Why is so much of FP about Types instead of Functions?

Donovan Crichton

October 2023

Preliminaries

- Slides and Examples available at:
  https://github.com/donovancrichton/Talks
- This talk: BFPG/WhyIsFPAboutTypes

About Me

NATURAM PRIMUM COGNOSCERE RERUM

Australian National University

- PhD Candidate
- Computing Foundations
- School of Computing

Griffith UNIVERSITY

- Visiting Scholar
- Trusted Systems Lab
- IIIS

ASD AUSTRALIAN SIGNALS DIRECTORATE

- ASD Co-Lab Scholar

What is a Function?

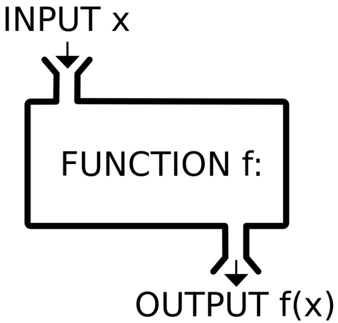

Figure: A "black-box" depiction of a function[1]

[1]Source https://en.wikipedia.org/wiki/Function_(mathematics)#/media/File:Function_machine2.svg

What can we do with a function?

- Give it its argument.

What can we do with a function?

- Give it its argument.
- Look at its result.

What can we do with a function?

- Give it its argument.
- Look at its result.
- ...

What can we do with a function?

- Give it its argument.
- Look at its result.
- ...
- ...

What can we do with a function?

- Give it its argument.
- Look at its result.
- ...
- ...
- Maybe we need to think about this some more...

What is a function, Georg Cantor?

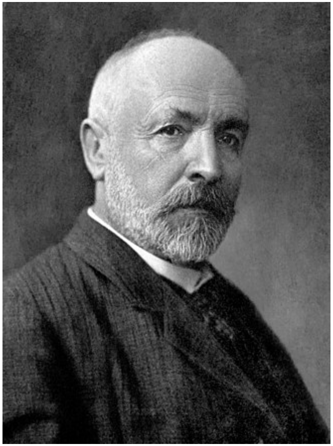

Figure: Georg Cantor (1845 - 1918)[1]

[1]Source `https://en.wikipedia.org/wiki/Georg_Cantor#/media/File:Georg_Cantor_(Portr%C3%A4t).jpg`

Sets

- Sets are concerned with collections of discrete mathematical objects.
  e.g $A = \{1, 2, 3, ...\}$.

Sets

- Sets are concerned with collections of discrete mathematical objects.
  e.g $A = \{1, 2, 3, ...\}$.
- The order of elements does not matter.

Sets

- Sets are concerned with collections of discrete mathematical objects.
  e.g $A = \{1, 2, 3, ...\}$.
- The order of elements does not matter.
- Each element my appear only once.

Sets

- Sets are concerned with collections of discrete mathematical objects.
  e.g $A = \{1, 2, 3, ...\}$.
- The order of elements does not matter.
- Each element my appear only once.
- Given an set X and an index i we may chose $x_i \in X$. Think of this like a function $f : (Set(X), \mathbb{N}) \rightarrow X$.
  e.g $f(A, 2) = 2$.

Sets

- Sets are concerned with collections of discrete mathematical objects.
  e.g $A = \{1, 2, 3, ...\}$.
- The order of elements does not matter.
- Each element my appear only once.
- Given an set X and an index i we may chose $x_i \in X$. Think of this like a function $f : (Set(X), \mathbb{N}) \to X$.
  e.g $f(A, 2) = 2$.
- See [Halmos, 1960] for more!

Relations

- A relation is a set of pairs and specifically is useful for denoting a one-to-many relationship.

Relations

- A relation is a set of pairs and specifically is useful for denoting a one-to-many relationship.
- i.e $1 < 2$, $1 < 3$, $1 < 4$ and so on.

Relations

- A relation is a set of pairs and specifically is useful for denoting a one-to-many relationship.
- i.e $1 < 2$, $1 < 3$, $1 < 4$ and so on.
- To the WHITEBOARD[1] for a demo.

[1] I sadly did not have time to typeset this diagram for tonight's talk, but I will upload the completed diagram to GitHub before the end of the week.

Relations

- A relation is a set of pairs and specifically is useful for denoting a one-to-many relationship.
- i.e $1 < 2$, $1 < 3$, $1 < 4$ and so on.
- To the WHITEBOARD[1] for a demo.

Question for the pub!

Why are functions the default 'building-blocks' in most programming languages? Why not relations? All functions are relations, but not all relations are functions.

[1] I sadly did not have time to typeset this diagram for tonight's talk, but I will upload the completed diagram to GitHub before the end of the week.

Functions

- Functions can have a one-to-one, or many-to-one relationship, but never one-to-many.

Functions

- Functions can have a one-to-one, or many-to-one relationship, but never one-to-many.
- There are special classes of *injective* functions, where every input gives only one output, and all outputs are distinct from one-another.

Functions

- Functions can have a one-to-one, or many-to-one relationship, but never one-to-many.
- There are special classes of *injective* functions, where every input gives only one output, and all outputs are distinct from one-another.
- There are special classes of *surjective* functions, where the entire output set can be reached through one-or-more inputs.

Functions

- Functions can have a one-to-one, or many-to-one relationship, but never one-to-many.
- There are special classes of *injective* functions, where every input gives only one output, and all outputs are distinct from one-another.
- There are special classes of *surjective* functions, where the entire output set can be reached through one-or-more inputs.
- Functions that are surjective and injective are said to be *bijective*.

Functions

- Functions can have a one-to-one, or many-to-one relationship, but never one-to-many.
- There are special classes of *injective* functions, where every input gives only one output, and all outputs are distinct from one-another.
- There are special classes of *surjective* functions, where the entire output set can be reached through one-or-more inputs.
- Functions that are surjective and injective are said to be *bijective*.
- To the WHITEBOARD[2] for a demo.

[2] As before, I will typeset the accompanying diagrams and upload online.

What can we do with set-theoretic functions?

- We can now easily model *composition*.

What can we do with set-theoretic functions?

- We can now easily model *composition*.
- Let's see this on the WHITEBOARD[3].

[3]actual diagrams coming soon to the online version, I promise!

What can we do with set-theoretic functions?

- We can now easily model *composition*.
- Let's see this on the WHITEBOARD[3].
- ...

[3] actual diagrams coming soon to the online version, I promise!

What can we do with set-theoretic functions?

- We can now easily model *composition*.
- Let's see this on the WHITEBOARD[3].
- ...
- ...

[3] actual diagrams coming soon to the online version, I promise!

What can we do with set-theoretic functions?

- We can now easily model *composition*.
- Let's see this on the WHITEBOARD[3].
- ...
- ...
- Isn't this just giving a function it's argument and checking the result though?

[3] actual diagrams coming soon to the online version, I promise!

What can we do with set-theoretic functions?

- We can now easily model *composition*.
- Let's see this on the WHITEBOARD[3].
- ...
- ...
- Isn't this just giving a function it's argument and checking the result though?
- Let's ask someone else...

[3] actual diagrams coming soon to the online version, I promise!

What can we do with functions, Alonzo Church?

Figure: Alonzo Church (1903 - 1995)[1]

[1] Source https://en.wikipedia.org/wiki/Alonzo_Church#/media/File:Alonzo_Church.jpg

The Untyped Lambda Calculus

The Grammar for LC.

$M, N ::= x, y, z, \ldots$	Variables.
$\mid \lambda x.N$	Abstraction.
$\mid M\ N$	Application

An Idris Example (some liberties with types)
$\lambda x.\neg x$ True

```
module Lambda
f : Bool -> Bool
f = \x => not x

b : Bool
b = f True
```

Functions are limited.

Can we apply any function to any argument?

Clearly we cannot, saying $(3 < 7) + 12$ does not make sense, nor does $(-4) \vee 7$.
Operations are often defined as being closed under a particular set.
Closures are really speaking about the type of operations.

Are there other ways we can categorise functions besides the type of their input and output?

We can also classify functions based on properties of their behaviour. Sometimes we don't care what specific types a function operates on, so long as there is some valid operation defined for those types.

Typed vs Untyped composition

See WHITEBOARD.

Types as a class of behaviour

See WHITEBOARD.

To functional programming

Is everything really a function? (Really?)

The argument goes that a pure functional language is just an elaboration of some variant of a typed lambda calculus. Let's briefly investigate this.

Values as functions.

Are these functions the same?

```
module Values
%default total

seven : Int
seven = 7

seven' : () -> Int
seven' = \x => 7
```

Values as functions.

Are these functions the same?

```
module Values
%default total

seven : Int
seven = 7

seven' : () -> Int
seven' = \x => 7
```

The humble unit type.

The unit type is special. Denoted `()` or $\top$, the unit type is defined as having only one single element, also `()`, or sometimes $*$. The unit element can *always* be constructed. This implies that `seven` is the same as `seven'` as we can always construct the argument to `seven'`.

Data types as functions.

Can we do the same with data types?

We can see this with church encoding, and have a small example below, interested readers should see [Pierce, 2002, p. 58-68].

Data types as functions.

Can we do the same with data types?

We can see this with church encoding, and have a small example below, interested readers should see [Pierce, 2002, p. 58-68].

Church encoding of the Boolean type

```
module ChurchBools
data Boolean : Type where
  True : Boolean
  False : Boolean

true : a -> a -> a
true = \t => \f => t

false : a -> a -> a
false = \t => \f => f
```

Sum Types

The canonical sum type: Either

```
module Sum
%default total
%hide Prelude.Either

data Either : Type -> Type -> Type where
  Left : a -> Either a b
  Right : b -> Either a b
```

Product Types

The canonical product type: Pair

```
module Product
%default total
%hide Prelude.Pair

data Pair : Type -> Type -> Type where
  MkPair : a -> b -> Pair a b
```

Algebraic Data Types

A higher-kinded, parametrically polymorphic, sum of product type: List

```
module List
%hide Prelude.List
%default total

data List : Type -> Type where
  Nil : List a
  (::) : a -> List a -> List a
```

Dependent Types and Type-classes

See actual Idris demo on verified functor and verified applicative.

Is compositionality important, John Hughes?

Figure: John Hughes[1]

[1] Source https://en.wikipedia.org/wiki/John_Hughes_(computer_scientist)#/media/File:John_Hughes_(computer_scientist).jpg

Why Functional Programming Matters Hughes [1989]

Modularity through HOF and composition

John wrote a seminal paper in 1989 on the benefits of functional programming. He argues that modularity and compositionality lead to reusable, readable, debugable code.

Conclusion

FP is about types

Types allow us to talk about well-behaved compositionality and talk about functions when defined by properties on sets.
In very expressive type systems we can even prove those properties.

Conclusion

FP is about types

Types allow us to talk about well-behaved compositionality and talk about functions when defined by properties on sets.
In very expressive type systems we can even prove those properties.

Functions are boring in general, but interesting in specific

At the most general view, all a function can do produce an output given an input. Functions become interesting once you restrict this view to functions-with-particular-properties.

Conclusion

FP is about types

Types allow us to talk about well-behaved compositionality and talk about functions when defined by properties on sets.
In very expressive type systems we can even prove those properties.

Functions are boring in general, but interesting in specific

At the most general view, all a function can do produce an output given an input. Functions become interesting once you restrict this view to functions-with-particular-properties.

Closing comment for the pub

I hypothesise that the more expressive a type system becomes, the less compositional it becomes, but the guarantees of behaviour become stronger.
Is this something we can objectively measure?
Is there a sweet spot for type systems for programming?

References

P. R. Halmos. *Naive set theory*. van Nostrand, 1960.

J. Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

B. C. Pierce. *Types and programming languages*. MIT press, 2002.