

Recap

- **A type is a set of values**
 - "A function returns a value of type T" = The value returned from the function will always be in the set T.
- **Why types?**
 - Helps detect some mistakes
 - Added convenience because we can use the same name for similar operations on different types
 - No need for +int +float, etc.
 - Can say tune(object), and the right thing will happen whether object is a piano or a fish

Dimensions of the design space for type systems, continued

- **Type compatibility/type equivalence**
 - Question: When a value of type T1 is expected, is it ok to supply a value of T2?
 - Name compatibility
 - Definition: Two types T1 and T2 are name compatible if $T1 = T2$.
 - What about subtypes?
 - What are subtypes?
 - Pascal, Ada
 - Example: type age = 0 .. 120
 - Ada subtypes do not define a new type: Age is of type integer, even though its range is limited. Age is a subtype.
 - Different subtypes of a given type are considered to be compatible among themselves and with the supertype.
 - The following program will pass the Pascal type check:


```
program types;
type age = 0 .. 120;
var a : integer;
var my_age : age;
begin
  a := 122;
  my_age = a;
  writeln(my_age)
end.
```
 - Structural compatibility
 - Definition: Two types T1 and T2 are structurally compatible if
 - They are name compatible, or
 - They are defined by applying the same type constructor to structurally compatible types.
 - Example 1: C
 - typedef int customer_id;
 - typedef int car_id;
 - Can pass a customer_id wherever a product_id is required.
 - Example 2: C

- `typedef struct { char *name, int age } customer;`
- `typedef struct { char *make, int price } car;`
- Cannot pass a customer struct where a product struct is required.
- Example 2: Haskell
 - `type Customer = (String, Int)`
 - `type Car = (String, Int)`

```
beater :: Car
beater = ("Honda", 3000)
```

```
make :: Car -> String
make (make, price) = make
```

```
deadbeat :: Customer
deadbeat = ("John Doe", 42)
```

```
name :: Customer -> String
name (name, age) = name
```

```
name(deadbeat) = "John Doe"
name(beater) = "Honda"
```

- Can pass a Car wherever a Customer is required.
- **Type conversion**
 - When two types are not compatible, can one value be converted to a value of the other type?
 - Coercions: Automatic conversions
 - `float f = 3 + 4.0;`
 - Ada does not have automatic conversions
 - `i := INTEGER(4.0);`
 - Javascript converts pretty much everything
 - `"4" + 4 = 8`
 - If it looks like an integer, we'll treat it as one.
 - Ruby people call this "Duck typing" (If it quacks like a duck, it is one.)
 - Casting: explicit conversions
 - `int i = (int)4.0;`
 - Going "around" the type system (untyped semantics)
- **Monomorphic types vs. polymorphic types**
 - Most languages with object-oriented features use name compatibility instead of structure compatibility. Why? (Will become clear in this section.)
 - Simple, strong type system:
 - Every constant, variable, and routine has a declared type.
 - Every operation requires an operand of that exact type.
 - Such a system is called monomorphic (Greek: "single shape")
 - Every object belongs to one and only one type

- Polymorphism: A value has more than one type
 - Example:
 - An Integer is also a Rational; a Rational is also a Number; a Float is also Number
 - Can use Integers and Floats wherever Numbers are required
- Polymorphic features in most or all languages
 - Type compatibility and coercion move us away from string monomorphism
- Classification of polymorphic features:
 - **Polymorphism**
 - **Universal**
 - **Parametric**
 - Generic functions
 - Can be applied to values of any type
 - Function that reverses a list can work on lists of any type.


```
myreverse :: [a] -> [a]
myreverse = foldl (flip (:)) []
```
 - Because the `>=` works on any ordinal type, a function that computes the max of two arguments can work on arguments of any ordinal type.


```
mymax :: Ord t => (t, t) -> t
mymax (a, b)
  | a >= b = a
  | otherwise = b
```
 - Ada and C++ fake this with templates, but it's not the same; should be considered ad-hoc polymorphism.
 - **Inclusion**
 - Subtyping
 - Example: Java
 - Data structures like Vector work on any subtype of Object
 - They are, however, treated like Objects, and need to be casted back to their subtype when retrieved from the Vector.
 - **Ad hoc**
 - **Overloading**
 - The `+` operator in C can be used with integers or floats.
 - Purely a syntactic convenience: Bound to "int+" or "float+" depending on the context.
 - **Coercion**
 - `3 + 4.0` in C
 - 3 coerced into 3.0 before the appropriate overloaded `+` operator is applied.

Polymorphism and type checking in Haskell - Unification

- Polymorphic function definitions

- Could write the length function as follows:


```
length :: [String] -> Int
length [] = 0
length (a:x) = 1 + length x
```
- However, this would only for for lists of strings. Really, there is nothing in the code for length that is specific to strings, so it should work for lists of all types. How do we write that?


```
length :: [t] -> Int
```
- Here, t (starts with a lowercase letter) is a type variable: It can stand for any type.
- **Type of function compositions (from the Thompson book)**
 - Consider the following functions


```
f (a, c) = (a, ['a' .. c])
g (m, l) = m + length l
```
 - What should their types be?


```
f :: (t, Char) -> (t, [Char])
g :: (Int, [u]) -> Int
```
 - Now define a new function h that is the composition of the two


```
h = g . f          or          h y = g (f y)
```
 - For this to work, the output type of f must be compatible with the input type of g: (t, [Char]) must be compatible with (Int, [u]). Can we come up with instances of t and u so that this is true?


```
t = Int
u = Char
```

 - Both become (Int, [Char])
 - How do we find these instances in general?
 - Need to **unify** the types
 - Find the most general types that satisfy both sets of constraints
 - Example 2:
 - $e :: (t, [t])$
 - $h :: (u, v) \rightarrow v$
 - Want to apply h to e: (h e)
 - Must unify (t, [t]) with (u, v)
 - $e :: (t, [t])$
 - $h :: (t, [t]) \rightarrow [t]$
 - This is the most general unification because any other unification can be found by subsequent substitution.
 - Example 3:
 - Unify $[Int] \rightarrow [Int]$ and $t \rightarrow [u]$
 - Substitute [Int] for t and Int for u
 - Example 4:
 - Unify $[Int] \rightarrow [Int]$ and $t \rightarrow [t]$
 - Impossible, so the unification will fail.

Reading assignment for next time: Chapter 6 (Object-oriented languages)