Dependent types for practical use

Who is this for?

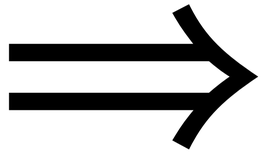
OOP

Generic/polymorphic types

Functional programming ?

About myself

- Software engineer
  - Mobile applications
  - Game development
  - Backend & microservices



- Research
  - Dependent types
  - Ergonomics & UX
  - Type theories

Dependent types

Need

Want

Avoid crashes

```
LinkedList<String> cars = new LinkedList<String>();
System.out.println(cars.get(3));
```

Runtime Error

```
Exception in thread "main" java.lang.IndexOutOfBoundsException:
Index: 3, Size: 0
```

Constrain programs

```
DateTimeFormatter format =
    DateTimeFormatter.ofPattern("dd.MM.yyyy");
```

OK

```
DateTimeFormatter format =
    DateTimeFormatter.ofPattern("");
```

Not OK

Avoid crashes

```
cars : Vect 0 String
cars = []

main : IO ()
main = printLn (index 3 cars)
```

Compiler Error:

```
Mismatch between: 0 and S ?len
```

Constrain programs

```
let myFormat = Main.formatString "yyyy/MM/dd"
```

OK

```
let myFormat = Main.formatString ""
```

Compiler Error: `Can't find an implementation for StringFormat ""`

Dependent types

Want?

Why would you want dependent types?

- Informative types
- Informative error messages
- Interactive editing
- Flexibility

Idris Demo

So, what *are* dependent types?

C	Terms	depend on	Terms
Java generics	Terms	depend on	Types
Haskell type families	Types	depend on	Types
Idris	Types	depend on	Terms

First class types

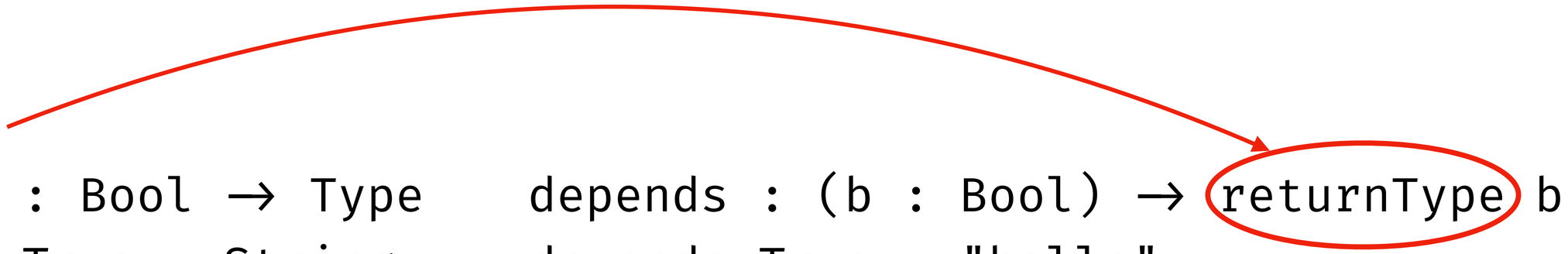
C	First class pointers
Java	First class objects
Haskell	First class functions
Idris	First class types

First-class types example

```
returnType : Bool → Type
returnType True = String
returnType False = Int

depends : (b : Bool) → returnType b
depends True = "hello"
depends False = 42
```

First-class types example



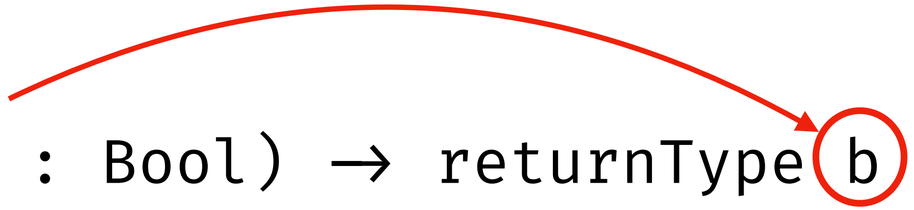
```
returnType : Bool → Type
returnType True = String
returnType False = Int

depends : (b : Bool) → returnType b
depends True = "hello"
depends False = 42
```

First-class types example

```
returnType : Bool → Type
returnType True = String
returnType False = Int
```

```
depends : (b : Bool) → returnType b
depends True = "hello"
depends False = 42
```



Heterogeneous lists

```
List<a>
```

```
["hello", 32, {"user": "John"}]
```

```
List<<String, Int, User>>
```

```
["hello", 32, {"user": "John"}]
```

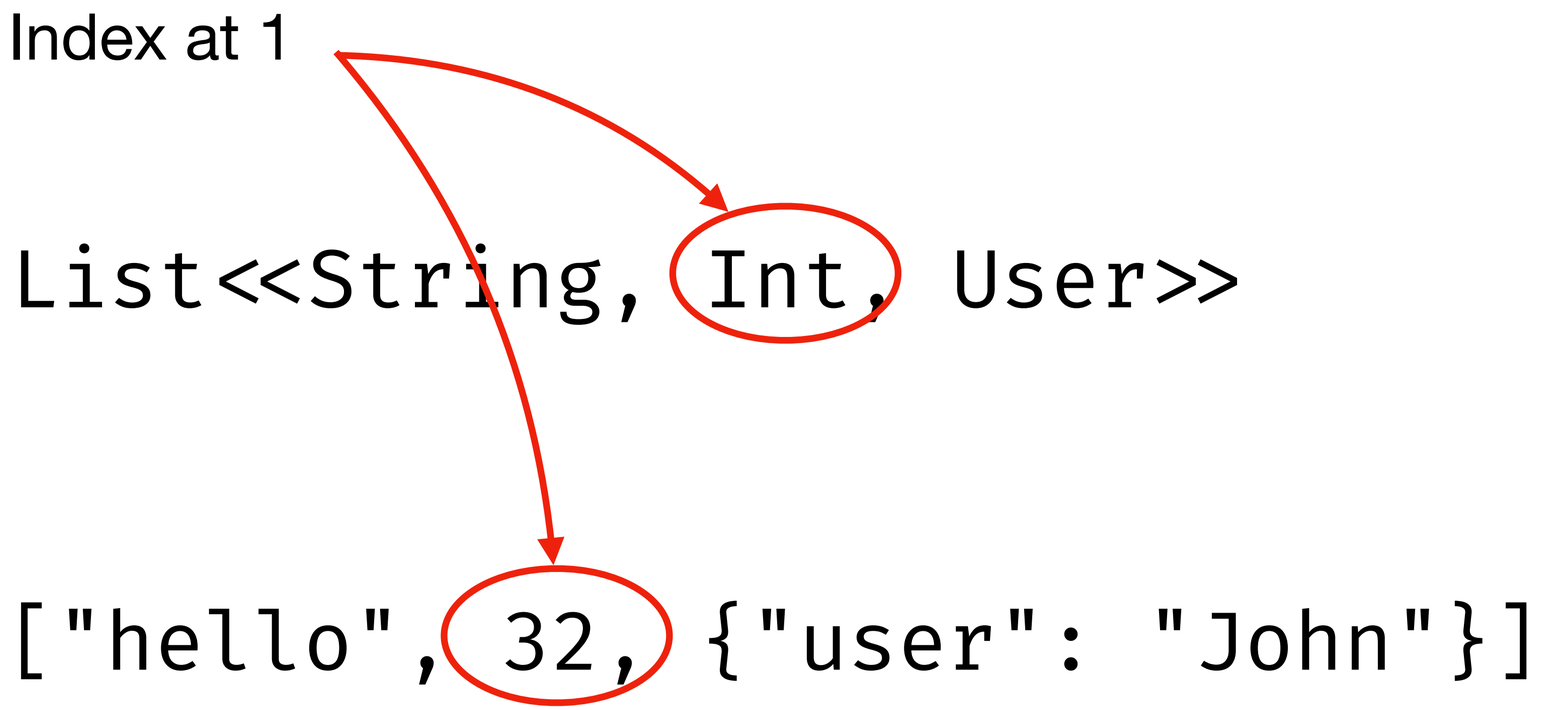
Indexing a heterogenous list

Index at 1

```
List<<String, Int, User>>
```

```
["hello", 32, {"user": "John"}]
```

Indexing a heterogenous list



```
list : List<<String, Int, User>>

list = ["hello", 32, {"user": "John"}]

indexAt 1 list : Int
```

Heterogenous Demo

"How do I return sometimes a value and sometimes an array depending on the data?"

- Someone on discord

```
type ValueReturn = Object | List<Object>
```

"How do I return sometimes a value and sometimes an array depending on the data?"

- Someone on discord

```
type ValueReturn = if isObj {
                     Object
                   } else {
                     List<Object>
                   }
```

JSON Demo

Libraries

Recombine

```
iotServer : DepServer ? ? ?
iotServer = "lights" / ("living" / Lens livingroomLight
                        &&&
                        "bedroom" / Lens bedroomLight
                        &&&
                        "kitchen" / Lens kitchenLight)
              &&&
              "boiler" / Lens serverBoilerLens


entireServer : DepServer ? ? ?
entireServer = "todo" / todoServer
          +&&&+ "calculator" / calculator
          +&&&+ "iot" / iotServer
```

Recombine

```
runServer Normal entireServer initialState
```

Collie

```
Turns : Command "TOP"
Turns = MkCommand
  { description = "A deeply nested
example"
  , subcommands = turns $ turns []
  , modifiers = []
  , arguments = lotsOf filePath
  } where
```

```
turns : Fields Command → Fields Command
turns cmds = [ "left"  ::= left cmds
             , "right" ::= right cmds
             ]
```

By Ohad Kammar & Guillaume Allais: https://github.com/ohad/collie

Collie

```
left : Fields Command → Command "left"
left cmds = MkCommand
  { description = "Took a left turn"
  , subcommands = cmds
  , modifiers   = []
  , arguments   = none
  }
```

```
right : Fields Command → Command "right"
 right cmds = MkCommand
   { description = "Took a right turn"
   , subcommands = cmds
   , modifiers   = []
   , arguments   = none
   }
```

Collie

```
handle : Turns ⇝ IO ()

handle
 = [ (\ args ⇒ let files = fromMaybe Prelude.Nil args.arguments in
                  putStrLn "Received the files: \{show files}")
   , "right" ::= [ const $ putStrLn "Took a right turn"
                 , "left"  ::= [ const $ putStrLn "Back to the start (rl)" ]
                  , "right" ::= [ const $ putStrLn "Half turn, rightwise" ]
                  ]
   , "left"  ::= [ const $ putStrLn "Took a left turn"
                  , "right" ::= [ const $ putStrLn "Back to the start (lr)" ]
                  , "left"  ::= [ const $ putStrLn "Half turn, leftwise" ]
                 ]
   ]
```

Can't we do this with meta-prgramming?

Meta-programming
is
programming

What we haven't seen

- Linear types and quantities
- Performance and low-level programming
- Protocols

End

Thank you