Week 8: Type Systems

CSC324 Principles of Programming Languages

David Liu, Department of Computer Science

The problem of binary representation
**Denotational semantics**

The meaning of an expression as an abstract mathematical value.

Intuitively, “what this expression evaluates to.”

Denotational semantics allows us to envision all kinds of values (numbers, strings, lists, maps, etc.)

Computer hardware represents only 0’s and 1’s.
(+ 1 "hi")

Types
**type**

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

**type system** (of a programming language)

the rules governing the use of types in a program, and how those types affect language semantics
Programming languages are forms of communication.

Code communicates what or how we want to compute.

A type system expresses **constraints** on our computation.

When are type constraints checked?
dynamic typing

types are checked during program evaluation

(Python, JavaScript, Ruby, Racket)
**static typing**

Types are checked in the program AST

(C, Java, Haskell)

---

Dynamically-typed languages accept a larger set of expressible programs than statically-typed languages.

They are generally more flexible—for better or for worse.
WHEN YOU’RE COMPILING YOUR CODE

FOR THE FIRST TIME
Haskell's type system

Inspecting types in ghci: :type (or :t)
Built-in types: Bool, Integer, String/[Char]
Function types

\[
&& : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\]

-> is right-associative: Functions are automatically curried.

\[
&& : \text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})
\]

Product types (struct-like types)

\[
data \text{Point} = \text{P Integer Integer}
\]

Point is the type name.

P is a value constructor.

\[
\text{> : type P}
\]

P :: Integer -> Integer -> Point
Sum types (enum-like types)

```haskell
data Day = Monday | Tuesday | Wednesday
          | Thursday | Friday
```

Day is the **type name**.

Monday, Tuesday etc. are (nullary) **value constructors**.

```haskell
> :type Monday
Monday :: Day
```

---

**Algebraic data types**

An **algebraic data type** is a data type formed by any combination of sum and product types.

```haskell
data List = Empty
          | Cons Integer List

mapList :: (Integer -> Integer) -> List -> List
mapList _ Empty = Empty
mapList f (Cons x xs) = Cons (f x) (mapList f xs)
```
Algebraic data types vs. inheritance

We can view each value constructor in an algebraic data type as a “subclass” of the type, but:

- No “inheritance” of attributes/methods
- All types are closed to extension (“final”)

Polymorphism
generic (or parametric) polymorphism
the ability for an entity to behave the in same way regardless of “input” or “contained” type
Haskell lists are generically polymorphic.

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

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

*a* is a type variable/parameter.

List/[] is a type constructor (function from types to types)

Functions can be generically polymorphic.

```haskell
identity :: a -> a
identity x = x

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

What is the type of `[]`?

```haskell
> :type []
[[]] :: [a]
```

`[]` is a generically-polymorphic value!

---

We can use the empty list anywhere!

```haskell
[True, False, True] ++ []
["David", "is", "cool"] ++ []
"David is cool" ++ []
```
“Wildcards” in the type system

> :type undefined
undefined :: a

> :type error
error :: String -> a

Constraints help us reason about programs
If a function is generically polymorphic, there are many things it can’t do.

\[
f :: a \rightarrow a
f x = \ldots
\]

\[
f x = x
\]
Only removal, duplication, and reordering operations.
f :: (a -> b) -> a -> b
f x y = ...

Impossible! (Outside of undefined, error, etc.)

f :: a -> b
f x = ...

Impossible! (Outside of undefined, error, etc.)
What about Java?

```java
<T> T f(T x) {
    ...
}
```
**Theorems for free**

Let $T$ be any concrete type, and let $xs$ be any list of type $[T]$, and $f$ any function of the form $T \rightarrow \_$. Then for any generically polymorphic function with signature $\text{mix} :: [a] \rightarrow [a],$

$$\text{mix} \ (\text{map} \ f \ xs) = \text{map} \ f \ (\text{mix} \ xs)$$

But what about $(+)$?

$$(+) :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$$

$$(+) :: a \rightarrow a \rightarrow a$$
Type classes

**ad hoc polymorphism**

the ability for an entity to behave differently on different “input” or “contained” types
**type class** (in Haskell)

a set of types defined by an interface (set of functions) that the types must implement

```haskell
class Eq a where
  (==) :: a -> a -> Bool

class Show a where
  show :: a -> String
```

```haskell
> :type (==)
(==) :: Eq a => a -> a -> Bool

> :type show
show :: Eq => a -> String
```