intuitionistic propositional logic correspond to the operators of typed \( \lambda \)-calculus.

- **proofs-as-programs**: A program is actually a proof of the theorem described by its type signature.

- Became the foundation for modern program-proof co-development and formal methods-based verification of computer programs
Exercise: Is the following type inhabited? If so, write a System F term having that type.

\[ \tau = \text{bool} \to (\text{int} \to \text{void}) \to \forall \alpha. (\alpha \times \alpha) \]

1. Turn \( \tau \) into a proposition using \( \mathcal{I} \).

\[ \mathcal{I}(\tau) = ? \]

2. If \( \mathcal{I}(\tau) = F \) then \( \tau \) is uninhabited, so we’re done; otherwise we must construct a term having type \( \tau \)…
Exercise: Is the following type inhabited? If so, write a System F term having that type.

\[ \tau = \text{bool} \to (\text{int} \to \text{void}) \to \forall \alpha. (\alpha \times \alpha) \]

1. Turn \( \tau \) into a proposition using \( \mathcal{I} \).

\[
\mathcal{I}(\tau) = T \Rightarrow (T \Rightarrow F) \Rightarrow \forall \alpha: \text{bool}. (\alpha \land \alpha) \\
= T \Rightarrow (F \Rightarrow \forall \alpha. (\alpha \land \alpha)) \\
= T \Rightarrow T \\
= T \text{ (so it’s inhabited)}
\]

2. If \( \mathcal{I}(\tau) = F \) then \( \tau \) is uninhabited, so we’re done; otherwise we must construct a term having type \( \tau \)...
Type-inhabitation Problem Walkthrough

\[ \tau = \text{bool} \rightarrow (\text{int} \rightarrow \text{void}) \rightarrow \forall \alpha. (\alpha \times \alpha) \]

Strategy for finding a System F term having type \( \tau \):

Each inhabited primitive type and each type operator has a primitive term or term operator that constructs it:

<table>
<thead>
<tr>
<th>Type</th>
<th>Term Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>()</td>
</tr>
<tr>
<td>int</td>
<td>\ldots, -1, 0, 1, 2, 3, \ldots</td>
</tr>
<tr>
<td>bool</td>
<td>true, false</td>
</tr>
<tr>
<td>\times</td>
<td>(e_1, e_2)</td>
</tr>
<tr>
<td>+</td>
<td>in_{1+2}^1(e) or in_{1+2}^2(e)</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>\lambda v: \tau. e</td>
</tr>
<tr>
<td>\forall</td>
<td>\Lambda \alpha. e</td>
</tr>
</tbody>
</table>

Using this approach for this \( \tau \) yields:

\[ \lambda x: \text{bool} . \lambda y: (\text{int} \rightarrow \text{void}) . \Lambda \alpha. (\alpha \times \alpha) \]
Type-inhabitation Problem Walkthrough

\[ \tau = \text{bool} \rightarrow (\text{int} \rightarrow \text{void}) \rightarrow \forall \alpha. (\alpha \times \alpha) \]

Strategy for finding a System F term having type \( \tau \):

Each inhabited primitive type and each type operator has a primitive term or term operator that constructs it:

<table>
<thead>
<tr>
<th>Type</th>
<th>Term Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>()</td>
</tr>
<tr>
<td>int</td>
<td>\ldots, -1, 0, 1, 2, 3, \ldots</td>
</tr>
<tr>
<td>bool</td>
<td>true, false</td>
</tr>
<tr>
<td>\times</td>
<td>(e_1, e_2)</td>
</tr>
<tr>
<td>+</td>
<td>in_{1+\tau_2}(e) or in_{2+\tau_2}(e)</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>\lambda v: \tau. e</td>
</tr>
<tr>
<td>\forall</td>
<td>\Lambda \alpha. e</td>
</tr>
</tbody>
</table>

Using this approach for this \( \tau \) yields:

\[ \lambda x: \text{bool}. \lambda y: (\text{int} \rightarrow \text{void}). \Lambda \alpha. (\alpha, \alpha) \]

Why is this not a valid System F term?
Type-inhabitation Problem Walkthrough

\[ \tau = \text{bool} \to (\text{int} \to \text{void}) \to \forall \alpha. (\alpha \times \alpha) \]

Strategy for finding a System F term having type \( \tau \):

Each inhabited primitive type and each type operator has a primitive term or term operator that constructs it:

<table>
<thead>
<tr>
<th>Type</th>
<th>Term Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>()</td>
</tr>
<tr>
<td>int</td>
<td>( \ldots, -1, 0, 1, 2, 3, \ldots )</td>
</tr>
<tr>
<td>bool</td>
<td>\text{true}, \text{false}</td>
</tr>
<tr>
<td>( \times )</td>
<td>((e_1, e_2))</td>
</tr>
<tr>
<td>( + )</td>
<td>\text{in}<em>{\tau_1 + \tau_2}(e) \text{ or in}</em>{\tau_1 + \tau_2}(e)</td>
</tr>
<tr>
<td>( \to )</td>
<td>(\lambda v: \tau. e)</td>
</tr>
<tr>
<td>( \forall )</td>
<td>(\Lambda \alpha. e)</td>
</tr>
</tbody>
</table>

Using this approach for this \( \tau \) yields:

\[ \lambda x: \text{bool}. \lambda y: (\text{int} \to \text{void}). \Lambda \alpha. ( , ) \]

How to fix?
Type-inhabitation Problem Walkthrough

\[ \tau = \text{bool} \to (\text{int} \to \text{void}) \to \forall \alpha. (\alpha \times \alpha) \]

Strategy for finding a System F term having type \( \tau \):

Each inhabited primitive type and each type operator has a primitive term or term operator that constructs it, and each type operator has a term operator to destruct it:

<table>
<thead>
<tr>
<th>Type</th>
<th>Term Constructor</th>
<th>Term Destructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>()</td>
<td>N/A</td>
</tr>
<tr>
<td>int</td>
<td>\ldots, -1, 0, 1, 2, 3, \ldots</td>
<td>N/A</td>
</tr>
<tr>
<td>bool</td>
<td>\text{true}, \text{false}</td>
<td>N/A</td>
</tr>
<tr>
<td>\times</td>
<td>(e_1, e_2)</td>
<td>\pi_1 e \text{ or } \pi_2 e</td>
</tr>
<tr>
<td>+</td>
<td>\text{in}_1^{\tau_1 + \tau_2}(e) \text{ or } \text{in}_2^{\tau_1 + \tau_2}(e)</td>
<td>\text{case } e \text{ of } \ldots</td>
</tr>
<tr>
<td>\to</td>
<td>\lambda v : \tau. e</td>
<td>e_1 e_2 \text{ (application)}</td>
</tr>
<tr>
<td>\forall</td>
<td>\Lambda \alpha. e</td>
<td>e[\tau] \text{ (instantiation)}</td>
</tr>
</tbody>
</table>

Using this approach for this \( \tau \) yields:

\[ \lambda x : \text{bool}. \lambda y : (\text{int} \to \text{void}). \Lambda \alpha . (y3[\alpha], y3[\alpha]) \]

Sanity check: Variable instances (\( y \) and \( \alpha \) in this case) nowhere appear free.
Type-inhabitation Problem Walkthrough

\[ \tau = bool \to (int \to void) \to \forall \alpha.(\alpha \times \alpha) \]

Each inhabited primitive type and each type operator has a primitive term or term operator that constructs it, and each type operator has a term operator to destruct it:

<table>
<thead>
<tr>
<th>Type</th>
<th>Term Constructor</th>
<th>Term Destructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>()</td>
<td>N/A</td>
</tr>
<tr>
<td>int</td>
<td>\ldots,-1,0,1,2,3,\ldots</td>
<td>N/A</td>
</tr>
<tr>
<td>bool</td>
<td>true, false</td>
<td>N/A</td>
</tr>
<tr>
<td>×</td>
<td>(e_1,e_2)</td>
<td>\pi_1 e or \pi_2 e</td>
</tr>
<tr>
<td>+</td>
<td>in_{1+2}^1(e) or in_{1+2}^2(e)</td>
<td>case e of \ldots</td>
</tr>
<tr>
<td>→</td>
<td>\lambda v:\tau.e</td>
<td>e_1 e_2 (application)</td>
</tr>
<tr>
<td>∀</td>
<td>\Lambda \alpha.e</td>
<td>\tau (instantiation)</td>
</tr>
</tbody>
</table>

A shorter solution:

\[ \lambda x:bool.\lambda y:(int \to void).y3[\forall \alpha.(\alpha \times \alpha)] \]

Take-away: Once you have an argument of uninhabited type, you have something very powerful that can create other uninhabited terms. (Curry-Howard: This corresponds to implicative explosion \( F \Rightarrow F \).)
Exercise: Is the following type inhabited? If so, write a System F term having that type.

\[ \tau = \forall \alpha. \forall \beta. ((\alpha + \beta) \rightarrow (\beta + \alpha)) \]

Step 1: Decide whether \( \mathcal{I}(\tau) \) is tautological:
Sample Type-inhabitation Problem

Exercise: Is the following type inhabited? If so, write a System F term having that type.

\[ \tau = \forall \alpha. \forall \beta.((\alpha + \beta) \rightarrow (\beta + \alpha)) \]

Step 1: Decide whether \( \mathcal{I}(\tau) \) is tautological:

\[ \mathcal{I}(\tau) = \forall \alpha. \forall \beta.((\alpha \lor \beta) \Rightarrow (\beta \lor \alpha)) \]
Sample Type-inhabitation Problem

Exercise: Is the following type inhabited? If so, write a System F term having that type.

\[ \tau = \forall \alpha. \forall \beta. ((\alpha + \beta) \rightarrow (\beta + \alpha)) \]

Step 1: Decide whether \( I(\tau) \) is tautological:

\[ I(\tau) = \forall \alpha. \forall \beta. ((\alpha \lor \beta) \Rightarrow (\beta \lor \alpha)) \]

Step 2: If \( I(\tau) \) is tautological, build a term of type \( \tau \) using constructors and destructors.
Exercise: Is the following type inhabited? If so, write a System F term having that type.

\[ \tau = \forall \alpha. \forall \beta. ((\alpha + \beta) \rightarrow (\beta + \alpha)) \]

Step 1: Decide whether \( I(\tau) \) is tautological:

\[ I(\tau) = \forall \alpha. \forall \beta. ((\alpha \lor \beta) \Rightarrow (\beta \lor \alpha)) \]

Step 2: If \( I(\tau) \) is tautological, build a term of type \( \tau \) using constructors and destructors.

\[ \Lambda \alpha. \Lambda \beta. \lambda x: \alpha + \beta. ? \]
Exercise: Is the following type inhabited? If so, write a System F term having that type.

$$\tau = \forall \alpha. \forall \beta.((\alpha + \beta) \to (\beta + \alpha))$$

Step 1: Decide whether $I(\tau)$ is tautological:

$$I(\tau) = \forall \alpha. \forall \beta.((\alpha \lor \beta) \Rightarrow (\beta \lor \alpha))$$

Step 2: If $I(\tau)$ is tautological, build a term of type $\tau$ using constructors and destructors.

$$\Lambda \alpha. \Lambda \beta. \lambda x : \alpha + \beta. \text{case } x \text{ of } \text{in}_1(y) \to ? \quad | \quad \text{in}_2(z) \to ?$$
Sample Type-inhabitation Problem

Exercise: Is the following type inhabited? If so, write a System F term having that type.

\[ \tau = \forall \alpha. \forall \beta. ((\alpha + \beta) \rightarrow (\beta + \alpha)) \]

Step 1: Decide whether \( I(\tau) \) is tautological:

\[ I(\tau) = \forall \alpha. \forall \beta. ((\alpha \lor \beta) \Rightarrow (\beta \lor \alpha)) \]

Step 2: If \( I(\tau) \) is tautological, build a term of type \( \tau \) using constructors and destructors.

\[ \Lambda \alpha. \Lambda \beta. \lambda x: \alpha + \beta. \text{case } x \text{ of } \text{in}_1(y) \rightarrow \text{in}_2^{\beta+\alpha} y | \text{in}_2(z) \rightarrow \text{in}_1^{\beta+\alpha} z \]
Exercise: Are the following types inhabited? If so, write terms having these types.

\[ \tau_1 = \forall \alpha. (\alpha \rightarrow \text{void}) \quad \tau_2 = (\forall \alpha. \alpha) \rightarrow \text{void} \]

\[ \mathcal{I}(\tau_1) = \forall \alpha. (\alpha \Rightarrow F) \]

\[ \mathcal{I}(\tau_2) = (\forall \alpha. \alpha) \Rightarrow F \]
Exercise: Are the following types inhabited? If so, write terms having these types.

\[ \tau_1 = \forall \alpha. (\alpha \to \text{void}) \]
\[ \tau_2 = (\forall \alpha. \alpha) \to \text{void} \]
\[ \mathcal{I}(\tau_1) = \forall \alpha. (\alpha \Rightarrow F) \]
\[ \mathcal{I}(\tau_2) = (\forall \alpha. \alpha) \Rightarrow F \]
\[ = F \text{ (because } T \not\Rightarrow F) \]
\[ = F \Rightarrow F \]
\[ = T \]
Exercise: Are the following types inhabited? If so, write terms having these types.

\[ \tau_1 = \forall \alpha. (\alpha \rightarrow \text{void}) \]
\[ I(\tau_1) = \forall \alpha. (\alpha \Rightarrow F) \]
\[ = F \text{ (because } T \nRightarrow F) \]

\[ \tau_2 = (\forall \alpha. \alpha) \rightarrow \text{void} \]
\[ I(\tau_2) = (\forall \alpha. \alpha) \Rightarrow F \]
\[ = F \Rightarrow F \]
\[ = T \]

\[ (\lambda x: \text{void}. x) : (\forall \alpha. \alpha) \rightarrow \text{void} \]
Brokenness of fix

The fix operator must not be added lest the isomorphism break down.

Recall the typing rule for fix:

\[ \Gamma \vdash e : (\tau \to \tau') \to (\tau \to \tau') \]
\[ \Gamma \vdash \text{fix}(e) : \tau \to \tau' \]

With it we can derive:

\[ \{ (x, \text{unit} \to \text{void}) \} \vdash x : \text{unit} \to \text{void} \]
\[ \bot \vdash \lambda x:\text{unit}\to\text{void}.x : (\text{unit} \to \text{void}) \to (\text{unit} \to \text{void}) \]
\[ \bot \vdash \text{fix}(\lambda x:\text{unit}\to\text{void}.x) : \text{unit} \to \text{void} \]
\[ \bot \vdash \text{fix}(\lambda x:\text{unit}\to\text{void}.x)() : \text{void} \]
C-H Isomorphism and Derivation Rule Soundness

Two ways to understand the problem:

- \( e : \tau \) is like saying “\( e \) promises to return a \( \tau \).” But \( e \) breaks its promise if \( e \) is an infinite loop.

- \( e : \tau \) is like saying \( e \) is a \textbf{proof} of proposition \( \tau \). But the typing rule for \textit{fix} is unsound, so not a valid proof:

\[
\begin{align*}
\Gamma \vdash e : (\tau \to \tau') \to (\tau \to \tau') \\
\Gamma \vdash \text{fix}(e) : \tau \to \tau'
\end{align*}

\[
\frac{\frac{\Gamma \vdash e : (\tau \to \tau') \to (\tau \to \tau')}{\Gamma \vdash \text{fix}(e) : \tau \to \tau'} = \frac{(\tau \Rightarrow \tau') \Rightarrow (\tau \Rightarrow \tau')}{\tau \Rightarrow \tau'}
\]

Big idea: Typing rules are actually the rules of deductive propositional logic.

See Coq and Calculus of Inductive Constructions for much more on this.
Type Annotations

**Definition (type annotations):** In the syntax of System F, all mentions of types \( \tau \) (e.g., \( \lambda v : \tau . e \)), type variable binders (e.g., \( \Lambda \alpha . e \)), and type instantiations (e.g., \( e [\tau] \)) are called *type annotations*.

**Type-inference:** Given a System F term \( \hat{e} \) without any annotations, infer an annotated term \( e \) that is well-typed (if one exists).

**Type-checking:** Given a System F term \( e \), decide whether there exists a type \( \tau \) such that \( \bot \vdash e : \tau \) is derivable.

**Good news and bad news:**
- Type-checking is decidable for both \( \lambda \to \) and System F.
- Type-inference is decidable for \( \lambda \to \).
- Type-inference is undecidable for System F. 😞
Definition (shallow type): A type $\tau$ is shallow if no quantifiers are children of non-quantifiers in $\tau$’s AST.

Examples:
- $\text{int} \rightarrow \text{unit}$ is shallow (no quantifiers).
- $\forall \alpha.\forall \beta. (\beta \rightarrow \alpha)$ is shallow (both quantifiers at top of AST).
- $\forall \alpha. (\forall \beta. \beta) \rightarrow \alpha$ is not shallow ($\forall \beta$ is a child of $\rightarrow$).
- $(\forall \alpha. \alpha) \times (\forall \beta. \beta)$ is not shallow ($\forall \alpha$ and $\forall \beta$ are both children of $\times$).

If we limit System F to shallow types only, type-inference becomes decidable. 😊

Example:  
```
let apply f x = f x;;
apply = $\Lambda \alpha. \Lambda \beta. \lambda f: \alpha \rightarrow \beta. \lambda x: \alpha. (fx)$
```

```
let y = apply ((>)1) 5;;
y = apply[int][bool][(>)1]5
```
Hindley-Milner Type-inference

A representative core fragment of unannotated System F:

\[ \hat{e} ::= () \mid v \mid \lambda v.\hat{e} \mid \hat{e}_1\hat{e}_2 \]

Four steps:

1. Change unannotated term \( \hat{e} \) into an annotated but non-closed System F term \( e \) by adding unique, free type variables:

   \[ \lambda v.\hat{e} \leadsto \lambda v:\alpha.e \]
   \[ v \leadsto v[\alpha_1] \cdots [\alpha_n] \text{ when } \Gamma(v) = \forall\alpha_1 \ldots \forall\alpha_n.\tau \]

2. Infer a mapping \( \theta : \alpha \rightarrow \tau \) from the free type variables to their types (details next slides).

3. Substitute any type variables \( \alpha \in \theta^\leftarrow \) appearing free in \( e \) with their types \( \theta(\alpha) \).

4. There may still be some free type variables \( \alpha \) in \( e \). If so, add \( \Lambda\alpha \) to the start of \( e \) for each one to bind them (yielding a term of shallow type).
Hindley-Milner Type-inference

The main algorithm (step 2) can be expressed as a derivation of a judgment:

$$\theta, \Gamma \vdash e : \tau, \theta'$$

- $\theta : \alpha \rightarrow \tau$ maps type vars $\alpha$ whose types we’ve already inferred to their types $\tau$.
- $\Gamma : v \rightarrow \tau$ maps program variables $v$ to their types $\tau$.
- $e$ is the expression on which we are performing type-inference.
- $\tau$ is the type inferred for $e$.
- $\theta' : \alpha \rightarrow \tau$ records any new types $\tau$ we’ve inferred for free type variables $\alpha$ appearing in $e$.

Notations:

- $\tau[\theta]$ is capture-avoiding substitution of type vars $\alpha$ in $\tau$ with their types $\theta(\alpha)$.
- $\Gamma[\theta] = \{(v, \tau[\theta]) \mid \Gamma(v) = \tau\}$ is the same substitution in the image of $\Gamma$. 
Hindley-Milner Type-inference

\[
\begin{align*}
\theta, \Gamma \vdash () : \text{unit}, \theta \\
\Gamma(v) = \forall \beta_1 \ldots \forall \beta_n. \tau \\
\theta, \Gamma \vdash v[\alpha_1] \ldots [\alpha_n] : [\tau[\alpha_1/\beta_1] \ldots [\alpha_n/\beta_n]], \theta \\
\theta, \Gamma [v \mapsto \alpha] \vdash e : \tau, \theta' \\
\theta, \Gamma \vdash \lambda v : \alpha. e : \alpha \rightarrow \tau, \theta' \\
\theta, \Gamma \vdash e_1 : \tau_1, \theta_1 \quad \theta_1, \Gamma[\theta_1] \vdash e_2 : \tau_2, \theta_2 \quad \theta_3 = \mathcal{U}(\tau_1[\theta_2], \tau_2 \rightarrow \alpha) \quad \theta' = \theta_2 \sqcup \theta_3 \\
\theta, \Gamma \vdash e_1e_2 : \theta'(\alpha), \theta'
\end{align*}
\]
Type-inference for Function Application

\[ \theta, \Gamma \vdash e_1 : \tau_1, \theta_1 \quad \theta_1, \Gamma[\theta_1] \vdash e_2 : \tau_2, \theta_2 \quad \theta_3 = \mathcal{U}(\tau_1[\theta_2], \tau_2 \to \alpha) \quad \theta' = \theta_2 \sqcup \theta_3 \]

\[ \theta, \Gamma \vdash e_1 e_2 : \theta'(\alpha), \theta' \]

1. Infer a type \( \tau_1 \) for \( e_1 \).
2. Infer a type \( \tau_2 \) for \( e_2 \).
3. Types \( \tau_1 \) and \( \tau_2 \to \alpha \) must be identical (for some \( \alpha \)). **Unify** them:

**Definition (type unification):** The *unification* of types \( \tau_1 \) and \( \tau_2 \) is an instantiation \( \theta : \alpha \to \tau \) of their type variables that causes them to be identical:

\[ \mathcal{U}(\alpha, \alpha) = \bot \]
\[ \mathcal{U}(\text{unit}, \text{unit}) = \bot \]
\[ \mathcal{U}(\alpha, \tau) = \mathcal{U}(\tau, \alpha) = \{(\alpha, \tau)\} \text{ if } \alpha \text{ is not free in } \tau \]
\[ \mathcal{U}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') = \mathcal{U}(\tau_1, \tau_1') \sqcup \mathcal{U}(\tau_2, \tau_2') \]
\[ \mathcal{U} \text{ is undefined otherwise (type-inference rejects)} \]
Non-shallow Types

H-M type-inference only works on shallow-typed terms.

Optional Exercise: Come up with an OCaml program whose type is non-shallow, and try compiling it. What error does OCaml report?

Follow-up Optional Exercise: Use OCaml’s (experimental) --rectypes option to add non-shallow typing support (sacrifices full type-inference) and fix your program above.
Summary of λ-cube

- $\lambda \to$: simply-typed λ-calculus (no type quantifiers)
- $\lambda_2$ (System F): parametric polymorphism
- $\lambda_\omega$: parametrically polymorphic datatypes
  - OCaml is essentially $(\lambda_\omega \cap \text{shallow types}) \cup \text{fix}$
  - Haskell is essentially $\lambda_\omega \cup \text{fix}$
- $\lambda_{\Pi}$: dependent types (correspond to $\exists$ in propositional logic)
- $\lambda_C$: Calculus of Constructions (combines all)
  - Coq/Gallina is essentially $\lambda_C$