CS4450/7450
Chapter 3: Types and Type Classes
Principles of Programming Languages

Dr. William Harrison

University of Missouri

September 12, 2016

Let the type be your guide

What is the type of `foo`?

```
foo st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Let the type be your guide

What is the type of `foo`?

```
foo st = [ c | c <- st, c `elem` ['A'..'Z']]
```

What does `foo` do?

```
foo :: [Char] -> [Char]
foo st = [ c | c <- st, c `elem` ['A'..'Z']]
```

Let the type be your guide

What is the type of `foo`?

```
foo st = [ c | c <- st, c `elem` ['A'..'Z']]
```

What does `foo` do?

```
foo :: [Char] -> [Char]
foo st = [ c | c <- st, c `elem` ['A'..'Z']]
```

```
ghci> foo "A Connecticut Yankee in King
    Arthur's Court"
"ACYKAC"
```

Type Systems

Haskell has "static types with inference"

- **Type Checking:** given an expression e and a type t, check whether $e :: t$. E.g.,

```
("hey", True) :: (String,Bool) -- Yes!
("hey", True) :: (String,Char) -- No!
```

Type Systems

Haskell has "static types with inference"

- **Type Checking:** given an expression e and a type t, check whether $e :: t$. E.g.,

```
("hey", True) :: (String,Bool) -- Yes!
("hey", True) :: (String,Char) -- No!
```

- **Type Inference:** given an expression e, compute its type t (if it exists). E.g.,

```
("hey",True)   ⇝  (String,Bool)
"hey" + 99     ⇝  error!
```

Type Systems

Haskell has "static types with inference"

- **Type Checking:** given an expression e and a type t, check whether $e :: t$. E.g.,

```
("hey", True) :: (String,Bool) -- Yes!
("hey", True) :: (String,Char) -- No!
```

- **Type Inference:** given an expression e, compute its type t (if it exists). E.g.,

```
("hey",True)   ⇝  (String,Bool)
"hey" + 99     ⇝  error!
```

- **Static Types.** a type system for which the types of expressions are known at *compile-time*. I.e., the type of every expression is known by inspecting its code—and not by running it.

Type Variables

Reintroducing what we called "parametric polymorphism"

The following type means that, for all types `a` and `b`, the function `fst` can be applied.

```
ghci> :t fst
fst :: (a, b) -> a
```

Type Instances

Given:

```
ghci> :t fst
fst :: (a, b) -> a
```

"Instances" of `(a,b) -> a` determine how `fst` can be applied:

```
(Int,Char) -> Int                    -- fst (99,'A')
([Char],Float) -> [Char]             -- fst ("Hey",3.14)
(Int -> Int,Bool) -> Int -> Int      -- fst (id,True)
                  ⋮
```

`fst`, `fst`, `fst` all refer to **the same code**.

Type Classes

The following is a *type constraint*:

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

It means that `(==)` can be applied only at types in the `Eq` class.

There are many predefined classes in Haskell, including `Ord`, `Show`, `Enum`, `Num`, etc.