Administrivia

Homework 5 is due on Thursday by 11:59PM.

There will be no homework assigned over the break, but there will be a written project "proposal" due after the break.

The Simply Typed Lambda Calculus: The Curry-Howard Isomorphism

Type Theory and Mechanized Reasoning

Lecture 12

`CAS CS 400`

Objectives

Finish our discussion of strong normalization.

Look at data types in STLC.

Get a peek into the Curry–Howard Isomorphism.

Strong Normalization of STLC

Recall: The Simply Typed Lambda Calculus

$$f : \bot \to \bot \vdash \lambda x\,.\,fx : \bot \to \bot$$

Recall: The Simply Typed Lambda Calculus

$$f : \bot \to \bot \vdash \lambda x \,.\, f x : \bot \to \bot$$

The Simply Typed Lambda Calculus (STLC) is a type theory built on top of the (untyped) lambda calculus (ULC).

Recall: The Simply Typed Lambda Calculus

$$f : \bot \to \bot \vdash \lambda x \,.\, f x : \bot \to \bot$$

The Simply Typed Lambda Calculus (STLC) is a type theory built on top of the (untyped) lambda calculus (ULC).

It was created by Alonzo Church in the 1930s (after his system with using the lambda calculus was shown to be inconsistent).

Recall: The Simply Typed Lambda Calculus

$$f : \bot \to \bot \vdash \underbrace{\lambda x \, . fx}_{\lambda \text{ term}} : \bot \to \bot$$

The Simply Typed Lambda Calculus (STLC) is a type theory built on top of the (untyped) lambda calculus (ULC).

It was created by Alonzo Church in the 1930s (after his system with using the lambda calculus was shown to be inconsistent).

Recall: The Simply Typed Lambda Calculus

$$f : \bot \to \bot \vdash \underbrace{\boxed{\lambda x \,.\, f x}}_{\lambda\ \texttt{term}} : \underbrace{\boxed{\bot \to \bot}}_{\texttt{simple type}}$$

The Simply Typed Lambda Calculus (STLC) is a type theory built on top of the (untyped) lambda calculus (ULC).

It was created by Alonzo Church in the 1930s (after his system with using the lambda calculus was shown to be inconsistent).

Recall: The Simply Typed Lambda Calculus

$$f : \bot \to \bot \vdash \underbrace{\underbrace{\lambda x \,.\, fx}_{\lambda\ \texttt{term}} : \underbrace{\bot \to \bot}_{\texttt{simple type}}}_{\texttt{typing statement}}$$

The <u>Simply Typed Lambda Calculus (STLC)</u> is a type theory built on top of the (untyped) lambda calculus (ULC).

It was created by <u>Alonzo Church</u> in the 1930s (after his system with using the lambda calculus was shown to be inconsistent).

Recall: The Simply Typed Lambda Calculus

$$\underbrace{f : \bot \to \bot}_{\text{context}} \vdash \underbrace{\underbrace{\lambda x . fx}_{\lambda \text{ term}} : \underbrace{\bot \to \bot}_{\text{simple type}}}_{\text{typing statement}}$$

The Simply Typed Lambda Calculus (STLC) is a type theory built on top of the (untyped) lambda calculus (ULC).

It was created by Alonzo Church in the 1930s (after his system with using the lambda calculus was shown to be inconsistent).

Recall: The Simply Typed Lambda Calculus

$$\underbrace{\underbrace{f : \bot \to \bot}_{\text{context}} \vdash \underbrace{\underbrace{\lambda x . fx}_{\lambda\ \text{term}} : \underbrace{\bot \to \bot}_{\text{simple type}}}_{\text{typing statement}}}_{\text{typing judgment}}$$

The Simply Typed Lambda Calculus (STLC) is a type theory built on top of the (untyped) lambda calculus (ULC).

It was created by Alonzo Church in the 1930s (after his system with using the lambda calculus was shown to be inconsistent).

Simply Typed Lambda Calculus (Types)

$$\frac{}{\varnothing \vdash \bot : \mathsf{Type}} \qquad \frac{\varnothing \vdash A : \mathsf{Type} \qquad \varnothing \vdash B : \mathsf{Type}}{\varnothing \vdash A \to B : \mathsf{Type}}$$

<u>Type formation rules</u> are used to build types within and for judgments.

(These are the same as our inductive rules, but written as typing judgments)

Simply Typed Lambda Calculus (Terms)

$$\frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma, x : A \vdash x : A}\ (x \notin \Gamma) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \,.\, M : A \to B}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma, x : B \vdash M : A}\ (x \notin \Gamma) \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

`Term formation rules` are used to generate typeable terms in the simply typed lambda calculus.

Simply Typed Lambda Calculus (Terms)

start

$$\frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma, x : A \vdash x : A} \ (x \notin \Gamma)$$

abstraction

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \,.\, M : A \to B}$$

weakening

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma, x : B \vdash M : A} \ (x \notin \Gamma)$$

application

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

<u>Term formation rules</u> are used to generate typeable terms in the simply typed lambda calculus.

Normalization Theorem

Normalization Theorem

Theorem. Every lambda term which is typeable in STLC is strongly normalizing.

Normalization Theorem

Theorem. Every lambda term which is typeable in STLC is strongly normalizing.

» All programs we can write in STLC terminate.

Normalization Theorem

Theorem. Every lambda term which is typeable in STLC is strongly normalizing.

» All programs we can write in STLC terminate.

» In STLC, we can use *any* reduction strategy.

Induction on Derivations

$$\frac{\vdots}{\Gamma \vdash M : A}$$

If we want to prove that P holds of all *typeable* terms, we have to show that it holds of all terms M ***for any choice of the last inference rule in a derivation of M.***

The Application Case

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

We need to show that if P holds of M and it holds of N then it also holds of MN.

The Application Case (Stronger)

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

We need to show that if $P_{A \to B}$ holds of M and P_A holds of N then P_B also holds of MN.

Example: Free Variable Theorem

Example: Free Variable Theorem

Fact. If $\Gamma \vdash M : A$ then all free variables of M appear in Γ.

Example: Free Variable Theorem

Fact. If $\Gamma \vdash M : A$ then all free variables of M appear in Γ.

We can't derive a term in a context Γ which uses a variable not appearing in Γ, e.g.,

$$x : A, y : B \nvdash \lambda x . w : C \to D$$

Example: Free Variable Theorem

Fact. If $\Gamma \vdash M : A$ then all free variables of M appear in Γ.

We can't derive a term in a context Γ which uses a variable not appearing in Γ, e.g.,

$$x : A, y : B \nvdash \lambda x \,.\, w : C \to D$$

Let's prove this...

Normalization Theorem: Attempt One

Theorem. Every typeable term in STLC is SN.

Let's try to prove this...

The Problem Case

$$\frac{\Gamma \vdash \lambda x\,.\,M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : M)N : B}$$

The Problem Case

$$\frac{\Gamma \vdash \lambda x\,.\,M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : M)N : B}$$

How do we know $M[N/x]$ is strongly normalizing?

The Problem Case

$$\frac{\Gamma \vdash \lambda x\,.\,M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : M)N : B}$$

How do we know $M[N/x]$ is strongly normalizing?

The issue. Substitution can create redexes.

The Problem Case

$$\frac{\Gamma \vdash \lambda x \,.\, M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : M)N : B}$$

How do we know $M[N/x]$ is strongly normalizing?

The issue. Substitution can create redexes.

$$(\ldots (xQ) \ldots)[(\lambda y \,.\, N)/x] = (\ldots . ((\lambda y \,.\, N)Q) \ldots)$$

Understanding Check

Find an expression of the form $(\lambda x.M)N$ *such that* $M[N/x]$ *has more redexes than* $(\lambda x.M)N$*.*

The Trick

The Trick

We will prove something different *depending on the type* of the typeable term.

The Trick

We will prove something different *depending on the type* of the typeable term.

$$P_{\Gamma,\bot}(M) = M \text{ is SN}$$

$$P_{\Gamma,A\to B}(M) = N \text{ is SN implies } MN \text{ is SN}$$

$$\text{for any } N \text{ s.t. } \Gamma \vdash N : A$$

The Trick

We will prove something different *depending on the type* of the typeable term.

$$P_{\Gamma,\perp}(M) = M \text{ is SN}$$

$$P_{\Gamma,A\to B}(M) = N \text{ is SN implies } MN \text{ is SN}$$

$$\text{for any } N \text{ s.t. } \Gamma \vdash N : A$$

Lemma. $P_{\Gamma,A\to B}(M)$ implies M is SN.

Attempt Two

Theorem. For any context Γ, term M and type A, if $\Gamma \vdash M : A$ then $P_A(M)$ holds.

Let's try to prove this...

The Problem Case

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \,.\, M : A \to B}$$

How do we know that $(\lambda x \,.\, M)N$ is SN if $\Gamma \vdash N : A$ and N is SN?

We run into a similar issue as before...

Simultaneous Substitution

Theorem. If $y_1 : A_1 \ \ldots \ y_k : A_k \vdash M : B$ an

$$\Gamma \vdash N_1 : A_1 \text{ and } \ldots \text{ and } \Gamma \vdash N_k : A_k$$

then $\Gamma \vdash M[N_1/y_1][N_2/y_2]\ldots[N_k/y_k] : B$.

We will often write $M[\overrightarrow{N}/\vec{y}]$ when we want to substitute multiple values at once.

The Final Trick

Prove something stronger for a stronger inductive hypothesis.

$P_\Gamma(M) =$

- $\Gamma = y_1 : A_1 \ \ldots \ y_k : A_k$
- $\Gamma \vdash M : B$ for some B
- $\Gamma \vdash N_1 : A_1$ and $\ldots$ and $\Gamma \vdash N_k : A_k$
- each $N_1, \ldots, N_k$ are SN

then so is $M[\overrightarrow{N}/\vec{y}]$.

Normalization Theorem

$$P_{\Gamma,\perp}(M) = P_{\Gamma}(M)$$

$$P_{\Gamma,A\to B}(M) = P_{\Gamma}(MN) \text{ for any } N \text{ s.t.}$$

$$N \text{ is SN and } \Gamma \vdash N : A$$

Theorem. For any context Γ, term M, and type A, if $\Gamma \vdash M : A$ then $P_{\Gamma,A}(M)$.

Let's try one more time...

Data Types

Encoding Data

In the lambda calculus, we encoded data as lambda terms:

$$\mathsf{pair} = \lambda x\,.\,\lambda y\,.\,\lambda f\,.\,fxy$$

$$\mathsf{fst} = \lambda p\,.\,p(\lambda x\,.\,\lambda y\,.\,x)$$

$$\mathsf{snd} = \lambda p\,.\,p(\lambda x\,.\,\lambda y\,.\,y)$$

Exercise. *Reduce* $\mathsf{fst}\ (\mathsf{pair}\ I\ K)$ *to normal form.*

Types and Encoding

$$\vdash \lambda x\,.\,\lambda y\,.\,\lambda f\,.\,fxy : A \to B \to (A \to B \to C) \to C$$

Types and Encoding

$$\vdash \lambda x \,.\, \lambda y \,.\, \lambda f \,.\, fxy : A \to B \to (A \to B \to C) \to C$$

`We can derive` pair `in STLC, but there can be no notion of the` *type* $A \times B$`, only`

Types and Encoding

$$\vdash \lambda x \,.\, \lambda y \,.\, \lambda f \,.\, fxy : A \to B \to (A \to B \to C) \to C$$

`We can derive` pair `in STLC, but there can be no notion of the` *`type`* $A \times B$`, only`

$$A \times_C B = (A \to B \to C) \to C$$

Types and Encoding

$$\vdash \lambda x . \lambda y . \lambda f . fxy : A \to B \to (A \to B \to C) \to C$$

We can derive pair in STLC, but there can be no notion of the *type* $A \times B$, only

$$A \times_C B = (A \to B \to C) \to C$$

The type depends on the output of the "recursor".

Adding Data Types

Instead, it is more natural to add data types by augmenting the type system.

This means adding three things:

» type formation rules
» term formation rules
» computation rules (extensions of $\to_\beta$)

Unit

$$\frac{}{\varnothing \vdash \top : \mathrm{Type}}\ \textbf{type formation}$$

$$\frac{}{\vdash \mathrm{unit} : \top}\ \textbf{term formation}$$

unit has no computation rules

The unit type is a type with a single element.

It is a convenience, we could "encode" it with the type $\bot \to \bot$.

Product

$$\frac{\varnothing \vdash A : \mathsf{Type} \quad \varnothing \vdash B : \mathsf{Type}}{\varnothing \vdash A \times B : \mathsf{Type}}$$ **type formation**

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}\ M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}\ M : B}$$ **term formation**

$$\mathsf{fst}\langle M, N \rangle \to_\beta M$$
$$\mathsf{snd}\langle M, N \rangle \to_\beta N$$ **computation rules**

Example

$$\vdash \lambda p\,.\,\langle \mathsf{snd}\ p, \mathsf{fst}\ p \rangle : (A \times B) \to (B \times A)$$

`Let's derive this...`

Union

$$\frac{\varnothing \vdash A : \mathsf{Type} \quad \varnothing \vdash B : \mathsf{Type}}{\varnothing \vdash A + B : \mathsf{Type}}$$ **type formation**

$$\frac{\Gamma \vdash M : A + B \quad \Gamma \vdash f : A \to C \quad \Gamma \vdash g : B \to C}{\Gamma \vdash \mathsf{case}\ M\ f\ g : C}$$ **term formation**

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{inl} M : A + B} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathsf{inr} M : A + B}$$

$$\mathsf{case}\ (\mathsf{inl} M)\ f\ g \to_\beta f\ M$$
$$\mathsf{case}\ (\mathsf{inr}\ M)\ f\ g \to_\beta g\ M$$
computation rules

Example

$$x : A, y : B \vdash \lambda z\,.\,\mathsf{case}\ z\ (\lambda f\,.\,fx)\ (\lambda g\,.\,gy) : ((A \to C) + (B \to C)) \to C$$

Let's derive this...

Encoding with Unions and Products

Once we have unions and products, we can start encoding more data types, e.g.,

$$
\begin{aligned}
\mathsf{Bool} &\equiv \top + \top \\
\mathsf{true} &\equiv \mathsf{inl}\ \mathsf{unit} \\
\mathsf{false} &\equiv \mathsf{inr}\ \mathsf{unit} \\
\mathsf{ife} &\equiv \lambda b \,.\, \lambda x \,.\, \lambda y \,.\, \mathsf{case}\ b\ (\lambda b \,.\, x)\ (\lambda b \,.\, y)
\end{aligned}
$$

But we can't define *inductive* structures (yet).

Natural Numbers

$$\frac{}{\varnothing \vdash \mathsf{Nat} : \mathsf{Type}} \; \texttt{type formation}$$

$$\frac{}{\varnothing \vdash \mathsf{zero} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}\ M : \mathsf{Nat}} \; \texttt{term formation}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat} \quad \Gamma \vdash N : A \quad \Gamma \vdash f : \mathsf{Nat} \to A}{\Gamma \vdash \mathsf{recNat}\ M\ N\ f}$$

$$\begin{array}{l} \mathsf{recNat}\ \mathsf{zero}\ N\ f \to_\beta N \\ \mathsf{recNat}\ (\mathsf{suc}\ M)\ N\ f \to_\beta f\ (\mathsf{recNat}\ M\ N\ f) \end{array} \; \texttt{computation rules}$$

Example

$$m : \mathsf{Nat}, n : \mathsf{Nat} \vdash \mathsf{recNat}\ m\ n\ (\lambda x\,.\,\mathsf{suc}\ x)$$

Let's derive this...

The Curry-Howard Isomorphism

Warm Up

$$\vdash \lambda f\,.\,\lambda g\,.\,\lambda x\,.\,g(fx) : (A \to B) \to (B \to C) \to (A \to C)$$

`Let's derive this...`

Examples

$$\vdash \lambda f\,.\,\lambda g\,.\,\lambda x\,.\,fx : (A \to B) \to (B \to C) \to (A \to C)$$

$$x : A, y : B \vdash \lambda z\,.\,\mathsf{case}\ z\ (\lambda f\,.\,fx)\ (\lambda g\,.\,gy) : ((A \to C) + (B \to C)) \to C$$

$$\vdash \lambda p\,.\,\langle \mathsf{snd}\ p, \mathsf{fst}\ p\rangle : (A \times B) \to (B \times A)$$

Examples

$$\vdash (A \to B) \to (B \to C) \to (A \to C)$$

$$A, B \vdash ((A \to C) + (B \to C)) \to C$$

$$\vdash (A \times B) \to (B \times A)$$

`If we ignore all the lambda terms and variables...`

Examples

$$\vdash (A \to B) \to (B \to C) \to (A \to C)$$

$$A, B \vdash ((A \to C) \vee (B \to C)) \to C$$

$$\vdash (A \wedge B) \to (B \wedge A)$$

`And read × as ∧ and + as ∨...`

Examples

$$\vdash (A \to B) \to (B \to C) \to (A \to C)$$

$$A, B \vdash ((A \to C) \vee (B \to C)) \to C$$

$$\vdash (A \wedge B) \to (B \wedge A)$$

And read $\times$ as $\wedge$ and $+$ as $\vee$...

We've defined a proof system for propositional logic.

Modus Ponens

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

If we block out the lambda terms and read → as "implies", then the application rule is just modus ponens.

Modus Ponens

$$\frac{\Gamma \vdash \text{[redacted]}\ A \to B \qquad \Gamma \vdash \text{[redacted]}\ A}{\Gamma \vdash \text{[redacted]}\ B}$$

If we block out the lambda terms and read $\to$ as "implies", then the application rule is just <u>modus ponens</u>.

Deduction Theorem

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \,.\, M : A \to B}$$

`If we block out the lambda terms and variables,`
`this is a` <u>`deduction theorem`</u> `for proofs.`

Deduction Theorem

$$\frac{\Gamma, \text{[redacted]}\ A \vdash \text{[redacted]}\ B}{\Gamma \vdash \text{[redacted]}\ A \to B}$$

If we block out the lambda terms and variables, this is a deduction theorem for proofs.

What about Negation?

$$\neg P \equiv P \rightarrow (x \land \neg x)$$

What about Negation?

$$\neg P \equiv P \rightarrow (x \land \neg x)$$

In classical propositional logic, we could prove the following logical equivalence.

What about Negation?

$$\neg P \equiv P \to (x \land \neg x)$$

In classical propositional logic, we could prove the following logical equivalence.

In this proof system we're defining, we'll take $\neg P$ to be $P \to \bot$.

What about Negation?

$$\neg P \equiv P \to (x \land \neg x)$$

In classical propositional logic, we could prove the following logical equivalence.

In this proof system we're defining, we'll take $\neg P$ to be $P \to \bot$.

Remember. Like in Agda, $P \to \bot$ may be read as "P is empty".

DeMorgan's Law

$$x : A, y : B \vdash \lambda z\,.\,\mathsf{case}\ z\ (\lambda f\,.\,fx)\ (\lambda g\,.\,gy) : ((A \to \bot) + (B \to \bot)) \to \bot$$

We can read this as:

$$A, B \vdash (\neg A \vee \neg B)$$

Which is one of De Morgan's laws.

Empty

$$\frac{\Gamma \vdash M : \bot \quad \Gamma \vdash A : \mathsf{Type}}{\Gamma \vdash \mathsf{explode}\ M : A}$$

We will add one additional rule for ⊥ representing the principle of explosion.

If we can derive a term of type ⊥, then we can derive a term of any type.

The Curry-Howard Isomorphism

The Curry-Howard Isomorphism

The Curry–Howard Isomorphism is a recognition of the following paradigms of type theory:

The Curry-Howard Isomorphism

The Curry–Howard Isomorphism is a recognition of the following paradigms of type theory:

» Propositions as types
» Proofs as terms
» Proof simplification as program evaluation

The Curry-Howard Isomorphism

The Curry–Howard Isomorphism is a recognition of the following paradigms of type theory:

» Propositions as types
» Proofs as terms
» Proof simplification as program evaluation

What exactly the isomorphism *is* depends on your philosophical inclinations:

The Curry-Howard Isomorphism

The Curry–Howard Isomorphism is a recognition of the following paradigms of type theory:

- Propositions as types
- Proofs as terms
- Proof simplification as program evaluation

What exactly the isomorphism *is* depends on your philosophical inclinations:

- It is a literal isomorphism
- It is an analogy
- It is a sleight of hand

The Curry-Howard Isomorphism

The Curry–Howard Isomorphism is a recognition of the following paradigms of type theory:

- Propositions as types
- Proofs as terms
- Proof simplification as program evaluation

What exactly the isomorphism *is* depends on your philosophical inclinations:

- It is a literal isomorphism
- It is an analogy
- It is a sleight of hand

I prefer: type theory is logic.