

Total Functional Programming

...

Adam Hofmann

Why do We Totally Functional Program

Our compiler checks for any run time errors or infinite loops in our code, we can detect these kinds of problems before they occur.

We have an easier time with proving our programs correctness.

We have a simpler language design.





Introduction

a **mathematical 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 .

```
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;
}
```

You vs the Guy She Tells You Not to Worry About

	Imperative Programming	Functional Programming
Avoids Mutation		
Avoids Side Effects		

Functional Programming

Functional programming gives us a lot of nice things. We can be sure that our functions will always return the same output with the same input.

The functional programming style brings programming closer to mathematics.

Functional programming is programming with functions in a style that supports equational reasoning and proof by induction

Total Functions vs Partial Functions

A **Partial Function** is a function from A to B that maps some subset of A to B . So, not every element in A has a mapping to something in B .

We can think of this like division by 0

A **Total Function** maps all elements in A to some element of B . This means that a total function returns an element of B for every single element of A .

What in term'nation



	Functional Programming	Total Functional Programming
Avoids Mutation	✓	✓
Avoids Side Effects	✓	✓
Every Function Terminates	✗	✓
No run time errors	✗	✓

Everything You Know is a Lie

“Functional programming is a very good idea, but we haven’t got it quite right yet. What we have been doing up to now is weak functional programming. What we should be doing is total functional programming.”

- David Turner, implemented the first functional programming languages based on lazy evaluation, combinator graph reduction, and polymorphic types: SASL (1972), KRC (1981), and the commercially supported Miranda (1985).

What's Missing from Functional Programming

We claim to have simple induction, strong types with well defined domains.

But runtime errors and non terminating functions (\perp) throw this for a loop.

Our domains aren't what they seem, and this can throw curve balls into induction.

Why do we Totally Functional Program

Our compiler checks for any run time errors or infinite loops in our code, we can detect these kinds of problems before they occur

We have an easier time with proving our functions because they are totally mapped

We have a simpler language design because our domains and codomains are better defined.

Why Wouldn't we always Program with Total Functions

The language isn't Turing complete! We lose a huge range of functions, both terminating and nonterminating. Among the terminating ones are a interpreter for the language itself.

The language interpreter must be able to verify totality, how do we solve the halting problem?

How do we do input? Can we write an operating system?

Simpler Proofs

```
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
```

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

Proof: Consider when n is zero. We have $(\text{fib}' \ 0 \ f_k \ f_{k+1}) = f_k = f_{k+0}$ and additionally, $(\text{fib}' \ 0 \ f_{k+1} \ f_{k+2}) = f_{k+1} = f_{k+1+0}$, both from the definition of the function.

Assuming that $\exists m \in \text{Nat} \ (\text{fib}' \ m \ f_k \ f_{k+1}) = f_{k+m}$ and $(\text{fib}' \ m \ f_{k+1} \ f_{k+2}) = f_{k+m+1}$, we can observe from the above definition of fib' that

$(\text{fib}' \ m+1 \ f_k \ f_{k+1}) = (\text{fib}' \ m \ f_{k+1} \ f_{k+2}) = f_{k+m+1}$

So we have proven that our function is correct by the inductive hypothesis.

Simpler Proofs

The previous proof doesn't actually work for a functional programming language.

If e is of type Nat , we cannot assume $e - e = 0$ because e might be \perp

If we have $\text{foo} :: \text{Nat} \rightarrow \text{Nat}$

Can we prove that $\text{fib}(\text{foo}(10)) = \text{f_foo}(10)$ when $\text{foo}(10) = \perp$?

Simpler Language

Removing \perp allows us to have a much simpler language design and all around cleaner functions.

There is no standard decisions on how we deal with bottom, creating inconsistency and uncertainty across languages.

Can we say $\text{True} \ \&\& \ \perp = \text{True}$ or should it be \perp ? Why is this any different from $\perp \ \&\& \ \text{True}$?

Termination \square of Total Functions

So how do we solve the Halting Problem?

Obviously we can't

But the halting problem only says that there is no algorithm to decide if every function terminates.

Term'nation □ of Total Functions

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.

```
factorial n :: Nat -> Nat
factorial 0 = 1
factorial n+1 = (n+1) * factorial n
```

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


Walther Recursion

As a small improvement on this we can allow the work that is being done to move beyond removal by 1 and include any operation that we know is descending.

Walther recursion: Recursion where we are using operations that descend to the base case, but not necessarily by 1.

This doesn't increase our range of functions, but adds lot's of convenience.

Walther Recursion

```
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)
```

Term'nation □ of Total Functions

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
```

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

Codata

Placing the restriction of termination upon ourselves limits the amount of functions both terminating and nonterminating that we can write.

But a lot of the time we want things to go on forever, it's quite useful

So we consider something called *Codata*

Codata

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

Our stream doesn't have a definite end, there is no empty or nil construct signifying the end

How can this fit in a total language?

Codata

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

[Recursion on Codata]

We can only perform corecursion on codata and recursion on data.

Notice that corecursion creates (potentially infinite) codata, whereas ordinary recursion analyses (necessarily finite) data.

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

[Corecursion on Codata]

Why is Codata Total?

Corecursive functions over codata are still totally mapped, they aren't partial functions.

In our sum example (that was recursion on codata) I never actually would return a Nat.

However for build list I have indeed returned a stream of Nat's. We can decide to stop the corecursion at any time and move on, and we have a function for the calls that have been evaluated

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

What can we do with streams

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

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


💣💣 We have retained Input and OS's 💣 💣

Using codata we can now accept user input as a stream

This is the same idea as an I/O stream in C++.

We also have the ability to write OS's that are constantly creating data.

Summary

- We completely avoid run time errors or non terminations (unexpected crashes)
 - We can write programs that don't ever terminate as long as they are productive
 - We can prove things about our language easier
 - Our language is simpler, we don't have to consider stuff like \perp && True
-
- We might have to put more work into writing functions
 - We aren't Turing Complete
 - There is some terminating functions we cannot write.