What is a Type?

Canonical example: consider the expression of the form

\[
\text{(if big-ugly-expression}
\begin{array}{l}
(5\ 6)\\
a\text{-nice-value})
\end{array}
\]

which may be embedded deep inside a program. What type should a language translator (compiler/interpreter) assign to this expression? How will this expression behave? If \text{big-ugly-expression} is false, then the expression will produce a legal result. In this case, it is plausible for the type-checker to return the type of this value as the type of \text{a-nice-value}. But what if \text{big-ugly-expression} is true? Then the expression will generate a run-time error. Even worse, it is a statically detectable run-time error.

Type-checkers should assume all code fragments are meaningful (reachable in execution). Otherwise, why is the fragment included as part of the program? Hence, all type checkers will reject this expression – even if \text{big-ugly-expression} is obviously false (e.g., \text{big-ugly-expression} is the constant \text{false}).
Intuitive Assumptions in Type Checking

Idea 1: Types are names for sets of values.

Idea 2: The valid sets of ``input values'' for each program operation can be described in terms of types (most of the time).

Perhaps the second idea can be made completely true by imposing it as part of the contract for any operation. Example: `zip` in a functional language. In this case, contract should include check for equal lengths.

Idea 3: The application of program operations and the returning of values as the results of defined operations (methods, functions, procedures) induces constraints on program types.

The mathematical constraints are subtyping relationships:

(i) the type of an operation argument must be a subtype of its declared input type;

(ii) the type of the result returned by an operation must be a subtype of its declared result type.

In practice, most type systems force the type equality instead of type containment. It greatly simplifies the structure of the type systems.
Typed \( \lambda \)-Languages

The (simply) typed \( \lambda \)-calculus is the foundation of structural typing which is the overwhelmingly dominant typing discipline in functional (but not OO) languages.

Syntax:

\[
M :: = \lambda \ V : \tau \ . \ M \mid (M \ M) \mid V \mid C
\]

\[
\tau :: = D_1 \mid \ldots \mid D_n \mid \tau \rightarrow \tau
\]

where \( C \) is an optional set of constants (empty in the pure simply typed \( \lambda \)-calculus); \( D_1, \ldots, D_n \) are disjoint domains of primitive values (one mythical domain \( D \) for the pure simply typed \( \lambda \)-calculus) called \textit{primitive types} and \( V \) is the set of variable symbols. We will almost always work in extensions of the pure \( \lambda \)-calculus like Jam that include constant values and operations and primitive types. We will sometimes omit the typing for a variable introduced in a \( \lambda \)-expression.
Typing Rules for the (Simply) Typed \( \lambda \)-Calculus

A typing judgment has form: \( \Gamma \vdash M : \tau \)

where \( \Gamma \) (the Greek letter “capital gamma”) is called the type environment and consists of a set of typings of identifiers (each written in the form \( x : \tau \) where \( x \) is either a variable or a constant) and \( \tau \) is a type.

The inference rules for the pure (simply) typed \( \lambda \)-calculus are:

\[
\begin{align*}
\Gamma,\{x:\tau\} & \vdash x : \tau & \quad \text{(binding axiom)} \\
\Gamma,\{x:\sigma\} & \vdash M : \tau; \ x \text{ not free in } \Gamma \\
\hline
\Gamma & \vdash \lambda x. M : \sigma \to \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash M : \sigma \to \tau; \ \Gamma \vdash N : \sigma \\
\hline
\Gamma & \vdash (M \ N) : \tau
\end{align*}
\]
Typing Rules for Typed $\lambda$-Languages

- A *typing* for an expression $M$ given the type environment $\Gamma$ is an inference (proof) tree for a typing judgment of the form $\Gamma \vdash M : \tau$.

- Top level programs are typed with respect to a *base* type environment $\Gamma_0$ that contains the types of all program constants (including functions). For the *simply* typed $\lambda$-calculus, the base type environment is *empty*, because there are no constants in the pure simply typed $\lambda$-calculus.

- Typed $\lambda$-languages require exact matching between the input type of a function and the type of arguments to which it is applied. Why? There is no subtyping. Every value belongs to a unique type.

- Every constant $c$ in $C$ has a corresponding type in the base type environment $\Gamma_0$. The arity of each constant $c$ must match its type.

- Recall that the pure simply typed calculus *has no constants*. 