Typed λ-calculus

CS 4301/6371: Advanced Programming Languages

Kevin W. Hamlen

April 4, 2024

Syntax additions

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).

$$e ::= () \mid v \mid \lambda v{:}\tau.e \mid e_1 e_2$$
$$\tau ::= \texttt{unit} \mid \tau_1 \to \tau_2$$

Now we need a static semantics:

$$\Gamma : v \rightharpoonup \tau \qquad \text{(typing contexts)}$$
$$\Gamma \vdash e : \tau \qquad \text{(typing judgments)}$$

Typing Rules

$$\frac{}{\Gamma \vdash \texttt{()} : \texttt{unit}}$$

$$\frac{}{\Gamma \vdash v : \Gamma(v)}$$

$$\frac{\Gamma[v \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda v{:}\tau_1 . e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}$$

Operational Semantics

Operational semantics are unchanged:

$$\frac{e_1 \rightarrow_1 e_1'}{e_1 e_2 \rightarrow_1 e_1' e_2}$$

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

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

More simply-typed λ-calculus

More simple types and operations commonly included in $\lambda_{\rightarrow}$:

$e ::= () \mid v \mid \lambda v{:}\tau.e \mid e_1 e_2$	(as before)
$\mid n \mid e_1 \; aop \; e_2$	integers
$\mid \mathtt{true} \mid \mathtt{false} \mid e_1 \; bop \; e_2$	booleans
$\mid e_1 \; cmp \; e_2$	int comparisons
$\mid (e_1, e_2) \mid \pi_1 e \mid \pi_2 e$	pairs
$\mid \mathtt{in}_1^{\tau_1+\tau_2} e \mid \mathtt{in}_2^{\tau_1+\tau_2} e$	injections
$\mid (\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_1(v_1) \rightarrow e_1 \ \mathtt{\mid}\ \mathtt{in}_2(v_2) \rightarrow e_2)$	case distinction
$\tau ::= unit \mid int \mid bool \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid void$	types

Pairs

Pairs are like in OCaml:

- (e_1, e_2) constructs a pair of values (any types)
- π_1 extracts ("projects") the first value of a pair (like `fst` in OCaml)
- π_2 projects second value (like `snd`)
- Pairs have type $\tau_1 \times \tau_2$ (like $\tau_1 * \tau_2$ in OCaml)

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

Large-step:
$$\frac{e_1 \Downarrow u_1 \quad e_2 \Downarrow u_2}{(e_1, e_2) \Downarrow (u_1, u_2)} \qquad \frac{e \Downarrow (u_1, u_2) \quad i \in \{1, 2\}}{\pi_i e \Downarrow u_i}$$

Injections

Injections are like OCaml variant types:

- $\texttt{in}_1^{\tau_1+\tau_2}(e)$ and $\texttt{in}_2^{\tau_1+\tau_2}(e)$ are like writing $\texttt{Constructor1}(e)$ and $\texttt{Constructor2}(e)$ in OCaml, with the following type definition:

  $$\texttt{type t1\_plus\_t2 = Constructor1 of } \tau_1 \texttt{ | Constructor2 of } \tau_2$$

- Destruct injections with $(\texttt{case } e \texttt{ of } \texttt{in}_1(v_1) \to e_1 \texttt{ | } \texttt{in}_2(v_2) \to e_2)$
  - Works like `match–with` in OCaml
- Injections have type $\tau_1 + \tau_2$
- Restriction to only two variants is not really a limitation; just nest them (e.g., $\tau_1 + (\tau_2 + (\tau_3 + \cdots)))$.

Statics: $\dfrac{\Gamma \vdash e : \tau_i \quad i \in \{1,2\}}{\Gamma \vdash \texttt{in}_i^{\tau_1+\tau_2} e : \tau_1 + \tau_2}$ $\quad \dfrac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma[v_1 \mapsto \tau_1] \vdash e_1 : \tau \quad \Gamma[v_2 \mapsto \tau_2] \vdash e_2 : \tau}{\Gamma \vdash (\texttt{case } e \texttt{ of } \texttt{in}_1(v_1) \to e_1 \texttt{ | } \texttt{in}_2(v_2) \to e_2) : \tau}$

Large-step: $\dfrac{e \Downarrow u \quad i \in \{1,2\}}{\texttt{in}_i^{\tau_1+\tau_2} e \Downarrow \texttt{in}_i u}$ $\quad \dfrac{e \Downarrow \texttt{in}_i u \qquad e_i[u/v_i] \Downarrow u' \qquad i \in \{1,2\}}{(\texttt{case } e \texttt{ of } \texttt{in}_1(v_1) \to e_1 \texttt{ | } \texttt{in}_2(v_2) \to e_2) \Downarrow u'}$

Void type

$$\tau ::= \mathit{unit} \mid \mathit{int} \mid \mathit{bool} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mathit{void}$$

Catalog of simple types:

- () is the only value of type unit
- integers have type int
- booleans have type 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 void

Why would we want a valueless type like void?

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

Example: $\lambda x{:}\mathit{void}.x$ is uncallable
Can encode Church numerals without risking expansion (e.g., $\lambda x{:}\mathit{void}.x = 0_{\mathbb{N}}$, $(\mathtt{false}, 0_{\mathbb{N}}) = 1_{\mathbb{N}}$, etc.)

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 τ and τ' for which we can complete the following derivation:

$$\frac{}{\perp \vdash \lambda x{:}\tau.xx : \tau \rightarrow \tau'}$$

Strong Normalization

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 τ and τ' for which we can complete the following derivation:

$$\dfrac{\overline{\{(x,\tau)\} \vdash xx : \tau'}}{\perp \vdash \lambda x{:}\tau.xx : \tau \to \tau'}$$

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 τ and τ' for which we can complete the following derivation:

$$\dfrac{\dfrac{\{(x,\tau)\} \vdash x : \tau \rightarrow \tau' \qquad \{(x,\tau)\} \vdash x : \tau}{\{(x,\tau)\} \vdash xx : \tau'}}{\perp \vdash \lambda x{:}\tau.xx : \tau \rightarrow \tau'}$$

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 τ and τ' for which we can complete the following derivation:

$$\dfrac{\dfrac{\{(x,\tau)\} \vdash x : \tau \rightarrow \tau' \qquad \{(x,\tau)\} \vdash x : \tau}{\{(x,\tau)\} \vdash xx : \tau'}}{\perp \vdash \lambda x{:}\tau.xx : \tau \rightarrow \tau'}$$

Conclusion: $\tau = \tau \rightarrow \tau'$ for some τ'.

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 τ and τ' for which we can complete the following derivation:

$$\dfrac{\dfrac{\{(x,\tau)\} \vdash x : \tau \rightarrow \tau' \qquad \{(x,\tau)\} \vdash x : \tau}{\{(x,\tau)\} \vdash xx : \tau'}}{\perp \vdash \lambda x{:}\tau.xx : \tau \rightarrow \tau'}$$

Conclusion: $\tau = \tau \rightarrow \tau'$ for some τ'.
Impossible! (τ can't be bigger than itself!)

Strong Normalization

Weird facts:

- It's impossible to write a non-terminating loop in $\lambda_{\to}$.
  - Full proof involves finding a *normal form* to which every term (eventually) reduces.
  - Languages with this property are called **strongly normalizing**.
- $\lambda_{\to}$ 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 `fix` acts like the Y-combinator:

Statics: $$\frac{\Gamma \vdash e : (\tau \rightarrow \tau') \rightarrow (\tau \rightarrow \tau')}{\Gamma \vdash \mathtt{fix}(e) : \tau \rightarrow \tau'}$$

Large-step: $$\frac{e \Downarrow \lambda v{:}\tau.e_0 \qquad e_0[\mathtt{fix}(e)/v] \Downarrow u}{\mathtt{fix}(e) \Downarrow u}$$

(Basis for `let rec` in OCaml)

Convention: From now on when we refer to "simply-typed λ-calculus ($\lambda_{\rightarrow}$)", we will assume it includes all of the aforementioned operators but not `fix`. To add `fix`, we will say "simply-typed λ-calculus with fixpoints."

Non-simple types

Extending $\lambda_{\rightarrow}$ to non-simple types:

1. parametric polymorphism (λ_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 ($\lambda_{\underline{\omega}}$)
   - OCaml example: `type 'a tree = Empty | Node of ('a * 'a tree * 'a tree)`
3. dependent types (λ_Π)
   - not available in OCaml or Haskell
   - Recommended language: Gallina (Coq)

In this class we will only study formalisms for System F.

The λ-cube

$$\begin{tikzcd}
& \lambda_{\omega} \arrow[rr] & & \lambda_{C} \\
\lambda_{2} \arrow[ur] \arrow[rr] & & \lambda_{\Pi 2} \arrow[ur] & \\
& \lambda_{\underline{\omega}} \arrow[uu] \arrow[rr] & & \lambda_{\Pi\underline{\omega}} \arrow[uu] \\
\lambda_{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda_{\Pi} \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Intro to System F

$$
\begin{array}{rll}
e ::= \cdots & \mid \Lambda\alpha.e & \text{polymorphic abstraction} \\
 & \mid e[\tau] & \text{polymorphic instantiation} \\
 & & \\
\tau ::= \cdots & \mid \alpha & \text{type variables} \\
 & \mid \forall\alpha.\tau & \text{universal types}
\end{array}
$$

Polymorphic abstractions are functions from *types* to terms:

$$
\frac{}{(\Lambda\alpha.e)[\tau] \rightarrow_1 e[\tau/\alpha]}
$$

Polymorphic Function Examples

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

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

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

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

$$\begin{aligned}
&(\Lambda\alpha.\Lambda\beta.\lambda f{:}\alpha{\rightarrow}\beta.\lambda x{:}\alpha.f\,x)[\mathit{int}][\mathit{bool}]((>)1)(3)\\
&\rightarrow_1 (\Lambda\beta.\lambda f{:}\mathit{int}{\rightarrow}\beta.\lambda x{:}\mathit{int}.f\,x)[\mathit{bool}]((>)1)(3)\\
&\rightarrow_1 (\lambda f{:}\mathit{int}{\rightarrow}\mathit{bool}.\lambda x{:}\mathit{int}.f\,x)((>)1)(3)\\
&\rightarrow_1 (\lambda x{:}\mathit{int}.((>)\,1\,x))(3)\\
&\rightarrow_1 (>)\,1\,3\\
&\rightarrow_1 \texttt{false}
\end{aligned}$$

Static Semantics of System F

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau'}{\Gamma \vdash e[\tau] : \tau'[\tau/\alpha]}$$

Example #1: Polymorphic identity function

$$\begin{aligned}
&(\Lambda\alpha.\lambda x{:}\alpha.x) && : \forall\alpha.(\alpha \to \alpha)\\
&(\Lambda\alpha.\lambda x{:}\alpha.x)[int] && : int \to int\\
&(\Lambda\alpha.\lambda x{:}\alpha.x)[int]3 && : int
\end{aligned}$$

Example #2: Polymorphic application function

$$\begin{aligned}
&(\Lambda\alpha.\Lambda\beta.\lambda f{:}\alpha{\to}\beta.\lambda x{:}\alpha.fx) && : \forall\alpha.\forall\beta.((\alpha \to \beta) \to \alpha \to \beta)\\
&(\Lambda\alpha.\Lambda\beta.\lambda f{:}\alpha{\to}\beta.\lambda x{:}\alpha.fx)[int] && : \forall\beta.((int \to \beta) \to int \to \beta)\\
&(\Lambda\alpha.\Lambda\beta.\lambda f{:}\alpha{\to}\beta.\lambda x{:}\alpha.fx)[int][bool] && : (int \to bool) \to int \to bool\\
&(\Lambda\alpha.\Lambda\beta.\lambda f{:}\alpha{\to}\beta.\lambda x{:}\alpha.fx)[int][bool]((>)1) && : int \to bool\\
&(\Lambda\alpha.\Lambda\beta.\lambda f{:}\alpha{\to}\beta.\lambda x{:}\alpha.fx)[int][bool]((>)1)(3) && : bool
\end{aligned}$$

Type Inhabitation

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

Q: Which System F types are not inhabited?

Type Inhabitation

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

Q: Which System F types are not inhabited?

Are there any besides $void$?

Type Inhabitation

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

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 $\forall\alpha.\alpha$:

$$void = \forall\alpha.\alpha$$

Type Inhabitation

Exercise: Define an algorithm $\mathcal{I} : \tau \to \{T, F\}$ that decides whether any System F type τ is inhabited.

$$\begin{aligned}
\mathcal{I}(int) &= T \\
\mathcal{I}(bool) &= T \\
\mathcal{I}(unit) &= \text{?} \\
\mathcal{I}(\tau_1 \times \tau_2) &= \text{?} \\
\mathcal{I}(\tau_1 + \tau_2) &= \text{?} \\
\mathcal{I}(\tau_1 \to \tau_2) &= \text{?} \\
\mathcal{I}(\forall \alpha . \tau) &= \text{?}
\end{aligned}$$

Type Inhabitation

Exercise: Define an algorithm $\mathcal{I} : \tau \to \{T, F\}$ that decides whether any System F type τ is inhabited.

$$\begin{aligned}
\mathcal{I}(int) &= T \\
\mathcal{I}(bool) &= T \\
\mathcal{I}(unit) &= T \\
\mathcal{I}(\tau_1 \times \tau_2) &= \mathcal{I}(\tau_1) \wedge \mathcal{I}(\tau_2) \\
\mathcal{I}(\tau_1 + \tau_2) &= \mathcal{I}(\tau_1) \vee \mathcal{I}(\tau_2) \\
\mathcal{I}(\tau_1 \to \tau_2) &= \mathcal{I}(\tau_1) \Rightarrow \mathcal{I}(\tau_2) \\
\mathcal{I}(\forall\alpha.\tau) &= \forall\alpha{:}bool, \mathcal{I}(\tau)
\end{aligned}$$

* 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

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 λ-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

Type-inhabitation Problem Walkthrough

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

$$\tau = bool \rightarrow (int \rightarrow void) \rightarrow \forall\alpha.(\alpha \times \alpha)$$

1. Turn τ into a proposition using $\mathcal{I}$.

   $$\mathcal{I}(\tau) = ?$$

2. If $\mathcal{I}(\tau) = F$ then τ is uninhabited, so we're done; otherwise we must construct a term having type τ...

Type-inhabitation Problem Walkthrough

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

$$\tau = bool \rightarrow (int \rightarrow void) \rightarrow \forall\alpha.(\alpha \times \alpha)$$

1. Turn τ into a proposition using $\mathcal{I}$.

$$\begin{aligned}\mathcal{I}(\tau) &= T \Rightarrow (T \Rightarrow F) \Rightarrow \forall\alpha{:}bool.(\alpha \wedge \alpha)\\ &= T \Rightarrow (F \Rightarrow \forall\alpha.(\alpha \wedge \alpha))\\ &= T \Rightarrow T\\ &= T \text{ (so it's inhabited)}\end{aligned}$$

2. If $\mathcal{I}(\tau) = F$ then τ is uninhabited, so we're done; otherwise we must construct a term having type τ...

Type-inhabitation Problem Walkthrough

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

Strategy for finding a System F term having type τ:

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

Type	**Term Constructor**
$unit$	()
int	$\ldots, -1, 0, 1, 2, 3, \ldots$
$bool$	$true, false$
$\times$	(e_1, e_2)
$+$	$\mathtt{in}_1^{\tau_1+\tau_2}(e)$ or $\mathtt{in}_2^{\tau_1+\tau_2}(e)$
$\to$	$\lambda v{:}\tau.e$
$\forall$	$\Lambda\alpha.e$

Using this approach for this τ yields:

Type-inhabitation Problem Walkthrough

$$\tau = bool \rightarrow (int \rightarrow void) \rightarrow \forall\alpha.(\alpha \times \alpha)$$

Strategy for finding a System F term having type τ:

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

Type	Term Constructor
$unit$	()
int	$\ldots, -1, 0, 1, 2, 3, \ldots$
$bool$	$true, false$
$\times$	(e_1, e_2)
$+$	$\mathtt{in}_1^{\tau_1+\tau_2}(e)$ or $\mathtt{in}_2^{\tau_1+\tau_2}(e)$
$\rightarrow$	$\lambda v{:}\tau.e$
$\forall$	$\Lambda\alpha.e$

Using this approach for this τ yields:

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

Why is this not a valid System F term?

Type-inhabitation Problem Walkthrough

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

Strategy for finding a System F term having type τ:

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

Type	**Term Constructor**
$unit$	()
int	$\ldots, -1, 0, 1, 2, 3, \ldots$
$bool$	$true, false$
$\times$	(e_1, e_2)
$+$	$\mathtt{in}_1^{\tau_1+\tau_2}(e)$ or $\mathtt{in}_2^{\tau_1+\tau_2}(e)$
$\to$	$\lambda v{:}\tau.e$
$\forall$	$\Lambda\alpha.e$

Using this approach for this τ yields:

$$\lambda x{:}bool.\lambda y{:}(int \to void).\Lambda\alpha.(\quad,\quad)$$

How to fix?

Type-inhabitation Problem Walkthrough

$$\tau = \mathit{bool} \to (\mathit{int} \to \mathit{void}) \to \forall\alpha.(\alpha \times \alpha)$$

Strategy for finding a System F term having type τ:

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**:

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

Using this approach for this τ yields:

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

Sanity check: Variable instances (y and α 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:

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

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 $F \Rightarrow F$.)

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:

Sample Type-inhabitation Problem

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

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

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

$$\mathcal{I}(\tau) = \forall\alpha.\forall\beta.((\alpha \vee \beta) \Rightarrow (\beta \vee \alpha))$$

Sample Type-inhabitation Problem

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

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

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

$$\mathcal{I}(\tau) = \forall\alpha.\forall\beta.((\alpha \vee \beta) \Rightarrow (\beta \vee \alpha))$$

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

Sample Type-inhabitation Problem

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

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

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

$$\mathcal{I}(\tau) = \forall\alpha.\forall\beta.((\alpha \vee \beta) \Rightarrow (\beta \vee \alpha))$$

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

$\Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha + \beta.?$

Sample Type-inhabitation Problem

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

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

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

$$\mathcal{I}(\tau) = \forall\alpha.\forall\beta.((\alpha \vee \beta) \Rightarrow (\beta \vee \alpha))$$

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

$$\Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha + \beta.\mathtt{case}\ x\ \mathtt{of}\ \mathtt{in}_1(y) \to \text{?} \qquad \mid \mathtt{in}_2(z) \to \text{?}$$

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:

$$\mathcal{I}(\tau) = \forall\alpha.\forall\beta.((\alpha \vee \beta) \Rightarrow (\beta \vee \alpha))$$

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

$$\Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha + \beta.\texttt{case}\ x\ \texttt{of}\ \texttt{in}_1(y) \rightarrow \texttt{in}_2^{\beta+\alpha} y \mid \texttt{in}_2(z) \rightarrow \texttt{in}_1^{\beta+\alpha} z$$

Tautologicality and Operation Order

Exercise: Are the following types inhabited? If so, write terms having these types.

$$\begin{aligned} \tau_1 &= \forall\alpha.(\alpha \to void) \\ \mathcal{I}(\tau_1) &= \forall\alpha.(\alpha \Rightarrow F) \\ &= \text{?} \end{aligned}$$

$$\begin{aligned} \tau_2 &= (\forall\alpha.\alpha) \to void \\ \mathcal{I}(\tau_2) &= (\forall\alpha.\alpha) \Rightarrow F \\ &= \text{?} \end{aligned}$$

Tautologicality and Operation Order

Exercise: Are the following types inhabited? If so, write terms having these types.

$$\begin{aligned} \tau_1 &= \forall\alpha.(\alpha \to \mathit{void}) \\ \mathcal{I}(\tau_1) &= \forall\alpha.(\alpha \Rightarrow F) \\ &= F \text{ (because } T \not\Rightarrow F\text{)} \end{aligned}$$

$$\begin{aligned} \tau_2 &= (\forall\alpha.\alpha) \to \mathit{void} \\ \mathcal{I}(\tau_2) &= (\forall\alpha.\alpha) \Rightarrow F \\ &= F \Rightarrow F \\ &= T \end{aligned}$$

Tautologicality and Operation Order

Exercise: Are the following types inhabited? If so, write terms having these types.

$$\begin{aligned} \tau_1 &= \forall\alpha.(\alpha \to \mathit{void}) \\ \mathcal{I}(\tau_1) &= \forall\alpha.(\alpha \Rightarrow F) \\ &= F \text{ (because } T \not\Rightarrow F\text{)} \end{aligned}$$

$$\begin{aligned} \tau_2 &= (\forall\alpha.\alpha) \to \mathit{void} \\ \mathcal{I}(\tau_2) &= (\forall\alpha.\alpha) \Rightarrow F \\ &= F \Rightarrow F \\ &= T \end{aligned}$$

$$(\lambda x{:}\mathit{void}.x) : (\forall\alpha.\alpha) \to \mathit{void}$$

Brokenness of fix

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

Recall the typing rule for `fix`:

$$\frac{\Gamma \vdash e : (\tau \to \tau') \to (\tau \to \tau')}{\Gamma \vdash \mathtt{fix}(e) : \tau \to \tau'}$$

With it we can derive:

$$\dfrac{\dfrac{\dfrac{\{(x, \mathit{unit} \to \mathit{void})\} \vdash x : \mathit{unit} \to \mathit{void}}{\bot \vdash \lambda x{:}\mathit{unit}{\to}\mathit{void}.x : (\mathit{unit} \to \mathit{void}) \to (\mathit{unit} \to \mathit{void})}}{\bot \vdash \mathtt{fix}(\lambda x{:}\mathit{unit}{\to}\mathit{void}.x) : \mathit{unit} \to \mathit{void}} \qquad \bot \vdash \mathtt{()} : \mathit{unit}}{\bot \vdash \mathtt{fix}(\lambda x{:}\mathit{unit}{\to}\mathit{void}.x)\mathtt{()} : \mathit{void}}$$

C-H Isomorphism and Derivation Rule Soundness

Two ways to understand the problem:

- $e : \tau$ is like saying "e promises to return a τ." But e breaks its promise if e is an infinite loop.
- $e : \tau$ is like saying e is a **proof** of proposition τ. But the typing rule for `fix` is unsound, so not a valid proof:

$$\mathcal{I}\left(\frac{\Gamma \vdash e : (\tau \to \tau') \to (\tau \to \tau')}{\Gamma \vdash \mathtt{fix}(e) : \tau \to \tau'}\right) = \frac{(\tau \Rightarrow \tau') \Rightarrow (\tau \Rightarrow \tau')}{\tau \Rightarrow \tau'}$$

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 τ (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 τ such that $\perp \vdash e : \tau$ is derivable.

Good news and bad news:

- Type-checking is decidable for both $\lambda_{\rightarrow}$ and System F.
- Type-inference is decidable for $\lambda_{\rightarrow}$.
- Type-inference is undecidable for System F. ☹

Shallow Types

Definition (shallow type): A type τ is *shallow* if no quantifiers are children of non-quantifiers in τ's AST.

Examples:

- $int \rightarrow unit$ 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;;
```
$\texttt{apply} = \Lambda\alpha.\Lambda\beta.\lambda f{:}\alpha{\rightarrow}\beta.\lambda x{:}\alpha.(f x)$

```
let y = apply ((>)1) 5;;
```
$\texttt{y} = \texttt{apply}[int][bool]((>)1)5$

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:

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

2. Infer a mapping $\theta : \alpha \rightharpoonup \tau$ 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 α 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:

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

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

Notations:

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

Hindley-Milner Type-inference

$$\frac{}{\theta, \Gamma \vdash \texttt{()} : \mathit{unit}, \theta} \tag{1}$$

$$\frac{\Gamma(v) = \forall\beta_1 \ldots \forall\beta_n.\tau}{\theta, \Gamma \vdash v[\alpha_1] \cdots [\alpha_n] : \tau[\alpha_1/\beta_1] \cdots [\alpha_n/\beta_n], \theta} \tag{2}$$

$$\frac{\theta, \Gamma[v \mapsto \alpha] \vdash e : \tau, \theta'}{\theta, \Gamma \vdash \lambda v{:}\alpha.e : \alpha \to \tau, \theta'} \tag{3}$$

$$\frac{\theta, \Gamma \vdash e_1 : \tau_1, \theta_1 \quad \theta_1, \Gamma[\theta_1] \vdash e_2 : \tau_2, \theta_2 \quad \theta_3 = \mathcal{U}(\tau_1[\theta_2], \tau_2 \to \alpha) \quad \theta' = \theta_2 \sqcup \theta_3}{\theta, \Gamma \vdash e_1 e_2 : \theta'(\alpha), \theta'} \tag{4}$$

Type-inference for Function Application

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

1. Infer a type τ_1 for e_1.
2. Infer a type τ_2 for e_2.
3. Types τ_1 and $\tau_2 \to \alpha$ must be identical (for some α). **Unify** them:

Definition (type unification): The *unification* of types τ_1 and τ_2 is an instantiation $\theta : \alpha \rightharpoonup \tau$ of their type variables that causes them to be identical:

$$\begin{aligned}
\mathcal{U}(\alpha, \alpha) &= \bot \\
\mathcal{U}(unit, unit) &= \bot \\
\mathcal{U}(\alpha, \tau) = \mathcal{U}(\tau, \alpha) &= \{(\alpha, \tau)\} \text{ if } \alpha \text{ is not free in } \tau \\
\mathcal{U}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') &= \mathcal{U}(\tau_1, \tau_1') \sqcup \mathcal{U}(\tau_2, \tau_2')
\end{aligned}$$

$\mathcal{U}$ is undefined otherwise (type-inference rejects)

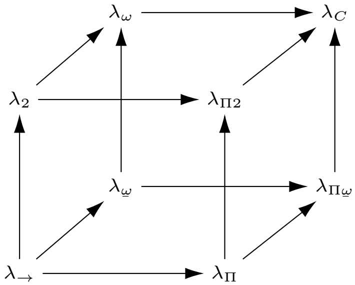
Non-shallow Types

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)
- λ_2 (System F): parametric polymorphism
- $\lambda_{\underline{\omega}}$: parametrically polymorphic datatypes
  - OCaml is essentially $(\lambda_\omega \cap \text{shallow types}) \cup \mathit{fix}$
  - Haskell is essentially $\lambda_\omega \cup \mathit{fix}$
- λ_Π: dependent types (correspond to $\exists$ in propositional logic)
- λ_C: Calculus of Constructions (combines all)
  - Coq/Gallina is essentially λ_C