The Correspondence Between Proofs and Programs

And how Mathematics informs Programming Language Design

Jules Poon

2024 Dec

whoami

- Jules
- Undergrad
- Interested in Programming Languages and Math
  - Currently interested in Algebra
  - Worked briefly on CPython’s JIT

whoami

- Jules
- Undergrad
- Interested in Programming Languages and Math
  - Currently interested in Algebra
  - Worked briefly on CPython's JIT
- **Will do SWE for money** { available for summer intern 2025 hmu }
  - juliapoo.github.io { full of cool stuff }

whoami

- Jules
- Undergrad
- Interested in Programming Languages and Math
  - Currently interested in Algebra
  - Worked briefly on CPython's JIT
- **Will do SWE for money** { available for summer intern 2025 hmu }
  - juliapoo.github.io { full of cool stuff }

Special thanks to @Patricia { linkedin.com/in/patmloi } for her invaluable feedback, without which this would have been a completely different talk.

Curry Howard Correspondence

$$\text{Mathematical Proofs} \iff \text{Programs}$$

- First noticed by Haskell Curry in 1934, before computers, or programming as we know today

Curry Howard Correspondence

$$\text{Mathematical Proofs} \iff \text{Programs}$$

- First noticed by Haskell Curry in 1934, before computers, or programming as we know today
- { personal opinion } One of the biggest bridge connecting Mathematics and Computer Science

Curry Howard Correspondence

$$\text{Mathematical Proofs} \iff \text{Programs}$$

- First noticed by Haskell Curry in 1934, before computers, or programming as we know today
- { personal opinion } One of the biggest bridge connecting Mathematics and Computer Science
- Majority of the writing on this is targeted at Mathematicians, not Computer People.

Curry Howard Correspondence

$$\text{Mathematical Proofs} \iff \text{Programs}$$

Curry Howard Correspondence

$$\text{Mathematical Proofs} \iff \text{Programs}$$

Uses of the Correspondence

Powers Interactive Theorem Provers

- **For Mathematicians**: Verifies a mathematical argument is sound
- **For Computer People**: Formal Verification of software/hardware

Introduction

Curry Howard Correspondence

$$\text{Mathematical Proofs} \iff \text{Programs}$$

Uses of the Correspondence

Powers Interactive Theorem Provers

- **For Mathematicians**: Verifies a mathematical argument is sound
- **For Computer People**: Formal Verification of software/hardware
  - Proving an implementation is correct for all inputs
  - Used in safety critical software (like airbags to ensure compliance)
  - Intel uses it to verify microcode

Introduction

Interactive Theorem Provers

Questions:

- What is a Mathematical Proof?

Introduction

Interactive Theorem Provers

Questions:

- What is a Mathematical Proof?
- What does it mean for reasoning to be sound?

Interactive Theorem Provers

Questions:

- What is a Mathematical Proof?
- What does it mean for reasoning to be sound?
- How to **program a computer to verify a proof's correctness**?

Interactive Theorem Provers

Questions:

- What is a Mathematical Proof?
- What does it mean for reasoning to be sound?
- How to **program a computer to verify a proof's correctness**?

Example of a Proposition

How can we prove the following?

$$\forall A, B \text{ boolean} : (A \wedge B) \rightarrow (B \wedge A)$$

$(A$ and $B)$ is equivalent to $(B$ and $A)$

Introduction

To Prove

$$\forall A, B \text{ boolean} : (A \land B) \leftrightarrow (B \land A)$$

To Prove

$$\forall A, B \text{ boolean} : (A \land B) \leftrightarrow (B \land A)$$

Attempt 1: Bruteforce (the usual way we test software)

A, B can either be `True` or `False`. We can try all 4 possibilities and show that the expression is always `True`.

To Prove

$$\forall A, B \text{ boolean} : (A \wedge B) \leftrightarrow (B \wedge A)$$

Attempt 1: Bruteforce (the usual way we test software)

A, B can either be `True` or `False`. We can try all 4 possibilities and show that the expression is always `True`.

A	B	$A \wedge B$	$B \wedge A$	
`False`	`False`	`False`	`False`	✓
`False`	`True`	`False`	`False`	✓
`True`	`False`	`False`	`False`	✓
`True`	`True`	`True`	`True`	✓

Introduction

Problem: What if the domain is infinite?

$$\forall x, y \in \mathbb{Z}_{\geq 0} : x + y \geq x$$

For any x, y integers ≥ 0, $x + y \geq x$

Problem: What if the domain is infinite?

$$\forall x, y \in \mathbb{Z}_{\geq 0} : x + y \geq x$$

For any x, y integers ≥ 0, $x + y \geq x$

- We can no longer try every possible value

Problem: What if the domain is infinite?

$$\forall x, y \in \mathbb{Z}_{\geq 0} : x + y \geq x$$

For any x, y integers ≥ 0, $x + y \geq x$

- We can no longer try every possible value
- We need to program the computer to reason.

To Prove

$$\forall x, y \in \mathbb{Z}_{\geq 0} : x + y \geq x$$

To Prove

$$\forall x, y \in \mathbb{Z}_{\geq 0} : x + y \geq x$$

Attempt 2: Define the constructs to the computer and compose theorems

The computer needs to know

- What $\mathbb{Z}_{\geq 0}$ is.
  - Understand all of its properties and statements you can say about it
- What $\forall$, $+$, $\geq$ means
- How to combine reasoning steps together in a sound way

To Prove

$$\forall x, y \in \mathbb{Z}_{\geq 0} : x + y \geq x$$

Attempt 2: Define the constructs to the computer and compose theorems

The computer needs to know

- What $\mathbb{Z}_{\geq 0}$ is.
  - Understand all of its properties and statements you can say about it
- What $\forall$, $+$, $\geq$ means
- How to combine reasoning steps together in a sound way

Very difficult problem! Gives rise to the idea of a Proof System.

Introduction

Proof System

A framework which one can prove statements.

Proof System

A framework which one can prove statements.

Consists of:

1. Formal Language: A language to write formulas in.
2. Rules of Inference: How to reason to prove statements.
3. Axioms: Assumptions, statements assumed true.

Example of a (non-trivial but easy) Mathematical Proof

Proposition: $\sqrt{2}$ is irrational (cannot be a fraction)

Example of a (non-trivial but easy) Mathematical Proof

Proposition: $\sqrt{2}$ is irrational (cannot be a fraction)

Proof:

1. Suppose $\sqrt{2} = \frac{a}{b}$ in simplified form.
2. Then $2b^2 = a^2$.
3. Since $2b^2$ is even, a^2 is even, so a is even.
4. Since a^2 is even, $2b^2$ is divisible by 4, so b^2 is even, and b has to be even.
5. Hence both a and b are even.
6. But $\frac{a}{b}$ is supposed to be simplified form, a contradiction!
7. Hence our assumption that $\sqrt{2} = \frac{a}{b}$ is not true!

Introduction

Reflection Questions:

- Can you figure out what you **need to define** to a computer to understand this proof?
- Can you figure how to encode ways one can **compose reasoning**?

Taster in what a Computer needs: { a rabbithole everywhere }

- What is a natural number?

Peano's 6 Axioms:

1. $\forall x, 0 \neq S(x)$
2. $\forall x, y(S(x) = S(y) \Rightarrow x = y)$
3. $\forall x(x + 0 = x)$

...

- What is an integer?

$$\mathbb{Z} \cong \mathbb{N}^2 / \sim, \text{where } (a, b) \sim (c, d) \text{ iff } a + d = b + c$$

- What is a fraction?

$$\mathbb{Q} \cong \mathbb{Z}^2 / \sim, \text{where } (a, b) \sim (c, d) \text{ iff } ad = bc$$

Work not in Proof Systems but in Programs

Work not in Proof Systems but in Programs

Proof System Side	**Programming Side**
Formula	Type
Proof	Term { valid program }
Formula has a Proof	Type has a Term
Simplification of Proof	Running of Term

Work not in Proof Systems but in Programs

Proof System Side	**Programming Side**
Formula	Type
Proof	Term { valid program }
Formula has a Proof	Type has a Term
Simplification of Proof	Running of Term

If we want to verify the Proof of a Formula,

1. Convert Formula to a Type in the programming language.
2. Convert Proof to a Term in the programming language.
3. Computer verifies the Term has the correct Type in the language.

For every **Proof System**,
we can define a **Programming Language** where
finding a **Proof**

$\iff$

finding a **Term** with the correct **Type**

Table of contents

Demonstrate the correspondence between Proofs and Programs:

We'll be constructing the most basic Programming Language and Proof System, and demonstrate a clear linkage between the two.

Table of contents

Demonstrate the correspondence between Proofs and Programs:

We'll be constructing the most basic Programming Language and Proof System, and demonstrate a clear linkage between the two.

1. Untyped Lambda Calculus { **programming language** }
2. Simply Typed Lambda Calculus { **programming language** }
3. Proof System ($\land$ and $\rightarrow$) { **proof system** }
4. Curry Howard Correspondence { **proof** $\leftrightarrow$ **programs** }
5. What now?

Untyped Lambda Calculus { programming language }

Untyped Lambda Calculus { programming language }

Lambda Expressions { in Python }

```
f = lambda x: x                    def f(x): return x
```

Lambda Expressions { in Untyped Lambda Calculus }

$\lambda x f.f(x)$ is a Term corresponding to `lambda x: lambda f: f(x)`

Untyped Lambda Calculus { programming language }

Lambda Expressions { in Python }

```
f = lambda x: x                    def f(x): return x
```

Lambda Expressions { in Untyped Lambda Calculus }

$\lambda xf.f(x)$ is a Term corresponding to `lambda x: lambda f: f(x)`

$$\lambda \underbrace{xf}_{\text{arguments}} . \underbrace{f(x)}_{\text{operation}}$$

Untyped Lambda Calculus { programming language }

Lambda Expressions { in Python }

```python
f = lambda x: x                  def f(x): return x
```

Lambda Expressions { in Untyped Lambda Calculus }

$\lambda x f . f(x)$ is a Term corresponding to `lambda x: lambda f: f(x)`

$$\lambda \underbrace{x f}_{\text{arguments}} . \underbrace{f(x)}_{\text{operation}}$$

- Arguments are Curried: $\lambda x f.\ \mathrm{op} \cong \lambda x.(\lambda f.\ \mathrm{op})$
  - Python: `lambda x,f: <op>` $\rightarrow$ `lambda x: lambda f: <op>`
  - Every “function” has 1 argument and 1 return value

Two Concepts of Untyped Lambda Calculus

$$\underbrace{\lambda x f}_{\text{abstraction}} \;.\; \underbrace{f(x)}_{\text{application}}$$

1. Abstraction aka function definition { Introduction of abstraction }
2. Application aka function calling { Elimination of abstraction }

Untyped Lambda Calculus { programming language }

Two Concepts of Untyped Lambda Calculus

$$\underbrace{\lambda x f}_{\text{abstraction}} \;.\; \underbrace{f(x)}_{\text{application}}$$

1. Abstraction aka function definition { Introduction of abstraction }
2. Application aka function calling { Elimination of abstraction }

Language Features should come in pairs of Introduction and Elimination.

- Introduction: Definition
- Elimination: Consequences of Definition

Gives a language some nice properties.

Eliminating Brackets

- Brackets `()` are used to indicate Order of Operations
- Impose rules to avoid writing extra brackets for clarity

Untyped Lambda Calculus { programming language }

Eliminating Brackets

- Brackets () are used to indicate Order of Operations
- Impose rules to avoid writing extra brackets for clarity

Rules of Order of Operation

- Application is left-associative
  - MNP is $(M(N))(P)$

Eliminating Brackets

- Brackets `()` are used to indicate Order of Operations
- Impose rules to avoid writing extra brackets for clarity

Rules of Order of Operation

- Application is left-associative
  - MNP is $(M(N))(P)$
- Application has higher precedence than Abstraction { like $\times$ vs $+$ }
  - $\lambda x.MN$ is $\lambda x.(MN)$ and **not** $(\lambda x.M)N$

Eliminating Brackets

- Brackets `()` are used to indicate Order of Operations
- Impose rules to avoid writing extra brackets for clarity

Rules of Order of Operation

- Application is left-associative
  - MNP is $(M(N))(P)$
- Application has higher precedence than Abstraction { like $\times$ vs $+$ }
  - $\lambda x.MN$ is $\lambda x.(MN)$ and **not** $(\lambda x.M)N$

$$\lambda x.xz(\lambda y.xy) \iff \lambda x.(x(z)(\lambda y.x(y)))$$

Executing an example program

1. $(\lambda xy.y)(\lambda x.x)(\lambda x.x)$

Executing an example program

1. $(\lambda \underbrace{x}_{\text{variable}} y.y) \underbrace{(\lambda x.x)}_{\text{argument}} (\lambda x.x)$

- We replace **variable** x in the body of $(\lambda xy.y)$ with the **argument** $(\lambda x.x)$.
  - Since the body of $(\lambda xy.y)$ is $\lambda y.y$, which does not contain x
  - We simply return the body $\lambda y.y$.
  - $(\lambda xy.y)(\lambda x.x) \rightarrow (\lambda y.y)$
- In Python: `(lambda x: lambda y: y)(lambda x: x) -> (lambda y: y)`

Executing an example program

1. $(\lambda xy.y)(\lambda x.x)(\lambda x.x)$
2. $(\lambda y.y)(\lambda x.x)$

Executing an example program

1. $(\lambda xy.y)(\lambda x.x)(\lambda x.x)$
2. $(\lambda \underbrace{y}_{\text{variable}} .y) \underbrace{(\lambda x.x)}_{\text{argument}}$

- We replace **variable** y in the body of $(\lambda y.y)$ with the **argument** $(\lambda x.x)$.
  - Since the body of $(\lambda y.y)$ is y,
  - We replace the body $y \to \lambda x.x$ and return it.
  - $(\lambda y.y)(\lambda x.x) \to (\lambda x.x)$
- In Python: `(lambda y: y)(lambda x: x) -> (lambda x: x)`

Executing an example program

1. $(\lambda xy.y)(\lambda x.x)(\lambda x.x)$
2. $(\lambda y.y)(\lambda x.x)$
3. $\lambda x.x$

Executing an example program

1. $(\lambda xy.y)(\lambda x.x)(\lambda x.x)$
2. $(\lambda y.y)(\lambda x.x)$
3. $\lambda x.x$ { stop when we can't perform Application }

When we can't reduce a term anymore, we call the term normal.

- We write $M \twoheadrightarrow N$ if we can reduce a term M to a term N.
- $(\lambda xy.y)(\lambda x.x)(\lambda x.x) \twoheadrightarrow \lambda x.x$

Executing an example program

1. $(\lambda x.xx)(\lambda x.xx)$

Executing an example program

1. $(\lambda x.xx)(\lambda x.xx)$
2. $(\lambda x.xx)(\lambda x.xx)$
3. ...

Executing an example program

1. $(\lambda x.xx)(\lambda x.xx)$
2. $(\lambda x.xx)(\lambda x.xx)$
3. ...

Program above does not converge. It has no normal form.

- Later, we'll see that Types avoid such Terms that do not converge.

Simply Typed Lambda Calculus { programming language }

Simply Typed Lambda Calculus { programming language }

Simply Typed Lambda Calculus

Lambda Calculus but every Term is Typed

- Term t has a Type T, written as $t : T$. { like Python's Type Annotations }
- Later, we'll map every Type into a Formula.

Simply Typed Lambda Calculus

Lambda Calculus but every Term is Typed

- Term t has a Type T, written as $t : T$. { like Python's Type Annotations }
- Later, we'll map every Type into a Formula.

Types

- Atomic Types $A, B, \ldots$. Basic building blocks for Types.
- Composite Types. Types built-upon other Types.
  - { we'll see them later } $A \to B$, $A \times B$

Typing Rules

Rule λI: If $y : B$, then $\lambda x^A.y : A \to B$

- x^A states x variable is of type A.
- λI is rule for Abstraction (I for Introduction)

Rule λE: If $f : A \to B$ and $x : A$, then $fx : B$

- λE is rule for Application (E for Elimination)

Typing Rules

Rule λI: If $y : B$, then $\lambda x^A.y : A \to B$

- x^A states x variable is of type A.
- λI is rule for Abstraction (I for Introduction)

Notation

$$\frac{\text{Premises}}{\text{Conclusion}}\ \text{Name-of-rule} \implies \frac{\begin{array}{c}[x : A]^x \ \text{if we can assume that } x : A, \\ \vdots \\ y : B \ \text{we'll get } y : B,\end{array}}{\underbrace{\lambda x^A.y : A \to B}_{\text{then } \lambda x^A.y:A\to B}}\ \underbrace{\lambda I^x}_{\text{name of rule}}$$

Simply Typed Lambda Calculus { programming language }

Typing Rules

$$\dfrac{\begin{array}{c}[x : A]^x \\ \vdots \\ y : B\end{array}}{\lambda x^A . y : A \to B}\lambda I^x \qquad \dfrac{f : A \to B \quad x : A}{f x : B}\lambda E$$

Examples

$$\lambda x^A f^{A \to B}.fx \ : \ A \to ((A \to B) \to B)$$

We take $\to$ to be right-associative:

- $A \to ((A \to B) \to B)$ is written as $A \to (A \to B) \to B$
- Functional programmers might recognise this notation for typing functions

Examples

$$\lambda x^A f^{A \to B}.fx \ : \ A \to ((A \to B) \to B)$$

We can form a Justification Tree for the Type by composing typing rules.

Examples

$$\lambda x^A f^{A \to B}.fx \ : \ A \to ((A \to B) \to B)$$

$$[f : A \to B]^{f} \qquad [x : A]^{x}$$

- We first try to type the body fx
- We know we can assume $f : A \to B$ and $x : A$.
- We'll track these assumptions as f and x.

Examples

$$\lambda x^A f^{A \to B}.fx \;:\; A \to ((A \to B) \to B)$$

$$\dfrac{[f : A \to B]^{f} \qquad [x : A]^{x}}{fx : B}\lambda E$$

- Next we can apply rule λE to type $fx : B$

Simply Typed Lambda Calculus { programming language }

Examples

$$\lambda x^A f^{A \to B}.fx \ : \ A \to ((A \to B) \to B)$$

$$\dfrac{\dfrac{[f : A \to B]^{f} \qquad [x : A]^{x}}{fx : B}\lambda E}{\lambda f^{A \to B}.fx : (A \to B) \to B}\lambda I^{f}$$

- Next we can apply rule λI^f to consume the assumption $[f : A \to B]^f$.

Simply Typed Lambda Calculus { programming language }

Examples

$$\lambda x^A f^{A\to B}.fx \;:\; A \to ((A \to B) \to B)$$

$$\dfrac{\dfrac{\dfrac{[f : A \to B]^{f} \qquad [x : A]^{x}}{fx : B}\lambda E}{\lambda f^{A\to B}.fx : (A \to B) \to B}\lambda I^{f}}{\lambda x^A f^{A\to B}.fx : A \to (A \to B) \to B}\lambda I^{x}$$

- Next we can apply rule λI^x to consume the assumption $[x : A]^x$.

Examples

$$\lambda x^A f^{A\to B}.fx \;:\; A \to ((A \to B) \to B)$$

$$\dfrac{\dfrac{\dfrac{[f : A \to B]^{f} \qquad [x : A]^{x}}{fx : B}\lambda E}{\lambda f^{A\to B}.fx : (A \to B) \to B}\lambda I^{f}}{\lambda x^A f^{A\to B}.fx : A \to (A \to B) \to B}\lambda I^{x}$$

- $[f : A \to B]^f$ must accompany a λI^f rule.
- $[x : A]^x$ must accompany a λI^x rule.

Simply Typed Lambda Calculus { **programming language** }

It'll be nice if our langauge can return more than 1 value.

It'll be nice if our langauge can return more than 1 value.

Adding Pairs in the Language

- $\langle x, y \rangle$ introduces a Pair
- π_1 and π_2 eliminates a Pair:
  - $\pi_1(\langle x, y \rangle) = x$, $\pi_2(\langle x, y \rangle) = y$

It'll be nice if our langauge can return more than 1 value.

Adding Pairs in the Language

- $\langle x, y \rangle$ introduces a Pair
- π_1 and π_2 eliminates a Pair:
  - $\pi_1(\langle x, y \rangle) = x$, $\pi_2(\langle x, y \rangle) = y$

Typing Rules for Pairs: Product Types

$$\frac{X : A \qquad Y : B}{\langle X, Y \rangle : \underbrace{A \times B}_{\text{product type}}} \pi I \qquad \frac{L : A \times B}{\pi_1 L : A} \pi_1 E \qquad \frac{L : A \times B}{\pi_2 L : B} \pi_2 E$$

Example with Product Types

Function that reverses the order of a Pair:

$$\lambda x^{A \times B}.\langle \pi_2 x, \pi_1 x \rangle \;\; : \;\; A \times B \to B \times A$$

Example with Product Types

Function that reverses the order of a Pair:

$$\lambda x^{A\times B}.\langle \pi_2 x, \pi_1 x\rangle \;:\; A\times B \to B\times A$$

$$\dfrac{\dfrac{\dfrac{[x : A\times B]^{x}}{\pi_2 x : B}\,\pi_2 E \qquad \dfrac{[x : A\times B]^{x}}{\pi_1 x : A}\,\pi_1 E}{\langle \pi_2 x, \pi_1 x\rangle : B\times A}\,\pi I}{\lambda x^{A\times B}.\langle \pi_2 x, \pi_1 x\rangle \;:\; A\times B \to B\times A}\,\lambda I^{x}$$

Simply Typed Lambda Calculus { programming language }

Typing terms

Given an untyped term, we can assign Types to make the program valid in the Simply Typed Lambda Calculus

- E.g., Function is applied with the correct types.

Typing terms

Given an untyped term, we can assign Types to make the program valid in the Simply Typed Lambda Calculus

- E.g., Function is applied with the correct types.
- $(\lambda x^{A}.x)\,\underbrace{(\lambda x^{A}.x)}_{\text{Type } A\to A}$ is not correctly typed. (`TypeError`)

Typing terms

Given an untyped term, we can assign Types to make the program valid in the Simply Typed Lambda Calculus

- E.g., Function is applied with the correct types.
- $(\lambda x^{A}.x)\ \underbrace{(\lambda x^{A}.x)}_{\text{Type } A\to A}$ is not correctly typed. (`TypeError`)
- $(\lambda x^{A\to A}.x)(\lambda x^{A}.x)$ is correctly typed.

Typing terms

Given an untyped term, we can assign Types to make the program valid in the Simply Typed Lambda Calculus

- E.g., Function is applied with the correct types.
- $(\lambda x^{A}.x)\underbrace{(\lambda x^{A}.x)}_{\text{Type } A\to A}$ is not correctly typed. (`TypeError`)
- $(\lambda x^{A\to A}.x)(\lambda x^{A}.x)$ is correctly typed.

Our example program $(\lambda xy.y)(\lambda x.x)(\lambda x.x)$ can be typed as such:

$$(\lambda x^{A\to A}y^{A\to A}.y)(\lambda x^{A}.x)(\lambda x^{A}.x)$$

Are all Untyped Lambda terms Typeable (in our language)?

No. $(\lambda x.xx)(\lambda x.xx)$ is not typeable.

- Intuition: It runs forever, we need a recursive type to represent such terms. This feature does not exist in our very simple language.

Are all Untyped Lambda terms Typeable (in our language)?

No. $(\lambda x.xx)(\lambda x.xx)$ is not typeable.

- Intuition: It runs forever, we need a recursive type to represent such terms. This feature does not exist in our very simple language.

Types restricts what are considered programs.

- Intended. Gives our language some nice properties.

Simply Typed Lambda Calculus is Strongly Normalising

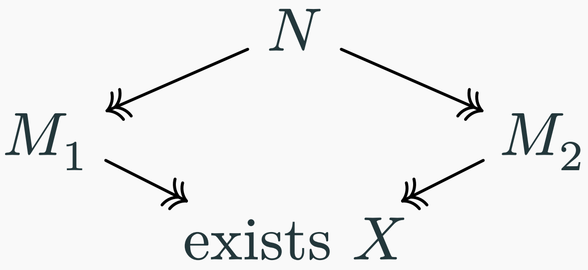
- Informal: All programs finish evaluating in finite steps.

Simply Typed Lambda Calculus is Strongly Normalising

- Informal: All programs finish evaluating in finite steps.

Simply Typed Lambda Calculus has the Church-Rosser property

- Informal: No matter how we evaluate, we'll get the same normal form.
- If $N \twoheadrightarrow M_1$ and $N \twoheadrightarrow M_2$, then there exists an X with $M_1 \twoheadrightarrow X$ and $M_2 \twoheadrightarrow X$.



All programs in

Simply Typed Lambda Calculus

evaluate in finite steps to a

unique normal form

Simply Typed Lambda Calculus { **programming language** }

Determine if two programs are equivalent:

1. Evaluate both programs (finishes in finite steps)
2. Compare results (equal up to variable renaming)

Simply Typed Lambda Calculus { programming language }

Determine if two programs are equivalent:

1. Evaluate both programs (finishes in finite steps)
2. Compare results (equal up to variable renaming)

Example:

$$(\lambda x^{A\to A} y^{A\to A}.y)(\lambda x^A.x)(\lambda x^A.x) \twoheadrightarrow \lambda x^A.x$$

$$(\lambda x^{A\to A}.x)(\lambda z^A.z) \twoheadrightarrow \lambda z^A.z$$

Since $\lambda x^A.x$ and $\lambda z^A.z$ are equal up to variable renaming, both programs are equivalent.

Proof System ($\wedge$ and $\to$) { proof system }

Proof System ($\land$ and $\to$) { proof system }

Language for Formulas

- Consists of atomic (hypothesis) represented as letters $A, B, C, \ldots$
  - atomics can either be `True` or `False`
- Logical connectors $\underbrace{\land}_{\text{and}}$ and $\underbrace{\to}_{\text{implies}}$, and () to indicate order of operations

E.g., $A \to B \to (B \land A)$ is a Formula:

- If we assume A, and we assume B, then $(B \land A)$.

Rules of Inference

For ∧ connective:

$$\frac{A \quad B}{A \wedge B}\wedge I \qquad \frac{A \wedge B}{A}\wedge_1 E \qquad \frac{A \wedge B}{B}\wedge_2 E$$

Rules of Inference

For ∧ connective:

$$\frac{A \quad B}{A \land B}\land I \qquad \frac{A \land B}{A}\land_1 E \qquad \frac{A \land B}{B}\land_2 E$$

For → connective:

if by assuming A (track hypothesis with x) $[A]^x$

$\vdots$

we can conclude B

then, $A \to B$ via rule I^x

$$\frac{\begin{array}{c}[A]^x \\ \vdots \\ B\end{array}}{A \to B}\to I^x \qquad \frac{A \to B \quad A}{B}\to E$$

Proof System ($\wedge$ and $\rightarrow$) { proof system }

Example: Prove that $(A \wedge B) \rightarrow (B \wedge A)$

Example: Prove that $(A \wedge B) \rightarrow (B \wedge A)$

$$\frac{[A \wedge B]^x}{B} \wedge_2 E \qquad \frac{[A \wedge B]^x}{A} \wedge_1 E$$

- Lets assume $(A \wedge B)$ is `true` (we'll track this hypothesis with x).
- From inference rules $\wedge_2 E$ and $\wedge_1 E$, we'll obtain B and A is `true`.

Example: Prove that $(A \wedge B) \rightarrow (B \wedge A)$

$$\dfrac{\dfrac{[A \wedge B]^{x}}{B}\,\wedge_2 E \qquad \dfrac{[A \wedge B]^{x}}{A}\,\wedge_1 E}{B \wedge A}\,\wedge I$$

- From $\wedge I$, we can conclude $B \wedge A$ is `true`

Example: Prove that $(A \wedge B) \rightarrow (B \wedge A)$

$$\dfrac{\dfrac{\dfrac{[A \wedge B]^{x}}{B}\ \wedge_2 E \qquad \dfrac{[A \wedge B]^{x}}{A}\ \wedge_1 E}{B \wedge A}\ \wedge I}{(A \wedge B) \rightarrow (B \wedge A)}\ \rightarrow I^{x}$$

- Finally, with rule, $\rightarrow I^{x}$ we consume the hypothesis $[A \wedge B]^{x}$.

Curry Howard Correspondence { proof $\leftrightarrow$ programs }

Correspondence

Proof that $(A \wedge B) \rightarrow (B \wedge A)$

$$\dfrac{\dfrac{\dfrac{[A \wedge B]^x}{B}\,\wedge_2 E \qquad \dfrac{[A \wedge B]^x}{A}\,\wedge_1 E}{B \wedge A}\,\wedge I}{(A \wedge B) \rightarrow (B \wedge A)}\,{\rightarrow}I^x$$

Type of $\lambda x^{A \times B}.\langle \pi_2 x, \pi_1 x \rangle$

$$\dfrac{\dfrac{\dfrac{[x : A \times B]^x}{\pi_2 x : B}\,\pi_2 E \qquad \dfrac{[x : A \times B]^x}{\pi_1 x : A}\,\pi_1 E}{\langle \pi_2 x, \pi_1 x \rangle : B \times A}\,\pi I}{\lambda x^{A \times B}.\langle \pi_2 x, \pi_1 x \rangle \ : \ A \times B \rightarrow B \times A}\,\lambda I^x$$

Correspondence

Proof that $(A \wedge B) \to (B \wedge A)$

$$\dfrac{\dfrac{\dfrac{[A \wedge B]^x}{B}\,\wedge_2 E \qquad \dfrac{[A \wedge B]^x}{A}\,\wedge_1 E}{B \wedge A}\,\wedge I}{(A \wedge B) \to (B \wedge A)}\,\to I^x$$

Type of $\lambda x^{A \times B}.\langle \pi_2 x, \pi_1 x \rangle$

$$\dfrac{\dfrac{\dfrac{[x : A \times B]^x}{\pi_2 x : B}\,\pi_2 E \qquad \dfrac{[x : A \times B]^x}{\pi_1 x : A}\,\pi_1 E}{\langle \pi_2 x, \pi_1 x \rangle : B \times A}\,\pi I}{\lambda x^{A \times B}.\langle \pi_2 x, \pi_1 x \rangle \;:\; A \times B \to B \times A}\,\lambda I^x$$

Correspondence

Proof that $(A \wedge B) \rightarrow (B \wedge A)$

$$\dfrac{\dfrac{\dfrac{[A \wedge B]^x}{B}\wedge_2 E \qquad \dfrac{[A \wedge B]^x}{A}\wedge_1 E}{B \wedge A}\wedge I}{(A \wedge B) \rightarrow (B \wedge A)}\rightarrow I^x$$

Type of $\lambda x^{A \times B}.\langle \pi_2 x, \pi_1 x \rangle$

$$\dfrac{\dfrac{\dfrac{[x : A \times B]^x}{\pi_2 x : B}\pi_2 E \qquad \dfrac{[x : A \times B]^x}{\pi_1 x : A}\pi_1 E}{\langle \pi_2 x, \pi_1 x \rangle : B \times A}\pi I}{\lambda x^{A \times B}.\langle \pi_2 x, \pi_1 x \rangle \ : \ A \times B \rightarrow B \times A}\lambda I^x$$

Correspondence

Formulae	**Types**
Atomic hypothesis $A, B, \ldots$	Atomic types $A, B, \ldots$
Logical connector $\rightarrow$	Function type $\rightarrow$
Logical connector $\wedge$	Product Type $\times$

Correspondence

Proofs	**Programs**
Inference for $\to I^x$ and $\to E$	Types for λI^x and λE

$$\dfrac{\begin{array}{c}[A]^x \\ \vdots \\ B\end{array}}{A \to B} \to I^x \qquad \dfrac{A \to B \quad A}{B} \to E$$

$$\dfrac{\begin{array}{c}[x : A]^x \\ \vdots \\ y : B\end{array}}{\lambda x^A . y : A \to B} \lambda I^x \qquad \dfrac{f : A \to B \quad x : A}{f x : B} \lambda E$$

Correspondence

Proofs	**Programs**
Inference for $\wedge I$ and $\wedge_1 E$ and $\wedge_2 E$	Types for $\wedge I$ and $\wedge_1 E$ and $\wedge_2 E$

$$\frac{A \quad B}{A \wedge B}\wedge I$$

$$\frac{A \wedge B}{A}\wedge_1 E \qquad \frac{A \wedge B}{B}\wedge_2 E$$

$$\frac{X : A \quad Y : B}{\langle X, Y \rangle : A \times B}\pi I$$

$$\frac{L : A \times B}{\pi_1 L : A}\pi_1 E \qquad \frac{L : A \times B}{\pi_2 L : B}\pi_2 E$$

Correspondence

Proofs	Programs
Normalising (simplifying) of Proof	Normalising (running) of Program
There's a finite algorithm that says if two proofs are equivalent.	Simply Typed Lambda Calculus is Strongly Normalising and has the Church Rossier Property. So, there's a finite algorithm that can determine if two Terms are equivalent.

Correspondence

Proofs	**Programs**
Normalised proofs of a formula only uses "concepts" present in the formula.	Language features comes in pairs of Introduction and Elimination
E.g., Proof of $A \to (A \to B) \to B$ does not need $\land$.	

Curry Howard Correspondence { **proof $\leftrightarrow$ programs** }

Proving that $A \to (A \to B) \to B$

Curry Howard Correspondence { proof $\leftrightarrow$ programs }

Proving that $A \to (A \to B) \to B$

1. Convert formula $A \to (A \to B) \to B$ into the type $A \to (A \to B) \to B$

Curry Howard Correspondence { proof $\leftrightarrow$ programs }

Proving that $A \to (A \to B) \to B$

1. Convert formula $A \to (A \to B) \to B$ into the type $A \to (A \to B) \to B$
2. Find a term (program) that has the type: $\lambda x^A f^{A \to B}.fx$

Curry Howard Correspondence { proof $\leftrightarrow$ programs }

Proving that $A \to (A \to B) \to B$

1. Convert formula $A \to (A \to B) \to B$ into the type $A \to (A \to B) \to B$
2. Find a term (program) that has the type: $\lambda x^A f^{A \to B}.fx$
3. Convert the justification tree for the type of the term into a proof.

$$\dfrac{\dfrac{\dfrac{[f : A \to B]^f \qquad [x : A]^x}{fx : B}\lambda E}{\lambda f^{A\to B}.fx : (A \to B) \to B}\lambda I^f}{\lambda x^A f^{A \to B}.fx : A \to (A \to B) \to B}\lambda I^x \implies \dfrac{\dfrac{\dfrac{[A \to B]^f \qquad [A]^x}{B}\to E}{(A \to B) \to B}\to I^f}{A \to (A \to B) \to B}\to I^x$$

Curry Howard Correspondence { proof $\leftrightarrow$ programs }

Simplifying a proof that $A \to B \to B \wedge A$

Simplifying a proof that $A \to B \to B \wedge A$

Roundabout proof:

1. Assume A and B, we have $A \wedge B$ { by rule $\wedge I$ }.
2. Since we've previously shown that $A \wedge B \to B \wedge A$, the result holds.

Simplifying a proof that $A \to B \to B \wedge A$

Roundabout proof:

1. Assume A and B, we have $A \wedge B$ { by rule $\wedge I$ }.
2. Since we've previously shown that $A \wedge B \to B \wedge A$, the result holds.

Proof corresponds to program: $\lambda x^A y^B . \underbrace{(\lambda x^{A \times B} . \langle \pi_2 x, \pi_1 x \rangle)}_{\text{proof that } A \wedge B \to B \wedge A} \langle x, y \rangle$

Curry Howard Correspondence { proof $\leftrightarrow$ programs }

Simplifying a proof that $A \to B \to B \wedge A$

Roundabout proof:

1. Assume A and B, we have $A \wedge B$ { by rule $\wedge I$ }.
2. Since we've previously shown that $A \wedge B \to B \wedge A$, the result holds.

Proof corresponds to program: $\lambda x^A y^B . \underbrace{\left(\lambda x^{A \times B} . \langle \pi_2 x, \pi_1 x \rangle\right)}_{\text{proof that } A \wedge B \to B \wedge A} \langle x, y \rangle$

Normalised program: $\lambda x^A y^B . \langle y, x \rangle$

Simplifying a proof that $A \to B \to B \wedge A$

Roundabout proof:

1. Assume A and B, we have $A \wedge B$ { by rule $\wedge I$ }.
2. Since we've previously shown that $A \wedge B \to B \wedge A$, the result holds.

Proof corresponds to program: $\lambda x^A y^B . \underbrace{\big(\lambda x^{A \times B} . \langle \pi_2 x, \pi_1 x \rangle\big)}_{\text{proof that } A \wedge B \to B \wedge A} \langle x, y \rangle$

Normalised program: $\lambda x^A y^B . \langle y, x \rangle$

Normalised proof: Assume A and B, we have $B \wedge A$ { by rule $\wedge I$ }.

What now?

What now?

Proofs	Programs
Logical connector $\to$ (implication)	Function definition & application { Haskell Curry, 1934 }
Logical connector $\wedge$ (and)	Product Types { William Howard, 1969 }
Logical connector $\vee$ (or)	Sum Types/Enums { William Howard, 1969 }
Quantifiers $\forall$ (for all) and $\exists$ (exists)	Dependent Types/Types depend on values • E.g., Array type paired with its length `int[5]` { William Howard, 1969 }
Second-order intuitionistic predicate logic	Polymorphism/Generic Programming { Girard & Reynolds, 1972/1974 }
Intuitionist $\to$ Classical Logic	Continuous Passing { Tim Griffin, 1990 }

Programming Language Design
is often seen as **ad-hoc**.

Curry-Howard Correspondence
gives us a **solid theory**
of certain language features

Thank you!

Summary

1. Untyped Lambda Calculus { **programming language** }
2. Simply Typed Lambda Calculus { **programming language** }
3. Proof System ($\wedge$ and $\rightarrow$) { **proof system** }
4. Curry Howard Correspondence { **proof** $\leftrightarrow$ **programs** }
5. What now?