Simply Typed \(\lambda\)-Calculus + Polymorphism

\(\lambda\)-Calculus Cheatsheet

**Syntax**
\[
E ::= v \mid \lambda v.E \mid E E
\]

**Reduction Rules**
- **\(\alpha\)-rule**: \(\lambda x.e \rightarrow \lambda y.(y/x)\) if \(y \notin \text{PV}(E)\)
- **\(\beta\)-rule**: \((\lambda x.e_1) e_2 \rightarrow e_1[e_2/x]\)
- **\(\eta\)-rule**: \((\lambda x.e\ x) \rightarrow e\) if \(x \notin \text{PV}(E)\)

**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 ite False

(Untyped) \(\lambda\)-Calculus Semantics

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

- **Problem #2**
  - \(\lambda\)-calculus semantics is inconsistent.
  
  \[
  \underbrace{u} = \lambda x. \text{not}\ (x\ x) \quad \Rightarrow \quad \underbrace{u\ u} = \text{TRUE} \quad \Leftrightarrow \quad \underbrace{u\ u} = \text{FALSE}
  \]
  
  \[
  (u\ u) \rightarrow^* \text{not}\ (u\ u)
  \]
  
  (as usual, paradox is caused by self-application.)

Types, intuitively

Type of the Untyped

- What is the type of \(\lambda f\ x. f\ (x\ f)\)?

Function types use the notation:

\[
\text{(argument type)} \rightarrow \text{(result type)}
\]

\[\text{all untyped } \lambda\text{-calculus terms have the "type".}\]
Simply Typed λ-Calculus

- **Ingredients:**
  - **B** — set of base types (e.g., int, nat, real, bool)
  - **τ = B ∪ {τ → τ}** — closure under ‘→’ (function type constructor)
  - **C** — set of (typed) term constants (e.g., (1, True))

- **Extended syntax for expressions:**
  - \[E ::= v \mid \lambda v : \tau . E \mid E \cdot E \mid c\]
    - variable
    - typed abstraction
    - application
    - constant

- **Typing rules**

1. **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.

2. **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 \( e \).
From Checking to Inference

Type Checking

int f(int x) {
    return x+1;
}

int g(int y) {
    return f(y+1)*2;
}

Type Inference

\[
\begin{align*}
\text{Let } f & \equiv \lambda x. x + 2 \\
\text{Then } f & \equiv \lambda x. x + 2
\end{align*}
\]

Type Inference: Basic Idea

- What is the type of \( f \)?
  - \( \text{plus (+)} \) has type \( \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \)
  - \( 2 \) has type \( \text{nat} \)
  - Since \( \text{plus} \) is applied to \( x \) we need \( x : \text{nat} \)
  - \( \Rightarrow f \equiv \lambda x : \text{nat}. x + 2 \) has type \( \text{nat} \rightarrow \text{nat} \)

Type Inference: Basic Idea

- Plan
  - \textbf{Step 1.} Assign type variables to sub-terms
  - \textbf{Step 2.} Generate type constraints
  - \textbf{Step 3.} Solve constraints

Step 1. Assign type variables

\[
\begin{align*}
T_0 & = T_1 \rightarrow T_3 \\
T_3 & = T_2 \rightarrow T_1 \\
T_5 & = T_4 \rightarrow T_1 \\
T_1 & = \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}
\end{align*}
\]

Step 2. Generate type constraints

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

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

\[ T_0 = T_1 \rightarrow T_2 \]
\[ T_3 = T_1 \rightarrow T_2 \]
\[ T_4 = \text{nat} \]
\[ T_5 = \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \]

Unify

\[ T_0 = T_1 \rightarrow T_2 \]
\[ T_3 = T_4 \rightarrow T_2 \]
\[ T_5 = T_1 \rightarrow T_4 \rightarrow T_2 \]
\[ T_4 = \text{nat} \]
\[ T_5 = \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \]

Unify

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

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

What was that **Unify**?

- Unify \( t_1 \) and \( t_2 \):  
  - for type expressions, \( t_1 \neq t_2 \):  
    - If \( t_1 = T_i \) ⇒ admit \( T_i = t_1 \)  
    - If \( t_2 = T_j \) ⇒ admit \( T_j = t_2 \)  
    - If \( t_1 = s_1 \rightarrow t_1', t_2 = s_2 \rightarrow t_2' \) ⇒  
      - Unify \( s_1 \) and \( s_2 \)  
      - Unify \( t_1' \) and \( t_2' \)  
    - If none of the above ⇒ unification fails.

**Pros and Cons**

- Good news  
  - Weird expressions like \( 2 \text{ite} \text{False} \) are not well-typed.  
  - Self application \( \text{u u} \) is not well-typed — paradox avoided.

**Pros and Cons**

- Bad news  
  - Last lecture's definitions of \( \text{plus, ite, etc.} \) are not well-typed either.

⇒ These will all have to become constants.  
Each of them will need a dedicated derivation rule.

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

⇒ We will have to add a new syntactic construct for recursion, with more dedicated derivation rules.

**Polymorphism**
Polymorphism

What is the type of \( \lambda f \, x. f \, (f \, x) \)?

Answer: it depends!

\[
\lambda (f : A \rightarrow A) \, (x : A), \; f \, (f \, x) : \forall A. (A \rightarrow A) \rightarrow A \rightarrow A
\]

\( \alpha \) is a type variable

Extended syntax —

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

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

Type
“abstraction”
(generalization)

Type
“application”
(instantiation)

\[
\Lambda A. \lambda (f : A \rightarrow A) \, (x : A), \; f \, (f \, x) : \forall A. (A \rightarrow A) \rightarrow A \rightarrow A
\]

Extended semantics —

\[
\Gamma \vdash e : A
\]

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

\[
\Gamma \vdash e : A
\]

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

Everything has a price

- 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.

```
let 2 = \f x. f (f x) in
let succ = \n f x. f (f x) in
2 succ

let 2 = \A. \alpha (f : A \rightarrow A) \, (x : A), \; f \, (f \, x) in
let succ = \n : \forall A. (A \rightarrow A) \rightarrow A \rightarrow A.
  \alpha (f : A \rightarrow A) \, (x : A), \; f \, (f \, x) in
2 succ
```

Untyped

- Simply typed + polymorphism
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 ⇒ type inference is undecidable
    - (requires high-order unification)

Everything has a price

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

\[ \text{Cannot be typed — even with polymorphism} \]
- No recursion — no game!
⇒ Special syntactic form: fix

\[ \text{fix e} \rightarrow e (\text{fix e}) \]

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)

\[ \text{Disclaimer: personal opinion} \]
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 \( \lambda \)-calculus.
  - Better to implement ‘unify’ first.
  - Simplifying assumption: base types are lowercase
    - Feel free to name your type variables T1, T2, etc.

\[ \text{Input}: \text{an expression with some type annotations} \]
\[ \text{Output}: \]
  - The fully annotated expression
  - The type of the expression