Staying Pure in the Real World

@eddsteel September 2013

hootsuite
Social Media Management

Imperative code **executes statements** to produce **effects**.

Functional code **evaluates expressions** to produce **values**.

Functions are values too.

Pure Functions

"referentially transparent", "deterministic", "nullipotent"

==

data in/ data out

Pure Functions

Without context or side-effects, pure functions are

- easier to refactor
- easier to test
- easier to parallelise
- easier to understand

than impure functions

Pure Functions

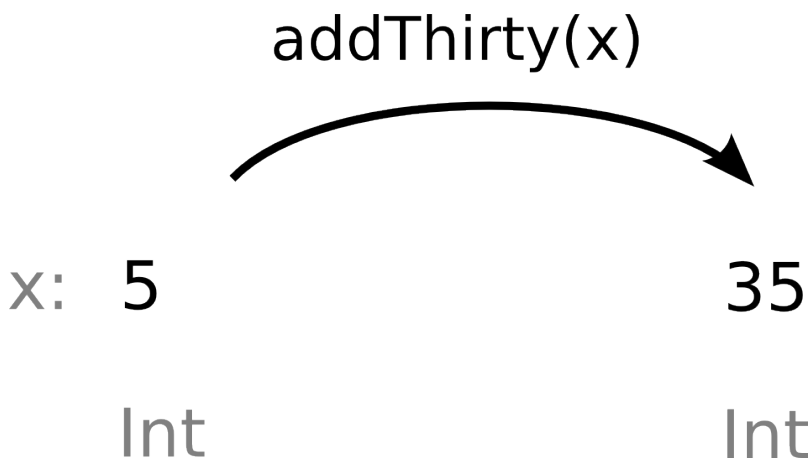
Can be written in

- haskell
- scala
- javascript
- PHP

(ascending order of difficulty)

addThirty

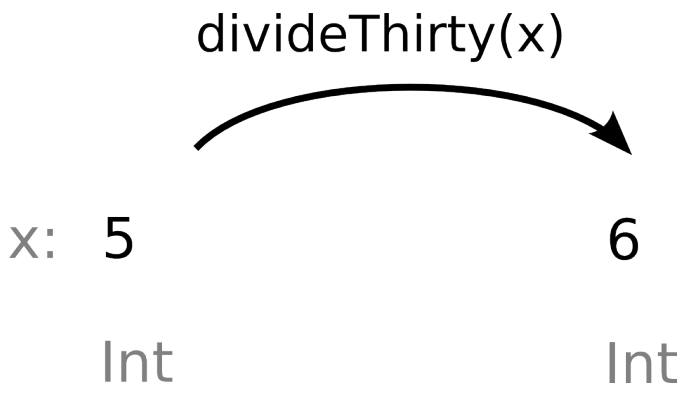
```
def addThirty(x: Int): Int = x + 30
```



Pure

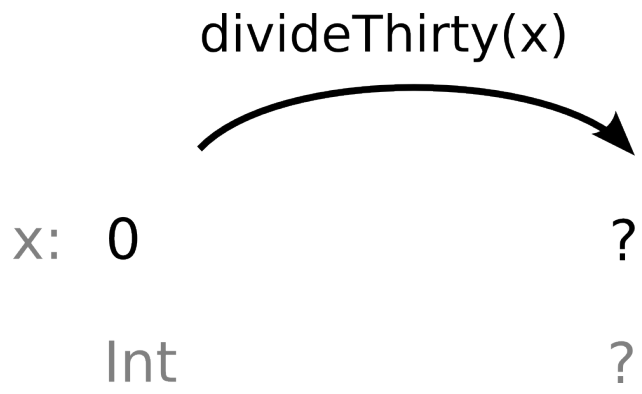
divideThirty

```
def divideThirty(x: Int): Int =
  if(x == 0) throw new ArithmeticException
  else (30 / x).toInt
```

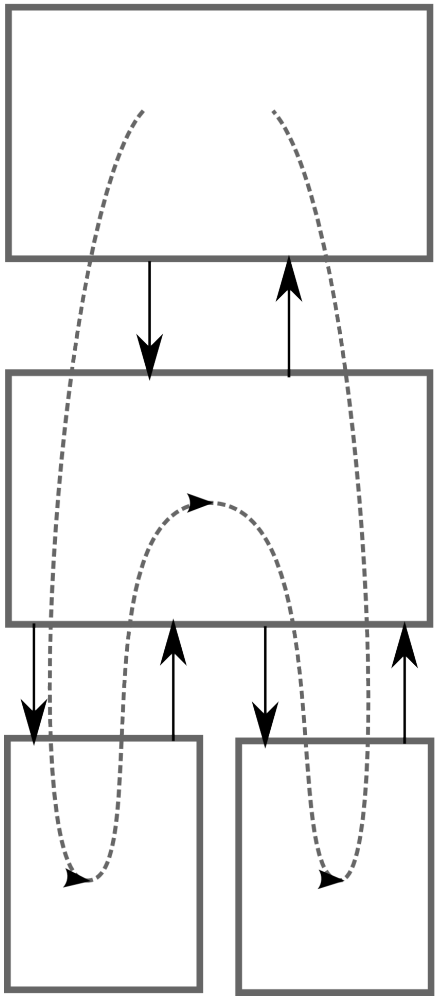


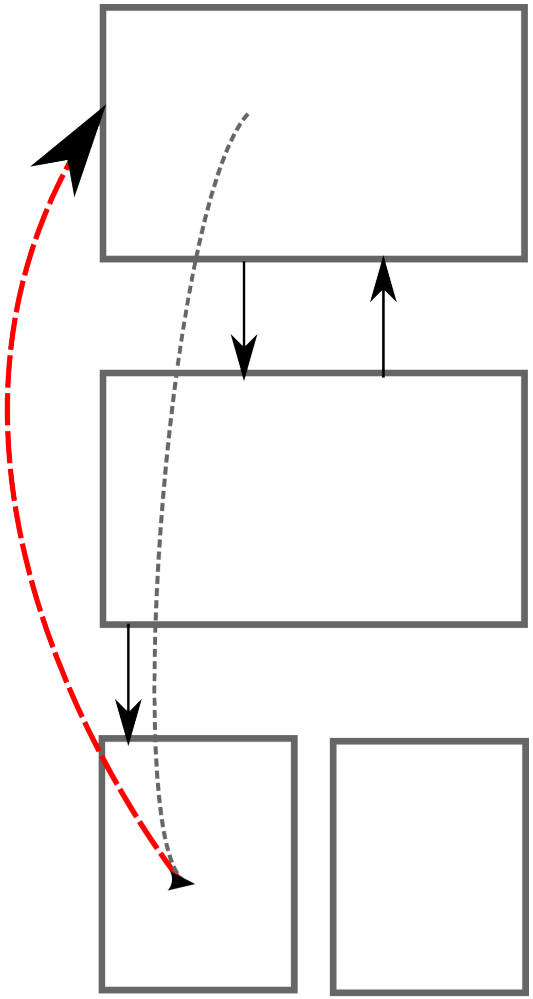
divideThirty

```
def divideThirty(x: Int): Int =
  if(x == 0) throw new ArithmeticException
  else (30 / x).toInt
```



The real world





Impure

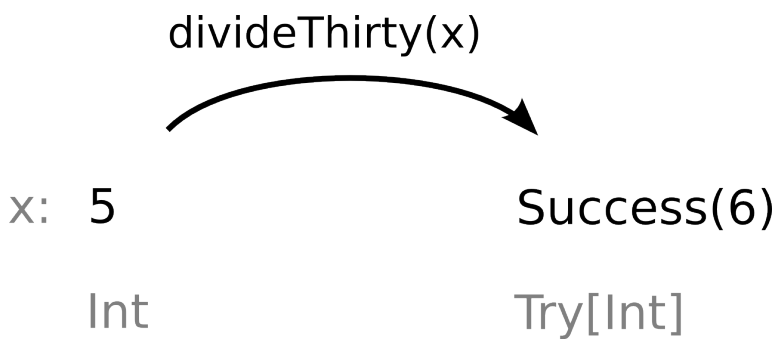
WWUD?

Can we express this control flow as a value?

That would maintain purity, and its benefits.

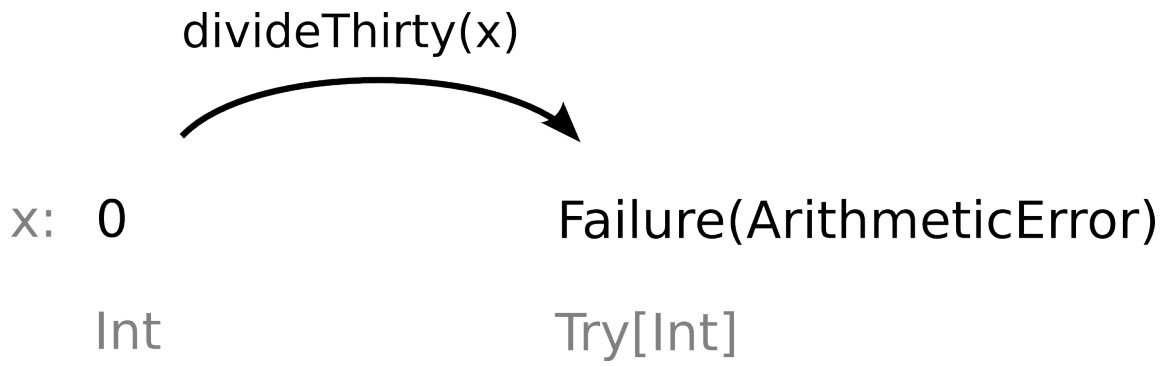
divideThirty

```
import scala.util.{Try, Success, Failure}

def divideThirty(x: Int): Try[Int] =
  if(x == 0) Failure(new ArithmeticException)
  else Success((30 / x).toInt)
```



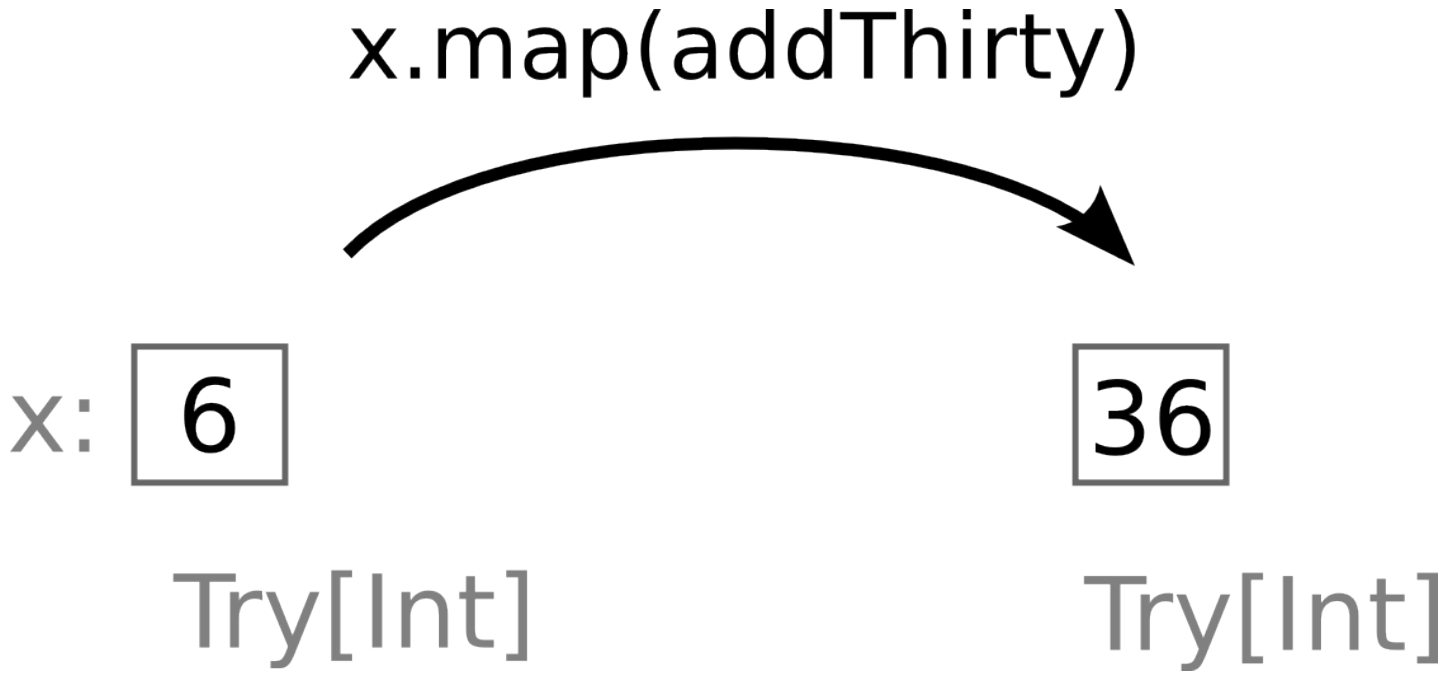
divideThirty

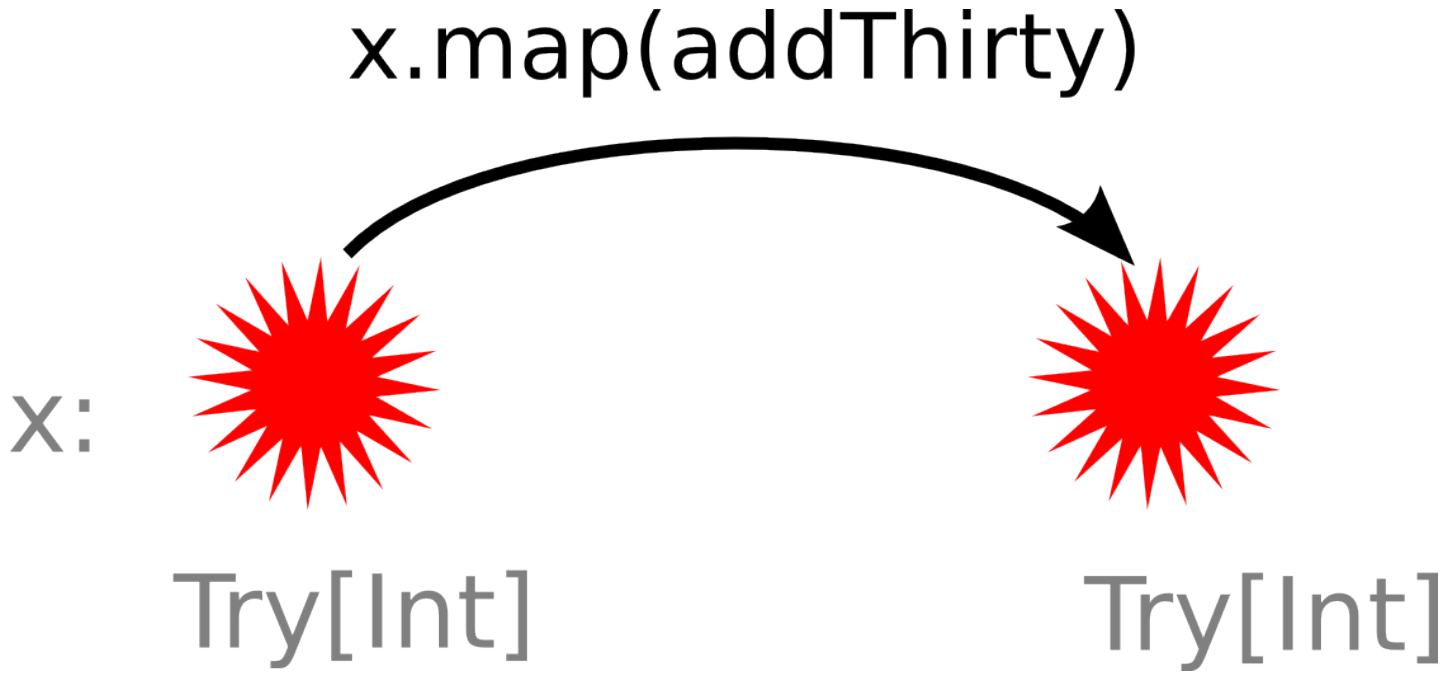
```
import scala.util.{Try, Success, Failure}

def divideThirty(x: Int): Try[Int] =
  if(x == 0) Failure(new ArithmeticException)
  else Success((30 / x).toInt)
```



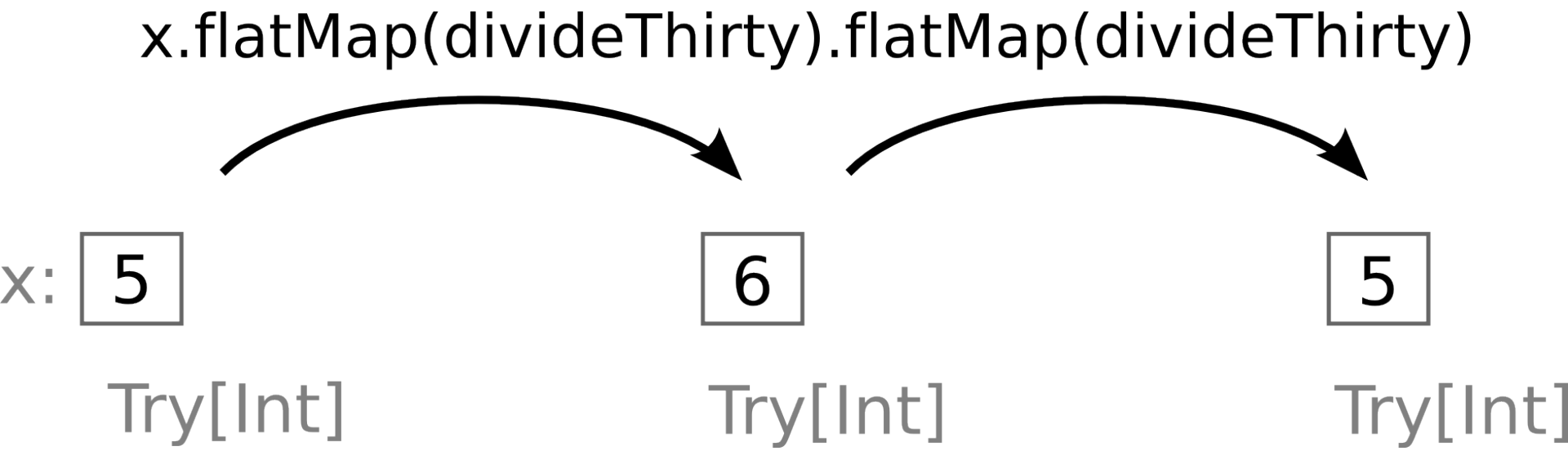
Pure

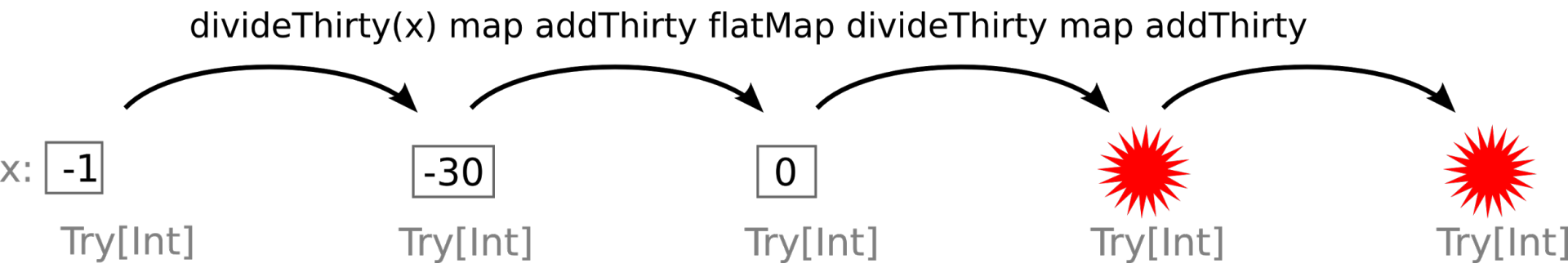
So now I have to rewrite all my functions to accept Try [Int] instead of Int?





But I could chain functions easily, when everything was an Int.





This is kind of useful

```
import scala.util.{Try, Success, Failure}

def processRequest(in: Request): Try[Response] =
  for {
     validated <- validateRequest(in)
     parsed <- parseJson(validated.jsonContent)
     processed <- process(parsed)
     result <- summarise(processed)
     json <- createJson(result)
     out <- Response(json)
  } yield out
```

(monad)

This is kind of useful

- Try[T] potential failures
- Future[T] asynchronous results
- Option[T] potential nulls
- List[T] collections

Your pure functions can work with all of them, without modification.

Have questions?