Preliminaries

- Why Types at All?
- Sets and Set Notation
- Meet the Universes

Why Types at All?

Sets and Set Notation

Meet the Universes

Va and **Ty**, Oh My!

> If you wish to make an apple pie from scratch, you must first invent the universe. — Carl Sagan

We're not doing anything as complex as baking here, but before we get started talking about types, we do need to invent a few **universes**. Well, OK, we don't *need* to, and most authors don't do so, but I personally find it helpful in sorting out my own thinking.

Basically, a universe is a set in the mathematical sense containing some things that we'll need to keep referring to over and over again. For example, for a given programming language, we'll say that the universe of values for that language is the set of all values that can be assigned to variables, evaluated from expressions, returned from functions and so forth. Similarly, we'll say that the universe of types is the set of all valid types in a given language.

By convention, we'll always denote universes by two-letter abbreviations in bold; **Va** for values, **Ty** for types, and so on.

Example (Python): In Python, **Va** contains values like `2`, `object()`, `"dadaesque"`, and `lambda x: x ** 2`. Similarly, **Ty** contains types like `int`, `str`, and `list`.

This notation pays off almost immediately. Indeed, one of the most basic things about type systems is that a type is a set of valid values of that type. This axiom is very easy to write out using universes and just a little bit of math!

Types Are Sets of Values
Ty $\subseteq \wp($**Va**$)$

Here, $\subseteq$ means "is a subset of or equal to," and $\wp$ means "the set of all subsets of," sometimes also known as the "power set." For example, $\wp(\{0, 1, 2\}) = \{\{\}, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{1, 2\}, \{0, 2\}, \{0, 1, 2\}\}$. These two mathy symbols interact in quite a useful way, too. If $X \subseteq Y$, then $X \in \wp(Y)$ and vice versa, where $\in$ means "is an element of."

We'll see more examples of this notation coming in handy throughout the rest of the book, but lets skip ahead a few chapters to see a spoiler for one more way this comes in handy. Suppose you're looking at a language that allows you to define new data types like structs or enums. For example,

in Rust you might write something like this to represent a pair of two integers:

```
struct PairOfInts {
    first: u64,
    second: u64
}
```

If we want this definition to generalize, we might use a *type parameter* instead:

```
struct PairOf<T> {
    first: T,
    second: T
}
```

Clearly, `PairOfInts` and `PairOf<u64>` are both types, and thus live in **Ty**, but what about `PairOf` on its own, before we fill in a specific type for `T`? We can think of `PairOf` as a mathematical function from **Ty** to **Ty**, written **Ty** → **Ty**. Put differently, `PairOf` is a recipe that, when given a type, tells you how to make a new type from the type that you passed in as `T`. This way of thinking about type parameters is common enough that many functional languages call things like `PairOf` a *type constructor* rather than a type in and of itself.

Tc: The Universe of Typeclasses

TODO: move this much, much later

If something is worth doing once, it's worth doing again, right? Having established all this machinery for powersets, elements, subsets, universes, and the like, can we use all that notational goodness to do something else fun? Let's go on and create a new universe!

Typeclasses Are Sets of Types
Tc ⊆ ℘(**Ty**)

Here, we've defined a new universe **Tc** as being the set of valid sets of types, known somewhat confusingly as *typeclasses*, *traits*, *concepts*, or *shapes* depending on what language you're working with. Basically, typeclasses let us constrain type parameters like that in `PairOf<T>` above. For example, suppose you have a function `log_to_console(x)`, and want to make sure that whatever type `x` is, it's at least printable.

Using typeclasses, you might write that as:

```
fn log_to_console<T: Debug>(x: T) {
    ...
}
```

In universes notation, we say that `log_to_console<u64>` is in **Va**, but `log_to_console` is in **Ty** → **Va**; that is, `log_to_console` is a mathematical function from types to values.

Aside: Kinds

Some languages and authors spell the universe of types as * instead. This is most common when discussing *kinds*, another way we can abstract from **Ty**. Basically, if a typeclass is a set of valid types that we can use to constrain type parameters, kinds go the other way and tell us how we can make new types in the first place.

In our examples above, `PairOfInts` and `PairOf<u64>` both have *kind* **Ty**, while `PairOf` has kind **Ty** → **Ty**. If we generalized `PairOf` to allow heterogeneous pairs like `PairOf<bool, u64>`, that would give us a kind of **Ty** → **Ty** → **Ty**. For most languages, that's as hairy as kinds get, a way of counting how many type parametes a given type constructor has: **Ty** has zero type parameters (it's just a type), **Ty** → **Ty** is the kind of type constructors that take a single type parameter, and so forth.

Some languages let you do some delightfully abstract things with kinds, though. Suppose your language has a type constructor `List` with kind **Ty** → **Ty**. Then if you want to pass `PairOf` itself to another type constructor, that would need to have kind (**Ty** → **Ty**) → **Ty**, an example of what's known as a *higher kinded type constructor*. We'll largely ignore higher kinds in this omnibus, as typeclasses are much *much* more common in imperative, object oriented languages than higher-kinded type constructors. Rather, we'll focus on using typeclasses to narrow basic kinds, like in the `Debug` → **Ty** example above, where `Debug` ∈ **Tc** (that is, `Debug` ⊆ **Ty**, as not all types are debuggable).

Glossary of Universes

If some of these universes don't make sense to you yet, don't worry. This is meant more as a reference that you can come back to as you need.

Name	Definition	Examples	
Va	Values	`3`, `"s"`, `lambda x: x`	
Ty	Types	`int`, `str`	**Ty** ⊆ $\wp$(**Va**)

Name	Definition	Examples	
Tc	Typeclasses	`Show`, `Debug`, `Serialize`	**Tc** $\subseteq \wp($**Ty**$)$
Fn	Functions	`lambda x: x`	
Di	Dimensions (compile-time constants)		
Dc	Design constraints (either type or dimension parameters)		**Ty** $\cup$ **Di**

Note that languages will differ substantially in the details of how each of these universes work; for example, in Python and Rust, **Fn** is a proper subset of **Va**, but this is not true of C.