The Simply Typed Lambda Calculus: An Introduction

Type Theory and Mechanized Reasoning

Lecture 12

CAS CS 400

Introduction

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.

Objectives

Discuss De Bruijn indices.

Introduce the simply typed lambda calculus (STLC).

Show that STLC is strongly normalizing (SN).

"Agda" Tutorial: De Bruijn Indices

Motivation

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

Motivation

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

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

Motivation

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

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

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

Motivation

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

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.

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

What about free variables?

What about free variables?

We can use numbers larger than the depth of the formula.

$$\lambda x\,.\,xz \Longrightarrow \lambda(1\ 2)$$

What about free variables?

We can use numbers larger than the depth of the formula.

$$\lambda x\,.\,xz \Longrightarrow \lambda(1\ 2)$$

Or we can use the "locally nameless representation":

$$\lambda x\,.\,xz \Longrightarrow \lambda(1\ z)$$

What about free variables?

We can use numbers larger than the depth of the formula.

$$\lambda x\,.\,xz \Longrightarrow \lambda(1\ 2)$$

Or we can use the "locally nameless representation":

$$\lambda x\,.\,xz \Longrightarrow \lambda(1\ z)$$

We keep free variables as they are, and use De Bruijn indices for bound variables.

Let's try it in Agda.

Pros and Cons

- We no longer need to consider $=_\alpha$, equality is structural equality.
- β–reduction is more difficult, De–Bruijn indices need to change.
- De Bruijn indexed terms are harder to read.

Simply Typed Lambda Calculus: Motivation

Recall: Normalization

Recall: Normalization

Definition. A term is **weakly normalizing** if it has a normal form.

Recall: Normalization

Definition. A term is **weakly normalizing** if it has a normal form.

Definition. A term is **strongly normalizing** if it has no infinite reduction sequences.

Recall: Normalization

Definition. A term is **weakly normalizing** if it has a normal form.

Definition. A term is **strongly normalizing** if it has no infinite reduction sequences.

Strong normalization means we don't need to think about the reduction strategy.

Recall: The Omega Combinator

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

Recall: The Omega Combinator

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

This is the infinite-loop combinator:

Recall: The Omega Combinator

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

`This is the infinite-loop combinator:`

$$(\lambda x \,.\, xx)(\lambda x \,.\, xx) \to_\beta (xx)[\lambda x \,.\, xx/x] = (\lambda x \,.\, xx)(\lambda x \,.\, xx)$$

Recall: The Omega Combinator

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

This is the infinite-loop combinator:

$$(\lambda x\,.\,xx)(\lambda x\,.\,xx) \to_\beta (xx)[\lambda x\,.\,xx/x] = (\lambda x\,.\,xx)(\lambda x\,.\,xx)$$

Not all lambda terms are strongly normalizing.

Is there a "natural" subset of strongly normalizing lambda terms?

How do we delineate such a subset?

Type Theory (At a High Level)

```
# let f (x : int) : int = x;;
val f : int -> int = <fun>
# f "two";;
Line 1, characters 2-7:
1 | f "two";;
      ^^^^^
Error: This expression has type string but an expression was expected of type
         int
# let g (x : string) : string = x;;
val g : string -> string = <fun>
# g (f 2);;
Line 1, characters 2-7:
1 | g (f 2);;
      ^^^^^
Error: This expression has type int but an expression was expected of type
         string
```

Type Theory (At a High Level)

```
# let f (x : int) : int = x;;
val f : int -> int = <fun>
# f "two";;
Line 1, characters 2-7:
1 | f "two";;
      ^^^^^
Error: This expression has type string but an expression was expected of type
         int
# let g (x : string) : string = x;;
val g : string -> string = <fun>
# g (f 2);;
Line 1, characters 2-7:
1 | g (f 2);;
      ^^^^^
Error: This expression has type int but an expression was expected of type
         string
```

Types are used to describe the behavior and compositionality of a program.

Type Theory (At a High Level)

```
# let f (x : int) : int = x;;
val f : int -> int = <fun>
# f "two";;
Line 1, characters 2-7:
1 | f "two";;
      ^^^^^
Error: This expression has type string but an expression was expected of type
         int
# let g (x : string) : string = x;;
val g : string -> string = <fun>
# g (f 2);;
Line 1, characters 2-7:
1 | g (f 2);;
      ^^^^^
Error: This expression has type int but an expression was expected of type
         string
```

Types are used to describe the behavior and compositionality of a program.

We can't apply an **int -> int** to a **string**, or presume that an **int -> int** applied to an **int** is a **string**.

Types and the ω Combinator

```
# let omega x = x x;;
Line 1, characters 16-17:
1 | let omega x = x x;;
                    ^
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
```

Types and the ω Combinator

```
# let omega x = x x;;
Line 1, characters 16-17:
1 | let omega x = x x;;
                    ^
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
```

We can't even write the combinator ω in a typed language (like OCaml).

Types and the ω Combinator

```
# let omega x = x x;;
Line 1, characters 16-17:
1 | let omega x = x x;;
                    ^
Error: This expression has type 'a -> 'b
       but an expression was expected of type 'a
       The type variable 'a occurs inside 'a -> 'b
```

We can't even write the combinator ω in a typed language (like OCaml).

x is expected to by a function and the argument of a function.

What is a type?

```
k : {A B : Set} -> A -> B -> A
k x y = x
```

type

What is a type?

```
k : {A B : Set} -> A -> B -> A
k x y = x
```

type

A type is a syntactic object which annotates a program and describes how it behaves.

What is a type?

```
k : {A B : Set} -> A -> B -> A
k x y = x
```

type

A type is a syntactic object which annotates a program and describes how it behaves.

The "syntactic" part is important. *We* write the annotation.

What is a type?

```
k : {A B : Set} -> A -> B -> A
k x y = x
```

type

A type is a syntactic object which annotates a program and describes how it behaves.

The "syntactic" part is important. *We* write the annotation.

This is what allows type-checking to happen at compile-time (we don't need semantic information about the term).

The Simply Typed Lambda Calculus

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

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).

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.

The Simply Typed Lambda Calculus

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

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.

The Simply Typed Lambda Calculus

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

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.

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 **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.

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.

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.

Simple Types

Simple Types

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

Simple Types

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

- $\perp$ is a simple type.

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 $A \to B$.

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

Simple Types in Agda

```
data SType : Set where
  B : SType
  _=>_ : SType -> SType -> SType

e1 : SType
e1 = B => B

e2 : SType
e2 = (B => B) => (B => (B => B))
```

(We use **=>** because **->** is special syntax in Agda.)

Typing Statements

$$M : A$$

Typing Statements

$$M : A$$

Definition. A typing statement is a lambda term M together with a simple type A.

Typing Statements

$$M : A$$

Definition. A typing statement is a lambda term M together with a simple type A.

This reads "M is of type A".

Typing Statements

$$\boxed{M} : A$$

subject

Definition. A typing statement is a lambda term M together with a simple type A.

This reads "M is of type A".

Typing Statements

$$\boxed{M}_{\text{subject}} : A$$

Definition. A typing statement is a lambda term M together with a simple type A.

This reads "M is of type A".

A typing statement is meaningless without more information.

How can I know the type of $\lambda x.y$ without knowing the type of y?

Contexts

$$x : A, y : B, \ldots, z : C$$

Contexts

$$x : A, y : B, \ldots, z : C$$

Definition. A **context** is a sequence of typing statements in which the subject is a variable.

Contexts

$$x : A, y : B, \ldots, z : C$$

Definition. A **context** is a sequence of typing statements in which the subject is a variable.

This reads "assuming x is of type A and y is of type B..."

Contexts

$$x : A, y : B, \ldots, z : C$$

Definition. A **context** is a sequence of typing statements in which the subject is a variable.

This reads "assuming x is of type A and y is of type B..."

General typing statements are understood with respect to a context.

Typing Judgements

$$y_1 : A_1, \ldots, y_k : A_k \vdash M : B$$

Typing Judgements

$$y_1 : A_1, \ldots, y_k : A_k \vdash M : B$$

Definition. A typing judgement is a context together with a typing statement. It reads:

Typing Judgements

$$y_1 : A_1, \ldots, y_k : A_k \vdash M : B$$

Definition. A typing judgement is a context together with a typing statement. It reads:

"M is of type B given that
y_1 is of type A_1 and ... and
y_k is of type A_k"

Typing Judgements

$$y_1 : A_1, \ldots, y_k : A_k \vdash M : B$$

Definition. A typing judgement is a context together with a typing statement. It reads:

"M is of type B given that
y_1 is of type A_1 and ... and
y_k is of type A_k"

Recall: Judgements (Sequents)

$$\Phi_1, \ldots, \Phi_k \vdash_{\mathscr{P}} \Psi$$

Recall: Judgements (Sequents)

$$\Phi_1, \ldots, \Phi_k \vdash_{\mathscr{P}} \Psi$$

This reads "If $\Phi_1, \ldots, \Phi_k$ hold then Ψ holds."

Recall: Judgements (Sequents)

$$\underbrace{\Phi_1, \ldots, \Phi_k}_{\texttt{context}} \vdash_{\mathscr{P}} \Psi$$

This reads "If $\Phi_1, \ldots, \Phi_k$ hold then Ψ holds."

$\Phi_1, \ldots, \Phi_k$ are assumptions, a.k.a antecedents, a.k.a the context.

Recall: Judgements (Sequents)

$$\underbrace{\Phi_1, \ldots, \Phi_k}_{\text{context}} \vdash_{\mathscr{P}} \Psi$$

This reads "If $\Phi_1, \ldots, \Phi_k$ hold then Ψ holds."

$\Phi_1, \ldots, \Phi_k$ are assumptions, a.k.a antecedents, a.k.a the context.

Ψ is a conclusion, a.k.a. consequent.

Recall: Judgements (Sequents)

$$\underbrace{\Phi_1, \ldots, \Phi_k}_{\texttt{context}} \vdash_{\mathscr{P}} \Psi$$

This reads "If $\Phi_1, \ldots, \Phi_k$ hold then Ψ holds."

$\Phi_1, \ldots, \Phi_k$ are assumptions, a.k.a antecedents, a.k.a the context.

Ψ is a conclusion, a.k.a. consequent.

$\Phi_1, \ldots, \Phi_k$ and Ψ are statements. For resolution that meant **clauses**, for STLC it means **typing statements.**

Recall: Inference Rules

$$\frac{J_1 \quad J_2 \quad \ldots \quad J_n}{J_{n+1}} \ (\texttt{condition})$$

In **inference rule** an way of describing an individual step in a derivation.

It reads: "If the judgments $J_1, \ldots, J_n$ hold and the **condition** is met, then judgment J_{n+1} holds."

Recall: Derivation Trees

Recall: Derivation Trees

A type system $\mathscr{T}$ is defined by a collection of inference rules.

Recall: Derivation Trees

A type system $\mathscr{T}$ is defined by a collection of inference rules.

Definition. A **derivation tree for** $\mathscr{T}$ is a tree in which every node is a judgment and a node with its parents represent inference rules.

Recall: Derivation Trees

A type system $\mathscr{T}$ is defined by a collection of inference rules.

Definition. A **derivation tree for $\mathscr{T}$** is a tree in which every node is a judgment and a node with its parents represent inference rules.

Leaves are called **axioms.**

Recall: Derivation Trees

A type system $\mathscr{T}$ is defined by a collection of inference rules.

Definition. A **derivation tree for** $\mathscr{T}$ is a tree in which every node is a judgment and a node with its parents represent inference rules.

Leaves are called **axioms.**

We say that $\Gamma \vdash \Psi$ holds if there is a derivation tree which has this judgment at the root.

Type Typing Statements

$$\Gamma \vdash A : \mathsf{Type}$$

As an abuse of notation, we will write the above judgment to mean that A is a valid simple type.

Note that the context is unnecessary because **types have no variables.**

Simply Typed Lambda Calculus (Types)

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

`These are the same as our inductive rules, but written as typing judgments.`

Example

Let's derive $\emptyset \vdash (\bot \to \bot) \to (\bot \to \bot)$.

Simply Typed Lambda Calculus (Terms)

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

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

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

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

(x does not appear in Γ)

Simply Typed Lambda Calculus (Terms)

start

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

abstraction

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

weakening

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

application

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

(x does not appear in Γ)

Example

Let's derive $y : \bot \to \bot \vdash \lambda x \,.\, \lambda z \,.\, yz : \bot \to \bot \to \bot$ *.*

Curry Typing vs. Church Typing

Curry Typing vs. Church Typing

We're consider a **domain-free** (Currying-Typed) version of STLC. Terms are standard lambda terms:

$$\vdash \lambda f.\lambda x.fx : (\perp \to \perp) \to \perp \to \perp$$

Curry Typing vs. Church Typing

We're consider a **domain-free** (Currying-Typed) version of STLC. Terms are standard lambda terms:

$$\vdash \lambda f.\lambda x.fx : (\perp \to \perp) \to \perp \to \perp$$

In **domainful** (Church-Typed) versions, abstractions are annotated with types:

$$\vdash \lambda f^{\perp \to \perp}.\lambda x^{\perp}.fx : (\perp \to \perp) \to \perp \to \perp$$

Uniqueness of Types

Theorem. In domainful STCL, if

$$\Gamma \vdash M : A \text{ and } \Gamma \vdash M : B$$

then $A = B$.

This is not true for the domain-free version.

Strong Normalization of STLC

Typeability

Typeability

Definition. We say that a lambda term M is **typeable** in STLC if there is a context Γ and simple type A such that $\Gamma \vdash M : A$.

Typeability

Definition. We say that a lambda term M is **typeable** in STLC if there is a context Γ and simple type A such that $\Gamma \vdash M : A$.

Example. The term $\lambda x . xx$ is not typeable, whereas $\lambda x . \lambda y . x$ is typeable.

Normalization

Normalization

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

Normalization

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

This means:

Normalization

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

This means:

- All programs we can write in STLC terminate (STLC is not *Turing-complete*)

Normalization

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

This means:

- All programs we can write in STLC terminate (STLC is not *Turing-complete*)
- In STLC, we can use *any* reduction strategy

Weak Normalization of STLC

Alan Turing proved that STLC is weakly normalizing (every term has a normal form).

He did this in a letter, several decades before the first proof was made public.

An Early Proof of Normalization by A.M. Turing

R.O. Gandy

Dedicated to H.B. Curry on the occasion of his 80th birthday

In the extract printed below, Turing shows that every formula of Church's simple type theory has a normal form. The extract is the first page of an unpublished (and incomplete) typescript entitled 'Some theorems about Church's system'. (Turing left his manuscripts to me; they are deposited in the library of King's College, Cambridge). An account of this system was published by Church in 'A formulation of the simple theory of types' (J. Symbolic Logic 5 (1940), pp. 56-68). Church had previously described the system in lectures given at Princeton (1937-38) which Turing attended; he was a graduate student at Princeton 1936-1938. He is mentioned as having contributed to results about the system in footnote 12 of Church's paper. In an undated letter to M.H.A.

Motivation

Motivation

Since ULC has non-normalizing terms, the proof better use types in a meaningful way.

Motivation

Since ULC has non-normalizing terms, the proof better use types in a meaningful way.

We will induct on the *structure of derivations*.

Motivation

Since ULC has non-normalizing terms, the proof better use types in a meaningful way.

We will induct on the *structure of derivations*.

The trick will be, we will have a different inductive hypothesis for derivations of terms with different types.

Motivation

Since ULC has non-normalizing terms, the proof better use types in a meaningful way.

We will induct on the *structure of derivations*.

The trick will be, we will have a different inductive hypothesis for derivations of terms with different types.

(we will roughly follow some very nice notes by Beta Ziliani and Derek Dreyer)

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 applied**.

Reminder: STLC Inference Rules

$$\frac{\Gamma \vdash A : \text{Type}}{\Gamma, x : A \vdash x : A}$$

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

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : \text{Type}}{\Gamma, x : B \vdash M : A}$$

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

Example

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

Let's prove this...

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

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

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

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

New substitutions can make new redexes.

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.

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

The Trick

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

$P_{\Gamma,\bot}(M) = M$ is strongly normalizing

$P_{\Gamma,A\to B}(M) =$ for any N such that $\Gamma \vdash N : A$ and N is SN, MN is also 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 $y_1 : A_1, \ldots, y_k : A_k \vdash M : B$ and

$\Gamma \vdash N_1 : A_1$ and ... 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 a stronger claim so that we have a stronger induction hypothesis.

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

$\Gamma \vdash N_1 : A_1$ and ... and $\Gamma \vdash N_1 : A_k$ and

each $N_1, \ldots, N_k$ are SN, then so is M.