About Me

My name is Nathan (he/him). Outside of teaching, I'm interested generally in type theory, logic, and computation.

I'm primarily interested in teaching fundamentals courses, theory courses and PL courses, but I am also interested in getting more experience by teaching courses outside of my field. I also hope to be able to teach some advanced courses for undergraduates interested in logic.

I'd like to stay connected to the research community (this is, of course, why I find NE such an exciting opportunity) and ideally use teaching as a tool to further my own research and create spaces for new young researchers.

Above all, I'm interested in becoming a part of a community. I'd like to participate in all that I can to help strengthen the department with regards to representation and accessibility.

About Me

My name is Nathan (he/him). Outside of teaching, I'm interested generally in type theory, logic, and computation.

I'm primarily interested in teaching fundamentals courses, theory courses and PL courses, but I am also interested in getting more experience by teaching courses outside of my field. I also hope to be able to teach some advanced courses for undergraduates interested in logic.

I'd like to stay connected to the research community (this is, of course, why I find NE such an exciting opportunity) and ideally use teaching as a tool to further my own research and create spaces for new young researchers.

Above all, I'm interested in becoming a part of a community. I'd like to participate in all that I can to help strengthen the department with regards to representation and accessibility.

About Me

My name is Nathan (he/him). Outside of teaching, I'm interested generally in type theory, logic, and computation.

I'm primarily interested in teaching fundamentals courses, theory courses and PL courses, but I am also interested in getting more experience by teaching courses outside of my field. I also hope to be able to teach some advanced courses for undergraduates interested in logic.

I'd like to stay connected to the research community (this is, of course, why I find NE such an exciting opportunity) and ideally use teaching as a tool to further my own research and create spaces for new young researchers.

Above all, I'm interested in becoming a part of a community. I'd like to participate in all that I can to help strengthen the department with regards to representation and accessibility.

About Me

My name is Nathan (he/him). Outside of teaching, I'm interested generally in type theory, logic, and computation.

I'm primarily interested in teaching fundamentals courses, theory courses and PL courses, but I am also interested in getting more experience by teaching courses outside of my field. I also hope to be able to teach some advanced courses for undergraduates interested in logic.

I'd like to stay connected to the research community (this is, of course, why I find NE such an exciting opportunity) and ideally use teaching as a tool to further my own research and create spaces for new young researchers.

Above all, I'm interested in becoming a part of a community. I'd like to participate in all that I can to help strengthen the department with regards to representation and accessibility.

About this Lecture

This is a rough draft of a lecture for a course I'd like to design on the λ-calculus and type theory. I want the course fall somewhere between theoretical and practical, with some work on dependent type theory and proof engineering, but also on the meta-theory no one seems to be covering in depth anymore.

It's based heavily on *Type Theory and Formal Proof* by Rob Nederpelt and Herman Geuvers, as well as *CSMC 22500: Type Theory*, a course taught by Stuart Kurtz at UChicago.

I expect students to be upper-class CS undergraduates with some experience in math, or math undergraduates with some experience in CS, but I could imagine it being useful even as a review for graduate students in logic who had to learn the λ-calculus on their own (like so many of us).

About this Lecture

This is a rough draft of a lecture for a course I'd like to design on the λ-calculus and type theory. I want the course fall somewhere between theoretical and practical, with some work on dependent type theory and proof engineering, but also on the meta-theory no one seems to be covering in depth anymore.

It's based heavily on *Type Theory and Formal Proof* by Rob Nederpelt and Herman Geuvers, as well as *CSMC 22500: Type Theory*, a course taught by Stuart Kurtz at UChicago.

I expect students to be upper-class CS undergraduates with some experience in math, or math undergraduates with some experience in CS, but I could imagine it being useful even as a review for graduate students in logic who had to learn the λ-calculus on their own (like so many of us).

About this Lecture

This is a rough draft of a lecture for a course I'd like to design on the λ-calculus and type theory. I want the course fall somewhere between theoretical and practical, with some work on dependent type theory and proof engineering, but also on the meta-theory no one seems to be covering in depth anymore.

It's based heavily on *Type Theory and Formal Proof* by Rob Nederpelt and Herman Geuvers, as well as *CSMC 22500: Type Theory*, a course taught by Stuart Kurtz at UChicago.

I expect students to be upper-class CS undergraduates with some experience in math, or math undergraduates with some experience in CS, but I could imagine it being useful even as a review for graduate students in logic who had to learn the λ-calculus on their own (like so many of us).

Lecture 2: An Introduction to the Lambda Calculus

CS@@@@: Type Theory
January 30, 2023

Northeastern Demo Lecture

Objectives

- Motivate the lambda calculus.
  - What is a function, really?
  - The lambda calculus may be more familiar than you think.
- Define the lambda calculus.
  - Define λ-terms.
  - Define notion of computing/reducing λ-terms.
- Try to understand as best we can the subtleties of these definitions.
  - Figure out what the heck is going on with substitution and bound variables.
- Begin to understand the connection between the lambda calculus and computation.
- Get as much practice as we can with all these notions.

Objectives

- Motivate the lambda calculus.
  - What is a function, really?
  - The lambda calculus may be more familiar than you think.
- Define the lambda calculus.
  - Define λ-terms.
  - Define notion of computing/reducing λ-terms.
- Try to understand as best we can the subtleties of these definitions.
  - Figure out what the heck is going on with substitution and bound variables.
- Begin to understand the connection between the lambda calculus and computation.
- Get as much practice as we can with all these notions.

Objectives

- Motivate the lambda calculus.
  - What is a function, really?
  - The lambda calculus may be more familiar than you think.
- Define the lambda calculus.
  - Define λ-terms.
  - Define notion of computing/reducing λ-terms.
- Try to understand as best we can the subtleties of these definitions.
  - Figure out what the heck is going on with substitution and bound variables.
- Begin to understand the connection between the lambda calculus and computation.
- Get as much practice as we can with all these notions.

Objectives

- Motivate the lambda calculus.
  - What is a function, really?
  - The lambda calculus may be more familiar than you think.
- Define the lambda calculus.
  - Define λ-terms.
  - Define notion of computing/reducing λ-terms.
- Try to understand as best we can the subtleties of these definitions.
  - Figure out what the heck is going on with substitution and bound variables.
- Begin to understand the connection between the lambda calculus and computation.
- Get as much practice as we can with all these notions.

Objectives

- Motivate the lambda calculus.
  - What is a function, really?
  - The lambda calculus may be more familiar than you think.
- Define the lambda calculus.
  - Define λ-terms.
  - Define notion of computing/reducing λ-terms.
- Try to understand as best we can the subtleties of these definitions.
  - Figure out what the heck is going on with substitution and bound variables.
- Begin to understand the connection between the lambda calculus and computation.
- Get as much practice as we can with all these notions.

Keywords

- λ-Terms
- α-Equivalence and Renaming
- Variables and Substitution
- β-Reduction
- Compatibility, Reduction Paths, and Equivalences
- Normal Forms

What is a lambda? (A bit of history)

1932 Alonzo Church defines a system for formalizing mathematics which has the λ-calculus implicitly at its core.

1935 Stephen Cole Kleene and J. Barkley Rosser demonstrate that this system is inconsistent, so it can't be used as a foundation for mathematics.

1936 Church distills his system to its functional part, which becomes the λ-calculus. The 'λ' is used to denote abstraction, i.e., the conversion of an expression with abstract variables into a function on one of it's variables.

**** (A lot happens...)

2006 Felice Cardone and J. Roger Hindley claim that the 'λ' was chosen based on type setting.

2016 Dana Scott claims that the choice was arbitrary.

2022 Henk Barendregt claims Church developed the λ-calculus to show that a question his doctoral advisor asked him to solve was impossible.

What is a lambda? (A bit of history)

1932 Alonzo Church defines a system for formalizing mathematics which has the λ-calculus implicitly at its core.

1935 Stephen Cole Kleene and J. Barkley Rosser demonstrate that this system is inconsistent, so it can't be used as a foundation for mathematics.

1936 Church distills his system to its functional part, which becomes the λ-calculus. The 'λ' is used to denote abstraction, i.e., the conversion of an expression with abstract variables into a function on one of it's variables.

**** (A lot happens...)

2006 Felice Cardone and J. Roger Hindley claim that the 'λ' was chosen based on type setting.

2016 Dana Scott claims that the choice was arbitrary.

2022 Henk Barendregt claims Church developed the λ-calculus to show that a question his doctoral advisor asked him to solve was impossible.

What is a lambda? (A bit of history)

1932 Alonzo Church defines a system for formalizing mathematics which has the λ-calculus implicitly at its core.

1935 Stephen Cole Kleene and J. Barkley Rosser demonstrate that this system is inconsistent, so it can't be used as a foundation for mathematics.

1936 Church distills his system to its functional part, which becomes the λ-calculus. The 'λ' is used to denote abstraction, i.e., the conversion of an expression with abstract variables into a function on one of it's variables.

**** (A lot happens...)

2006 Felice Cardone and J. Roger Hindley claim that the 'λ' was chosen based on type setting.

2016 Dana Scott claims that the choice was arbitrary.

2022 Henk Barendregt claims Church developed the λ-calculus to show that a question his doctoral advisor asked him to solve was impossible.

What is a lambda? (A bit of history)

1932 Alonzo Church defines a system for formalizing mathematics which has the λ-calculus implicitly at its core.

1935 Stephen Cole Kleene and J. Barkley Rosser demonstrate that this system is inconsistent, so it can't be used as a foundation for mathematics.

1936 Church distills his system to its functional part, which becomes the λ-calculus. The 'λ' is used to denote abstraction, i.e., the conversion of an expression with abstract variables into a function on one of it's variables.

**** (A lot happens...)

2006 Felice Cardone and J. Roger Hindley claim that the 'λ' was chosen based on type setting.

2016 Dana Scott claims that the choice was arbitrary.

2022 Henk Barendregt claims Church developed the λ-calculus to show that a question his doctoral advisor asked him to solve was impossible.

What is a lambda? (A bit of history)

1932 Alonzo Church defines a system for formalizing mathematics which has the λ-calculus implicitly at its core.

1935 Stephen Cole Kleene and J. Barkley Rosser demonstrate that this system is inconsistent, so it can't be used as a foundation for mathematics.

1936 Church distills his system to its functional part, which becomes the λ-calculus. The 'λ' is used to denote abstraction, i.e., the conversion of an expression with abstract variables into a function on one of it's variables.

**** (A lot happens...)

2006 Felice Cardone and J. Roger Hindley claim that the 'λ' was chosen based on type setting.

2016 Dana Scott claims that the choice was arbitrary.

2022 Henk Barendregt claims Church developed the λ-calculus to show that a question his doctoral advisor asked him to solve was impossible.

What is a lambda? (A bit of history)

1932 Alonzo Church defines a system for formalizing mathematics which has the λ-calculus implicitly at its core.

1935 Stephen Cole Kleene and J. Barkley Rosser demonstrate that this system is inconsistent, so it can't be used as a foundation for mathematics.

1936 Church distills his system to its functional part, which becomes the λ-calculus. The 'λ' is used to denote abstraction, i.e., the conversion of an expression with abstract variables into a function on one of it's variables.

**** (A lot happens...)

2006 Felice Cardone and J. Roger Hindley claim that the 'λ' was chosen based on type setting.

2016 Dana Scott claims that the choice was arbitrary.

2022 Henk Barendregt claims Church developed the λ-calculus to show that a question his doctoral advisor asked him to solve was impossible.

What is a lambda? (A bit of history)

1932 Alonzo Church defines a system for formalizing mathematics which has the λ-calculus implicitly at its core.

1935 Stephen Cole Kleene and J. Barkley Rosser demonstrate that this system is inconsistent, so it can't be used as a foundation for mathematics.

1936 Church distills his system to its functional part, which becomes the λ-calculus. The 'λ' is used to denote abstraction, i.e., the conversion of an expression with abstract variables into a function on one of it's variables.

**** (A lot happens...)

2006 Felice Cardone and J. Roger Hindley claim that the 'λ' was chosen based on type setting.

2016 Dana Scott claims that the choice was arbitrary.

2022 Henk Barendregt claims Church developed the λ-calculus to show that a question his doctoral advisor asked him to solve was impossible.

What is a calculus?

What we usually think of as calculus is misnomer, it was originally called the *infinitesimal/integral/differential calculus*.

Definition (Calculus, Informally)

A **calculus** is a system for performing calculations. It's the rule which govern how certain things behave.

The λ-calculus is a set of rules which governs how functions behave and how their values are calculated.

What is a calculus?

What we usually think of as calculus is misnomer, it was originally called the *infinitesimal/integral/differential calculus*.

Definition (Calculus, Informally)

A **calculus** is a system for performing calculations. It's the rule which govern how certain things behave.

The λ-calculus is a set of rules which governs how functions behave and how their values are calculated.

What is a calculus?

What we usually think of as calculus is misnomer, it was originally called the *infinitesimal/integral/differential calculus*.

Definition (Calculus, Informally)

A **calculus** is a system for performing calculations. It's the rule which govern how certain things behave.

The λ-calculus is a set of rules which governs how functions behave and how their values are calculated.

But wait, what is a function?

Seems obvious right? A mathematician might say...

Definition (Function as a Graph)

A **function** f from the set X to the set Y is a set of pairs $\{(x, y) : x \in X, y \in Y\}$ such that for all $x \in X$ there is a unique y such that $(x, y) \in f$.

But this feels unsatisfying for a programmer, where a function is really a procedure or description, e.g., 'take x and add 1 to it'.

Question. What about the identity function 'take x and return it'? What does that look like as a set function?

The λ-calculus is a set of rules which governs how functions presented as rules or procedures behave and how their values are calculated.

But wait, what is a function?

Seems obvious right? A mathematician might say...

Definition (Function as a Graph)

A **function** f from the set X to the set Y is a set of pairs $\{(x, y) : x \in X, y \in Y\}$ such that for all $x \in X$ there is a unique y such that $(x, y) \in f$.

But this feels unsatisfying for a programmer, where a function is really a procedure or description, e.g., 'take x and add 1 to it'.

Question. What about the identity function 'take x and return it'? What does that look like as a set function?

The λ-calculus is a set of rules which governs how functions presented as rules or procedures behave and how their values are calculated.

But wait, what is a function?

Seems obvious right? A mathematician might say...

Definition (Function as a Graph)

A **function** f from the set X to the set Y is a set of pairs $\{(x, y) : x \in X, y \in Y\}$ such that for all $x \in X$ there is a unique y such that $(x, y) \in f$.

But this feels unsatisfying for a programmer, where a function is really a procedure or description, e.g., 'take x and add 1 to it'.

Question. What about the identity function 'take x and return it'? What does that look like as a set function?

The λ-calculus is a set of rules which governs how functions presented as rules or procedures behave and how their values are calculated.

But wait, what is a function?

Seems obvious right? A mathematician might say...

Definition (Function as a Graph)

A **function** f from the set X to the set Y is a set of pairs $\{(x, y) : x \in X, y \in Y\}$ such that for all $x \in X$ there is a unique y such that $(x, y) \in f$.

But this feels unsatisfying for a programmer, where a function is really a procedure or description, e.g., 'take x and add 1 to it'.

Question. What about the identity function 'take x and return it'? What does that look like as a set function?

The λ-calculus is a set of rules which governs how functions presented as rules or procedures behave and how their values are calculated.

The λ-Calculus (A First Approximation)

An observation. The λ-calculus can be thought of as the fragment of your favorite programming language with only variables and anonymous functions. It's not an accident that anonymous functions in `Python` and `Racket` are called `lambdas`.

In `Python`:

```
1 lambda x : x
2 lambda x : (lambda y : x)
3 lambda f : (lambda g : (lambda x : f(g(x))))
4 lambda x : x(x)
```

Important

Everything in the lambda calculus is a function!

The λ-Calculus (A First Approximation)

An observation. The λ-calculus can be thought of as the fragment of your favorite programming language with only variables and anonymous functions. It's not an accident that anonymous functions in `Python` and `Racket` are called `lambdas`.

In `Python`:

```
1 lambda x : x
2 lambda x : (lambda y : x)
3 lambda f : (lambda g : (lambda x : f(g(x))))
4 lambda x : x(x)
```

Important

Everything in the lambda calculus is a function!

The λ-Calculus (A First Approximation)

An observation. The λ-calculus can be thought of as the fragment of your favorite programming language with only variables and anonymous functions. It's not an accident that anonymous functions in `Python` and `Racket` are called `lambdas`.

In `Python`:

```
1 lambda x : x
2 lambda x : (lambda y : x)
3 lambda f : (lambda g : (lambda x : f(g(x))))
4 lambda x : x(x)
```

Important

Everything in the lambda calculus is a function!

λ-Terms

Definition (λ-terms)

Fix a countable collection of variables $V = \{v_1, v_2, \dots\}$. The set Λ is define inductively as follows.

- $V \subset \Lambda$ *(all* **variables** *are λ-terms)*.
- If $v_i \in V$ and $M \in \Lambda$, then $(\lambda v_i.M) \in \Lambda$ *(any terms can be* **abstracted***)*.
- If $M \in \Lambda$ and $N \in \Lambda$, then $(MN) \in \Lambda$ *(any term can be* **applied** *to any other term)*.

This definition is **inductive**, which means this is the **only** way to create λ-terms.

Abstraction takes an expression with a variable and turns it into a function on that variable. (Think higher order functions!)

'x with a hat' $\Longrightarrow$ the function which puts a hat on its argument

λ-Terms

Definition (λ-terms)

Fix a countable collection of variables $V = \{v_1, v_2, \dots\}$. The set Λ is define inductively as follows.

- $V \subset \Lambda$ *(all* **variables** *are λ-terms)*.
- If $v_i \in V$ and $M \in \Lambda$, then $(\lambda v_i.M) \in \Lambda$ *(any terms can be* **abstracted***)*.
- If $M \in \Lambda$ and $N \in \Lambda$, then $(MN) \in \Lambda$ *(any term can be* **applied** *to any other term)*.

This definition is **inductive**, which means this is the **only** way to create λ-terms.

Abstraction takes an expression with a variable and turns it into a function on that variable. (Think higher order functions!)

'x with a hat' $\Longrightarrow$ the function which puts a hat on its argument

λ-Terms

Definition (λ-terms)

Fix a countable collection of variables $V = \{v_1, v_2, \dots\}$. The set Λ is define inductively as follows.

- $V \subset \Lambda$ *(all* **variables** *are λ-terms)*.
- If $v_i \in V$ and $M \in \Lambda$, then $(\lambda v_i.M) \in \Lambda$ *(any terms can be* **abstracted***)*.
- If $M \in \Lambda$ and $N \in \Lambda$, then $(MN) \in \Lambda$ *(any term can be* **applied** *to any other term)*.

This definition is **inductive**, which means this is the **only** way to create λ-terms.

Abstraction takes an expression with a variable and turns it into a function on that variable. (Think higher order functions!)

> 'x with a hat' $\Longrightarrow$ the function which puts a hat on its argument

Examples of λ-Terms

$(\lambda v_1.v_1)$	(identity)
$(\lambda v_1.(\lambda v_2.v_1))$	(K)
$(\lambda v_1.(\lambda v_2.(\lambda v_3.(v_1(v_2 v_3)))))$	(composition)
$(\lambda v_1.(v_1 v_1))$	(Δ)

Important

These are syntactic objects! They don't mean anything until we've given them meaning. In particular, the parentheses are part of the syntax.

Examples of λ-Terms

$(\lambda v_1.v_1)$	(identity)
$(\lambda v_1.(\lambda v_2.v_1))$	(K)
$(\lambda v_1.(\lambda v_2.(\lambda v_3.(v_1(v_2 v_3)))))$	(composition)
$(\lambda v_1.(v_1 v_1))$	(Δ)

Important

These are syntactic objects! They don't mean anything until we've given them meaning. In particular, the parentheses are part of the syntax.

Meta-Syntactic Conventions

- Application associates to the left (just like most programming languages), e.g., MNP is short for $(MN)P$.
- We'll use any variable name we want (within reason), e.g., $(\lambda x.x)$.
- We'll drop parentheses when there is no ambiguity, e.g., $\lambda x.\lambda y.x$.
- We'll write multiple arguments together, e.g., $\lambda xy.x$ is short for $\lambda x.\lambda y.x$.

$\lambda x.x$	(identity)
$\lambda xy.x$	(K)
$\lambda fgx.f(gx)$	(composition)
$\lambda x.xx$	(Δ)

Practice Problems

1. Is $xxxxxxx$ a valid term?
2. Is $\lambda x.x$ the same term as $\lambda y.y$? (this is a trick question!)
3. Write down the term $\lambda xyz.x(yx)z$ without our meta-syntactic conventions.
4. Describe the function that term $\lambda xf.fx$ represents in your favorite human-spoken language.

So far...

We've seen the syntax of the λ-calculus. We've hinted a connection between λ-terms and programs. So: **how do we run them?**

The idea. **Reduction as computation.** For the λ-calculus, this means *applying* our functions, but this concept comes up in general reduction systems.

Example (Computing algebraic expressions)

$$\begin{aligned}(2 * 3) + ((4 * 5) - 8) &\longrightarrow 6 + ((4 * 5) - 8) \\ &\longrightarrow 6 + (20 - 8) \\ &\longrightarrow 6 + 12 \\ &\longrightarrow 18\end{aligned}$$

So far...

We've seen the syntax of the λ-calculus. We've hinted a connection between λ-terms and programs. So: **how do we run them?**

The idea. **Reduction as computation.** For the λ-calculus, this means *applying* our functions, but this concept comes up in general reduction systems.

Example (Computing algebraic expressions)

$$\begin{aligned}(2 * 3) + ((4 * 5) - 8) &\longrightarrow 6 + ((4 * 5) - 8) \\ &\longrightarrow 6 + (20 - 8) \\ &\longrightarrow 6 + 12 \\ &\longrightarrow 18\end{aligned}$$

So far...

We've seen the syntax of the λ-calculus. We've hinted a connection between λ-terms and programs. So: **how do we run them?**

The idea. **Reduction as computation.** For the λ-calculus, this means *applying* our functions, but this concept comes up in general reduction systems.

Example (Computing algebraic expressions)

$$\begin{aligned}(2 * 3) + ((4 * 5) - 8) &\longrightarrow 6 + ((4 * 5) - 8) \\ &\longrightarrow 6 + (20 - 8) \\ &\longrightarrow 6 + 12 \\ &\longrightarrow 18\end{aligned}$$

β-Reduction

The idea. Applying a function to a term should result in the expression in the body of the function with the variable substituted at its argument.

$$(\lambda x.\text{`}x \text{ with a hat'})\text{cat} \longrightarrow_\beta \text{`cat with a hat'}$$
$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

where $M[N/x]$ is the result of substituting N for x in M. Furthermore, we should be able to do this anywhere in the term.

Example

$$\begin{aligned}(((\lambda fgx.f(gx))M)N)P &\to_\beta ((\lambda gx.M(gx))N)P \\ &\to_\beta (\lambda x.M(Nx))P \\ &\to_\beta M(NP)\end{aligned}$$

β-Reduction

The idea. Applying a function to a term should result in the expression in the body of the function with the variable substituted at its argument.

$$(\lambda x.\text{`}x \text{ with a hat'})\text{cat} \longrightarrow_\beta \text{`cat with a hat'}$$

$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

where $M[N/x]$ is the result of substituting N for x in M. Furthermore, we should be able to do this anywhere in the term.

Example

$$\begin{aligned}(((\lambda fgx.f(gx))M)N)P &\to_\beta ((\lambda gx.M(gx))N)P \\ &\to_\beta (\lambda x.M(Nx))P \\ &\to_\beta M(NP)\end{aligned}$$

β-Reduction

The idea. Applying a function to a term should result in the expression in the body of the function with the variable substituted at its argument.

$$(\lambda x.\text{`}x \text{ with a hat'})\text{cat} \longrightarrow_\beta \text{`cat with a hat'}$$
$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

where $M[N/x]$ is the result of substituting N for x in M. Furthermore, we should be able to do this anywhere in the term.

Example

$$\begin{aligned}(((\lambda fgx.f(gx))M)N)P &\to_\beta ((\lambda gx.M(gx))N)P \\ &\to_\beta (\lambda x.M(Nx))P \\ &\to_\beta M(NP)\end{aligned}$$

β-Reduction

The idea. Applying a function to a term should result in the expression in the body of the function with the variable substituted at its argument.

$$(\lambda x.\text{`}x \text{ with a hat'})\text{cat} \longrightarrow_\beta \text{`cat with a hat'}$$
$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

where $M[N/x]$ is the result of substituting N for x in M. Furthermore, we should be able to do this anywhere in the term.

Example

$$\begin{aligned}(((\lambda fgx.f(gx))M)N)P &\to_\beta ((\lambda gx.M(gx))N)P \\ &\to_\beta (\lambda x.M(Nx))P \\ &\to_\beta M(NP)\end{aligned}$$

Substitution

Definition (Substitution, Informally)

The **substitution of** x **by** N **in** M, written $M[N/x]$, is the result of replacing all instances of x in M by N.

Example

$$(\lambda z.x(yz)y)[\lambda q.q/y] = \lambda z.x((\lambda q.q)z)(\lambda q.q))$$

First Problem. What about $(\lambda y.y)[\lambda q.q/y]$?

Second Problem. What about $(\lambda y.z)[y/z]]$?

Substitution

Definition (Substitution, Informally)

The **substitution of** x **by** N **in** M, written $M[N/x]$, is the result of replacing all instances of x in M by N.

Example

$$(\lambda z.x(yz)y)[\lambda q.q/y] = \lambda z.x((\lambda q.q)z)(\lambda q.q))$$

First Problem. What about $(\lambda y.y)[\lambda q.q/y]$?

Second Problem. What about $(\lambda y.z)[y/z]]$?

Substitution

Definition (Substitution, Informally)

The **substitution of** x **by** N **in** M, written $M[N/x]$, is the result of replacing all instances of x in M by N.

Example

$$(\lambda z.x(yz)y)[\lambda q.q/y] = \lambda z.x((\lambda q.q)z)(\lambda q.q))$$

First Problem. What about $(\lambda y.y)[\lambda q.q/y]$?

Second Problem. What about $(\lambda y.z)[y/z]]$?

Variable Scope

Definition (Free and Bound Variables)

The set $\mathrm{FV}(M)$ of **free variables** in M is defined inductively as follows.

$$\mathrm{FV}(x) \triangleq \{x\}$$
$$\mathrm{FV}(\lambda x.M) \triangleq \mathrm{FV}(M) - \{x\}$$
$$\mathrm{FV}(MN) \triangleq \mathrm{FV}(\mathrm{M}) \cup \mathrm{FV}(\mathrm{N})$$

We can define **bound variables** $\mathrm{BV}(M)$ in a similar way.

Question. What are the free variables of $x(\lambda x.xy)$?

The Barendregt Convention. We will write bound variables so that they are distinct from each other and any free variable.

Variable Scope

Definition (Free and Bound Variables)

The set $\mathrm{FV}(M)$ of **free variables** in M is defined inductively as follows.

$$\mathrm{FV}(x) \triangleq \{x\}$$
$$\mathrm{FV}(\lambda x.M) \triangleq \mathrm{FV}(M) - \{x\}$$
$$\mathrm{FV}(MN) \triangleq \mathrm{FV}(\mathrm{M}) \cup \mathrm{FV}(\mathrm{N})$$

We can define **bound variables** $\mathrm{BV}(M)$ in a similar way.

Question. What are the free variables of $x(\lambda x.xy)$?

The Barendregt Convention. We will write bound variables so that they are distinct from each other and any free variable.

Variable Scope

Definition (Free and Bound Variables)

The set $\mathrm{FV}(M)$ of **free variables** in M is defined inductively as follows.

$$\mathrm{FV}(x) \triangleq \{x\}$$
$$\mathrm{FV}(\lambda x.M) \triangleq \mathrm{FV}(M) - \{x\}$$
$$\mathrm{FV}(MN) \triangleq \mathrm{FV}(\mathrm{M}) \cup \mathrm{FV}(\mathrm{N})$$

We can define **bound variables** $\mathrm{BV}(M)$ in a similar way.

Question. What are the free variables of $x(\lambda x.xy)$?

The Barendregt Convention. We will write bound variables so that they are distinct from each other and any free variable.

Substitution, Actually

Definition (Renaming and Capture-Avoiding Substitution)

The **renaming of** y **to** z **in** M, written $M^{y \to z}$, is the term that results from replacing all free occurrences of y with the variable z in M. The **substitution of** x **by** N **in** M, written $M[N/x]$, is defined inductively as follows.

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

$$(\lambda y.M)[N/x] \triangleq \begin{cases} \lambda y.M & x = y \\ \lambda y.M[N/x] & y \notin \mathrm{FV}(N) \\ \lambda z.M^{y \to z}[N/x] & \text{o.w. } (z \notin \mathrm{FV}(N) \cup \mathrm{FV}(M) \cup \mathrm{BV}(M)) \end{cases}$$

$$(PQ)[N/x] \triangleq P[N/x]Q[N/x]$$

The condition that $z \notin \mathrm{FV}(N) \cup \mathrm{FV}(M)$ ensures that free variables are not captured by the bound variable z.

Practice Problems

1. Write down formal inductive definitions for bound variables and for the variable renaming operation.
2. In the definition of substitution, why do we require that the renamed variable does not appear in $\mathrm{BV}(M)$? Try to come up with an example.
3. Where is the definition of substitution underspecified? What can be done to make it more specific?
4. Give the result of the following substitutions.[1]
   - $(\lambda x.y(\lambda y.xy))[\lambda z.zx/y]$
   - $(\lambda y.yyx)[yz/x]$

[1] This is based on Problem 1.5 from TTFP.

So far... (part 2)

We've defined λ-terms and we've given a rough sketch of β-reduction as a form of computation for the λ-calculus:

$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

where $M[N/x]$ is the result of substituting N for x in M. Furthermore, we should be able to do this anywhere in the term.

Question. How do we handle the "anywhere in the term" part of the definition of β-reduction? And what about computations with more than one step?

So far... (part 2)

We've defined λ-terms and we've given a rough sketch of β-reduction as a form of computation for the λ-calculus:

$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

where $M[N/x]$ is the result of substituting N for x in M. Furthermore, we should be able to do this anywhere in the term.

Question. How do we handle the "anywhere in the term" part of the definition of β-reduction? And what about computations with more than one step?

Notions of Reductions

We've seen:

$$\lambda x.M \longrightarrow_\alpha \lambda z.M^{x\to z} \quad (z \notin \mathsf{FV}(M) \cup \mathsf{BV}(M))$$
$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

Definition (Notion of Reduction)

A **notion of reduction** is a binary relation on λ-terms. For example,

$$\beta = \{(\ (\lambda x.M)N\ ,\ M[N/x]\ ) : M, N \in \Lambda, x \in V\}.$$

Question. Can we lift this to a relation on all terms?

Notions of Reductions

We've seen:

$$\lambda x.M \longrightarrow_\alpha \lambda z.M^{x\to z} \quad (z \notin \mathsf{FV}(M) \cup \mathsf{BV}(M))$$
$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

Definition (Notion of Reduction)

A **notion of reduction** is a binary relation on λ-terms. For example,

$$\beta = \{(\ (\lambda x.M)N\ ,\ M[N/x]\ ) : M, N \in \Lambda, x \in V\}.$$

Question. Can we lift this to a relation on all terms?

Notions of Reductions

We've seen:

$$\lambda x.M \longrightarrow_\alpha \lambda z.M^{x\to z} \quad (z \notin \mathsf{FV}(M) \cup \mathsf{BV}(M))$$
$$(\lambda x.M)N \longrightarrow_\beta M[N/x]$$

Definition (Notion of Reduction)

A **notion of reduction** is a binary relation on λ-terms. For example,

$$\beta = \{(\ (\lambda x.M)N\ ,\ M[N/x]\ ) : M, N \in \Lambda, x \in V\}.$$

Question. Can we lift this to a relation on all terms?

Redexes and Compatibility

One way. The **compatibility closure** of a notion of reduction R, written $\to_R$, is the smallest relation satisfying the following closure properties.

- $M \to_R M'$ implies $\lambda x.M \to_R \lambda x.M'$
- $M \to_R M'$ implies $MN \to_R M'N$
- $N \to_R N'$ implies $MN \to_R MN'$

Another way. The set of **subterms** $\mathrm{sub}(M)$ of a term M is defined formally as follows.

$$\mathrm{sub}(x) \triangleq \{x\}$$

$$\mathrm{sub}(\lambda x.M) \triangleq \mathrm{sub}(M) \cup \{\lambda x.M\}$$

$$\mathrm{sub}(MN) \triangleq \mathrm{sub}(M) \cup \mathrm{sub}(N) \cup \{MN\}$$

An R-**redex** of a term M is a subterm N of M such that there is a term N' with $(N, N') \in R$. Then

$$(\ldots N \ldots) \to_R (\ldots N' \ldots)$$

Redexes and Compatibility

One way. The **compatibility closure** of a notion of reduction R, written $\to_R$, is the smallest relation satisfying the following closure properties.

- $M \to_R M'$ implies $\lambda x.M \to_R \lambda x.M'$
- $M \to_R M'$ implies $MN \to_R M'N$
- $N \to_R N'$ implies $MN \to_R MN'$

Another way. The set of **subterms** $\mathrm{sub}(M)$ of a term M is defined formally as follows.

$$\begin{aligned} \mathrm{sub}(x) &\triangleq \{x\} \\ \mathrm{sub}(\lambda x.M) &\triangleq \mathrm{sub}(M) \cup \{\lambda x.M\} \\ \mathrm{sub}(MN) &\triangleq \mathrm{sub}(M) \cup \mathrm{sub}(N) \cup \{MN\} \end{aligned}$$

An R-**redex** of a term M is a subterm N of M such that there is a term N' with $(N, N') \in R$. Then

$$(\ldots N \ldots) \to_R (\ldots N' \ldots)$$

Some Examples of Reductions

Lemma

These two definition of $\rightarrow_R$ are equivalent.

Example

- The redexes in the following term are underlined

$$v(\underline{(\lambda x.\underline{(\lambda y.yx)z})v})$$

- $v((\lambda x.\underline{(\lambda y.yx)z})v) \rightarrow_\beta v((\lambda x.zx)v)$
- $v(\underline{(\lambda x.(\lambda y.yx)z)v}) \rightarrow_\beta v((\lambda y.zv)z)$

Question. And what about more than one computation step?

Some Examples of Reductions

Lemma

These two definition of $\rightarrow_R$ are equivalent.

Example

- The redexes in the following term are underlined

$$v(\underline{(\lambda x.\underline{(\lambda y.yx)z})v})$$

- $v((\lambda x.\underline{(\lambda y.yx)z})v) \rightarrow_\beta v((\lambda x.zx)v)$
- $v(\underline{(\lambda x.(\lambda y.yx)z)v}) \rightarrow_\beta v((\lambda y.zv)z)$

Question. And what about more than one computation step?

Some Examples of Reductions

Lemma

These two definition of $\rightarrow_R$ are equivalent.

Example

- The redexes in the following term are underlined

$$v(\underline{(\lambda x.\underline{(\lambda y.yx)z})v})$$

- $v((\lambda x.\underline{(\lambda y.yx)z})v) \rightarrow_\beta v((\lambda x.zx)v)$
- $v(\underline{(\lambda x.(\lambda y.yx)z)v}) \rightarrow_\beta v((\lambda y.zv)z)$

Question. And what about more than one computation step?

Reduction Paths

One way. The **reflexive transitive closure** of a notion of reduction R, written $\twoheadrightarrow_R$, is the smallest relation satisfying

- $M \to_R N$ implies $M \twoheadrightarrow_R N$ (closure)
- $M \twoheadrightarrow_R M$ (reflexivity)
- $M \twoheadrightarrow_R N$ and $N \twoheadrightarrow_R P$ implies $M \twoheadrightarrow_R P$ (transitivity)

Another way. An R-**path** for a notion of reduction R starting at M is a sequences of terms $M_1, \ldots, M_k$ such that

$$M = M_1 \to_R M_2 \to_R \cdots \to_R M_k$$

(Note that this definition also works for infinite sequences)

We write $M \twoheadrightarrow_\beta N$ if there is an (possibly empty) R-path starting at M and ending at N. We call N a **reduct** of M.

Reduction Paths

One way. The **reflexive transitive closure** of a notion of reduction R, written $\twoheadrightarrow_R$, is the smallest relation satisfying

- $M \to_R N$ implies $M \twoheadrightarrow_R N$ (closure)
- $M \twoheadrightarrow_R M$ (reflexivity)
- $M \twoheadrightarrow_R N$ and $N \twoheadrightarrow_R P$ implies $M \twoheadrightarrow_R P$ (transitivity)

Another way. An R-**path** for a notion of reduction R starting at M is a sequences of terms $M_1, \ldots, M_k$ such that

$$M = M_1 \to_R M_2 \to_R \cdots \to_R M_k$$

(Note that this definition also works for infinite sequences)

We write $M \twoheadrightarrow_\beta N$ if there is an (possibly empty) R-path starting at M and ending at N. We call N a **reduct** of M.

Examples of Reduction Paths

Lemma

These two definitions of $\twoheadrightarrow_R$ are equivalent.

Example

- $(\lambda fgx.f(gx))abc \twoheadrightarrow_\beta a(bc)$
- $v((\lambda x.(\lambda y.yx)z)v) \twoheadrightarrow_\beta v(zv)$
- $(\lambda x.xx)(\lambda x.xx) \twoheadrightarrow_\beta (\lambda x.xx)(\lambda x.xx)$

Examples of Reduction Paths

Lemma

These two definitions of $\twoheadrightarrow_R$ are equivalent.

Example

- $(\lambda fgx.f(gx))abc \twoheadrightarrow_\beta a(bc)$
- $v((\lambda x.(\lambda y.yx)z)v) \twoheadrightarrow_\beta v(zv)$
- $(\lambda x.xx)(\lambda x.xx) \twoheadrightarrow_\beta (\lambda x.xx)(\lambda x.xx)$

Equivalences

One way. For a notion of reduction R the **equivalence closure**, written $=_R$ is the smallest relation satisfying

- $M \twoheadrightarrow_R N$ implies $M =_R N$
- $M =_R M$
- $M =_R N$ implies $N =_R M$
- $M =_R N$ and $N =_R P$ implies $M =_R P$.

Another way. We write $M =_R N$ if there is a term P such that $M \twoheadrightarrow_R P$ and $N \twoheadrightarrow_R P$. This naturally captures the idea of computing two terms to see if they are the same.

Theorem

These two definitions are equivalent for α, β, and γ.

Equivalences

One way. For a notion of reduction R the **equivalence closure**, written $=_R$ is the smallest relation satisfying

- $M \twoheadrightarrow_R N$ implies $M =_R N$
- $M =_R M$
- $M =_R N$ implies $N =_R M$
- $M =_R N$ and $N =_R P$ implies $M =_R P$.

Another way. We write $M =_R N$ if there is a term P such that $M \twoheadrightarrow_R P$ and $N \twoheadrightarrow_R P$. This naturally captures the idea of computing two terms to see if they are the same.

Theorem

These two definitions are equivalent for α, β, and γ.

Equivalences

One way. For a notion of reduction R the **equivalence closure**, written $=_R$ is the smallest relation satisfying

- $M \twoheadrightarrow_R N$ implies $M =_R N$
- $M =_R M$
- $M =_R N$ implies $N =_R M$
- $M =_R N$ and $N =_R P$ implies $M =_R P$.

Another way. We write $M =_R N$ if there is a term P such that $M \twoheadrightarrow_R P$ and $N \twoheadrightarrow_R P$. This naturally captures the idea of computing two terms to see if they are the same.

Theorem

These two definitions are equivalent for α, β, and γ.

β-Equivalence

Important

β-Equivalence gives us a notion of semantic equivalence for λ-terms.

Per the previous slides, one simple way to tell if two terms are β equivalent is to show that they share a reduct.

Example

Even though neither reduces to the other

$$(\lambda x.gx)u =_\beta (\lambda y.yu)g$$

since they both reduce to gu.

β-Equivalence

Important

β-Equivalence gives us a notion of semantic equivalence for λ-terms.

Per the previous slides, one simple way to tell if two terms are β equivalent is to show that they share a reduct.

Example

Even though neither reduces to the other

$$(\lambda x.gx)u =_\beta (\lambda y.yu)g$$

since they both reduce to gu.

β-Equivalence

Important

β-Equivalence gives us a notion of semantic equivalence for λ-terms.

Per the previous slides, one simple way to tell if two terms are β equivalent is to show that they share a reduct.

Example

Even though neither reduces to the other

$$(\lambda x.gx)u =_\beta (\lambda y.yu)g$$

since they both reduce to gu.

α-Equivalence

An Observation. Self-substitution may not preserve syntactic equivalence, e.g.,

$$(\lambda y.y)[y/y] = \lambda z.z$$

for some variable z.

Important (Barendregt Convention Revisited)

We consider syntactic equivalence ($\equiv$) up to α-Equivalence, and use distinct bound variables when writing down λ-terms to emphasize this.

Said another way, when we write $M =_R N$, we really mean $M =_{R\alpha} N$, where we think of $R\alpha$ as the notion of reduction $R \cup \alpha$.

Example. We can use the symbol $\mathsf{I} \equiv \lambda x.x$ without worrying about which variable we used. Furthermore, we can write things like $\mathsf{II} =_\beta \mathsf{I}$

α-Equivalence

An Observation. Self-substitution may not preserve syntactic equivalence, e.g.,

$$(\lambda y.y)[y/y] = \lambda z.z$$

for some variable z.

Important (Barendregt Convention Revisited)

We consider syntactic equivalence ($\equiv$) up to α-Equivalence, and use distinct bound variables when writing down λ-terms to emphasize this.

Said another way, when we write $M =_R N$, we really mean $M =_{R\alpha} N$, where we think of $R\alpha$ as the notion of reduction $R \cup \alpha$.

Example. We can use the symbol $\mathsf{I} \equiv \lambda x.x$ without worrying about which variable we used. Furthermore, we can write things like $\mathsf{II} =_\beta \mathsf{I}$

α-Equivalence

An Observation. Self-substitution may not preserve syntactic equivalence, e.g.,

$$(\lambda y.y)[y/y] = \lambda z.z$$

for some variable z.

Important (Barendregt Convention Revisited)

We consider syntactic equivalence ($\equiv$) up to α-Equivalence, and use distinct bound variables when writing down λ-terms to emphasize this.

Said another way, when we write $M =_R N$, we really mean $M =_{R\alpha} N$, where we think of $R\alpha$ as the notion of reduction $R \cup \alpha$.

Example. We can use the symbol $\mathsf{I} \equiv \lambda x.x$ without worrying about which variable we used. Furthermore, we can write things like $\mathsf{II} =_\beta \mathsf{I}$

α-Equivalence

An Observation. Self-substitution may not preserve syntactic equivalence, e.g.,

$$(\lambda y.y)[y/y] = \lambda z.z$$

for some variable z.

Important (Barendregt Convention Revisited)

We consider syntactic equivalence ($\equiv$) up to α-Equivalence, and use distinct bound variables when writing down λ-terms to emphasize this.

Said another way, when we write $M =_R N$, we really mean $M =_{R\alpha} N$, where we think of $R\alpha$ as the notion of reduction $R \cup \alpha$.

Example. We can use the symbol $\mathsf{I} \equiv \lambda x.x$ without worrying about which variable we used. Furthermore, we can write things like $\mathsf{II} =_\beta \mathsf{I}$

Normal Forms

Definition

An R-**normal form** is a term N such that N has no R-redexes. A term M is R-**normalizing** if there is an R-normal form N such that $M \twoheadrightarrow_R N$.

Example

- x, xy, $\lambda x.x$, $\lambda xy.x$ are β normal forms.
- $(\lambda x.x)(\lambda y.y)$ is not a β normal form.
- $v((\lambda x.(\lambda y.yx)z)v)$ is β-normalizing.
- $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ is not β-normalizing.

Lemma

If $M =_\beta N$ and N is a normal form, then $M \twoheadrightarrow_\beta N$.

So our notion semantic equivalence and the value of a computation are compatible.

Normal Forms

Definition

An R-**normal form** is a term N such that N has no R-redexes. A term M is R-**normalizing** if there is an R-normal form N such that $M \twoheadrightarrow_R N$.

Example

- x, xy, $\lambda x.x$, $\lambda xy.x$ are β normal forms.
- $(\lambda x.x)(\lambda y.y)$ is not a β normal form.
- $v((\lambda x.(\lambda y.yx)z)v)$ is β-normalizing.
- $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ is not β-normalizing.

Lemma

If $M =_\beta N$ and N is a normal form, then $M \twoheadrightarrow_\beta N$.

So our notion semantic equivalence and the value of a computation are compatible.

Normal Forms

Definition

An R-**normal form** is a term N such that N has no R-redexes. A term M is R-**normalizing** if there is an R-normal form N such that $M \twoheadrightarrow_R N$.

Example

- x, xy, $\lambda x.x$, $\lambda xy.x$ are β normal forms.
- $(\lambda x.x)(\lambda y.y)$ is not a β normal form.
- $v((\lambda x.(\lambda y.yx)z)v)$ is β-normalizing.
- $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ is not β-normalizing.

Lemma

If $M =_\beta N$ and N is a normal form, then $M \twoheadrightarrow_\beta N$.

So our notion semantic equivalence and the value of a computation are compatible.

Normal Forms

Definition

An R-**normal form** is a term N such that N has no R-redexes. A term M is R-**normalizing** if there is an R-normal form N such that $M \twoheadrightarrow_R N$.

Example

- x, xy, $\lambda x.x$, $\lambda xy.x$ are β normal forms.
- $(\lambda x.x)(\lambda y.y)$ is not a β normal form.
- $v((\lambda x.(\lambda y.yx)z)v)$ is β-normalizing.
- $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ is not β-normalizing.

Lemma

If $M =_\beta N$ and N is a normal form, then $M \twoheadrightarrow_\beta N$.

So our notion semantic equivalence and the value of a computation are compatible.

Practice Problems

1. Write down the subterms and the redexes of $((\lambda xy.yx)z)((\lambda z.z)w)$.
2. Find two terms M and N such that $M \to_\beta N$ and N has more redexes than M.
3. Let $S \equiv \lambda xyz.xz(yz)$ and $K \equiv \lambda xy.x$. Show that $SKK \twoheadrightarrow_\beta I$.
4. Find a term M with is β-normalizing, but there is an infinite R-path starting at M.
5. Are there α normal forms?

Final Recap

- We've defined λ-terms, in a couple ways. These are the syntactic basis for the lambda calculus.
- We've discussed notions of reduction, particularly β-reduction as way to think about the operational semantics of the lambda calculus.
- We've seen how tricky getting these notions exactly right can be.
- We've looked at normal forms as a notion of the value of a computation of a λ-term.
- We've gotten some practice looking at how reductions behave.

Looking Forward

- Are β-normal forms the "return values" of a computation? If so, β-normal forms should probably be unique. (hint. Church-Rosser Theorem)
- So we've figured out substitution. Is there really no better option? (hint. De Bruijn indices)
- We've said the λ-calculus is used for computation. But how much can we actually do? (hint. Church Encodings and the Fixed-Point theorem)

A Challenge Problem

Definition (Reduction Graphs)

The β-reduction graph of a term M is a directed graph on the set $\{N : M \twoheadrightarrow_\beta N\}$ with edges generated by $\to_\beta$.

1. Draw the reduction graphs for Ω, $KI\Omega$, and $v((\lambda x.(\lambda y.yx)z)v)$.
2. Write a term whose reduction graph is the directed 3-cyclic graph.

(This may be easier with a picture)[2]

[2] See handwritten notes for more details.