Type Unions

Terminology: *types* are prescriptive. Some examples:

$$\begin{aligned}\text{unit} &:= ()\\ \text{bool} &:= \text{false} \mid \text{true}\\ \text{order} &:= \text{LESS} \mid \text{EQUAL} \mid \text{GREATER}\\ \text{nat} &:= \text{zero} \mid \text{succ}(\text{nat})\end{aligned}$$

Product types (Cartesian product, tuples): $\tau_1 \times \tau_2$.

What about "combining" two types? One idea: `Union`$\{\tau_1, \tau_2\}$. However:

```
const MightFail{T} = Union{T,String}
const IntMightFail = MightFail{Int}            Union{Int,String}
const StringMightFail = MightFail{String}      String (!)
```

Alternative: *sum types* ("tagged disjoint unions"): $\tau_1 + \tau_2$.

```
const MightFail{T} = Ok{T} | Error{String}
```

Benefits:

- More modular – cases are necessarily disjoint.
- Easier to develop efficient data structures, since sums are "concrete".
- Guaranteed *case exhaustivity* checking! Easy to refactor code.

Dynamic Typing and Dynamic Dispatch

In PL theory, types are checked without executing code (i.e. at "compile-time"). Therefore, "types" in Julia aren't types, formally. Instead, Julia has one type – `Any`, an *extensible sum*[1] – where each "type" is a *case*, tagging a value.

$$\texttt{Any} := \texttt{Int(int)} \mid \texttt{Float(float)} \mid \texttt{String(string)} \mid \ldots$$

- All Julia values (of type `Any`) are "tagged". Therefore, calling a function requires a case analysis ("method table lookup"), incurring a runtime performance penalty. **(!)** This explains why dynamic dispatch is avoided in critical-path code!
- Similarly, since each Julia "function" is a mutable registry covering domain `Any`, type piracy is inherently possible.

Conclusions:

- This is *less expressive* than having multiple types! There's no way to guarantee a given input has a given shape; instead, programmers must rely on knowledge of how the compiler optimizes in order to write performant code.
- However, `Any` can coexist with other types! For example, ML[2]-style languages support generative types alongside exn, an extensible sum type, like Julia's `Any`.
- Otherwise, gradual types[3,4] could be the answer?

[1] Practical Foundations for Programming Languages, Second Edition (Harper): Chapter 33
[2] The Definition of Standard ML (Milner, Tofte, Harper, MacQueen)
[3] What is Gradual Typing (Siek)
[4] Gradual Type Theory (New, Licata, Ahmed)

Abstract Types

What does it mean for `x::T`, with `T` abstract?

- `x::U`, for some concrete `U` with `U <: T`

What properties should `U` have if `U <: T`? Often, there is an (implicit) collection of functions which should be implemented for `U`. For example:

- If `U <: Number`, then `Base.:+(::U, ::U)::U`, `Base.:*(::U, ::U)::U`, etc. should be defined.
- If `U <: AbstractArray{Elt}`, then `Base.length(::U)::Int`, `Base.getindex(::U, ::Int)::Elt`, etc. should be defined.

Therefore, `U <: T` means that `U` implements an interface specified by `T`. In other languages, this feature is often called *typeclasses* or *traits*.[5]

Benefits:

- Traits can still form a hierarchy. For example, `trait Ordered{T} <: Eq{T}`.
- Instances can be automatically derived. For example, `implement Eq{Vector{T}} where Eq{T}`.
- Traits can *also* form more complex lattices ("multiple inheritance")!
- Similar to dynamic dispatch, traits simulate "overloading," allowing a given function to be implemented differently for various types. However, traits are a zero-cost abstraction[6], allowing the compiler to generate specialized code *before* runtime, leading to faster execution.
- No more runtime ambiguity errors!

[5] `radical-julia` (Grodin)

[6] Abstraction without overhead: traits in Rust (Turon)