Type Systems

A programming language is a form of communication (with the computer, among others).

The code we write expresses what/how we want the computer to compute.

Types are a way to express **constraints**.

type

a set of values, and (implicitly) a set of behaviours on those values

type system

the rules governing the use of types in a program, and how types affect the program semantics

```
(+ 1 “hi”)
```

```
(or #t (+ 1 “hi”))

True || (1 + “hi”)
```

dynamic typing

types are checked during the evaluation of a program

static typing

types are checked before the evaluation of a program (i.e., an operation on the abstract syntax tree)

From fighting the @(*#&$ compiler…

…to having a conversation with it.

A brief introduction to Haskell’s type system

Inspecting types: `:type`

Built-in types: `Int Bool [Char]/String`

Function types and automatic currying

```
(&&) :: Bool -> (Bool -> Bool)
```

Bool -> (Int -> (String -> Bool))

Declaring types

```
data <type-name> = <type-expr>
```

Struct-like types (product types)

$$\texttt{data}\ \underbrace{\texttt{Point}}_{\text{type name}} = \underbrace{\texttt{P}}_{\text{value constructor}}\ \texttt{Int Int}$$

Enum-like types (sum types)

```
data Day = Mo | Tu | We | Th | Fr | Sa | Su
```

Declaring types

```
data <type-name> = <type-expr>
```

algebraic data type

a type formed by any combination of sum and product types

```
data Shape = Circle Point Int
           | Rect Point Point
```

algebraic data types vs. inheritance

- no inheritance of methods
- no inheritance of attributes
- closed (can’t add new constructors)

```
data IntList = Empty
             | Cons Int IntList


data StringList = Empty
                | Cons String StringList


data BoolList = Empty
              | Cons Bool BoolList
```

Polymorphism

Greek: "poly" (many) and "morphe" (form)

generic (or **parametric**) **polymorphism**

the ability for an entity to behave in the same way regardless of “input” or “contained” type

Haskell lists are generically polymorphic (abstract version)

```
data List a = Empty
            | Cons a (List a)
```

type variable/parameter

Haskell lists are generically polymorphic (built-in version)

```
data [] a = []
          | (:) a ([] a)
```

Haskell lists are generically polymorphic (built-in version)

```
data [] a = []
          | (:) a ([] a)
```

a is a **type variable/parameter**

[] is a **type constructor** (“function” from types to types)

```haskell
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

$(++) :: [a] \rightarrow [a] \rightarrow [a]$

The empty list [ ] is a generically polymorphic value.

```
[True, False, True] ++ []
["david", "is", "cool"] ++ []
"david is cool" ++ []
```

During type-checking, the compiler determines how to instantiate the type variable so that the expression type-checks.

```
> :type []
[] :: [t]

> :type undefined
undefined :: a

> :type error
error :: [Char] -> a
```

Deriving constraints from generic polymorphism

If a function is generically polymorphic, there are many things it *can't* do.

```
f :: a -> a

f x = x
```

```
f :: a -> [a]

f x =
```

[]

[x]

[x, x]

[x, x, x]

[x, x, x, x, ..., x]

```
f :: [a] -> [a]

f x =
```

remove duplications
permutations

index-based

($)

```
f :: (a -> b) -> a -> b

f x y = x y
```

```
f :: a -> b

f x =
```

undefined

(not) Deriving constraints from function genericity

```
<T> T f(T x) {
  ...
}
```

(not) Deriving constraints from function genericity

```
<T> T f(T x) {
  return x;
}
```

(not) Deriving constraints from function genericity

```
<T> T f(T x) {
  blowUpUofT();
  return x;
}
```

One last example (“theorem for free”)

Given…

```
map :: (a -> b) -> [a] -> [b]

f  :: a -> b          (any function of this type)
xs :: [a]             (any list of this type)
r  :: [c] -> [c]      (any function of this type)
```

```
r (map f xs) == map f (r xs)
```

But what about `(+)`?

```
(+) :: Int -> Int -> Int
```

```
(+) :: a -> a -> a
```