# Checking Proofs with Computers

## and the Curry-Howard Correspondence

Bryan Lu

July 9, 2024

# 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 (Rocq, 2002)
- The CompCert C Compiler (Rocq, 2009)
- seL4 kernel (Isabelle/HOL, 2010)
- The Feit-Thompson theorem about symsets of odd size (Rocq, 2012)

# Proofs Verified by Computers

Proofs/programs checked by *proof assistants*:

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


Q: ⇒ ⇒ /?

Q: 🤔 ⇒ 💻 ⇒ ✅/❌?

A: ✨*programming languages*✨!

# Simply-Typed 🐑-da Calculus

# Syntax

Terms in the simply-typed -da calculus:

$$e ::= c \mid x \mid (x : t) \to e \mid e_1\, e_2$$

Some examples of terms:

$$\texttt{true},\ \texttt{false},\ *,\ x,\ y,\ (x) \to x,\ ((x) \to \texttt{true})\ *$$

# Syntax

Terms in the simply-typed 🐑-da calculus:

$$e ::= c \mid x \mid 🐑(x : t) \to e \mid e_1\, e_2$$

Some examples of terms:

$$\texttt{true},\ \texttt{false},\ *,\ x,\ y,\ 🐑(x) \to x,\ (🐑(x) \to \texttt{true})\ *$$

All terms have types:

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

Basic types include `bool`, `unit`, etc.

$$\texttt{true} : \texttt{bool} \quad \texttt{false} : \texttt{bool} \quad * : \texttt{unit} \quad 🐑(x) \to \texttt{true} : t \to \texttt{bool}$$

# Examples: Functions

## Example

- $e_1 = 🐑(x : t) \to x \ : \ ???$

# Examples: Functions

## Example

- $e_1 = \lambda(x : t) \to x \ : \ t \to t$ identity function
  - $e_1 \ \texttt{true} = (\lambda(x : t) \to x) \ \texttt{true} = \texttt{true}$

# Examples: Functions

## Example

- $e_1 = \lambda(x : t) \to x \ : \ t \to t$ identity function
  - $e_1 \ \texttt{true} = (\lambda(x : t) \to x) \ \texttt{true} = \texttt{true}$
- $e_2 = \lambda(x : t_1) \to \lambda(y : t_2) \to x$

# Examples: Functions

## Example

- $e_1 = \lambda(x : t) \to x \ : \ t \to t$ identity function
  - $e_1 \ \texttt{true} = (\lambda(x : t) \to x) \ \texttt{true} = \texttt{true}$
- $e_2 = \lambda(x, y) \to x \ : \ ???$

# Examples: Functions

## Example

- $e_1 = \lambda(x : t) \to x \ : \ t \to t$ identity function
  - $e_1 \ \texttt{true} = (\lambda(x : t) \to x) \ \texttt{true} = \texttt{true}$
- $e_2 = \lambda(x, y) \to x \ : \ t_1 \to (t_2 \to t_1)$

# Examples: Functions

## Example

- $e_1 = \lambda(x : t) \to x \ : \ t \to t$ — identity function
  - $e_1 \ \texttt{true} = (\lambda(x : t) \to x) \ \texttt{true} = \texttt{true}$
- $e_2 = \lambda(x, y) \to x \ : \ t_1 \to t_2 \to t_1$ — constant function
  - NOT the same as $(t_1 \to t_2) \to t_1$!

# Examples: Functions

## Example

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

# Examples: Functions

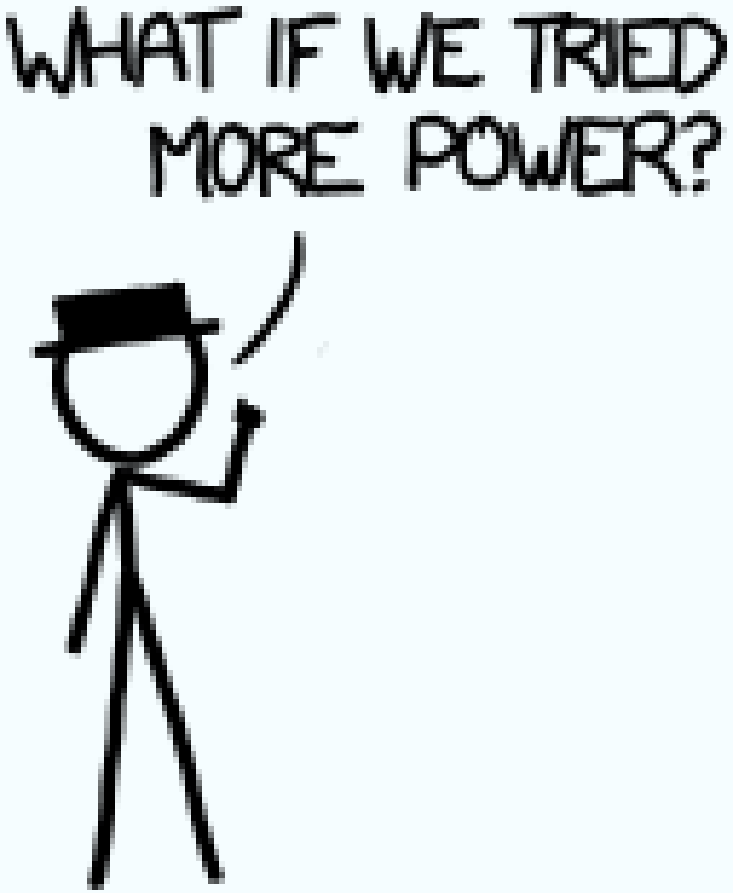
## Example

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

# Exercise/Questions?

*What is the type of the following term?*

$$e_4 = \text{🐣}(x, y, z) \to x\ z\ (y\ z)$$


WHAT IF WE TRIED
MORE POWER?

# Let’s Talk About Types!

# The Natural Numbers

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

$$\begin{array}{l}\texttt{inductive}\ \mathbb{N} \\ \mid O : \mathbb{N} \\ \mid S : \mathbb{N} \to \mathbb{N}\end{array}$$

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

# The Natural Numbers

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

$$\begin{aligned}&\texttt{inductive}\ \mathbb{N}\\ &\mid O : \mathbb{N}\\ &\mid S : \mathbb{N} \to \mathbb{N}\end{aligned}$$

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

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

# The Natural Numbers

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

$$\begin{aligned}&\texttt{inductive}\ \mathbb{N}\\ &\mid O : \mathbb{N}\\ &\mid S : \mathbb{N} \to \mathbb{N}\end{aligned}$$

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

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

$$O \sim 0 \in \mathbb{N} \qquad S \sim \mathsf{succ} : \mathbb{N} \to \mathbb{N}$$

# The Natural Numbers

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

$$\begin{aligned}&\texttt{inductive}\ \mathbb{N}\\ &\mid O : \mathbb{N}\\ &\mid S : \mathbb{N} \to \mathbb{N}\end{aligned}$$

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

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

$$O \sim 0 \in \mathbb{N} \qquad S \sim \mathsf{succ} : \mathbb{N} \to \mathbb{N}$$
$$S\ O \sim \mathsf{succ}(0) = 1 \in \mathbb{N}$$

# The Natural Numbers

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

$$\begin{aligned}&\texttt{inductive}\ \mathbb{N}\\ &\mid O : \mathbb{N}\\ &\mid S : \mathbb{N} \to \mathbb{N}\end{aligned}$$

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

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

$$\begin{aligned}&O \sim 0 \in \mathbb{N} \qquad S \sim \mathsf{succ} : \mathbb{N} \to \mathbb{N}\\ &S\ O \sim \mathsf{succ}(0) = 1 \in \mathbb{N}\\ &S\ (S\ O) \sim \mathsf{succ}(\mathsf{succ}(0)) = \mathsf{succ}(1) = 2 \in \mathbb{N}\end{aligned}$$

# The Natural Numbers

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

$$\begin{aligned}&\texttt{inductive}\ \mathbb{N}\\ &\mid O : \mathbb{N}\\ &\mid S : \mathbb{N} \to \mathbb{N}\end{aligned}$$

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

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

$$\begin{aligned}&O \sim 0 \in \mathbb{N} \qquad S \sim \mathsf{succ} : \mathbb{N} \to \mathbb{N}\\ &S\ O \sim \mathsf{succ}(0) = 1 \in \mathbb{N}\\ &S\ (S\ O) \sim \mathsf{succ}(\mathsf{succ}(0)) = \mathsf{succ}(1) = 2 \in \mathbb{N}\\ &\vdots\end{aligned}$$

# 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 : ℕ → bool =
  🐑(x : ℕ) → match x with
  | O ⇒ true
  | S k ⇒ 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 : ℕ → bool =
  🐑(x : ℕ) → match x with
  | O ⇒ true
  | S k ⇒ false
```

```
add : ℕ → ℕ → ℕ =
  🐑(x, y) → match y with
  | O ⇒ x
  | S k ⇒ S (add x k)
```

# Data Structures

Inductive types allow us to model lists of numbers:

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

# Data Structures

Inductive types allow us to model lists of numbers:

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

**Example**

# Data Structures

Inductive types allow us to model lists of numbers:

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

$$\begin{array}{rrcl} & \texttt{nil} & \sim & [\,] \\ \texttt{cons}\ (S\ O) & \texttt{nil} & \sim & [1] \end{array}$$

# Data Structures

Inductive types allow us to model lists of numbers:

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

**Example**

$$\begin{aligned} \texttt{nil} &\sim [\,] \\ \texttt{cons}\ (S\ O)\ \texttt{nil} &\sim [1] \\ \texttt{cons}\ (S\ (S\ O))\ (\texttt{cons}\ (S\ O)\ \texttt{nil}) &\sim \end{aligned}$$

# Data Structures

Inductive types allow us to model lists of numbers:

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

**Example**

$$\begin{aligned}
\texttt{nil} &\sim [\,] \\
\texttt{cons}\ (S\ O)\ \texttt{nil} &\sim [1] \\
\texttt{cons}\ (S\ (S\ O))\ (\texttt{cons}\ (S\ O)\ \texttt{nil}) &\sim [2, 1]
\end{aligned}$$

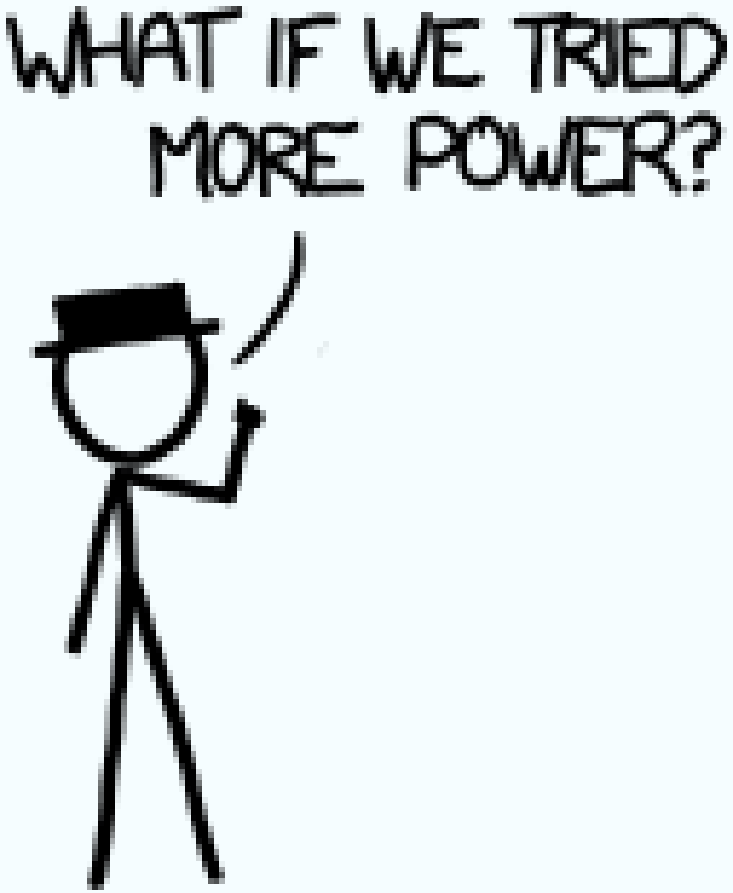
# Data Structures

Inductive types allow us to model lists of numbers:

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

**Example**

$$
\begin{aligned}
&\texttt{nil} && \sim && [\,] \\
&\texttt{cons}\ (S\ O)\ \texttt{nil} && \sim && [1] \\
\texttt{cons}\ (S\ (S\ O))\ (&\texttt{cons}\ (S\ O)\ \texttt{nil}) && \sim && [2, 1] \\
& && && \vdots
\end{aligned}
$$


WHAT IF WE TRIED
MORE POWER?

# Combined Syntax

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

$$t ::= x \mid T \mid \text{🐑}(x : t_1) \to t_2 \mid t_1\, t_2 \mid \mathbb{T}$$

What's changed?

# Combined Syntax

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

$$t ::= x \mid T \mid \text{🐑}(x : t_1) \to t_2 \mid t_1\, t_2 \mid \mathbb{T}$$

What's changed?

- Dependent function types
  - Functions whose type is $\text{🐑}(x : t_1) \to t_2$, where $t_2$ may contain $x$
  - "$t_1 \to t_2$" $= \text{🐑}(\_ : t_1) \to t_2$ ($t_2$ doesn't depend on $x$)

# Combined Syntax

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

$$t ::= x \mid T \mid \text{🐑}(x : t_1) \to t_2 \mid t_1\, t_2 \mid \mathbb{T}$$

What's changed?

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

# Application: Lists (again)

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

$$
\begin{aligned}
&\texttt{inductive natlist}\\
&\mid \texttt{nil} : \texttt{natlist}\\
&\mid \texttt{cons} : \textcolor{white}{\bullet}(a : \mathbb{N}) \to \textcolor{white}{\bullet}(t : \texttt{natlist}) \to \texttt{natlist}
\end{aligned}
$$

# Application: Lists (again)

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

$$\begin{array}{l}\texttt{inductive natlist}\\ \mid \texttt{nil} : \texttt{natlist}\\ \mid \texttt{cons} : \textcolor{yellow}{\bullet}(a : \mathbb{N}) \to \textcolor{yellow}{\bullet}(t : \texttt{natlist}) \to \texttt{natlist}\end{array}$$

Why not build a list for *any* type?

$$\begin{array}{l}\texttt{inductive list} : \mathbb{T} \to \mathbb{T}\\ \mid \texttt{nil} : \textcolor{yellow}{\bullet}(\alpha : \mathbb{T}) \to \texttt{list}\ \alpha\\ \mid \texttt{cons} : \textcolor{yellow}{\bullet}(\alpha : \mathbb{T}) \to \textcolor{yellow}{\bullet}(a : \alpha) \to \textcolor{yellow}{\bullet}(t : \texttt{list}\ \alpha) \to \texttt{list}\ \alpha\end{array}$$

# Application: Lists (again)

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

$$\begin{array}{l}\texttt{inductive natlist}\\ \mid \texttt{nil} : \texttt{natlist}\\ \mid \texttt{cons} : \textcolor{yellow}{\text{🐑}}(a : \mathbb{N}) \to \textcolor{yellow}{\text{🐑}}(t : \texttt{natlist}) \to \texttt{natlist}\end{array}$$

Why not build a list for *any* type?

$$\begin{array}{l}\texttt{inductive list} : \mathbb{T} \to \mathbb{T}\\ \mid \texttt{nil} : \textcolor{yellow}{\text{🐑}}(\alpha : \mathbb{T}) \to \texttt{list}\ \alpha\\ \mid \texttt{cons} : \textcolor{yellow}{\text{🐑}}(\alpha : \mathbb{T}, a : \alpha, t : \texttt{list}\ \alpha) \to \texttt{list}\ \alpha\end{array}$$

# 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
  - $(\lambda(x\ y) \to x)\ w\ z$ depends on $w$.

# Terms vs. Types

Terms and types can depend on other terms and types:

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

# Terms vs. Types

Terms and types can depend on other terms and types:

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

$$\begin{aligned}&\texttt{inductive list} : \mathbb{T} \to \mathbb{T}\\ &\mid \texttt{nil} : \texttt{🐑}(\alpha : \mathbb{T}) \to \texttt{list}\ \alpha\\ &\mid \texttt{cons} : \texttt{🐑}(\alpha : \mathbb{T}, a : \alpha, t : \texttt{list}\ \alpha) \to \texttt{list}\ \alpha\end{aligned}$$

# Example: Type that Depends on a Term

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

$$\begin{aligned}&\texttt{inductive list} : \mathbb{T} \to \mathbb{T}\\ &\mid \texttt{nil} : \textcolor{yellow}{\bullet}(\alpha : \mathbb{T}) \to \texttt{list}\ \alpha\\ &\mid \texttt{cons} : \textcolor{yellow}{\bullet}(\alpha : \mathbb{T}, a : \alpha, t : \texttt{list}\ \alpha) \to \texttt{list}\ \alpha\end{aligned}$$

Now consider:

$$\begin{aligned}&\texttt{inductive vector} : \mathbb{T} \to \mathbb{N} \to \mathbb{T}\\ &\mid \texttt{nil} : \textcolor{yellow}{\bullet}(\alpha : \mathbb{T}) \to \texttt{vector}\ \alpha\ O\\ &\mid \texttt{cons} : \textcolor{yellow}{\bullet}(\alpha : \mathbb{T}, a : \alpha, n : \mathbb{N}, t : \texttt{vector}\ \alpha\ n) \to \texttt{vector}\ \alpha\ (S\ n)\end{aligned}$$

# Examples: Inductive Types

Some important inductive dependent types:

» `inductive combine :` $(\alpha : \mathbb{T}) \to (\beta : \mathbb{T}) \to \mathbb{T}$
| `pair :` $\alpha \to \beta \to$ `combine` $\alpha\ \beta$

# Examples: Inductive Types

Some important inductive dependent types:

- $\texttt{inductive combine} : \forall(\alpha : \mathbb{T}) \to \forall(\beta : \mathbb{T}) \to \mathbb{T}$
  $\mid \texttt{pair} : \alpha \to \beta \to \texttt{combine}\ \alpha\ \beta$
  - $\texttt{pair}\ (S\ O)\ \texttt{true} : \texttt{combine}\ \mathbb{N}\ \texttt{bool}$

# Examples: Inductive Types

Some important inductive dependent types:

» $\texttt{inductive combine} : \prod(\alpha : \mathbb{T}) \to \prod(\beta : \mathbb{T}) \to \mathbb{T}$
$\mid \texttt{pair} : \alpha \to \beta \to \texttt{combine}\ \alpha\ \beta$

❖ $\texttt{pair}\ (S\ O)\ \texttt{true} : \texttt{combine}\ \mathbb{N}\ \texttt{bool}$

» $\texttt{inductive choose} : \prod(\alpha : \mathbb{T}) \to \prod(\beta : \mathbb{T}) \to \mathbb{T}$
$\mid \texttt{fst} : \alpha \to \texttt{choose}\ \alpha\ \beta$
$\mid \texttt{snd} : \beta \to \texttt{choose}\ \alpha\ \beta$

# Examples: Inductive Types

Some important inductive dependent types:

» $\texttt{inductive combine} : \prod(\alpha : \mathbb{T}) \to \prod(\beta : \mathbb{T}) \to \mathbb{T}$
$\mid \texttt{pair} : \alpha \to \beta \to \texttt{combine}\ \alpha\ \beta$

- ❖ $\texttt{pair}\ (S\ O)\ \texttt{true} : \texttt{combine}\ \mathbb{N}\ \texttt{bool}$

» $\texttt{inductive choose} : \prod(\alpha : \mathbb{T}) \to \prod(\beta : \mathbb{T}) \to \mathbb{T}$
$\mid \texttt{fst} : \alpha \to \texttt{choose}\ \alpha\ \beta$
$\mid \texttt{snd} : \beta \to \texttt{choose}\ \alpha\ \beta$

- ❖ $\texttt{fst}\ (S\ (S\ O)) : \texttt{choose}\ \mathbb{N}\ \texttt{bool}$
- ❖ $\texttt{snd false} : \texttt{choose}\ \mathbb{N}\ \texttt{bool}$
- ❖ $\texttt{snd false} : \texttt{choose unit bool}$

# Examples: Inductive Types

Some important inductive dependent types:

- `inductive combine` : $(\alpha : \mathbb{T}) \to (\beta : \mathbb{T}) \to \mathbb{T}$
  | `pair` : $\alpha \to \beta \to$ `combine` $\alpha\ \beta$
  - `pair` $(S\ O)$ `true` : `combine` $\mathbb{N}$ `bool`
- `inductive choose` : $(\alpha : \mathbb{T}) \to (\beta : \mathbb{T}) \to \mathbb{T}$
  | `fst` : $\alpha \to$ `choose` $\alpha\ \beta$
  | `snd` : $\beta \to$ `choose` $\alpha\ \beta$
  - `fst` $(S\ (S\ O))$ : `choose` $\mathbb{N}$ `bool`
  - `snd false` : `choose` $\mathbb{N}$ `bool`
  - `snd false` : `choose unit bool`
- `inductive none` : $\mathbb{T}$

# Examples: Inductive Types

Some important inductive dependent types:

- `inductive combine` : $(\alpha : \mathbb{T}) \to (\beta : \mathbb{T}) \to \mathbb{T}$
  | `pair` : $\alpha \to \beta \to$ `combine` $\alpha\ \beta$
  - `pair` ($S\ O$) `true` : `combine` $\mathbb{N}$ `bool`
- `inductive choose` : $(\alpha : \mathbb{T}) \to (\beta : \mathbb{T}) \to \mathbb{T}$
  | `fst` : $\alpha \to$ `choose` $\alpha\ \beta$
  | `snd` : $\beta \to$ `choose` $\alpha\ \beta$
  - `fst` ($S$ ($S$ $O$)) : `choose` $\mathbb{N}$ `bool`
  - `snd false` : `choose` $\mathbb{N}$ `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, $P \to Q$?

# 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, $P \to Q$? — function turning $p \to q$.

# 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, $P \to Q$? — function turning $p \to q$.
- 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, $P \to Q$? — function turning $p \to q$.
- negation, $\neg P$? — function turning $p \to$ (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, $P \to Q$? — function turning $p \to q$.
- negation, $\neg P$? — function turning $p \to$ (proof of false).

Suppose $S$ is a set and $T : S \to \mathrm{Prop}$ 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, $P \to Q$? — function turning $p \to q$.
- negation, $\neg P$? — function turning $p \to$ (proof of false).

Suppose $S$ is a set and $T : S \to \text{Prop}$ 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, $P \to Q$? — function turning $p \to q$.
- negation, $\neg P$? — function turning $p \to$ (proof of false).

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

- $\forall x \in S, T(x)$? — function turning $(x \in S) \to$ 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, $P \to Q$? — function turning $p \to q$.
- negation, $\neg P$? — function turning $p \to$ (proof of false).

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

- $\forall x \in S, T(x)$? — function turning $(x \in S) \to$ 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, $P \to Q$? — function turning $p \to q$.
- negation, $\neg P$? — function turning $p \to$ (proof of false).

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

- $\forall x \in S, T(x)$? — function turning $(x \in S) \to$ 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 |
|---|---|---|---|
| ??? | $P \wedge Q$ | ??? | $(p, q)$ |
| ??? | $P \vee Q$ | ??? | $(1, p), (2, q)$ |
| ??? | $P \to Q$ | ??? | $f : p \mapsto q$ |
| ??? | true | ??? | $\exists$ |
| ??? | false | ??? | $\nexists$ |
| ??? | $\neg P$ | ??? | $f : p \mapsto \bot$ |
| ??? | $\forall x \in P, Q(x)$ | ??? | $f : x \mapsto q(x)$ |
| ??? | $\exists x \in P, Q(x)$ | ??? | $(x, q(x))$ |

# The Curry-Howard Correspondence

*types* $\rightleftharpoons$ *propositions* *terms* $\rightleftharpoons$ *proofs*

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

# 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` $b$ | $(1, p), (2, q)$ |
| ??? | $P \to Q$ | ??? | $f : p \mapsto q$ |
| ??? | true | ??? | $\exists$ |
| ??? | false | ??? | $\nexists$ |
| ??? | $\neg P$ | ??? | $f : p \mapsto \bot$ |
| ??? | $\forall x \in P, Q(x)$ | ??? | $f : x \mapsto q(x)$ |
| ??? | $\exists x \in P, Q(x)$ | ??? | $(x, q(x))$ |

# 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` $b$ | $(1, p), (2, q)$ |
| $\alpha \to \beta$ | $P \to Q$ | 🐑 : $a \mapsto b$ | $f : p \mapsto q$ |
| ??? | true | ??? | $\exists$ |
| ??? | false | ??? | $\nexists$ |
| ??? | $\neg P$ | ??? | $f : p \mapsto \bot$ |
| ??? | $\forall x \in P, Q(x)$ | ??? | $f : x \mapsto q(x)$ |
| ??? | $\exists x \in P, Q(x)$ | ??? | $(x, q(x))$ |

# 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` $b$ | $(1, p), (2, q)$ |
| $\alpha \to \beta$ | $P \to Q$ | 🐑 $: a \mapsto b$ | $f : p \mapsto q$ |
| `unit` | true | $*$ | $\exists$ |
| ??? | false | ??? | $\nexists$ |
| ??? | $\neg P$ | ??? | $f : p \mapsto \bot$ |
| ??? | $\forall x \in P, Q(x)$ | ??? | $f : x \mapsto q(x)$ |
| ??? | $\exists x \in P, Q(x)$ | ??? | $(x, q(x))$ |

# 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` $b$ | $(1, p), (2, q)$ |
| $\alpha \to \beta$ | $P \to Q$ | 🐑 : $a \mapsto b$ | $f : p \mapsto q$ |
| `unit` | true | $*$ | $\exists$ |
| `none` | false | ❌ | $\nexists$ |
| ??? | $\neg P$ | ??? | $f : p \mapsto \bot$ |
| ??? | $\forall x \in P, Q(x)$ | ??? | $f : x \mapsto q(x)$ |
| ??? | $\exists x \in P, Q(x)$ | ??? | $(x, q(x))$ |

# 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` $b$ | $(1, p), (2, q)$ |
| $\alpha \to \beta$ | $P \to Q$ | 🐑 : $a \mapsto b$ | $f : p \mapsto q$ |
| `unit` | true | $*$ | $\exists$ |
| `none` | false | ✗ | $\nexists$ |
| $\alpha \to$ `none` | $\neg P$ | 🐑 : $a \mapsto \lightning$ | $f : p \mapsto \bot$ |
| ??? | $\forall x \in P, Q(x)$ | ??? | $f : x \mapsto q(x)$ |
| ??? | $\exists x \in P, Q(x)$ | ??? | $(x, q(x))$ |

# 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` $b$ | $(1, p), (2, q)$ |
| $\alpha \to \beta$ | $P \to Q$ | $a \mapsto b$ | $f : p \mapsto q$ |
| `unit` | true | $*$ | $\exists$ |
| `none` | false | ✗ | $\nexists$ |
| $\alpha \to$ `none` | $\neg P$ | $a \mapsto \lightning$ | $f : p \mapsto \bot$ |
| $(a : \alpha) \to \beta$ | $\forall x \in P, Q(x)$ | $a \mapsto b(a)$ | $f : x \mapsto q(x)$ |
| ??? | $\exists x \in P, Q(x)$ | ??? | $(x, q(x))$ |

# 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` $b$ | $(1, p), (2, q)$ |
| $\alpha \to \beta$ | $P \to Q$ | 🐍 $: a \mapsto b$ | $f : p \mapsto q$ |
| `unit` | true | $*$ | $\exists$ |
| `none` | false | ✗ | $\nexists$ |
| $\alpha \to$ `none` | $\neg P$ | 🐍 $: a \mapsto \lightning$ | $f : p \mapsto \bot$ |
| 🐍$(a : \alpha) \to \beta$ | $\forall x \in P, Q(x)$ | 🐍 $: a \mapsto b(a)$ | $f : x \mapsto q(x)$ |
| ??? | $\exists x \in P, Q(x)$ | ??? | $(x, q(x))$ |

$\Downarrow$

checking proof correctness $\rightleftharpoons$ type-checking term

# Demo!

# Resources

- Rocq
  - 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:

$$\texttt{inductive equal} : \prod(\alpha : \mathbb{T}) \to \alpha \to \alpha \to \mathbb{T}$$
$$\mid\ \texttt{refl} : \prod(\alpha : \mathbb{T}) \to \prod(a : \alpha) \to \texttt{equal}\ \alpha\ a\ a$$

# The equality type?

*What type corresponds to the equality proposition?*

We actually can just make an inductive type for this:

$$\texttt{inductive equal} : (\alpha : \mathbb{T}) \to \alpha \to \alpha \to \mathbb{T}$$
$$\mid \texttt{refl} : (\alpha : \mathbb{T}) \to (a : \alpha) \to \texttt{equal}\ \alpha\ a\ a$$

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 $\mathbb{N}$:

$$\begin{aligned}&\texttt{inductive leq} : (m : \mathbb{N}) \to (n : \mathbb{N}) \to \mathbb{T}\\ &\mid \texttt{leq-O} : (n : \mathbb{N}) \to \texttt{leq}\ O\ n\\ &\mid \texttt{leq-S} : (m : \mathbb{N}, n : \mathbb{N}, H : \texttt{leq}\ m\ n) \to \texttt{leq}\ (S\ m)\ (S\ n)\end{aligned}$$

# 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 $\mathbb{N}$:

$$\begin{aligned}&\texttt{inductive leq} : \prod(m : \mathbb{N}) \to \prod(n : \mathbb{N}) \to \mathbb{T}\\ &\mid \texttt{leq-O} : \prod(n : \mathbb{N}) \to \texttt{leq}\ O\ n\\ &\mid \texttt{leq-S} : \prod(m : \mathbb{N}, n : \mathbb{N}, H : \texttt{leq}\ m\ n) \to \texttt{leq}\ (S\ m)\ (S\ n)\end{aligned}$$

Here, we can get a proof of this proposition if we either generate it directly from the fact that all naturals are $\geq$ O, 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:

$$
\begin{aligned}
&\texttt{struct group} : (\alpha : \mathbb{T}) \to \mathbb{T} \quad \{ \\
&\mid \quad \texttt{mul} : \alpha \to \alpha \to \alpha \\
&\mid \quad \texttt{id} : \alpha \\
&\mid \quad \texttt{inv} : \alpha \to \alpha \\
&\mid \quad \texttt{mul\_id} : (x : \alpha) \to \texttt{mul}\ x\ \texttt{id} = x \ldots \}
\end{aligned}
$$

# 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:

$$
\begin{aligned}
&\texttt{struct group} : \Pi(\alpha : \mathbb{T}) \to \mathbb{T} \quad \{ \\
&\mid \quad \texttt{mul} : \alpha \to \alpha \to \alpha \\
&\mid \quad \texttt{id} : \alpha \\
&\mid \quad \texttt{inv} : \alpha \to \alpha \\
&\mid \quad \texttt{mul\_id} : \Pi(x : \alpha) \to \texttt{mul}\ x\ \texttt{id} = x \ldots \}
\end{aligned}
$$

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.