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
- information hiding via signatures in ML.

Documentation

- the only widely used form of program specification.
- unlike comments, types cannot become outdated.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
- secure information flow, secrecy

Language Design

- types imply an associated set of values and expressions
- 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 "typesafe" if its type system rejects all "eventually stuck" expressions.

Do not allow "eventually stuck" expressions to be classified

- if 3+12 then 4 else 2
  type error: 3+12 is not of type bool

Necessarily rules out some good programs as well

- if true then 3 else false
  type error: 3 and false do not have the same type

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 type

Simple Typing for Booleans

$$
\begin{array}{lcl}
e & ::= & \ldots \\
 & & \text{true} \\
 & & \text{false} \\
 & & \text{if } e_1 \text{ then } e_2 \text{ else } e_3
\end{array}
$$

Introduce a type representing expressions that evaluate to booleans.

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

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 4 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

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

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

$$\frac{}{\text{if (if true then false else true) then 3 else 4} : \text{Nat}}$$

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

Simple Typing for Functions

$$
\begin{array}{lll}
e & ::= & \ldots \\
& & x \\
& & \lambda x.e \\
& & e_1\ e_2
\end{array}
$$

Add a function type.

$$
\begin{array}{lll}
T & ::= & \text{Bool} \\
& & T_1 \rightarrow T_2
\end{array}
$$

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.)

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."

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

- 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:

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

A Typing Derivation

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

$$\frac{}{\{\} \vdash \lambda f : \text{Bool} \rightarrow \text{Nat}.(f\ \text{true}) :}$$

Typechecking Recursion

$\text{fix} \equiv \lambda g.(\lambda f.g(f\ f))(\lambda f.g(f\ f))$

Problem: self-application cannot be typechecked.

$$\frac{}{\Gamma \vdash \lambda x : T_1.(x\ x) : T_1 \rightarrow T}$$

In fact, the simply-typed lambda calculus is strongly normalizing: every well-typed term's evaluation terminates in a value!

Recursion as a Primitive

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

Operational Semantics

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

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

Example

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

Recursion as a Primitive (cont.)

Typing Rule

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

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

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

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

Recursion as a Primitive (cont.)

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

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

With recursion as a primitive, the simply-typed lambda calculus is Turing-complete!

Of course, it is therefore no longer strongly normalizing.

- $(\text{fix } \lambda x : \text{Bool}.x) : \text{Bool}$

Are language designers out of a job?

Type Soundness

Intuitively, the type system should rule out eventually stuck expressions.

Recall that e is eventually stuck if $e \stackrel{*}{\longrightarrow} e'$ and e' is stuck.

Theorem (Type Soundness) If $\vdash e : T$ then either e's evaluation does not terminate or there exists v such that $e \stackrel{*}{\longrightarrow} v$ and $\vdash v : T$.

- $\vdash e : T$ is shorthand for $\{\} \vdash e : T$

Soundness = Progress + Preservation

Theorem (Type Soundness) If $\vdash e : T$ then either e's evaluation does not terminate or there exists v such that $e \xrightarrow{*} v$ and $\vdash v : T$.

A well-typed term e is not stuck:

Theorem (Progress): If $\vdash e : T$, then either e is a value or there exists e' such that $e \longrightarrow e'$.

The $\longrightarrow$ relation preserves typing:

Theorem (Type Preservation): If $\Gamma \vdash e : T$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : T$.

Proving Progress

Theorem (Progress): If $\vdash e : T$, then either e is a value or there exists e' such that $e \longrightarrow e'$.

- Proof by strong induction on the depth of the derivation tree for $\vdash e : T$.
- Case analysis of the last rule used in the derivation.
- Inductive hypothesis allows us to assume the theorem for sub-expressions of e.
- Relies on a "canonical forms" lemma.

Canonical Forms

The typing rules make implicit assumptions about the values of various types. For example, T-If assumes that true and false are the only values of type Bool:

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

The Canonical Forms Lemma validates these assumptions, which are needed by the Progress and Type Preservation proofs.

Lemma (Canonical Forms):

1. If $\Gamma \vdash v : \text{Bool}$, then v is either true or false.
2. If $\Gamma \vdash v : T_1 \rightarrow T_2$, then v has the form $\lambda x : T_1.e$.

Proof "by inspection" of the typing rules.

Proving Type Preservation

Theorem (Type Preservation): If $\Gamma \vdash e : T$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : T$.

- Proof by strong induction on the depth of the derivation tree for $\Gamma \vdash e : T$.
- Case analysis of the last rule used in the derivation.
- Sub-case analysis of the last rule used in the derivation of $e \longrightarrow e'$.
- Inductive hypothesis allows us to assume the theorem for sub-expressions of e.
- Relies on a "substitution" lemma.

Substitution

To prove Type Preservation, we need to reason about the type correctness of the substitution function.

Well-typed substitution produces well-typed terms:

Lemma (Substitution): If $\Gamma \cup \{x : T\} \vdash e' : T'$ and $\Gamma \vdash v : T$, then $\Gamma \vdash [x \mapsto v]e' : T'$.

Proof by strong induction on the depth of the derivation tree for $\Gamma \cup \{x : T\} \vdash e' : T'$

Straightforward but somewhat tedious.