Maths, Programs, and The Curry-Howard Correspondence

Ruthie Gawley

April 2026

Contents

1 Introduction

```
theorem composition (A B C : Prop)(f:  A → B)(g:  B → C)(a:  A) : C :=
    g (f a)
```

I know what you are thinking; a block of code is a very strange way to start an essay about mathematics. However, this block of code, written in the programming language Lean, is actually the computerised version of a mathematical proof! Throughout this essay I will be gradually exploring this example to demonstrate that theorem proving and computer programming are really one and the same - an incredible mapping between mathematics and computer science known as the Curry-Howard Correspondence.

2 First Line

2.1 What is Type?

```
theorem composition (A B C : Prop)(f:  A → B)(g:  B → C)(a:  A) : C :=
```

In this section, we will explore the first line of this computerised proof as a foothold in the universe of the Curry-Howard Correspondence and 'types' in particular. This line is essentially a set-up, defining the key statements we will need and giving the overarching goal of the theorem. Before we start, let's get some intuition by considering the equivalent statement in normal mathematical notation:

$$\overbrace{\forall A, B, C \in \mathcal{P}, (A \to B, B \to C, A)}^{\text{Given that}} \quad \overbrace{\vdash C}^{\text{We want to prove}}$$

> This may look intimidating at first - but it just says that this theorem applies to propositions ($\mathcal{P}$) (any logical/mathematical statements) A, B, and C for which:
>
> A implies B ($A \to B$)
> B implies C ($B \to C$)
> A is true (A)
>
> And the goal is to prove that C is then true.

Looking back at the initial line of code, there are similarities with the mathematical translation - but the exact mapping is not apparently obvious. That is because the code is heavily reliant on the idea of a 'term' and a 'type'. The Curry-Howard Correspondence is based on a form of mathematics called type theory. The main property of type theory is that any term in an expression (such as `A`, `B`, `C`, `f`, `g`, or anything else) has a type - a classification which it falls into. To wrap our heads around this, we will work through the types and terms in this line of code.

`(A B C : Prop)` This defines *terms* `A`, `B` and `C` as *terms* of *type* `Prop`
Isomorphism: A, B, and C are propositions

`(f : A → B)` *Term* `f` which is of *type* `A → B`
Isomorphism: The assumption that A implies B

`(g : B → C)` *Term* `g` which is of *type* `B → C`
Isomorphism: The assumption that B implies C

`(a : A)` *Term* `a` which is of *type* `A`
Isomorphism: The assumption that A is true

`composition (A B C : Prop) (f: A → B) (g: B → C) (a: A) : C :=`
The *Term* `composition` has the *type* `(A B C : Prop) → (A → B) → (B → C) → A → C`
Isomorphism: The theorem 'composition' intends to prove that given propositions A, B, and C and the properties $(A \to B)$ and $(B \to C)$ as well as that A is true, C is true
In mathematical notation, this statement is equivalent to ...

$$\overbrace{\forall A, B, C \in \mathcal{P}, (A \to B, B \to C, A)}^{\text{Given that}} \quad \overbrace{\vdash C}^{\text{We want to prove}}$$

We can see a clear isomorphism between mathematical propositions and types. Mathematical propositions like A, A→B, and B→C have not been directly defined; instead they have been defined as terms like a, g and f which the *types* of A, A→B, and B→C. Moreover, the type of our theorem, 'composition', is the overall proposition we are aiming to prove. Hence, it is types (categories of terms) model mathematical propositions. This "propositions as types" mapping may seem convoluted but will be incredibly useful later when we actually start proving the theorem. It is one of the underlying concepts of the Curry-Howard Isomorphism.

2.2 What is a Type really?

Before we dive into more detail, it is worth pausing and understanding the nuances of types in type theory. You may have noticed something which seems, at surface level, paradoxical. How are `A`, `B` and `C` both terms of type 'Proposition' and the types for `f` and `g`? This is because every statement in Type Theory has a type. Hence, even the types have types. So, while `A → C` is a type, both `A` and `C` are also terms which have types themselves. Any statement can be seen as a term, or a type, depending on context.

However, there is still a risk of paradox. The idea of every object having a type sounds very similar to set theory, in which objects fall into a set. In set theory, without strict regulations we encounter a logical trap; Russel's paradox. Mathematician Bertrand Russel constructed the following set:

Set R: The set of all sets that do not contain themselves

Set R presents a logical contradiction. If it does contain itself, it shouldn't; if it does not contain itself, it should (this is worth thinking about for a bit!). It is possible to construct a similar paradox using the idea of a type containing itself in type theory, Girard's paradox. Hence, unaltered, type theory would have a paradoxical flaw which prevents it from mirroring mathematics as required for Curry-Howard.

To prevent this, all types fall into a strictly ordered hierarchy of 'Sorts'. Every term must make a type which is one above it in this hierarchy To give an example, `A` is a Sort 1 type, and `Prop` is a Sort 2 type. Because `Prop` is one sort level above `A`,

`(A : Prop)`
A is a term of type Prop

is a valid proposition. However, assigning types out of order is invalid. So, a statement such as `(A : A)` would be invalid as both term and type are Sort 1. Hence paradox by self-ownership is impossible. Type theory is safe!

I hope this close consideration of the true nature of types helps to build intuition as to what a type really is. However, one (big) thing is still missing for our demonstration of the Curry-Howard Correspondence: the ability to actually manipulate terms and types instead of statically defining them. This is what is coming up in Line 2.

3 Second Line

3.1 Manipulating Types

```
theorem composition (A B C : Prop)(f:  A → B)(g:  B → C)(a:  A) : C :=
    g (f a)
```

The second line of this program seems mercifully short in contrast to the gargantuan first line, which provided the goal and initial definitions of our terms and types. However, this second line has an important role; it needs to actually prove the theorem we have defined! That means manipulating the statements to demonstrate that we can reach C. In essence,

$$\overbrace{\forall A, B, C \in \mathcal{P}, (A \to B, B \to C, A)}^{\text{Given that}} \qquad \overbrace{\vdash C}^{\text{Actually prove this part now}}$$

Through the Curry-Howard Correspondence (propositions are types!) that is equivalent to using the terms given to us in the first line, `f` `g` and `a`, (and, most importantly, their types) to produce another term which has type we are trying to prove: `C`

But how can we actually manipulate types to show this? Actually, we can apply terms to each other to alter their types, treating them like functions. Taking a step back, a function is a rule that takes an input, carries out an operation, and hence maps it onto an output.

The ability of terms to act as functions, and manipulate other terms, can create new terms with new types. By the Curry-Howard Correspondence, creating new types means creating

new propositions. This is exactly what proof is about! Let's see how functions will aid our proof as we get stuck into line 2.

> Starting with the innermost function,
> `f a`
> Translation: apply `f` as a function to the term `a`
> Types: `a` has the type `A`. `f` has the type `(A → B)`. Hence `f` applied to `a` maps type `A` onto type `B`. So `f a` has type `B`.
> Isomorphism to Maths: Given A is true, if $A \to B$ (A implies B) then B is true.
>
> `g (f a)`
> Translation: now, apply `g` to `f a`
> Types: `f a` has type `B`, derived above. `g` has the type `(B → C)`. Hence, `g` applied to `f a` maps type `B` onto type `C`. Overall, `g (f a)` has type `C`.
> Isomorphism to Maths: Given B is true, $B \to C$ implies C is true.

The headline: the final type of `g (f a)` is `C`!

3.2 The Final Type

That is the final piece of the Curry-Howard puzzle! We started with the term `a`, of type `A`. Then we applied the term `f`, of type `A → B`. Then we applied the term `g`, of type `B → C`. The final type of the term `g(f a)` was `C`. Looking at the types involved, we have...

$$\texttt{(A B C : Prop)} \to \texttt{(A} \to \texttt{B)} \to \texttt{(B} \to \texttt{C)} \to \texttt{A} \to \texttt{C}$$

which is the type of the theorem, `composition`, that we set out to prove in the first place! Because we have shown that it is possible to reach this proposition, we have shown that the corresponding mathematical proposition:

$$\forall A, B, C \in \mathcal{P}, (A \to B, B \to C, A) \quad \vdash \quad C$$

is completely correct!

This is a living and breathing example of the Curry-Howard Correspondence; types correspond to propositions, so when our program demonstrates that you can build the desired type it has verified the proposition. Hence, it is a proof. A program is a proof!

4 Conclusion

Thank you for joining me on a trek through the weird, wonderful, and isomorphism-rich world of the Curry-Howard Correspondence. We have used a concrete example to establish that types are propositions; we can build and manipulate types by using terms as functions; and a program which produces the correct type is a proof. Essentially, with the power of types and functions we can build mathematics. This process of translating proofs into programs is a fascinating one at the forefront of current research, which I have only been able to scratch the surface of ... but to leave you with the most enduring message: types are pretty cool!

Bibliography

[1] Terence Tao. *Machine Assisted Proof*. Notices of the American Mathematical Society, Vol. 71, No. 1, 2024. Available at: https://www.ams.org/journals/notices/202401/rnoti-p18.pdf

[2] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. *Theorem Proving in Lean 4*. 2023. Available at: https://leanprover.github.io/theorem_proving_in_lean4/

[3] Brown University Department of Computer Science. *Girard's Paradox*. CS1951X: Formal Proof and Verification. Available at: https://cs.brown.edu/courses/cs1951x/docs/logic/girard.html

[4] Michael Ryan Clarkson. *The Curry-Howard Correspondence*. YouTube, 2020. Available at: https://www.youtube.com/watch?v=GdcOy6zVFC4

[5] Jeremy Avigad and Patrick Massot. *Mathematics in Lean*. 2023. Available at: https://leanprover-community.github.io/mathematics_in_lean/