Types to the rescue!

Sebastian Rysztuń
October 24, 2018

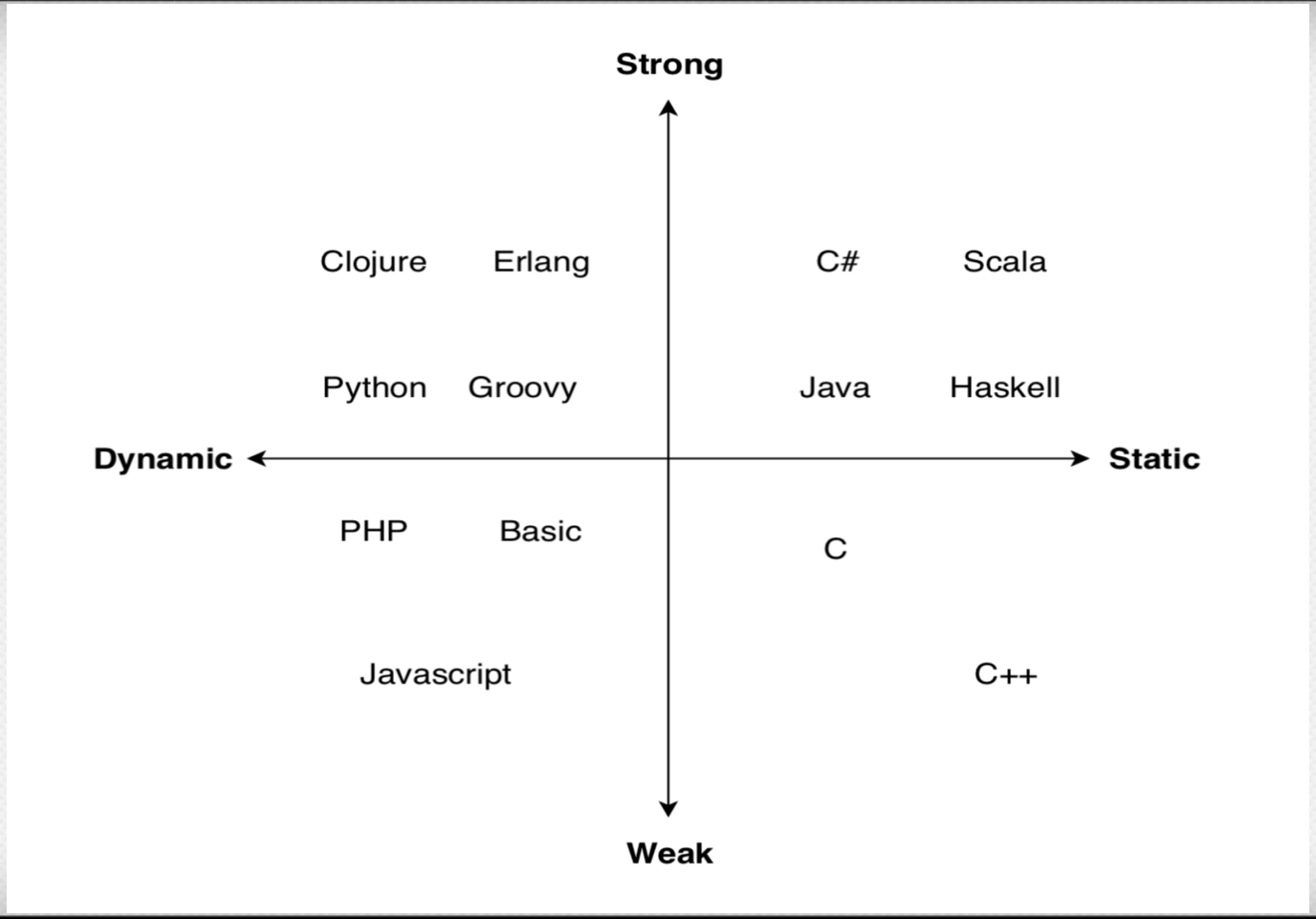
Agenda

- Types
  - What
  - Why
  - How

Definition

- type is an attribute of data which tells the compiler or interpreter how the programmer intends to use the data
- we can group languages based on how they treat types

Static/Dynamic/Strong/Weak



Interpreter vs Compiler

Interpreter

- convert code into machine code line by line when the program is run
- all errors in runtime

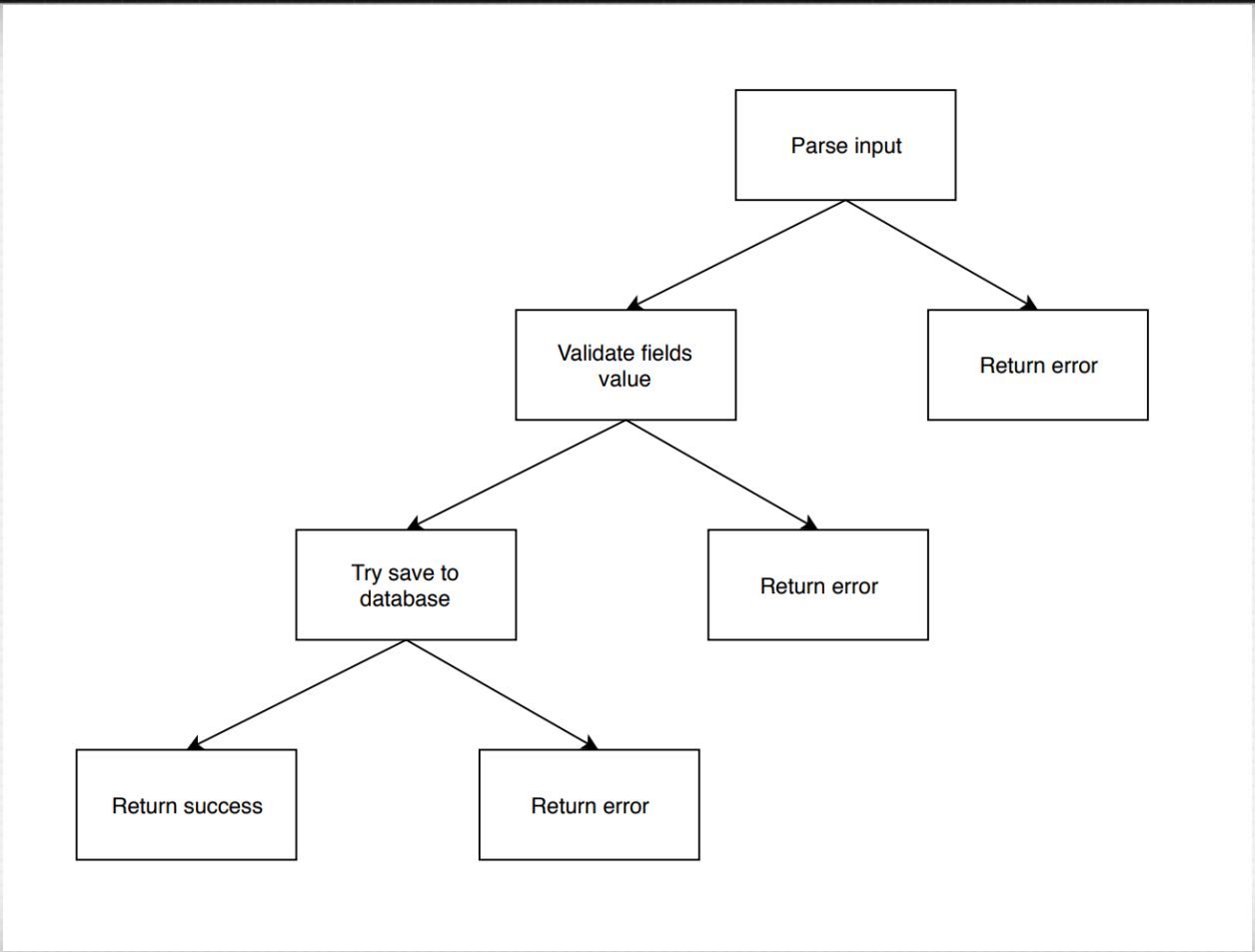
Compiler

- code will be parsed and analyzed
- If no error -> conversion of source code to machine code
- many errors are caught before runtime

Types as flow control

- Cyclomatic complexity
  - number of paths through a particular piece of code
  - More places where your program can end execution or it branches off you have, it is harder to understand it a.k.a. sphagetti code

Types as flow control



Types as flow control

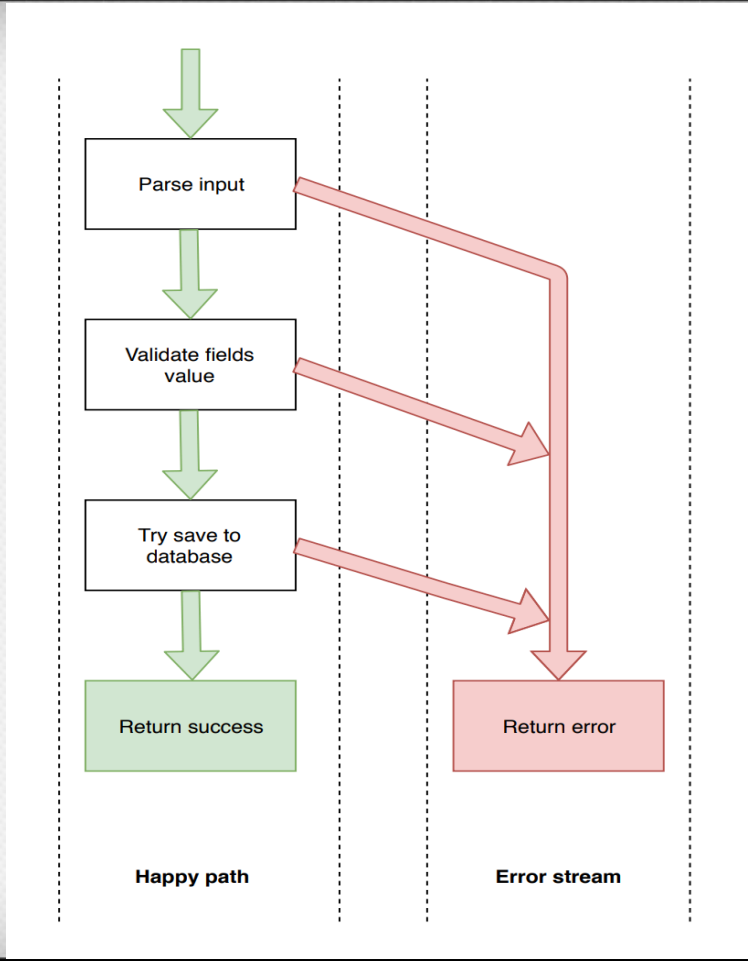
```
def createUser(request: Request): Result = {
  var userToCreate;
  try {
    userToCreate = parseInput(request)
  } catch {
    return "Couldn't parse input"
  }
  val isValid = validateUser(userToCreate)
  if(!isValid) {
    return "User is invalid"
  }

  var result;
  try {
    result = saveUser(userToCreate)
  } catch {
    return "Couldn't save user to database"
  }
  if(result) {
    return "User created successfully"
  } else {
    return "User already exist"
  }
}
```

Types as flow control

- Railway pattern
  - functional approach to error handling
  - two tracks
    - happy path
    - error stream

Types as flow control



Types as flow control

```
def createUser(request: Request): Result = {
   val userCreationF =  for {
     userToCreate <- Future.fromTry(parseInput(request))
     _ <- Future.fromTry(validateUser(userToCreate))
     result <- saveUser(userToCreate)
   } yield result

   userCreationF.map {
      case Success(userSavedToDB) if userSavedToDB == true => "User created successfully"
      case Success(userAlreadyExist) => "User already exist"
      case Failure(ex) if ex.isInstanceOf[ParseErrorException] => "Couldn't parse input"
      case Failure(ex) if ex.isInstanceOf[ValidationException] => "User is invalid"
      case Failure(ex) if ex.isInstanceOf[DbException] => "Couldn't save user to database"
    }
 }
```

Types as flow control

First version of code

Types as domain description

- String with name is something other that String with email
- Too many primitive types – easier to make mistake
- Data types/Value classes/Opaque data type
  - https://docs.scala-lang.org/overviews/core/value-classes.html

Types as flow control

Second version of code

Types as documentation

- Our type can describe precisely what is possible inside function
- Compiler will keep an eye on us to keep it updated and truthfully

Types as documentation

We can return null? Use **Option**

```
sealed abstract class Option[+A]
final case class Some[+A] extends Option[A]
case object None extends Option[Nothing]
```

Types as documentation

We can throw exception? Use **Try**

```
sealed abstract class Try[+T]
final case class Failure[+T](exception: Throwable) extends Try[T]
final case class Success[+T](value: T) extends Try[T]
```

Types as documentation

We want to return one from the two types?

Use **Either**

```
sealed abstract class Either[+A, +B]
Left[+A](value: A)
Right[+B](value: B)
e.g. Either[Throwable, T] ~= Try[T]
```

Types as documentation

Third version of code

Types as documentation

- What can be next?
  - **Throwing exception is heavy operation, we can get rid of Stack trace in many cases and use** https://www.scala-lang.org/api/2.13.1/scala/util/control/NoStackTrace.html
  - **Why we need throw exceptions? We can use Either instead**
  - **Separate declarative business logic from side effects and infrastracture (imperative code) e.g. Using Final Tagless approach to model our application**

Types as documentation

- We can go even further beyond, e.g. encode definition of our API as types
  - Endpoints: **https://github.com/julienrf/endpoints** - defining communication protocols over HTTP between applications
  - Tapir: **https://github.com/softwaremill/tapir** - describe HTTP API endpoints as immutable Scala values
- Or encode mathematical ML calculation on type level:
  - **https://www.youtube.com/watch?v=BfaBeT0pRe0**

Summary 1/2

- the more things checked by compiler, the less things we need to check
- richer and better IDE support during developement
- more things compiler must do, more time compilation will take

Summary 2/2

- "primitive types are not the proper types”
- try avoid branching to make your code easier to understand – railway pattern
- types are your best documentation
- types are not for free, but ... are an investment that pays off in the long run

Questions?

Thank you!