The Lambda Calculus: An Introduction

Type Theory and Mechanized Reasoning
Lecture 10

CAS CS 400

Introduction

Administrivia

Homework 4 is due on *Thursday by 11:59PM*.

There are notes posted in the course repository and the course website.

Objectives

Agda tutorial: Talk about equality.

Introduce the the syntax and semantics of the lambda calculus.

Agda Tutorial: Propositional Equality

Recall: NonEmtpy

```
data NonEmpty {A : Set} : List A -> Set where
    isNonEmpty :
        (x : A) ->
        (xs : List A) ->
        NonEmpty (x :: xs)

test : NonEmpty (1 :: 2 :: [])
test = isNonEmpty 1 (2 :: [])
```

Recall: NonEmtpy

```
data NonEmpty {A : Set} : List A -> Set where
    isNonEmpty :
        (x : A) ->
        (xs : List A) ->
        NonEmpty (x :: xs)

test : NonEmpty (1 :: 2 :: [])
test = isNonEmpty 1 (2 :: [])
```

NonEmpty has one constructor, and the parameter in the type is a nonempty list.

Recall: NonEmtpy

```
data NonEmpty {A : Set} : List A -> Set where
    isNonEmpty :
        (x : A) ->
        (xs : List A) ->
        NonEmpty (x :: xs)

test : NonEmpty (1 :: 2 :: [])
test = isNonEmpty 1 (2 :: [])
```

NonEmpty has one constructor, and the parameter in the type is a nonempty list.

It is impossible to construct something of whose type is ***NonEmpty [].***

Recall: Head

```
head : {A : Set} -> (l : List A) -> NonEmpty l -> A
head (x :: l) _ = x

foo : NonEmpty [] -> Nat
foo = head [] -- we cannot apply this function to anything

bar : NonEmpty (1 :: []) -> Nat
bar = head (1 :: [])
```

Recall: Head

```
head : {A : Set} -> (l : List A) -> NonEmpty l -> A
head (x :: l) _ = x

foo : NonEmpty [] -> Nat
foo = head [] -- we cannot apply this function to anything

bar : NonEmpty (1 :: []) -> Nat
bar = head (1 :: [])
```

We can use NonEmpty to limit the behavior of our **head** function.

Recall: Head

```
head : {A : Set} -> (l : List A) -> NonEmpty l -> A
head (x :: l) _ = x

foo : NonEmpty [] -> Nat
foo = head [] -- we cannot apply this function to anything

bar : NonEmpty (1 :: []) -> Nat
bar = head (1 :: [])
```

We can use NonEmpty to limit the behavior of our **head** function.

After we apply head to an an argument **l**, we are "blocked" if **l** is empty.

Propositional Equality

```
data _=P_ {A : Set} (x : A) : A -> Set where
    refl : x =P x

foo : 2 =P 2
foo = refl
```

Propositional Equality

```
data _=P_ {A : Set} (x : A) : A -> Set where
    refl : x =P x

foo : 2 =P 2
foo = refl
```

Equality has a single constructor **refl** whose parameters are identical.

Propositional Equality

```
data _=P_ {A : Set} (x : A) : A -> Set where
    refl : x =P x

foo : 2 =P 2
foo = refl
```

Equality has a single constructor **refl** whose parameters are identical.

*It is impossible to construct something of type **2 =P 3** (or equality between non-identical terms).*

Computation and Equality

```
2+2=4 : (2 + 2) =P 4
2+2=4 = refl
```

Computation and Equality

```
2+2=4 : (2 + 2) =P 4
2+2=4 = refl
```

By *identical* we mean they have identical values.

Computation and Equality

```
2+2=4 : (2 + 2) =P 4
2+2=4 = refl
```

By *identical* we mean they have identical values.

This allows us to compare terms after computation *within* types.

Computation and Equality

```
2+2=4 : (2 + 2) =P 4
2+2=4 = refl
```

By *identical* we mean they have identical values.

This allows us to compare terms after computation *within* types.

Use Case: Unit Tests

```
assert-equal : {A : Set} -> (x y : A) -> Set
assert-equal actual expected  = actual =P expected

test1 : assert-equal (3 + 4) (4 + 3)
test1 = refl
```

Use Case: Unit Tests

```
assert-equal : {A : Set} -> (x y : A) -> Set
assert-equal actual expected  = actual =P expected

test1 : assert-equal (3 + 4) (4 + 3)
test1 = refl
```

We can use this to embed unit tests in our code.

Use Case: Unit Tests

```
assert-equal : {A : Set} -> (x y : A) -> Set
assert-equal actual expected  = actual =P expected

test1 : assert-equal (3 + 4) (4 + 3)
test1 = refl
```

We can use this to embed unit tests in our code.

If **actual** is the same as **expected**, then **refl** should be the value of **test1**.

Computing with Equality

```
cong :
  {A B : Set} ->
  {x y : A} ->
  (f : A -> B) -> (x =P y) -> (f x) =P (f y)
cong f refl = refl
```

Computing with Equality

```
cong :
  {A B : Set} ->
  {x y : A} ->
  (f : A -> B) -> (x =P y) -> (f x) =P (f y)
cong f refl = refl
```

(Let's do a demo)

Computing with Equality

```
cong :
  {A B : Set} ->
  {x y : A} ->
  (f : A -> B) -> (x =P y) -> (f x) =P (f y)
cong f refl = refl
```

(Let's do a demo)

*If **x** and **y** are the same, then so are **f x** and **f y**.*

Computing with Equality

```
cong :
  {A B : Set} ->
  {x y : A} ->
  (f : A -> B) -> (x =P y) -> (f x) =P (f y)
cong f refl = refl
```

(Let's do a demo)

*If **x** and **y** are the same, then so are **f x** and **f y**.*

When we pattern match on equality, **refl** is the only possible value. (In the demo, we see what happens to the types.)

More interesting cases: Induction

```
20+0=20 : (20 + 0) =P 20
20+0=20 = refl

n+0=n : (n : Nat) -> (n + 0) =P n
n+0=n n = ?
```

More interesting cases: Induction

```
20+0=20 : (20 + 0) =P 20
20+0=20 = refl

n+0=n : (n : Nat) -> (n + 0) =P n
n+0=n n = ?
```

We can compute **(20 + 0)** to get the value **20.**

More interesting cases: Induction

```
20+0=20 : (20 + 0) =P 20
20+0=20 = refl

n+0=n : (n : Nat) -> (n + 0) =P n
n+0=n n = ?
```

`We can compute` **`(20 + 0)`** `to get the value` **`20.`**

`But we can't compute` **`(n + 0)`**`,` *`we don't know what`* ***`n`*** *`is.`*

More interesting cases: Induction

```
n+0=n : (n : Nat) -> (n + 0) =P n
n+0=n zero = refl
n+0=n (suc n) = cong suc (n+0=n n)
```

More interesting cases: Induction

```
n+0=n : (n : Nat) -> (n + 0) =P n
n+0=n zero = refl
n+0=n (suc n) = cong suc (n+0=n n)
```

But we can pattern match on **n**, so that we know what **n** "looks like".

More interesting cases: Induction

```
n+0=n : (n : Nat) -> (n + 0) =P n
n+0=n zero = refl
n+0=n (suc n) = cong suc (n+0=n n)
```

But we can pattern match on **n**, so that we know what **n** "looks like".

If n = 0, then 0 = 0.

More interesting cases: Induction

```
n+0=n : (n : Nat) -> (n + 0) =P n
n+0=n zero = refl
n+0=n (suc n) = cong suc (n+0=n n)
```

But we can pattern match on **n**, so that we know what **n** "looks like".

If n = 0, then 0 = 0.

If n = 1 + k and k + 0 = k then (1 + k) + 0 = (1 + k).

More interesting cases: Induction

```
n+0=n : (n : Nat) -> (n + 0) =P n
n+0=n zero = refl
n+0=n (suc n) = cong suc (n+0=n n)
```

But we can pattern match on **n**, so that we know what **n** "looks like".

If n = 0, then 0 = 0.

If n = 1 + k and k + 0 = k then (1 + k) + 0 = (1 + k).

What does this argument sound like?

Understanding Check

Write a function

```
sym : {A : Set} -> (x y : A) -> x =P y -> y =P x
```

What does this function express?

The Lambda Calculus: Motivation

What is the lambda calculus?

$f =$ `(add 1 to a single argument x)`

vs.

$$f = \{(0,1), (1,2), (2,3), (3,4), \ldots\}$$

What is the lambda calculus?

$f =$ `(add 1 to a single argument x)`

`vs.`

$$f = \{(0,1), (1,2), (2,3), (3,4), \ldots\}$$

`The lambda calculus is a framework for reasoning about functions as procedures.`

What is the lambda calculus?

$f =$ `(add 1 to a single argument x)`

vs.

$$f = \{(0,1), (1,2), (2,3), (3,4), \ldots\}$$

The lambda calculus is a framework for reasoning about functions as procedures.

This is as opposed to functions as sets.

What is the lambda calculus?

$f =$ `(add 1 to a single argument x)`

vs.

$$f = \{(0,1), (1,2), (2,3), (3,4), \ldots\}$$

The lambda calculus is a framework for reasoning about functions as procedures.

This is as opposed to functions as sets.

(I like to think about it as a "theory of substitution", as we will see)

Some History

1932 Created by A. Church in an attempt to define a foundations of mathematics

1935 Church's system was proven inconsistent by S. Kleene and J. Rosser

1936 Church distills functional part into what is known as the λ–calculus

Anonymous Functions

```
lambda x: x

lambda x: lambda y: x

lambda f: lambda x: f(x)

lambda x: x(x)

(lambda x: x(x))(lambda x: x(x))
```

Informal definition. The (closed) **lambda calculus** is given by the collection of Python programs you could write with only single argument anonymous functions and variables.

Syntax

High Level

High Level

```
If the lambda calculus is about functions, then
we need to be able to
```

High Level

If the lambda calculus is about functions, then we need to be able to

- Refer to the arguments of functions

High Level

If the lambda calculus is about functions, then we need to be able to

- Refer to the arguments of functions
- Apply one function to a value

High Level

If the lambda calculus is about functions, then we need to be able to

- Refer to the arguments of functions
- Apply one function to a value
- Build functions out of old ones (or values).

Lambda Terms

`(Fix a set of variables.)`

Lambda Terms

(Fix a set of variables.)

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

Lambda Terms

(Fix a set of variables.)

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

- Every variable x is a lambda term.

variables

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

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)
- If M is a lambda term, then so is $(\lambda x . M)$ for any variable x

variables

application

abstraction

Examples

$$x, y$$

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

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

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

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

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

Examples

$$x, y$$

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

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

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

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

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

We will use **meta-variables** to refer to specific lambda terms (e.g., K).

Examples

$$x, y$$

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

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

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

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

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

We will use **meta-variables** to refer to specific lambda terms (e.g., K).

But K is not a part of the syntax.

What about values?

What about values?

`What do we apply lambda terms to?`

What about values?

`What do we apply lambda terms to?`

`Other lambda terms (i.e., other functions)`

What about values?

`What do we apply lambda terms to?`

`Other lambda terms (i.e., other functions)`

`How is this useful?`

What about values?

`What do we apply lambda terms to?`

`Other lambda terms (i.e., other functions)`

`How is this useful?`

`We can encode values as lambda terms.`

In Agda

```
Var : Set
Var = Nat

data LTerm : Set where
  var : Var -> LTerm
  app : LTerm -> LTerm -> LTerm
  abs : Var -> LTerm -> LTerm
```

Because this is an inductive definition, we can readily define it in Agda.

Examples (In Agda)

```
x : Var
x = 0

y : Var
y = 1

i : LTerm
i = abs x (var x)

k : LTerm
k = abs x (abs y (var x))

a : LTerm
a = abs x (abs y (app (var x) (var y)))

omega : LTerm
omega = abs x (app (var x) (var x))

omom : LTerm
omom = app omega omega
```

Syntactic Conventions

- Application has higher precedence than abstraction, so $\lambda x\,.\,xy = \lambda x\,.\,(xy)$
- Application associates to the left, so $MNP = (MN)P$
- Abstraction "associates to the right", so $\lambda x\,.\,\lambda y\,.\,x = \lambda x\,.\,(\lambda y\,.\,x)$

Examples (Again)

$$x, y$$

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

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

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

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

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

Examples (In Agda) (Again)

```
infixr 10 _$_
infixr 5 lam_=>_

data LTerm : Set where
  [_] : Var -> LTerm
  _$_ : LTerm -> LTerm -> LTerm
  lam_=>_ : Var -> LTerm -> LTerm
```

```
x : Var
x = 0

y : Var
y = 1

i : LTerm
i = lam x => [ x ]

k : LTerm
k = lam x => lam y => [ x ]

a : LTerm
a = lam x => lam y => [ x ] $ [ y ]

omega : LTerm
omega = lam x => [ x ] $ [ x ]

omom : LTerm
omom = omega $ omega
```

Semantics

Meaning

Meaning

Question. *What is the meaning of a lambda term?*

Meaning

Question. *What is the meaning of a lambda term?*

If a lambda term represents a function then maybe it should be a function in the "traditional" sense.

Meaning

Question. *What is the meaning of a lambda term?*

If a lambda term represents a function then maybe it should be a function in the "traditional" sense.

$$\texttt{(add 1 to x)} \implies \{(0,1), (1,2), (2,3), \ldots\}$$

Meaning

Question. *What is the meaning of a lambda term?*

If a lambda term represents a function then maybe it should be a function in the "traditional" sense.

$$\texttt{(add 1 to x)} \Longrightarrow \{(0,1),(1,2),(2,3),\ldots\}$$

This is tricky.

Denotational Semantics

Denotational Semantics

Definition (Informal). The **denotation** of a lambda term (or any program) is the function it represents.

Denotational Semantics

Definition (Informal). The **denotation** of a lambda term (or any program) is the function it represents.

The idea. We map terms into a mathematical space functions.

Denotational Semantics

Definition (Informal). The **denotation** of a lambda term (or any program) is the function it represents.

The idea. We map terms into a mathematical space functions.

Question. *What would the set function for $\lambda x.x$ look like?*

Meaning

Meaning

Question. *What is the meaning of a lambda term?*

Meaning

Question. *What is the meaning of a lambda term?*

If a lambda terms represents a program, maybe it should be the value of the program after running (i.e., *evaluating*) it.

Meaning

Question. *What is the meaning of a lambda term?*

If a lambda terms represents a program, maybe it should be the value of the program after running (i.e., *evaluating*) it.

$$\texttt{(add 1 to x)} \Longrightarrow \texttt{(add 1 to x)}$$
$$\texttt{(add 1 to x) 12} \Longrightarrow \texttt{13}$$

Operational Semantics

Operational Semantics

`Operational Semantics refers to the rules for evaluating a lambda term.`

Operational Semantics

Operational Semantics refers to the rules for evaluating a lambda term.

Question. *What should* $(\lambda x\,.\,x)(\lambda x\,.\,x)y$ *evaluate to?*

Operational Semantics

Operational Semantics refers to the rules for evaluating a lambda term.

Question. *What should* $(\lambda x\,.\,x)(\lambda x\,.\,x)y$ *evaluate to?*

This is also tricky, but a bit more manageable.

Free and Bound Variables

$$(\lambda x\,.\,y\,\overset{\text{bound}}{\boxed{x}})\,\overset{\text{free}}{\boxed{x}}$$

A variable x is **bound** if it appears in the body of an abstraction over x.

Otherwise it is **free**.

Substitution

Substitution

Definition. Substitution of N for a free variable x in M, written $M[N/x]$ is defined recursively on M.

Substitution

Definition. Substitution of N for a free variable x in M, written $M[N/x]$ is defined recursively on M.

- $y[N/x] = \begin{cases} N & y = x \\ y & \texttt{otherwise} \end{cases}$

Substitution

Definition. Substitution of N for a free variable x in M, written $M[N/x]$ is defined recursively on M.

- $y[N/x] = \begin{cases} N & y = x \\ y & \texttt{otherwise} \end{cases}$

- $(M_1 M_2)[N/x] = (M_1[N/x])(M_2[N/x])$

Substitution

Definition. Substitution of N for a free variable x in M, written $M[N/x]$ is defined recursively on M.

- $y[N/x] = \begin{cases} N & y = x \\ y & \text{otherwise} \end{cases}$
- $(M_1 M_2)[N/x] = (M_1[N/x])(M_2[N/x])$
- $(\lambda y \,.\, M)[N/x] = \begin{cases} \lambda y \,.\, M & y = x \\ \lambda y \,.\, M[N/x] & \text{otherwise} \end{cases}$

Substitution

Definition. Substitution of N for a free variable x in M, written $M[N/x]$ is defined recursively on M.

- $y[N/x] = \begin{cases} N & y = x \\ y & \text{otherwise} \end{cases}$

- $(M_1 M_2)[N/x] = (M_1[N/x])(M_2[N/x])$

- $(\lambda y\,.\,M)[N/x] = \begin{cases} \lambda y\,.\,M & y = x \\ \lambda y\,.\,M[N/x] & \text{otherwise} \end{cases}$

this is not quite right.

Alpha-Equivalence

$$\lambda x\,.\,xz =_\alpha \lambda y\,.\,yz \neq_\alpha \lambda z\,.\,zz$$

Alpha-Equivalence

$$\lambda x\,.\,xz =_\alpha \lambda y\,.\,yz \neq_\alpha \lambda z\,.\,zz$$

Definition (Informal). Lambda terms M and N are α-equivalent if they are the same up to *valid* renaming bound variables.

Alpha-Equivalence

$$\lambda x \,.\, xz =_\alpha \lambda y \,.\, yz \neq_\alpha \lambda z \,.\, zz$$

Definition (Informal). Lambda terms M and N are α-equivalent if they are the same up to *valid* renaming bound variables.

We cannot rename bound variables to existing free variables.

Alpha-Equivalence

$$\lambda x\,.\,xz =_\alpha \lambda y\,.\,yz \neq_\alpha \lambda z\,.\,zz$$

Definition (Informal). Lambda terms M and N are α–equivalent if they are the same up to *valid* renaming bound variables.

We cannot rename bound variables to existing free variables.

We will consider terms up to α–equivalence.

Captured Variables

$$\lambda x\,.\,y \neq_\alpha \lambda x\,.\,x = \lambda x\,.\,y[x/y]$$

Captured Variables

$$\lambda x\,.\,y \neq_\alpha \lambda x\,.\,x = \lambda x\,.\,y[x/y]$$

We would like that substitution *preserves* α
–equivalence.

Captured Variables

$$\lambda x\,.\,y \neq_\alpha \lambda x\,.\,x = \lambda x\,.\,y[x/y]$$

We would like that substitution *preserves* α
−equivalence.

Our current definition does not do this.

Substitution (Again)

Definition. Substitution of N for x in M, written $M[N/x]$ is defined recursively on M.

- $y[N/x] = \begin{cases} N & y = x \\ y & \text{otherwise} \end{cases}$
- $(M_1 M_2)[N/x] = (M_1[N/x])(M_2[N/x])$
- $(\lambda y \,.\, M)[N/x] = \begin{cases} \lambda y \,.\, M & y = x \\ \lambda y \,.\, M[N[z/y]/x] & \text{otherwise} \end{cases}$

where z does not appear free in M or N

Substitution (Again)

Definition. Substitution of N for x in M, written $M[N/x]$ is defined recursively on M.

- $y[N/x] = \begin{cases} N & y = x \\ y & \text{otherwise} \end{cases}$
- $(M_1M_2)[N/x] = (M_1[N/x])(M_2[N/x])$
- $(\lambda y\,.\,M)[N/x] = \begin{cases} \lambda y\,.\,M & y = x \\ \lambda y\,.\,M[N[z/y]/x] & \text{otherwise} \end{cases}$

where z does not appear free in M or N

Beta Reduction

Beta Reduction

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

Beta Reduction

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

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

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 $\lambda x\,.\,M \to_\beta \lambda x\,.\,M'$

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.

Evaluation

Definition. A β–**normal form** is a term M such that there is no N where $M \to_\beta N$.

Normal forms cannot be further reduced, they are like the values of a computation.

Evaluation is the processing of trying to reduce a term to normal form.

Evaluation

Definition. A β–**normal form** is a term M such that there is no N where $M \rightarrow_\beta N$.

Normal forms cannot be further reduced, they are like the values of a computation.

Evaluation is the processing of trying to reduce a term to normal form.

Questions

Do all terms have normal forms?

If a term has a normal form, is there always a way to find it?

If a term has a normal form, is it unique?