Automatic Theorem Provers 2 + Dependent Types

Hype for Types

March 9, 2026

Review: Last Week

SAT: Why It Exists

The problem: hardware verification requires checking every possible input. A 64-bit circuit has 2^{64} cases. Testing can't cover them.

SAT: Why It Exists

The problem: hardware verification requires checking every possible input. A 64-bit circuit has 2^{64} cases. Testing can't cover them.

SAT reframes the question: instead of checking all cases, ask "is there *any* assignment that makes this formula true?"

SAT: Why It Exists

The problem: hardware verification requires checking every possible input. A 64-bit circuit has 2^{64} cases. Testing can't cover them.

SAT reframes the question: instead of checking all cases, ask "is there *any* assignment that makes this formula true?"

Example

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

SAT: Why It Exists

The problem: hardware verification requires checking every possible input. A 64-bit circuit has 2^{64} cases. Testing can't cover them.

SAT reframes the question: instead of checking all cases, ask "is there *any* assignment that makes this formula true?"

Example

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

Assignment $A{=}T,\ B{=}F,\ C{=}T$: $(T) \wedge (T) \wedge (T) = \mathsf{T}$ ✓ satisfiable.

SAT: Why It Exists

The problem: hardware verification requires checking every possible input. A 64-bit circuit has 2^{64} cases. Testing can't cover them.

SAT reframes the question: instead of checking all cases, ask "is there *any* assignment that makes this formula true?"

Example

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

Assignment $A{=}T$, $B{=}F$, $C{=}T$: $(T) \wedge (T) \wedge (T) = \mathsf{T}$ ✓ satisfiable.

If no assignment satisfies it: **UNSAT** — and that's a proof.

SAT: Why It Exists

The problem: hardware verification requires checking every possible input. A 64-bit circuit has 2^{64} cases. Testing can't cover them.

SAT reframes the question: instead of checking all cases, ask "is there *any* assignment that makes this formula true?"

Example

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

Assignment $A{=}T$, $B{=}F$, $C{=}T$: $(T) \wedge (T) \wedge (T) = \mathrm{T}$ ✓ satisfiable.

If no assignment satisfies it: **UNSAT** — and that's a proof.

UNSAT is verification

Ask: "is there any input where property P fails?"

UNSAT $\Rightarrow$ P holds for *all* inputs. Automatically.

DPLL: How SAT Doesn't Explode

The problem: 2^n assignments. For $n = 100$, naive search is hopeless.

DPLL: How SAT Doesn't Explode

The problem: 2^n assignments. For $n = 100$, naive search is hopeless.

Unit propagation: forced assignments

Clauses: $(\neg A \vee B)$ and (A).

DPLL: How SAT Doesn't Explode

The problem: 2^n assignments. For $n = 100$, naive search is hopeless.

Unit propagation: forced assignments

Clauses: $(\neg A \vee B)$ and (A).
(A) is unit: A must be true. No choice.

DPLL: How SAT Doesn't Explode

The problem: 2^n assignments. For $n = 100$, naive search is hopeless.

Unit propagation: forced assignments

Clauses: $(\neg A \vee B)$ and (A).
(A) is unit: A must be true. No choice.
$(\neg A \vee B)$ becomes $(F \vee B) = (B)$: B must be true too.
One forced assignment cascaded into another. Zero guessing.

DPLL: How SAT Doesn't Explode

The problem: 2^n assignments. For $n = 100$, naive search is hopeless.

Unit propagation: forced assignments

Clauses: $(\neg A \vee B)$ and (A).
(A) is unit: A must be true. No choice.
$(\neg A \vee B)$ becomes $(F \vee B) = (B)$: B must be true too.
One forced assignment cascaded into another. Zero guessing.

Backtracking: when forced assignments run out, guess. If contradiction arises, undo and try the other branch.

DPLL: How SAT Doesn't Explode

The problem: 2^n assignments. For $n = 100$, naive search is hopeless.

Unit propagation: forced assignments

Clauses: $(\neg A \vee B)$ and (A).
(A) is unit: A must be true. No choice.
$(\neg A \vee B)$ becomes $(F \vee B) = (B)$: B must be true too.
One forced assignment cascaded into another. Zero guessing.

Backtracking: when forced assignments run out, guess. If contradiction arises, undo and try the other branch.

Conflict-driven clause learning (CDCL): modern solvers record *why* each conflict happened, so they never revisit the same mistake from any direction.

DPLL: How SAT Doesn't Explode

The problem: 2^n assignments. For $n = 100$, naive search is hopeless.

Unit propagation: forced assignments

Clauses: $(\neg A \vee B)$ and (A).
(A) is unit: A must be true. No choice.
$(\neg A \vee B)$ becomes $(F \vee B) = (B)$: B must be true too.
One forced assignment cascaded into another. Zero guessing.

Backtracking: when forced assignments run out, guess. If contradiction arises, undo and try the other branch.

Conflict-driven clause learning (CDCL): modern solvers record *why* each conflict happened, so they never revisit the same mistake from any direction.

The result

NP-complete in theory. Millions of variables in seconds in practice.

SMT: SAT Over Real Programs

The problem: programs use integers, arrays, strings. SAT can't reason about `x + y > 10`.

SMT: SAT Over Real Programs

The problem: programs use integers, arrays, strings. SAT can't reason about `x + y > 10`.

SMT (Satisfiability Modulo Theories): SAT engine + theory solvers, cooperating.

SMT: SAT Over Real Programs

The problem: programs use integers, arrays, strings. SAT can't reason about `x + y > 10`.

SMT (Satisfiability Modulo Theories): SAT engine + theory solvers, cooperating.

The architecture (DPLL(T))

Formula: $(x > 0) \ \wedge \ (y < x) \ \wedge \ (y \geq 5)$

SMT: SAT Over Real Programs

The problem: programs use integers, arrays, strings. SAT can't reason about `x + y > 10`.

SMT (Satisfiability Modulo Theories): SAT engine + theory solvers, cooperating.

The architecture (DPLL(T))

Formula: $(x > 0) \ \wedge \ (y < x) \ \wedge \ (y \geq 5)$

SAT picks truth values for the atoms. Theory solver checks: can $x > 0$, $y < x$, $y \geq 5$ hold simultaneously?

SMT: SAT Over Real Programs

The problem: programs use integers, arrays, strings. SAT can't reason about `x + y > 10`.

SMT (Satisfiability Modulo Theories): SAT engine + theory solvers, cooperating.

The architecture (DPLL(T))

Formula: $(x > 0) \ \wedge \ (y < x) \ \wedge \ (y \geq 5)$

SAT picks truth values for the atoms. Theory solver checks: can $x > 0$, $y < x$, $y \geq 5$ hold simultaneously?

Yes: $x = 6, y = 5$. **sat.** If the theory said no: conflict clause $\rightarrow$ SAT backtracks.

SMT: SAT Over Real Programs

The problem: programs use integers, arrays, strings. SAT can't reason about `x + y > 10`.

SMT (Satisfiability Modulo Theories): SAT engine + theory solvers, cooperating.

The architecture (DPLL(T))

Formula: $(x > 0) \ \wedge \ (y < x) \ \wedge \ (y \geq 5)$

SAT picks truth values for the atoms. Theory solver checks: can $x > 0$, $y < x$, $y \geq 5$ hold simultaneously?

Yes: $x = 6, y = 5$. **sat.** If the theory said no: conflict clause $\rightarrow$ SAT backtracks.

Z3 (Microsoft Research)

The dominant SMT solver. Handles integers, arrays, bitvectors, strings.

Ask: "is there any input where this property fails?"

`unsat` = proved correct for all inputs. `sat` = here's your counterexample.

Agda: Why It Exists

The problem: SMT works for bounded, finite properties. It can't prove "this sort is correct for every possible list." It doesn't do unbounded induction.

Agda: Why It Exists

The problem: SMT works for bounded, finite properties. It can't prove "this sort is correct for every possible list." It doesn't do unbounded induction.

Proof assistants: you write the proof; the machine checks every step. Handles universal claims.

Agda: Why It Exists

The problem: SMT works for bounded, finite properties. It can't prove "this sort is correct for every possible list." It doesn't do unbounded induction.

Proof assistants: you write the proof; the machine checks every step. Handles universal claims.

The simplest example

```
-- The TYPE is the claim: for any Nat n, n + 0 equals n
+-zero : (n : Nat) -> n + 0 == n
+-zero zero     = refl                          -- 0 + 0 = 0, by
    definition
+-zero (suc n) = cong suc (+-zero n)    -- inductive step
```

Agda: Why It Exists

The problem: SMT works for bounded, finite properties. It can't prove "this sort is correct for every possible list." It doesn't do unbounded induction.

Proof assistants: you write the proof; the machine checks every step. Handles universal claims.

The simplest example

```
-- The TYPE is the claim: for any Nat n, n + 0 equals n
+-zero : (n : Nat) -> n + 0 == n
+-zero zero     = refl                          -- 0 + 0 = 0, by
    definition
+-zero (suc n) = cong suc (+-zero n)    -- inductive step
```

The type is the claim. The body is the proof.
Agda accepting this = machine-certified for **all** naturals. Not just 0 through 100.

Agda: Why It Exists

The problem: SMT works for bounded, finite properties. It can't prove "this sort is correct for every possible list." It doesn't do unbounded induction.

Proof assistants: you write the proof; the machine checks every step. Handles universal claims.

The simplest example

```
-- The TYPE is the claim: for any Nat n, n + 0 equals n
+-zero : (n : Nat) -> n + 0 == n
+-zero zero     = refl                          -- 0 + 0 = 0, by
    definition
+-zero (suc n) = cong suc (+-zero n)    -- inductive step
```

The type is the claim. The body is the proof.
Agda accepting this = machine-certified for **all** naturals. Not just 0 through 100.

The key difference

Tests check $n = 0, 1, 2, \ldots$ and hope. This proves it for every n. No exceptions.

The +-comm Proof: Why It's Non-Trivial

Why does $n + m = m + n$ need work? Look at how + is defined:

```
zero  + m = m                 -- 0 + m just returns m
suc n + m = suc (n + m)       -- (1+n) + m peels off the suc and
    recurses
```

The +-comm Proof: Why It's Non-Trivial

Why does $n + m = m + n$ need work? Look at how + is defined:

```
zero    + m = m                 -- 0 + m just returns m
suc n + m = suc (n + m)   -- (1+n) + m peels off the suc and
    recurses
```

The asymmetry

`0 + m`: left rule fires. Reduces to `m` by definition. **Free.**

The +-comm Proof: Why It's Non-Trivial

Why does $n + m = m + n$ need work? Look at how + is defined:

```
zero    + m = m                 -- 0 + m just returns m
suc n + m = suc (n + m)         -- (1+n) + m peels off the suc and
    recurses
```

The asymmetry

`0 + m`: left rule fires. Reduces to `m` by definition. **Free.**

`m + 0`: `m` is a variable. Neither rule fires. **Stuck.**

The +-comm Proof: Why It's Non-Trivial

Why does $n + m = m + n$ need work? Look at how + is defined:

```
zero  + m = m                -- 0 + m just returns m
suc n + m = suc (n + m)      -- (1+n) + m peels off the suc and
    recurses
```

The asymmetry

`0 + m`: left rule fires. Reduces to `m` by definition. **Free.**

`m + 0`: `m` is a variable. Neither rule fires. **Stuck.**

`m + 0 == m` cannot be proved by `refl` — it needs induction on `m`.

The +-comm Proof: Why It's Non-Trivial

Why does $n + m = m + n$ need work? Look at how + is defined:

```
zero   + m = m                  -- 0 + m just returns m
suc n + m = suc (n + m)         -- (1+n) + m peels off the suc and
    recurses
```

The asymmetry

`0 + m`: left rule fires. Reduces to `m` by definition. **Free.**

`m + 0`: `m` is a variable. Neither rule fires. **Stuck.**

`m + 0 == m` cannot be proved by `refl` — it needs induction on `m`.

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero     m = sym (+-zero m)              -- use the +-zero
    lemma
+-comm (suc n) m = trans (cong suc (+-comm n m))
                         (sym (+-suc m n)) -- inductive step
```

The +-comm Proof: Why It's Non-Trivial

Why does $n + m = m + n$ need work? Look at how + is defined:

```
zero  + m = m                -- 0 + m just returns m
suc n + m = suc (n + m)      -- (1+n) + m peels off the suc and
    recurses
```

The asymmetry

`0 + m`: left rule fires. Reduces to `m` by definition. **Free.**

`m + 0`: `m` is a variable. Neither rule fires. **Stuck.**

`m + 0 == m` cannot be proved by `refl` — it needs induction on `m`.

```
+-comm : (n m : Nat) -> n + m == m + n
+-comm zero    m = sym (+-zero m)              -- use the +-zero
    lemma
+-comm (suc n) m = trans (cong suc (+-comm n m))
                         (sym (+-suc m n)) -- inductive step
```

The recursive call `+-comm n m` *is* the inductive hypothesis. Induction is just recursion.

Curry-Howard: The Connection

The observation: logic and type theory are the same formal system, viewed differently.

Curry-Howard: The Connection

The observation: logic and type theory are the same formal system, viewed differently.

`sym` is both a function and a logical rule

```
sym :  a == b -> b == a
```

Curry-Howard: The Connection

The observation: logic and type theory are the same formal system, viewed differently.

`sym` is both a function and a logical rule

```
sym :  a == b -> b == a
```

As a function: give it a proof of $a = b$, get a proof of $b = a$.
As a logical rule: from $a = b$ derive $b = a$.

Curry-Howard: The Connection

The observation: logic and type theory are the same formal system, viewed differently.

`sym` is both a function and a logical rule

```
sym :  a == b -> b == a
```

As a function: give it a proof of $a = b$, get a proof of $b = a$.
As a logical rule: from $a = b$ derive $b = a$.
These are not analogous. They are **the same object.**

Curry-Howard: The Connection

The observation: logic and type theory are the same formal system, viewed differently.

`sym` is both a function and a logical rule

```
sym :  a == b -> b == a
```

As a function: give it a proof of $a = b$, get a proof of $b = a$.
As a logical rule: from $a = b$ derive $b = a$.
These are not analogous. They are **the same object.**

Logic	**Types**	**Example**
Proposition	Type	`n + m == m + n`
Proof	Program	body of `+-comm`
True	Inhabited type	`Nat`, `Bool`
$P \Rightarrow Q$	`P -> Q`	`sym`, `trans`
$\forall x.\ P(x)$	`(x:A)->P x`	`+-comm`

Curry-Howard: The Connection

The observation: logic and type theory are the same formal system, viewed differently.

`sym` is both a function and a logical rule

```
sym :  a == b -> b == a
```

As a function: give it a proof of $a = b$, get a proof of $b = a$.
As a logical rule: from $a = b$ derive $b = a$.
These are not analogous. They are **the same object.**

Logic	**Types**	**Example**
Proposition	Type	`n + m == m + n`
Proof	Program	body of `+-comm`
True	Inhabited type	`Nat`, `Bool`
$P \Rightarrow Q$	`P -> Q`	`sym`, `trans`
$\forall x.\ P(x)$	`(x:A)->P x`	`+-comm`

Every row is a function you've already written. The table just tells you what it means logically.

What Regular Types Can and Cannot Say

What a Type Actually Is

A type is a set of allowed values. The type checker verifies membership.

What a Type Actually Is

A type is a set of allowed values. The type checker verifies membership.

In any mainstream language

```
def double(x: int) -> int:
    return x * 2
```

What a Type Actually Is

A type is a set of allowed values. The type checker verifies membership.

In any mainstream language

```
def double(x: int) -> int:
    return x * 2
```

`int -> int`: values from the set of integers in; values from the set of integers out.
The type checker asks: is `x` an integer? Is the return value an integer?

What a Type Actually Is

A type is a set of allowed values. The type checker verifies membership.

In any mainstream language

```
def double(x: int) -> int:
    return x * 2
```

`int -> int`: values from the set of integers in; values from the set of integers out.
The type checker asks: is `x` an integer? Is the return value an integer?
It does not ask: is the output $2 \times x$? Is it positive? Is it even related to the input?

What a Type Actually Is

A type is a set of allowed values. The type checker verifies membership.

In any mainstream language

```
def double(x: int) -> int:
    return x * 2
```

`int -> int`: values from the set of integers in; values from the set of integers out.
The type checker asks: is `x` an integer? Is the return value an integer?
It does not ask: is the output $2 \times x$? Is it positive? Is it even related to the input?

The thing types check: kind

Simple types answer "what kind of value?" — integer, string, list, function.
They are completely silent about *which* value, or what's true about it.

The Same Problem, Everywhere

Sort

```
def sort(xs: list[int]) -> list[int]: ...
```

This type is satisfied by a correct sort, by `lambda xs: []`, by reversing — anything.

The Same Problem, Everywhere

Sort

```
def sort(xs: list[int]) -> list[int]: ...
```

This type is satisfied by a correct sort, by `lambda xs: []`, by reversing — anything.

Function	**What the type can't say**
`head(xs)`	`xs` is non-empty
`lookup(arr, i)`	`i` is within bounds
`zip(xs, ys)`	`xs` and `ys` have equal length
`sort(xs)`	output is sorted and a permutation of input
`encrypt(k, m)`	`k` has not been used before

The Same Problem, Everywhere

Sort

```
def sort(xs: list[int]) -> list[int]: ...
```

This type is satisfied by a correct sort, by `lambda xs: []`, by reversing — anything.

Function	**What the type can't say**
`head(xs)`	`xs` is non-empty
`lookup(arr, i)`	`i` is within bounds
`zip(xs, ys)`	`xs` and `ys` have equal length
`sort(xs)`	output is sorted and a permutation of input
`encrypt(k, m)`	`k` has not been used before

The pattern

Every one of these is a condition on a *value*, not on its kind.
The type "list of integers" says nothing about length, order, or membership.
You need a type that can reference the actual value.

Where There Is Room to Change

Simple types: `List[int]` — the set of all lists of integers.

What if the set could be defined by a condition on a value?

Where There Is Room to Change

Simple types: `List[int]` — the set of all lists of integers.
What if the set could be defined by a condition on a value?

Today: structure only

- `int`: any integer
- `List[int]`: any such list
- `int -> int`: any such function

The value is irrelevant to the type.
Types and values live in separate worlds.

What we could express

- an integer *that is positive*
- a list *of length exactly n*
- a function *that returns sorted output*

The type mentions a value.
Values and types talk to each other.

Where There Is Room to Change

Simple types: `List[int]` — the set of all lists of integers.
What if the set could be defined by a condition on a value?

Today: structure only

- `int`: any integer
- `List[int]`: any such list
- `int -> int`: any such function

The value is irrelevant to the type.
Types and values live in separate worlds.

What we could express

- an integer *that is positive*
- a list *of length exactly n*
- a function *that returns sorted output*

The type mentions a value.
Values and types talk to each other.

The gap that dependent types fill

In a simple type system, there is no slot in the type language to write "length = 5" or "output is sorted." Dependent types add that slot. The type can now mention a specific value.

Where There Is No Room to Change

Not everything is open. Some constraints are load-bearing for the whole system.

Where There Is No Room to Change

Not everything is open. Some constraints are load-bearing for the whole system.

Fixed: decidability

Type checking must terminate.
A type whose membership requires solving the halting problem is not allowed.
Agda's termination checker enforces this.

Fixed: checked at compile time

If checking required running the program, you'd lose the whole benefit.

Fixed: soundness

The type system must never lie.
If it certifies a value has type T, it must actually have type T.

Open: expressiveness

The *language* of types can be extended to mention values.
This is the only thing dependent types change.

Where There Is No Room to Change

Not everything is open. Some constraints are load-bearing for the whole system.

Fixed: decidability

Type checking must terminate.
A type whose membership requires solving the halting problem is not allowed.
Agda's termination checker enforces this.

Fixed: checked at compile time

If checking required running the program, you'd lose the whole benefit.

Fixed: soundness

The type system must never lie.
If it certifies a value has type T, it must actually have type T.

Open: expressiveness

The *language* of types can be extended to mention values.
This is the only thing dependent types change.

The dependent types move in one sentence

Allow the type language to reference values from the value language.
Keep termination, compile-time checking, and soundness intact.

Dependent Types

What Is a Dependent Type?

In every language you've used, types and values are separate things. A type describes what *kind* of value something is. The actual value is irrelevant to the type.

What Is a Dependent Type?

In every language you've used, types and values are separate things. A type describes what *kind* of value something is. The actual value is irrelevant to the type.

A dependent type is a type that mentions a specific value.

What Is a Dependent Type?

In every language you've used, types and values are separate things. A type describes what *kind* of value something is. The actual value is irrelevant to the type.

A dependent type is a type that mentions a specific value.

The idea in one line

```
Vec A n    -- a list of exactly n elements of type A
```

What Is a Dependent Type?

In every language you've used, types and values are separate things. A type describes what *kind* of value something is. The actual value is irrelevant to the type.

A dependent type is a type that mentions a specific value.

The idea in one line

```
Vec A n    -- a list of exactly n elements of type A
```

`n` is not a type variable. It's a real `Nat` value sitting inside the type.
`Vec A 3` and `Vec A 4` are **different types** — just like `Int` and `String` are.

What Is a Dependent Type?

In every language you've used, types and values are separate things. A type describes what *kind* of value something is. The actual value is irrelevant to the type.

A dependent type is a type that mentions a specific value.

The idea in one line

```
Vec A n    -- a list of exactly n elements of type A
```

`n` is not a type variable. It's a real `Nat` value sitting inside the type.
`Vec A 3` and `Vec A 4` are **different types** — just like `Int` and `String` are.
The type *depends on* the value `n`. That's where the name comes from.

What Is a Dependent Type?

In every language you've used, types and values are separate things. A type describes what *kind* of value something is. The actual value is irrelevant to the type.

A dependent type is a type that mentions a specific value.

The idea in one line

```
Vec A n    -- a list of exactly n elements of type A
```

`n` is not a type variable. It's a real `Nat` value sitting inside the type.
`Vec A 3` and `Vec A 4` are **different types** — just like `Int` and `String` are.
The type *depends on* the value `n`. That's where the name comes from.

Why this is new

`List[int]` says: a list of integers. Silent about length, order, contents.
`Vec Int 5` says: a list of *exactly five* integers. The type carries a guarantee.

What Is a Dependent Type?

In every language you've used, types and values are separate things. A type describes what *kind* of value something is. The actual value is irrelevant to the type.

A dependent type is a type that mentions a specific value.

The idea in one line

```
Vec A n    -- a list of exactly n elements of type A
```

`n` is not a type variable. It's a real `Nat` value sitting inside the type.
`Vec A 3` and `Vec A 4` are **different types** — just like `Int` and `String` are.
The type *depends on* the value `n`. That's where the name comes from.

Why this is new

`List[int]` says: a list of integers. Silent about length, order, contents.
`Vec Int 5` says: a list of *exactly five* integers. The type carries a guarantee.
The type checker can now enforce facts about values, not just kinds.

Phantom Types in SML: The Idea

Goal: encode the length of a list *in its type*, using only what SML gives us.

Phantom Types in SML: The Idea

Goal: encode the length of a list *in its type*, using only what SML gives us.

Tag the type with a phantom parameter

```
(* Dummy types -- no values, just labels *)
datatype zero  = ZERO_  (* never constructed *)
datatype 'n succ = SUCC_ (* never constructed *)

(* 'a vlist carries a phantom 'n tracking length *)
datatype ('a, 'n) vlist =
    Nil  : ('a, zero) vlist
  | Cons : 'a * ('a, 'n) vlist -> ('a, 'n succ) vlist
```

Phantom Types in SML: The Idea

Goal: encode the length of a list *in its type*, using only what SML gives us.

Tag the type with a phantom parameter

```
(* Dummy types -- no values, just labels *)
datatype zero  = ZERO_  (* never constructed *)
datatype 'n succ = SUCC_ (* never constructed *)

(* 'a vlist carries a phantom 'n tracking length *)
datatype ('a, 'n) vlist =
    Nil  : ('a, zero) vlist
  | Cons : 'a * ('a, 'n) vlist -> ('a, 'n succ) vlist
```

`Nil` has type `('a, zero) vlist`.

`Cons(1, Nil)` has type `(int, zero succ) vlist`.

Phantom Types in SML: The Idea

Goal: encode the length of a list *in its type*, using only what SML gives us.

Tag the type with a phantom parameter

```
(* Dummy types -- no values, just labels *)
datatype zero  = ZERO_  (* never constructed *)
datatype 'n succ = SUCC_ (* never constructed *)

(* 'a vlist carries a phantom 'n tracking length *)
datatype ('a, 'n) vlist =
    Nil  : ('a, zero) vlist
  | Cons : 'a * ('a, 'n) vlist -> ('a, 'n succ) vlist
```

`Nil` has type `('a, zero) vlist`.

`Cons(1, Nil)` has type `(int, zero succ) vlist`.

The length tag lives entirely in the type. At runtime, `zero` and `succ` vanish.

Phantom Types in SML: The Idea

Goal: encode the length of a list *in its type*, using only what SML gives us.

Tag the type with a phantom parameter

```
(* Dummy types -- no values, just labels *)
datatype zero  = ZERO_  (* never constructed *)
datatype 'n succ = SUCC_ (* never constructed *)

(* 'a vlist carries a phantom 'n tracking length *)
datatype ('a, 'n) vlist =
    Nil  : ('a, zero) vlist
  | Cons : 'a * ('a, 'n) vlist -> ('a, 'n succ) vlist
```

`Nil` has type `('a, zero) vlist`.
`Cons(1, Nil)` has type `(int, zero succ) vlist`.
The length tag lives entirely in the type. At runtime, `zero` and `succ` vanish.

Now `head` can demand non-empty

```
fun head (Cons(x, _) : ('a, 'n succ) vlist) = x
```

Passing `Nil` is a type error — `zero` $\neq$ `'n succ`.
This *looks* like a dependent type. It isn't. We'll see why.

Phantom Types: Where the Illusion Breaks

The problem: the tag is decoration. SML can't compute with it.

Phantom Types: Where the Illusion Breaks

The problem: the tag is decoration. SML can't compute with it.

append — what type does it return?

```
(* We want: ('a, 'm) vlist -> ('a, 'n) vlist
                              -> ('a, 'm + 'n) vlist *)
(* But 'm + 'n is not a type expression in SML. *)
(* The best we can write: *)
fun append Nil          ys = ys
  | append (Cons(x,xs)) ys = Cons(x, append xs ys)
(* Return type? SML infers ('a, 'n succ) vlist ---
   it loses the arithmetic entirely.                *)
```

Phantom Types: Where the Illusion Breaks

The problem: the tag is decoration. SML can't compute with it.

append — what type does it return?

```
(* We want: ('a, 'm) vlist -> ('a, 'n) vlist
                              -> ('a, 'm + 'n) vlist *)
(* But 'm + 'n is not a type expression in SML. *)
(* The best we can write: *)
fun append Nil          ys = ys
  | append (Cons(x,xs)) ys = Cons(x, append xs ys)
(* Return type? SML infers ('a, 'n succ) vlist ---
   it loses the arithmetic entirely.                *)
```

`'m + 'n` is not expressible as a type in SML.

The type checker cannot add phantom tags. It can only *match* them.

Phantom Types: Where the Illusion Breaks

The problem: the tag is decoration. SML can't compute with it.

append — what type does it return?

```
(* We want: ('a, 'm) vlist -> ('a, 'n) vlist
                              -> ('a, 'm + 'n) vlist *)
(* But 'm + 'n is not a type expression in SML. *)
(* The best we can write: *)
fun append Nil          ys = ys
  | append (Cons(x,xs)) ys = Cons(x, append xs ys)
(* Return type? SML infers ('a, 'n succ) vlist ---
   it loses the arithmetic entirely.                *)
```

`'m + 'n` is not expressible as a type in SML.

The type checker cannot add phantom tags. It can only *match* them.

The fundamental wall

Phantom types can enforce *fixed* structural constraints (non-empty, same tag).

They cannot express *computed* relationships like $m + n$, $n - 1$, or $n < k$.

The tags are inert. You can't do arithmetic on types in SML.

The Gap: Types and Values Are Separate Worlds

Why can't SML do what we want? The root cause:

The Gap: Types and Values Are Separate Worlds

Why can't SML do what we want? The root cause:

SML: two separate universes

Types live at compile time.
Values live at runtime.

`zero` as a *type*: a compile-time label.
`0` as a *value*: a runtime integer.

They are completely unrelated.
You cannot use a value inside a type.
You cannot compute a type from a value.

Agda: one universe

`Nat` is a type *and* its values are usable inside other types.

`Vec A 3`: the 3 here is a genuine `Nat` value sitting inside a type.

The type checker can evaluate `m + n`.
It knows `Vec A (2 + 3) = Vec A 5`.
The arithmetic is real.

The Gap: Types and Values Are Separate Worlds

Why can't SML do what we want? The root cause:

SML: two separate universes

Types live at compile time.
Values live at runtime.

`zero` as a *type*: a compile-time label.
`0` as a *value*: a runtime integer.

They are completely unrelated.
You cannot use a value inside a type.
You cannot compute a type from a value.

Agda: one universe

`Nat` is a type *and* its values are usable inside other types.

`Vec A 3`: the 3 here is a genuine `Nat` value sitting inside a type.

The type checker can evaluate `m + n`.
It knows `Vec A (2 + 3) = Vec A 5`.
The arithmetic is real.

The dependent types move — precisely stated

Collapse the wall between the type universe and the value universe.
Let types be indexed by *actual values*. Then `Vec A n` means what it says.

So What's the Actual Difference?

Both phantom types and dependent types let you write `head` safely. The difference shows up the moment you try to *compute* with the index.

So What's the Actual Difference?

Both phantom types and dependent types let you write `head` safely. The difference shows up the moment you try to *compute* with the index.

Phantom type — a label

`'n` is a compile-time tag.
`zero succ` means "tagged once."
It is *not* the number 1.

The type checker can match tags.
It **cannot** add them.

```
(* can't write this in SML *)
'm + 'n
```

`'m + 'n` is not a type expression.

Dependent type — a value

`n` is a real `Nat`.
3 means the number 3.
It *is* the value.

The type checker can evaluate.
It **can** compute `m + n`.

```
-- this typechecks in Agda
Vec A (m + n)
```

`Vec A (2+3) = Vec A 5`.

So What's the Actual Difference?

Both phantom types and dependent types let you write `head` safely. The difference shows up the moment you try to *compute* with the index.

Phantom type — a label

`'n` is a compile-time tag.
`zero succ` means "tagged once."
It is *not* the number 1.

The type checker can match tags.
It **cannot** add them.

```
(* can't write this in SML *)
'm + 'n
```

`'m + 'n` is not a type expression.

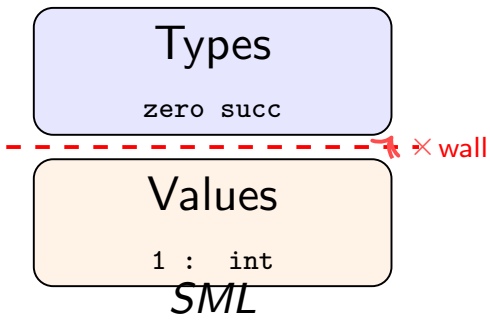
Dependent type — a value

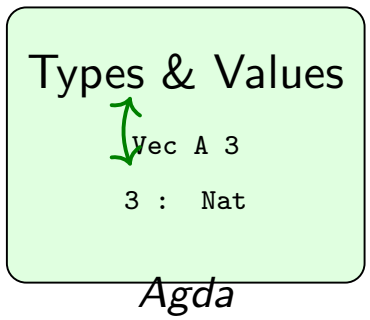
`n` is a real `Nat`.
`3` means the number 3.
It *is* the value.

The type checker can evaluate.
It **can** compute `m + n`.

```
-- this typechecks in Agda
Vec A (m + n)
```

`Vec A (2+3) = Vec A 5`.





Vec: A Type That Carries a Value

`head` on an empty list crashes. That's a runtime error we've accepted forever. With dependent types, it becomes a compile-time type error instead.

Vec: A Type That Carries a Value

`head` on an empty list crashes. That's a runtime error we've accepted forever. With dependent types, it becomes a compile-time type error instead.

Vec A n: a list of *exactly* n elements, where n is baked into the type itself.

```
data Vec (A : Set) : Nat -> Set where
  []    : Vec A zero                  -- empty list has length
      0
  _::_  : A -> Vec A n -> Vec A (suc n)   -- cons adds
      exactly 1
```

Vec: A Type That Carries a Value

`head` on an empty list crashes. That's a runtime error we've accepted forever. With dependent types, it becomes a compile-time type error instead.

Vec A n: a list of *exactly* n elements, where n is baked into the type itself.

```
data Vec (A : Set) : Nat -> Set where
  []    : Vec A zero                    -- empty list has length
      0
  _::_  : A -> Vec A n -> Vec A (suc n)  -- cons adds
      exactly 1
```

Now head is total

```
head : Vec A (suc n) -> A    -- only callable on length >=
    1
```

Vec: A Type That Carries a Value

`head` on an empty list crashes. That's a runtime error we've accepted forever. With dependent types, it becomes a compile-time type error instead.

Vec A n: a list of *exactly* n elements, where n is baked into the type itself.

```
data Vec (A : Set) : Nat -> Set where
  []    : Vec A zero                    -- empty list has length 0
  _::_  : A -> Vec A n -> Vec A (suc n)  -- cons adds exactly 1
```

Now head is total

```
head : Vec A (suc n) -> A    -- only callable on length >= 1
```

`Vec A zero` does not unify with `Vec A (suc n)`.
Passing an empty vector is a **compile-time type error.**
Not a Maybe. Not a crash. The bad call is *inexpressible*.

Vec: Relationships Between Lengths

The real payoff: once lengths live in types, you can link them across arguments.

Vec: Relationships Between Lengths

The real payoff: once lengths live in types, you can link them across arguments.

zip enforces matching lengths

```
zip : Vec A n -> Vec B n -> Vec (A * B) n
```

Vec: Relationships Between Lengths

The real payoff: once lengths live in types, you can link them across arguments.

zip enforces matching lengths

```
zip : Vec A n -> Vec B n -> Vec (A * B) n
```

Both arguments must have the same `n`.
Zipping a length-3 and a length-4 list: the `n`s don't unify. **Type error at the call site.**

Vec: Relationships Between Lengths

The real payoff: once lengths live in types, you can link them across arguments.

zip enforces matching lengths

```
zip : Vec A n -> Vec B n -> Vec (A * B) n
```

Both arguments must have the same `n`.
Zipping a length-3 and a length-4 list: the `n`s don't unify. **Type error at the call site.**
Output is exactly length `n`. The type proves it — no test needed.

Vec: Relationships Between Lengths

The real payoff: once lengths live in types, you can link them across arguments.

zip enforces matching lengths

```
zip : Vec A n -> Vec B n -> Vec (A * B) n
```

Both arguments must have the same `n`.
Zipping a length-3 and a length-4 list: the `n`s don't unify. **Type error at the call site.**
Output is exactly length `n`. The type proves it — no test needed.

append adds lengths arithmetically

```
append : Vec A m -> Vec A n -> Vec A (m + n)
```

Vec: Relationships Between Lengths

The real payoff: once lengths live in types, you can link them across arguments.

zip enforces matching lengths

```
zip : Vec A n -> Vec B n -> Vec (A * B) n
```

Both arguments must have the same `n`.
Zipping a length-3 and a length-4 list: the `n`s don't unify. **Type error at the call site.**
Output is exactly length `n`. The type proves it — no test needed.

append adds lengths arithmetically

```
append : Vec A m -> Vec A n -> Vec A (m + n)
```

Return type is `m + n`. The type checker verifies the arithmetic.
A buggy append that silently drops elements would fail to typecheck.

Fin: Making the Constraint the Type

Every language has runtime bounds checks on array access. What if the type made an out-of-bounds index *impossible to construct*?

Fin: Making the Constraint the Type

Every language has runtime bounds checks on array access. What if the type made an out-of-bounds index *impossible to construct*?

Fin n: the type of natural numbers strictly less than n.

```
data Fin : Nat -> Set where
  zero : Fin (suc n)              -- 0 is valid for any length
       >= 1
  suc  : Fin n -> Fin (suc n) -- if k < n, then k+1 < n+1
```

Fin: Making the Constraint the Type

Every language has runtime bounds checks on array access. What if the type made an out-of-bounds index *impossible to construct*?

Fin n: the type of natural numbers strictly less than n.

```
data Fin : Nat -> Set where
  zero : Fin (suc n)              -- 0 is valid for any length
       >= 1
  suc  : Fin n -> Fin (suc n)     -- if k < n, then k+1 < n+1
```

`Fin 3` contains exactly: 0, 1, 2. `Fin 0` is empty.

Fin: Making the Constraint the Type

Every language has runtime bounds checks on array access. What if the type made an out-of-bounds index *impossible to construct*?

Fin n: the type of natural numbers strictly less than n.

```
data Fin : Nat -> Set where
  zero : Fin (suc n)              -- 0 is valid for any length
       >= 1
  suc  : Fin n -> Fin (suc n) -- if k < n, then k+1 < n+1
```

Fin 3 contains exactly: 0, 1, 2. **Fin 0** is empty.

```
lookup : Vec A n -> Fin n -> A     -- no bounds check
    needed
```

Fin: Making the Constraint the Type

Every language has runtime bounds checks on array access. What if the type made an out-of-bounds index *impossible to construct*?

Fin n: the type of natural numbers strictly less than n.

```
data Fin : Nat -> Set where
  zero : Fin (suc n)              -- 0 is valid for any length
       >= 1
  suc  : Fin n -> Fin (suc n) -- if k < n, then k+1 < n+1
```

`Fin 3` contains exactly: 0, 1, 2. `Fin 0` is empty.

```
lookup : Vec A n -> Fin n -> A      -- no bounds check
    needed
```

To call `lookup` on a length-5 vector, you need a `Fin 5`.
`Fin 5` contains only 0–4. There is no constructor that produces index 5.

Fin: Making the Constraint the Type

Every language has runtime bounds checks on array access. What if the type made an out-of-bounds index *impossible to construct*?

Fin n: the type of natural numbers strictly less than n.

```
data Fin : Nat -> Set where
  zero : Fin (suc n)              -- 0 is valid for any length
       >= 1
  suc  : Fin n -> Fin (suc n) -- if k < n, then k+1 < n+1
```

`Fin 3` contains exactly: 0, 1, 2. `Fin 0` is empty.

```
lookup : Vec A n -> Fin n -> A      -- no bounds check
    needed
```

To call `lookup` on a length-5 vector, you need a `Fin 5`.
`Fin 5` contains only 0–4. There is no constructor that produces index 5.
An out-of-bounds index **has no type**. It cannot be written.

The Limits

The Specification Problem

There's a limit that no amount of formal machinery fixes. Here it is.

The Specification Problem

There's a limit that no amount of formal machinery fixes. Here it is.

Therac-25 (1985–1987)

Radiation therapy machine. Killed at least three patients.

The Specification Problem

There's a limit that no amount of formal machinery fixes. Here it is.

Therac-25 (1985–1987)

Radiation therapy machine. Killed at least three patients.
The software passed all its tests. It satisfied its specification.
The spec didn't model a race condition that was only reachable when an operator typed quickly between two specific commands.

The Specification Problem

There's a limit that no amount of formal machinery fixes. Here it is.

Therac-25 (1985–1987)

Radiation therapy machine. Killed at least three patients.
The software passed all its tests. It satisfied its specification.
The spec didn't model a race condition that was only reachable when an operator typed quickly between two specific commands.
The proof was correct. The spec was wrong.

The Specification Problem

There's a limit that no amount of formal machinery fixes. Here it is.

Therac-25 (1985–1987)

Radiation therapy machine. Killed at least three patients.
The software passed all its tests. It satisfied its specification.
The spec didn't model a race condition that was only reachable when an operator typed quickly between two specific commands.
The proof was correct. The spec was wrong.

What you actually get from formal verification

A proof that your code matches your spec.
Not that your spec matches the real world.

Writing the spec correctly is the hard part. The proof doesn't help with that.

Where This Sits

Each step in this progression catches a class of bugs earlier:

Where This Sits

Each step in this progression catches a class of bugs earlier:

System	What it can express	Checked when
No types	Nothing	Never
Simple types	What *kind* of value	Compile time
Assertions	What's true at this point	Runtime
Dependent types	What's true *about the value*	Compile time
Proof assistants	Any property you can state	Compile time

Where This Sits

Each step in this progression catches a class of bugs earlier:

System	What it can express	Checked when
No types	Nothing	Never
Simple types	What *kind* of value	Compile time
Assertions	What's true at this point	Runtime
Dependent types	What's true *about the value*	Compile time
Proof assistants	Any property you can state	Compile time

Dependent types are the step where "this index is in bounds" stops being a comment and becomes something the compiler enforces.

Where This Sits

Each step in this progression catches a class of bugs earlier:

System	What it can express	Checked when
No types	Nothing	Never
Simple types	What *kind* of value	Compile time
Assertions	What's true at this point	Runtime
Dependent types	What's true *about the value*	Compile time
Proof assistants	Any property you can state	Compile time

Dependent types are the step where "this index is in bounds" stops being a comment and becomes something the compiler enforces.

Next time

Types that describe not *what* a value is, but *how* it gets used.
Effect types, linear types, session types — same idea, different axis.