Total Functional Programming

Adam Hofmann

March 3, 2017

1 Introduction

Traditionally, we define a function as something like

A relation that uniquely associates members of one set with members of another set. More formally, a function from A to B is an object f such that every a in A is uniquely associated with an object f(a) in B. [1]

However, in computer science, our functions are usually messier and more convoluted than we describe. For example consider:

```
int sum = 0;

// f: Int → Int
int f(int &a) {
  cout << a;
  for (int i = a; i >= 0; i--) {
    sum += a/i;
  }
  a++;
  return sum;
}
```

There's a few things wrong with this function.

- The fact that "a" changes value is never indicated.
- The fact that "a" will be printed to the screen is never indicated
- The function never terminates for negative integers
- The function crashes for non-negative integers

In function programming, bullet points 1 and 2 are not of concern because we disallow mutation (changing the value of variables) and side effects (things that effect the state outside the function, like printing to the screen). Functional programming is driven by the idea that our functions should be transparent and held to what we write.

	Imperative Programming	Functional Programming
Mutation	Yes	No
Side Effects	Yes	No

	Functional Programming	Total Functional Programming
Mutation	No	No
Side Effects	No	No
Termination	No	Yes
Run-time errors	Yes	No

One of the biggest things we get from functional programming is how much easier it is to reason about our programs. In imperative programming, we can have global variables that are changing each time a function is run, meaning that calling a function with the same value back to back can have a different output each time if the function depends and mutates a global value. This makes our programs very hard to reason about and provides lots of motivation to functional program. Essentially in functional programming all we have to worry about is the input and the output, and it's not important what happens in between.

In total functional programming, all of our functions are total. A total function is a function that maps every acceptable input value to a corresponding output value. The contrast to this is a partial function which only maps a subset of the inputs to the outputs. For example, division is a partial function because there is no integer or rational number we can return for something like $5/0$.

This essentially means that we are removing every single run time error and infinite loop in our programs. If there is an infinite loop in a function, then that input has no output in the range that we declared, so it can't be total. The reason we remove run time errors is because we can verify that every single function we are calling on data is valid for that data type, and don't run into cases where a function is not valid on the data we are using mid execution.

We can represent run time errors or non-terminations in an infinite loop as

$\perp$.

We can see more motivation for total functional programming by observing the below:

```
infinite :: Nat -> Nat
infinite n = 1 + infinite n
```

we have *infinite n = 1 + infinite n* and thus 0 = 1, which obviously doesn't play nicely with traditional mathematics. Since *infinite n* is this $\perp$ natural number, it adds uncertainty and edge cases to the mathematics we are performing. We can no longer relying on the fact that Nat - Nat = 0 for every natural number without taking into account $\perp$.

Remember that a total function must have every item in it's domain mapped to an item in it's range. That means that if we are allowing something like undefined in our set of Strings or Natural numbers, every single function that consumes a String or Natural number must handle undefined. The more likely scenario when designing a total language is not allowing undefined in our set of strings, to allow proof theory and function design simpler.

So, the immediate benefit of removing $\perp$ should be obvious: we can no longer have run time errors or programs that don't terminate. What isn't obvious is if this benefit is worth the restrictions that we place upon ourselves. For the rest of this talk I will cover the less obvious benefits of total functional programming, as well as the disadvantages of total functional programming accompanied by clever ways to mitigate these disadvantages.

2 Relationship To Mathematics

To begin, when we are writing total functions, they are extremely similar to mathematics. When we write functions we are often writing both a mathematical definition as well as an algorithm to compute the function.

Consider an (efficient) function that computes the nth Fibonacci number:

```
fib' :: Nat → Nat → Nat → Nat
fib' 0 a b = a
fib' (n+1) a b = fib' n b (a+b)

fib :: Nat → Nat
fib n = fib' n 0 1
```

As well as a proof of correctness for our function:

Theorem: $\forall n \in Nat$ (fib' n f_k f_{k+1}) = f_{k+n}, where f_n is the nth Fibonacci number

Proof: Consider when n is zero. We have (fib' 0 f_k f_{k+1}) = f_k = f_{k+0} and additionally, (fib' 0 f_{k+1} f_{k+2}) = f_{k+1} = f_{k+1+0}, both from the definition of the function.
Assuming that $\exists m \in Nat$ *(fib' m* f_k f_{k+1}*)* = f_{k+m} and *(fib' m* f_{k+1} f_{k+2}*)* = f_{k+m+1}, we can observe from the above definition of fib' that
(fib' m+1 f_k f_{k+1}*)* = *(fib' m* f_{k+1} f_{k+2}*)* = f_{k+m+1}
So we have proven that our function is correct by the inductive hypothesis.

You may have noted that fib and fib' are both total functions, and this helps us immensely in the proof theory. Had the functions not been total, we have additional considerations; what do we do with *(fib* $\perp$*)*? What is $f_\perp$? What is $f_\perp$+f_k?

Even though we can easily write an always terminating and run time error free fib function in functional programming without much trouble, there is still the possibility of calling fib(foo(5)) and foo(5) = $\perp$.

Clearly by making all our functions total, we have a much simpler proof theory and a strong tie to mathematics where the proof of functions follows easily from the definition.

3 Simpler Language

In the introduction, it was briefly mentioned that we can remove things like undefined, null or none. This, along with removing $\perp$ allows us to have a much simpler language design and all around cleaner functions. Consider the definitions:

```
Bool = True | False

doStuff :: Bool → Bool → Nat
```

The possible cases for doStuff are (True True), (True False), (False True), (False False). But what if we have to consider a case where Bool is $\perp$ or undefined as we do in non-total languages? Now we have 16 possible cases to consider. By limiting the domain of our functions we can write much simpler functions that don't have to deal with edge cases, and thus any proofs can also disregard those edge cases.

Another advantage gained is not having to deal with unfamiliar edge cases that are not standard across languages. For example it is not clear how we deal with $\perp \bigwedge True$ or $True \bigwedge \perp$. If we simply evaluate from the left until we find true we have different outputs when we expect they should be the same. If we are strict on there being no errors it might be confusing for programmers to understanding what is happening. Essentially, there is no standard way to deal with $\perp$, and this can cause uncertainty across the language.

4 Interpreting Total Functions

One of the main obstacles in total functional programming would seem to be our interpreter recognizing and accepting total functions. To create a general algorithm that recognizes whether functions terminate or not would be to solve the Halting Problem and is proven to be impossible. What we can settle for however is making an algorithm that determines termination for a lot of functions, and if it can't then it's decided as non-terminating.

In the most basic sense we can force all of our functions to be structural recursion where we reduce one step of the recursive definition of the data for each step. For example, consuming the head of a list or subtracting a natural by 1. Essentially, working down by one on a recursive structure.

```
count list n :: List -> Nat -> Nat
count [] n    = 0
count (n:tail) n = 1 + (count tail n)
count (x:tail) n = (count tail n)
```

This obviously limits the functions that we can write, but also more importantly restricts the way that we can write functions. If every single function has to be recognizable as structural recursion, writing functions becomes much more difficult. As a small improvement on this we can allow Walther Recursion and include any operation that performs recognizable removal. Essentially if we know the work in a recursive call descends from the basic removal, we can still allow it. made This doesn't increase our range of functions, but adds lot's of convenience. For example, we know when division is doing work on the natural numbers, so we can accept the following:

```
sub a b :: Nat -> Nat -> Nat // a - b
sub a 0 = a
sub 0 b = 0
sub a+1 b+1 = sub a b
```

```
quotient a b :: Nat -> Nat -> Nat // a / b
quotient a 0 = 0
quotient 0 b = 0
quotient a b && a < b = 0
quotient a b = 1 + quotient (sub a b) b
```

```
gcd a b :: Nat -> Nat -> Nat
gcd a b && (a == b) = a
gcd a b && (a >  b) = gcd(a-b, b)
gcd a b = gcd(a, b-a)
```

In the above, sub is a structurally recursive definition, while quotient and gcd are both Walther Recursive. In quotient's recursive call we can see that a is always descending by b, and because the function terminates if b = 0 we can accept this as a terminating function. We can say similar things for the gcd function.

To actually add convenience to the interpreter we can make use of, we can make use of Big O notation. If we know the time complexity of a function, we can make an upper bound on the number of calls it will take a function to terminate, that is essentially assuring the interpreter that the function will terminate, and still easily prove that our function is correct.

```
seq a :: Nat -> Nat // a - b
seq 0 = 1
seq 1 = 1
seq a = 1 + seq (a * 3) / 4
```

```
seq` a a :: Nat -> Nat // a - b
seq` n 0 = 1
seq` 0 n = 1
seq` 1 n = 1
seq` a n+1 = 1 + (seq ((a * 3) / 4) n)
```

In the above example we have an ascending operation and a descending one, and it may not be clear to our compiler that this is still strictly descending, so we can put an upper bound of a iterations and then guarantee termination.

5 Codata

Seeing that we have placed the restriction of every function terminating upon ourselves, it can be quite worrying to consider the magnitude of useful actions that we can perform, and it brings into question the usefulness of a total functional programming language. Can we accept user input? Can we write an operating system which is something we never really want to terminate?

Our savior in this sense is *codata*. Codata describes an infinite data structure, like a stream. We can think of a stream as a feed of information that has no definite end, a common example is an input stream where there is no definite end.

We can see the difference between codata and data by looking at two example definitions:

```
data List Type = Nil | Cons Type List
data Stream Type = Cons Type Stream
```

Our stream type doesn't have an empty or Nil construct, it goes on forever! (Note that codata does not necessarily have to be infinite, that is we can have a finite or infinite codata). Now this seems as if it has no place in our total functional programming language, as evaluation of a stream doesn't seem possible. If we create an infinite stream that looks like (1 (2 (3 (...)))) and try to run something like below:

```
sum :: Stream Nat -> Nat
Cons n s = n + sum s
```

It's clear that sum will never terminate.

The key to making sure that we can use codata in our total language is ensuring that recursion is used on data and *corecursion* is used on codata. The difference between recursion and corecursion is in recursion we descend on our data, and in corecursion we instead ascend on the result. As Turner says in his paper on Total Functional Programming, "corecursion creates (potentially infinite) codata, whereas ordinary recursion analyses (necessarily finite) data." For example, consider the below corecursive function:

```
buildlist :: Nat → stream Nat
buildlist n = Cons n (buildlist n+1)
```

We can see that after every iterate, buildlist has returned a steam of Nat, as promised by the function. Thus, corecursion on codata is still total as long as it is consistently productive.

Codata allows us to do some fun things like write an algorithm to compute a stream of all Fibonacci numbers. If we use lazy evalutation, this stream will only ever be evaluated up to what is requested, so if I request the 100th number, only at that time will the first 100 Fibonacci numbers be evaluated. Codata has lots of utility where we want a list of everything up to a certain point, or that list is important in computation. For example, when we are checking if a number is prime, we only have to check if any lower primes are factors of that number, so it is clear that a stream of primes could be useful.

```
fibs :: Stream Nat
fibs = f 0 1

f :: Nat -> Nat -> Stream Nat
f a b = Cons (a (f b (a+b))
```

In addition, we can now handle user input with a stream where it is acceptable to wait an indefinite amount of time. We can also write an operating system where a stream is constantly being created but we don't necessarily have termination.

In addition to corecursion there also exists coinduction that allows us to prove things about codata and corecursion. Understanding coinduction requires more knowledge than can be taught in the span of this talk, and will thus be skipped.

6 Concluding Remarks

Total functional programming clearly has a wide array of advantages and disadvantages. In general, the restriction on functions and the way that functions are written can cause a high barrier to entry and make us step outside of our comfort zone when programming. However there is a wide array of benefits to be gained from total functional programming and there are many reputed. supporters of total functional programming out there. It remains to be seen if total functional programming will ever hit a mainstream

market, but it would seem obvious that functional programming would first have to make that step.

7 Further Reading

None of this talk would have been possible without the work of David A. Turner and his papers on total and strong functional programming. For further knowledge I have attached links Turner's paper on total functional programming and a link to read more on coinduction.

`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.364&rep=rep1&type=pdf`
`http://adam.chlipala.net/cpdt/html/Coinductive.html`

Other topics to pursue include greater knowledge of Walther Recursion and ways to extend our interpreter to allow more functions to be considered total.

References

[1] Stover Christopher, and Weisstein, Eric W. "Function." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Function.html