Type Systems
\[\vdash\]
Simply-typed \(\lambda\)-Calculus
Previously, grammar of $\lambda$-calculus

$$E ::= x \mid \lambda x. E \mid E E$$

(var)  (abstraction)  (application)

- In the typed $\lambda$-calculus, we will indicate the type of each argument of a function:

$$\lambda x : \tau . E \quad \text{where} \quad x \text{ is of type } \tau$$
Types

- what is a type?

- a set of related values

- e.g., "int" type describes integer literals \( ...,-2,-1,0,1,2,... \)
  and all expressions that evaluate to integers

- we use the notation \( t_1 \rightarrow t_2 \) to describe a function
  from \( t_1 \) to \( t_2 \)

- e.g., \( \text{int} \rightarrow \text{int} \) describes all function from
  integers to integers
Adding some types

- We will extend the language to support two new types
  - `int`, with values $n \in \mathbb{Z}$
  - `unit`, with the value `()`

```
:\text{in} \lambda x : \tau. E,
\tau :: = \text{int} \mid \text{unit} \mid \tau \rightarrow \tau
```
Grammar of simply-typed $\lambda$-calculus

**types**

$\tau ::= \text{int} \mid \text{unit} \mid \tau \rightarrow \tau$

**values**

$v ::= n \mid () \mid \lambda x: \tau . E$

**expressions**

$E ::= x \mid \lambda x: \tau . E \mid E E \mid n \mid E + E \mid ()$

ADD

\[
\frac{E_1 \rightarrow n_1, E_2 \rightarrow n_2}{n_1 + n_2 \rightarrow n} \quad \text{arith}
\]
- Types do not change the operational semantics (evaluation) of programs!
- But we can use types to help reason about our programs and to ensure the absence of type-related bugs (e.g., 8 + ()
- when used in this way, we refer to our language additions as a type system
- another form of (embedded) semantics!
Well-Typed Programs

guarantee: any well-typed program will not get stuck

- an expression $E$ is stuck if $E$ is not a value and there is no $E'$ such that $E \rightarrow E'$

  e.g., $\lambda x.x + 10$ is stuck!
  
  () 8 is stuck!
The Typing Relation: $\Gamma \vdash E : T$

- $\Gamma \vdash E : T$ asserts that under typing context $\Gamma$, $E$ has type $T$.

- A typing context is a set of variables and their types. e.g., $\emptyset$, $\{x : \text{int}, y : \text{unit} \to \text{unit}\}$

- If $\exists T$ where $\Gamma \vdash E : T$, we say $E$ is well-typed under $\Gamma$.

- If $\Gamma = \emptyset$, we say $E$ is well-typed.
Type Safety

Formally: `if \( T \vdash E : T \) and \( E \rightarrow^* E' \) then 'either \( E' \) is a value or \( \exists E'' \) s.t. \( E' \rightarrow E'' \)`

Two parts:

"Preservation": `if \( T \vdash E : T \) and \( E \rightarrow^* E' \) then \( T \vdash E' : T \)`

"Progress": `if \( T \vdash E : T \) then 'either \( E \) is a value or \( \exists E' \) s.t. \( E \rightarrow E' \)`
Simply-typed λ-calculus type rules

\[
\begin{align*}
\text{INT} & \quad \Gamma \vdash n : \text{int} \\
\text{UNIT} & \quad \Gamma \vdash () : \text{unit} \\
\text{VAR} & \quad \Gamma \vdash x : \tau \quad \{x : \tau\} \subseteq \Gamma
\end{align*}
\]
Simply-typed $\lambda$-calculus type rules

\[
\text{ADD} \quad \frac{\Gamma + E_1 : \text{int}, \Gamma + E_2 : \text{int}}{\Gamma + E_1 + E_2 : \text{int}}
\]

\[
\text{ABS} \quad \frac{\Gamma \vdash x : \tau \vdash E : \tau'}{\Gamma \vdash \lambda x : \tau. E : \tau \rightarrow \tau'}
\]

\[
\text{APP} \quad \frac{\Gamma + E_2 : \tau' \quad \Gamma + E_1 : \tau' \rightarrow \tau}{\Gamma + E_1 E_2 : \tau}
\]
eg. prove + (\(\lambda x: \text{int}. \, 2 + x\)) 10 : \text{int}
Type Safety in the Simply-typed λ-Calculus

guarantee: a program that gets stuck is not well-typed

—are all programs that do not get stuck well-typed?

NO!

e.g., \((\lambda x:\text{unit}.x)\,4 \rightarrow^\beta 4\)

\((\text{idemity})\)

e.g., \((\lambda x.x\,x\,x\,x\,x)\)(\lambda x.x\,x\,x\,x\,x)

what are their types?  can't be \(t \rightarrow t'\)

because it's applied to itself, which would make the first

\((t \rightarrow t') \rightarrow \ldots\)
Type Safety in the Simply-typed $\lambda$-Calculus

guarantee: a program that gets stuck is not well-typed

- what we can say is that all well-typed programs terminate

i.e., if $E : \tau$, we can reduce $E$ to normal form (i.e., no redexes)

- to properly describe the types of $\lambda x.x$ and recursive functions, we need a more sophisticated type system!
Type Inference

- what if function arguments don't type annotations?

  e.g. \( \lambda x. \lambda y. \lambda z. \text{if } (y \ (4+x)) \text{ then } x \text{ else } z \)

- can we still perform type checking?

  \( \text{infer type : } \lambda x : \text{int}. \ \lambda y : \text{int} \rightarrow \text{ bool}. \ \lambda z : \text{int} \rightarrow \).

- how can we formally define this process?
Type constraints

given type context $\Gamma$ and expression $E$, we would like to
formally specify a set of constraints that must be satisfied
so that $E$ is well-typed under $\Gamma$

- constraints will take the form of a set of equations
  between types and type variables
e.g. $C = \{ X = \text{int}, Y = \text{int} \to \text{int} \}$

- satisfying the constraints means finding a mapping from
type variables to types s.t. all the equations are in agreement.
Deriving type variables + constraints

- update $T ::= \text{int} | \text{unit} | T \to T | X$

- new typing relation $\Gamma \vdash E : T \triangleright C$ — $E$ has type $T$ if $C$ satisfied

- update rules to help derive values of $T$ and $C$

\[
\text{eq}\cdot\text{C-ADD} \quad \frac{\Gamma \vdash E_1 : T_1 \triangleright C_1, \Gamma \vdash E_2 : T_2 \triangleright C_2}{\Gamma \vdash E_1 + E_2 : \text{int} \triangleright C_1 \cup C_2 \cup \{T_1 = \text{int}, T_2 = \text{int}\}}
\]
Deriving type variables + constraints

\[ \begin{align*}
\text{C-ABS} & \quad \frac{\Gamma \left\{ x : \tau_i \right\} \vdash E : \tau_2 \Delta C}{\Gamma \vdash \lambda x : \tau_1 . E : \tau_1 \rightarrow \tau_2 \Delta C} \\
\text{C-APP} & \quad \frac{\Gamma \vdash E_1 : \tau_1 \Delta C_1 , \Gamma \vdash E_2 : \tau_2 \Delta C_2 , C = C_1 U C_2 U \{ \tau_i = \tau_2 \rightarrow X \}}{\Gamma \vdash E_1 E_2 : X \Delta C}
\end{align*} \]
e.g., derive type constraints for the expression:

\[ \lambda a : X. \lambda b : Y. 4 + (b \ (2 + a)) \]

\[
\Gamma \vdash 2 : \text{int} \triangleright \emptyset \quad \Gamma \vdash a : X \triangleright \emptyset
\]

\[
\Gamma \vdash b : Y \triangleright \emptyset \quad \Gamma \vdash 2 + a : \text{int} \triangleright \{ \text{int} = \text{int}, X = \text{int} \}
\]

\[
\Gamma \vdash 4 : \text{int} \triangleright \emptyset \quad \Gamma \vdash b \ (2 + a) : \text{int} \triangleright \{ \text{int} = \text{int}, X = \text{int}, Y = \text{int} \rightarrow \text{int} \rightarrow \text{int} \}
\]

\[
\Gamma \{ a : X, b : Y \} \vdash 4 + (b \ (2 + a)) : \text{int} \triangleright \{ \text{int} = \text{int}, X = \text{int}, Y = \text{int} \rightarrow \text{int} \rightarrow \text{int} \}
\]

\[
\Gamma \{ a : X, b : Y \} \vdash \lambda b : Y. 4 + (b \ (2 + a)) : Y \rightarrow \text{int} \triangleright \{ \text{int} = \text{int}, X = \text{int}, Y = \text{int} \rightarrow \text{int} \rightarrow \text{int} \}
\]

\[\vdash \lambda a : X. \lambda b : Y. 4 + (b \ (2 + a)) : X \rightarrow Y \rightarrow \text{int} \triangleright \{ \text{int} = \text{int}, X = \text{int}, Y = \text{int} \rightarrow \text{int} \rightarrow \text{int} \}\]
e.g., derive type constraints for the expression:

\[ \lambda a : X. \lambda b : Y. 4 + (b (z + a)) \]

\[ C = \{ \text{int = int}, \ X = \text{int}, \ Y = \text{int} \to \text{int}, \ Z = \text{int} \} \]

- to "solve" this set of constraints, we will find a mapping from type variables to types that simultaneously satisfies all its equations.

- can be approached as an instance of the more general "unification problem"
The Unification Problem

- modeled as a set of equations \( \exists l_1 = r_1, l_2 = r_2, \ldots \)?

where \( l_i, r_i \) e terms \( U \) variables

- terms are symbols up some arity

- variables can be mapped to terms

- a substitution \( \sigma \) maps variables to terms, and is said to unify \( l = r \) if \( \sigma(l) = \sigma(r) \)

- solving a unification problem means finding a substitution that unifies all its equations.
Unification Algorithm

4 operations:

- given $C = \{ l_1 = r_1, l_2 = r_2, \ldots \}$

1. **Delete**: trivial equations of form $L = L$ can be removed

2. **Decompose**: $f(l_1, l_2, \ldots, l_n) = f(r_1, r_2, \ldots, r_n)$ can be replaced with the set $\{ l_1 = r_1, l_2 = r_2, \ldots, l_n = r_n \}$

3. **Orient**: $L = r$ can be replaced with $r = L$ if $r$ is a variable and $L$ isn't

4. **Eliminate**: $L = r$, where $L$ is a variable, can be used to substitute $r$ for $L$ elsewhere in $C$
Unification Algorithm

4 operations:
-given $C = \{ l_1 = r_1, l_2 = r_2, \ldots \}$

$C$ is in solved form if:

- $l_1, l_2, \ldots$ are distinct variables
- $r_1, r_2, \ldots$ do not contain any variables
e.g., solve the unification problem \( \{ \alpha = f(x), g(\alpha, \alpha) = g(\alpha, \beta) \} \)

\[
\{ \alpha = f(x), g(\alpha, \alpha) = g(\alpha, \beta) \}
\]

eliminate \( \alpha \)

\[
\{ \alpha = f(x), g(f(x), f(x)) = g(f(x), \beta) \}
\]

decompose \( g \)

\[
\{ \alpha = f(x), f(x) = f(x), f(x) = \beta \}
\]

delete

\[
\{ \alpha = f(x), f(x) = \beta \}
\]

orient

\[
\{ \alpha = f(x), \beta = f(x) \}
\]

\[
\sigma = \{ \alpha \mapsto f(x), \beta \mapsto f(x) \}
\]
Type inference as Unification

e.g., solve \{ \text{int} = \text{int}, X = \text{int}, Y = \text{int} \to Z, Z = \text{int} \}\}

delete
\{ X = \text{int}, Y = \text{int} \to Z, Z = \text{int} \}\}

eliminate Z
\{ X = \text{int}, Y = \text{int} \to \text{int}, Z = \text{int} \}\}

\sigma = \{ X \mapsto \text{int}, Y \mapsto \text{int} \to \text{int}, Z \mapsto \text{int} \}\}