Type Systems

&

Simply-typed λ-Calculus

Previously, grammar of λ-calculus

$$E ::= \underset{\text{(var)}}{x} \;\Big|\; \underset{\text{(abstraction)}}{\lambda x. E} \;\Big|\; \underset{\text{(application)}}{E\ E}$$

- in the typed λ-calculus, we will indicate the type τ "tau" of each argument of a function:

$$\lambda x : \tau . E$$

x is of type τ

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., int → 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 $()$

$\therefore$ in $\lambda x : \tau . E$,

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

Grammar of simply-typed λ-calculus

types $\tau ::= \text{int} \mid \text{unit} \mid \tau \to \tau$

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

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

$$\text{ADD}\ \frac{E_1 \to n_1, \quad E_2 \to n_2}{n_1 + n_2 \xrightarrow[\text{arith}]{} n}$$

Type Systems

- 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 $E \longrightarrow E'$

e.g., $\lambda x. x + 10$ is stuck!

$()\ 8$ is stuck!

The Typing Relation : $\vdash$

$\Gamma \vdash E : \tau$ asserts that under typing context Γ, E has type τ

gamma

- a typing context is a set of variables and their types

  e.g., $\emptyset$, $\{x : \text{int}, y : \text{unit} \to \text{unit}\}$

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

- if $\Gamma = \emptyset$, we say E is well-typed

Type Safety

Formally: if $\vdash E : \tau$ and $E \rightarrow^* E'$ then either E' is a value or $\exists E''$ s.t. $E' \rightarrow E''$

Two parts:

"**Preservation**": if $\vdash E : \tau$ and $E \rightarrow^* E'$ then $\vdash E' : \tau$

"**Progress**": if $\vdash E : \tau$ then either E is a value or $\exists E'$ s.t. $E \rightarrow E'$

Simply-typed λ-calculus type rules

$$\text{INT}\quad \frac{}{\Gamma \vdash n : \text{int}}$$

any context

$$\text{UNIT}\quad \frac{}{\Gamma \vdash () : \text{unit}}$$

$$\text{VAR}\quad \frac{}{\Gamma \vdash x : \tau}\quad \{x : \tau\} \subseteq \Gamma$$

Simply-typed λ-calculus type rules

$$\text{ADD}\quad \frac{\Gamma \vdash E_1 : \text{int},\ \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 + E_2 : \text{int}}$$

$$\text{ABS}\quad \frac{\Gamma \cup \{x : \tau\} \vdash E : \tau'}{\Gamma \vdash \lambda x : \tau . E : \tau \to \tau'}$$

$$\text{APP}\quad \frac{\Gamma \vdash E_2 : \tau' \quad \Gamma \vdash E_1 : \tau' \to \tau}{\Gamma \vdash E_1\, E_2 : \tau}$$

eg. prove $\vdash (\lambda x : int.\ 2 + x)\ 10 : int$

$$\text{APP}\ \dfrac{\text{ABS}\ \dfrac{\text{ADD}\ \dfrac{\text{INT}\ \dfrac{}{\Gamma \vdash 2 : int} \quad , \quad \text{VAR}\ \dfrac{}{\Gamma \vdash x : int}}{\Gamma\{x : int\} \vdash 2 + x : int}}{\vdash \lambda x : int.\ 2 + x \ : int \to int} \qquad \text{INT}\ \dfrac{}{\vdash 10 : int}}{\vdash (\lambda x : int.\ 2 + x)\ 10 : int}$$

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., $\underset{\text{(identity)}}{(\lambda x{:}\mathrm{int}.\,x)}\ 4 \xrightarrow[\beta]{} 4$

e.g., $\underbrace{(\lambda x.\,x\ x)}\underbrace{(\lambda x.\,x\ x)}$

what are their types?

can't be $t \to t'$, because it's applied to itself, which would make the first $(t \to t') \to \ldots$

Type Safety in the Simply-typed λ-Calculus

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

— what we can say is that all well-typed programs <u>terminate</u>

i.e., if $\vdash E : \tau$, 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. $\lambda x. \lambda y. \lambda z.\ \underbrace{\text{if } (y\ (4+x)) \text{ then } x \text{ else } z}$

- can we still perform type checking?

  infer type: $\lambda x{:}\text{int}.\ \lambda y{:}\text{int} \to \text{bool}.\ \lambda z{:}\text{int}.\ \ldots$

- how can we formally define this process?

Type constraints

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

- constraints will take the form of a set of equations between types and type variables

type vars

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

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

Deriving type variables + constraints

type variable $X, Y, Z, \ldots$

- update $\tau ::= \text{int} \mid \text{uint} \mid \tau \to \tau \mid X$
- new typing relation $\Gamma \vdash E : \tau \rhd C$ — E has type τ if C satisfied
- update rules to help derive values of τ and C

e.g.: C-ADD
$$\frac{\Gamma \vdash E_1 : \tau_1 \rhd C_1 \quad , \quad \Gamma \vdash E_2 : \tau_2 \rhd C_2}{\Gamma \vdash E_1 + E_2 : \text{int} \rhd C_1 \cup C_2 \cup \{\tau_1 = \text{int}, \tau_2 = \text{int}\}}$$

Deriving type variables + constraints

$$\text{C-ABS}\ \frac{\Gamma \cup \{x:\tau_1\} \vdash E : \tau_2 \triangleright C}{\Gamma \vdash \lambda x:\tau_1 . E : \tau_1 \rightarrow \tau_2 \triangleright C}$$

$$\text{C-APP}\ \frac{\Gamma \vdash E_1 : \tau_1 \triangleright C_1, \quad \Gamma \vdash E_2 : \tau_2 \triangleright C_2, \quad C = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \rightarrow X\}}{\Gamma \vdash E_1\ E_2 : X \triangleright C}$$

e.g., derive type constraints for the expression:

$$\lambda a{:}X.\ \lambda b{:}Y.\ 4 + (b\ (2 + a))$$

$$\begin{array}{c}
\Gamma \vdash 2 : \text{int} \triangleright \emptyset \qquad \Gamma \vdash a : X \triangleright \emptyset \\ \hline
\Gamma \vdash b : Y \triangleright \emptyset \qquad \Gamma \vdash 2+a : \text{int} \triangleright \{\text{int} \doteq \text{int},\ X \doteq \text{int}\} \\ \hline
\Gamma \vdash 4 : \text{int} \triangleright \emptyset \qquad \Gamma \vdash b\,(2+a) : Z \triangleright \{\text{int} \doteq \text{int},\ X \doteq \text{int},\ Y \doteq \text{int} \to Z\} \\ \hline
\Gamma\{a:X,\ b:Y\} \vdash 4 + (b\,(2+a)) : \text{int} \triangleright \{\text{int} \doteq \text{int},\ X \doteq \text{int},\ Y \doteq \text{int} \to Z,\ Z \doteq \text{int}\} \\ \hline
\Gamma\{a:X\} \vdash \lambda b{:}Y.\,4 + (b\,(2+a)) : Y \to \text{int} \triangleright \{\quad '' \quad\} \\ \hline
\vdash \lambda a{:}X.\,\lambda b{:}Y.\,4 + (b\,(2+a)) : X \to Y \to \text{int} \triangleright \{\quad '' \quad\}
\end{array}$$

e.g., derive type constraints for the expression:

$$\lambda a{:}X.\ \lambda b{:}Y.\ 4 + (b\ (z + a))$$

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

— 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 $\{ l_1 = r_1, l_2 = r_2, \ldots \}$ where $l_i, r_i \in$ terms $\cup$ variables
  - terms are symbols w/ some arity — e.g. $x, y, f(x), g(x, y, f(x))$
  - variables can be mapped to terms — e.g. α, β
- a substitution σ 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 w/ $r = l$ 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 $C = \{ l_1 = r_1, l_2 = r_2, \dots \}$

    - C is in solved form if:

        - $l_1, l_2, \dots$ are distinct variables

        - $r_1, r_2, \dots$ 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 α

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

decompose g

$$\{\alpha = f(x), f(x) = f(x), f(x) = \beta\}$$

delete

$$\{\alpha = f(x), f(x) = \beta\}$$

orient

$$\{\alpha = f(x), \beta = f(x)\}$$

$$\boxed{\sigma = \{\alpha \mapsto f(x), \beta \mapsto f(x)\}}$$

Type Inference as Unification

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

delete

$\{ X = int \ , \ Y = int \to Z \ , \ Z = int \}$

eliminate Z

$\{ X = int \ , \ Y = int \to int \ , \ Z = int \}$

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