
Checking Proofs with Computers

and the Curry-Howard Correspondence

Bryan Lu

A Little About Me

- » HCSSiM 2019 alum
- » Graduate Student in Math @ University of Washington
 - ❖ BA Math/Computer Science from Cornell University
- » Combinatorics! ❤️



A Little About Me

- » HCSSiM 2019 alum
- » Graduate Student in Math @ University of Washington
 - ❖ BA Math/Computer Science from Cornell University
- » Combinatorics! ❤️
 - ❖ had a formal verification phase



Context

Computational Power in Math

Computational Power in Math

- » The four-color theorem (1976)
- » Optimal sphere-packing (1998)

Computational Power in Math

- » The four-color theorem (1976)
- » Optimal sphere-packing (1998)
- » “Happy ending problem” for hexagon – 17 points (2006)
- » Solving any Rubik’s Cube – 20 moves (2010)
- » Minimum number of clues in sudoku – 17 (2012)

Proofs Verified by Computers

Proofs/programs checked by *proof assistants*:

Proofs Verified by Computers

Proofs/programs checked by *proof assistants*:

- » The four-color theorem (Coq, 2002)
- » The CompCert C Compiler (Coq, 2009)
- » seL4 kernel (Isabelle/HOL, 2010)
- » The Feit-Thompson theorem about groups of odd size (Coq, 2012)

Proofs Verified by Computers

Proofs/programs checked by *proof assistants*:

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



Q: 🤔 \Rightarrow 💡 \Rightarrow ✓/✗?



Q: 🤔 ⇒ 💻 ⇒ ✅/✖?

A: ✨*programming languages*✨!

Simply-Typed -da Calculus

Syntax

Terms in the simply-typed λ -da calculus:

$$e ::= c \mid x \mid \lambda(x : t) \rightarrow e \mid e_1 e_2$$

Some examples of terms:

true, false, *, x , y , $\lambda(x) \rightarrow x$, $(\lambda(x) \rightarrow \text{true}) *$

Syntax

Terms in the simply-typed λ -da calculus:

$$e ::= c \mid x \mid \lambda(x : t) \rightarrow e \mid e_1 e_2$$

Some examples of terms:

true, false, *, x , y , $\lambda(x) \rightarrow x$, $(\lambda(x) \rightarrow \text{true}) *$

All terms have types:

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

Basic types include bool, unit, etc.

true : bool false : bool * : unit $\lambda(x) \rightarrow \text{true} : t \rightarrow \text{bool}$

Examples: Functions

Example

» $e_1 = \text{λ}(x : t) \rightarrow x : ???$

Examples: Functions

Example

- » $e_1 = \lambda(x : t) \rightarrow x : t \rightarrow t$ identity function
 - ❖ $e_1 \text{ true} = (\lambda(x : t) \rightarrow x) \text{ true} = \text{true}$

Examples: Functions

Example

- » $e_1 = \lambda(x : t) \rightarrow x : t \rightarrow t$ identity function
 - ❖ $e_1 \text{ true} = (\lambda(x : t) \rightarrow x) \text{ true} = \text{true}$
- » $e_2 = \lambda(x : t_1) \rightarrow \lambda(y : t_2) \rightarrow x$

Examples: Functions

Example

- » $e_1 = \lambda(x : t) \rightarrow x : t \rightarrow t$ identity function
 - ❖ $e_1 \text{ true} = (\lambda(x : t) \rightarrow x) \text{ true} = \text{true}$
- » $e_2 = \lambda(x, y) \rightarrow x : ???$

Examples: Functions

Example

- » $e_1 = \lambda(x : t) \rightarrow x : t \rightarrow t$ identity function
 - ❖ $e_1 \text{ true} = (\lambda(x : t) \rightarrow x) \text{ true} = \text{true}$
- » $e_2 = \lambda(x, y) \rightarrow x : t_1 \rightarrow (t_2 \rightarrow t_1)$

Examples: Functions

Example

- » $e_1 = \lambda(x : t) \rightarrow x : t \rightarrow t$ identity function
 - ❖ $e_1 \text{ true} = (\lambda(x : t) \rightarrow x) \text{ true} = \text{true}$
- » $e_2 = \lambda(x, y) \rightarrow x : t_1 \rightarrow t_2 \rightarrow t_1$ constant function
 - ❖ NOT the same as $(t_1 \rightarrow t_2) \rightarrow t_1$!

Examples: Functions

Example

- » $e_1 = \lambda(x : t) \rightarrow x : t \rightarrow t$ identity function
 - ❖ $e_1 \text{ true} = (\lambda(x : t) \rightarrow x) \text{ true} = \text{true}$
- » $e_2 = \lambda(x, y) \rightarrow x : t_1 \rightarrow t_2 \rightarrow t_1$ constant function
 - ❖ NOT the same as $(t_1 \rightarrow t_2) \rightarrow t_1$!
- » $e_3 = \lambda(x, y) \rightarrow x y : ???$

Examples: Functions

Example

- » $e_1 = \lambda(x : t) \rightarrow x : t \rightarrow t$ identity function
 - ❖ $e_1 \text{ true} = (\lambda(x : t) \rightarrow x) \text{ true} = \text{true}$
- » $e_2 = \lambda(x, y) \rightarrow x : t_1 \rightarrow t_2 \rightarrow t_1$ constant function
 - ❖ NOT the same as $(t_1 \rightarrow t_2) \rightarrow t_1$!
- » $e_3 = \lambda(x, y) \rightarrow x \ y : (t_1 \rightarrow t_2) \rightarrow t_1 \rightarrow t_2$ evaluation function
 - ❖ $e_3 \ e_1 \ * = e_1 \ * = *$

Exercise/Questions?

What is the type of the following term?

$$e_4 = \text{λ}(x, y, z) \rightarrow x z (y z)$$



WHAT IF WE TRIED
MORE POWER?



Let's Talk About Types!

The Natural Numbers

Introducing \mathbb{N} with *inductive types*:

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

O and S are called *constructors* (also *introduction rules*).

The Natural Numbers

Introducing \mathbb{N} with *inductive types*:

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

O and S are called *constructors* (also *introduction rules*).

How do these constructors allow us to model $\{0, 1, 2, \dots\}$?

The Natural Numbers

Introducing \mathbb{N} with *inductive types*:

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

O and S are called *constructors* (also *introduction rules*).

How do these constructors allow us to model $\{0, 1, 2, \dots\}$?

$O \sim 0 \in \mathbb{N}$ $S \sim \text{succ} : \mathbb{N} \rightarrow \mathbb{N}$

The Natural Numbers

Introducing \mathbb{N} with *inductive types*:

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

O and S are called *constructors* (also *introduction rules*).

How do these constructors allow us to model $\{0, 1, 2, \dots\}$?

$$\begin{aligned} O &\sim 0 \in \mathbb{N} & S &\sim \text{succ} : \mathbb{N} \rightarrow \mathbb{N} \\ S\ O &\sim \text{succ}(0) = 1 \in \mathbb{N} \end{aligned}$$

The Natural Numbers

Introducing \mathbb{N} with *inductive types*:

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

O and S are called *constructors* (also *introduction rules*).

How do these constructors allow us to model $\{0, 1, 2, \dots\}$?

$$O \sim 0 \in \mathbb{N} \quad S \sim \text{succ} : \mathbb{N} \rightarrow \mathbb{N}$$

$$S\ O \sim \text{succ}(0) = 1 \in \mathbb{N}$$

$$S\ (S\ O) \sim \text{succ}(\text{succ}(0)) = \text{succ}(1) = 2 \in \mathbb{N}$$

The Natural Numbers

Introducing \mathbb{N} with *inductive types*:

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

O and S are called *constructors* (also *introduction rules*).

How do these constructors allow us to model $\{0, 1, 2, \dots\}$?

$$O \sim 0 \in \mathbb{N} \quad S \sim \text{succ} : \mathbb{N} \rightarrow \mathbb{N}$$

$$S\ O \sim \text{succ}(0) = 1 \in \mathbb{N}$$

$$S\ (S\ O) \sim \text{succ}(\text{succ}(0)) = \text{succ}(1) = 2 \in \mathbb{N}$$

\vdots

Induction Principles

Induction principles (elimination rules) allow you to define functions on inductive types.

Induction Principles

Induction principles (elimination rules) allow you to define functions on inductive types.

Example

The induction principle of \mathbb{N} requires us to specify values for the “base case” and “inductive step.”

Induction Principles

Induction principles (elimination rules) allow you to define functions on inductive types.

Example

The induction principle of \mathbb{N} requires us to specify values for the “base case” and “inductive step.”

```
is_zero :  $\mathbb{N} \rightarrow \text{bool} =$ 
    $(x : \mathbb{N}) \rightarrow \text{match } x \text{ with}$ 
  | 0  $\Rightarrow$  true
  | S k  $\Rightarrow$  false
```

Induction Principles

Induction principles (elimination rules) allow you to define functions on inductive types.

Example

The induction principle of \mathbb{N} requires us to specify values for the “base case” and “inductive step.”

```
is_zero :  $\mathbb{N} \rightarrow \text{bool} =$       add :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} =$ 
   $\lambda(x : \mathbb{N}) \rightarrow \text{match } x \text{ with}$      $\lambda(x, y) \rightarrow \text{match } y \text{ with}$ 
  | 0  $\Rightarrow$  true                      | 0  $\Rightarrow$  x
  | S k  $\Rightarrow$  false                   | S k  $\Rightarrow$  S (add x k)
```

Data Structures

Inductive types allow us to model lists of numbers:

```
inductive natlist
| nil : natlist
| cons :   $a : \mathbb{N}$  →   $t : \text{natlist}$  → natlist
```

Data Structures

Inductive types allow us to model lists of numbers:

```
inductive natlist
| nil : natlist
| cons :  $\lambda(a : \mathbb{N}) \rightarrow \lambda(t : \text{natlist}) \rightarrow \text{natlist}$ 
```

Example

Data Structures

Inductive types allow us to model lists of numbers:

```
inductive natlist
| nil : natlist
| cons :   $a : \mathbb{N}$  →   $t : \text{natlist}$  → natlist
```

Example

nil ~ []

Data Structures

Inductive types allow us to model lists of numbers:

```
inductive natlist
| nil : natlist
| cons :  (a : N) →  (t : natlist) → natlist
```

Example

	nil	~	[]	
cons	(S O)	nil	~	[1]

Data Structures

Inductive types allow us to model lists of numbers:

```
inductive natlist
| nil : natlist
| cons :  $\lambda(a : \mathbb{N}) \rightarrow \lambda(t : \text{natlist}) \rightarrow \text{natlist}$ 
```

Example

	nil	\sim	[]			
cons	($S O$)	nil	\sim	[1]		
cons	($S(S O)$)	(cons	($S O$)	nil)	\sim	

Data Structures

Inductive types allow us to model lists of numbers:

```
inductive natlist
| nil : natlist
| cons :  $\lambda(a : \mathbb{N}) \rightarrow \lambda(t : \text{natlist}) \rightarrow \text{natlist}$ 
```

Example

	nil	\sim	[]			
cons	($S O$)	nil	\sim	[1]		
cons	($S(S O)$)	(cons	($S O$)	nil)	\sim	[2, 1]

Data Structures

Inductive types allow us to model lists of numbers:

```
inductive natlist
| nil : natlist
| cons :  $\lambda(a : \mathbb{N}) \rightarrow \lambda(t : \text{natlist}) \rightarrow \text{natlist}$ 
```

Example

	nil	\sim	[]	
	cons (S O)	nil	\sim	[1]
cons (S (S O))	(cons (S O) nil)	\sim	[2, 1]	
⋮				



WHAT IF WE TRIED
MORE POWER?



Combined Syntax

Combine rules for making terms vs. types into one rule:

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

What's changed?

Combined Syntax

Combine rules for making terms vs. types into one rule:

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

What's changed?

» Dependent function types

- ❖ Functions whose type is $\lambda(x : t_1) \rightarrow t_2$, where t_2 may contain x
- ❖ " $t_1 \rightarrow t_2$ " = $\lambda(_) : t_1 \rightarrow t_2$ (t_2 doesn't depend on x)

Combined Syntax

Combine rules for making terms vs. types into one rule:

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

What's changed?

- » Dependent function types
 - ❖ Functions whose type is $\lambda(x : t_1) \rightarrow t_2$, where t_2 may contain x
 - ❖ " $t_1 \rightarrow t_2$ " = $\lambda(_) : t_1 \rightarrow t_2$ (t_2 doesn't depend on x)
- » Introducing the *sort* \mathbb{T} , the “type” of types
 - ❖ $\text{bool} : \mathbb{T}$, $\text{unit} : \mathbb{T}$, $\mathbb{N} : \mathbb{T}$, etc.

Application: Lists (again)

Recall our type for lists of \mathbb{N} s:

```
inductive natlist
| nil : natlist
| cons :  $\lambda(a : \mathbb{N}) \rightarrow \lambda(t : \text{natlist}) \rightarrow \text{natlist}$ 
```

Application: Lists (again)

Recall our type for lists of \mathbb{N} s:

```
inductive natlist
| nil : natlist
| cons :  $\lambda(a : \mathbb{N}) \rightarrow \lambda(t : \text{natlist}) \rightarrow \text{natlist}$ 
```

Why not build a list for *any* type?

```
inductive list :  $\mathbb{T} \rightarrow \mathbb{T}$ 
| nil :  $\lambda(\alpha : \mathbb{T}) \rightarrow \text{list } \alpha$ 
| cons :  $\lambda(\alpha : \mathbb{T}) \rightarrow \lambda(a : \alpha) \rightarrow \lambda(t : \text{list } \alpha) \rightarrow \text{list } \alpha$ 
```

Application: Lists (again)

Recall our type for lists of \mathbb{N} s:

```
inductive natlist
| nil : natlist
| cons :  $\alpha : \mathbb{N} \rightarrow$   $t : \text{natlist} \rightarrow \text{natlist}$ 
```

Why not build a list for *any* type?

```
inductive list :  $\mathbb{T} \rightarrow \mathbb{T}$ 
| nil :  $\alpha : \mathbb{T} \rightarrow$  list  $\alpha$ 
| cons :  $\alpha : \mathbb{T}, a : \alpha, t : \text{list } \alpha \rightarrow \text{list } \alpha$ 
```

Terms vs. Types

Terms and types can depend on other terms and types:

Terms vs. Types

Terms and types can depend on other terms and types:

- » Terms depending on terms:

Terms vs. Types

Terms and types can depend on other terms and types:

- » Terms depending on terms: function application

Terms vs. Types

Terms and types can depend on other terms and types:

- » Terms depending on terms: function application
 - ❖ $(\lambda(x y) \rightarrow x) w z$ depends on w .

Terms vs. Types

Terms and types can depend on other terms and types:

- » Terms depending on terms: function application
 - ❖ $(\lambda(x y) \rightarrow x) w z$ depends on w .
- » Terms depending on types:

Terms vs. Types

Terms and types can depend on other terms and types:

- » Terms depending on terms: function application
 - ❖ $(\lambda(x y) \rightarrow x) w z$ depends on w .
- » Terms depending on types: polymorphic functions

Terms vs. Types

Terms and types can depend on other terms and types:

- » Terms depending on terms: function application
 - ❖ $(\lambda(x y) \rightarrow x) w z$ depends on w .
- » Terms depending on types: polymorphic functions
 - ❖ $(\lambda(x) \rightarrow x)$ depends on the type of the input x .

Terms vs. Types

Terms and types can depend on other terms and types:

- » Terms depending on terms: function application
 - ❖ $(\lambda(x y) \rightarrow x) w z$ depends on w .
- » Terms depending on types: polymorphic functions
 - ❖ $(\lambda(x) \rightarrow x)$ depends on the type of the input x .
- » Types depending on types:

Terms vs. Types

Terms and types can depend on other terms and types:

- » Terms depending on terms: function application
 - ❖ $(\lambda(x y) \rightarrow x) w z$ depends on w .
- » Terms depending on types: polymorphic functions
 - ❖ $(\lambda(x) \rightarrow x)$ depends on the type of the input x .
- » Types depending on types: type constructors

Terms vs. Types

Terms and types can depend on other terms and types:

- » Terms depending on terms: function application
 - ❖ $(\lambda(x y) \rightarrow x) w z$ depends on w .
- » Terms depending on types: polymorphic functions
 - ❖ $(\lambda(x) \rightarrow x)$ depends on the type of the input x .
- » Types depending on types: type constructors
 - ❖ $\text{list } \alpha$ depends on the type α .

Terms vs. Types

Terms and types can depend on other terms and types:

- » Terms depending on terms: function application
 - ❖ $(\lambda(x y) \rightarrow x) w z$ depends on w .
- » Terms depending on types: polymorphic functions
 - ❖ $(\lambda(x) \rightarrow x)$ depends on the type of the input x .
- » Types depending on types: type constructors
 - ❖ $\text{list } \alpha$ depends on the type α .
- » Types depending on terms: *dependent types*
 - ❖ ???

Example: Type that Depends on a Term

Adapt our polymorphic list to record the length of the list!
Recall:

```
inductive list : T → T
| nil : (α : T) → list α
| cons : (α : T, a : α, t : list α) → list α
```

Example: Type that Depends on a Term

Adapt our polymorphic list to record the length of the list!
Recall:

```
inductive list : T → T
| nil : (α : T) → list α
| cons : (α : T, a : α, t : list α) → list α
```

Now consider:

```
inductive vector : T → N → T
| nil : (α : T) → vector α 0
| cons : (α : T, a : α, n : N, t : vector α n) → vector α (S n)
```

Examples: Inductive Types

Some important inductive dependent types:

```
» inductive combine :  $\alpha : \mathbb{T} \rightarrow \beta : \mathbb{T} \rightarrow \mathbb{T}$ 
| pair :  $\alpha \rightarrow \beta \rightarrow \text{combine } \alpha \beta$ 
```

Examples: Inductive Types

Some important inductive dependent types:

```
» inductive combine :  $\alpha : \mathbb{T} \rightarrow \beta : \mathbb{T} \rightarrow \mathbb{T}$ 
| pair :  $\alpha \rightarrow \beta \rightarrow \text{combine } \alpha \beta$ 
❖ pair (S O) true : combine N bool
```

Examples: Inductive Types

Some important inductive dependent types:

```
» inductive combine :  $\alpha : \mathbb{T} \rightarrow \beta : \mathbb{T} \rightarrow \mathbb{T}$ 
| pair :  $\alpha \rightarrow \beta \rightarrow \text{combine } \alpha \beta$ 
  ♦ pair (S O) true : combine N bool
inductive choose :  $\alpha : \mathbb{T} \rightarrow \beta : \mathbb{T} \rightarrow \mathbb{T}$ 
» | fst :  $\alpha \rightarrow \text{choose } \alpha \beta$ 
| snd :  $\beta \rightarrow \text{choose } \alpha \beta$ 
```

Examples: Inductive Types

Some important inductive dependent types:

```
» inductive combine : ⚡(α : T) → ⚡(β : T) → T
| pair : α → β → combine α β
  ♦ pair (S O) true : combine N bool

inductive choose : ⚡(α : T) → ⚡(β : T) → T
» | fst : α → choose α β
| snd : β → choose α β
  ♦ fst (S (S O)) : choose N bool
  ♦ snd false : choose N bool
  ♦ snd false : choose unit bool
```

Examples: Inductive Types

Some important inductive dependent types:

- » inductive combine : $\textcolor{brown}{\lambda}(\alpha : \mathbb{T}) \rightarrow \textcolor{brown}{\lambda}(\beta : \mathbb{T}) \rightarrow \mathbb{T}$
 - | pair : $\alpha \rightarrow \beta \rightarrow \text{combine } \alpha \beta$
 - ❖ pair (S O) true : combine N bool
- » inductive choose : $\textcolor{brown}{\lambda}(\alpha : \mathbb{T}) \rightarrow \textcolor{brown}{\lambda}(\beta : \mathbb{T}) \rightarrow \mathbb{T}$
 - | fst : $\alpha \rightarrow \text{choose } \alpha \beta$
 - | snd : $\beta \rightarrow \text{choose } \alpha \beta$
 - ❖ fst (S (S O)) : choose N bool
 - ❖ snd false : choose N bool
 - ❖ snd false : choose unit bool
- » inductive none : \mathbb{T}

Examples: Inductive Types

Some important inductive dependent types:

- » inductive combine : $\textcolor{brown}{\lambda}(\alpha : \mathbb{T}) \rightarrow \textcolor{brown}{\lambda}(\beta : \mathbb{T}) \rightarrow \mathbb{T}$
 - | pair : $\alpha \rightarrow \beta \rightarrow \text{combine } \alpha \beta$
 - ❖ pair (S O) true : combine N bool
- » inductive choose : $\textcolor{brown}{\lambda}(\alpha : \mathbb{T}) \rightarrow \textcolor{brown}{\lambda}(\beta : \mathbb{T}) \rightarrow \mathbb{T}$
 - | fst : $\alpha \rightarrow \text{choose } \alpha \beta$
 - | snd : $\beta \rightarrow \text{choose } \alpha \beta$
 - ❖ fst (S (S O)) : choose N bool
 - ❖ snd false : choose N bool
 - ❖ snd false : choose unit bool
- » inductive none : \mathbb{T}
 - ❖ There are NO terms of type none! (*uninhabited*)

Revisiting Logic

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q$?

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q$? pair of proofs (p, q) .

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q$? pair of proofs (p, q) .
- » disjunction, $P \vee Q$?

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q$? pair of proofs (p, q) .
- » disjunction, $P \vee Q$? proof + tag: $(1, p)$ or $(2, q)$.

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q?$ pair of proofs $(p, q).$
- » disjunction, $P \vee Q?$ proof + tag: $(1, p)$ or $(2, q).$
- » implication, $P \rightarrow Q?$

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q?$ pair of proofs $(p, q).$
- » disjunction, $P \vee Q?$ proof + tag: $(1, p)$ or $(2, q).$
- » implication, $P \rightarrow Q?$ function turning $p \rightarrow q.$

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q?$ pair of proofs $(p, q).$
- » disjunction, $P \vee Q?$ proof + tag: $(1, p)$ or $(2, q).$
- » implication, $P \rightarrow Q?$ function turning $p \rightarrow q.$
- » negation, $\neg P?$

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q?$ pair of proofs $(p, q).$
- » disjunction, $P \vee Q?$ proof + tag: $(1, p)$ or $(2, q).$
- » implication, $P \rightarrow Q?$ function turning $p \rightarrow q.$
- » negation, $\neg P?$ function turning $p \rightarrow (\text{proof of false}).$

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q?$ pair of proofs $(p, q).$
- » disjunction, $P \vee Q?$ proof + tag: $(1, p)$ or $(2, q).$
- » implication, $P \rightarrow Q?$ function turning $p \rightarrow q.$
- » negation, $\neg P?$ function turning $p \rightarrow (\text{proof of false}).$

Suppose S is a set and $T : S \rightarrow \text{Prop}$ a proposition over S . What is a proof of the quantified statement ...

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q?$ pair of proofs $(p, q).$
- » disjunction, $P \vee Q?$ proof + tag: $(1, p)$ or $(2, q).$
- » implication, $P \rightarrow Q?$ function turning $p \rightarrow q.$
- » negation, $\neg P?$ function turning $p \rightarrow (\text{proof of false}).$

Suppose S is a set and $T : S \rightarrow \text{Prop}$ a proposition over S . What is a proof of the quantified statement ...

- » $\forall x \in S, T(x)?$

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q?$ pair of proofs $(p, q).$
- » disjunction, $P \vee Q?$ proof + tag: $(1, p)$ or $(2, q).$
- » implication, $P \rightarrow Q?$ function turning $p \rightarrow q.$
- » negation, $\neg P?$ function turning $p \rightarrow (\text{proof of false}).$

Suppose S is a set and $T : S \rightarrow \text{Prop}$ a proposition over S . What is a proof of the quantified statement ...

- » $\forall x \in S, T(x)?$ function turning $(x \in S) \rightarrow \text{proof } t(x).$

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q?$ pair of proofs $(p, q).$
- » disjunction, $P \vee Q?$ proof + tag: $(1, p)$ or $(2, q).$
- » implication, $P \rightarrow Q?$ function turning $p \rightarrow q.$
- » negation, $\neg P?$ function turning $p \rightarrow (\text{proof of false}).$

Suppose S is a set and $T : S \rightarrow \text{Prop}$ a proposition over S . What is a proof of the quantified statement ...

- » $\forall x \in S, T(x)?$ function turning $(x \in S) \rightarrow \text{proof } t(x).$
- » $\exists x \in S, T(x)?$

Computing with Proofs

Suppose P, Q are propositions with proofs p, q , respectively.
How do we computationally get a proof of ...

- » conjunction, $P \wedge Q?$ pair of proofs $(p, q).$
- » disjunction, $P \vee Q?$ proof + tag: $(1, p)$ or $(2, q).$
- » implication, $P \rightarrow Q?$ function turning $p \rightarrow q.$
- » negation, $\neg P?$ function turning $p \rightarrow (\text{proof of false}).$

Suppose S is a set and $T : S \rightarrow \text{Prop}$ a proposition over S . What is a proof of the quantified statement ...

- » $\forall x \in S, T(x)?$ function turning $(x \in S) \rightarrow \text{proof } t(x).$
- » $\exists x \in S, T(x)?$ pair of instance $x \in S$, proof $t(x).$

The Correspondence

The Curry-Howard Correspondence

types \rightleftharpoons propositions terms \rightleftharpoons proofs

The Curry-Howard Correspondence

types \rightleftharpoons propositions terms \rightleftharpoons proofs

Type	Proposition	Expression	Proof
???	$P \wedge Q$???	(p, q)
???	$P \vee Q$???	$(1, p), (2, q)$
???	$P \rightarrow Q$???	$f : p \mapsto q$
???	true	???	\exists
???	false	???	\emptyset
???	$\neg P$???	$f : p \mapsto \perp$
???	$\forall x \in P, Q(x)$???	$f : x \mapsto q(x)$
???	$\exists x \in P, Q(x)$???	$(x, q(x))$

The Curry-Howard Correspondence

types \rightleftharpoons propositions terms \rightleftharpoons proofs

Type	Proposition	Expression	Proof
combine $\alpha \beta$	$P \wedge Q$	pair $a b$	(p, q)
???	$P \vee Q$???	$(1, p), (2, q)$
???	$P \rightarrow Q$???	$f : p \mapsto q$
???	true	???	\exists
???	false	???	\nexists
???	$\neg P$???	$f : p \mapsto \perp$
???	$\forall x \in P, Q(x)$???	$f : x \mapsto q(x)$
???	$\exists x \in P, Q(x)$???	$(x, q(x))$

The Curry-Howard Correspondence

types \rightleftharpoons propositions terms \rightleftharpoons proofs

Type	Proposition	Expression	Proof
combine $\alpha \beta$	$P \wedge Q$	pair $a b$	(p, q)
choose $\alpha \beta$	$P \vee Q$	fst a , snd b	$(1, p), (2, q)$
???	$P \rightarrow Q$???	$f : p \mapsto q$
???	true	???	\exists
???	false	???	\nexists
???	$\neg P$???	$f : p \mapsto \perp$
???	$\forall x \in P, Q(x)$???	$f : x \mapsto q(x)$
???	$\exists x \in P, Q(x)$???	$(x, q(x))$

The Curry-Howard Correspondence

types \rightleftharpoons propositions terms \rightleftharpoons proofs

Type	Proposition	Expression	Proof
combine $\alpha \beta$	$P \wedge Q$	pair $a b$	(p, q)
choose $\alpha \beta$	$P \vee Q$	fst a , snd b	$(1, p), (2, q)$
$\alpha \rightarrow \beta$	$P \rightarrow Q$	 : $a \mapsto b$	$f : p \mapsto q$
???	true	???	\exists
???	false	???	\nexists
???	$\neg P$???	$f : p \mapsto \perp$
???	$\forall x \in P, Q(x)$???	$f : x \mapsto q(x)$
???	$\exists x \in P, Q(x)$???	$(x, q(x))$

The Curry-Howard Correspondence

types \rightleftharpoons propositions terms \rightleftharpoons proofs

Type	Proposition	Expression	Proof
combine $\alpha \beta$	$P \wedge Q$	pair $a b$	(p, q)
choose $\alpha \beta$	$P \vee Q$	fst a , snd b	$(1, p), (2, q)$
$\alpha \rightarrow \beta$	$P \rightarrow Q$	 : $a \mapsto b$	$f : p \mapsto q$
unit	true	*	\exists
???	false	???	\nexists
???	$\neg P$???	$f : p \mapsto \perp$
???	$\forall x \in P, Q(x)$???	$f : x \mapsto q(x)$
???	$\exists x \in P, Q(x)$???	$(x, q(x))$

The Curry-Howard Correspondence

types \rightleftharpoons propositions terms \rightleftharpoons proofs

Type	Proposition	Expression	Proof
combine $\alpha \beta$	$P \wedge Q$	pair $a b$	(p, q)
choose $\alpha \beta$	$P \vee Q$	fst a , snd b	$(1, p), (2, q)$
$\alpha \rightarrow \beta$	$P \rightarrow Q$	 : $a \mapsto b$	$f : p \mapsto q$
unit	true	*	\exists
none	false		\nexists
???	$\neg P$???	$f : p \mapsto \perp$
???	$\forall x \in P, Q(x)$???	$f : x \mapsto q(x)$
???	$\exists x \in P, Q(x)$???	$(x, q(x))$

The Curry-Howard Correspondence

types \rightleftharpoons propositions terms \rightleftharpoons proofs

Type	Proposition	Expression	Proof
combine $\alpha \beta$	$P \wedge Q$	pair $a b$	(p, q)
choose $\alpha \beta$	$P \vee Q$	fst a , snd b	$(1, p), (2, q)$
$\alpha \rightarrow \beta$	$P \rightarrow Q$	 : $a \mapsto b$	$f : p \mapsto q$
unit	true	*	\exists
none	false		\nexists
$\alpha \rightarrow \text{none}$	$\neg P$	 : $a \mapsto \perp$	$f : p \mapsto \perp$
???	$\forall x \in P, Q(x)$???	$f : x \mapsto q(x)$
???	$\exists x \in P, Q(x)$???	$(x, q(x))$

The Curry-Howard Correspondence

types \rightleftharpoons propositions terms \rightleftharpoons proofs

Type	Proposition	Expression	Proof
combine $\alpha \beta$	$P \wedge Q$	pair $a b$	(p, q)
choose $\alpha \beta$	$P \vee Q$	fst a , snd b	$(1, p), (2, q)$
$\alpha \rightarrow \beta$	$P \rightarrow Q$	 : $a \mapsto b$	$f : p \mapsto q$
unit	true	*	\exists
none	false		\nexists
$\alpha \rightarrow \text{none}$	$\neg P$	 : $a \mapsto \perp$	$f : p \mapsto \perp$
 $(a : \alpha) \rightarrow \beta$	$\forall x \in P, Q(x)$	 : $a \mapsto b(a)$	$f : x \mapsto q(x)$
???	$\exists x \in P, Q(x)$???	$(x, q(x))$

The Curry-Howard Correspondence

types \rightleftharpoons propositions terms \rightleftharpoons proofs

Type	Proposition	Expression	Proof
combine $\alpha \beta$	$P \wedge Q$	pair $a b$	(p, q)
choose $\alpha \beta$	$P \vee Q$	fst a , snd b	$(1, p), (2, q)$
$\alpha \rightarrow \beta$	$P \rightarrow Q$	$\text{λ } a : a \mapsto b$	$f : p \mapsto q$
unit	true	*	\exists
none	false	✗	∅
$\alpha \rightarrow \text{none}$	$\neg P$	$\text{λ } a : a \mapsto \perp$	$f : p \mapsto \perp$
$\text{λ } (a : \alpha) \rightarrow \beta$	$\forall x \in P, Q(x)$	$\text{λ } a : a \mapsto b(a)$	$f : x \mapsto q(x)$
???	$\exists x \in P, Q(x)$???	$(x, q(x))$



checking proof correctness \rightleftharpoons type-checking term

Demo!

Resources

- » Coq
 - ❖ Software Foundations
- » Lean
 - ❖ The Natural Number Game
 - ❖ leanprover-community (mathlib), Xena Project
- » Agda (less beginner-friendly)
 - ❖ HoTTEST Summer School 2022, HoTT Book
 - ❖ agda-unimath, TypeTopology
 - ❖ cubical Agda, 1Lab



Secret Slides

What's the type of a sort?

Can we have $\mathbb{T} : \mathbb{T}$?

What's the type of a sort?

Can we have $\mathbb{T} : \mathbb{T}$?

NO! It's actually not consistent to allow $\mathbb{T} : \mathbb{T}$ – the argument of this is similar to “the barber who shaves everyone who doesn't shave themselves”, i.e. Bertrand's paradox. Instead, we have *Girard's paradox* which breaks this construction.

What's the type of a sort?

Can we have $\mathbb{T} : \mathbb{T}$?

NO! It's actually not consistent to allow $\mathbb{T} : \mathbb{T}$ – the argument of this is similar to “the barber who shaves everyone who doesn't shave themselves”, i.e. Bertrand's paradox. Instead, we have *Girard's paradox* which breaks this construction.

In fact, we actually need $\mathbb{T} : \mathbb{T}_1$, and then $\mathbb{T}_1 : \mathbb{T}_2$, etc. The \mathbb{T}_i s are called *type universes* and this is more of a technical point that's need to make things work, but often isn't relevant.

The equality type?

What type corresponds to the equality proposition?

The equality type?

What type corresponds to the equality proposition?

We actually can just make an inductive type for this:

```
inductive equal :  $\alpha : \mathbb{T} \rightarrow \alpha \rightarrow \alpha \rightarrow \mathbb{T}$ 
| refl :  $\alpha : \mathbb{T} \rightarrow \alpha : \alpha \rightarrow \text{equal } \alpha \ a \ a$ 
```

The equality type?

What type corresponds to the equality proposition?

We actually can just make an inductive type for this:

```
inductive equal :  $\alpha : \mathbb{T} \rightarrow \alpha \rightarrow \alpha \rightarrow \mathbb{T}$ 
| refl :  $\alpha : \mathbb{T} \rightarrow \alpha : \alpha \rightarrow \text{equal } \alpha \ a \ a$ 
```

There is only one constructor of this type, and it represents the reflexivity of equality (where the two sides of the equality are equivalent).

Defining complicated propositions?

How do we get more complicated propositions?

Defining complicated propositions?

How do we get more complicated propositions?

We can just make more types with appropriate constructors.
For instance, consider the \leq relation on \mathbb{N} :

```
inductive leq :  $\lambda(m : \mathbb{N}) \rightarrow \lambda(n : \mathbb{N}) \rightarrow \mathbb{T}$ 
| leq-0 :  $\lambda(n : \mathbb{N}) \rightarrow \text{leq } 0\ n$ 
| leq-S :  $\lambda(m : \mathbb{N}, n : \mathbb{N}, H : \text{leq } m\ n) \rightarrow \text{leq } (S\ m)\ (S\ n)$ 
```

Defining complicated propositions?

How do we get more complicated propositions?

We can just make more types with appropriate constructors.
For instance, consider the \leq relation on \mathbb{N} :

```
inductive leq :  $\lambda(m : \mathbb{N}) \rightarrow \lambda(n : \mathbb{N}) \rightarrow \mathbb{T}$ 
| leq-0 :  $\lambda(n : \mathbb{N}) \rightarrow \text{leq } 0\ n$ 
| leq-S :  $\lambda(m : \mathbb{N}, n : \mathbb{N}, H : \text{leq } m\ n) \rightarrow \text{leq } (S\ m)\ (S\ n)$ 
```

Here, we can get a proof of this proposition if we either generate it directly from the fact that all naturals are ≥ 0 , or by reducing successor functions on either side of the \leq until we get the above case. This is again inductive!

Defining algebraic structures?

How does one encode an algebraic structure as a type?

Defining algebraic structures?

How does one encode an algebraic structure as a type?

Record types accomplish this, bundling together the type for the underlying set and all necessary operations/properties.
For example, here's a group:

```
struct group :  ( $\alpha : \mathbb{T}$ )  $\rightarrow \mathbb{T}$  {  
|   mul :  $\alpha \rightarrow \alpha \rightarrow \alpha$   
|   id :  $\alpha$   
|   inv :  $\alpha \rightarrow \alpha$   
|   mul_id :  ( $x : \alpha$ )  $\rightarrow$  mul x id = x ... }
```

Defining algebraic structures?

How does one encode an algebraic structure as a type?

Record types accomplish this, bundling together the type for the underlying set and all necessary operations/properties. For example, here's a group:

```
struct group :  $\alpha : \mathbb{T}$  →  $\mathbb{T}$  {  
|   mul :  $\alpha \rightarrow \alpha \rightarrow \alpha$   
|   id :  $\alpha$   
|   inv :  $\alpha \rightarrow \alpha$   
|   mul_id :  $\alpha(x : \alpha) \rightarrow \text{mul } x \text{ id} = x \dots$  }
```

One can also encode hierarchies of algebraic structures in types as well, e.g. a ring is a group + structure. Record types are also useful for encoding \exists propositions.