Automatic Theorem Provers

Hype for Types

February 23, 2026

Who Do You Trust?

Who Do You Trust?

"How do you know something is true?"

Who Do You Trust?

"How do you know something is true?"

We trust things all the time:

- A bridge — engineers signed off on it

Who Do You Trust?

"How do you know something is true?"

We trust things all the time:

- A bridge — engineers signed off on it
- A drug — trials ran and the data checked out

Who Do You Trust?

"How do you know something is true?"

We trust things all the time:

- A bridge — engineers signed off on it
- A drug — trials ran and the data checked out
- A friend — track record built over time

Who Do You Trust?

"How do you know something is true?"

We trust things all the time:

- A bridge — engineers signed off on it
- A drug — trials ran and the data checked out
- A friend — track record built over time
- A proof — because... we read it and it seemed fine?

Who Do You Trust?

"How do you know something is true?"

We trust things all the time:

- A bridge — engineers signed off on it
- A drug — trials ran and the data checked out
- A friend — track record built over time
- A proof — because... we read it and it seemed fine?

Who Do You Trust?

"How do you know something is true?"

We trust things all the time:

- A bridge — engineers signed off on it
- A drug — trials ran and the data checked out
- A friend — track record built over time
- A proof — because... we read it and it seemed fine?

The problem

All of these are *social* trust. What would *logical* certainty look like?

The Four Color Theorem

Claim: Any map can be colored with 4 colors so no adjacent regions share a color.

The Four Color Theorem

Claim: Any map can be colored with 4 colors so no adjacent regions share a color.

- 1879: Alfred Kempe publishes a proof
- Gets cited, built upon, taught
- . . . for **11 years**

The Four Color Theorem

Claim: Any map can be colored with 4 colors so no adjacent regions share a color.

- 1879: Alfred Kempe publishes a proof
- Gets cited, built upon, taught
- . . . for **11 years**

1890

Percy Heawood finds a fatal flaw.
The proof was **wrong**.

The Four Color Theorem

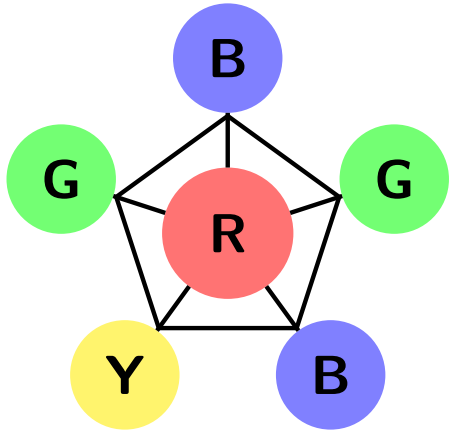
Claim: Any map can be colored with 4 colors so no adjacent regions share a color.

- 1879: Alfred Kempe publishes a proof
- Gets cited, built upon, taught
- . . . for **11 years**

1890

Percy Heawood finds a fatal flaw.
The proof was **wrong**.

Smart people wrote it. Smart people reviewed it. Still wrong.



Every region touches the center.
Outer ring is a 5-cycle (odd).

3 colors fail. 4 are needed.

So What Would Perfect Verification Look Like?

Something so mechanical that *checking it requires no judgment* — just following rules.

So What Would Perfect Verification Look Like?

Something so mechanical that *checking it requires no judgment* — just following rules.

The actual proof arrived in 1976

Appel and Haken proved it — with a computer checking hundreds of cases no human could verify by hand.

So What Would Perfect Verification Look Like?

Something so mechanical that *checking it requires no judgment* — just following rules.

The actual proof arrived in 1976

Appel and Haken proved it — with a computer checking hundreds of cases no human could verify by hand.

And in 2005, Georges Gonthier *fully formalized* it in Coq.
Every step machine-verified. No "it seemed fine" anywhere.

So What Would Perfect Verification Look Like?

Something so mechanical that *checking it requires no judgment* — just following rules.

The actual proof arrived in 1976

Appel and Haken proved it — with a computer checking hundreds of cases no human could verify by hand.

And in 2005, Georges Gonthier *fully formalized* it in Coq.
Every step machine-verified. No "it seemed fine" anywhere.

That's what we're talking about today.

Contracts and Code

Humans Invented Contracts

Contracts exist because natural language is ambiguous.

Humans Invented Contracts

Contracts exist because natural language is ambiguous.

A real software contract clause (paraphrased)

"The software shall perform substantially in accordance with the accompanying materials under normal use."

Humans Invented Contracts

Contracts exist because natural language is ambiguous.

A real software contract clause (paraphrased)

"The software shall perform substantially in accordance with the accompanying materials under normal use."

- What counts as *substantially*?
- What is *normal use*?
- Who decides when you're in violation?

Humans Invented Contracts

Contracts exist because natural language is ambiguous.

A real software contract clause (paraphrased)

"The software shall perform substantially in accordance with the accompanying materials under normal use."

- What counts as *substantially*?
- What is *normal use*?
- Who decides when you're in violation?

The dream

A contract so precise its meaning is *unambiguous* — and a machine can check if you violated it.

Code is a Contract

Every function you write is an implicit promise:

"Give me these inputs, I'll give you this output."

Code is a Contract

Every function you write is an implicit promise:

> *"Give me these inputs, I'll give you this output."*

But who enforces that promise?

- **Tests** — check *some* cases, never all of them

Code is a Contract

Every function you write is an implicit promise:

> *"Give me these inputs, I'll give you this output."*

But who enforces that promise?

- **Tests** — check *some* cases, never all of them
- **Code review** — checks as far as your reviewer's attention holds

Code is a Contract

Every function you write is an implicit promise:

> *"Give me these inputs, I'll give you this output."*

But who enforces that promise?

- **Tests** — check *some* cases, never all of them
- **Code review** — checks as far as your reviewer's attention holds
- **Types** — check *what kind* of thing, not *what's true* of it

Code is a Contract

Every function you write is an implicit promise:

> *"Give me these inputs, I'll give you this output."*

But who enforces that promise?

- **Tests** — check *some* cases, never all of them
- **Code review** — checks as far as your reviewer's attention holds
- **Types** — check *what kind* of thing, not *what's true* of it

Code is a Contract

Every function you write is an implicit promise:

> *"Give me these inputs, I'll give you this output."*

But who enforces that promise?

- **Tests** — check *some* cases, never all of them
- **Code review** — checks as far as your reviewer's attention holds
- **Types** — check *what kind* of thing, not *what's true* of it

The question

What if we stated the contract precisely enough that a machine could verify it for *all* inputs, *forever*?

Enter: Theorem Provers

A **theorem prover** verifies mathematical statements *mechanically*.

Enter: Theorem Provers

A **theorem prover** verifies mathematical statements *mechanically*.

Three flavors worth knowing:

Kind	**You provide**	**It does**
Proof Assistant	A proof you wrote	Checks every step
SMT Solver	A goal	Finds proof or counterexample
Model Checker	A system spec	Exhaustively checks properties

Enter: Theorem Provers

A **theorem prover** verifies mathematical statements *mechanically*.

Three flavors worth knowing:

Kind	**You provide**	**It does**
Proof Assistant	A proof you wrote	Checks every step
SMT Solver	A goal	Finds proof or counterexample
Model Checker	A system spec	Exhaustively checks properties

Proof Assistants: Agda, Coq, Lean **SMT:** Z3, CVC5 **Model Checkers:** TLA+, Alloy

How SMT Solvers Work

SAT: The Foundation

Intuition

You have a puzzle with boolean constraints. **SAT** asks: is there an assignment of true/false to variables that satisfies all of them?

SAT: The Foundation

Intuition

You have a puzzle with boolean constraints. **SAT** asks: is there an assignment of true/false to variables that satisfies all of them?

Example

$$(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee \neg C)$$

SAT: The Foundation

Intuition

You have a puzzle with boolean constraints. **SAT** asks: is there an assignment of true/false to variables that satisfies all of them?

Example

$$(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee \neg C)$$

Try $A = \top,\ B = \bot$:

- First clause: $\top \vee \bot = \top$ ✓
- Second clause: $\bot \vee C$ forces $C = \top$ ✓
- Third clause: $\top \vee \bot = \top$ ✓

SAT: The Foundation

Intuition

You have a puzzle with boolean constraints. **SAT** asks: is there an assignment of true/false to variables that satisfies all of them?

Example

$$(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee \neg C)$$

Try $A = \top,\ B = \bot$:

- First clause: $\top \vee \bot = \top$ ✓
- Second clause: $\bot \vee C$ forces $C = \top$ ✓
- Third clause: $\top \vee \bot = \top$ ✓

Satisfiable with $A = \top,\ B = \bot,\ C = \top$.

DPLL: The Elegant Core

Sound familiar?

This looks a lot like the search Prolog does — pick a value, follow consequences, backtrack on failure. That intuition is exactly right. DPLL *is* that search, made systematic and efficient.

DPLL: The Elegant Core

Sound familiar?

This looks a lot like the search Prolog does — pick a value, follow consequences, backtrack on failure. That intuition is exactly right. DPLL *is* that search, made systematic and efficient.

1. **Pick** a variable, assign true

DPLL: The Elegant Core

Sound familiar?

This looks a lot like the search Prolog does — pick a value, follow consequences, backtrack on failure. That intuition is exactly right. DPLL *is* that search, made systematic and efficient.

1. **Pick** a variable, assign true
2. **Propagate** — if A is true and we need $\neg A \vee C$, then C must be true

DPLL: The Elegant Core

Sound familiar?

This looks a lot like the search Prolog does — pick a value, follow consequences, backtrack on failure. That intuition is exactly right. DPLL *is* that search, made systematic and efficient.

1. **Pick** a variable, assign true
2. **Propagate** — if A is true and we need $\neg A \vee C$, then C must be true
3. **Conflict?** Backtrack and flip

DPLL: The Elegant Core

Sound familiar?

This looks a lot like the search Prolog does — pick a value, follow consequences, backtrack on failure. That intuition is exactly right. DPLL *is* that search, made systematic and efficient.

1. **Pick** a variable, assign true
2. **Propagate** — if A is true and we need $\neg A \vee C$, then C must be true
3. **Conflict?** Backtrack and flip
4. Repeat until satisfied or proven impossible

DPLL: The Elegant Core

Sound familiar?

This looks a lot like the search Prolog does — pick a value, follow consequences, backtrack on failure. That intuition is exactly right. DPLL *is* that search, made systematic and efficient.

1. **Pick** a variable, assign true
2. **Propagate** — if A is true and we need $\neg A \vee C$, then C must be true
3. **Conflict?** Backtrack and flip
4. Repeat until satisfied or proven impossible

DPLL: The Elegant Core

Sound familiar?

This looks a lot like the search Prolog does — pick a value, follow consequences, backtrack on failure. That intuition is exactly right. DPLL *is* that search, made systematic and efficient.

1. **Pick** a variable, assign true
2. **Propagate** — if A is true and we need $\neg A \vee C$, then C must be true
3. **Conflict?** Backtrack and flip
4. Repeat until satisfied or proven impossible

Why it's fast in practice

Unit propagation collapses the search tree aggressively.
The "magic" is *clever backtracking*, not brute force.

From SAT to SMT

Intuition

SAT only knows about Booleans, however real programs have integers, arrays, strings. **SMT** layers *theories* on top of SAT to handle these.

From SAT to SMT

Intuition

SAT only knows about Booleans, however real programs have integers, arrays, strings. **SMT** layers *theories* on top of SAT to handle these.

Z3 in action

```
x, y = Ints('x y')
solve(x + y == 7, x > 2, y > 2)
# => x = 3, y = 4

solve(x > 5, x < 3)
# => no solution
```

From SAT to SMT

Intuition

SAT only knows about Booleans, however real programs have integers, arrays, strings. **SMT** layers *theories* on top of SAT to handle these.

Z3 in action

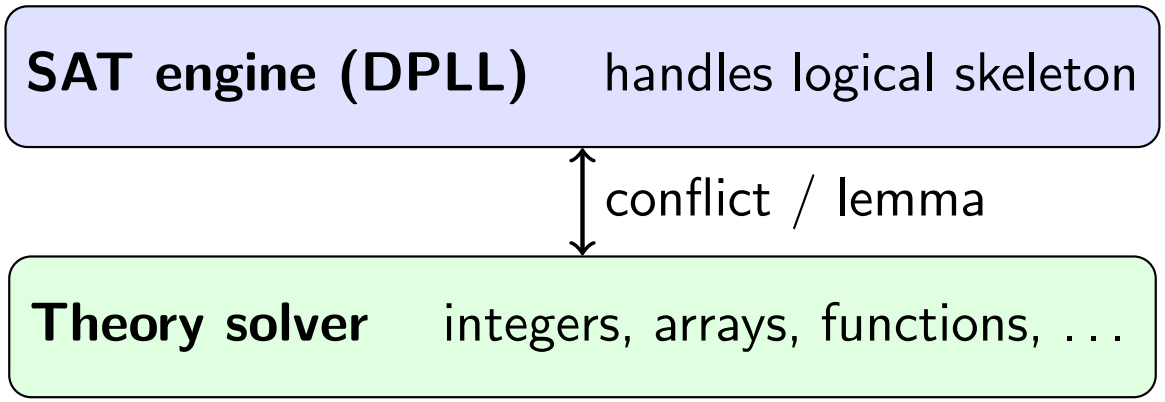
```
x, y = Ints('x y')
solve(x + y == 7, x > 2, y > 2)
# => x = 3, y = 4

solve(x > 5, x < 3)
# => no solution
```

Z3 didn't try every integer. It *reasoned* that $x > 5$ and $x < 3$ are contradictory.

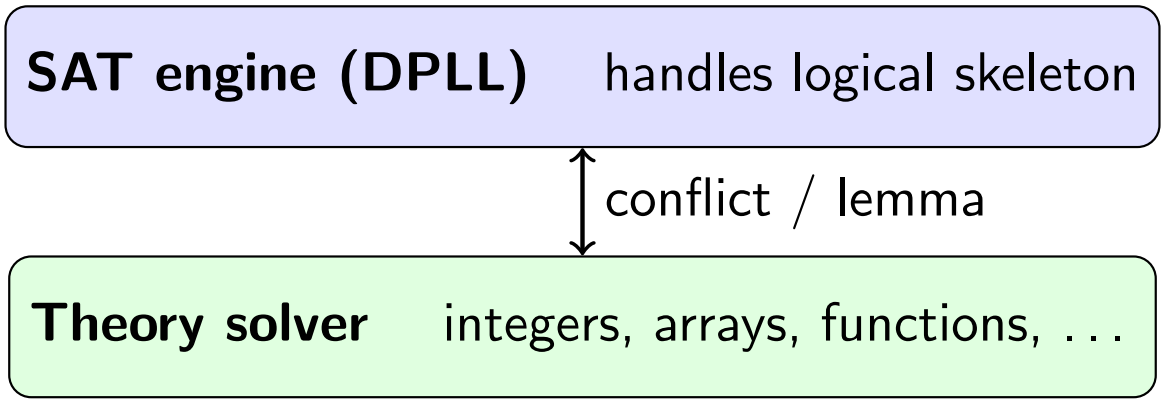
The DPLL(T) Architecture

SMT solvers run two layers in lockstep:



The DPLL(T) Architecture

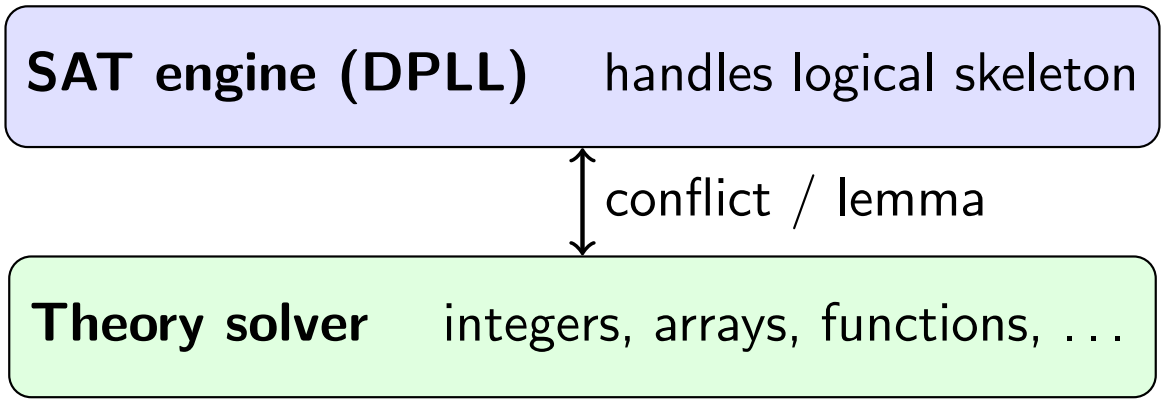
SMT solvers run two layers in lockstep:



1. SAT engine picks a combination of sub-formulas that *could* be true
2. Theory solver checks if it's *actually* consistent
3. If not: theory solver hands back a conflict, SAT engine backtracks

The DPLL(T) Architecture

SMT solvers run two layers in lockstep:



1. SAT engine picks a combination of sub-formulas that *could* be true
2. Theory solver checks if it's *actually* consistent
3. If not: theory solver hands back a conflict, SAT engine backtracks

Want to reason about a new domain? Just swap the theory solver.

Proof Assistants & Type Theory

Proof Assistants

Intuition

Instead of *finding* a proof, you *write* one — and the machine checks every step with zero tolerance for gaps.

Proof Assistants

Intuition

Instead of *finding* a proof, you *write* one — and the machine checks every step with zero tolerance for gaps.

Think of it as a **compiler for mathematical arguments**.

- Compiles $\Rightarrow$ the argument is airtight

Proof Assistants

Intuition

Instead of *finding* a proof, you *write* one — and the machine checks every step with zero tolerance for gaps.

Think of it as a **compiler for mathematical arguments**.

- Compiles $\Rightarrow$ the argument is airtight
- Doesn't compile $\Rightarrow$ there's a gap somewhere

Proof Assistants

Intuition

Instead of *finding* a proof, you *write* one — and the machine checks every step with zero tolerance for gaps.

Think of it as a **compiler for mathematical arguments**.

- Compiles $\Rightarrow$ the argument is airtight
- Doesn't compile $\Rightarrow$ there's a gap somewhere
- No "this seems right" — only "this typechecks"

Proof Assistants

Intuition

Instead of *finding* a proof, you *write* one — and the machine checks every step with zero tolerance for gaps.

Think of it as a **compiler for mathematical arguments**.

- Compiles $\Rightarrow$ the argument is airtight
- Doesn't compile $\Rightarrow$ there's a gap somewhere
- No "this seems right" — only "this typechecks"

Proof Assistants

Intuition

Instead of *finding* a proof, you *write* one — and the machine checks every step with zero tolerance for gaps.

Think of it as a **compiler for mathematical arguments**.

- Compiles $\Rightarrow$ the argument is airtight
- Doesn't compile $\Rightarrow$ there's a gap somewhere
- No "this seems right" — only "this typechecks"

We'll use **Agda** — simultaneously a proof assistant *and* a programming language.

(That's not a coincidence. It'll make sense in a minute.)

Proving Commutativity in Agda (1/5)

Goal

Prove $n + m = m + n$ for *all* natural numbers. Not tested on examples — *proven*.

Start with just the type signature — the *claim*:

```
+-comm : (n m : Nat) -> n + m == m + n
```

Proving Commutativity in Agda (1/5)

Goal

Prove $n + m = m + n$ for *all* natural numbers. Not tested on examples — *proven*.

Start with just the type signature — the *claim*:

```
+-comm : (n m : Nat) -> n + m == m + n
```

- `(n m : Nat)` — for all natural numbers `n` and `m`
- `n + m == m + n` — this equality holds
- This line *is* the theorem statement

Proving Commutativity in Agda (1/5)

Goal

Prove $n + m = m + n$ for *all* natural numbers. Not tested on examples — *proven*.

Start with just the type signature — the *claim*:

```
+-comm : (n m : Nat) -> n + m == m + n
```

- `(n m : Nat)` — for all natural numbers `n` and `m`
- `n + m == m + n` — this equality holds
- This line *is* the theorem statement

Nothing proven yet

Until we implement the body, Agda rejects the file. The claim is typed but unverified.

Proving Commutativity in Agda (2/5)

We prove by induction. First, the base case:

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero m = sym (+-zero m)
```

Proving Commutativity in Agda (2/5)

We prove by induction. First, the base case:

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero m = sym (+-zero m)
```

- Pattern match on `n = zero`
- Need to show: `zero + m == m + zero`
- `+-zero m` is a lemma: `m + zero == m`
- `sym` flips an equality: `a == b` becomes `b == a`

Proving Commutativity in Agda (2/5)

We prove by induction. First, the base case:

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero m = sym (+-zero m)
```

- Pattern match on `n = zero`
- Need to show: `zero + m == m + zero`
- `+-zero m` is a lemma: `m + zero == m`
- `sym` flips an equality: `a == b` becomes `b == a`

Key observation

`sym` and `+-zero` are just *other proofs* — functions we're calling. Building a proof is function composition.

Proving Commutativity in Agda (3/5)

Now add the inductive case (stubbed out):

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero    m = sym (+-zero m)
+-comm (suc n) m = ...
```

Proving Commutativity in Agda (3/5)

Now add the inductive case (stubbed out):

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero    m = sym (+-zero m)
+-comm (suc n) m = ...
```

- `suc n` matches any number ≥ 1
- Need to show: `suc n + m == m + suc n`
- We can call `+-comm n m` recursively to get the hypothesis for n

Proving Commutativity in Agda (3/5)

Now add the inductive case (stubbed out):

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero    m = sym (+-zero m)
+-comm (suc n) m = ...
```

- `suc n` matches any number ≥ 1
- Need to show: `suc n + m == m + suc n`
- We can call `+-comm n m` recursively to get the hypothesis for n

The punchline

The inductive hypothesis is just a recursive call.
Recursion on programs is induction on proofs.

Proving Commutativity in Agda (4/5)

Fill in the inductive case:

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero    m = sym (+-zero m)
+-comm (suc n) m = trans (cong suc (+-comm n m))
                         (sym (+-suc m n))
```

Proving Commutativity in Agda (4/5)

Fill in the inductive case:

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero     m = sym (+-zero m)
+-comm (suc n) m = trans (cong suc (+-comm n m))
                         (sym (+-suc m n))
```

- `+-comm n m` — inductive hypothesis: `n + m == m + n`
- `cong suc` — lift to: `suc(n+m) == suc(m+n)`
- `+-suc m n` — lemma: `m + suc n == suc(m + n)`
- `trans` — chains two equalities end-to-end

Proving Commutativity in Agda (4/5)

Fill in the inductive case:

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero    m = sym (+-zero m)
+-comm (suc n) m = trans (cong suc (+-comm n m))
                         (sym (+-suc m n))
```

- `+-comm n m` — inductive hypothesis: `n + m == m + n`
- `cong suc` — lift to: `suc(n+m) == suc(m+n)`
- `+-suc m n` — lemma: `m + suc n == suc(m + n)`
- `trans` — chains two equalities end-to-end

The pattern

`trans`, `cong`, `sym` are *proof combinators*.
We're assembling a proof out of smaller ones, like lego bricks.

Proving Commutativity in Agda (5/5)

The complete proof:

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero    m = sym (+-zero m)
+-comm (suc n) m = trans (cong suc (+-comm n m))
                         (sym (+-suc m n))
```

Proving Commutativity in Agda (5/5)

The complete proof:

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero    m = sym (+-zero m)
+-comm (suc n) m = trans (cong suc (+-comm n m))
                         (sym (+-suc m n))
```

1. The *type* is the theorem. The *body* is the proof.
2. Induction is recursion. Proof composition is function application.
3. Agda accepting this file means the theorem is *certified*. Not tested. **Proven.**

Proving Commutativity in Agda (5/5)

The complete proof:

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero    m = sym (+-zero m)
+-comm (suc n) m = trans (cong suc (+-comm n m))
                         (sym (+-suc m n))
```

1. The *type* is the theorem. The *body* is the proof.
2. Induction is recursion. Proof composition is function application.
3. Agda accepting this file means the theorem is *certified*. Not tested. **Proven.**

Writing this program is writing a proof.
This isn't a coincidence. It's Curry-Howard.

The Curry-Howard Correspondence

Types Are Propositions

Look at the type signature of our proof again:

```
+-comm : (n m : Nat) -> n + m == m + n
```

Types Are Propositions

Look at the type signature of our proof again:

```
+-comm : (n m : Nat) -> n + m == m + n
```

That's not documentation. It's not a comment.
It *is* the proposition: $\forall\, n\, m \in \mathbb{N},\ n + m = m + n$.

Types Are Propositions

Look at the type signature of our proof again:

```
+-comm : (n m : Nat) -> n + m == m + n
```

That's not documentation. It's not a comment.
It *is* the proposition: $\forall\, n\, m \in \mathbb{N},\ n + m = m + n$.

The function body is the proof.
They are **the same object**, seen from two angles.

Types Are Propositions

Look at the type signature of our proof again:

```
+-comm : (n m : Nat) -> n + m == m + n
```

That's not documentation. It's not a comment.
It *is* the proposition: $\forall\, n\, m \in \mathbb{N},\ n + m = m + n$.

The function body is the proof.
They are **the same object**, seen from two angles.

This generalizes

Every type in Agda is a proposition.
Every program that typechecks is a proof of its type.
The type checker *is* a proof verifier.

The Curry-Howard Correspondence

Logic	Types
Proposition	Type
Proof	Program
True	Inhabited type (has a value)
False	Empty type (no values)
$A \wedge B$	`A * B` (pair)
$A \vee B$	`A + B` (sum type)
$A \Rightarrow B$	`A -> B` (function)
$\forall x.\, P(x)$	Dependent function type

The Curry-Howard Correspondence

Logic	Types
Proposition	Type
Proof	Program
True	Inhabited type (has a value)
False	Empty type (no values)
$A \wedge B$	`A * B` (pair)
$A \vee B$	`A + B` (sum type)
$A \Rightarrow B$	`A -> B` (function)
$\forall x.\, P(x)$	Dependent function type

This is not a metaphor. It is a theorem.

You have been doing logic every time you wrote a type signature.

Curry-Howard: What It Means in Practice

```
+-comm : (n m : Nat) -> n + m == m + n
```

Read as logic: $\forall\, n\, m \in \mathbb{N},\ n + m = m + n$

Curry-Howard: What It Means in Practice

```
+-comm : (n m : Nat) -> n + m == m + n
```

Read as logic: $\forall\, n\, m \in \mathbb{N},\ n + m = m + n$

The type checker is verifying a *proof*, not just a type.
Agda accepting the file *is* the proof being certified.

Curry-Howard: What It Means in Practice

```
+-comm : (n m : Nat) -> n + m == m + n
```

Read as logic: $\forall\, n\, m \in \mathbb{N},\ n + m = m + n$

The type checker is verifying a *proof*, not just a type.
Agda accepting the file *is* the proof being certified.

This is how CompCert works

CompCert is a verified C compiler. Every optimization pass has a type that *is* its correctness proof.

A pass that compiles *is* a proof it preserves program semantics.
Spec and implementation are the same object.

Why Does This Matter?

CompCert

A C compiler where every optimization pass is *formally proven correct*.

CompCert

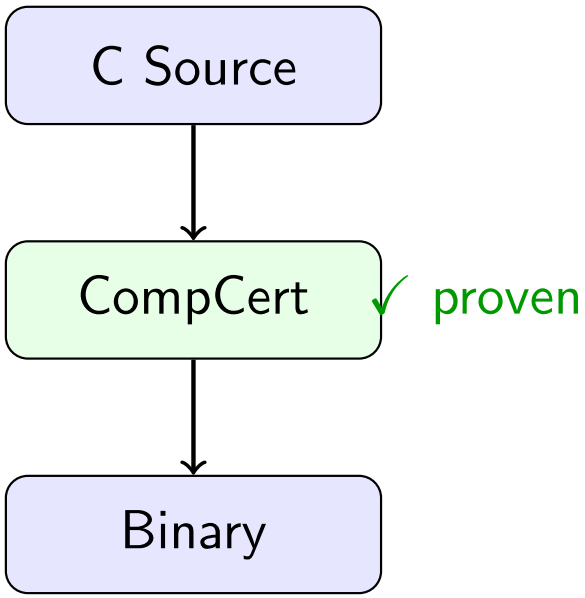
A C compiler where every optimization pass is *formally proven correct*.

- Binary is *provably* equivalent to the source
- Used in aviation and medical devices
- A testing paper found bugs in *every other major C compiler* — but not CompCert

CompCert

A C compiler where every optimization pass is *formally proven correct*.

- Binary is *provably* equivalent to the source
- Used in aviation and medical devices
- A testing paper found bugs in *every other major C compiler* — but not CompCert



The point

The compiler isn't a black box you trust. It's a proof you can inspect.

Amazon & TLA+

Amazon engineers model distributed systems in TLA+ *before* writing code.

Amazon & TLA+

Amazon engineers model distributed systems in TLA+ *before* writing code.

In practice

For DynamoDB, S3, and other core services: write a formal spec first. The model checker exhaustively explores all possible orderings of events.

Amazon & TLA+

Amazon engineers model distributed systems in TLA+ *before* writing code.

In practice

For DynamoDB, S3, and other core services: write a formal spec first. The model checker exhaustively explores all possible orderings of events.

- Found bugs that survived months of design review
- Subtle race conditions invisible to human reasoning

Amazon & TLA+

Amazon engineers model distributed systems in TLA+ *before* writing code.

In practice

For DynamoDB, S3, and other core services: write a formal spec first. The model checker exhaustively explores all possible orderings of events.

- Found bugs that survived months of design review
- Subtle race conditions invisible to human reasoning

Why this works

Distributed systems have too many states to reason about by hand.
A model checker doesn't get tired. It checks *all of them*.

seL4: A Verified OS Kernel

A microkernel whose entire implementation is *proven correct in Coq*.

seL4: A Verified OS Kernel

A microkernel whose entire implementation is *proven correct in Coq*.

What is proven, formally:

- Functional correctness — it does what the spec says
- Security — no unauthorized information flow
- Real-time — bounded response times

seL4: A Verified OS Kernel

A microkernel whose entire implementation is *proven correct in Coq*.

What is proven, formally:

- Functional correctness — it does what the spec says
- Security — no unauthorized information flow
- Real-time — bounded response times

Deployed in military drones, autonomous vehicles, safety-critical infrastructure.

seL4: A Verified OS Kernel

A microkernel whose entire implementation is *proven correct in Coq*.

What is proven, formally:

- Functional correctness — it does what the spec says
- Security — no unauthorized information flow
- Real-time — bounded response times

Deployed in military drones, autonomous vehicles, safety-critical infrastructure.

Why this is hard

An OS kernel mediates everything between software and hardware. seL4 doesn't just test for correctness. It *proves* it.

The Frontier

Where This Is Going

- **AlphaProof** — DeepMind solving IMO problems with certified formal proofs, not approximations

Where This Is Going

- **AlphaProof** — DeepMind solving IMO problems with certified formal proofs, not approximations
- **LLMs + formal verification** — models write proof sketches, verifiers certify them. Creativity from the LLM, certainty from the checker.

Where This Is Going

- **AlphaProof** — DeepMind solving IMO problems with certified formal proofs, not approximations
- **LLMs + formal verification** — models write proof sketches, verifiers certify them. Creativity from the LLM, certainty from the checker.
- **Verified AI** — formally proving safety properties of neural networks before deployment

Where This Is Going

- **AlphaProof** — DeepMind solving IMO problems with certified formal proofs, not approximations
- **LLMs + formal verification** — models write proof sketches, verifiers certify them. Creativity from the LLM, certainty from the checker.
- **Verified AI** — formally proving safety properties of neural networks before deployment
- **The gap is closing** — the Lean community is formalizing all of undergraduate math. Agda and Coq are more usable than ever.

Where This Is Going

- **AlphaProof** — DeepMind solving IMO problems with certified formal proofs, not approximations
- **LLMs + formal verification** — models write proof sketches, verifiers certify them. Creativity from the LLM, certainty from the checker.
- **Verified AI** — formally proving safety properties of neural networks before deployment
- **The gap is closing** — the Lean community is formalizing all of undergraduate math. Agda and Coq are more usable than ever.

Where This Is Going

- **AlphaProof** — DeepMind solving IMO problems with certified formal proofs, not approximations
- **LLMs + formal verification** — models write proof sketches, verifiers certify them. Creativity from the LLM, certainty from the checker.
- **Verified AI** — formally proving safety properties of neural networks before deployment
- **The gap is closing** — the Lean community is formalizing all of undergraduate math. Agda and Coq are more usable than ever.

This field is **active**. The interesting work is happening now.

The Ask

Next time you write a function, ask yourself:

"What is this function's contract?

Could I state it precisely enough

that a machine could check it?"

The Ask

Next time you write a function, ask yourself:

"What is this function's contract?
Could I state it precisely enough
that a machine could check it?"

That question is the first step toward software you can *actually* trust.

The Ask

Next time you write a function, ask yourself:

"What is this function's contract?
Could I state it precisely enough
that a machine could check it?"

That question is the first step toward software you can *actually* trust.

Go further

- **PLFA** — Programming Language Foundations in Agda: `https://plfa.github.io`
- **Z3 tutorial**: `https://ericpony.github.io/z3py-tutorial`
- **Natural Number Game** (Lean 4): `https://adam.math.hhu.de`
- **TLA+** — Leslie Lamport's video course, free on YouTube

Get Hype!!! Happy break + more to come!