Explicit Substitutions Part 1: STLC and Background

slides: https://github.com/donovancrichton/Talks

Donovan Crichton

April 2025

The goal of this talk

- How to read (theory) syntax.
- STLC - Simply Typed Lambda Calculus.
- Substituions in anger - De-Bruijn Indicies.
- Explicit Subsitutions.
- Explicit Substitutions in modern type theory.

Reading Syntax: Grammar

Natural Numbers (Peano)

$$\mathbb{N} ::= Z \mid S\,\mathbb{N}$$

Untyped Lambda Caluclus

$$\begin{aligned} V &::= x, y, z, \ \ldots & \\ M, N &::= V & \text{Variable.} \\ &\mid M\,N & \text{Application.} \\ &\mid \lambda V.M & \text{Abstraction.} \end{aligned}$$

Reading Syntax: Grammar

Natural Numbers (Peano)

$$\mathbb{N} ::= \; Z \mid S\,\mathbb{N}$$

Reading Syntax: Grammar

Our <symbol> name...

Natural Numbers (Peano)

$$\boxed{\mathbb{N}} ::= Z \mid S\,\mathbb{N}$$

Reading Syntax: Grammar

...is defined in the following ways:

Natural Numbers (Peano)

$$\mathbb{N} \boxed{::=} Z \mid S\,\mathbb{N}$$

Reading Syntax: Grammar

The letter Z by itself.

Natural Numbers (Peano)

$$\mathbb{N} ::= \boxed{Z} \mid S\,\mathbb{N}$$

Reading Syntax: Grammar

...or...

Natural Numbers (Peano)

$$\mathbb{N} ::= \; Z \boxed{\,|\,} S\,\mathbb{N}$$

Reading Syntax: Grammar

The letter S followed by a space, followed by any $\mathbb{N}$.

Natural Numbers (Peano)

$$\mathbb{N} ::= \; Z \mid \boxed{S\,\mathbb{N}}$$

Reading Syntax: Grammar

Our set of expressions/terms called V...

Untyped Lambda Caluclus

$$
\begin{aligned}
\boxed{V} &::= x, y, z, \ \ldots \\
M, N &::= V && \text{Variable.} \\
&\mid M\ N && \text{Application.} \\
&\mid \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

Reading Syntax: Grammar

is given, or defined by:

Untyped Lambda Caluclus

$$
\begin{aligned}
V \boxed{::=}\ & x, y, z,\ \ldots \\
M, N ::=\ & V && \text{Variable.} \\
\mid\ & M\ N && \text{Application.} \\
\mid\ & \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

Reading Syntax: Grammar

'x', 'y', 'z', or *any other lower case letter* (lower case words also implied)

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= \boxed{x, y, z, \ \ldots} \\
M, N &::= V && \text{Variable.} \\
&\mid M\ N && \text{Application.} \\
&\mid \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

Reading Syntax: Grammar

Our lambda terms, denoted by N or M
(other capital letters implied).

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= x, y, z, \ \dots \\
\boxed{M, N} &::= V && \text{Variable.} \\
&\mid M\,N && \text{Application.} \\
&\mid \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

Reading Syntax: Grammar

are given by:

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= \; x, y, z, \; \ldots \\
M, N &\boxed{::=} \; V && \text{Variable.} \\
&\mid M \; N && \text{Application.} \\
&\mid \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

Reading Syntax: Grammar

A V...

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= \; x, y, z, \; \ldots \\
M, N &::= \; \boxed{V} && \text{Variable.} \\
&\quad\; \mid M\ N && \text{Application.} \\
&\quad\; \mid \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

Reading Syntax: Grammar

…or,

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= \; x, y, z, \; \ldots \\
M, N &::= \; V && \text{Variable.} \\
& \;\; \boxed{\mid} \; M\,N && \text{Application.} \\
& \;\; \mid \; \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

Reading Syntax: Grammar

A lambda term (M), followed by a space, followed by another lambda term (N).

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= x, y, z, \ \dots \\
M, N &::= V && \text{Variable.} \\
&\mid \boxed{M\,N} && \text{Application.} \\
&\mid \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

Reading Syntax: Grammar

Or,

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= \; x, y, z, \; \ldots \\
M, N &::= \; V && \text{Variable.} \\
&\quad \mid M \; N && \text{Application.} \\
&\quad \boxed{\mid} \; \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

Reading Syntax: Grammar

The λ symbol, followed by a V element,
followed by a ".", followed by a lambda term (M)

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= x, y, z, \ \ldots & \\
M, N &::= V & \text{Variable.} \\
&\mid M\ N & \text{Application.} \\
&\mid \boxed{\lambda V.M} & \text{Abstraction.}
\end{aligned}
$$

Why does Grammar look like this?

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= x, y, z, \; \ldots \\
M, N &::= V && \text{Variable.} \\
&\mid M\, N && \text{Application.} \\
&\mid \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

Why does Grammar look like this?

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= x, y, z, \ \ldots \\
M, N &::= V && \text{Variable.} \\
&\mid M\ N && \text{Application.} \\
&\mid \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

- The smallest possible definition.

Why does Grammar look like this?

Untyped Lambda Caluclus

$$
\begin{aligned}
V &::= x, y, z, \; \ldots \\
M, N &::= V && \text{Variable.} \\
&\mid M\,N && \text{Application.} \\
&\mid \lambda V.M && \text{Abstraction.}
\end{aligned}
$$

- The smallest possible definition.
- Can be used to generate arbitrary elements.

What about in programming?

Natural Numbers (Peano)

```haskell
data Nat = Z | S Nat

zero :: Nat
zero = Z

one :: Nat
one = S Z

two :: Nat
two = S (S Z)

three :: Nat
three = S two
```

What about in programming?

Untyped Lambda Caluclus (Idris)

```
V : Type
V = String

data Λ = Var V
       | App Λ Λ
       | Abs V Λ

id : Λ
id = Abs "x" (Var "x")

const : Λ
const = Abs "a" (Abs "b" (Var "a"))
```

Church Encoding - Naturals and Booleans

Church Booleans

$$\text{True} = \lambda a.\lambda b.a$$
$$\text{False} = \lambda a.\lambda b.b$$

Church Encoding - Naturals and Booleans

Church Booleans

$$\text{True} = \lambda a.\lambda b.a$$
$$\text{False} = \lambda a.\lambda b.b$$

Church Naturals

$$0 = \lambda f.\lambda x.x$$
$$1 = \lambda f.\lambda x.f\,x$$
$$2 = \lambda f.\lambda x.f\,(f\,x)$$
$$\vdots$$

Church Encoding - Functions and Predicates

We use our definitions of True and False from earlier.

Functions and Predicates

$$\begin{aligned} \text{Succ}(n, f, x) &= \lambda n.\lambda f.\lambda x.\lambda f\,(n\ f\ x) \\ \text{Add}(m, n, f, x) &= \lambda m.\lambda n.\lambda f.\lambda x.m\ f\,(n\ f\ x) \\ \text{IsZero}(n) &= \lambda n.n\ (\lambda x.\text{False})\ \text{True} \end{aligned}$$

Church Encoding - Functions and Predicates

As homework, trace through IsZero and convince yourself that IsZero 0 returns True, and IsZero for any other number returns false.

Functions and Predicates

$$\begin{aligned}
\text{Succ}(n, f, x) &= \lambda n.\lambda f.\lambda x.\lambda f\,(n\ f\ x) \\
\text{Add}(m, n, f, x) &= \lambda m.\lambda n.\lambda f.\lambda x.m\ f\,(n\ f\ x) \\
\text{IsZero}(n) &= \lambda n.n\ (\lambda x.\lambda a.\lambda b.b)\ \lambda a.\lambda b.a
\end{aligned}$$

Reading Syntax: Typing Rules

Is this scary?

Typing Rules

$$\frac{\Gamma \vdash A \ \ \text{Type} \qquad \Gamma, x : A \vdash B \ \ \text{Type}}{\Gamma \vdash A \to B \ \ \text{Type}}(\mathit{Ty\text{-}Arrow})$$

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

$$\frac{\Gamma \vdash A \to B \ \text{Type} \qquad \Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}(\mathit{App})$$

The Program Context

The current scope of our program.

The Program Context

$$\Gamma, \Delta ::= \diamond \mid \Gamma, x : A$$

The Program Context

Our context called Γ or Δ, etc...

The Program Context

$$\boxed{\Gamma, \Delta} ::= \diamond \mid \Gamma, x : A$$

The Program Context

Is defined by:

The Program Context

$$\Gamma, \Delta \boxed{::=} \diamond \mid \Gamma, x : A$$

The Program Context

Empty (like the empty list) represented by $\diamond$.

The Program Context

$$\Gamma, \Delta ::= \boxed{\diamond} \mid \Gamma, x : A$$

The Program Context

Or,

The Program Context

$$\Gamma, \Delta ::= \diamond \boxed{\mid} \Gamma, x : A$$

The Program Context

Our program scope, extended with a variable 'x' of type 'A'.

The Program Context

$$\Gamma, \Delta ::= \diamond \mid \boxed{\Gamma, x : A}$$

Example Contexts

What is the context of this program?

```
-- assuming   empty context here.
-- assuming A → B : Type here.
data Bool = True | False

not : Bool -> Bool
not True  = False
not False = True
```

Example Contexts

What is the context of this program?

```
-- assuming   empty context here.
-- assuming A → B : Type here.
data Bool = True | False

not : Bool -> Bool
not True  = False
not False = True
```

Answer.

$$\Gamma = ???$$

Example Contexts

What is the context of this program?

```
-- assuming   empty context here.
-- assuming A → B : Type here.
data Bool = True | False

not : Bool -> Bool
not True  = False
not False = True
```

Answer.

$$\Gamma = \diamond, \ldots$$

Example Contexts

What is the context of this program?

```
-- assuming   empty context here.
-- assuming A → B : Type here.
data Bool = True | False

not : Bool -> Bool
not True  = False
not False = True
```

Answer.

$$\Gamma = \diamond, A \to B : \text{Type}, \ldots$$

Example Contexts

What is the context of this program?

```
-- assuming   empty context here.
-- assuming A → B : Type here.
data Bool = True | False

not : Bool -> Bool
not True  = False
not False = True
```

Answer.

$$\Gamma = \diamond, A \to B : \text{Type}, \text{Bool} : \text{Type}, \text{True} : \text{Bool}, \text{False} : \text{Bool}, \ldots$$

Example Contexts

What is the context of this program?

```
-- assuming   empty context here.
-- assuming A → B : Type here.
data Bool = True | False

not : Bool -> Bool
not True  = False
not False = True
```

Answer.

$$\Gamma = \diamond, A \rightarrow B : \text{Type}, \text{Bool} : \text{Type}, \text{True} : \text{Bool}, \text{False} : \text{Bool}, \text{not} : \text{Bool} \rightarrow \text{Bool}$$

The Function Type Formation Rule

If the derivations (true statements) hold above the line (premise), then the derivations hold below the line (conclusion).

The Type Formation Rule

$$\frac{\qquad}{\qquad}\,(\textit{Ty-Arrow})$$

The Function Type Formation Rule

...from an arbitrary context, Γ

The Type Formation Rule

$$\frac{\boxed{\Gamma}}{}\;(\textit{Ty-Arrow})$$

The Function Type Formation Rule

...we may derive (produce, obtain...)

The Type Formation Rule

$$\frac{\Gamma\boxed{\vdash}}{}(\textit{Ty-Arrow})$$

The Function Type Formation Rule

...an arbitrary type, called A

The Type Formation Rule

$$\frac{\Gamma \vdash \boxed{A \text{ Type}}}{\phantom{\Gamma \vdash A \text{ Type}}}\,(\mathit{Ty\text{-}Arrow})$$

The Function Type Formation Rule

AND

The Type Formation Rule

$$\frac{\Gamma \vdash A \text{ Type} \quad \boxed{}}{}\,(\textit{Ty-Arrow})$$

The Function Type Formation Rule

...From the same arbitrary context Γ, extended with a variable 'x' of type A

The Type Formation Rule

$$\frac{\Gamma \vdash A \text{ Type} \qquad \boxed{\Gamma, x : A}}{} (\textit{Ty-Arrow})$$

The Function Type Formation Rule

...we may derive

The Type Formation Rule

$$\frac{\Gamma \vdash A \text{ Type} \qquad \Gamma, x : A \boxed{\vdash}}{}\,(\mathit{Ty\text{-}Arrow})$$

The Function Type Formation Rule

some arbitrary type B.

The Type Formation Rule

$$\frac{\Gamma \vdash A \ \mathsf{Type} \qquad \Gamma, x : A \vdash \boxed{B \ \mathsf{Type}}}{}\,(\textit{Ty-Arrow})$$

The Function Type Formation Rule

Then, from that arbitrary Γ (program scope)

The Type Formation Rule

$$\frac{\Gamma \vdash A \text{ Type} \qquad \Gamma, x : A \vdash B \text{ Type}}{\boxed{\Gamma}} \, (\textit{Ty-Arrow})$$

The Function Type Formation Rule

...we may derive

The Type Formation Rule

$$\frac{\Gamma \vdash A \;\mathsf{Type} \qquad \Gamma, x : A \vdash B \;\mathsf{Type}}{\Gamma \boxed{\vdash}} (\mathit{Ty\text{-}Arrow})$$

The Function Type Formation Rule

...an arrow type between them

The Type Formation Rule

$$\frac{\Gamma \vdash A \ \mathsf{Type} \qquad \Gamma, x : A \vdash B \ \mathsf{Type}}{\Gamma \vdash \boxed{A \to B}}(\textit{Ty-Arrow})$$

The Function Type Formation Rule

...that is also a type.

The Type Formation Rule

$$\frac{\Gamma \vdash A \text{ Type} \qquad \Gamma, x : A \vdash B \text{ Type}}{\Gamma \vdash A \to B \boxed{\text{ Type}}}(\textit{Ty-Arrow})$$

The Function Term Formation Rule

If the premises hold above, so the conclusion holds below.

The Function Term Former

$$\frac{\qquad}{\qquad}(Abs)$$

The Function Term Formation Rule

From our program scope Γ we can obtain an arbitrary function type $A \to B$.

The Function Term Former

$$\frac{\boxed{\Gamma \vdash A \to B \text{ Type}}}{}\,(Abs)$$

The Function Term Formation Rule

..And..

The Function Term Former

$$\frac{\Gamma \vdash A \to B \text{ Type} \quad \boxed{}}{} (Abs)$$

The Function Term Formation Rule

From our program scope $\Gamma, x : A$ we can derive a term $M : B$.

The Function Term Former

$$\frac{\Gamma \vdash A \to B \text{ Type} \qquad \boxed{\Gamma, x : A \vdash M : B}}{}(Abs)$$

The Function Term Formation Rule

Then we may derive a lambda abstraction term, where x has type A and the body is M, where the entire abstraction has type $A \to B$.

The Function Term Former

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

The Function Term Elimination Rule

If the premises hold above, so the conclusion holds below.

The Function Term Eliminator

$$\frac{\qquad}{\qquad}(App)$$

The Function Term Elimination Rule

From our program scope Γ we can obtain some term M with type $A \to B$.

The Function Term Eliminator

$$\frac{\boxed{\Gamma \vdash M : A \to B}}{}\,(App)$$

The Function Term Elimination Rule

And, from Γ we can also obtain a term N of type A.

The Function Term Eliminator

$$\frac{\Gamma \vdash M : A \to B \qquad \boxed{\Gamma \vdash N : A}}{}(App)$$

The Function Term Elimination Rule

Then we can produce a term $M\,N$ of type B.

The Function Term Eliminator

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\boxed{\Gamma \vdash M\,N : B}}(App)$$

STLC: Grammar

The Simply Typed Lambda Calculus: Grammar

The Simply Typed Lambda Calculus: Grammar

$V ::= \ x, y, z, \ \dots$ Variable Set

STLC: Grammar

The Simply Typed Lambda Calculus: Grammar

$$
\begin{aligned}
V &::= \; x, y, z, \; \ldots && \text{Variable Set} \\
\alpha, \beta &::= \; * \mid \alpha \to \beta && \text{Types}
\end{aligned}
$$

STLC: Grammar

The Simply Typed Lambda Calculus: Grammar

$$V ::= \; x, y, z, \; \ldots \qquad \text{Variable Set}$$

$$\alpha, \beta ::= \; * \mid \alpha \to \beta \qquad \text{Types}$$

$$M, N ::= V \mid M\, N \mid \lambda(V : \alpha).M \qquad \text{Terms}$$

STLC: Grammar

The Simply Typed Lambda Calculus: Grammar

$$
\begin{aligned}
V &::= \; x, y, z, \; \ldots && \text{Variable Set} \\
\alpha, \beta &::= \; * \mid \alpha \to \beta && \text{Types} \\
M, N &::= V \mid M\, N \mid \lambda(V : \alpha).M && \text{Terms} \\
\Gamma &::= \; \diamond \mid \Gamma, x : \alpha && \text{Contexts}
\end{aligned}
$$

STLC: Grammar

The Simply Typed Lambda Calculus: Grammar

$$
\begin{aligned}
V &::= \; x, y, z, \; \ldots && \text{Variable Set} \\
\alpha, \beta &::= \; * \mid \alpha \to \beta && \text{Types} \\
M, N &::= V \mid M\, N \mid \lambda(V : \alpha).M && \text{Terms} \\
\Gamma &::= \; \diamond \mid \Gamma, x : \alpha && \text{Contexts}
\end{aligned}
$$

Why $x : \alpha$ in the Contexts set, and not $V : \alpha$?

STLC: Grammar

The Simply Typed Lambda Calculus: Grammar

$$
\begin{aligned}
V &::= x, y, z, \; ... && \text{Variable Set} \\
\alpha, \beta &::= * \mid \alpha \to \beta && \text{Types} \\
M, N &::= V \mid M\,N \mid \lambda(V : \alpha).M && \text{Terms} \\
\Gamma &::= \diamond \mid \Gamma, x : \alpha && \text{Contexts}
\end{aligned}
$$

Why $x : \alpha$ in the Contexts set, and not $V : \alpha$?

V is the variable set in our *object* language, meaning it is part of the syntax of the language we are defining.
x represents a variable in our meta language - the language we use to specify and reason about our object language.

STLC: Typing Rules/Judgements

STLC: Typing Rules

$$\frac{\Gamma \vdash \alpha \text{ Type} \qquad \Gamma, x : \alpha \vdash \beta \text{ Type}}{\Gamma \vdash \alpha \to \beta \text{ Type}}(\to\textit{-Form})$$

STLC: Typing Rules/Judgements

STLC: Typing Rules

$$\frac{\Gamma \vdash \alpha \text{ Type} \qquad \Gamma, x : \alpha \vdash \beta \text{ Type}}{\Gamma \vdash \alpha \to \beta \text{ Type}} (\to\textit{-Form})$$

$$\frac{\Gamma \vdash \mathit{alpha} \to \beta \text{ Type} \qquad \Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda(x : \alpha).M : \alpha \to \beta} (\to\textit{-Intro})$$

STLC: Typing Rules/Judgements

STLC: Typing Rules

$$\frac{\Gamma \vdash \alpha \ \mathsf{Type} \qquad \Gamma, x : \alpha \vdash \beta \ \mathsf{Type}}{\Gamma \vdash \alpha \to \beta \ \mathsf{Type}} (\to\textit{-Form})$$

$$\frac{\Gamma \vdash \alpha \to \beta \ \mathsf{Type} \qquad \Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda(x : \alpha).M : \alpha \to \beta} (\to\textit{-Intro})$$

$$\frac{\Gamma \vdash \alpha \to \beta \ \mathsf{Type} \qquad \Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash M\, N : \beta} (\to\textit{-Elim})$$

Free Variables

FV

A variable is x considered *free* in a lambda term if it refers to a binding outside the *scope* of the current lambda term. Defined inductively.

Free Variables

The set *FV* of free variables.

A variable is x considered *free* in a lambda term if it refers to a binding outside the *scope* of the current lambda term. Defined inductively.

$$\begin{aligned} FV(V) &= V \\ FV(M\ N) &= FV(M) \cup FV(N) \\ FV(\lambda(V : \alpha).M) &= FV(M) \backslash V \end{aligned}$$

Free Variables

The set *FV* of free variables.

A variable is x considered *free* in a lambda term if it refers to a binding outside the *scope* of the current lambda term. Defined inductively.

$$
\begin{aligned}
FV(V) &= V \\
FV(M\,N) &= FV(M) \cup FV(N) \\
FV(\lambda(V:\alpha).M) &= FV(M) \backslash V
\end{aligned}
$$

Examples

$$
\begin{array}{ll}
\lambda(x:\alpha \to \beta).\lambda(y:\beta).x\,y & FV = \emptyset \\
\lambda(x:\alpha \to \beta).\boxed{\lambda(y:\beta).x\,y} & \mathrm{FV} = \{x\}
\end{array}
$$

α-equivalence/conversion

$\lambda(x : \beta).x = \lambda(y : \beta).y$

Two lambda terms are said to be α-*equivalent* if we can rename bound variables and still have the same term. We may α-convert or α-rename a lambda term into another α-equivalence term.

α-equivalence/conversion

$\lambda(x:\beta).x = \lambda(y:\beta).y$

Two lambda terms are said to be α-*equivalent* if we can rename bound variables and still have the same term. We may α-convert or α-rename a lambda term into another α-equivalence term.

Examples of α-equivalence.

$$\lambda(x:\beta).x = \lambda(y:\beta).y$$

α-equivalence/conversion

$\lambda(x : \beta).x = \lambda(y : \beta).y$

Two lambda terms are said to be α-*equivalent* if we can rename bound variables and still have the same term. We may α-convert or α-rename a lambda term into another α-equivalence term.

Examples of α-equivalence.

$$\lambda(x : \beta).x = \lambda(y : \beta).y$$
$$\lambda(x : \beta).\lambda(x : \beta).x = \lambda(y : \beta).\lambda(x : \beta).x$$

α-equivalence/conversion

$\lambda(x : \beta).x = \lambda(y : \beta).y$

Two lambda terms are said to be α-*equivalent* if we can rename bound variables and still have the same term. We may α-convert or α-rename a lambda term into another α-equivalence term.

Examples of α-equivalence, and non-equivalence.

$$\lambda(x : \beta).x = \lambda(y : \beta).y$$
$$\lambda(x : \beta).\lambda(x : \beta).x = \lambda(y : \beta).\lambda(x : \beta).x$$
$$\lambda(x : \beta).\lambda(x : \beta).x \neq \lambda(y : \beta).\lambda(x : \beta).y$$

Substitution

$M[x := N]$ or $M[N/x]$ or $M[x \mapsto N]$

Substitution is a *meta* operation that allows us to perform computation with lambda calculus. We may replace (almost) any variable x that occurs in term M with term N.

Substitution

$M[x := N]$ or $M[N/x]$ or $M[x \mapsto N]$

Substitution is a *meta* operation that allows us to perform computation with lambda calculus. We may replace (almost) any variable x that occurs in term M with term N.

Example Substitutions

$$(\lambda(x : \mathbb{N}).2 + x)[3/x] = \lambda(x : \mathbb{N}).2 + 3$$

Substitution

$M[x := N]$ or $M[N/x]$ or $M[x \mapsto N]$

Substitution is a *meta* operation that allows us to perform computation with lambda calculus. We may replace (almost) any variable x that occurs in term M with term N.

Example Substitutions

$$(\lambda(x : \mathbb{N}).2 + x)[3/x] = \lambda(x : \mathbb{N}).2 + 3$$
$$(\lambda(x : \beta \to \beta).x)[\lambda(y : \beta).y/x] = \lambda(x : \beta \to \beta).\lambda(y : \beta).y$$

Substitution

$M[x := N]$ or $M[N/x]$ or $M[x \mapsto N]$

Substitution is a *meta* operation that allows us to perform computation with lambda calculus. We may replace (almost) any variable x that occurs in term M with term N.

Example Substitutions

$$(\lambda(x : \mathbb{N}).2 + x)[3/x] = \lambda(x : \mathbb{N}).2 + 3$$

$$(\lambda(x : \beta \to \beta).x)[\lambda(y : \beta).y/x] = \lambda(x : \beta \to \beta).\lambda(y : \beta).y$$

$$(\lambda(x : \beta).y)[x/y] \neq \lambda(x : \beta).x$$

Capture Avoiding Substituion

Rule 1. $x[T/x] = T$

When substituting into a variable, we just have the term.

Capture Avoiding Substituion

Rule 1. $x[T/x] = T$

When substituting into a variable, we just have the term.

Rule 2. $y[T/x] = y$ if $x \neq y$

Substituting by a different variable does nothing.

Capture Avoiding Substituion

Rule 1. $x[T/x] = T$

When substituting into a variable, we just have the term.

Rule 2. $y[T/x] = y$ if $x \neq y$

Substituting by a different variable does nothing.

Rule 3. $(M\ N)[T/x] = M[T/x]\ N[T/x]$

Substitution distributes over application.

Capture Avoiding Substituion

Rule 1. $x[T/x] = T$

When substituting into a variable, we just have the term.

Rule 2. $y[T/x] = y$ if $x \neq y$

Substituting by a different variable does nothing.

Rule 3. $(M\ N)[T/x] = M[T/x]\ N[T/x]$

Substitution distributes over application.

Rule 4. $(\lambda(x : \alpha).M)[T/x] = \lambda(x : \alpha).M$

Abstraction substitution is neutral for the binding variable.

Capture Avoiding Substituion

Rule 1. $x[T/x] = T$

When substituting into a variable, we just have the term.

Rule 2. $y[T/x] = y$ if $x \neq y$

Substituting by a different variable does nothing.

Rule 3. $(M\ N)[T/x] = M[T/x]\ N[T/x]$

Substitution distributes over application.

Rule 4. $(\lambda(x : \alpha).M)[T/x] = \lambda(x : \alpha).M$

Abstraction substitution is neutral for the binding variable.

Rule 5. $(\lambda(y : \alpha).M)[T/x] = \lambda(y : \alpha).M[T/x]$ if:
$x \neq y$ and $y \notin FV(T)$

Substitution is conditional through an abstraction body.

Computation

β-reduction: $(\lambda(x : \alpha).M)\ N \to M[N/x]$

β-reduction is the essence of computation for the STLC.
If we see an application after an abstraction, we may perform the given substitution then strip off the lambda binder.

Computation

β-reduction: $(\lambda(x:\alpha).M)\ N \to M[N/x]$

β-reduction is the essence of computation for the STLC.
If we see an application after an abstraction, we may perform the given substitution then strip off the lambda binder.

Example of *beta*-reduction.

$$
\begin{aligned}
(\lambda(x:\alpha).x)\ y &\to \lambda(x:\alpha).x[y/x] && \text{replace abs-app with substitution.}\\
&\to \lambda(x:\alpha).y && \text{perform the substitution.}\\
&\to y && \text{eliminate the relevant binder.}
\end{aligned}
$$

De Bruijn Indices: (1972) Lambda Calculus Notation with Nameless Dummies ...

Nicolaas Govert
De Bruijn

De-Bruijn Indices

Replacing letters with numbers (why?)

Instead of using variable names to denote variable terms, we use numbers to denote a distance from the relevant binder. Best shown by example.

De-Bruijn Indices

Replacing letters with numbers (why?)

Instead of using variable names to denote variable terms, we use numbers to denote a distance from the relevant binder. Best shown by example.

De-Bruijn Indexed Lambda Term Examples

$$\lambda(x : \alpha).\lambda(y : \alpha).\lambda(z : \alpha).y\,(x\,z) = \lambda\alpha.\lambda\alpha.\lambda\alpha.1\ (2\ 0)$$

De-Bruijn Indices

Replacing letters with numbers (why?)

Instead of using variable names to denote variable terms, we use numbers to denote a distance from the relevant binder. Best shown by example.

De-Bruijn Indexed Lambda Term Examples

$$\lambda(x : \alpha).\lambda(y : \alpha).\lambda(z : \alpha).y\,(x\,z) = \lambda\alpha.\lambda\alpha.\lambda\alpha.1\ (2\ 0)$$
$$\lambda(x : \alpha).x = \lambda\alpha.0$$

De-Bruijn Indices

Replacing letters with numbers (why?)

Instead of using variable names to denote variable terms, we use numbers to denote a distance from the relevant binder. Best shown by example.

De-Bruijn Indexed Lambda Term Examples

$$\lambda(x : \alpha).\lambda(y : \alpha).\lambda(z : \alpha).y\,(x\,z) = \lambda\alpha.\lambda\alpha.\lambda\alpha.2\;(3\;1)$$

$$\lambda(x : \alpha).x = \lambda\alpha.1$$

$$\lambda(y : \alpha).y = \lambda\alpha.1$$

Lift-and-shift

Substitution: Given $(\lambda\alpha.M)\ N$

1. Find the instances of the binding sites in $\lambda\alpha.M$.
2. Decrement the free variables in M to account for the removal of the outer binder.
3. Replace the binding sites with N but increment any necessary free variables so they occur under the appropriate number of binders.

(I think we need an example).

Lift-and-shift

1. Identify binding sites in the λ abstraction.

$\lambda\alpha.\lambda\beta.2\ 3))\ (\lambda\alpha.1\ 4) \rightarrow \lambda\alpha.\lambda\beta.\Box\ 3$

2. Decrement free variables that are affected by the removal of the outer binder.

$\lambda\alpha.\lambda\beta.\Box\ 3 \rightarrow \lambda\beta.\Box\ 2$

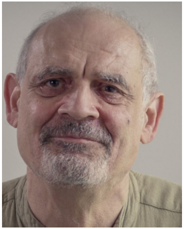
3. Increment the free variables in the applied term to match the number of binders.

$\lambda\beta.\Box\ 2 \rightarrow \lambda\beta.(\lambda\alpha.1\ \boxed{4+1})\ 2$
$\lambda\beta.\Box\ 2 \rightarrow \lambda\beta.(\lambda\alpha.1\ 5)\ 2$

Explicit Substitutions: (1991) Explicit Substitutions

Martin Abadi

Luca Cardelli



Pierre-Louis Curien

Jean-Jacques Levy

Extending the STLC with Explicit Substitutions

STLC with Explicit Substitutions: Grammar

$$\alpha, \beta ::= \ * \mid \alpha \to \beta \qquad \text{Types}$$

Extending the STLC with Explicit Substitutions

STLC with Explicit Substitutions: Grammar

$\alpha, \beta ::= \ * \mid \alpha \to \beta$	Types
$M, N ::= 1 \mid M\ N \mid \lambda\alpha.M \mid M[\gamma]$	Terms

Extending the STLC with Explicit Substitutions

STLC with Explicit Substitutions: Grammar

$$\begin{aligned} \alpha, \beta &::= \; * \mid \alpha \to \beta && \text{Types} \\ M, N &::= 1 \mid M\,N \mid \lambda\alpha.M \mid M[\gamma] && \text{Terms} \\ \gamma, \delta &::= id \mid \uparrow \mid \gamma, M : \alpha \mid \gamma \circ \delta && \text{Substitutions} \end{aligned}$$

Extending the STLC with Explicit Substitutions

STLC with Explicit Substitutions: Grammar

$\alpha, \beta ::= \ * \mid \alpha \to \beta$	Types
$M, N ::= 1 \mid M\,N \mid \lambda\alpha.M \mid M[\gamma]$	Terms
$\gamma, \delta ::= id \mid \uparrow \mid \gamma, M : \alpha \mid \gamma \circ \delta$	Substitutions
$\Gamma ::= \ \diamond \mid \Gamma, \alpha$	Contexts

Additional Terms

$1 \in$ Terms

1 is De Bruijn index 1, the variable pointing to the most recent binder.
This replaces our need for a variable term in our lambda calculus.

Additional Terms

$1 \in$ Terms

1 is De Bruijn index 1, the variable pointing to the most recent binder.
This replaces our need for a variable term in our lambda calculus.

$M[\gamma] \in$ Terms

A substitution of γ in M because *gamma* tells us what and where.

Explicit Substitutions