Type Systems

"A type system is a syntactic method for automatically proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

Each program phrase is given a type.

- Can be viewed as an abstract execution of the program.
- Types are calculated compositionally, from the types of subphrases.

The type of a program phrase determines its legal usage.

- Ensures the absence of run-time type errors.
- Necessarily conservative.

Why Static Type Systems?

Early detection of errors

- typos
- misuses of a value
- violations of finite-state protocols

Abstraction and Modularity

- encapsulation in OO languages
- abstract types in functional languages

Documentation

- the only widely used form of program specification!
- unlike comments, cannot become outdated.

Why Static Type Systems? (cont.)

"Safety"

- "a safe language is one that protects its own abstractions"
- Java bytecode verification
- SPIN, Proof Carrying Code, Typed Assembly Language

Language Design

- types define the legal programs

Efficiency

- The original motivation for types.
- Allows the compiler to use specialized representations for primitive types.
- Can eliminate dynamic checks.

Type Safety

"Well-typed programs do not go wrong." – Robin Milner

Caution: Safety is a language-specific notion.

Formally, we'll call a language "safe" if its type system rejects all "eventually stuck" expressions.

Do not allow "eventually stuck" expressions to be classified

- if 3 then 3 else 2

Necessarily rules out some good programs as well

- if true then 3 else false

A Warning

The notion of "stuck expression" is just a theoretical device

Fewer things are stuck than you might expect

- null dereference
- out-of-bounds array access

More things are stuck than you might expect

- accessing a private field from outside a class
- violating the abstraction of a datatype

Simple Typing for Booleans

$$e \;::=\; \ldots \mid \text{true} \mid \text{false} \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

Introduce a type representing expressions that evaluate to booleans.

$$T \;::=\; \text{Bool}$$

Define a typing judgement (or typing relation).

- $e : T$.
- read "expression e has type T"

Use inference rules to define the typing relation.

$$\frac{}{\text{true} : \text{Bool}}\;(\text{T-True}) \qquad \frac{}{\text{false} : \text{Bool}}\;(\text{T-False})$$

Simple Typing for Booleans (cont.)

$$\frac{e_1 : \text{Bool} \quad e_2 : T \quad e_3 : T}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T}\;(\text{T-If})$$

- The use of T twice ensures that both branches have the same type.

Conservative

- "if 3 then 3 else 2" fails to typecheck
- "if true then 3 else false" fails to typecheck

How could we redesign the type system to accept "if true then 3 else false"? At what cost?

A Typing Derivation

$$\frac{}{\text{true} : \text{Bool}}\;(\text{T-True}) \quad \frac{}{\text{false} : \text{Bool}}\;(\text{T-False}) \quad \frac{e_1 : \text{Bool} \quad e_2 : T \quad e_3 : T}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T}\;(\text{T-If})$$

A derivation tree "calculates" the type of an expression.

$$\frac{\frac{}{\text{true} : \text{Bool}} \quad \frac{}{\text{false} : \text{Bool}} \quad \frac{\frac{}{\text{true} : \text{Bool}} \quad \frac{}{\text{false} : \text{Bool}} \quad \frac{}{\text{true} : \text{Bool}}}{\text{if true then false else true} : \text{Bool}}}{\text{if true then false else (if true then false else true)} : \text{Bool}}$$

The type of an expression is computed in a single derivation.

In contrast, the value of an expression is computed (via our operational semantics) using multiple derivations.

It is possible to define a "big-step" operational semantics, similar in style to our typing rules.

Simple Typing for Functions

$$e ::= \ldots \quad x \quad \lambda x.e \quad e_1\ e_2$$

Add a function type.

$$T ::= \text{Bool} \quad T_1 \to T_2$$

Problem: What is the type of x in $\lambda x.e$? It cannot be deduced compositionally.

- Could infer the type of x by its usage in e.
- Could explicitly annotate x with its type.

Simple Typing for Functions (cont.)

$$e ::= \ldots \quad \lambda x{:}\ T.e$$

Typecheck the function body in the context of the argument type.

$$\frac{x : T_1 \vdash e : T_2}{(\lambda x : T_1.e) : T_1 \to T_2}\ \text{(T-Abs)}$$

Use the context to provide a type for a free variable.

$$\frac{}{x : T \vdash x : T}\ \text{(T-Var)}$$

Example

$$\frac{\dfrac{}{x : \text{Bool} \vdash x : \text{Bool}}\ \text{(T-Var)}}{(\lambda x : \text{Bool}.x) : \text{Bool} \to \text{Bool}}\ \text{(T-Abs)}$$

In general, there may be multiple free variables...

Type Environments

The metavariable Γ represents type environments, which are sets of (variable name, type) pairs, each pair denoted $x : T$.

The typing relation is now $\Gamma \vdash e : T$, read "expression e has type T under the typing assumptions in Γ."

$$\frac{\Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash (\lambda x : T_1.e) : T_1 \to T_2}\ \text{(T-Abs)}$$

- Rename x if necessary, so that Γ has at most one pair for a given variable name.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(T-Var)}$$

Function application:

$$\frac{\Gamma \vdash e_1 : T_2 \to T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T}\ \text{(T-App)}$$

A Typing Derivation

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(T-Var)} \qquad \frac{\Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash (\lambda x : T_1.e) : T_1 \to T_2}\ \text{(T-Abs)} \qquad \frac{\Gamma \vdash e_1 : T_2 \to T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T}\ \text{(T-App)}$$

$$\frac{\dfrac{\dfrac{f : \text{Bool} \to \text{Bool} \in \{f : \text{Bool} \to \text{Bool}\}}{\{f : \text{Bool} \to \text{Bool}\} \vdash f : \text{Bool} \to \text{Bool}} \quad \dfrac{}{\{f : \text{Bool} \to \text{Bool}\} \vdash \text{true} : \text{Bool}}}{\{f : \text{Bool} \to \text{Bool}\} \vdash f\ \text{true} : \text{Bool}}}{\{\} \vdash \lambda f : \text{Bool} \to \text{Bool}.(f\ \text{true}) : (\text{Bool} \to \text{Bool}) \to \text{Bool}}$$

Simply Typed Lambda Calculus (with Booleans)

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{(T-Var)}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}} \text{(T-True)}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{Bool}} \text{(T-False)}$$

$$\frac{\Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash (\lambda x : T_1.e) : T_1 \rightarrow T_2} \text{(T-Abs)}$$

$$\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{(T-If)}$$

$$\frac{\Gamma \vdash e_1 : T_2 \rightarrow T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T} \text{(T-App)}$$

Typechecking Church Numerals

Let Num $\equiv$ (Bool $\rightarrow$ Bool) $\rightarrow$ Bool $\rightarrow$ Bool

- zero:Num $\equiv \lambda s : \text{Bool} \rightarrow \text{Bool}.\lambda z : \text{Bool}.z$
- one:Num $\equiv \lambda s : \text{Bool} \rightarrow \text{Bool}.\lambda z : \text{Bool}.s\ z$
- two:Num $\equiv \lambda s : \text{Bool} \rightarrow \text{Bool}.\lambda z : \text{Bool}.s(s\ z)$

succ : Num $\rightarrow$ Num $\equiv \lambda n : \text{Num}.\lambda s : \text{Bool} \rightarrow \text{Bool}.\lambda z : \text{Bool}.s(n\ s\ z)$

Problem 1: Can apply succ to $(\lambda s : \text{Bool} \rightarrow \text{Bool}.\lambda z : \text{Bool}.\text{true})$

Problem 2: Can addition be defined in terms of succ?

- plus : Num $\rightarrow$ Num $\rightarrow$ Num $\not\equiv \lambda n_1 : \text{Num}.\lambda n_2 : \text{Num}.(n_1\ \text{succ}\ n_2)$

These kinds of problems are the motivation for language designers.

Solution: Add primitive values and types for numbers.

Typechecking Recursion

fix $\equiv \lambda g.(\lambda f.g(f\ f))(\lambda f.g(f\ f))$

Problem: Self-application cannot be typechecked.

$$\frac{\dfrac{\dfrac{? = T_2 \rightarrow T}{\{x :?\} \vdash x : T_2 \rightarrow T} \quad \dfrac{? = T_2}{\{x :?\} \vdash x : T_2}}{\{x :?\} \vdash (x\ x) : T}}{\{\} \vdash \lambda x :?.(x\ x) :? \rightarrow T}$$

Recursion as a Primitive

Add a new kind of expression of the form "fix e"

Operational Semantics

$$\frac{e \longrightarrow e'}{\text{fix } e \longrightarrow \text{fix } e'} \text{(E-Fix)}$$

$$\frac{}{\text{fix } (\lambda x : T.e) \longrightarrow [x \mapsto \text{fix } (\lambda x : T.e)]e} \text{(E-FixRed)}$$

Typing Rule

$$\frac{\Gamma \vdash e : T \rightarrow T}{\Gamma \vdash \text{fix } e : T} \text{(T-Fix)}$$

Recursion Example

$$\frac{e \longrightarrow e'}{\text{fix } e \longrightarrow \text{fix } e'} \text{(E-Fix)} \qquad \frac{}{\text{fix } (\lambda x : T.e) \longrightarrow [x \mapsto \text{fix } (\lambda x : T.e)]e} \text{(E-FixRed)}$$

$$\frac{\Gamma \vdash e : T \rightarrow T}{\Gamma \vdash \text{fix } e : T} \text{(T-Fix)}$$

Let $\text{NatFun} \equiv \text{Nat} \rightarrow \text{Nat}$

$\text{factf} : \text{NatFun} \rightarrow \text{NatFun}$
$\equiv \lambda f : \text{NatFun}.\lambda n : \text{Nat}.\text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)$

$\text{fact} : \text{Nat} \rightarrow \text{Nat} \equiv \text{fix factf}$

$\text{fix factf} \longrightarrow \lambda n : \text{Nat}.\text{if } n = 0 \text{ then } 1 \text{ else } n * (\text{fix factf})(n-1)$