Administrivia

Homework 7 is due by Thursday at 11:59PM.

The BUGWU is on strike.

Case Study: STLC in Agda

Type Theory and Mechanized Reasoning
Lecture 16

CAS CS 400

Outline

See how to represent the simply typed lambda calculus ***in Agda***.

Prove meta-theoretic lemmas about STLC, leading to a proof of ***type preservation.***

Recap

Recall: Lambda Terms

```
(Fix a set of variables.)
```

Recall: Lambda Terms

(Fix a set of variables.)

Definition. The collection of **lambda terms** is defined inductively.

Recall: Lambda Terms

(Fix a set of variables.)

Definition. The collection of **lambda terms** is defined inductively.

- Every variable x is a lambda term.

variables

Recall: Lambda Terms

(Fix a set of variables.)

Definition. The collection of **lambda terms** is defined inductively.

- Every variable x is a lambda term.
- If M and N are lambda terms, then so is (MN)

variables

application

Recall: Lambda Terms

(Fix a set of variables.)

Definition. The collection of **lambda terms** is defined inductively.

- Every variable x is a lambda term. variables
- If M and N are lambda terms, then so is (MN) application
- If M is a lambda term, then so is $(\lambda x . M)$ for any variable x abstraction

Recall: Examples

$$x, y$$

$$I \triangleq \lambda x \,.\, x$$

$$K \triangleq \lambda x \,.\, \lambda y \,.\, x$$

$$A \triangleq \lambda x \,.\, \lambda y \,.\, xy$$

$$\omega \triangleq \lambda x \,.\, xx$$

$$\Omega \triangleq \omega\omega = (\lambda x \,.\, xx)(\lambda x \,.\, xx)$$

Recall: Motivating De Bruijn Indices

$$\lambda x \,.\, xz =_\alpha \lambda y \,.\, yz =_\alpha \lambda \boxed{\bullet} \,.\, \boxed{\bullet}\, z$$

Recall: Motivating De Bruijn Indices

$$\lambda x\,.\,xz =_\alpha \lambda y\,.\,yz =_\alpha \lambda \boxed{\bullet}\,.\,\boxed{\bullet}\,z$$

`We always consider terms up to` $=_\alpha$.

Recall: Motivating De Bruijn Indices

$$\lambda x\,.\,xz =_\alpha \lambda y\,.\,yz =_\alpha \lambda \boxed{\bullet}\,.\,\boxed{\bullet}\,z$$

We always consider terms up to $=_\alpha$.

What we *really* want is to be able to replace the binding variable with a pointer.

Recall: Motivating De Bruijn Indices

$$\lambda x\,.\,xz =_\alpha \lambda y\,.\,yz =_\alpha \lambda\,\boxed{\bullet}\,.\,\boxed{\bullet}\,z$$

We always consider terms up to $=_\alpha$.

What we *really* want is to be able to replace the binding variable with a pointer.

In math speak, we want to give a canonical element for the α–equivalence class.

Recall: De Bruijn Indices

$$\lambda\ (\lambda\ 1\ (\lambda\ 1))\ (\lambda\ 2\ 1)$$

The idea. Bound variables are represented as numbers, the depth away from the *binding site*.

$$M ::= \mathbb{N} \mid \lambda M \mid MM$$

This gives an incredibly simple grammar.

https://commons.wikimedia.org/wiki/File:De_Bruijn_index_illustration_1.svg

Recall: Free Variables and De Bruijn Indices

$$\lambda x\,.\,x(yz) \longrightarrow \lambda.1(23)$$

Today, we will be using numbers ***larger than the depth*** of the term to represent free variables.

(This will make contexts easier to represent.)

(There is also a very nice trick for representing De Bruijn indices using dependent types.)

demo

(let's define these in Agda)

Recall: The Simply Typed Lambda Calculus

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

Recall: The Simply Typed Lambda Calculus

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

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

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

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

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

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

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

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

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

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

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

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).

Recall: Simple Types

Recall: Simple Types

Definition. The collection of **simple types** is defined inductively as follows.

Recall: Simple Types

Definition. The collection of **simple types** is defined inductively as follows.

- $\perp$ is a simple type.

Recall: Simple Types

Definition. The collection of **simple types** is defined inductively as follows.

- $\perp$ is a simple type.
- If A and B are simple types, then is $A \to B$.

Recall: Simple Types

Definition. The collection of **simple types** is defined inductively as follows.

- $\perp$ is a simple type.
- If A and B are simple types, then is $A \to B$.

Examples. $\perp \to \perp$, $(\perp \to \perp) \to (\perp \to (\perp \to \perp))$

Simply Typed Lambda Calculus (Types)

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

`Type formation rules` 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)

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

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

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

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

`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.

demo

(let's define these in Agda)

Variable Shifting

$$\lambda x \,.\, x(\lambda y \,.\, y(x(zw))) \longrightarrow \lambda.0(\lambda.0(1(23))) \longrightarrow \lambda.0(\lambda.0(1(56)))$$

One of the trickier aspects of working with De Bruijn indices is that we often have to ***shift around the values of free variables***.

We will write $\text{shift}_{m,p}(M)$ for the function which increases all free variables at least value m by p.

Example: Weakening

$$x : A \vdash \lambda y\,.\,x : C \to A \xrightarrow{\texttt{weakening}} x : A, z : B \vdash \lambda y\,.\,x : C \to A$$

$$A \vdash \lambda.1 : C \to A \qquad A, B \vdash \lambda.2 : C \to A$$

```
When we represent the variables in a context,
they are in increasing order from right to left.
```

```
So weakening requires changing the typed term.
```

demo

(let's define these in Agda)

Recall: Induction on Derivations

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

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**.**

Thinning Lemma

Theorem. If $\Gamma, \Delta \vdash M : A$ and x does not appear in Δ, then $\Gamma, x : B, \Delta \vdash M : A$.

Using De Bruijn indices:

If $\Gamma, \Delta \vdash M : A$ and $|\Gamma| = m$, then $\Gamma, B, \Delta \vdash \mathsf{shift}_{|\Gamma|,1}(M) : A$

Proof. By induction on the structure of derivations.

demo

(let's define these in Agda)

Simultaneous Substitution

Let M be a term with free variables $\vec{x} = x_1, \ldots, x_k$.
We define $M[\overrightarrow{N}/\vec{x}]$ inductively as follows.

» $x_i[N_1/x_1]\ldots[N_k/x_k] = N_i$ lookup

» $(MP)[\overrightarrow{N}/\vec{x}] = (M[\overrightarrow{N}/\vec{x}])(P[\overrightarrow{N}/\vec{x}])$ recurse

» $(\lambda M)[\overrightarrow{N}/\vec{x}] = \lambda(M[\overrightarrow{N}'/\vec{x}])$ where $N_i' = \mathsf{shift}_{0,1}(N_i)$ recurse and shift

Recall: Simultaneous Substitution

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

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

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

Proof. By induction on the structure of derivations.

demo

(let's define these in Agda)

Recall: Beta Reduction

Recall: Beta Reduction

Definition. We define the relation $M \to_\beta N$ as follows.

Recall: Beta Reduction

Definition. We define the relation $M \to_\beta N$ as follows.

- $(\lambda x \,.\, M)N \to_\beta M[N/x]$

Recall: Beta Reduction

Definition. We define the relation $M \to_\beta N$ as follows.

- $(\lambda x \,.\, M)N \to_\beta M[N/x]$
- $M \to_\beta M'$ implies $MN \to_\beta M'N$ and $NM \to_\beta NM'$ and $\lambda x \,.\, M \to_\beta \lambda x \,.\, M'$

Recall: Beta Reduction

Definition. We define the relation $M \to_\beta N$ as follows.

- $(\lambda x\,.\,M)N \to_\beta M[N/x]$
- $M \to_\beta M'$ implies $MN \to_\beta M'N$ and $NM \to_\beta NM'$ and $\lambda x\,.\,M \to_\beta \lambda x\,.\,M'$

This is a **relation** not a function.

Type Preservation

Theorem. If $\Gamma \vdash M : A$ and $M \to_\beta N$ then $\Gamma \vdash N : A$.

Beta reduction doesn't change typability, or the type.

Proof. By induction on the β–reduction relation...(!)

demo

(let's define these in Agda)