Simply-Typed Lambda Calculus

(Slides mostly follow Dan Grossman’s teaching materials)
Review of untyped $\lambda$-calculus

• Syntax: notation for defining functions

  (Terms) $M, N ::= x \mid \lambda x. M \mid M N$

• Semantics: reduction rules

  $$(\lambda x. M)N \rightarrow M[N/x] \quad (\beta)$$

  $\frac{M \rightarrow M'}{M N \rightarrow M' N}$  \hspace{1cm}  $\frac{N \rightarrow N'}{M N \rightarrow M N'}$  \hspace{1cm}  $\frac{M \rightarrow M'}{\lambda x. M \rightarrow \lambda x. M'}$
\[(\lambda x. M)N \rightarrow M[N/x]\] (**\(\beta\)**)

\[(\lambda f. \lambda z. f (f z)) (\lambda y. y+x) \rightarrow \lambda z. (\lambda y. y+x) ((\lambda y. y+x) z) \rightarrow \lambda z. (\lambda y. y+x) (z+x) \rightarrow \lambda z. z+x+x\]

\[
\frac{M \rightarrow M'}{M \ N \rightarrow M' \ N}
\]

\[
\frac{N \rightarrow N'}{M \ N \rightarrow M \ N'}
\]

\[
\frac{M \rightarrow M'}{\lambda x. M \rightarrow \lambda x. M'}
\]
Review of untyped $\lambda$-calculus

$$(\lambda x. \, x \, x) \, (\lambda x. \, x \, x)$$

$\rightarrow (\lambda x. \, x \, x) \, (\lambda x. \, x \, x)$$

$\rightarrow \ldots$

This class: adding a type system

(We will see that well-typed terms in STLC always terminate.)
Why types

• Type checking catches “simple” mistakes early
  • Example: 2 + true + “a”

• (Type safety) Well-typed programs will not go wrong
  • Ensure execution never reach a “meaningless” state
  • But “meaningless” depends on the semantics (each language typically defines some as type errors and others run-time errors)

• Typed programs are easier to analyze and optimize
  • Compilers can generate better code (e.g. access components of structures by known offset)

Cons: impose constraints on the programmer
  • Some valid programs might be rejected
Why **formal** type systems

- Many typed languages have informal descriptions of the type systems (e.g., in language reference manuals)
- A fair amount of careful analysis is required to avoid false claims of type safety
- A formal presentation of a type system is a precise specification of the type checker
- And allows formal proofs of type safety
What we will study about types

• Type system
  • Typing rules: assign types to terms
  • Type safety (soundness of typing rules): well-typed terms cannot go wrong

• Connection to constructive propositional logic
  • Curry-Howard isomorphism: “Propositions are Types”, “Proofs are Programs”
Adding types to $\lambda$-calculus – wrong attempt

(Types) $\tau, \sigma ::= T \mid \text{fun}$
Adding types to λ-calculus – wrong attempt

• Typing judgment (to assign types to terms)
  \[ \vdash M : \tau \]
  
  M is of type \( \tau \)

Judgment

• A statement \( J \) about certain formal properties
• Has a derivation \( \vdash J \) (i.e. “a proof”)
• Has a meaning (“judgment semantics”) \( \models J \)

• Typing rules (to derive the typing judgment)
Adding types to $\lambda$-calculus – wrong attempt

Typing rules

\[
\begin{align*}
\vdash (\lambda x. M) : \text{fun} \\
\vdash M : \text{fun} & \quad \vdash N : T \\
\hline
\vdash M \, N : T
\end{align*}
\]

Not type safe, since well-typed terms may go wrong (reduce to a “meaningless” state)

e.g. $((\lambda f. f \, 1) \, 3)$ will go “wrong”, though $\vdash (\lambda f. f \, 1) \, 3 : \text{int}$
Adding types to $\lambda$-calculus – getting it right

• **Classify functions** using argument and result types
  • ($\lambda x. x$) and ($\lambda f. f 1$) should be of different types: ($(\lambda x. x) 3$) is acceptable, but ($(\lambda f. f 1) 3$) is not
  • Explicitly specify **argument types** in function syntax

• **Type-check function bodies**, which have **free variables**
  • Types of free variables are the **context**: type of $(f 1)$ depends on the type of $f$
Simply-typed $\lambda$-calculus (STLC)

(Types) $\tau, \sigma ::= T \mid \sigma \rightarrow \tau$

An infinite number of types:
- int $\rightarrow$ int, int $\rightarrow$ (int $\rightarrow$ int), (int $\rightarrow$ int) $\rightarrow$ int, ...

$\rightarrow$ is right-associative: $\tau \rightarrow \tau \rightarrow \tau$ is $\tau \rightarrow (\tau \rightarrow \tau)$
Simply-typed $\lambda$-calculus (STLC)

(Types)  $\tau, \sigma ::= T \mid \sigma \rightarrow \tau$

(Terms)  $M, N ::= x \mid \lambda x : \tau. M \mid M \, N$
Reduction rules

\[
\frac{(\lambda x: \tau. M)N \rightarrow M[N/x]}{(\beta)}
\]

\[
\frac{M \rightarrow M'}{M N \rightarrow M' N}
\]

\[
\frac{N \rightarrow N'}{M N \rightarrow M N'}
\]

\[
\frac{M \rightarrow M'}{\lambda x: \tau. M \rightarrow \lambda x: \tau. M'}
\]

Same as untyped \(\lambda\)-calculus
Typing judgment

\( \Gamma \vdash M : \tau \)

• **Typing context** (a set of typing assumptions)

\( \Gamma ::= \cdot \mid \Gamma, x : \tau \)

  • Include types of all the **free variables** in \( M \) (each free variable \( x \) is of type \( \tau \))
  
  • Empty context \( \cdot \) is for closed terms

• Under \( \Gamma \), \( M \) is a **well-typed** term of type \( \tau \)
Typing rules

\[
\begin{align*}
\Gamma, \ x : \tau & \vdash \ x : \tau \quad \text{(var)} \\
\Gamma \vdash \ M : \sigma \rightarrow \tau & \quad \Gamma \vdash \ N : \sigma \\
\Gamma \vdash \ M \ N : \tau & \quad \text{(app)} \\
\Gamma, \ x : \sigma & \vdash \ M : \tau \\
\Gamma \vdash \ (\lambda x : \sigma. \ M) : \sigma \rightarrow \tau & \quad \text{(abs)}
\end{align*}
\]
Typing derivation examples

\[
\begin{array}{c}
\Gamma, \ x : \tau \vdash x : \tau \\
\hline
\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma \\
\hline
\Gamma \vdash MN : \tau \\
\end{array}
\] (app)

\[
\begin{array}{c}
\Gamma, x : \sigma \vdash M : \tau \\
\hline
\Gamma \vdash (\lambda x : \sigma. M) : \sigma \rightarrow \tau \\
\end{array}
\] (abs)

\[
\begin{array}{c}
\hline
x : \tau \vdash x : \tau \\
\end{array}
\] (var)

\[
\begin{array}{c}
\cdot \vdash (\lambda x : \tau. x) : \tau \rightarrow \tau \\
\end{array}
\]
Typing derivation examples

\[
\Gamma, x : \tau \vdash x : \tau \quad \text{(var)}
\]

\[
\begin{align*}
\Gamma \vdash M : \sigma \to \tau & \quad \Gamma \vdash N : \sigma \quad \text{(app)} \\
\hline
\Gamma \vdash MN : \tau 
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : \sigma & \vdash M : \tau \\
\hline
\Gamma \vdash (\lambda x : \sigma. M) : \sigma \to \tau \quad \text{(abs)}
\end{align*}
\]

\[
\begin{align*}
\hline
x : \tau, y : \sigma & \vdash x : \tau \\
\hline
\hline
\Gamma, x : \sigma & \vdash M : \tau \\
\hline
\hline
\Gamma \vdash \lambda y : \sigma. x : \sigma \to \tau \quad \text{(abs)}
\end{align*}
\]

\[
\begin{align*}
\hline
x : \tau & \vdash (\lambda y : \sigma. x) : \sigma \to \tau \\
\hline
\hline
\Gamma, x : \tau & \vdash \lambda x : \tau. \lambda y : \sigma. x : \tau \to \sigma \to \tau \quad \text{(abs)}
\end{align*}
\]

\[
\begin{align*}
\hline
\cdot & \vdash (\lambda x : \tau. \lambda y : \sigma. x) : \tau \to \sigma \to \tau
\end{align*}
\]
Typing derivation examples

\[
\begin{align*}
\Gamma, x : \tau & \vdash x : \tau \quad \text{(var)} \\
\Gamma \vdash M : \sigma \to \tau & \quad \Gamma \vdash N : \sigma \\
\hline
\Gamma \vdash M \; N : \tau \quad \text{(app)}
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : \sigma & \vdash M : \tau \\
\hline
\Gamma \vdash (\lambda x : \sigma. M) : \sigma \to \tau \quad \text{(abs)}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\Gamma \vdash M : \sigma \to \tau \\
\vdash \lambda x : \tau. x : \tau \to \tau \\
\vdash \lambda y : \tau. x \; y : \tau \\
\hline
\vdash \lambda x : \tau \to \tau. \lambda y : \tau. x \; y : (\tau \to \tau) \to \tau \to \tau \quad \text{(abs)}
\end{array}
\end{align*}
\]
Soundness and completeness

- A **sound** type system never accepts a program that can go wrong
  - No false negatives
  - The language is **type-safe**

- A **complete** type system never rejects a program that can’t go wrong
  - No false positives

- However, for any Turing-complete PL, the set of programs that may go wrong is undecidable
  - Type system cannot be sound and complete
  - Choose soundness, try to reduce false positives in practice
Soundness – well-typed terms in STLC never go wrong

Theorem (Type Safety):

If $\cdot \vdash M : \tau$ and $M \rightarrow^* M'$, then

$\cdot \vdash M' : \tau$, and either $M' \in \text{Values}$ or $\exists M''. M' \rightarrow M''$

Defined in language semantics (e.g. $\lambda$-abstraction, constants)

That is, the reduction of a well-typed term either diverges, or terminates in a value of the expected type.

Follows from two key lemmas (next page).
Soundness – well-typed terms in STLC never go wrong

• **Preservation (subject reduction):** well-typed terms reduce only to well-typed terms of the same type

  \[
  \text{If } \cdot \vdash M : \tau \text{ and } M \rightarrow M', \text{ then } \cdot \vdash M' : \tau
  \]

• **Progress:** a well-typed term is either a value or can be reduced

  \[
  \text{If } \cdot \vdash M : \tau, \text{ then either } M \in \text{Values or } \exists M'. M \rightarrow M'
  \]
Not complete – the type system may reject terms that do not go wrong

• \((\lambda x. (x (\lambda y. y)) (x\,3))\, (\lambda z. z)\)

Cannot find \(\sigma, \tau\) such that

\[
x: \sigma \vdash (x\,(\lambda y. y))(x\,3) : \tau
\]

because we have to pick one type for \(x\)

• But \((\lambda x. (x\,(\lambda y. y))\,(x\,3))\,(\lambda z. z)\)

\[
\rightarrow ((\lambda z. z)\,(\lambda y. y))\,((\lambda z. z)\,3)
\]

\[
\rightarrow (\lambda y. y)\,3 \rightarrow 3
\]
Well-typed terms in STLC always terminate  (strong normalization theorem)

• Recall \((\lambda x. \ x \ x) \ (\lambda x. \ x \ x)\)
  
  \[ \rightarrow (\lambda x. \ x \ x) \ (\lambda x. \ x \ x) \]
  
  \[ \rightarrow \ldots \]

• \((\lambda x. \ x \ x) \ (\lambda x. \ x \ x)\) cannot be assigned a type

\[
\begin{align*}
  x : \sigma & \vdash x : ? \\
  x : \sigma & \vdash x : \sigma \\
  \hline
  x : \sigma & \vdash x \ x : ?
\end{align*}
\]

Expect \(\sigma\) to be in the form of \(\sigma \rightarrow \tau\), which is impossible!
Main points of STLC

(Types) \( \tau, \sigma ::= T \mid \sigma \rightarrow \tau \)

(Terms) \( M, N ::= x \mid \lambda x : \tau. M \mid M N \)

Reduction rules

\[
(\lambda x : \tau. M) N \rightarrow M[N/x] \quad (\beta)
\]

Typing rules

\[
\frac{\Gamma, x : \tau \vdash x : \tau}{\Gamma \vdash x : \tau} \quad (\text{var})
\]

\[
\frac{\Gamma \vdash x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma. M) : \sigma \rightarrow \tau} \quad (\text{abs})
\]

\[
\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M N : \tau} \quad (\text{app})
\]

Soundness (type safety)
Adding stuff

Use STLC as a foundation for understanding other common language constructs

• Extend the syntax (types & terms)
• Extend the operational semantics (reduction rules)
• Extend the type system (typing rules)
• Extend the soundness proof (new proof cases)
Adding product type

(Types) \( \tau, \sigma ::= \ldots \mid \sigma \times \tau \)

(Terms) \( M, N ::= \ldots \mid <M, N> \mid \text{proj1 } M \mid \text{proj2 } M \)

Consider structures in C:

```c
struct date{
    int year;
    int month;
    int day;
}
```
Product type

(Types) \( \tau, \sigma ::= \ldots | \sigma \times \tau \)

(Terms) \( M, N ::= \ldots | <M, N> | \text{proj1 } M | \text{proj2 } M \)

Reduction rules

\[
\begin{align*}
\text{proj1 } <M, N> &\rightarrow M \\
M &\rightarrow M' \\
<M, N> &\rightarrow <M', N> \\
M &\rightarrow M' \\
\text{proj1 } M &\rightarrow \text{proj1 } M'
\end{align*}
\]

\[
\begin{align*}
\text{proj2 } <M, N> &\rightarrow N \\
N &\rightarrow N' \\
<M, N> &\rightarrow <M, N'> \\
M &\rightarrow M' \\
\text{proj2 } M &\rightarrow \text{proj2 } M'
\end{align*}
\]
Product type

(Types) \( \tau, \sigma ::= \ldots \mid \sigma \times \tau \)

(Terms) \( M, N ::= \ldots \mid <M, N> \mid \text{proj1 } M \mid \text{proj2 } M \)

Typing rules

\[
\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash <M, N> : \sigma \times \tau} \quad \text{(pair)}
\]

\[
\frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \text{proj1 } M : \sigma} \quad \text{(proj1)}
\]

\[
\frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \text{proj2 } M : \tau} \quad \text{(proj2)}
\]
Typing derivation example

\[
\begin{array}{rcl}
\begin{array}{c}
\Downarrow \quad \text{(var)} \\
x : \sigma \times \tau \vdash x : \sigma \times \tau
\end{array} & \quad & \begin{array}{c}
\Downarrow \quad \text{(var)} \\
x : \sigma \times \tau \vdash x : \sigma \times \tau
\end{array} \\
\begin{array}{c}
\Downarrow \quad \text{(proj2)} \\
x : \sigma \times \tau \vdash \text{proj2 } x : \tau
\end{array} & \quad & \begin{array}{c}
\Downarrow \quad \text{(proj1)} \\
x : \sigma \times \tau \vdash \text{proj1 } x : \sigma
\end{array} \\
\begin{array}{c}
\Downarrow \quad \text{(pair)} \\
x : \sigma \times \tau \vdash \langle \text{proj2 } x, \text{proj1 } x \rangle : \tau \times \sigma
\end{array}
\end{array}
\]

\[
\Downarrow \quad \text{(abs)} \\
\cdot \vdash (\lambda x : \sigma \times \tau. \langle \text{proj2 } x, \text{proj1 } x \rangle) : (\sigma \times \tau) \rightarrow (\tau \times \sigma)
\]
Soundness theorem (type safety)

• Preservation:

\[ \text{If } \cdot \vdash M : \tau \text{ and } M \rightarrow M', \text{ then } \cdot \vdash M' : \tau \]

• Progress:

\[ \text{If } \cdot \vdash M : \tau, \text{ then either } M \in \text{Values} \text{ or } \exists M'. M \rightarrow M' \]
Adding sum type

(Types) \( \tau, \sigma ::= \ldots | \sigma + \tau \)

(Terms) \( M, N ::= \ldots | \text{left } M | \text{right } M | \text{case } M \text{ do } M_1 M_2 \)

Consider unions in C:

```c
union data{
    int i;
    float f;
    char c;
}
```

Using the same location for multiple data.
Can contain only one value at any given time.
Adding sum type

(Types) \( \tau, \sigma ::= \ldots \mid \sigma + \tau \)

(Terms) \( M, N ::= \ldots \mid \text{left } M \mid \text{right } M \mid \text{case } M \text{ do } M1 \ M2 \)

Subclasses in Java:

```java
abstract class t { abstract t' m(); }
class A extends t { t1 x; t' m(){...}}
class B extends t { t2 x; t' m(){...}}
...
e.m();
```

```java
case e do A.m B.m
```
Adding sum type

(Types) $\tau, \sigma ::= ... \mid \sigma + \tau$

(Terms) $M, N ::= ... \mid \text{left } M \mid \text{right } M \mid \text{case } M \text{ do } M_1 M_2$

In Coq:

```coq
Inductive bool : Set :=
| true : bool
| false : bool.

Definition not (b : bool) : bool :=
  match b with
  | true => false
  | false => true
  end.
```
Sum type: reduction rules

\[
\begin{align*}
\text{case (left } M \text{) do } M1 & \ M2 \rightarrow M1 \ M \\
\text{case (right } M \text{) do } M1 & \ M2 \rightarrow M2 \ M
\end{align*}
\]

\[
\begin{array}{c}
M \rightarrow M' \\
\hline
\text{case } M \text{ do } M1 \ M2 \rightarrow \text{case } M' \text{ do } M1 \ M2 \\
\text{M1} \rightarrow \text{M1'} \\
\hline
\text{case } M \text{ do } M1 \ M2 \rightarrow \text{case } M \text{ do } M1' \ M2 \\
\text{M2} \rightarrow \text{M2'} \\
\hline
\text{case } M \text{ do } M1 \ M2 \rightarrow \text{case } M \text{ do } M1 \ M2'
\end{array}
\]
Sum type: typing rules

\[ \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \text{left} M : \sigma + \tau} \] (left) \quad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \text{right} M : \sigma + \tau} \] (right)

\[ \frac{\Gamma \vdash M : \sigma + \tau \quad \Gamma \vdash M1 : \sigma \to \rho \quad \Gamma \vdash M2 : \tau \to \rho}{\Gamma \vdash \text{case} M \text{ do } M1 \ M2 : \rho} \] (case)
Typing derivation examples

\[
\begin{align*}
\var{x : \tau} & \vdash x : \tau \quad \text{(var)} \\
\vdash x : \tau \vdash \text{left } x : \tau + \sigma \quad \text{(proj2)} \\
\vdash x : \tau \vdash \text{left } x : \tau + \rho \quad \text{(proj1)} \\
\vdash x : \tau \vdash \text{left } x, \text{left } x : (\tau + \sigma) \times (\tau + \rho) \quad \text{(pair)} \\
\vdash (\lambda x : \tau. \text{left } x, \text{left } x) : \tau \rightarrow (\tau + \sigma) \times (\tau + \rho) \quad \text{(abs)}
\end{align*}
\]

other side can be anything
Soundness theorem (type safety)

• Preservation:

If \( \cdot \vdash M : \tau \) and \( M \rightarrow M' \), then \( \cdot \vdash M' : \tau \)

• Progress:

If \( \cdot \vdash M : \tau \), then either \( M \in \text{Values} \) or \( \exists M'. M \rightarrow M' \)
Products vs. sums

• “logical duals” (more on this later)
  • To make a $\sigma \times \tau$, we need a $\sigma$ and a $\tau$
  • To make a $\sigma + \tau$, we need a $\sigma$ or a $\tau$
  • Given a $\sigma \times \tau$, we can get a $\sigma$ or a $\tau$ or both (our “choice”)
  • Given a $\sigma + \tau$, we must be prepared for either a $\sigma$ or a $\tau$ (the value’s “choice”)
Main points till now

• STLC extended with products and sums:

(Types) $\tau, \sigma ::= T \mid \sigma \to \tau \mid \sigma \times \tau \mid \sigma + \tau$

(Terms) $M, N ::= x \mid \lambda x : \tau. M \mid M N$

| $<M, N> \mid \text{proj1 } M \mid \text{proj2 } M$
| $\text{left } M \mid \text{right } M \mid \text{case } M \text{ do } M1 M2$

• Next: recursion
Recursion

• Recall in untyped λ-calculus, every term has a fixpoint
  • Fixpoint combinator is a higher-order function h satisfying
    for all f, \((h f)\) gives a fixpoint of f
    \[ h f = f (h f) \]
  • Turing’s fixpoint combinator Θ
    Let \( A = \lambda x. \lambda y. y (x x y) \) and \( \Theta = A A \)
  • Church’s fixpoint combinator Y
    Let \( Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \)
Recursion

• Recall “strong normalization theorem”: well-typed terms in STLC always terminate
  • Extensions so far (products & sums) preserve termination

• Recursion is not allowed by the typing rules: it is impossible to find types for fixed-point combinators

• So we add an explicit construct for recursion

(Terms) \( M, N ::= ... \ | \ fix\ M \)

(Types) \( \tau, \sigma ::= ... \) (no new types)
Reduction rules for fix

\[
\text{fix } \lambda x. M \rightarrow M[\text{fix } \lambda x. M/x] \quad M \rightarrow M' \\
\text{fix } M \rightarrow \text{fix } M'
\]

(\text{fix } \lambda f. \lambda n. \text{ if } (n == 0) \text{ then } 1 \text{ else } n * f(n-1) ) 3

\rightarrow (\lambda n. \text{ if } (n == 0) \text{ then } 1 \text{ else } n^*((\text{fix } \lambda f. \lambda n. \text{ if } (n == 0) \text{ then } 1 \text{ else } n * f(n-1))(n-1))) 3

\rightarrow \text{if } (3 == 0) \text{ then } 1 \text{ else } 3 * ((\text{fix } \lambda f. \lambda n. \text{ if } (n == 0) \text{ then } 1 \text{ else } n * f(n-1))(3-1))

\rightarrow 3 * ((\text{fix } \lambda f. \lambda n. \text{ if } (n == 0) \text{ then } 1 \text{ else } n * f(n-1))(3-1))

\rightarrow ...
Typing fix

\[
\Gamma \vdash M : \tau \to \tau \\
\Gamma \vdash \text{fix } M : \tau
\]

• Math explanation: If \( M \) is a function from \( \tau \) to \( \tau \), then \( \text{fix } M \), the fixed-point of \( M \), is some \( \tau \) with the fixed-point property.

• Operational explanation: \( \text{fix } \lambda x. M' \) reduces to \( M'[\text{fix } \lambda x. M'/x] \).
  • The substitution means \( x \) and \( \text{fix } \lambda x. M' \) need the same type.
  • The result means \( M' \) and \( \text{fix } \lambda x. M' \) need the same type.

• Soundness (type safety) is straightforward.

• But strong normalization is eliminated.
Main points till now

• STLC with products and sums:

  (Types)  \( \tau, \sigma ::= T \mid \sigma \rightarrow \tau \mid \sigma \times \tau \mid \sigma + \tau \)

  (Terms)  \( M, N ::= x \mid \lambda x : \tau. M \mid M \ N \)

           \mid \langle M, N \rangle \mid \text{proj1 } M \mid \text{proj2 } M \)

           \mid \text{left } M \mid \text{right } M \mid \text{case } M \text{ do } M1 \ M2 \)

• We can also add recursion

• Next: Curry-Howard isomorphism
Curry-Howard Isomorphism

• What we did:
  • Define a programming language
  • Define a type system to rule out “bad” programs

• What logicians do:
  • Define logic propositions
    • E.g.  \( p, q ::= B \mid p \land q \mid p \lor q \mid p \implies q \)
  • Define a proof system to prove “good” propositions

• Turn out to be related
  • Propositions are Types
  • Proofs are Programs
Curry-Howard Isomorphism

• Slogans
  • Propositions are Types
  • Proofs are Programs

In this class, we will show correspondence between formulas of constructive propositional logic

(Prop) \( p, q \ ::= \ B \mid p \Rightarrow q \mid p \land q \mid p \lor q \)

and types of STLC with products and sums

(Types) \( \tau, \sigma \ ::= T \mid \sigma \rightarrow \tau \mid \sigma \times \tau \mid \sigma + \tau \)
Examples of terms and types

\[ \lambda x : \tau. \ x \]

has type

\[ \tau \rightarrow \tau \]
Examples of terms and types

\[ \lambda x: \tau. \lambda f: \tau \to \sigma. f \, x \]

has type

\[ \tau \to (\tau \to \sigma) \to \sigma \]
Examples of terms and types

\[ \lambda f : \tau \to \sigma \to \rho. \ \lambda x : \sigma. \ \lambda y : \tau. \ f \ y \ x \]

has type

\[ (\tau \to \sigma \to \rho) \to \sigma \to \tau \to \rho \]
Examples of terms and types

\[ \lambda x: \tau. \langle \text{left } x, \text{left } x \rangle \]

has type

\[ \tau \rightarrow ((\tau + \sigma) \times (\tau + \rho)) \]
Examples of terms and types

\[ \lambda f: \tau \to \rho. \lambda g: \sigma \to \rho. \lambda x: \tau + \sigma. \text{(case } x \text{ do } f \ g) \]

has type

\[ (\tau \to \rho) \to (\sigma \to \rho) \to (\tau + \sigma) \to \rho \]
Examples of terms and types

$\lambda x: \tau \times \sigma. \lambda y: \rho. \langle \langle y, \text{proj1 } x \rangle, \text{proj2 } x \rangle$

has type

$(\tau \times \sigma) \rightarrow \rho \rightarrow ((\rho \times \tau) \times \sigma)$
Empty and nonempty types

Have seen several “nonempty” types (closed terms of that type)

\[
\begin{align*}
\tau & \to \tau \\
\tau & \to (\tau \to \sigma) \to \sigma \\
(\tau & \to \sigma \to \rho) \to \sigma \to \tau \to \rho \\
\tau & \to ((\tau + \sigma) \times (\tau + \rho)) \\
(\tau & \to \rho) \to (\sigma \to \rho) \to (\tau + \sigma) \to \rho \\
(\tau & \times \sigma) \to \rho \to ((\rho \times \tau) \times \sigma)
\end{align*}
\]

There’re also lots of “empty” types (no closed terms of that type)

\[
\begin{align*}
\tau & \\
\tau & \to \sigma \\
\tau + (\tau & \to \sigma) \\
\tau & \to (\sigma \to \tau) \\
\sigma
\end{align*}
\]

*How to know whether a type is nonempty?*
How to know whether a type is nonempty?

Let’s replace $\to$ with $\Rightarrow$, $\times$ with $\land$, $+$ with $\lor$: 

$\tau \Rightarrow \tau$
$\tau \Rightarrow (\tau \Rightarrow \sigma) \Rightarrow \sigma$
$(\tau \Rightarrow \sigma \Rightarrow \rho) \Rightarrow \sigma \Rightarrow \tau \Rightarrow \rho$
$\tau \Rightarrow ((\tau \lor \sigma) \land (\tau \lor \rho))$
$(\tau \Rightarrow \rho) \Rightarrow (\sigma \Rightarrow \rho) \Rightarrow (\tau \lor \sigma) \Rightarrow \rho$
$(\tau \land \sigma) \Rightarrow \rho \Rightarrow ((\rho \land \tau) \land \sigma)$

$\tau$
$\tau \Rightarrow \sigma$
$\tau \lor (\tau \Rightarrow \sigma)$
$\tau \Rightarrow (\sigma \Rightarrow \tau) \Rightarrow \sigma$

**Can be proved in propositional logic**
(corresponding to nonempty types – have closed terms)

**Cannot be proved in propositional logic**
(corresponding to empty types – no closed terms)
Example – propositional-logic proof

\( \Gamma \vdash p \)

\[
\begin{align*}
\tau, \sigma \vdash \tau \Rightarrow \sigma & \quad \tau, \sigma \vdash \tau \Rightarrow \tau \\
\hline
\tau, \tau \Rightarrow \sigma & \vdash \sigma \\
\hline
\tau \vdash (\tau \Rightarrow \sigma) \Rightarrow \sigma \\
\hline
\vdash \tau \Rightarrow (\tau \Rightarrow \sigma) \Rightarrow \sigma
\end{align*}
\]
Propositional logic (natural deduction)

(Prop) \[ p, q ::= B \mid p \Rightarrow q \mid p \land q \mid p \lor q \]

(Ctxt) \[ \Gamma ::= \cdot \mid \Gamma, p \]

\[ \frac{\Gamma, p \vdash p}{\Gamma \vdash p} \quad \frac{\Gamma, p \vdash q}{\Gamma \vdash p \Rightarrow q} \quad \frac{\Gamma \vdash p \Rightarrow q \quad \Gamma \vdash p}{\Gamma \vdash q} \]

\[ \frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \land q} \quad \frac{\Gamma \vdash p \land q}{\Gamma \vdash p} \quad \frac{\Gamma \vdash p \land q}{\Gamma \vdash q} \]

\[ \frac{\Gamma \vdash p}{\Gamma \vdash p \lor q} \quad \frac{\Gamma \vdash q}{\Gamma \vdash p \lor q} \]

\[ \frac{\Gamma \vdash p \lor q \quad \Gamma \vdash p \Rightarrow r \quad \Gamma \vdash q \Rightarrow r}{\Gamma \vdash r} \]
This is exactly our type system, erasing terms, replacing → with ⇒, × with ∧, + with ∨

\[
\Gamma, \ x : \tau \vdash x : \tau \quad \text{(var)}
\]

\[
\Gamma, x : \sigma \vdash M : \tau \\
\Gamma \vdash (\lambda x: \sigma. M) : \sigma \to \tau \quad \text{(abs)}
\]

\[
\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma \\
\Gamma \vdash MN : \tau \quad \text{(app)}
\]

\[
\Gamma \vdash M : \sigma \times \tau \\
\Gamma \vdash <M, N> : \sigma \times \tau \quad \text{(pair)}
\]

\[
\Gamma \vdash M : \sigma \times \tau \\
\Gamma \vdash \text{proj1} M : \sigma \quad \text{(proj1)}
\]

\[
\Gamma \vdash M : \sigma \times \tau \\
\Gamma \vdash \text{proj2} M : \tau \quad \text{(proj2)}
\]

\[
\Gamma \vdash M : \sigma \\
\Gamma \vdash \text{left} M : \sigma + \tau \quad \text{(left)}
\]

\[
\Gamma \vdash M : \tau \\
\Gamma \vdash \text{right} M : \sigma + \tau \quad \text{(right)}
\]

\[
\Gamma \vdash M : \sigma + \tau \\
\Gamma \vdash M1 : \sigma \to \rho \\
\Gamma \vdash M2 : \tau \to \rho \\
\Gamma \vdash \text{case} M \text{ do } M1 \ M2 : \rho \quad \text{(case)}
\]
Curry-Howard isomorphism

- Given a well-typed closed term, take the typing derivation, erase the terms, and have a propositional-logic proof.

- Given a propositional-logic proof, there exists a closed term with that type.

- A term that type-checks is a proof — it tells you exactly how to derive the logic formula corresponding to its type.

- Constructive (hold that thought) propositional logic and simply-typed lambda-calculus with pairs and sums are the same thing.
  - Computation and logic are deeply connected.
  - $\lambda$ is no more or less made up than implication.
Revisit our examples: “terms are proofs”

\[ \lambda x: \tau. \; x \]

is a proof that

\[ \tau \Rightarrow \tau \]
Revisit our examples: “terms are proofs”

\[ \lambda x: \tau. \lambda f: \tau \to \sigma. f \, x \]

is a proof that

\[ \tau \Rightarrow (\tau \Rightarrow \sigma) \Rightarrow \sigma \]
Revisit our examples: “terms are proofs”

\[ \lambda f: \tau \rightarrow \sigma \rightarrow \rho. \lambda x: \sigma. \lambda y: \tau. f \ y \ x \]

is a proof that

\[ (\tau \Rightarrow \sigma \Rightarrow \rho) \Rightarrow \sigma \Rightarrow \tau \Rightarrow \rho \]
Revisit our examples: “terms are proofs”

\[ \lambda x: \tau. \text{<left } x, \text{ left } x> \]

is a proof that

\[ \tau \Rightarrow ((\tau \lor \sigma) \land (\tau \lor \rho)) \]
Revisit our examples: “terms are proofs”

\[
\lambda f: \tau \to \rho. \ \lambda g: \sigma \to \rho. \ \lambda x: \tau + \sigma. \ (\text{case } x \text{ do } f \ g)
\]

is a proof that

\[
(\tau \Rightarrow \rho) \Rightarrow (\sigma \Rightarrow \rho) \Rightarrow (\tau \lor \sigma) \Rightarrow \rho
\]
Revisit our examples: “terms are proofs”

\[ \lambda x: \tau \times \sigma. \lambda y: \rho. \langle \langle y, \text{proj1 } x \rangle, \text{proj2 } x \rangle \]

is a proof that

\[ (\tau \land \sigma) \Rightarrow \rho \Rightarrow ((\rho \land \tau) \land \sigma) \]
Coq example: proof can be written as functional program

Proof of commutativity of addition on nat in Coq.

```coq
plus_comm = 
fun n m : nat => 
nat_ind (fun n0 : nat => n0 + m = m + n0) 
  (plus_n_0 m) 
  (fun (y : nat) (H : y + m = m + y) => 
    eq_ind (S (m + y)) 
      (fun n0 : nat => S (y + m) = n0) 
      (f_equal S H) 
      (m + S y) 
      (plus_n_Sm m y)) n 
  : forall n m : nat, n + m = m + n
```
Why care?

Because:

• This is just fascinating
• Don’t think of logic and computing as distinct fields
• Thinking “the other way” can help you know what’s possible/impossible
• Can form the basis for automated theorem provers
• Type systems should not be ad hoc piles of rules!

So, every typed λ-calculus is a proof system for some logic...

Is STLC with pairs and sums a complete proof system for propositional logic? Almost...
Classical vs. constructive

• Classical propositional logic has the “law of the excluded middle”:

\[ \Gamma \vdash p \lor (p \Rightarrow q) \quad \text{Think } "p \lor \neg p" \]

• STLC does not support it: e.g. no closed term has type \( \rho + (\rho \to \sigma) \)

• Logics without this rule are called “constructive” or “intuitionistic”.
  
  • Formulae are only considered "true" when we have direct evidence (“proofs produce examples”)

• STLC does not support it: e.g. no closed term has type \( \rho + (\rho \to \sigma) \)
Example classical proof

• Theorem: There exist two irrational numbers $a$ and $b$ such that $a^b$ is rational.

• Can be proved using “the law of exclusive middle”.
  • It’s known that $\sqrt{2}$ is irrational.
  • Consider the number $\sqrt{2}^{\sqrt{2}}$.
    • If it is rational, the proof is complete, and $a = b = \sqrt{2}$.
    • If it is irrational, then let $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. Then $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2} \times \sqrt{2}} = (\sqrt{2})^2 = 2$, and the proof is complete.

• Constructive logics would not accept this argument.
Classical vs. constructive

• In constructive logics, “branch on possibilities” by making “excluded middle” an explicit assumption:

\[(p \lor (p \Rightarrow q)) \land (p \Rightarrow r) \land ((p \Rightarrow q) \Rightarrow r) \Rightarrow r\]

• “if any number is either rational or irrational, then there exist two irrational numbers \(a\) and \(b\) such that \(a^b\) is rational”
What about “fix”? 

• A “non-terminating proof” is no proof at all 
• Remember the typing rule 

\[
\Gamma \vdash M : \tau \rightarrow \tau \\
\therefore \Gamma \vdash \text{fix}M : \tau 
\] 

• It lets us prove anything! E.g. \(\text{fix} \lambda x:\tau. x\) has type \(\tau\) 
• So the “logic” is inconsistent
Last word on Curry-Howard

- Not just constructive propositional logic & STLC
- *Every* logic has a corresponding typed system
  - Classical logics
  - Inconsistent logics

- If you remember one thing:

\[
\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma \\
\hline
\Gamma \vdash M \, N : \tau \quad \text{(app)}
\]

\[
\Gamma \vdash p \Rightarrow q \quad \Gamma \vdash p \\
\hline
\Gamma \vdash q \quad \text{(\Rightarrow \text{-elim})}
\]