Dependent types for practical use
Who is this for?

- OOP
- Type theory

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

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

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

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

OK

Not OK
Avoid crashes

cars : Vect 0 String

cars = []

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

Compiler Error:
Mismatch between: 0 and 3 ?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?
<table>
<thead>
<tr>
<th></th>
<th>Terms</th>
<th>depend on</th>
<th>Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java generics</td>
<td>Terms</td>
<td>depend on</td>
<td>Types</td>
</tr>
<tr>
<td>Haskell type families</td>
<td>Types</td>
<td>depend on</td>
<td>Types</td>
</tr>
<tr>
<td>Idris</td>
<td>Types</td>
<td>depend on</td>
<td>Terms</td>
</tr>
</tbody>
</table>
First class types
<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>Java</th>
<th>Haskell</th>
<th>Idris</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>First class pointers</td>
<td>First class objects</td>
<td>First class functions</td>
<td>First class types</td>
</tr>
</tbody>
</table>
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
depends : (b : Bool) → returnType b

returnType True = String
depends True = "hello"

returnType False = Int
depends False = 42
Heterogeneous lists
List<
["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<<String, Int, User>>

["hello", 32, {"user": "John"}]
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>
}
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
  ]
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 \to\ IO ()

handle
  = [ (\ args \Rightarrow 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-programming?
Meta-programming is programming
What we haven't seen

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

Thank you