Types

- A program variable can assume a range of values during the execution of a program.
- An upper bound of such a range is called a type of the variable:
  - A variable of type "bool" is supposed to assume only boolean values.
  - If \( x \) has type "bool" then the boolean expression "\( \neg(x) \)" has a sensible meaning during every run of the program.

Typed and Untyped Languages

- Untyped languages:
  - The language does not restrict the range of values for a given variable.
  - Operations might be applied to inappropriate arguments.
  - The behavior in such cases might be unspecified.
  - The pure λ-calculus is an extreme case of an untyped language. (However, its semantics is completely specified.)

- Typed languages:
  - Variables are assigned (non-trivial) types.
  - A type system is the component of a language that keeps track of types.
  - Types might or might not appear in the program itself.
  - Languages can be explicitly typed or implicitly typed.

Errors and Soundness (Overview)

- An important application of type systems is to prevent execution errors during program execution.
- Languages where no program gives rise to execution errors are type-sound languages.

Trapped Execution Errors

- Trapped execution errors:
  - Cause the computation to stop immediately.
  - Give rise to well-specified behavior.
  - Sometimes enforced with hardware support.
  - E.g., division by zero.
  - E.g., dereferencing the address 0 (on many systems).
  - Even languages with powerful type systems permit trapped execution errors.

Untrapped Execution Errors

- Untrapped execution errors:
  - Behavior is unspecified (depends on machine state).
  - E.g., accessing past the end of an array.
  - E.g., jumping to an address in the data segment.
  - A program is deemed safe if it does not cause untrapped errors.
  - Languages where all programs are safe are safe languages.
Good Behavior

- For a given language we designate a set of forbidden errors:
  - At least all the untrapped errors.
  - Maybe some trapped errors as well.
- A program fragment that does not give rise to forbidden errors has good behavior.
- A language where all legal programs have good behavior is strongly checked.
  - No untrapped errors occur.
  - The programmer is responsible for avoiding some or all trapped errors.

Caveats, Perspective

- These definitions are somewhat simplistic.
  - Being typed, or being explicitly typed, can be seen as a matter of degrees.
  - Even the notion of execution error is difficult to make precise in a simple, general manner.
- The definitions pertain to overlapping concepts, with several ways of saying the same things (or almost).
  - A crucial, pervasive distinction is between run-time and compile-time notions.

Caveats, Perspective (Cont.)

- Many classes of execution errors are beyond the capability of (decidable) type systems.
- Type systems can be designed to help catch a variety of errors:
  - traced vs. untraced (garbage collection errors)
  - static vs. dynamic (for run-time code generation errors)
  - unclassified vs. secret (information flow violations)

Safe Languages

- There are typed languages that are not safe (weakly typed languages).
- All safe languages use types (either statically or dynamically).

<table>
<thead>
<tr>
<th></th>
<th>Typed</th>
<th>Untyped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Safe</td>
<td>Unsafe</td>
</tr>
<tr>
<td>Dynamic</td>
<td>ML, Java, ...</td>
<td>C, C++, ...</td>
</tr>
<tr>
<td></td>
<td>Scheme</td>
<td>some Lisp</td>
</tr>
</tbody>
</table>

- We will be concerned mainly with statically typed languages.

Preventing Forbidden Errors - Static Checking

- Forbidden errors can be caught by a combination of static and run-time checking.
- Detecting certain errors statically is undecidable in most languages.
- Static checking is however often preferable, even when it is incomplete.
  - Detects errors early, before testing and shipping.
  - Helps in compilation.
  - Supported by types.
  - E.g., ML, Modula-3, Java.

Preventing Forbidden Errors - Dynamic Checking

- Dynamic checking is required when static checking is undecidable and the approximations are not very good.
  - E.g., array-bounds checking.
- Run-time encodings of types are still used (e.g., in Java's instanceof).
- Dynamic checking delays the manifestation of errors.
- Dynamic checking can benefit from hardware support (e.g., nil-pointer checking).
Why Typed Languages?

- Development
  - Typechecking catches early many mistakes.
  - Types reduces debugging time.
  - Typed signatures are a powerful basis for design.
  - Typed signatures enable separate compilation.
- Maintenance
  - Types act as checked specifications.
  - Types can enforce abstractions.
- Execution
  - Static checking reduces the need for dynamic tests.
  - Safe languages are easier to analyze statically.
  - Compilers can generate better code.

Why Not Typed Languages?

- Type disciplines (particularly static ones) impose constraints on the programmer.
  - Some reasonable programs might be rejected.
  - But often they can be made well-typed easily.
- Dynamic safety checks can be costly.
  - 30% is a possible cost of bounds-checking in a tight loop.
  - Memory management should probably be automatic
    ⇒ We need a garbage collector.
  - In practice, the overall cost is much smaller.
  - Some applications are justified in using weakly typed languages.

Properties of Type Systems

- How do types differ from other program annotations
  - Types are more precise than comments.
  - Types are more easily mechanizable than program specifications.
- Expected properties of type systems:
  - Types should be enforceable.
  - Types should be checkable algorithmically.
  - Typing rules should be transparent:
    - It should be easy to predict whether a program will typecheck,
      or see why it does not.

What is a Type: Defining Type Equivalence

- Most type systems include a relation of type equivalence.
- Are X and Y equivalent?
  - type X = Real
  - type Y = Real
  - When they fail to match by virtue of having distinct type names, we have by-name equivalence.
  - When they match by virtue of being associated with similar types, we have structural equivalence.

Type Equivalence (Cont.)

- Most compilers use a combination of by-name and structural equivalence.
  (sometimes without a proper specification).
- Structural equivalence has some advantages:
  - It can be defined easily, without strange special cases.
  - It easily allows "anonymous" types.
  - It works well with data sent over networks, with persistence.
  - It works well with program sources developed in pieces.
  - It can be limited through "branding":
    type X = Real brand Temperature
    type Y = Real brand Speed

When Types Don't Match: Coercions

- Many languages do not give up when types don’t match.
  - Instead, they apply coercions.
  - Sometimes the coercion happens dynamically (and costs).
- Languages vary in their use of coercions.
  - For languages with lots of basic types (e.g., COBOL) frequent coercions are necessary.
  - Many languages allow coercions at least for numeric types.
Coercions (Cont.)

- Silent coercions have pluses and minuses:
  - They free the programmer from tedious conversions.
  - Typechecking becomes harder to predict.
  - Simple errors may become serious mistakes.
  - If a coercion does copying, then data structures may not be shared as intended.

Why Formal Type Systems?

- Many typed languages have informal descriptions of type systems (e.g., in language reference manuals).
- A fair amount of careful analysis is required to avoid false claims of type safety.
- A formal presentation of a type system is a precise specification of the type checker.
  - And it allows formal proofs of type safety.
- But even informal knowledge of the principles of type systems help.

Formalizing a Type System

A multi-step process:
1. Syntax
   - Of expressions (programs)
   - Of types
   - Issues of binding and scoping
2. Static semantics (typing rules)
   - Define the typing judgments and its derivation rules
3. Dynamic semantics (e.g., operational semantics)
   - Define the evaluation judgment and its derivation rules
4. Type soundness (i.e., a soundness theorem)
   - To relate the static and dynamic semantics

Typing Judgments

- Judgments
  - A judgment is a statement \( J \) about certain formal entities
  - It may be valid (universally true): \( \vdash J \)
  - It may be provable: \( \Gamma \vdash J \)
- A common form of the typing judgment: \( \Gamma \vdash e : \tau \)
  - \( e \) is an expression and \( \tau \) is a type
- \( \Gamma \) is a set of type assignments for the free variables of \( e \).
  - Defined by the grammar
  - Usually viewed as a set of type assignments
  - Type assignments for variables not free in \( e \) are not relevant
  - E.g., \( x : \text{int}, y : \text{int} \quad \vdash x + y : \text{int} \)

Typing Derivations

- A typing derivation is a derivation of a typing judgment.
- Example:

  \[
  \begin{array}{c}
  \Gamma \vdash \text{int} \\
  x : \tau \in \Gamma \\
  \Gamma \vdash x : \tau \\
  \Gamma \vdash \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
  \hline
  \Gamma \vdash e_1 + e_2 : \text{int}
  \end{array}
  \]

  - We say that \( \Gamma \vdash e : \tau \) to denote that there is a derivation of this typing judgment.
  - Type checking: given \( \Gamma, e \) and \( \tau \) find a derivation.
  - Type inference: given \( \Gamma \) and \( e \), find \( \tau \) and a derivation.
Proving Type Soundness

- A typing assertion should have a truth value.
- We define what it means for a value to have a type:
  \[ v \in \mathbb{K}_\tau \]
  (e.g., \( 5 \in \mathbb{K}_{\text{int}} \) and \( \text{true} \in \mathbb{K}_{\text{bool}} \))
- We then define what it means for an expression to have a type:
  \[ e \in \mid \tau \mid \text{ iff } \forall v. (e \downarrow v \Rightarrow v \in \mathbb{K}_\tau) \]
- We prove type soundness:
  If \( \vdash e : \tau \) then \( e \in \mid \tau \mid \)
  or equivalently:
  If \( \vdash e : \tau \) and \( e \downarrow v \) then \( v \in \mathbb{K}_\tau \)
- This implies safe execution (since the result of an unsafe execution is not in \( \mathbb{K}_\tau \) for any \( \tau \)).

Proving Type Soundness: Variants

- The approach just outlined works with respect to a big-step operational semantics.
- There are others, in particular a denotational approach:
  If \( \vdash e : \tau \) then the meaning of \( e \) is in the meaning of \( \tau \).

Next

- We will give formal description of first-order type systems (no type variables):
  - Simple types (integers and booleans)
  - Function types (simply typed \( \lambda \)-calculus)
  - Structured types (products and sums)
  - Imperative types (references and exceptions)
  - Recursive types
  - Subtypes
- The type systems of most common languages are first-order (e.g., Pascal).

Next (Cont.)

- Then we move to second-order type systems:
  - Polymorphism
  - Abstract types
- Later we will also treat objects and object types.

Simply-Typed Lambda Calculus

- Syntax:
  Terms: \( e ::= x \mid \lambda x : \tau. e \mid e_1 e_2 \mid n \mid e_1 + e_2 \mid \text{iszero } e \)
  \mid \text{true} \mid \text{false} \mid \text{not } e \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3
- Types: \( \tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2 \)
- \( \tau_1 \rightarrow \tau_2 \) is the function type
- \( \rightarrow \) associates to the right
  - Formal arguments have typing annotations
- This language is also called F₁
- We may make slight changes at base types (change primitives booleans and integers, add types).

Static Semantics of F₁

- The typing judgment
  \[ \Gamma \vdash e : \tau \]
- The typing rules
  \[
  \begin{align*}
  x : \tau & \in \Gamma & & \Gamma, x : \tau, e : \tau' \vdash e : \tau' \\
  \Gamma \vdash x : \tau & & \Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau' \\
  \Gamma \vdash e_1 : \tau_1 \rightarrow \tau & & \Gamma \vdash e_2 : \tau_2 \\
  \Gamma \vdash e_1 e_2 : \tau
  \end{align*}
  \]
Static Semantics of F₁ (Cont.)

- More typing rules

\[
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}
\]

Typing Derivation in F₁

- Consider the term

\[
\lambda x : \text{int}. \lambda b : \text{bool}. \text{if } b \text{ then } f x \text{ else } x
\]
- With the typing assignment \( f : \text{int} \to \text{int} \)

\[
\begin{align*}
\Gamma &\vdash f : \text{int} \to \text{int} \\
\Gamma &\vdash x : \text{int} \\
\Gamma &\vdash b : \text{bool} \\
\Gamma &\vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : \text{int}
\end{align*}
\]

where \( \Gamma = f : \text{int} \to \text{int}, x : \text{int}, b : \text{bool} \)

Typechecking in F₁

- Type checking is easy because:
  - Typing rules are syntax directed.
  - Typing rules are compositional.
  - All local variables are annotated with types.
- In fact, type inference is also easy for F₁.
- Without type annotations, an expression need not have a unique type:

\[
\begin{align*}
\vdash \lambda x : \tau. e \\
\vdash \lambda x : \tau. e
\end{align*}
\]

Operational Semantics of F₁

- Judgment:

\[
\frac{}{e \Downarrow v}
\]

Values

\[v ::= n \mid \text{true} \mid \text{false} \mid \lambda x : \tau. e\]

Typability

- We may erase types from expressions, systematically:

\[
\begin{align*}
\text{erase}(x) &= x \\
\text{erase}(e_1 e_2) &= \text{erase}(e_1) \text{erase}(e_2) \\
\text{erase}(\lambda x : \tau. e) &= \lambda x. \text{erase}(e)
\end{align*}
\]

- Then we may ask whether an untyped expression is typable (in a given environment \( \Gamma \), which is empty when the expression is a combinator):

Given \( e \), are there \( e' \) and \( \tau \) such that

\[
\text{erase}(e') = e \quad \text{and} \quad \Gamma \vdash e' : \tau ?
\]

For example, \( \lambda x. x \) is typable in the empty environment (in more than one way).

Operational Semantics of F₁ (Cont.)

- Call-by-value evaluation rules (sample)

\[
\begin{align*}
\lambda x : \tau. e &\Downarrow e_1 \Downarrow e_2 \Downarrow [n2/x]' \Downarrow v \\
e_1 \Downarrow n_1 &\quad e_2 \Downarrow n_2 \quad n = n_1 + n_2 \\
\downarrow &\downarrow \\
e_1 \Downarrow \text{true} &\quad e_2 \Downarrow v \quad \text{if } e_1 \text{ then } e_1 \text{ else } e_2 \Downarrow v \\
e_1 \Downarrow \text{false} &\quad e_2 \Downarrow v \quad \text{if } e_1 \text{ then } e_1 \text{ else } e_2 \Downarrow v
\end{align*}
\]
Type Soundness for $F_1$

- **Theorem:**
  - If $\vdash e : \tau$ and $e \Downarrow v$ then $\vdash v : \tau$.
  - Also called, subject reduction theorem, type preservation theorem.
- **Try to prove it by induction on $e$:**
  - It won't work because $[v_2/x]e'_1$ in the evaluation of $e_1 e_2$.
  - Same problem with induction on $\vdash e : \tau$.
- **Try to prove it by induction on $\tau$:**
  - It won't work because $e_1$ has a "bigger" type than $e_1 e_2$.
- **Try to prove it by induction on $e \Downarrow v$:**
  - To address the issue of $[v_2/x]e'_1$.
  - It works!

Significance of Type Soundness

- The theorem says that the result of an evaluation has the same type as the initial expression.
- The theorem does not say that:
  - The evaluation never gets stuck (e.g., trying to apply a non-function, to add non-integers, etc.).
  - The evaluation terminates.
- Even though both of the above facts are true of $F_1$.
- We may use a small-step semantics to prove that the execution never gets stuck.
- Exercise: Execution always terminates in $F_1$.

Small-Step Contextual Semantics for $F_1$

- We define redexes:
  - $r ::= n_1 + n_2 | \text{if } b \text{ then } e_1 \text{ else } e_2 | (\lambda x: \tau. e_1) v_2$
- and contexts:
  - $H ::= \cdot | H_1 + e_2 | n_1 + H_2 | \text{if } H \text{ then } e_1 \text{ else } e_2 | H_1 e_2 | (\lambda x: \tau. e_1) H_2$
- and local reduction rules:
  - $n_1 + n_2 \rightarrow n_1 + n_2$
  - if true then $e_1$ else $e_2$ $\rightarrow$ $e_1$
  - if false then $e_1$ else $e_2$ $\rightarrow$ $e_2$
  - $(\lambda x: \tau. e_1) v_2 \rightarrow [v_2/x] e_1$
- and one global reduction rule:
  - $H[r] \rightarrow H[e]$ if $r \rightarrow e$
An Alternative: Explicit Errors

- We may also obtain these properties from big-step operational semantics or denotational semantics.
- For this, we introduce an error value "wrong" in evaluation rules for "wrong" expressions, or as the meaning of those expressions.
- "wrong" does not have a typing rule.
- The main theorem is then that "well-typed programs don't go wrong".

Product Types - Static Semantics

- Extend the syntax with (binary) tuples
  \[ e ::= \ldots | (e_1, e_2) \]
  \[ \tau ::= \ldots | \tau_1 \times \tau_2 \]
- This language is sometimes called F_1.
- Same typing judgment \( \Gamma \vdash e : \tau \)
  \[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \]
  \[ \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \]
  \[ \Gamma \vdash \text{fst} e : \tau_1 \quad \Gamma \vdash \text{snd} e : \tau_2 \]

Product Types: Dynamic Semantics and Soundness

- New form of values: \[ v ::= \ldots | (v_1, v_2) \]
- New (big step) evaluation rules:
  \[ e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \]
  \[ (e_1, e_2) \Downarrow (v_1, v_2) \]
  \[ e \Downarrow (v_1, v_2) \quad e \Downarrow (e_1, e_2) \]
  \[ \text{fst} e \Downarrow v_1 \quad \text{snd} e \Downarrow v_2 \]
- New contexts: \[ H ::= \ldots | (H_1, e_2) | (v_1, H_2) | \text{fst} H | \text{snd} H \]
- New redexes:
  \[ \text{fst} (v_1, v_2) \rightarrow v_1 \]
  \[ \text{snd} (v_1, v_2) \rightarrow v_2 \]
- Type soundness holds just as before.

Records

- Records are like tuples with labels.
- New form of expressions
  \[ e ::= \ldots | \{ \text{L}_1 = e_1, \ldots, \text{L}_n = e_n \} | e.L \]
- New form of values
  \[ v ::= \{ \text{L}_1 = v_1, \ldots, \text{L}_n = v_n \} \]
- New form of types
  \[ \tau ::= \ldots | \tau_1 \times \tau_2 \]
- \text{... follows the model of F}_1.
- Typing rules
- Evaluation rules
- Type soundness

Sum Types

- We need types of the form
  - either an int or a float
  - either 0 or a pointer
  - either true or false
- These are called disjoint union types.
- New form of expressions and types
  \[ e ::= \ldots | \text{injl} e | \text{injr} e \]
  \[ e ::= \ldots | \text{injl} x \rightarrow e_1 | \text{injr} y \rightarrow e_2 \]
  \[ e ::= \ldots | \tau_1 \times \tau_2 \]
- A value of type \( \tau_1 \times \tau_2 \) is either a \( \tau_1 \) or a \( \tau_2 \).
- Like union in C or Pascal, but safe.
- Distinguishing between components is under compiler control.
- Case is a binding operator: \( x \) is bound in \( e_1 \) and \( y \) is bound in \( e_2 \).

Examples with Sum Types

- Consider the type "unit" with a single element called *.
- The type "optional integer" defined as "unit + int".
  - Useful for optional arguments or return values
    - No argument: \( \text{injl} \) *
    - Argument is \( \text{injr} \) 0
  - To use the argument you must test the kind of argument.
    - Case and \( \text{injl} x \rightarrow "\text{no_arg_case}" | \text{injr} y \rightarrow "\ldots y\ldots" \)
    - \( \text{injl} \) and \( \text{injr} \) are tags and case is tag checking
- Bool is a union type: \( \text{bool} = \text{unit} + \text{unit} \)
  - \text{true} is \( \text{injl} * \)
  - \text{false} is \( \text{injr} * \)
  - \( \text{if} \) then \( e_1 \) else \( e_2 \) is \( e ::= \ldots | \text{injl} x \rightarrow e_1 | \text{injr} y \rightarrow e_2 \)
Static Semantics of Sum Types

- New typing rules

\[ \Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2 \]
\[ \Gamma \vdash \text{injl} \, e : \tau_1 + \tau_2 \quad \Gamma \vdash \text{injr} \, e : \tau_1 + \tau_2 \]
\[ \Gamma \vdash e_1 : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \]
\[ \Gamma, y : \tau_2 \vdash e_2 : \tau \]
\[ \Gamma \vdash \text{case} \, e_1 \, \text{of} \, \text{injl} \, x \Rightarrow e_1 \mid \text{injr} \, y \Rightarrow e_2 : \tau \]

- Types are not unique anymore (without more annotations for sums):

  \[ \text{injl} : \text{int} + \text{bool} \]
  \[ \text{injl} : \text{int} + (\text{int} \rightarrow \text{int}) \]

Dynamic Semantics of Sum Types

- New values

\[ \nu ::= \ldots \mid \text{injl} \, \nu \mid \text{injr} \, \nu \]

- New evaluation rules

\[ e \downarrow \nu \quad e \downarrow \nu \]
\[ \text{injl} \, e \downarrow \text{injl} \, \nu \quad \text{injr} \, e \downarrow \text{injr} \, \nu \]
\[ e \downarrow \text{injl} \, \nu \quad [v/x]e_1 \downarrow \nu' \]
\[ \text{case} \, e \, \text{of} \, \text{injl} \, x \Rightarrow e_1 \mid \text{injr} \, y \Rightarrow e_r : \downarrow \nu' \]
\[ e \downarrow \text{injr} \, \nu \quad [v/x]e_r \downarrow \nu' \]
\[ \text{case} \, e \, \text{of} \, \text{injl} \, x \Rightarrow e_1 \mid \text{injr} \, y \Rightarrow e_r : \downarrow \nu' \]

Type Soundness for \( F_1 \):

- Type soundness still holds.

- There is no way to use a \( \tau_1 + \tau_2 \) inappropriately.

- The key is that the only way to use a \( \tau_1 + \tau_2 \) is with case, which ensures that one is not using a \( \tau_1 \) as a \( \tau_2 \).

- In C or Pascal checking the tag is the responsibility of the programmer (unsafe).