Values and Types

- We divide the universe of values according to **types**

- A **type** is a **set** of values and a set of **operations** on them

- How is each value **represented**? How is each operation **implemented**?

| Integers with +,-,etc.   | Booleans with &,|,~   |
|-------------------------|----------------|
| machine integer         | machine bit or byte  |
| HW instruction          | HW instruction/sequence |

<table>
<thead>
<tr>
<th>Arrays with read,update</th>
<th>Functions with apply</th>
</tr>
</thead>
<tbody>
<tr>
<td>contiguous block of memory</td>
<td>closures</td>
</tr>
<tr>
<td>address arithmetic</td>
<td>jsr instruction</td>
</tr>
<tr>
<td>+ indirect addressing</td>
<td></td>
</tr>
</tbody>
</table>
Atomic vs Constructed Types

- **Atomic** (or primitive) types are those whose values cannot be taken apart or constructed by user code.
  - Can only manipulate using built-in language operators.
  - Typically includes the types that have direct hardware support, e.g., integers, floats, pointers, instructions.

- **Composite** types are built from other types using type constructors.
  - User code can construct values, inspect/modify internals.
  - E.g. arrays, records, unions, function.
Static Type Checking

- High-level languages differ from machine code in that explicit types appear and type violations are usually caught at some point.

- **Static type checking** is probably most common: FORTRAN, Algol, Pascal, C/C++, Java, Scala, etc.

- Types are associated with *identifiers* (variables, parameters, functions).

- Every use of an identifier can be checked for type-correctness *before* program is run.

- “Well-typed programs don’t go wrong” (*if* type system is *sound*).

- Compilers can generate *efficient code* because it knows how each value is represented.

- Type declarations provide useful *documentation* for code.
Dynamic Type Checking

- Dynamic type checking occurs in LISP, Smalltalk, Python, JavaScript, many other scripting languages.

- Types are attached to values (usually as explicit tags).

- The type associated with an identifier can vary.

- Correctness of operations can’t (in general) be checked until run time.

- Type violations become checked run-time errors.

- Generating optimized code and value representations is hard.

- Programs can be harder to read.
Static Type Systems

- Main goal of a type system is to characterize programs that won’t “go wrong" at runtime.

- Informally, we want to avoid programs that confuse types, e.g. by trying to add booleans to reals, or take the square root of a string.

- More formally, we can give a set of typing rules (sometimes called static semantics) from which we can derive typing judgments about programs.

- Program is well-typed if-and-only-if we can derive a typing judgment for it.
Typing Judgments

Each judgment has the form

\[ TE \vdash e : t \]

Intuitively this says that expression \( e \) has type \( t \), under the assumption that the type of each free variable in \( e \) is given by the type environment \( TE \).

We write \( TE(x) \) for the result of looking up \( x \) in \( TE \), and \( TE + \{x \mapsto t\} \) for the type environment obtained from \( TE \) by extending it with a new binding from \( x \) to \( t \).
Rules for a simple language

\[
\begin{align*}
TE(x) &= t \\
\frac{TE \vdash x : t}{(\text{Var})}
\end{align*}
\]

\[
\begin{align*}
TE \vdash i : \text{Int} \\
(i) 
\end{align*}
\]

\[
\begin{align*}
TE \vdash e_1 : \text{Int} & \quad TE \vdash e_2 : \text{Int} \\
\frac{}{TE \vdash (+ e_1 e_2) : \text{Int}} (\text{Add})
\end{align*}
\]

\[
\begin{align*}
TE \vdash e_1 : \text{Int} & \quad TE \vdash e_2 : \text{Int} \\
\frac{}{TE \vdash (\leq e_1 e_2) : \text{Bool}} (\text{Leq})
\end{align*}
\]

\[
\begin{align*}
TE \vdash e_1 : \text{Int} & \quad TE \vdash e_2 : \text{Int} & \quad TE + \{x \mapsto t_1\} \vdash e_2 : t_2 \\
\frac{}{TE \vdash (\text{let } x e_1 e_2) : t_2} (\text{Let})
\end{align*}
\]

\[
\begin{align*}
TE(x) &= t & \quad TE \vdash e : t \\
\frac{}{TE \vdash (\text{:=} x e) : t} (\text{Assgn})
\end{align*}
\]

\[
\begin{align*}
TE \vdash e_1 : \text{Bool} & \quad TE \vdash e_2 : t & \quad TE \vdash e_3 : t \\
\frac{}{TE \vdash (\text{if } e_1 e_2 e_3) : t} (\text{If})
\end{align*}
\]

\[
\begin{align*}
TE \vdash e_1 : \text{Bool} & \quad TE \vdash e_2 : t \\
\frac{}{TE \vdash (\text{while } e_1 e_2) : \text{Int}} (\text{While})
\end{align*}
\]

Assumes just two types: Int and Bool
Static Type Checking

- We can turn the typing rules into a recursive type-checking algorithm.

- A type checker is very similar to the evaluators we have already built:
  - It is parameterized by a type environment.
  - It dispatches according to the syntax of the expression being checked (note there is exactly one rule for each expression form).
  - It calls itself recursively on sub-expressions.
Static Type Checking (2)

But there are some differences:

- Type checker returns a type, not a value
- It must examine every possible execution path, but just once
  - e.g. it examines both arms of a conditional expression (not just one)
  - e.g. if our language has functions, it processes the body of each function only once, no matter how many places the function is called from

Most languages require the types of function parameters and return values to be declared explicitly. The type checker can use this info to check separately that applications of the function are correctly typed and that the body of the function is correctly typed.
Flexibility of Dynamic Type Checking

- Static type checking offers the great advantage of catching errors early.

- And it generally supports more efficient execution.

- So why ever consider dynamic type checking?

- Simplicity. For short or simple programs, it’s nice to avoid the need to declare the types of identifiers.

- Flexibility. Static type checking is inherently more conservative about what programs it allows.
Conservative Typing

- For example, suppose function \( f \) happens always to return \( \text{false} \). Then

\[
(\text{if } f() \text{ then } "a" \text{ else } 2) + 2
\]

will never cause a run-time type error, but it will still be rejected by a static type system.

- Dynamic typing allows container data structures to contain mixtures of values of arbitrary types, e.g.

\[
\text{List}(2, \text{true}, 3.14)
\]
Type Inference

Some statically typed languages, like ML (and to a lesser extent Scala), offer alternative ways to regain the flexibility of dynamic typing, via type inference and polymorphism.

Type inference works like this:

- The types of identifiers are automatically inferred from the way they are used.
- The programmer is no longer required to declare the types of identifiers (although this is still permitted).
- Method requires that the types of operators and literals is known.
Inference Examples

(let f (fun (x) (+ x 2))
  (@ f y))

The type of x must be int because it is used as an arg to +. So the type of f must be int → int (i.e. the type of functions that expect an int argument and return an int result), and y must be an int.

(let f (fun (x) (cons x nil))
  (@ f true))

Suppose x has some type t. Then the type of f must be t → (list t). Since f is applied to a bool, we must have t = bool.

For the moment, we assume that f must be given a unique monomorphic type; we will relax this later…
Systematic Inference

Here’s a harder example:

\[
\begin{align*}
\text{(let f (fun (x) (if x p q))} \\
\quad (+ 1 (@ f r)))
\end{align*}
\]

Can only infer types by looking at both the function’s body and its application.

In general, we can solve the inference task by extracting a collection of typing constraints from the program’s AST, and then finding a simultaneous solution for the constraints using unification.

Extracted constraints tell us how types must be related if we are to be able to find a typing derivation. Each node generates one or more constraints.
To handle this example, we’ll need some extra typing rules:

\[
\frac{TE + \{x \mapsto t_1\} \vdash e : t_2}{TE \vdash (\text{fun } (x) \ e) : t_1 \to t_2}
\]  \quad \text{(Fn)}

\[
\frac{TE \vdash e_1 : t_1 \to t_2 \quad TE \vdash e_2 : t_1}{TE \vdash (@ e_1 \ e_2) : t_2}
\]  \quad \text{(Appl)}
Solution: \( t_1 = t_7 = t_8 = t_9 = t_3 = t_5 = t_p = t_6 = t_q = \text{int} \)
\( t_4 = t_x = t_{11} = t_r = \text{bool} \)
\( t_2 = t_f = t_{10} = \text{bool} \to \text{int} \)

Node Rule Constraints
1 Let \( t_f = t_2 \) \( t_1 = t_7 \)
2 Fun \( t_2 = t_x \to t_3 \)
3 If \( t_4 = \text{bool} \) \( t_3 = t_5 = t_6 \)
4 Var \( t_4 = t_x \)
5 Var \( t_5 = t_p \)
6 Var \( t_5 = t_q \)
7 Add \( t_7 = t_8 = t_9 = \text{int} \)
8 Int \( t_8 = \text{int} \)
9 Appl \( t_{10} = t_{11} \to t_9 \)
10 Var \( t_{10} = t_f \)
11 Var \( t_{11} = t_r \)
Drawbacks of Inference

Consider this variant example:

\[
\text{(let } f \text{ (fun } (x) (\text{if } x \text{ p false}) \text{ (+ 1 } (@ f \text{ r}))\text{)}
\]

Now the body of \( f \) returns type \( \text{bool} \), but it is used in a context expecting an \( \text{int} \).

The corresponding extracted constraints will be inconsistent; no solution can be found. Can report a type error to the programmer.

But which is wrong, the definition of \( f \) or the use? No good way to associate the error message with a single program point.