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.**
- machine integer
- HW instruction

**Booleans with &, |, ~**
- machine bit or byte
- HW instruction/sequence

**Arrays with read, update**
- contiguous block of memory
- address arithmetic
- + indirect addressing

**Functions with apply**
- closures
- jsr instruction
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 the 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.

- 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

\[ TE(x) = t \]
\[ \frac{}{TE \vdash x : t} \quad \text{(Var)} \]

\[ TE \vdash i : \text{Int} \]
\[ \frac{}{(\text{Int})} \]

\[ TE \vdash e_1 : \text{Int} \quad TE \vdash e_2 : \text{Int} \]
\[ \frac{}{TE \vdash (+ e_1 e_2) : \text{Int}} \quad \text{(Add)} \]

\[ TE \vdash e_1 : \text{Int} \quad TE \vdash e_2 : \text{Int} \]
\[ \frac{}{TE \vdash (\leq e_1 e_2) : \text{Bool}} \quad \text{(Leq)} \]

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

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

Assumes just two types: \text{Int} and \text{Bool}

Matches language we used for dynamic semantics before
The typing rules are just (another) formal system in which judgments can be derived.

How do we connect this system with our slogan that “well-typed programs don’t go wrong”?

First we need an auxiliary judgment system assigning types to values, written $\models v : t$.

For example, we would have $\models i : \text{Int}$ for every integer $i$, $\models \text{true} : \text{Bool}$, and $\models \text{false} : \text{Bool}$.

We also add a special value $\text{error}$ which does not belong to any type: $\not\models \text{error} : t$

We extend this notation to environments and stores, and write $\models E, S : TE$ iff $\text{dom}(E) = \text{dom}(TE)$ and $\models S(E(x)) : TE(x), \forall x \in \text{dom}(E)$. 
Dynamic Semantics Revisited

Recall our formal dynamic semantics for this language, using judgments like this: \( \langle e, E, S \rangle \downarrow \langle v, S' \rangle \)

Previously, we said expressions corresponding to runtime errors simply had no applicable rule (they were “stuck”)

Let’s change the system slightly by adding new rules so that all expressions corresponding to runtime errors evaluate to \text{error} instead.
Type Safety Theorem

Now, if everything has been defined correctly (our type system is **sound**), we should be able to prove a theorem roughly like this:

\[
\text{If } TE \vdash e : t \text{ and } \models E, S : TE \text{ and } \langle e, E, S \rangle \Downarrow \langle v, S' \rangle \text{ then } \models v : t.
\]

In other words, well-typed programs evaluate to values of the expected type; so in particular, they can't evaluate to **error**, which belongs to no type.
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.