Free Yourself With Constraints

Balaji Sivaraman (@balajisivaraman)

ThoughtWorks

November 3, 2016

About Me

- Primarily worked on Java/Spring stack
- Bitten by the FP bug in 2012 thanks to Scala
- Currently have an affinity for Functional Programming/Type Theory
- Occasionally dabble with Haskell, Idris, Purescript etc.

Agenda

- Introduction [1] [2]
- Ground Rules
- Constraints Already Imposed By FP
- Why They Aren't Enough?
- What Needs To Be Done?
- Conclusion
- Questions?

Ground Rules

Properties of programs considered

- Reasoning
- Readability
- Reusability

Properties of programs not considered

- Performance

Constraints Already Imposed By FP

- Pure Functions - Code should be series of function calls instead of instruction executions
- Immutability - Functions cannot modify global variables, throw errors etc.

Why Are They Necessary?

- Referential Transparency - Functions must produce same output given same input values
- Equational Reasoning - Function calls can be replaced by the values they compute to understand code more easily
- Ease of Refactoring

Let's take a simple function

Can you guess what this function does just by looking at it?

```
function add(var a, var b) {
  //...
}
```

How did we do it?

- Name Inference
- Number of Arguments
- General Experience and Common Sense

But wait...

```
function add(var a, var b) {
  return a.toString() + b.toString();
}
```

What Happened There?

- Necessity to look under the covers
- Other forms of reasoning can and will fail at some point
- Not enough information for the reader
- Too much power to the implementor
- Mentally constrained input values

Types to the Rescue

Can you guess what this function does just by looking at it?

```
def add(a: Int, b: Int): Int = {
  //...
}
```

Much Better

- Function takes 2 Integers
- And returns an Integer
- Mental constraints have been made explicit
- So it is safe to assume it does what it says?

But Wait...

```scala
def add(a: Int, b: Int): Int = {
  a - b
}
```

Types Alone Aren't Enough

- Still too much power to the implementor
- Still not enough information for the reader

Parametricity

Philip Wadler [3] writes:

Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.

The purpose of this paper is to explain the trick.

A Simple Parametrically Polymorphic Function

Can you guess what this function does just by looking at it?

```
def doSomething[A](a: A, b: A): A = {
  //...
}
```

How did we do it?

- Power of parametric polymorphism
- Very little power for the implementor
- Lot of info available to the reader from definition

Another (familiar?) Parametrically Polymorphic Function

Can you guess what this function does just by looking at it?

```scala
def doSomethingElse[A, B](a: List[A])(f: A => B): List[B] =
{
  //...
}
```

But wait...

```
def doSomethingElse[A, B](a: List[A])(f: A => B): List[B] =
{
  List(f(a.head))
}
```

What Happened There?

- Parametric polymorphism alone isn't enough
- Implementor sometimes still has too much info to work with
- Reader sometimes still has to look under the covers to feel really safe

Functor Typeclass

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

- Here F[_] denotes a higher-kinded type
- Think of them as analogous to higher-order functions at the type level, i.e. they take a type themselves

Why Type Classes and HKTs?

- Instances are created for the Functor typeclass for each `F[_]`, i.e. List, Option etc.
- These instances are distinct entities, completely separate from the Typeclass itself
- Just like HOFs, HKTs give you the ability to abstract over the type constructor itself, which has numerous benefits in practice

Different Instances

```
implicit object ListFunctor extends Functor[List] {
  def map[A, B](fa: List[A])(f: A => B): List[B] = fa.map(f)
}

implicit object OptionFunctor extends Functor[Option] {
  def map[A, B](fa: Option[A])(f: A => B): Option[B] = fa.map(f)
}
```

How is this better?

- Only one instance of each class in our entire application
- Typeclasses are a purely compile-time construct, search and verification of instances is at compile-time
- Functor laws as property tests (QuickCheck, ScalaCheck etc.)
- Created instances have to adhere to Functor laws

Reusability

```scala
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]

  def lift[A, B](f: A => B): F[A] => F[B] = ???
}
```

Reusability

```
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]

  def lift[A, B](f: A => B): F[A] => F[B] = fa => map(fa)(f)
}
```

Reusability

Code

Reusability

- Many typeclasses already available (Scalaz, Cats) - Functor, Applicative, Monad, Traverse etc
- Easy to define instances for our classes or new typeclasses themselves
- Typeclasses, by definition, are parametrically polymorphic giving them many compile time benefits

Conclusion

- Code we write is always constrained, whether implicitly or explicitly
- Making them explicit leads to better ways to reason about and understand our code
- The more constrained our code, the more easy to read and refactor it is
- Parametricity is the bare minimum constraint we should work with
- Prefer typeclasses over inheritance if the language allows it
- Higher kinded types enable a lot of reuse
- Always reach for the most constrained abstraction, leaving the implementor with fewer choices

References

Tony Morris. *YOW! West 2016 Tony Morris - Parametricity, Functional Programming, Types*. Youtube. 2016. URL: https://www.youtube.com/watch?v=qBvFsA3dglk.

Kris Nuttycombe. *LambdaConf 2015 - Parametricity The Essence of Information Hiding Kris Nuttycombe*. Youtube. 2015. URL: https://www.youtube.com/watch?v=v6de5KWFY6M.

Philip Wadler. "Theorems for free!" In: *FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE*. ACM Press, 1989, pp. 347–359.

Thank you!

Slides source available at: https://github.com/balajisivaraman/constraints