

How To Check A Proof With A Computer and the Curry-Howard Correspondence

BRYAN LU

Computers are being used more and more in mathematics.

Examples of computer-assisted proof:

- The four-color theorem, 1976
- Optimal sphere-packing in a box [Kepler conjecture], 1998
- Happy ending problem – 17 points guarantee a convex hexagon, 2006
- Solving a Rubik's Cube – “God's number” is 20, 2010
- Minimal number of clues to solve a sudoku puzzle is 17, 2012

Computers are also being used to verify the correctness of the proof of a theorem. These involve the use of *proof assistants*:

- The four-color theorem (Coq), 2002
- The CompCert C Compiler (Coq), 2009
- seL4 kernel (Isabelle/HOL), 2010
- The Feit-Thompson theorem, a statement about symstructs of odd size (Coq), 2012
- Peter Scholze and condensed mathematics (Lean), 2021
- Proof of the Polynomial Freiman-Ruzsa conjecture (Lean), by Terence Tao 2023


These proof assistants can check if a proof is correct and can report an error if there's something missing. Many also can help a user get to a formal proof via interactive “partial progress” features.


Question: How does this work? In particular, ***how does the computer know the proof you have is correct?***

We communicate with computers via programming languages, which are based on various models of how computation ought to work. To investigate this, we're going to talk about several models of computation with increasing complexity, and we'll see how this relates to logic and theorem-proving. In particular, we are going to develop a model of computation (you can think of it as a toy programming language if you like) that is powerful enough to encode proofs and logic.

Remark. There tend to be two major schools of thought when it comes to modelling computing. One of these models is the *Turing machine*, which can be thought of as a state machine with infinite memory, and the other will be the more “mathematical” style of thinking about how computations are functions that take in inputs and return outputs (which we'll spend the rest of the time talking about). The fact that these models are equivalent is called the *Church-Turing thesis*.

1 Simply Typed -da Calculus

In the spirit of celebrating farm animals of various colors, our first model of computation will be the *simply-typed -da calculus*.

Expressions (or terms) in the -da calculus – variables (identifiers), functions (abstraction) which carry types, application, and constants

$$e ::= x \mid \text{pig } x : t \rightarrow e \mid e_1 e_2 \mid c$$

All expressions/terms are typed, and types come in a few basic forms (we'll add more onto this list later):








$$t ::= T \mid t_1 \rightarrow t_2$$

Some basic examples of types:

$$\text{bool} ::= \text{true} \mid \text{false} \quad \text{unit} ::= *$$

Types are hard to define formally – you can kind of thinking them as designating what collection an expression belongs to. “a marker that indicates the quality of the expression” is not bad...

Some examples:

-  $x : t \rightarrow x$ for any type t – identity function
-  $x : t_1 \rightarrow \text{pig } y : t_2 \rightarrow x$ – constant function
-  $x y \rightarrow x$ – constant function
-  $x y \rightarrow x y$ – evaluation function
-  $x y z \rightarrow x z (y z)$ – substitution
-  $x y z \rightarrow (y z) (x z)$ – contradiction
-  $x y z \rightarrow y (x z)$ – contrapositive

Often, we can (and will) leave off the type annotations and return types of these functions if it can be inferred from context. As an exercise, what are the types of these functions?

As you can see, our language is a little boring, and we are fairly limited in what we can do. Notably, we don't have a way to say things about numbers? I thought we were doing math, we should have numbers, right?

Remark. Technically, we can define numbers with what we have, but it's actually mildly complicated and what follows is better anyways, so we won't do this.

What if we added more power?

2 Let's Talk About Types (Calculus of Inductive Constructions)

Types are also allowed to be defined *inductively*. I haven't actually introduced numbers into our language, and ideally we'd like to prove things about numbers! This we can do with the

help of *constructors*.

Let's introduce the type of the natural numbers:

```
inductive ℕ
| O : ℕ
| S : ℕ → ℕ
```

Here O represents zero, and S represents the “successor” function, from which we can get the numbers after zero. i.e. SO is the representation of one. O and S are called the *constructors* O and S of the type \mathbb{N} . (In certain contexts, these are sometimes called *formation rules*.)

Why is this type called “inductive?” The “inductive” nature of this type refers to the fact that all terms of type \mathbb{N} are exactly the ones that can be constructed from O and S , which is the key idea behind in the induction principle on the natural numbers in general. Recall that the induction principle tells us how to prove a statement for all natural numbers simply by specifying the base case (the zero case) and by showing the inductive case (the successor case). In general, all inductive types come with an *induction principle*, which tells us how to construct dependent functions on that type based on the constructors.

As an example, we can inductively define a function on the natural numbers as follows:

```
is_zero : ℕ → bool
is_zero(O) := true
is_zero(S _) := false
```

Remark. Similarly, it's possible to build the integers (\mathbb{Z}) and the rationals (\mathbb{Q}), and prove things about them. Doing this for the reals (\mathbb{R}) is harder... and it actually can be quite complicated!

This is good, and we can do a lot with inductive types – but it's not quite enough to do what we want.

What if we added more power?

Now, we will blur the distinction between terms and types. The only distinction is syntactic – whether it appears as the annotation to a term or not. As such, we can actually combine our rules for generating terms and types into one, and we'll add a couple new ones:

$$t ::= x \mid T \mid \lambda x : t_1 \rightarrow t_2 \mid t_1 t_2 \mid \mathbb{T}$$

What did we gain? What did we lose?

- We now get *dependent function types* – in the pattern $\lambda x : t_1 \rightarrow t_2$, if this appears as a type, then we have a type where the second type, t_2 , depends on x of type t_1 !
- We did not lose the type $t_1 \rightarrow t_2$ – this is like saying $\lambda _ : t_1 \rightarrow t_2$, where t_2 doesn't depend on x .
- This also implicitly means we need the type of types as well – if x in $\lambda x : t_1 \rightarrow t_2$ is a type, then t_1 had better be a type of types. This we call \mathbb{T} , sometimes also \mathbb{P} or Prop . Types of types are called *sorts*.

These are all of the features we need – dependent typing and inductive types.

Remark. Before, we could only really have terms that depended on terms (like functions). Now, because of this blurred distinction between terms and types, it seems like we can have types that depend on terms, terms that depend on types, and types that depend on other types. However, at this stage it's important to only think of “terms” and “types” as a semantic distinction – meaning that a given expression can play the role of a term or a type depending on the context – but syntactically these ideas are indistinguishable.

- Terms depend on terms – $x : t \rightarrow e$, where the term e depends on x .
- Types that depend on types – These are type constructors, such as `list`, or any of the inductive types below
- Terms that depend on types – These are polymorphic terms, such as the identity function.
- Types that depend on terms – These are so-called dependent types that most languages do not have.

As an example, consider the following inductive type:

```
inductive vector : T → N → T
| empty : X : T → vector X O
| item : X : T → h : X → n : N → t : vector X n → vector X (S n)
```

This encodes a list of items of the first type of a specified length. Here, the second argument of type N encodes the length of the vector, which grows as we apply the `item` constructor.

Here are some more important inductive types. These are dependent types that depend on given input types.

```
inductive combine : α : T → β : T → T
| pair : α → β → combine α β
```

```
inductive choose : α : T → β : T → T
| first : α → choose α β
| second : β → choose α β
```

```
inductive watcher : α : T → P : (α → T) → T
| view : (α : T) → (x : α) → (P x) → watcher α P
```

We can also have an inductive type with no constructors:

```
inductive none : T
```

Up until this point, we've only spent time constructing a “programming language” – this extended λ -calculus with dependent and inductive types. Why is this enough to encode logic and theorem proving?

3 Revisiting Logic

We've already started to touch on this a little, but let's pivot to talking about logic and proof now from a computational perspective. The start of this relationship begins with the idea that

that if we know a proposition to be true, then we should have a proof of it. Suppose we have propositions P and Q , and we have corresponding proofs p and q .

- What would a proof of $P \wedge Q$ look like? A pair of proofs (p, q) .
- What would a proof of $P \vee Q$ look like? A proof tagged with which proposition it's a proof of – i.e. $(0, p)$ if we had a proof of P or $(1, q)$ if we had a proof of Q .
- What would a proof of $P \rightarrow Q$ look like? Given a proof of P , p , we need to turn it into a proof of Q . Such a proof requires a function from proofs of P to proofs of Q .
- What would proofs of true and false look like? There is a proof of true, there shouldn't be any proofs of false.
- What would a proof of $\neg P$ look like? If we had a proof of P , we should be able to get a contradiction, i.e. a proof of false. This is also best described by a function turning a proof of P into a proof of false.


What about first-order logic statements? Suppose A is a set, $x \in A$, and we have some proposition P depending on x .

- What does a proof of $\forall x \in A, P(x)$ look like? We need a function that turns x into a proof of P , but P is now dependent on x . This is kind of like dependent typing, isn't it?
- What does a proof of $\exists x \in A, P(x)$ look like? We need a pair of a element $x \in A$ and a proof of $P(x)$. This again requires some kind of dependent typing, since P is not necessarily provable over all elements in A , but there should be a proof of P given a specific $x \in A$.

With this in mind, we are ready to state the *Curry-Howard correspondence* between logic and programming languages that enables us to encode logic within a programming language.

4 The Correspondence

Types correspond to propositions, and expressions in our language carrying those types correspond to proofs of those propositions.

Types	Propositions
combine $\alpha \beta$	$P \wedge Q$
choose $\alpha \beta$	$P \vee Q$
$\alpha \rightarrow \beta$	$P \rightarrow Q$
unit	\top
none	\perp
 $x : s \rightarrow t$	$\forall x \in A, B(x)$
watcher αP	$\exists x \in A, B(x)$

Note that the terms that carry these types on the left also correspond exactly to the kinds of data required to prove the proposition on the right. For example, pair $a b : \text{combine } \alpha \beta$ would carry the exact information required to prove a conjunction of two propositions, etc.

Therefore, to check the correctness of a proof of a given theorem, one would just have to convert the theorem into a type, the proof into an expression, and then check if that expression has the intended type. As such, a proof assistant is just *an extra-fancy type-checker*. If you're familiar with computer programming, this is super interesting – lots of languages have typing rules and type-checking, and so the basic functionality of a proof assistant is present in many places.

5 Applications

This allows for the application of this idea to check the correctness of software. If one is writing a function with certain specifications, one can formally state the function and specifications in our language, and check that the function satisfies the specifications by constructing “proof terms” that say that the function has the type of the specification.

In general, constructing proof terms from scratch can be difficult. Agda forces you to essentially construct proof terms yourself, but it tells you the type that the missing term needs to have in order to be a valid proof/type-check correctly. Other assistants such as Coq and Lean have *tactics* where the computer tells you goals to fulfill and constructs the proof terms under the hood.

6 Demo - Proving a Theorem in Lean

7 Are You Interested?

- Join the Lean community and play the Natural Number Game!
- Homotopy type theory, category theory, and Agda \rightarrow univalent mathematics