Simply Typed λ-Calculus + Polymorphism
λ-Calculus Cheatsheet

Syntax

\[ E ::= v \mid \lambda v. E \mid E \; E \]

Reduction Rules

\( \alpha \)-rule: \( \lambda x. e \rightarrow \lambda y. e[y/x] \) if \( y \notin \text{FV}(E) \)

\( \beta \)-rule: \( (\lambda x. e_1) \; e_2 \rightarrow e_1[e_2/x] \)

\( \eta \)-rule: \( (\lambda x. e \; x) \rightarrow e \) if \( x \notin \text{FV}(E) \)

Redex

\( (\lambda x. E) \; E \)

Normal Form

An expression without redexes
(Untyped) $\lambda$-Calculus Semantics

- What is the normal form of

$$(\lambda f \, x. \, f \, (f \, x)) \, (\lambda a \, b \, c. \, a \, b \, c) \, (\lambda x \, y. \, y)$$

2 → $$(\lambda a \, b \, c. \, a \, b \, c) \, ((\lambda a \, b \, c. \, a \, b \, c) \, (\lambda x \, y. \, y))$$

ite → $$(\lambda a \, b \, c. \, a \, b \, c) \, (\lambda x \, y. \, y) \, b \, c$$

False → $$(\lambda b \, c. \, (\lambda a \, b \, c. \, a \, b \, c) \, (\lambda x \, y. \, y) \, b \, c) \, b \, c$$

0
(Untyped) $\lambda$-Calculus Semantics

- **Problem #1**
  - $\lambda$-calculus assigns a semantics for every term (even when the operation does not match the operands).
    
    \[ \text{2 ite False} \rightarrow^* 0 \]

- **Problem #2**
  - $\lambda$-calculus semantics is inconsistent.
    
    \[ u \triangleq \lambda x. \text{not } (x \ x) \]
    \[ (u \ u) \rightarrow^* \text{not } (u \ u) \]

    \[ \Rightarrow \quad u \ u \equiv \text{TRUE} \iff u \ u \equiv \text{FALSE} \]

  - (as usual, paradox is caused by self-application.)
Simply Typed $\lambda$-Calculus

- **Ingredients:**
  - $B$ — set of **base types** (e.g., \{int, nat, real, bool\})
  - $\tau = B \cup \{\tau \rightarrow \tau\}$ — closure under `\rightarrow` (function type ctor)
  - $C$ — set of (typed) **term constants** (e.g., \{1, TRUE\})
  - Extended syntax for expressions:
    $$ E ::= v \mid \lambda v: \tau. E \mid E E \mid c $$
    - variable
    - **typed** abstraction
    - application
    - constant

- **Typing rules**
Typing Rules

• $e : T$ — expression $e$ has type $T$

**Constant**

\[
\begin{align*}
0 : \text{nat} & \quad 1 : \text{nat} & \quad 2 : \text{nat} & \quad \ldots & \quad \text{TRUE} : \text{bool}
\end{align*}
\]

**Application**

\[
\begin{align*}
e_1 : A \rightarrow B & \quad e_2 : A \\
\hline
\quad e_1 e_2 : B
\end{align*}
\]

**Abstraction**

\[
\begin{align*}
\lambda v : A. e & : A \rightarrow B
\end{align*}
\]
Typing Rules

- $\Gamma \vdash e : T$ — $e$ has type $T$ in environment $\Gamma$

**constant**

- $\Gamma \vdash 0 : \text{nat}$
- $\Gamma \vdash 1 : \text{nat}$
- $\Gamma \vdash 2 : \text{nat}$
- $\Gamma \vdash \text{TRUE} : \text{bool}$

**application**

- $\Gamma \vdash e_1 : A \rightarrow B$
- $\Gamma \vdash e_2 : A$

- $\Gamma \vdash e_1 e_2 : B$

**abstraction**

- $\Gamma, v : \text{A} \vdash e : \text{B}$

- $\Gamma \vdash \lambda v : A. e : A \rightarrow B$
Typing Rules

- \( \Gamma \vdash e : T \quad \text{— \ e has type \( T \) in \textit{environment} \( \Gamma \) \)

\[
\begin{align*}
\text{constant} & \quad \text{variable} \\
\Gamma \vdash 0 : \text{int} & \quad \Gamma \vdash 1 : \text{int} \quad \ldots \\
\text{application} & \quad \text{abstraction} \\
\Gamma \vdash e_1 : A \to B & \quad \Gamma \vdash e_2 : A \\
\Gamma \vdash e_1 \ e_2 : B \\
\Gamma, v : A \vdash e : B \\
\Gamma \vdash \lambda v : A. e : A \to B
\end{align*}
\]

\( \text{Vars} \rightarrow \tau \)
From Checking to Inference

• **Type Checking**
  Given an expression e and a type environment for the free variables of e, check if e is well-typed and return its type.

• **Type Inference**
  Given an expression e *with partial or no type annotations*, compute the types of all variables in e, as well as the type of e.
int f(int x) {
    return x+1;
};
int g(int y) {
    return f(y+1)*2;
};
Type Inference: Basic Idea

\[ f \triangleq \lambda x. x + 2 \]

• What is the type of \( f \)?
  
  ‣ \texttt{plus (+)} has type \( \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \)
  
  ‣ \texttt{2} has type \( \text{nat} \)
  
  ‣ Since \texttt{plus} is applied to \( x \) we need \( x : \text{nat} \)

\[ f \triangleq \lambda x : \text{nat}. \ x + 2 \quad \text{has type} \quad \text{nat} \rightarrow \text{nat} \]
Type Inference: Basic Idea

\[ f \triangleq \lambda x. x + 2 \]

- **Plan**
  - **Step 1.** Assign type variables to sub-terms
  - **Step 2.** Generate type constraints
  - **Step 3.** Solve constraints
Step 1. Assign type variables

\[ f \equiv \lambda x. x + 2 \]
Step 2. Generate type constraints

\[ f \triangleq \lambda x. x + 2 \]

\[
\begin{align*}
\Gamma \vdash e_1 : A \
\Gamma \vdash e_2 : A \\
\Gamma \vdash e_1 \ e_2 : B
\end{align*}
\]

\[
\begin{align*}
\Gamma, v : A & \vdash e : B \\
\Gamma & \vdash \lambda v : A. e : A \to B
\end{align*}
\]

\[
\begin{align*}
T_0 = T_1 \to T_2 \\
T_3 = T_4 \to T_2 \\
T_5 = T_1 \to T_3 \\
T_4 = \text{nat} \\
T_5 = \text{nat} \to \text{nat} \to \text{nat}
\end{align*}
\]
Step 3. Solve the constraints

\[ f \triangleq \lambda x. x + 2 \]
Step 3. Solve the constraints

\[ f \equiv \lambda x. x + 2 \]

\( T_0 = T_1 \rightarrow T_2 \)

\( T_3 = T_4 \rightarrow T_2 \)

\( T_5 = T_1 \rightarrow T_3 \)

\( T_4 = \text{nat} \)

\( T_5 = \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \)
Step 3. Solve the constraints

\[
f \triangleq \lambda x. x + 2
\]
Step 3. Solve the constraints

\[ f \triangleq \lambda x. x + 2 \]

\[ T_0 = T_1 \rightarrow T_2 \]
\[ T_3 = T_4 \rightarrow T_2 \]
\[ T_5 = T_1 \rightarrow T_4 \rightarrow T_2 \]
\[ T_4 = \text{nat} \]
\[ T_1 \rightarrow T_4 \rightarrow T_2 = \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \]
Step 3. Solve the constraints

\[ f \triangleq \lambda x. x + 2 \]
Step 3. Solve the constraints

\[ f \triangleq \lambda x. x + 2 \]

\[ T_0 = \text{nat} \rightarrow \text{nat} \]

\[ T_3 = \text{nat} \rightarrow \text{nat} \]

\[ T_5 = \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \]

\[ T_4 = \text{nat} \]

\[ T_1 \rightarrow \text{nat} \rightarrow T_2 = \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \]

\[ T_1 = \text{nat} \]

\[ T_2 = \text{nat} \]
What was that **Unify**?

- **Unify** $t_1$ and $t_2$ \(\triangleq\)
  
  - If $t_1 = T_i \Rightarrow$ admit $T_i = t_2$
  - If $t_2 = T_j \Rightarrow$ admit $T_j = t_1$
  - If $t_1 = s_1' \rightarrow t_1',\ t_2 = s_2' \rightarrow t_2' \Rightarrow$
    - Unify $s_1'$ and $s_2'$
    - Unify $t_1'$ and $t_2'$
  - If none of the above $\Rightarrow$ unification fails.
Pros and Cons

• Good news
  ▸ Weird expressions like \texttt{2 ite False} are not well-typed.
  ▸ Self application \texttt{u u} is not well-typed — paradox avoided.
Pros and Cons

• Bad news
  ‣ Last lecture’s definitions of \texttt{plus}, \texttt{ite}, \texttt{etc.} are not well-typed either.
   
   $$\Rightarrow$$ These will all have to become \texttt{constants}.
   Each of them will need a dedicated \texttt{derivation rule}.

  ‣ The recursion combinator (\texttt{Y}) is not well-typed either!

$$\Rightarrow$$ We will have to add a new \texttt{syntactic construct} for recursion, with more dedicated \texttt{derivation rules}.
Polymorphism

$P_1$ $\rightarrow$ $P_2$
Polymorphism
Polymorphism

- What is the type of $\lambda f \ x. \ f \ (f \ x)$?
  - Answer: it depends!

$\lambda (f: A \rightarrow A) \ (x: A). \ f \ (f \ x) : (A \rightarrow A) \rightarrow A \rightarrow A$

$A$ is a type variable
Polymorphism

- Extended syntax —

\[ \tau ::= B \mid \alpha \mid \tau \to \tau \mid \forall \alpha. \tau \]

\[ E ::= \cdots \mid \Lambda \alpha. E \mid E[\tau] \]

\[ \Lambda A. \lambda (f:A \to A) (x:A). f (f x) : \forall A. (A \to A) \to A \to A \]

\( \alpha \) is a type variable *meta-variable!*

- type
  - “abstraction” (generalization)
  - “application” (instantiation)
Polymorphism

• Extended syntax —

\[ \tau ::= B \mid \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau \]

\[ E ::= \cdots \mid \Lambda \alpha. E \mid E[\tau] \]

• Extended semantics —

**instantiation**

\[ \Gamma \vdash e : \forall \alpha. A \]

\[ \Gamma \vdash e[B] : A[B/\alpha] \]

**generalization**

\[ \Gamma \vdash e : A \]

\[ \Gamma \vdash \Lambda \alpha.e : \forall \alpha. A \]

\(\alpha\) is a type variable *meta-variable!*

Simple types + Polymorphism = **System-F**
Everything has a price

\[
\begin{align*}
\text{let } 2 &= \lambda f \text{ x. } f (f \text{ x}) \text{ in} \\
\text{let } \text{succ} &= \lambda n \text{ f x. } \\
& \quad \quad f (n \text{ f x}) \text{ in} \\
2 \text{ succ}
\end{align*}
\]

Untyped

\[
\begin{align*}
\text{let } 2 &= \Lambda A. \lambda (f:A \rightarrow A) \text{ (x:A). } f (f \text{ x}) \text{ in} \\
\text{let } \text{succ} &= \lambda (n: \forall A. (A \rightarrow A) \rightarrow A \rightarrow A). \\
& \quad \quad \Lambda B. \lambda (f:B \rightarrow B) \text{ (x:B). } f (n[B] f \text{ x}) \text{ in} \\
2[\forall A. (A \rightarrow A) \rightarrow A \rightarrow A] \text{ succ}
\end{align*}
\]

Simply typed + polymorphism

- Type inference is gravely needed
  - Unfortunately, it cannot eliminate 100% of type annotations

- Although Church encoding can be typed, it is rarely used this way
  - Spoiler: inductive types are used instead.
Everything has a price

- Some complexity results
  - Hindley-Milner: arguments types are monomorphic
    - type inference is **EXPTIME-complete**
      - (using +/- the same algorithm)
  - Full System-F \(\Rightarrow\) type inference is **undecidable**
    - (requires *high-order unification*)
Everything has a price

\[ Y \triangleq \lambda f. ((\lambda x. (f (x x))) (\lambda x. (f (x x)))) \]

- Cannot be typed — even with polymorphism
  - No recursion — no game!
  - Special syntactic form: \texttt{fix f}

\[ \Gamma \vdash \text{fix } e : A \]

\[ \frac{\Gamma \vdash e : A \to A}{\Gamma \vdash \text{fix } e : A} \]

(recall from untyped: \[ Y \ H \rightarrow^* H \ (Y \ H) \])
Limitations

• What still makes our lives difficult:
  ‣ Subtyping (a.k.a. inheritance)
  ‣ Ad-hoc polymorphism (a.k.a. overloading)
  ‣ Dynamic dispatch (which has a flavor of both)

(Disclaimer: personal opinion)

This is the main cause for the gap between functional and object-oriented programming languages.
Exercise #1

• Implement a type checker with inference for simply typed λ-calculus.
  ▸ Better to implement ‘unify’ first.
  ▸ Simplifying assumption: base types are lowercase
    ◦ Feel free to name your type variables T1, T2, etc.

- **Input**: an expression with *some* type annotations
- **Output**:
  ▸ The fully annotated expression
  ▸ The type of the expression