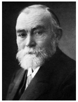
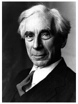
theorem begin

Foundations 1, F29FA1
Lecture 1

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

- Earlier, functions only dealt with numbers.
- *General definition of function 1879* is key to Frege's *formalisation of logic*. Now, functions take complex arguments and return complex results.
- Computer programs are generalized functions.
- *Self-application of functions* (e.g., fun is fun) was at the heart of *Russell's paradox 1902*.
- In programming we need to apply a program to other programs (and sometimes to itself).
- To *avoid paradox* Russell controled function application via *type theory*. Most programming languages use type theory to ensure correctness and termination.

- Programming languages have various degrees of typing
- Russell *1903* gives the first type theory: the *Ramified Type Theory* (RTT).
- RTT is used in Russell and Whitehead's Principia Mathematica 1910–1912.
- RTT was eventually simplified to STT (simple type theory) which is used in a number of programming paradigms.
- STT was not powerful enough so other more powerful type systems developed like polymorphic type theory (and you have worked with polymorphism in a number of programming languages).

- *Simple theory of types* (STT): Ramsey *1926*, Hilbert and Ackermann *1928*.
- Church's *simply typed λ-calculus* $\lambda{\to}$ 1940 $=$ λ-calculus $+$ STT.
- The hierarchies of types (and orders) as found in RTT and STT are *unsatisfactory*.
- Hence, birth of *different systems of functions and types*, each with *different functional power*.

- Hence, birth of *different programming languages* each with *different expressiveness and speed*.
- Frege's functions $\neq$ Principia's functions $\neq$ *λ-calculus* functions.
- Which notions of functions are needed in programming languages?
- *Non-first-class functions* allow us to stay at a lower order (keeping decidability, typability, computability, etc.) without losing the flexibility of the higher-order aspects.

In the 19th century, the need for a more *precise* style in mathematics arose, because controversial results had appeared in analysis.

- 1821: Many of these controversies were solved by the work of Cauchy. E.g., he introduced *a precise definition of convergence* in his *Cours d'Analyse*
- 1872: Due to the more *exact definition of real numbers* given by Dedekind, the rules for reasoning with real numbers became even more precise.
- 1895-1897: Cantor began formalizing *set theory* and made contributions to *number theory*.
- 1889: Peano formalized *arithmetic*, but did not treat logic or quantification.

- 1879: Frege was not satisfied with the use of natural language in mathematics:

  > *"...I found the inadequacy of language to be an obstacle; no matter how unwieldy the expressions I was ready to accept, I was less and less able, as the relations became more and more complex, to attain the precision that my purpose required."*
  >
  > (Begriffsschrift, *Preface*)

- Frege therefore presented *Begriffsschrift*, the first formalisation of logic giving logical concepts via symbols rather than natural language.

> *"[Begriffsschrift's] first purpose is to provide us with the most reliable test of the validity of a chain of inferences and to point out every presupposition that tries to sneak in unnoticed, so that its origin can be investigated."*
>
> (Begriffsschrift, *Preface)*

The introduction of a *very general definition of function* was the key to the formalisation of logic. Frege defined what we will call the Abstraction Principle.

Abstraction Principle

> *"If in an expression, [...] a simple or a compound sign has one or more occurrences and if we regard that sign as replaceable in all or some of these occurrences by something else (but everywhere by the same thing), then we call the part that remains invariant in the expression a function, and the replaceable part the argument of the function."*

(*Begriffsschrift*, Section 9)

- E.g., 2 apple = apple + apple
  2 chair = chair + chair
  Hence the function $2x = x + x$.
- Frege put *no restrictions* on what could play the role of *an argument*.
- An argument could be a *number* (as was the situation in analysis), but also a *proposition*, or a *function*.
- he *result of applying* a function to an argument did not have to be a number.

Frege was aware of some typing rule that does not allow to substitute functions for object variables or objects for function variables:

> *" Now just as functions are fundamentally different from objects, so also functions whose arguments are and must be functions are fundamentally different from functions whose arguments are objects and cannot be anything else. I call the latter first-level, the former second-level."*
>
> *(*Function and Concept*, pp. 26–27)*

The *Begriffsschrift*, however, was only a prelude to Frege's writings.

- In Grundlagen der Arithmetik he argued that mathematics can be seen as a branch of logic.
- In Grundgesetze der Arithmetik he described the elementary parts of arithmetics within an extension of the logical framework of *Begriffsschrift*.
- Frege approached the *paradox threats for a second time* at the end of Section 2 of his *Grundgesetze*.
- He did not *apply a function to itself*, but to its course-of-values.
- "the function $\Phi(x)$ has the same *course-of-values* as the function $\Psi(x)$" if:

> *"$\Phi(x)$ and $\Psi(x)$ always have the same value for the same argument."*
>
> (Grundgesetze, *p. 7)*

- E.g., let $\Phi(x)$ be $x \wedge \neg x$, and $\Psi(x)$ be $x \leftrightarrow \neg x$, for all propositions x.
- All essential information of a function is contained in its graph.
- So a system in which a function can be applied to its own graph should have similar possibilities as a system in which a function can be applied to itself.
- Frege *excluded the paradox threats* by *forbidding self-application*, but due to his *treatment of courses-of-values* these threats were able to *enter his system through a back door*.
- In 1902, Russell wrote to Frege that he had *discovered a paradox* in Frege's system.

- Frege's system was *not the only paradoxical* one.
- The Russell Paradox can be derived in *Peano's system* as well, as well as on *Cantor's Set Theory* by defining the class $K \stackrel{\mathrm{def}}{=} \{x \mid x \notin x\}$ and deriving $K \in K \longleftrightarrow K \notin K$.
- Paradoxes were already widely known in *antiquity*.
- The oldest logical paradox: the *Liar's Paradox* "This sentence is not true", also known as the Paradox of Epimenides. It is referred to in the Bible (Titus 1:12) and is based on the confusion between language and meta-language.
- The *Burali-Forti paradox* is the first of the modern paradoxes. It is a paradox within Cantor's theory on ordinal numbers.
- *Cantor's paradox* on the largest cardinal number occurs in the same field. It discovered by Cantor around 1895, but was not published before 1932.

- Logicians considered these paradoxes to be *out of the scope of logic*:
  The *Liar's Paradox* can be regarded as a problem of *linguistics*.
  The *paradoxes of Cantor and Burali-Forti* occurred in what was considered in those days a *highly questionable* part of mathematics: Cantor's Set Theory.
- The Russell Paradox, however, was *a paradox that could be formulated in all* the systems that were presented at the end of the 19th century (except for Frege's *Begriffsschrift*). It was at the very basics of logic. It could not be disregarded, and a solution to it had to be found.
- In 1903-1908, Russell suggested the use of *types* to solve the problem

- *"In all the above contradictions there is a common characteristic, which we may describe as self-reference or reflexiveness. [...] In each contradiction something is said about* all *cases of some kind, and from what is said a new case seems to be generated, which both is and is not of the same kind as the cases of which* all *were concerned in what was said."*

  (Mathematical logic as based on the theory of types)
- Russell's plan was, *to avoid the paradoxes* by *avoiding all possible self-references*.

- Russell postulated the *"vicious circle principle"*:
- "Whatever involves all of a collection *must not be one* of the collection."

  (Mathematical logic as based on the theory of types)
- Russell implements this principle *very strictly* using *types*.

- Ramsey considers it essential to divide the paradoxes into two parts:
- logical or syntactical paradoxes (like the Russell paradox, and the Burali-Forti paradox) are removed

  > *"by pointing out that a propositional function cannot significantly take itself as argument, and by dividing functions and classes into a hierarchy of types according to their possible arguments."*
  >
  > *(The Foundations of Mathematics, p. 356)*

- Semantical paradoxes are excluded by the hierarchy of orders. These paradoxes (like the Liar's paradox, and the Richard Paradox) are based on the confusion of language and meta-language. These paradoxes are, therefore, not of a purely mathematical or logical nature.

- Our course is about programs (special kinds of functions) and fast, efficient, terminating computations.
- Historically, *functions* have long been treated as a kind of *meta-objects*.
- However, we want to reason about our programs (does my program terminate? is it fast, etc).
- Church's lambda calculus made every function a first-class citizen.
- Functions have gone through a long process of evolution involving various degrees of abstraction/construction/instantiation/evaluation.
- Programs too have and are going through a long process of evolution.

- *Aims* To acquaint the students with the syntax and semantics of lambda calculus and reduction strategies. Solving mutually recursive equations and fixed point theorems. Substitution, call by name, call by value, termination.
- *Learning Outcomes of lambda calculus part* Competence in lambda calculus, different variable techniques (de Bruijn indices), semantics of small programs.

- *Main References*
  1. Chris Hankin, An introduction to lambda calculi for computer scientists. King's college publications, Texts in Computing, Volume 2, 164 pages. ISBN 0-9543006-5-3.
  2. Mike Gordon, Programming Language Theory and Implementation. Prentice Hall. ISBN 0-13-730409-9.
  3. Henk Barendregt, the syntax and semantics of the lambda calculus. North-Holland.

Functions as first class objects

- Functional programming is based on the notion of function and of function application.
- In functional programming, functions are first class objects and they can be applied to themselves, or to other functions leading either other functions as result.
- For example, *add* is a function that takes two numbers and returns a number.
- *add 1* is also a function that takes a number and adds 1 to it.

Polymorphic functions

- In addition to this higher order nature of functions in functional programming, we have the polymorphic nature, which enables us to write one function only and specialise the function to whichever type we are working with.
- For example, the identity function which takes numbers and return numbers, takes lists and returns lists, etc.
- So we can have:
  $\mathrm{Id}_{\mathcal{N}} : \mathcal{N} \mapsto \mathcal{N}$
  $\mathrm{Id}_{\mathrm{Lists}} : \mathrm{Lists} \mapsto \mathrm{Lists}$

One simple language can represent all that

- It might be surprising to know that notions of higher order, polymorphism, functional application, recursion and many other functional programming notions can be captured in a very precise way in a very simple language.
- This simple language contains simply functional abstraction and functional application.
- In the next few lectures we will see how we can capture parts of functional programming in such a language, the type free λ- calculus.

This first lecture was an introduction. It is not examinable. From next lecture, everything is examinable.

theorem begin

Foundations 1, F29FA1
Lecture 2

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

One simple language can represent all that

- It might be surprising to know that notions of higher order, polymorphism, functional application, recursion and many other functional programming notions can be captured in a very precise way in a very simple language.
- This simple language contains simply functional abstraction and functional application.
- In the next few lectures we will see how we can capture parts of functional programming in such a language, the type free λ- calculus.

The syntax of the λ-calculus: Variables versus meta-variables

- Let $\mathcal{V} = \{x, y, z, x', y', z', x_1, y_1, z_1, \dots\}$ be an infinite set of *term variables*. Elements of $\mathcal{V}$ are also called *object variables*. They are the *real* variables which will appear in the terms.
- We let $v, v', v'', v_1, v_2, \cdots$ range over $\mathcal{V}$.
  We call $v, v', v'', v_1, v_2, \cdots$, *meta-variables*.
  These are variables used to *talk* about the object variables.
- Here, $\mathcal{V}$ is a set and $x \neq y$, $x \neq z$, $x \neq x'$, ...
  $y \neq x$, $y \neq z$, $y \neq x'$, ...
  etc....
- However, we don't know which of the vs, v's, etc are different. It all depends on what we intend them to be.

The syntax of the λ-calculus: λ-terms

- The set of classical λ-terms or λ-expressions $\mathcal{M}$ is given by: $\mathcal{M} ::= \mathcal{V} \mid (\lambda\mathcal{V}.\mathcal{M}) \mid (\mathcal{M}\mathcal{M})$.
- Hence, an element of $\mathcal{M}$ is either:
  - a *variable* v from $\mathcal{V}$ which can be x, or y, etc..
  - or an *abstraction* $(\lambda v.A)$ where A is any λ-term in $\mathcal{M}$.
  - or an *application* (AB) where A and B are any λ-terms in $\mathcal{M}$.
- We let $A, B, C \cdots$ range over $\mathcal{M}$.

Examples of λ-terms

- a variable x_1
- an abstraction $(\lambda x.x)$
- an abstraction $(\lambda y.y)$
- an application (xx)
- an abstraction $(\lambda x.(xx))$ whose *body* (xx) is an application
- an abstraction $(\lambda x.(\lambda y.x))$ whose *body* $(\lambda y.x)$ is an abstraction
- an abstraction $(\lambda x.(\lambda y.(xy)))$ whose *body* $(\lambda y.(xy))$ is an abstraction
- an application $((\lambda x.x)(\lambda y.y))$ whose *left* and *right parts* are $(\lambda x.x)$ and $(\lambda y.y)$ resp.
- an application $((\lambda x.(xx))(\lambda x.(xx)))$ whose *left* and *right parts* are $(\lambda x.(xx))$.

The meaning of λ-expressions

- This simple language is surprisingly rich. Its richness comes from the freedom to create and apply (higher order) functions to other functions (and even to themselves).
- To explain the meaning of these three sorts of expressions, let us imagine a model D where every λ-expression denotes an element of that model (which is a function).
- I.e., the meaning of expressions is a function $: \mathcal{M} \mapsto D.$
- For this to work, we need an interpretation function or an *environment* σ which maps every *variable* of $\mathcal{V}$ into a specific element of the model D. *I.e.* $\sigma : \mathcal{V} \mapsto D.$

Models of the λ-calculus

- Such a model was not obvious for more than forty years.
- In fact, for a domain D to be a model of λ-calculus, it requires that the set of functions from D to D be included in D.
- Moreover, we know from Cantor's theorem that the domain D is much smaller than the set of functions from D to D.
- Dana Scott was armed by this theorem in his attempt to show the non-existence of the models of the λ-calculus.
- To his surprise, he proved the opposite of what he set out to show. He found in 1969 a model which has opened the door to an extensive area of research in computer science.

- Here is the intuitive meaning of the three λ-expressions:
  - *Variables* Functions denoted by variables are determined by what the variables are bound to in the *environment* σ.
  - *Function application* Let A and B be λ-expressions. The expression (AB) denotes the result of applying the function denoted by A to the function denoted by B.
  - *Abstraction* Let v be a variable and A be an expression which may or may not contain occurrences of v. Then, in an environment σ, $(\lambda v.A)$ denotes the function that maps an input value a to the output value which denotes the meaning of A in the environment σ where v is bound to a.

Environments and the meaning of variables

- Expressions have variables, and variables take values depending on the environment.
- Assume model D.
- Let $\mathrm{ENV} = \{\sigma \mid \sigma : \mathcal{V} \mapsto D\}$ be the collection of environments.
- For example, if D contains the natural numbers, then one σ could take x to 1, y to 2, z to 3, etc.
- In that case, the meaning of x is $\sigma(x) = 1$, the meaning of y is $\sigma(y) = 2$, etc.

The meaning of application

- The meaning of (AB) is the functional application of the meaning of A to the meaning of B.
- So, if the meaning of A is the identity function, and the meaning of B is the number 3 then the meaning of (AB) is the application of the identity function to 3 which gives 3.

The meaning of abstraction

- The meaning of $(\lambda v.A)$ in an environment σ, is to be the function which takes an object a and returns the function which denotes the meaning of A in the environment σ where v is bound to a.
- For example, $(\lambda x.x)$ denotes the identity function.
- $(\lambda x.(\lambda y.x))$ denotes the function which takes two arguments and returns the first.

Example of past exam questions on meaning of expressions

- Find the meaning of the expression $\lambda xy.xx$.
  - Answer: The meaning of the expression $\lambda xy.xx$ is the function which takes two arguments, ignores the second and applies the first to itself.
- Find the meaning of the expression $\lambda x.y$.
  - Answer: The meaning of the expression $\lambda x.y$ is the constant function which no matter what you give it, it returns the value of y.

Notational convention

- As parentheses are cumbersome, we will use the following notational convention:
  1. Functional application associates to the left. So *ABC denotes* $((AB)C)$.
  2. The body of a λ is anything that comes after it. So, *instead of* $(\lambda v.(A_1 A_2 \ldots A_n))$, *we write* $\lambda v.A_1 A_2 \ldots A_n$.
  3. A sequence of λ's is compressed to one. So $\lambda xyz.t$ *denotes* $\lambda x.(\lambda y.(\lambda z.t))$.

- As a consequence of these notational conventions we get:
  1. *Parentheses may be dropped:* (AB) and $(\lambda v.A)$ are written AB and $\lambda v.A$.
  2. *Application has priority over abstraction:* $\lambda x.yz$ means $\lambda x.(yz)$ and not $(\lambda x.y)z$.

Subterms

- We define the notion of *subterms*

$$
\begin{array}{lcl}
\text{Subterms}(v) & = & \{v\} \\
\text{Subterms}(\lambda v.A) & = & \text{Subterms}(A) \cup \{\lambda v.A\} \\
\text{Subterms}(AB) & = & \text{Subterms}(A) \cup \text{Subterms}(B) \cup \{AB\}
\end{array}
$$

- For example:
  $\text{Subterms}((\lambda x.x)(yz)) = \{x, y, z, \lambda x.x, yz, (\lambda x.x)(yz)\}$

Exercises for Tutorials 1

- 1. Find the meaning of the following expressions:
  1. $(\lambda x.x)$
  2. $(\lambda x.(xx))$
  3. $(\lambda x.(\lambda y.x))$
  4. $(\lambda x.(\lambda y.(xy)))$
  5. $((\lambda x.x)(\lambda x.x))$
- 2. Remove as many parenthesis as possible from the following:
  1. $(\lambda x.(xy))$
  2. $((\lambda y.y)(\lambda x.(xy)))$
  3. $((\lambda x.(xy))(\lambda x.(xy)))$
  4. $(\lambda x.(\lambda y.x))$
  5. $(\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$

- 3. Insert the full amount of parenthesis in the following:
    1. $y'x(yz)(\lambda x'.x'y)$
    2. $(\lambda xyz.xz(yz))x'y'z'$
    3. $x'(\lambda xyz.xz(yz))y'z'$
- Do exercises 1(a) and 1(b) (on subterms) of class test 2014 (see webpage of course for previous class tests).
- Other exercises you can do to practice the material of lecture 2 include whenever you see any expression A, add all parenthesis to A, remove all parenthesis from A, find all subterms of A, think of the meaning of A.

Foundations 1, F29FA1
Lecture 3

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

Subterms

- We define the notion of *subterms*

$$\begin{array}{lcl} \text{Subterms}(v) & = & \{v\} \\ \text{Subterms}(\lambda v.A) & = & \text{Subterms}(A) \cup \{\lambda v.A\} \\ \text{Subterms}(AB) & = & \text{Subterms}(A) \cup \text{Subterms}(B) \cup \{AB\} \end{array}$$

- For example:
  $\text{Subterms}((\lambda x.x)(yz)) = \{x, y, z, \lambda x.x, yz, (\lambda x.x)(yz)\}$

Trees of terms

- We can draw the terms graphically as trees. We use δ for application:

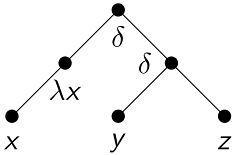


Figure: The tree of $(\lambda x.x)(yz)$

- Note that subterms are easy to see now. They are all the subtrees of the tree of a term.

Example of past exam questions on notational conventions

- Insert the full amount of parenthesis in the expression $(\lambda xyz.xz(yz))uvw$
  - Answer: $((((\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))u)v)w)$.
- Consider the expression $(((\lambda x.(\lambda y.(xy)))(\lambda z.(zz)))(\lambda x.x))$. Remove as many parenthesis as possible from this expression without changing its meaning.
  - Answer: $(\lambda xy.xy)(\lambda z.zz)(\lambda x.x)$.
- Insert the full amount of parenthesis in the expression $(\lambda y.xy)x(\lambda xyz.xyz)$.
  - Answer: $(((\lambda y.(xy))x)(\lambda x.(\lambda y.(\lambda z.((xy)z)))))$.
- Insert the full amount of parenthesis in the expression $x'(\lambda xyz.xz(yz))y'z'$.
  - Answer: $(((x'(\lambda x.(\lambda y.(\lambda z.((xz)(yz))))))y')z')$.

Example of past exam questions notational conventions and on subterms

- Consider the expression $(((\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))(\lambda x.(\lambda y.x)))(\lambda x.(\lambda y.(\lambda z.(x(yz))))))$. Remove as many parenthesis as possible from this expression without changing its meaning.
  - Answer: $(\lambda xyz.xz(yz))(\lambda xy.x)\lambda xyz.x(yz)$.
- Is $x(yz)$ a subterm of $ux(yz)$? Answer yes or no. If your answer is wrong you lose 2 marks.
  - Answer: No.
- Is $x'(\lambda xyz.xz(yz))y'z'$ a subterm of $(\lambda xyz.xz(yz))y'z'x'$? Answer yes or no. If your answer is wrong you lose 2 marks.
  - Answer: No.

Syntactic identity: definition which is cumbersome to work

- For NOW, we say that $A \equiv B$ iff A and B are exactly the same.
- For example, $x \equiv x$, $\lambda x.x \equiv \lambda x.x$.
- But $x \not\equiv y$, $\lambda x.x \not\equiv \lambda y.y$.
- We will see that this definition gives us extra problems and we will abandon it later.
- Note that if $AB \equiv A'B'$ then $A \equiv A'$ and $B \equiv B'$.
- Also, if $\lambda v.A \equiv \lambda v'.A'$ then $v \equiv v'$ and $A \equiv A'$.

Manipulating expressions

- We need to manipulate λ-expressions in order to get values.
- For example, we need to apply $(\lambda x.x)$ to y to obtain y.
- To do so, we must replace all occurrences of x in the body x of the function by the argument y.
- For this, we use the β-rule which says that $(\lambda v.A)B$ evaluates to *the body* A where v is substituted by B, written $A[v := B]$.
- This is written as: $(\lambda v.A)B \rightarrow_\beta A[v := B]$.
- However, one has to be careful. If we let $A[v := B]$ be the syntactic replacement of every v in A by B we get in deep trouble and we destroy our programming language. It would be of no use.

Strange: $(\lambda xy.xy)y$ and $(\lambda xz.xz)y$ have the same meaning, and if $(\lambda xy.xy)y \to_\beta \lambda y.yy$ and $(\lambda xz.xz)y \to_\beta \lambda z.yz$ then should we not get that $\lambda y.yy$ and $\lambda z.yz$ have the same meaning?

- After all, this is what we want in a programming language:
  - If a program A (in this case $(\lambda xy.xy)y$) evaluates to A' (in this case $\lambda y.yy$)
  - and a program B (in this case $(\lambda xz.xz)y$) evaluates to B' (in this case $\lambda z.yz$)
  - and if A and B have the same meaning,
  - then we want that B and B' also have the same meaning.

- The meaning of $\lambda xy.xy$ is the function which takes two arguments and applies the first to the second. This is also the meaning of $\lambda xz.xz$.
- Hence, the meaning of $(\lambda xy.xy)y$ is equal to the meaning of $(\lambda xz.xz)y$.
- Now, if $(\lambda xy.xy)y \to_\beta \lambda y.yy$ and $(\lambda xz.xz)y \to_\beta \lambda z.yz$ then the meaning of $\lambda y.yy$ must be equal to the meaning of $\lambda z.yz$.
- This is not the case however. The meaning of $\lambda y.yy$ is not equal to the meaning of $\lambda z.yz$. We will see this on the next slide.

The meaning of $\lambda y.yy$ is different from the meaning of $\lambda z.yz$

- Recall that $y \not\equiv z$.
- The meaning of $\lambda y.yy$ is the function which takes an argument a and applies it to itself giving $a(a)$.
- The meaning of $\lambda z.yz$ is the function which takes an argument a and applies the meaning of y (say g) to a.
- Since $f(a) = a(a)$ and $g(a) = g(a)$, obviously, $f \neq g$.
- Hence, the meaning of $\lambda y.yy \neq$ the meaning of $\lambda z.yz$.

Where did we go wrong?

- $(\lambda xy.xy)y$ and $(\lambda xz.xz)y$ have the same meaning *is correct*
- $(\lambda xy.xy)y$ evaluates to $\lambda y.yy$ *is false*
- $(\lambda xz.xz)y$ evaluates to $\lambda z.yz$ *is correct*
- $\lambda y.yy$ and $\lambda z.yz$ don't have the same meaning *is correct*

We need to correct the false statement.
$(\lambda xy.xy)y$ *does not evaluate* to $\lambda y.yy$

Variables and Substitution

- Evaluating $(\lambda xz.xz)y$ to $\lambda z.yz$ is perfectly acceptable. There is no problem with $(\lambda xz.xz)y \rightarrow_\beta \lambda z.yz$.
- But evaluating $(\lambda xy.xy)y$ to $\lambda y.yy$ is not acceptable. We should not accept $(\lambda xy.xy)y \rightarrow_\beta \lambda y.yy$.
- We will define the notions of *free* and *bound* occurrences of variables which will play an important role in avoiding the problem above.
- In fact, the λ is a variable binder, just like $\forall$ in logic.

Free and Bound occurrences of variables

- Take the two expressions x and $\lambda x.x$.
- What we have actually done in the second expression $\lambda x.x$ was to *bind* the variable x, so that the whole expression would not depend on x.
- In fact we could rename x by any other variable everywhere in $\lambda x.x$ and would still get an expression with the same meaning. If we rename x to y in $\lambda x.x$ we get $\lambda y.y$. $\lambda x.x$ has the same meaning as $\lambda y.y$.
- In the first expression x however, x is *free* (there is no λ that binds it) and cannot be renamed to another variable without changing the meaning of the expression.
- Even though $\lambda x.x$ is the same function as $\lambda y.y$, x is not the same as y.

- For a λ-term C, the set of free variables $FV(C)$ is defined inductively as follows:

$$\begin{array}{lll} FV(v) & =_{def} & \{v\} \\ FV(\lambda v.A) & =_{def} & FV(A) \setminus \{v\} \\ FV(AB) & =_{def} & FV(A) \cup FV(B) \end{array}$$

- Example (this was a previous exam question):
  Give the set of free variables of $\lambda y.yx(\lambda x.y(\lambda y.z)x)x'y'$.
  - $\{x, z, x', y'\}$.
- We need more. We need to talk about *occurrences* which are bound or free and about *scope*.

Scope and Occurrences

- We say that v is in the *scope* of λv in C if $\lambda v.A \in \text{Subterms}(C)$ and $v \in FV(A)$.
- For example, take $\lambda xy.xy$.
  - y° is in the scope of λy in $\lambda xy.xy^\circ$ because: $\lambda y.xy \in \text{Subterms}(\lambda xy.xy)$ and $y \in FV(xy)$.
  - x° is in the scope of λx in $\lambda xy.x^\circ y$ because: $\lambda xy.xy \in \text{Subterms}(\lambda xy.xy)$ and $x \in FV(\lambda y.xy)$.
- We can talk about the *(number of) occurrences* of a variable v in an expression A where we take into account the existence of v in A discounting the v's in the λv's.
- For example, x occurs twice in $(\lambda x.x^{\circ_1})x^{\circ_2}$ but zero times in $\lambda x.y$. The first occurrence of x in $(\lambda x.x^{\circ_1})x^{\circ_2}$ is labelled $^{\circ_1}$ whereas the second is labelled $^{\circ_2}$.

- The first occurrence of x in $(\lambda x.x^{\circ_1})x^{\circ_2}$ is in the scope of the λx and it is bound by that λ.
- The second occurrence of x in $(\lambda x.x^{\circ_1})x^{\circ_2}$ is not in the scope of any λ and it is free.
- Typical exam question: Now look at $\lambda y.yx(\lambda x.y(\lambda y.z)x)x'y'$. Label the occurrences of the variables and for each such occurrence say whether it is free or bound. For each bound occurrence, give the λ which binds it.
- Answer: $\lambda y.y^{\circ_1}x^{\circ_1}(\lambda x.y^{\circ_2}(\lambda y.z^{\circ_1})x^{\circ_2})x'^{\circ_1}y'^{\circ_1}$.
  y°_1} and y°_2} are both in the scope of the first λy and hence are bound by it.
  x°_1} is not in the scope of any λ, it is free.
  x°_2} is in the scope of the first λx and hence are bound by it.
  z°_1}, x'°_1} and y'°_1} are not in the scope of any λ, they are free.

Free and bound occurrences

- An occurrence of a variable v in a λ-expression A is free if that occurrence is not within the scope of a λv in A, otherwise it is bound.
- In $(\lambda x.yx)(\lambda y.xy)$, the first occurrence of y is free whereas the second is bound. Moreover, the first occurrence of x is bound whereas the second is free.
- In $\lambda y.x(\lambda x.yx)$ the first occurrence of x is free whereas the second is bound.
- In $(\lambda x.x)x$, the first occurrence of x is bound, yet the second occurrence is free.
- A *closed expression* is an expression in which all occurrences of variables are bound.

- Almost all λ-calculi identify terms that only differ in the name of their bound variables.
  - For example, since $\lambda x.x$ and $\lambda y.y$ have the same meaning (the identity function), they are usually identified.
- Substitution has to be handled with care due to the distinct roles played by bound and free occurrences of variables.
  - We know that $(\lambda xy.xy)y \to_\beta (\lambda y.xy)[x := y]$ but *we cannot* evaluate $(\lambda y.xy)[x := y]$ to $\lambda y.yy$.
  - After substitution, no free occurrences of a variable can become bound.
  - For example, $(\lambda y.xy)[x := y]$ must not return $\lambda y.yy$, since this would bind the free occurrence of x in $\lambda y.xy$.
  - $(\lambda y.xy)[x := y]$ should return something like $\lambda z.yz$.
  - $\lambda y.yy$ and $\lambda z.yz$ have different meanings.

How does $(\lambda y.xy)[x := y]$ return something like $\lambda z.yz$? Does any variable do? Can it be $\lambda v.yv$ for any variable $v \neq y$?

- We notice that x occurs free in $\lambda y.xy$ and we need to keep that occurrence free.
- We cannot evaluate $(\lambda y.xy)[x := y]$ to $\lambda y.yy$ since the occurrence that we wanted to keep free becomes bound.
- First we assume that terms that only differ in the name of their bound variables are identified (so $\lambda y.xy$ is the same as $\lambda z.xz$).
- This means that we need to revise the definition of syntactic identity given earlier (we will come to that later).

How does $(\lambda y.xy)[x := y]$ return something like $\lambda z.yz$? Does any variable do? Can it be $\lambda v.yv$ for any $v \neq y$?

- Then we rename the bound y in $\lambda y.xy$ to z obtaining $(\lambda z.xz)$, and then we perform the substitution $(\lambda z.xz)[x := y]$ which gives us $\lambda z.yz$.
- Note that if we rename the bound y in $\lambda y.xy$ to x obtaining $(\lambda x.x^{\circ}x)$, we bind the free occurrence of x°. So, this is not allowed. $\lambda x.xx$ does not have the same meaning as $\lambda y.xy$.
- As long as we identify terms that only differ in the name of their bound variables, we can choose any variable $v \neq y$ and $v \neq x$ to evaluate $(\lambda y.xy)[x := y]$ to $\lambda v.yv$.
- In other words, we can rename $\lambda y.xy$ to $\lambda z.xz$ or $\lambda x'.xx'$ or $\lambda x_1.xx_1$, etc... But we cannot rename it to $(\lambda x.xx)$.

What about $(\lambda y.zy)[z := xy]$?

- Again here, if we simply replace the free occurrence of z in $\lambda y.zy$ by xy, we get: $\lambda y.(xy)y$ and this results in binding the occurrence of the y which was free in xy. *Incorrect*
- So, first, we rename the y of $\lambda y.zy$ to something:
  - $\neq y$ since we need another variable instead of y
  - $\neq z$ because if we rename the the y of $\lambda y.zy$ to z we get: $\lambda z.z \circ z$ and the free occurrence of $z\circ$ is now bound. *Incorrect*
  - $\neq x$ because if we rename $\lambda y.zy$ to $\lambda x.zx$, and replace the z by xy obtaining $\lambda x.(x^\circ y)x$, we bind an occurrence of x° which was free in the xy of $[z := xy]$.
- So, we take any *other* variable in $\mathcal{V}$. Say x', obtaining $\lambda x'.zx'$.
- Then, we perform the substitution $(\lambda x'.zx')[z := xy]$ which gives us $\lambda x'.(xy)x'$. No free variable of $(\lambda x'.zx')$ or xy has become bound in $\lambda x'.(xy)x'$.

- What about $(\lambda y.xy)[x := xy]$?
- Again here, if we simply replace the free occurrence of x in $\lambda y.xy$ by xy, we get: $\lambda y.(xy)y$. *Incorrect*
- So, first, we rename the y of $\lambda y.xy$ to something $\neq y$, and $\neq x$, say z obtaining $\lambda z.xz$.
- Then, we perform the substitution $(\lambda z.xz)[x := xy]$ which gives us $\lambda z.(xy)z$.
- Note that $(\lambda y.zy)[z := xy] = (\lambda z.xz)[x := xy]$. Do you know why?

Grafting

- Remember that the λ-expressions represent programs and that we evaluate these programs via the β-rule:

$$(\lambda v.A)B \to_\beta A[v := B]$$

- Remember also that taking $A[v := B]$ as grafting (i.e., as simply the repalcement of all free occurrences of v in A by B) is problematic.
- $(\lambda xz.xz)y \to_\beta (\lambda z.xz)[x := y] \equiv \lambda z.yz$ is acceptable.
- But $(\lambda xy.xy)y \to_\beta (\lambda y.xy)[x := y] \equiv \lambda y.yy$ is not acceptable.
- In $\lambda y.xy$, before replacing x by y, we need to rename the bound variable to somthing $\neq x$ and $\neq y$. say z.
- So, we define substitution to take this into account.

Substitution

- For any A, B, v, we define $A[v := B]$ to be the result of substituting B for every free occurrence of v in A, as follows:

$$
\begin{array}{lll}
1.\ v[v := B] & \equiv & B \\
2.\ v'[v := B] & \equiv & v' \qquad\qquad \text{if } v \not\equiv v' \\
3.\ (AC)[v := B] & \equiv & A[v := B]C[v := B] \\
4.\ (\lambda v.A)[v := B] & \equiv & \lambda v.A \\
5.\ (\lambda v'.A)[v := B] & \equiv & \lambda v'.A[v := B] \qquad\qquad \text{if } v \not\equiv v' \\
 & & \text{and } (v' \not\in FV(B) \text{ or } v \not\in FV(A)) \\
6.\ (\lambda v'.A)[v := B] & \equiv & \lambda v''.A[v' := v''][v := B] \qquad \text{if } v \not\equiv v' \\
 & & \text{and } (v' \in FV(B) \text{ and } v \in FV(A)) \\
 & & \text{and } v'' \not\in FV(AB)
\end{array}
$$

theorem begin

Foundations 1, F29FA1
Lecture 4

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

Substitution

- For any A, B, v, we define $A[v := B]$ to be the result of substituting B for every free occurrence of v in A, as follows:

$$
\begin{array}{lcll}
1.\ v[v := B] & \equiv & B & \\
2.\ v'[v := B] & \equiv & v' & \text{if } v \not\equiv v' \\
3.\ (AC)[v := B] & \equiv & A[v := B]C[v := B] & \\
4.\ (\lambda v.A)[v := B] & \equiv & \lambda v.A & \\
5.\ (\lambda v'.A)[v := B] & \equiv & \lambda v'.A[v := B] & \text{if } v \not\equiv v' \\
 & & \multicolumn{2}{l}{\text{and } (v' \notin FV(B) \text{ or } v \notin FV(A))} \\
6.\ (\lambda v'.A)[v := B] & \equiv & \lambda v''.A[v' := v''][v := B] & \text{if } v \not\equiv v' \\
 & & \multicolumn{2}{l}{\text{and } (v' \in FV(B) \text{ and } v \in FV(A))} \\
 & & \multicolumn{2}{l}{\text{and } v'' \notin FV(AB)}
\end{array}
$$

Examples

1. $x[x := \lambda z.z] \equiv \lambda z.z$.
2. $y[x := \lambda z.z] \equiv y$.
3. $(xz)[x := \lambda z.z] \equiv (\lambda z.z)z$.
4. $(\lambda x.x)[x := (\lambda z.z)y] \equiv \lambda x.x$.
5. - $(\lambda y.xy)[x := (\lambda z.z)x_1] \equiv \lambda y.(\lambda z.z)x_1 y$.
     Note that $y \notin FV((\lambda z.z)x_1)$.
     Hence, no free variable of $(\lambda z.z)x_1$ will become bound by λy after substitution.
   - The following is *NOT CORRECT*:
     $(\lambda y.xy)[x := (\lambda z.z)y] \equiv \lambda y.(\lambda z.z)yy$.
     The free y in $(\lambda z.z)y$ became bound in $\lambda y.(\lambda z.z)yy$.
     Give the correct substitution.

How do we find v'' in clause 6?

- So, $(\lambda y.xy)[x := (\lambda z.z)y]$ must be $\not\equiv \lambda y.(\lambda z.z)yy$.
- Note that $y \in FV((\lambda z.z)y)$ and $x \in FV(xy)$. Hence, we need to use clause 6 to do the substitution $(\lambda y.xy)[x := (\lambda z.z)y]$.
- For clarity, let us take the simpler example: $(\lambda y.xy)[x := y]$. By clause 6, we can rename the y of $(\lambda y.xy)$ to anyone of the infinite number of variables in $\mathcal{V}$ as long as as we don't rename it to x. So, we can have:
  - $(\lambda y.xy)[x := y] \equiv \lambda x'.(xy)[y := x'][x := y] \equiv \lambda x'.yx'$ or
  - $(\lambda y.xy)[x := y] \equiv \lambda y'.(xy)[y := y'][x := y] \equiv \lambda y'.yy'$ or
  - $(\lambda y.xy)[x := y] \equiv \lambda z.(xy)[y := z][x := y] \equiv \lambda z.yz$ etc.
- This creates problems. $(\lambda y.xy)[x := y]$ can be anyone of an infinite set of expressions. Which one is the official result?
- By our earlier syntactic equality, $\lambda x'.yx' \not\equiv \lambda y'.yy' \not\equiv \lambda z.yz$.

- One way to get a unique result in the last clause of the above definition would be to order the list of variables $\mathcal{V}$ and then to take v'' to be the first variable in the ordered list $\mathcal{V}$ which is different from v and v' and which occurs after all the free variables of AB.
- For example, if the ascending order in $\mathcal{V}$ is

$$x, y, z, x', y', z', x'', y'', z'', \ldots$$

- then $(\lambda y.xy)[x := y]$ can only be $(\lambda z.yz)$ since z is the first variable of the ordered list which is after all the free variables of y and x.
- This however has its own complications. So we will abandon it NOW.

- Another way is to identify terms modulo the names of their bound variables, then in clause 6 of the definition of substitution, any $v'' \notin FV(AB)$ can be taken.
- I.e., if we take $\lambda x'.yx'$ to be the same as $\lambda y'.yy'$, $\lambda z.yz$, etc., then any chosen $v'' \notin FV(AB)$ can be taken.
- This is what we will do in our course. We will identify terms modulo the names of their bound variables.
- We treat $\lambda x'.yx'$, $\lambda y'.yy'$, $\lambda z.yz$, etc. to be the same term.
- This changes our definition of syntactic identity given in lecture 2.
  Now, $\lambda x'.yx' \equiv \lambda y'.yy' \equiv \lambda z.yz$.
- We say that such terms are *equal up to the name of bound variables*. We will come back to this after defining α-reduction.

- With our assumption that terms are equal up to the name of bound variables, we will review our two examples that invoke clause 6 of substitution.
- Example 1:
  - $(\lambda y.xy)[x := y] \equiv \lambda z.yz$ (where we renamed y to z in $\lambda y.xy$).
  - We could also rename y to x_3 say, and we get: $(\lambda y.xy)[x := y] \equiv \lambda x_3.yx_3$.
- Example 2:
  - $(\lambda y.xy)[x := (\lambda z.z)y] \equiv \lambda z.(\lambda z.z)yz$ (where we renamed y to z in $\lambda y.xy$).
  - We could also rename y to x_3 say, and we get: $(\lambda y.xy)[x := (\lambda z.z)y] \equiv \lambda x_3.(\lambda z.z)yx_3$. It is cleaner to do so.

Syntactic identity revised

- The definition of syntactic identity given in Lecture 2 said that *$A \equiv B$ iff A and B are exactly the same.*
- For example, $x \equiv x$, $\lambda x.x \equiv \lambda x.x$ but $x \not\equiv y$ and $\lambda x.x \not\equiv \lambda y.y$.
- With this old definition:
  - *If $A \equiv B$ then A and B have the same meaning.*
  - But *if A and B have the same meaning then we may not necessarily have $A \equiv B$.*
  - For example, $\lambda x.x$ has the same meaning as $\lambda y.y$ but $\lambda x.x \not\equiv \lambda y.y$.
- New definition of syntactic identity: *$A \equiv B$ iff A and B are exactly the same up to the name of their bound variables (i.e. A and B only differ in the name of their bound variables)*.
- Difficult question: Is it the case now that *$A \equiv B$ if and only if A and B have the same meaning?*

Syntactic identity revised

- For example, $x \equiv x$, $\lambda x.x \equiv \lambda y.y$, but $x \not\equiv y$.
- It remains that if $AB \equiv A'B'$ then $A \equiv A'$ and $B \equiv B'$.
- If $\lambda v.A \equiv \lambda v'.A'$ then $A' \equiv A[v := v']$.
  For example, $\lambda x.x \equiv \lambda y.y$ then $y \equiv x[x := y]$.
- For each of the following pair of terms, say whether they are syntactically equivalent.
  - x and x'.
  - $\lambda xy.xy$ and $\lambda yx.yx$.
  - $\lambda xy.xy$ and $\lambda yx.xy$.
  - $\lambda y.((\lambda zx.zx)z)$ and $\lambda x.((\lambda yz.yz)z)$
  - $\lambda zy.((\lambda zx.zx)z)$ and $\lambda yx.((\lambda yz.yz)z)$

Reduction

- Three notions of reduction will be studied in this section.
- The first is α-reduction which identifies terms up to variable renaming.
- The second is β-reduction which evaluates λ-terms.
- The third is η-reduction which is used to identify functions that return the same values for the same arguments (extensionality).
- β-reduction is used in every λ-calculus, whereas η-reduction and α-reduction may or may not be used.

- Now, look at $(\lambda v'.A)$. By our assumption that terms are equivalent up to the name of their bound variables, we can rename v' to any v'' we want, as long as $v'' \not\in FV(A)$.
- For example, we can rename the y of $\lambda y.xy$ to anything, except to x, since $x \in FV(xy)$.
- We call this renaming α-reduction.
- We write this as a rule as follows:

$$\lambda v'.A \to_\alpha \lambda v''.A[v' := v''] \text{ if } v'' \not\in FV(A)$$

- Note that the condition $v'' \not\in FV(A)$ is needed to avoid making free variables into bound ones.
- For example, $\lambda y.xy \to_\alpha \lambda z.xz$ but $\lambda y.xy \not\to_\alpha \lambda x.xx$.

- But, what do we do in $(\lambda y.xy)y$? How do we rename the y of $\lambda y.xy$ to somthing else, say z?
- Also, in $\lambda x.(\lambda y.xy)y$?
- We use the so-called *compatibility rules*:
  - $\dfrac{A \to_\alpha B}{AC \to_\alpha BC}$ $\qquad$ $\dfrac{A \to_\alpha B}{CA \to_\alpha CB}$ $\qquad$ $\dfrac{A \to_\alpha B}{\lambda x.A \to_\alpha \lambda x.B}$
- So
  - $\lambda y.xy \to_\alpha \lambda z.xz$ by (α)
  - $(\lambda y.xy)y \to_\alpha (\lambda z.xz)y$ by compatibility
  - $\lambda x.(\lambda y.xy)y \to_\alpha \lambda x.(\lambda z.xz)y$ by compatibility
- This is like when you tell me that $1 + 2 = 3$.
  I can use it to get that $5 + (1 + 2) = 5 + 3$,
  that $(1 + 2) + 5 = 3 + 5$,
  that $6 \times (1 + 2) = 6\times$, etc.

Transitivity and reflexivity

- Now, look at $(\lambda y.xy)(\lambda z.z)$
- $(\lambda y.xy)(\lambda z.z) \to_\alpha (\lambda y_1.xy_1)(\lambda z.z)$
- and $(\lambda y_1.xy_1)(\lambda z.z) \to_\alpha (\lambda y_1.xy_1)(\lambda z_1.z_1)$
- So, $(\lambda y.xy)(\lambda z.z) \to_\alpha (\lambda y_1.xy_1)(\lambda z.z) \to_\alpha (\lambda y_1.xy_1)(\lambda z_1.z_1)$
- We say: $(\lambda y.xy)(\lambda z.z) \twoheadrightarrow_\alpha (\lambda y_1.xy_1)(\lambda z_1.z_1)$
- Also, any A α-reduces to itself (in zero steps): $A \twoheadrightarrow_\alpha A$.

Alpha reduction

- $\rightarrow_\alpha$ is defined to be the least compatible relation closed under the axiom:

$$(\alpha) \qquad \lambda v.A \rightarrow_\alpha \lambda v'.A[v := v'] \qquad \text{where } v' \notin FV(A)$$

- We call $\lambda v.A$ an α-redex and we say that $\lambda v.A$ α-reduces to $\lambda v'.A[v := v']$.
- $\lambda x.x \rightarrow_\alpha \lambda y.y$. The α-redex $\lambda x.x$ α-reduces to $\lambda y.y$.
- $\lambda x.xy \not\rightarrow_\alpha \lambda y.yy$.
- We define $\twoheadrightarrow_\alpha$ to be the reflexive transitive closure of $\rightarrow_\alpha$.
- $\lambda z.(\lambda x.x)x \twoheadrightarrow_\alpha \lambda z.(\lambda y.y)x$.

Exercises

- 4. Use the definition of substitution (clauses 1..6) to evaluate the following (show all the evaluation steps):
  1. $(\lambda y.x(\lambda x.x))[x := \lambda y.yx]$.
  2. $(y(\lambda z.xz))[x := (\lambda y.zy)]$.
- Do exercises 2(a) and 2(b) of test 2013.
- Other exercises you can do to practice substitution is to evaluate A[v:=B] for any A, v, B you see, showing all the substitution steps.

theorem begin

Foundations 1, F29FA1
Lecture 5

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

Alpha reduction

- $\rightarrow_\alpha$ is defined to be the least compatible relation closed under the axiom:

$$(\alpha) \qquad \lambda v.A \rightarrow_\alpha \lambda v'.A[v := v'] \qquad \text{where } v' \notin FV(A)$$

- We call $\lambda v.A$ an α-redex and we say that $\lambda v.A$ α-reduces to $\lambda v'.A[v := v']$.
- $\lambda x.x \rightarrow_\alpha \lambda y.y$. The α-redex $\lambda x.x$ α-reduces to $\lambda y.y$.
- $\lambda x.xy \not\rightarrow_\alpha \lambda y.yy$.
- We define $\twoheadrightarrow_\alpha$ to be the reflexive transitive closure of $\rightarrow_\alpha$.
- $\lambda z.(\lambda x.x)x \twoheadrightarrow_\alpha \lambda z.(\lambda y.y)x$.

- Compatibility rules for β are defined similarly to those for α.
- $$\frac{A \to_\beta B}{AC \to_\beta BC}$$
- $$\frac{A \to_\beta B}{CA \to_\beta CB}$$
- $$\frac{A \to_\beta B}{\lambda x.A \to_\beta \lambda x.B}$$

Beta reduction

- $\rightarrow_\beta$ is defined to be the least compatible relation closed under the axiom:

$$(\beta) \qquad (\lambda v.A)B \rightarrow_\beta A[v := B]$$

- We say that $(\lambda v.A)B$ is a β-redex and that $(\lambda v.A)B$ β-reduces to $A[v := B]$.
- $(\lambda x.x)(\lambda z.z) \rightarrow_\beta \lambda z.z$
- We write $\twoheadrightarrow_\beta$ for the reflexive transitive closure of $\rightarrow_\beta$.
- $(\lambda x.\lambda y.\lambda z.xz(yz))(\lambda x.x)(\lambda x.x)y \twoheadrightarrow_\beta yy$.

- Here is a lemma about the interaction of β-reduction and substitution:
  Lemma: Let $A, B, C \in \mathcal{M}$.
  1. If $A \to_\beta B$ then $C[x := A] \twoheadrightarrow_\beta C[x := B]$.
  2. If $A \to_\beta B$ then $A[x := C] \to_\beta B[x := C]$.
- Proof: 1. By induction on the structure of C.
  2. By induction on the derivation $A \to_\beta B$ using the substitution lemma.
- For example: If $A \equiv (\lambda x.x)y$, $B \equiv y$ and $C \equiv xx$, then $A \to_\beta B$ and $C[x := A] \equiv A\underline{A} \to_\beta \underline{A}B \to_\beta BB \equiv C[x := B]$ and $A[x := C] \equiv A \to_\beta B \equiv B[x := C]$.

Eta reduction

- We define compatibility for η similarly to that of β and α.
- $\to_\eta$ is defined to be the least compatible relation closed under the axiom:

$$(\eta) \qquad \lambda v.Av \to_\eta A \qquad \text{for } v \notin FV(A)$$

- When $v \notin FV(A)$, we say that $\lambda v.Av$ is an η-redex and that $\lambda v.Av$ η-reduces to A.
- $\lambda x.(\lambda z.z)x \to_\eta \lambda z.z$.
- $\lambda x.xx \not\to_\eta x$.
- We use $\twoheadrightarrow_\eta$ to denote the reflexive, transitive closure of $\to_\eta$.
- For example: $\lambda y.(\lambda x.(\lambda z.z)x)y \twoheadrightarrow_\eta \lambda z.z$.

Our reduction relations. Let $r \in \{\beta, \alpha, \eta\}$.

- Recall the three reduction axioms we have so far:

$$
\begin{array}{lll}
(\alpha) & \lambda v.A \rightarrow_\alpha \lambda v'.A[v := v'] & \text{where } v' \notin FV(A) \\
(\beta) & (\lambda v.A)B \rightarrow_\beta A[v := B] & \\
(\eta) & \lambda v.Av \rightarrow_\eta A & \text{for } v \notin FV(A)
\end{array}
$$

- $\rightarrow_r$ is the least compatible relation closed under axiom (r).
- I.e., $A \rightarrow_r B$ if and only if one of the following holds:
    - A is the lefthand side of axiom (r) and B is its righthand side.
    - $\dfrac{A_1 \rightarrow_r A_2}{A \equiv A_1 C \rightarrow_r A_2 C \equiv B}$
    - $\dfrac{A_1 \rightarrow_r A_2}{A \equiv CA_1 \rightarrow_r CA_2 \equiv B}$
    - $\dfrac{A_1 \rightarrow_r A_2}{A \equiv \lambda v.A_1 \rightarrow_r \lambda v.A_2 \equiv B}$

Examples of $\to_r$ where r is β

- $(\lambda xyz.xyz)(\lambda x.xx) \to_\beta \lambda yz.(\lambda x.xx)yz$
- $\dfrac{(\lambda xyz.xyz)(\lambda x.xx) \to_\beta \lambda yz.(\lambda x.xx)yz}{(\lambda xyz.xyz)(\lambda x.xx)(\lambda x.x) \to_\beta (\lambda yz.(\lambda x.xx)yz)(\lambda x.x)}$
- $\dfrac{(\lambda xyz.xyz)(\lambda x.xx) \to_\beta \lambda yz.(\lambda x.xx)yz}{(\lambda x.x)((\lambda xyz.xyz)(\lambda x.xx)) \to_\beta (\lambda x.x)(\lambda yz.(\lambda x.xx)yz)}$
- $\dfrac{(\lambda xyz.xyz)(\lambda x.xx) \to_\beta \lambda yz.(\lambda x.xx)yz}{\lambda x'.(\lambda xyz.xyz)(\lambda x.xx) \to_\beta \lambda x'.\lambda yz.(\lambda x.xx)yz}$

Examples of $\rightarrow_r$ where r is β

- Note that $(\lambda xyz.xyz)(\lambda x.xx) \not\rightarrow_\beta (\lambda xyz.xyz)(\lambda x.xx)$.
- This is why we introduce a reflexive relation $\twoheadrightarrow_\beta$ which contains $\rightarrow_\beta$ and where $A \twoheadrightarrow_\beta A$ for any A.
- Hence, $(\lambda xyz.xyz)(\lambda x.xx) \twoheadrightarrow_\beta (\lambda xyz.xyz)(\lambda x.xx)$.
- Note also that, even though $(\lambda xyz.xyz)(\lambda x.xx) \rightarrow_\beta (\lambda yz.(\lambda x.xx)yz) \rightarrow_\beta (\lambda yz.yyz)$, $(\lambda xyz.xyz)(\lambda x.xx) \not\rightarrow_\beta (\lambda yz.yyz)$.
- This is why we also make $\twoheadrightarrow_\beta$ transitive.
- I.e., if $A \twoheadrightarrow_\beta B$ and $B \twoheadrightarrow_\beta C$ then $A \twoheadrightarrow_\beta C$.
- Hence, $(\lambda xyz.xyz)(\lambda x.xx) \twoheadrightarrow_\beta (\lambda yz.yyz)$.

- So, for any $r \in \{\beta, \alpha, \eta\}$, we define $\twoheadrightarrow_r$ to be the reflexive transitive closure of $\rightarrow_r$. This means that:
  - $\dfrac{A \rightarrow_r B}{A \twoheadrightarrow_r B}$
  - $A \twoheadrightarrow_r A$
  - $\dfrac{A \twoheadrightarrow_r B \qquad B \twoheadrightarrow_r C}{A \twoheadrightarrow_r C}$
- *Lemma*
  - $\twoheadrightarrow_r$ is compatible:
    $\dfrac{A \twoheadrightarrow_r B}{AC \twoheadrightarrow_r BC} \qquad \dfrac{A \twoheadrightarrow_r B}{CA \twoheadrightarrow_r CB} \qquad \dfrac{A \twoheadrightarrow_r B}{\lambda x.A \twoheadrightarrow_r \lambda x.B}$
  - If $A \twoheadrightarrow_r B$ then there are $A_1, A_2, \ldots A_n$ where $n \geq 0$ and $A \equiv A_1 \rightarrow_r A_2 \rightarrow_r \cdots \rightarrow_r A_n \equiv B$.

theorem begin

Foundations 1, F29FA1
Lecture 6

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

- So, for any $r \in \{\beta, \alpha, \eta\}$, we define $\twoheadrightarrow_r$ to be the reflexive transitive closure of $\rightarrow_r$. This means that:
  - $$\frac{A \rightarrow_r B}{A \twoheadrightarrow_r B}$$
  - $A \twoheadrightarrow_r A$
  - $$\frac{A \twoheadrightarrow_r B \qquad B \twoheadrightarrow_r C}{A \twoheadrightarrow_r C}$$
- *Lemma*
  - $\twoheadrightarrow_r$ is compatible:
    $$\frac{A \twoheadrightarrow_r B}{AC \twoheadrightarrow_r BC} \qquad \frac{A \twoheadrightarrow_r B}{CA \twoheadrightarrow_r CB} \qquad \frac{A \twoheadrightarrow_r B}{\lambda x.A \twoheadrightarrow_r \lambda x.B}$$
  - If $A \twoheadrightarrow_r B$ then there are $A_1, A_2, \ldots A_n$ where $n \geq 0$ and $A \equiv A_1 \rightarrow_r A_2 \rightarrow_r \cdots \rightarrow_r A_n \equiv B$.

- You can think of $\rightarrow_r$ as computation rules. When A computes to B, it is not necessarily the case that B computes to A.
- E.g., $(\lambda xyz.xyz)(\lambda x.xx) \rightarrow_\beta (\lambda yz.(\lambda x.xx)yz)$.
  But, $(\lambda yz.(\lambda x.xx)yz) \not\rightarrow_\beta (\lambda xyz.xyz)(\lambda x.xx)$.
- We introduce symmetry. We define $=_r$ to be the smallest relation which satsifies:
  - reflexive: $A =_r A$
  - transitive: $\dfrac{A =_r B \qquad B =_r C}{A =_r C}$
  - symmetric: $\dfrac{A =_r B}{B =_r A}$
  - contains $\rightarrow_r$: $\dfrac{A \rightarrow_r B}{A =_r B}$

- If $A =_r B$, we say that A and B are r-convertible.
- *Lemma:* $=_r$ is compatibe.
  - $\dfrac{A =_r B}{AC =_r BC}$ $\quad$ $\dfrac{A =_r B}{CA =_r CB}$ $\quad$ $\dfrac{A =_r B}{\lambda x.A =_r \lambda x.B}$
- Recall that $A \equiv B$ iff A and B are syntactically identical up to the name of their bound variables. Hence, $A \equiv B$ iff $A =_\alpha B$.

- If $A \rightarrow_{\beta} B$ or $A \rightarrow_{\eta} B$, we write $A \rightarrow_{\beta\eta} B$.
- We define $\twoheadrightarrow_{\beta\eta}$ to be the reflexive transitive closure of $\rightarrow_{\beta\eta}$.
- We define $=_{\beta\eta}$ to be the reflexive, symmetric and transitive closure of $\rightarrow_{\beta\eta}$.
- Again, $\twoheadrightarrow_{\beta\eta}$ and $=_{\beta\eta}$ are compatible.
- η-conversion equates two terms that have the same behaviour as functions and implies extensionality. I.e., for any $v \notin FV(A)$ and $v \notin FV(B)$, if $Av =_{\beta\eta} Bv$ then $A =_{\beta\eta} B$.

In Normal Form (Fully Evaluated)

- We say that A is in β-normal form, iff there are no β-redexes in A.
- $\lambda x.zx$ is in β-normal form but $(\lambda yx.yx)z$ is not.
- We say that A is in η-normal form, iff there are no η-redexes in A.
- $\lambda x.zx$ is not in η-normal form. But, $\lambda x.xx$ is in η-normal form.
- We say that A is in $\beta\eta$-normal form, iff there are no β-redexes and no η-redexes in A.
- $\lambda x.xx$ is in $\beta\eta$-normal form.
- Let $r \in \{\beta, \eta, \beta\eta\}$. Then, A is in r-normal form iff there are no r-redexes in A. I.e., there is no B such that $A \rightarrow_r B$.

Has Normal Form (Can be Fully Evaluated)

- Let $r \in \{\beta, \eta, \beta\eta\}$.
- We say that A has an r-normal form B iff $A =_r B$ and B is in r-normal form.
- For example, $(\lambda xyz.xyz)(\lambda x.xx)(\lambda x.x)x$ is not in β-normal form, but it has a β-normal form x.
- Not all terms have normal forms.
- $(\lambda x.xx)(\lambda x.xx)$ is not in β-normal form and there is no B such that $(\lambda x.xx)(\lambda x.xx) =_\beta B$ and B is in β-normal form.
- $(\lambda x.xx)(\lambda x.xx)$ does not have a β-normal form.
- We will see this later. For now, note that: $(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx)$.......

Weakly normalising and Strongly normalising terms

- Let $r \in \{\beta, \eta, \beta\eta\}$.
- A term A is *strongly r-normalising* (*always r-terminates*) iff there are no infinite r-reduction sequences starting at A.
- Example: $(\lambda x.xx)(\lambda x.xx)$ is not strongly β-normalising: $(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx)$.......
- A term A is *weakly r-normalising* (*weakly r-terminates*) iff there is a B in r-normal form such that $A \twoheadrightarrow_r B$.
- Example: $(\lambda x.xx)(\lambda y.y)z$ is weakly β-normalising: $(\lambda x.xx)(\lambda y.y)z \twoheadrightarrow_\beta z$.
- Is $(\lambda x.xx)(\lambda x.xx)$ weakly β-normalising?
- Is $(\lambda z.y)((\lambda x.xx)(\lambda x.xx))$ weakly β-normalising?

Propeties of terms

- Let $r \in \{\beta, \eta, \beta\eta\}$.
- *Lemma:* If A is strongly r-normalising then A is weakly r-normalising and A has an r-normal form.
- Does every expression have a r-normal form?
  can we keep r-reducing an expression until we reach an r-normal form?
  Is every expression weakly r-normalising?
  Is every expression strongly r-normalising?
- Recall that an expression A is weakly r-normalising iff $A \twoheadrightarrow_r B$ where B is in r-normal form.
- So, if $A \twoheadrightarrow_r B_1$ and $A \twoheadrightarrow_r B_2$ where B_1 and B_2 are in r-normal form, is it the case that $B_1 \equiv B_2$?
- Are r-normal forms unique? Are values of programs unique?

- Not all expressions have β-normal forms.
- If an r-normal form exists it is unique for $r \in \{\beta, \eta, \beta\eta\}$.
- The order of reduction will affect our reaching of a normal form of the expression.
- Sometime, a term may have a normal form, but we may not find this normal form if we use a reduction path which does not terminate.
- Sometime, the choice of redexes to be reduced does not affect the termination of our computation. Sometime, this choice may lead our computation to loop.
- There is a reduction strategy however which will terminate and find the final value (if such a value exists).

Exercises

- 5. β-reduce the following term until there are no more β-redexes (show all the reduction steps): $(\lambda xyz.xyz)(\lambda x.xx)(\lambda x.x)x$
- 6. Reduce $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x)$ until no β- or η-redexes remain (show all the reduction steps).
- 7. Show that $\lambda zx.(\lambda y.y)x \rightarrow_\eta \lambda zy.y$.
- 8. Is $(\lambda z.y)((\lambda x.xx)(\lambda x.xx))$ weakly β-normalising? Is it strongly β-normalising? Explain your answer.

Foundations 1, F29FA1
Lecture 7

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

Propeties of terms

- Let $r \in \{\beta, \eta, \beta\eta\}$.
- *Lemma:* If A is strongly r-normalising then A is weakly r-normalising and A has an r-normal form.
- Does every expression have a r-normal form?
  can we keep r-reducing an expression until we reach an r-normal form?
  Is every expression weakly r-normalising?
  Is every expression strongly r-normalising?
- Recall that an expression A is weakly r-normalising iff $A \twoheadrightarrow_r B$ where B is in r-normal form.
- So, if $A \twoheadrightarrow_r B_1$ and $A \twoheadrightarrow_r B_2$ where B_1 and B_2 are in r-normal form, is it the case that $B_1 \equiv B_2$?
- Are r-normal forms unique? Are values of programs unique?

We will see that:

- Not all expressions have β-normal forms.
- If an r-normal form exists it is unique for $r \in \{\beta, \eta, \beta\eta\}$.
- The order of reduction will affect our reaching of a normal form of the expression.
- Sometime, a term may have a normal form, but we may not find this normal form if we use a reduction path which does not terminate.
- Sometime, the choice of redexes to be reduced does not affect the termination of our computation. Sometime, this choice may lead our computation to loop.
- There is a reduction strategy however which will terminate and find the final value (if such a value exists).

- $\underline{(\lambda x.xx)(\lambda x.xx)}$ is not weakly β-normalising *you need to prove why it is not* (and hence is not strongly β-normalising).
- We can reduce in different orders and still preserve the values:
  $(\lambda y.\underline{(\lambda x.x)(\lambda z.z)})xy \rightarrow_\beta \underline{(\lambda y.\lambda z.z)x}y \rightarrow_\beta \underline{(\lambda z.z)y} \rightarrow_\beta y$ and
  $\underline{(\lambda y.(\lambda x.x)(\lambda z.z))x}y \rightarrow_\beta (\underline{(\lambda x.x)(\lambda z.z)})y \rightarrow_\beta \underline{(\lambda z.z)y} \rightarrow_\beta y$
- We omit the word *weakly*. So, when we say β-normalising, we mean weakly β-normalising.
- A term may be β-normalising but not strongly β-normalising:
  $\underline{(\lambda y.z)((\lambda x.xx)(\lambda x.xx))} \rightarrow_\beta z$ yet
  $(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \rightarrow_\beta (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \rightarrow_\beta \ldots$
- A term may grow after reduction:

$$\begin{aligned}
\underline{(\lambda x.xxx)(\lambda x.xxx)} \quad & \rightarrow_\beta \quad \underline{(\lambda x.xxx)(\lambda x.xxx)}(\lambda x.xxx) \\
& \rightarrow_\beta \quad \underline{(\lambda x.xxx)(\lambda x.xxx)}(\lambda x.xxx)(\lambda x.xxx) \\
& \rightarrow_\beta \quad \ldots
\end{aligned}$$

- Over expressions whose evaluation does not terminate, there is little we can do, so let us restrict our attention to those expressions whose evaluation terminates.
- β- and η-reduction can be seen as defining the steps that can be used for evaluating expressions to values.
- The values are intended to be themselves terms that cannot be reduced any further.
- Luckily, all orders lead to the same value/normal form (if it exists) of the expression for *r*-reduction where $r \in \{\beta, \beta\eta\}$.
- That is, if an expression *r*-reduces in two different ways to two values/*r*-normal forms, then those values are the same (up to α-conversion).

- Here are some ways to reduce $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x)$.
- In all cases, the same final answer is obtained.
- $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \rightarrow_\beta (\lambda yz.\underline{(\lambda x.x)z}(yz))(\lambda x.x) \rightarrow_\beta \underline{(\lambda yz.z(yz))(\lambda x.x)} \rightarrow_\beta \lambda z.z(\underline{(\lambda x.x)z}) \rightarrow_\beta \lambda z.zz$.
- $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \rightarrow_\beta \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)} \rightarrow_\beta \lambda z.\underline{(\lambda x.x)z}((\lambda x.x)z) \rightarrow_\beta \lambda z.z(\underline{(\lambda x.x)z}) \rightarrow_\beta \lambda z.zz$.
- $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \rightarrow_\beta \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)} \rightarrow_\beta \lambda z.(\lambda x.x)z(\underline{(\lambda x.x)z}) \rightarrow_\beta \lambda z.\underline{(\lambda x.x)z}z \rightarrow_\beta \lambda z.zz$.

Church-Rosser (CR)

Let $r \in \{\beta, \beta\eta\}$

- We would like that if A r-reduces to B and to C, then B and C r-reduce to the same term D.
- Luckily, the λ-calculus satisfies this property which is called the Church-Rosser property.
- *Theorem:* $\forall A, B, C \in \mathcal{M},\ \exists D \in \mathcal{M},$ such that: $(A \twoheadrightarrow_r B \wedge A \twoheadrightarrow_r C) \Rightarrow (B \twoheadrightarrow_r D \wedge C \twoheadrightarrow_r D)$.

- The Church-Rosser theorem says that the results of reductions do not depend on the order in which they are done:

$$
\begin{tikzcd}
 & A \arrow[dl, "r"'] \arrow[dr, "r"] & \\
B \arrow[dr, "r"'] & & C \arrow[dl, "r"] \\
 & D &
\end{tikzcd}
$$

- In arithmetic, you can think of this as follows:

$$\begin{tikzcd}[column sep=small] & 1+2+3 \arrow[dl, "+"'] \arrow[dr, "+"] & \\ 3+3 \arrow[dr, "+"'] & & 1+5 \arrow[dl, "+"] \\ & 6 & \end{tikzcd}$$

- In λ-calculus:

$$\begin{tikzcd}[column sep=small] & (\lambda z.(\lambda x.xx)(\lambda x.x)z)x \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\ (\lambda x.xx)(\lambda x.x)x \arrow[dr, "\beta"'] & & (\lambda z.(\lambda x.x)(\lambda x.x)z)x \arrow[dl, "\beta"] \\ & (\lambda x.x)(\lambda x.x)x & \end{tikzcd}$$

- We can also have

$$\begin{tikzcd}[column sep=small]
 & (\lambda z.(\lambda x.xx)(\lambda x.x)z)x \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\
(\lambda x.xx)(\lambda x.x)x \arrow[dr, "\beta"'] & & (\lambda z.(\lambda x.x)(\lambda x.x)z)x \arrow[dl, "\beta"] \\
 & (\lambda x.x)x &
\end{tikzcd}$$

- and

$$\begin{tikzcd}[column sep=small]
 & (\lambda z.(\lambda x.xx)(\lambda x.x)z)x \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\
(\lambda x.xx)(\lambda x.x)x \arrow[dr, "\beta"'] & & (\lambda z.(\lambda x.x)(\lambda x.x)z)x \arrow[dl, "\beta"] \\
 & x &
\end{tikzcd}$$

Corollaries of CR

- *Programs have unique values:* If $A \twoheadrightarrow_\beta B$ and $A \twoheadrightarrow_\beta C$ where B and C are in β-normal forms, then $B \equiv C$.

$$\begin{tikzcd} & A \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\ B & & C \end{tikzcd}$$

Corollaries of CR Continued

- *Equal programs have the same value:* If $A =_\beta B$ then there is a C such that $A \twoheadrightarrow_\beta C$ and $B \twoheadrightarrow_\beta C$.

$$
\begin{tikzcd}
A \arrow[dr, two heads, "\beta"'] & =_\beta & B \arrow[dl, two heads, "\beta"] \\
& C &
\end{tikzcd}
$$

Corollaries of CR continued

- *A program reduces to its β-normal form:* If A has a β-normal form B then $A \twoheadrightarrow_\beta B$.
- *Normal forms are unique:* If A has two β-normal forms B_1 and B_2 then $B_1 \equiv B_2$.
- If A is in β-normal form, and if $A =_\beta B$, then $B \twoheadrightarrow_\beta A$.
- If $A =_\beta B$ then either both A and B have the same β-normal form, or neither one has a β-normal form.
- *λ-calculus is consistent:* There are A, B such that $A \neq_\beta B$.
    - **Proof:** Let $A \equiv \lambda x.x$ and $B \equiv \lambda xy.y$. If $A =_\beta B$ then by unicity of normal forms, $A \equiv B$, but this is not the case. Hence $A \neq_\beta B$.

- So far we have answered two important questions.
    1. Terms evaluate to unique values.
    2. The λ-calculus is not trivial in the sense that it has more than one element.
- Let us recall however that a term may have a β-normal form yet the evaluation order we use may not find this β-normal form. For example, remember $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$
- Hence the question now is: given a term that has a β-normal form, can we find this β-normal form?
- This is an important question because to be able to compute with the λ-caluclus, we must be able to find the β-normal form of a term if it exists. *We must be able to find the value of a program that terminates.*
- Luckily we have a positive result to this question.

theorem begin

Foundations 1, F29FA1
Lecture 8

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

- So far we have answered two important questions.
  1. Terms evaluate to unique values.
  2. The λ-calculus is not trivial in the sense that it has more than one element.
- Let us recall however that a term may have a β-normal form yet the evaluation order we use may not find this β-normal form. For example, remember $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$
- Hence the question now is: given a term that has a β-normal form, can we find this β-normal form?
- This is an important question because to be able to compute with the λ-caluclus, we must be able to find the β-normal form of a term if it exists. *We must be able to find the value of a program that terminates.*
- Luckily we have a positive result to this question.

- Positive result: if a term has a β-normal form then there is a reduction strategy that finds this β-normal form.
- The positive result is given by *the normalisation theorem* given later which tells us that blind alleys in a reduction can be avoided by reducing the kind of *leftmost* β-redex whose beginning λ is as far to the left as possible.
- For example, if we β-reduce the underlined (leftmost) redex $\underline{(\lambda y.z)((\lambda x.xx)(\lambda x.xx))}$ we terminate with the β-normal form y.
- If however, we β-reduce the underlined (righmost) redex $(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)})$ we will loop forever and we will not find the β-normal form y.

Directions of redexes (left or right)

- Let A have the two β-redexes R_1, R_2. We say that R_1 is to the left (resp. right) of R_2 in A if the λ of R_1 is to the left (resp. right) of the λ of R_2 in A.
- For example, Let $A \equiv (\lambda y.(\lambda z.z)x)((\lambda xy.x)x)$.
  Let $R \equiv A$, $R_1 \equiv (\lambda z.z)x$ and $R_2 \equiv (\lambda xy.x)x$.
  R is to the left of R_1 and R_2. R_1 is to the left of R_2.
- $\underline{(\lambda y.z)((\lambda x.xx)(\lambda x.xx))}$ is to the left of $\underline{(\lambda x.xx)(\lambda x.xx)}$ in $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$.

Standardisation theorem

- A reduction path $A_0 \stackrel{R_0}{\to}_\beta A_1 \stackrel{R_1}{\to}_\beta A_2 \ldots$ is *standard* if for any pair (R_i, R_{i+1}), the λ of the redex R_{i+1} comes from a λ in A_i which is to the right of the λ of R_i in A_i.
- $\underline{(\lambda x.(\lambda y.xy)z)(\lambda z.z)} \to_\beta \underline{(\lambda y.(\lambda z.z)y)z} \to_\beta \underline{(\lambda z.z)z} \to_\beta z$ is standard.
- $(\lambda x.\underline{(\lambda y.xy)z})(\lambda z.z) \stackrel{\bullet}{\to}_\beta \underline{(\lambda x.xz)(\lambda z.z)} \to_\beta \underline{(\lambda z.z)z} \to_\beta z$ is not standard.
- $\underline{(\lambda x.(\lambda y.xy)z)(\lambda z.z)} \to_\beta (\lambda y.\underline{(\lambda z.z)y})z \stackrel{\bullet}{\to}_\beta \underline{(\lambda y.y)z} \to_\beta z$ is not standard.
- In a standard path, one reduces from left to right.
- *Standardisation theorem:* If $A \twoheadrightarrow_\beta B$ then there is a standard reduction path from A to B.

Normalisation theorem

- The leftmost β-reduction strategy is the reduction strategy that always β-reduces in a term A, the redex that is to the left of all other redexes in A.
- A reduction strategy *strat* is β-normalising iff, for any term A which has a β-normal form, β-reducing A using *strat* will lead to the β-normal of A.
- *Normalisation Theorem:* The leftmost β-reduction strategy is β-normalising.

Exercises

9. For each of the following terms, find its β-normal form if it exists or show that it does not have a β-normal form.

   1. $(\lambda x.xxx)(\lambda x.xx)(\lambda x.x)$
   2. $(\lambda x.xxx)(\lambda x.x)$

10. For each of the following terms, say whether it is strongly β-normalising, weakly β-normalising and whether it has a β-normal form (and in this case, give the β-normal form). In all cases, you must either prove your answer or give a counterexample.

1. $(\lambda x.xxxx)(\lambda x.xxx)((\lambda x.xx)(\lambda x.x))$
2. $(\lambda x.xxxx)(\lambda x.xxx)(\lambda x.xx)(\lambda x.x)$
3. $(\lambda x.xxxx)((\lambda x.xxx)(\lambda x.xx))(\lambda x.x)$

theorem begin

Foundations 1: F29FA1
Lecture 13

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

Leftmost Outermost

- *Leftmost outermost β-redex* The leftmost outermost β-redex of a term is the β-redex whose λ is the leftmost λ of the term.
  - $lmo(v) =$ undefined
  - $lmo(\lambda v.A) = lmo(A)$
  - $lmo(AB) = AB$ if AB is a β-redex
  - $lmo(AB) = lmo(A)$ if AB is not a β-redex and $lmo(A)$ is defined
  - $lmo(AB) = lmo(B)$ if AB is not a β-redex and $lmo(A)$ is undefined
- $\underline{(\lambda z.z((\lambda x.x)z))((\lambda x.x)(\lambda y.x)z)} \to_{\beta,lmo}$
  $(\underline{(\lambda x.x)(\lambda y.x)}z)((\lambda x.x)(((\lambda x.x)(\lambda y.x))z)) \to_{\beta,lmo}$
  $\underline{(\lambda y.x)z}((\lambda x.x)((\lambda x.x)(\lambda y.x)z)) \to_{\beta,lmo}$
  $x(\underline{(\lambda x.x)((\lambda x.x)(\lambda y.x)z)}) \to_{\beta,lmo}$
  $x(\underline{(\lambda x.x)(\lambda y.x)}z) \to_{\beta,lmo}$
  $x(\underline{(\lambda y.x)z}) \to_{\beta,lmo} xx$

Rightmost

- *Rightmost β-redex* The rightmost β-redex of a term is the β-redex whose λ is the rightmost λ of the term.
  - $rm(v) =$ undefined
  - $rm(\lambda v.A) =_{def} rm(A)$
  - $rm(AB) = rm(B)$ if $rm(B)$ is defined
  - $rm(AB) = AB$ if $rm(B)$ is undefined and AB is a β-redex
  - $rm(AB) = rm(A)$ if $rm(B)$ is undefined and AB is not a β-redex
- $(\lambda z.z((\lambda x.x)z))(\underline{(\lambda x.x)(\lambda y.x)}z) \rightarrow_{\beta,rm}$
  $(\lambda z.z((\lambda x.x)z))(\underline{(\lambda y.x)z}) \rightarrow_{\beta,rm}$
  $\underline{(\lambda z.z((\lambda x.x)z))x} \rightarrow_{\beta,rm}$
  $x(\underline{(\lambda x.x)x}) \rightarrow_{\beta,rm} xx$
- Note that reducing $(\lambda z.z((\lambda x.x)z))((\lambda x.x)(\lambda y.x)z)$ using the righmost strategy, gives a reduction path which is shorter than reducing it using the leftmost outermost strategy.

Leftmost outermost always reaches a β-normal form if it exists whereas rightmost may not

- The leftmost outermost redex of $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$ is the whole term itself and not $((\lambda x.xx)(\lambda x.xx))$.
- The rightmost redex of $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$ is $((\lambda x.xx)(\lambda x.xx))$.
- Recall that $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$ has a β-normal form z.
- If we use the leftmost outermost strategy, we can reach this β-normal form. $\underline{(\lambda y.z)((\lambda x.xx)(\lambda x.xx))} \to_{\beta,lmo} z$
- If we use the rightmost strategy, we will never reach the β-normal form. We will instead loop:
  $(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_{\beta,rm}$
  $(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_{\beta,rm}$
  $(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_{\beta,rm} \cdots$

Leftmost outermost leads to longer reductions paths than rightmost

- $\underline{(\lambda x.xxxx)((\lambda y.y)z)} \to_{\beta,lmo}$
  $\underline{((\lambda y.y)z)}((\lambda y.y)z)((\lambda y.y)z)((\lambda y.y)z) \to_{\beta,lmo}$
  $z\underline{((\lambda y.y)z)}((\lambda y.y)z)((\lambda y.y)z) \to_{\beta,lmo}$
  $zz\underline{((\lambda y.y)z)}((\lambda y.y)z) \to_{\beta,lmo}$
  $zzz((\lambda y.y)z) \to_{\beta,lmo}$
  $zzzz$
- $(\lambda x.xxxx)(\underline{(\lambda y.y)z}) \to_{\beta,rm}$
  $\underline{(\lambda x.xxxx)z} \to_{\beta,rm}$
  $zzzz.$

Head β-normal forms

- A is in head β-normal form if and only if $A \equiv \lambda x_1 x_2 .. x_n . y A_1 A_2 .. A_m$ where $n, m \geq 0$.
  Note that $A_1, A_2, .. A_m$ may still have β-redexes.
- Example: $\lambda x_1 x_2 . z((\lambda x.x)y)(\lambda x.x)$ is in head β-normal form.
- Note that this term still has a β-redex $(\lambda x.x)y$.
- We reach the head β-normal by using the head reduction strategy which always reduces the head β-redex until no head β-redex exists.

- The head β-redex is defined as follows:
  - $h(v) =$ undefined
  - $h(\lambda v.A) = h(A)$
  - $h(AB) = AB$ if AB is a β-redex
  - $h(AB) = h(A)$ if AB is not a β-redex and $h(A)$ is defined
  - $h(AB) =$ undefined if AB is not a β-redex and $h(A)$ is undefined
- $\underline{(\lambda x.xxxx)((\lambda y.y)z)} \to_{\beta,h}$ $(\underline{(\lambda y.y)z})((\lambda y.y)z)((\lambda y.y)z)((\lambda y.y)z) \to_{\beta,h}$ $z((\lambda y.y)z)((\lambda y.y)z)((\lambda y.y)z)$

Call by Name

- The call by name reduction strategy reduces the leftmost outermost redex, but not inside abstractions.
- Under the call by name strategy, abstractions are normal forms.
- The call by name reduction strategy always reduces the redex found by the function n:
  - $n(v) =$ undefined
  - $n(\lambda v.A) =$ undefined
  - $n(AB) = AB$ if AB is a β-redex
  - $n(AB) = n(A)$ if AB is not a β-redex and $n(A)$ is defined
  - $n(AB) = n(B)$ if AB is not a β-redex and $n(A)$ is undefined
- $(\lambda x.\lambda y.(\lambda z.z)xy)((\lambda x.x)x') \to_{\beta,n}$ $\lambda y.(\lambda z.z)((\lambda x.x)x')y$

Call by Leftmost and Value

- The call by leftmost and value reduction strategy reduces the leftmost outermost redex, but where the argument is a value and where no reductions take place inside abstractions.
- Under the call by leftmost and value strategy, abstractions are values.
- The call by leftmost and value reduction strategy always reduces the redex found by the function lv:
  - $lv(v) =$ undefined
  - $lv(\lambda v.A) =$ undefined
  - $lv(AB) = lv(B)$ if AB is a β-redex and B has a β-redex
  - $lv(AB) = AB$ if AB is a β-redex and B does not have a β-redex
  - $lv(AB) = lv(A)$ if AB is not a β-redex and $lv(A)$ is defined
  - $v(AB) =$ undefined if AB is not a β-redex and $lv(A)$ is undefined

- $(\lambda x.\lambda y.(\lambda z.z)xy)(\underline{(\lambda x.x)x'}) \rightarrow_{\beta,lv}$
  $\underline{(\lambda x.\lambda y.(\lambda z.z)xy)x'} \rightarrow_{\beta,lv}$
  $\lambda y.(\lambda z.z)x'y$
- $(\lambda x.xx((\lambda x.x)x))(\underline{(\lambda y.y)z}) \rightarrow_{\beta,lv}$
  $\underline{(\lambda x.xx((\lambda x.x)x))z} \rightarrow_{\beta,lv}$
  $zz((\lambda x.x)z)$

Call by Rightmost and Value

- The call by rightmost and value reduction strategy reduces the rightmost redex , but where the argument and the function are values
  - $rmv(v) =$ undefined
  - $rmv(\lambda v.A) =_{def} rmv(A)$
  - $rmv(AB) = rmv(B)$ if $rmv(B)$ is defined
  - $rmv(AB) = rmv(A)$ if $rmv(B)$ is undefined and $rmv(A)$ is defined
  - $rmv(AB) = AB$ if $rmv(B)$ and $rmv(A)$ are undefined and AB a β-redex
  - $rmv(AB) =$ undefined if AB has no β-redex.

- $(\lambda z.z((\lambda x.x)z))(\underline{(\lambda x.x)(\lambda y.x)}z) \to_{\beta,rmv}$
  $(\lambda z.z((\lambda x.x)z))(\underline{(\lambda y.x)z}) \to_{\beta,rmv}$
  $(\lambda z.z(\underline{(\lambda x.x)z}))x \to_{\beta,rmv}$
  $\underline{(\lambda z.zz)x} \to_{\beta,rmv}$
  xx

de Bruijn indices

- De Bruijn noted that due to the fact that terms as $\lambda x.x$ and $\lambda y.y$ are the *same*, one can find a λ-notation modulo α-conversion.
- Following de Bruijn, one can abandon variables and use indices instead.
- The idea of de Bruijn indices is to remove all the variable indices of the λ's and to replace their occurrences in the body of the term by the number which represents how many λ's one has to cross before one reaches the λ binding the particular occurrence at hand.

- $\lambda x.x$ is replaced by $\lambda 1$. That is, x is removed, and the x of the body x is replaced by 1 to indicate the λ it refers to.
- $\lambda x.\lambda y.xy$ is replaced by $\lambda\lambda 21$. That is, the x and y of λx and λy are removed whereas the x and y of the body xy are replaced by 2 and 1 respectively in order to refer back to the λs that bind them.
- Similarly, $\lambda z.(\lambda y.y(\lambda x.x))(\lambda x.xz)$ is replaced by $\lambda(\lambda 1(\lambda 1))(\lambda 12)$.

- Note that the above terms are all closed.
- What do we do if we had a term that has free variables?
- For example, how do we write $\lambda x.xz$ using de Bruijn's indices?
- In the presence of free variables, a *free variable list* which orders the variables must be assumed.
- For example, assume we take $x, y, z, \ldots$ to be the free variable list where x comes before y which is before z, etc.
- Then, in order to write terms using de Bruijn indices, we use the same procedure above for all the bound variables. For a free variable however, say z, we count as far as possible the λ's in whose scope z is, and then we continue counting in the free variable list using the order assumed.

- $\lambda x.xz$ translates into $\lambda 14$.
- $(\lambda x.xz)y$ translates into $(\lambda 14)2$.
- $(\lambda x.xz)x$ translates into $(\lambda 14)1$.

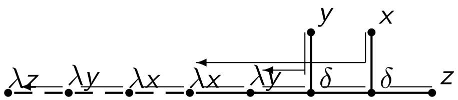
The syntax of the λ-calculus with de Bruijn indices

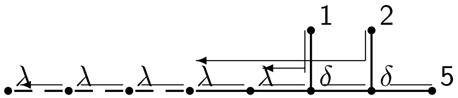
- We define Λ, the *set of terms with de Bruijn indices*, as follows:

$$\Lambda ::= \mathbb{N} \mid (\Lambda\Lambda) \mid (\lambda\Lambda)$$

- We use similar notational conventions as before:
  - Functional application associates to the left. So *ABC denotes $((AB)C)$.*
  - The body of a λ is anything that comes after it. So, *instead of $(\lambda(A_1 A_2 \ldots A_n))$, we write $\lambda A_1 A_2 \ldots A_n$.*
- Note here that we cannot compress a sequence of λ's to one. $\lambda\lambda 12$ is not the same as $\lambda 12$. The first is $\lambda z.\lambda y.yz$ and the second is $\lambda y.yx$.

The trees of terms: $\lambda x.\lambda y.zxy$ and $\lambda\lambda 521$





- Assume our variables are ordered as $x, y, z, x', y', z', \cdots$.
- Using the ω function given in the DATA SHEET, translate $M \equiv \lambda xy.xx$ and $N \equiv \lambda xyx'.xzx'$ and $MNxy'$ into a corresponding term with de Bruijn indices showing all the steps used in the translation.
- $\omega(M) = \omega_{[]}(\lambda xy.xx) = \lambda\omega_{[x]}(\lambda y.xx) = \lambda\lambda\omega_{[y,x]}(xx) = \lambda\lambda\omega_{[y,x]}(x)\omega_{[y,x]}(x) = \lambda\lambda 2\,2$.
- $\omega(N) = \omega(\lambda xyx'.xzx') = \omega_{[x,y,z]}(\lambda xyx'.xzx') = \lambda\omega_{[x,x,y,z]}(\lambda yx'.xzx') = \lambda\lambda\omega_{[y,x,x,y,z]}(\lambda x'.xzx') = \lambda\lambda\lambda\omega_{[x',y,x,x,y,z]}(xzx') = \lambda\lambda\lambda 3\,6\,1$.
- $\omega(MNy') = \omega_{[x,y,z,x',y']}(MNy') =$
  $\omega_{[x,y,z,x',y']}(MN)\omega_{[x,y,z,x',y']}(y') =$
  $\omega_{[x,y,z,x',y']}(M)\omega_{[x,y,z,x',y']}(N)\omega_{[x,y,z,x',y']}(y') =$
  $\omega(M)\omega(N)\,5 =$
  $(\lambda\lambda 2\,2)(\lambda\lambda\lambda 3\,6\,1)\,5$.

What about $\lambda z.y$?

- $\omega(\lambda z.y) = \omega_{[x,y]}(\lambda z.y) = \lambda\omega_{[z,x,y]}(y) = \lambda 3$.
- Also, $\omega(\lambda x.y) = \omega_{[x,y]}(\lambda x.y) = \lambda\omega_{[x,x,y]}(y) = \lambda 3$.

How do you define ω'?

- For $[x_1, \cdots, x_n]$ a list (not a set) of variables, we define $\omega'_{[x_1,\cdots,x_n]} : \mathcal{M}' \mapsto \Lambda'$ inductively by:
  $\omega'_{[x_1,\cdots,x_n]}(v_i) = \min\{j : v_i \equiv x_j\}$
  $\omega'_{[x_1,\cdots,x_n]}(\langle B\rangle A) = \langle\omega'_{[x_1,\cdots,x_n]}(B)\rangle\omega'_{[x_1,\cdots,x_n]}(A)$
  $\omega'_{[x_1,\cdots,x_n]}([x]A) = []\omega'_{[x,x_1,\cdots,x_n]}(A)$
- We define $\omega' : \mathcal{M}' \mapsto \Lambda'$ by: $\omega'(A) = \omega'_{[v_1,\cdots,v_n]}(A)$ where $FV(A) \subseteq \{v_1, \cdots, v_n\}$.
  So for example, if our variables are ordered as $x, y, z, x', y', z', \cdots$ then
  $\omega'([x][y][x']\langle x'\rangle\langle z\rangle x) = \omega'_{[x,y,z]}([x][y][x']\langle x'\rangle\langle z\rangle x) =$
  $[]\omega_{[x,x,y,z]}([y][x']\langle x'\rangle\langle z\rangle x) = [][]\omega_{[y,x,x,y,z]}([x']\langle x'\rangle\langle z\rangle x) =$
  $[][][]\omega_{[x',y,x,x,y,z]}(\langle x'\rangle\langle z\rangle x) = [][][]\langle 1\rangle\langle 6\rangle 3.$

How do we do β-reduction?

- Note that $(\lambda x.\lambda y.zxy)(\lambda x.yx)$ translates to $(\lambda\lambda 521)(\lambda 31)$
- Note that $\lambda y'.z(\lambda x.yx)y'$ translates to $\lambda 4(\lambda 41)1$.
- Since $\underline{(\lambda x\lambda y.zxy)(\lambda x.yx)} \to_\beta \lambda y'.z(\lambda x.yx)y'$, we want that $\underline{(\lambda\lambda 521)(\lambda 31)} \to_\beta \lambda 4(\lambda 41)1$.
- The body of $\lambda\lambda 521$ is $\lambda 521$ and the variable bound by the first λ of $\lambda\lambda 521$ is the 2.
- But $(\lambda 521)[2 := \lambda 31]$ does not give $\lambda 4(\lambda 41)1$.
- What is $(\lambda 521)[2 := \lambda 31]$? Is it $\lambda 5(\lambda 31)1$?

Foundations 1: F29FA1
Lecture 14

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

Recall lambda calculus

- $\mathcal{V} = \{x, y, z, \dots\}$ and $V, V', V_1, V_2, \dots$ range over V.
- $\mathcal{M}$ is the set of terms of the λ-calculus and let $A, B, C, \dots$ range over $\mathcal{M}$. We can also index metavariables: i.e., also, $A_1, A_2, B_1, ...$ are metavariables that range over $\mathcal{M}$.
- If $A \in \mathcal{M}$ then $A ::= V|(A_1A_2)|(\lambda V.A_1)$.
- As parentheses are cumbersome, we will use the following notational convention *(syntactic sugaring)*:
  1. Functional application associates to the left. So *ABC denotes $((AB)C)$.*
  2. The body of a λ is anything that comes after it. So, *instead of $(\lambda v.(A_1A_2\dots A_n))$, we write $\lambda v.A_1A_2\dots A_n$.*
  3. A sequence of λ's is compressed to one. So *$\lambda xyz.t$ denotes $\lambda x.(\lambda y.(\lambda z.t))$.*

- We define the notion of *subterms*

$$
\begin{array}{lcl}
\text{Subterms}(v) & = & \{v\} \\
\text{Subterms}(\lambda v.A) & = & \text{Subterms}(A) \cup \{\lambda v.A\} \\
\text{Subterms}(AB) & = & \text{Subterms}(A) \cup \text{Subterms}(B) \cup \{AB\}
\end{array}
$$

- Recall free and bound occurrences of variables.
- Recall that a variable can occur both free and bound in a term.
- Recall that a term is closed when all occurrences of variables in it are bound.
- Recall grafting and that it is dangerous because some free occurrences may become bound after grafting.
- Recall *$A \equiv B$ iff A and B are exactly the same up to the name of their bound variables (i.e. A and B only differ in the name of their bound variables)*.

Substitution

- For any A, B, v, we define $A[v := B]$ to be the result of substituting B for every free occurrence of v in A, as follows:

$$
\begin{array}{lcll}
1.\ v[v := B] & \equiv & B & \\
2.\ v'[v := B] & \equiv & v' & \text{if } v \not\equiv v' \\
3.\ (AC)[v := B] & \equiv & A[v := B]C[v := B] & \\
4.\ (\lambda v.A)[v := B] & \equiv & \lambda v.A & \\
5.\ (\lambda v'.A)[v := B] & \equiv & \lambda v'.A[v := B] & \text{if } v \not\equiv v' \\
 & & \text{and } (v' \notin FV(B) \text{ or } v \notin FV(A)) & \\
6.\ (\lambda v'.A)[v := B] & \equiv & \lambda v''.A[v' := v''][v := B] & \text{if } v \not\equiv v' \\
 & & \text{and } (v' \in FV(B) \text{ and } v \in FV(A)) & \\
 & & \text{and } v'' \notin FV(AB) &
\end{array}
$$

Our reduction relations. Let $r \in \{\beta, \alpha, \eta\}$.

- Recall the three reduction axioms we have so far:

(α)	$\lambda v.A \to_\alpha \lambda v'.A[v := v']$	where $v' \notin FV(A)$
(β)	$(\lambda v.A)B \to_\beta A[v := B]$	
(η)	$\lambda v.Av \to_\eta A$	for $v \notin FV(A)$

- $\to_r$ is the least compatible relation closed under axiom (r).
- I.e., $A \to_r B$ if and only if one of the following holds:
  - A is the lefthand side of axiom (r) and B is its righthand side.
  - $\dfrac{A_1 \to_r A_2}{A \equiv A_1 C \to_r A_2 C \equiv B}$
  - $\dfrac{A_1 \to_r A_2}{A \equiv CA_1 \to_r CA_2 \equiv B}$
  - $\dfrac{A_1 \to_r A_2}{A \equiv \lambda v.A_1 \to_r \lambda v.A_2 \equiv B}$

- You can think of $\to_r$ as computation rules. When A computes to B, it is not necessarily the case that B computes to A.
- E.g., $(\lambda xyz.xyz)(\lambda x.xx) \to_\beta (\lambda yz.(\lambda x.xx)yz)$.
  But, $(\lambda yz.(\lambda x.xx)yz) \not\to_\beta (\lambda xyz.xyz)(\lambda x.xx)$.
- We introduce symmetry. We define $=_r$ to be the smallest relation which satsifies:
  - reflexive: $A =_r A$
  - transitive: $\dfrac{A =_r B \quad\quad B =_r C}{A =_r C}$
  - symmetric: $\dfrac{A =_r B}{B =_r A}$
  - contains $\to_r$: $\dfrac{A \to_r B}{A =_r B}$
- If $A =_r B$, we say that A and B are r-convertible.

- Here is a lemma about the interaction of β-reduction and substitution:
  Lemma: Let $A, B, C \in \mathcal{M}$.
  1. If $A \to_\beta B$ then $C[x := A] \twoheadrightarrow_\beta C[x := B]$.
  2. If $A \to_\beta B$ then $A[x := C] \to_\beta B[x := C]$.
- Proof: 1. By induction on the structure of C.
  2. By induction on the derivation $A \to_\beta B$ using the substitution lemma.
- For example: If $A \equiv (\lambda x.x)y$, $B \equiv y$ and $C \equiv xx$, then $A \to_\beta B$ and $C[x := A] \equiv A\underline{A} \to_\beta \underline{A}B \to_\beta BB \equiv C[x := B]$ and $A[x := C] \equiv A \to_\beta B \equiv B[x := C]$.

- If $A \to_\beta B$ or $A \to_\eta B$, we write $A \to_{\beta\eta} B$.
- We define $\twoheadrightarrow_{\beta\eta}$ to be the reflexive transitive closure of $\to_{\beta\eta}$.
- We define $=_{\beta\eta}$ to be the reflexive, symmetric and transitive closure of $\to_{\beta\eta}$.
- Again, $\twoheadrightarrow_{\beta\eta}$ and $=_{\beta\eta}$ are compatible.
- η-conversion equates two terms that have the same behaviour as functions and implies extensionality. I.e., for any $v \notin FV(A)$ and $v \notin FV(B)$, if $Av =_{\beta\eta} Bv$ then $A =_{\beta\eta} B$.
  Proof:
  - $Av =_{\beta\eta} Bv \Rightarrow^{compatibility} \lambda v.Av =_{\beta\eta} \lambda v.Bv$
  - But, by η, since $v \notin FV(A)$ and $v \notin FV(B)$, we have $\lambda v.Av =_{\beta\eta} A$ and $\lambda v.Bv =_{\beta\eta} B$
  - Hence, $A =_{\beta\eta} \lambda v.Av =_{\beta\eta} \lambda v.Bv =_{\beta\eta} B$ Hence, $A =_{\beta\eta} B$.

In Normal Form (Fully Evaluated)

- We say that A is in β-normal form, iff there are no β-redexes in A.
- $\lambda x.zx$ is in β-normal form but $(\lambda yx.yx)z$ is not.
- We say that A is in η-normal form, iff there are no η-redexes in A.
- $\lambda x.zx$ is not in η-normal form. But, $\lambda x.xx$ is in η-normal form.
- We say that A is in $\beta\eta$-normal form, iff there are no β-redexes and no η-redexes in A.
- $\lambda x.xx$ is in $\beta\eta$-normal form.
- Let $r \in \{\beta, \eta, \beta\eta\}$. Then, A is in r-normal form iff there are no r-redexes in A. I.e., there is no B such that $A \to_r B$.

Has Normal Form (Can be Fully Evaluated)

- Let $r \in \{\beta, \eta, \beta\eta\}$.
- We say that A has an r-normal form B iff $A =_r B$ and B is in r-normal form.
- For example, $(\lambda xyz.xyz)(\lambda x.xx)(\lambda x.x)x$ is not in β-normal form, but it has a β-normal form x.
- Not all terms have normal forms.
- $(\lambda x.xx)(\lambda x.xx)$ is not in β-normal form and there is no B such that $(\lambda x.xx)(\lambda x.xx) =_\beta B$ and B is in β-normal form.
- $(\lambda x.xx)(\lambda x.xx)$ does not have a β-normal form.
- We will see this later. For now, note that:
  $(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx)$.......

Let $r \in \{\beta, \eta, \beta\eta\}$.

- A term A is *strongly r-normalising* (*always r-terminates*) iff there are no infinite r-reduction sequences starting at A.
- A term A is *weakly r-normalising* (*weakly r-terminates*) iff there is a B in r-normal form such that $A \twoheadrightarrow_r B$.
- *Lemma:* If A is strongly r-normalising then A is weakly r-normalising and A has an r-normal form.
- *Theorem: Church-Rosser* $\forall A, B, C \in \mathcal{M}, \ \exists D \in \mathcal{M}$, such: $(A \twoheadrightarrow_r B \wedge A \twoheadrightarrow_r C) \Rightarrow (B \twoheadrightarrow_r D \wedge C \twoheadrightarrow_r D)$.

$$\begin{tikzcd}
 & A \arrow[dl, "r"'] \arrow[dr, "r"] & \\
B \arrow[dr, "r"'] & & C \arrow[dl, "r"] \\
 & D &
\end{tikzcd}$$

- *Equal programs have the same value*
- *A program reduces to its β-normal form (value)*
- *Normal forms (values) are unique*
- If A is in β-normal form, and if $A =_\beta B$, then $B \twoheadrightarrow_\beta A$.
- If $A =_\beta B$ then either both A and B have the same β-normal form, or neither one has a β-normal form.
- *λ-calculus is consistent*

- *Standardisation theorem:* If $A \twoheadrightarrow_\beta B$ then there is a standard reduction path from A to B.
- *Normalisation Theorem:* The leftmost β-reduction strategy is β-normalising.

Implementing lambda calculus in SML

- datatype LEXP =
  APP of LEXP * LEXP | LAM of string * LEXP | ID of string;
- fun printLEXP (ID v) = print v
  | printLEXP (LAM (v,e)) =
  (print "(\\ "; print v; print "."; printLEXP e; print ")")
  | printLEXP (APP(e1,e2)) =
  (print "("; printLEXP e1; print " "; printLEXP e2; print ")");
- val vx = (ID "x");
  val vx = ID "x" : LEXP
- printLEXP vx;
  xval it = () : unit
- val vy = (ID "y");
  val vz = (ID "z");
  val t1 = (LAM("x",vx));
- printLEXP t1;
  (\x.x)val it = () : unit

- val t2 = (LAM("y",vx));
  val t3 = (APP(APP(t1,t2),vz));
  val t4 = (APP(t1,vz));
  val t5 = (APP(t3,t3));
  val t6 =
  (LAM("x",(LAM("y",(LAM("z",(APP(APP(vx,vz),(APP(vy,vz))))))))
- printLEXP t2;
  ((\y.x)val it = () : unit
- printLEXP t3;
  ((((\x.x) ((\y.x)) z)val it = () : unit
- printLEXP t4;
  (((\x.x) z)val it = () : unit
- printLEXP t5;
  (((((\x.x) ((\y.x)) z) ((((\x.x) ((\y.x)) z))val it = () : unit
- printLEXP t6;
  ((\x.((\y.((\z.((x z) (y z)))))))val it = () : unit

- val t7 = (APP(APP(t6,t1),t1));
  val t8 = (LAM("z",(APP(vz,(APP(LAM("x",vx),vz))))));
- printLEXP t7;
  (((\x.(\y.(\z.((x z) (y z))))) (\x.x)) (\x.x))val it = () : unit
- printLEXP t8;
  (\z.(z ((\x.x) z)))val it = () : unit
- Write an SML function is_var which checks whether an expression is a variable.
  Solution:
  ```
  fun is_var (ID id) = true
    | is_var _ = false;
  ```
- is_var vx;
  val it = true : bool
- is_var t6;
  val it = false : bool

- Write a function addlam which takes a variable v and a list of expressions and returns the same list, but where every expression starts with $\lambda v.$.
  Solution: fun addlam id [] = []
  | addlam id (e::l) = (LAM(id,e))::(addlam id l);
- \- addlam "x" [vx, t1];
  val it = [LAM ("x",ID "x"),LAM ("x",LAM (#,#))] : LEXP list
- \- printLEXP (List.hd (addlam "x" [vx, t1]));
  (\x.x)val it = () : unit
- \- printLEXP (List.last (addlam "x" [vx, t1]));
  (\x.(\x.x))val it = () : unit

- Recall that we wrote a function freeVars which gives the free variables of expressions.
- \- printLEXP t1;
  (\x.x)val it = () : unit
  \- freeVars t1;
  val it = [] : string list
- \- printLEXP t2;
  (\y.x)val it = () : unit
  \- freeVars t2;
  val it = ["x"] : string list
- Recall also free.
- \- free "x" t1;
  val it = false : bool
- \- free "x" t2;
  val it = true : bool

- Remember also subs
- \- subs vy "x" t2;
  val it = LAM ("x1",ID "y") : LEXP
- \- printLEXP (subs vy "x" t2);
  (\x1.y)val it = () : unit
- \- subs vy "y" t2; val it = LAM ("y",ID "x") : LEXP
- \- printLEXP (subs vy "y" t2);
  (\y.x)val it = () : unit

- Write an SML function addbackapp which takes a list *l* and an expression *e*2 and returns *l* where every terms is applied to *e*2.
  Solution:
  fun addbackapp [] e2 = []
  | addbackapp (e1::l) e2 = (APP(e1,e2)):: (addbackapp l e2);
- addbackapp [vx, t1] t2;
  val it = [APP (ID "x",LAM (#,#)),APP (LAM (#,#),LAM (#,#))] : LEXP list
- - printLEXP (List.hd (addbackapp [vx, t1] t2));
  (x (\y.x))val it = () : unit
- - printLEXP (List.last(addbackapp [vx, t1] t2));
  ((\x.x) (\y.x))val it = () : unit

- Write an SML function addfrontapp which takes an expression $e1$ and a list l and returns l where $e1$ is applied to every terms.
  Solution:
  fun addfrontapp e1 [] = []
  | addfrontapp e1 (e2::l) = (APP(e1,e2)):: (addfrontapp e1 l);
- \- addfrontapp t2 [vx, t1];
  val it = [APP (LAM (#,#),ID "x"),APP (LAM (#,#),LAM (#,#))] : LEXP list
- \- printLEXP (List.hd (addfrontapp t2 [vx, t1]));
  ((\y.x) x)val it = () : unit
- \- printLEXP (List.last(addfrontapp t2 [vx, t1]));
  ((\y.x) (\x.x))val it = () : unit

- Write an SML function printlistreduce which takes a list of elements and prints them one after another with an arrow in between.
  Solution: fun printlistreduce [] = ()
  | printlistreduce (e::[]) = ((printLEXP e) ; print "\n")
  | printlistreduce (e::l) =
  (printLEXP e; print "→" ; print "\n"; (printlistreduce l));
- \- printlistreduce [vx, t1];
  x→
  (\x.x)
  val it = () : unit
- \- printlistreduce [vx, t1, t2];
  x→
  (\x.x)→
  (\y.x)
  val it = () : unit

- Write an SML function which takes a β-redex and reduces it.
  Solution: fun red (APP(LAM(id,e1),e2)) = subs e2 id e1;
- ```
  - printLEXP t4;
  ((\x.x) z)val it = () : unit
  - red t4;
  val it = ID "z" : LEXP
  - printLEXP (red t4);
  zval it = () : unit
  ```
- ```
  - printLEXP t3;
  (((\x.x) (\y.x)) z)val it = () : unit
  - printLEXP t8;
  (\z.(z ((\x.x) z)))val it = () : unit
  - printLEXP (APP(t8, t3));
  ((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))val it = () : unit
  - printLEXP (red (APP(t8, t3)));
  ((((\x.x) (\y.x)) z) ((\x.x) (((\x.x) (\y.x)) z)))val it = () :
  unit
  ```

- Write an SML function which reduces a term to normal form (giving all the reduction steps) using the a-strategy in which the contracted redex is:
  a(AB) = a(A) if A has a β-redex
  a(AB) = a(B) if A does not have a β-redex and B has a β-redex
  a(AB) = AB if AB is a β-redex and neither A nor B has a β-redex.
  a(AB) = undefined if AB does not have a β-redex
  a(λ v.A) = a(A)
  a(v) = undefined

- Solution (note that unlike mreduce, this avoids redundancies):

```
fun areduce (ID id) = [(ID id)]
| areduce (LAM(id,e)) = (addlam id (areduce e))
| areduce (APP(e1,e2)) =
(let val l1 = if (has_redex e1) then (areduce e1) else [e1]
      val e3 = (List.last l1)
      val l2 = if (has_redex e2) then (areduce e2) else [e2]
      val e4 = (List.last l2)
      val l3 = (addfrontapp e3 l2)
      val l4 = (addbackapp l1 e2)
      val l5 = l4 @ (List.tl l3)
      in if (is_redex (APP(e3,e4))) then
      l5 @ (areduce (red (APP(e3,e4)))) else l5
end);
```

- Write an SML function printareduce which first a-reduces the term giving the list of all intermediary terms and then prints this list separating intermediary terms with $\rightarrow$.
  Solution: fun printareduce e = printlistreduce (areduce e);
- \- areduce t3;
  val it = [APP (APP (#,#),ID "z"),APP (LAM (#,#),ID "z"),ID "x"] : LEXP list
  \- printareduce t3;
  (((\x.x) (\y.x)) z)→
  ((\y.x) z)→
  x val it = () : unit
- printmreduce t3;
  (((\x.x) (\y.x)) z)→
  (((\x.x) (\y.x)) z)→
  ((\y.x) z)→
  ((\y.x) z)→
  x val it = () : unit

- \- printareduce t5;
  ((((\x.x) (\y.x)) z) (((\x.x) (\y.x)) z))→
  (((\y.x) z) (((\x.x) (\y.x)) z))→
  (x (((\x.x) (\y.x)) z))→
  (x ((\y.x) z))→
  (x x) val it = () : unit
- printmreduce t5;
  ((((\x.x) (\y.x)) z) (((\x.x) (\y.x)) z))→
  ((((\x.x) (\y.x)) z) (((\x.x) (\y.x)) z))→
  (((\y.x) z) (((\x.x) (\y.x)) z))→
  (((\y.x) z) (((\x.x) (\y.x)) z))→
  (x (((\x.x) (\y.x)) z))→
  (x (((\x.x) (\y.x)) z))→
  (x (((\x.x) (\y.x)) z))→
  (x ((\y.x) z))→
  (x ((\y.x) z))→
  (x x)→
  (x x)val it = () : unit

- \- areduce t8;
  val it = [LAM ("z",APP (#,#)),LAM ("z",APP (#,#))] :
  LEXP list
  \- printareduce t8;
  (\z.(z ((\x.x) z)))→
  (\z.(z z))
  val it = () : unit
- printmreduce t8;
  (\z.(z ((\x.x) z)))→
  (\z.(z ((\x.x) z)))→
  (\z.(z ((\x.x) z)))→
  (\z.(z z))
  (\z.(z z))
  val it = () : unit

- \- val t9 = APP(t8,t3);
  val t9 = APP (LAM ("z",APP #),APP (APP #,ID #)) : LEXP
- \- printLEXP t9;
  ((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))val it = () : unit
- \- areduce t9;
  val it = [APP (LAM (#,#),APP (#,#)),APP (LAM (#,#),APP (#,#)), APP (LAM (#,#),APP (#,#)),APP (LAM (#,#),ID "x"),APP (ID "x",ID "x")] : LEXP list
- \- printareduce t9;
  ((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))→
  ((\z.(z z)) (((\x.x) (\y.x)) z))→
  ((\z.(z z)) ((\y.x) z))→
  ((\z.(z z)) x)→
  (x x)
  val it = () : unit

- Similarly, printmreduce t9; gives: ((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))→
  ((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))→
  ((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))→
  ((\z.(z z)) (((\x.x) (\y.x)) z))→
  ((\z.(z z)) (((\x.x) (\y.x)) z))→
  ((\z.(z z)) (((\x.x) (\y.x)) z))→
  ((\z.(z z)) (((\x.x) (\y.x)) z))→
  ((\z.(z z)) ((\y.x) z))→
  ((\z.(z z)) ((\y.x) z))→
  ((\z.(z z)) x)→
  (x x)→
  (x x)→
  (x x)
  val it = () : unit

- fun subterms (ID id) = [ID id]
  | subterms (APP(e1,e2)) =
  [APP(e1,e2)] @ ((subterms e1) @ (subterms e2))
  | subterms (LAM(id2, e1)) = [LAM(id2, e1)] @ (subterms e1);
- - subterms t1;
  val it = [LAM ("x",ID "x"),ID "x"] : LEXP list
- - subterms t2;
  val it = [LAM ("y",ID "x"),ID "x"] : LEXP list
- - subterms t3;
  val it =
  [APP (APP (#,#),ID "z"),APP (LAM (#,#),LAM
  (#,#)),LAM ("x",ID "x"),ID "x",
  LAM ("y",ID "x"),ID "x",ID "z"] : LEXP list

Item Notation

- $\mathcal{V} = \{x, y, z, \dots\}$ and $V, V', V_1, V_2, \dots$ range over $\mathcal{V}$.
- $\mathcal{M}'$ is a set of terms over which $A', B', C', \dots$ range, where $A' ::= V| \; [V]A'_1| < A'_2 > A'_1$.
- Also here we can index metavariables: i.e., also, $A'_1, A'_2, B'_1, ...$ are metavariables that range over $\mathcal{M}'$.
- In $\mathcal{M}'$, you must understand $< A'_2 > A'_1$ to mean that A'_2 is input to A'_1 (i.e., that A'_1 is applied to A'_2).
- Call forms like $< A' >$ and $[v]$, *wagons*. Here, $< A' >$ is an applicator-wagon and $[v]$ is an abstractor-wagon. You can see that any term of $\mathcal{M}'$ is a number of wagons followed by a variable (which we call the heart of the term).

- E.g., the term $<z><[y]y>[x]<y>x$ consists of the four wagons $<z>$ and $<[y]y>$ and $[x]$ and $<y>$ and the heart x.
- Call any two wagons next to each other of the form $<A'>[v]$ lovers and call any wagon $<A'>$ which is not immediately left of $[v]$, a bachelor (also $[v]$ is called bachelor when it does not have a lover to its left).

Item Notation in SML: Way 1

- datatype IEXP = APPon of IEXP * IEXP | ABS of string * IEXP | IID of string;
- (*Prints a term*)
  fun printIEXP (IID v) = print v
  | printIEXP (ABS (v,e)) =
  (print "["; print v; print "]"; printIEXP e)
  | printIEXP (APPon(e1,e2)) =
  (print "<"; printIEXP e1; print ">"; printIEXP e2);
- val Ivx = (IID "x");
  val Ivy = (IID "y");
  val Ivz = (IID "z");
  val It1 = (ABS("x",Ivx));
  val It2 = (ABS("y",Ivx));

Item Notation in SML: Way 2

- datatype IEXP = IAPP of IEXP * IEXP | ILAM of string * IEXP | IID of string;
- (*Prints a term*)
  fun printIEXP (IID v) = print v
  | printIEXP (ILAM (v,e)) =
  (print "["; print v; print "]"; printIEXP e)
  | printIEXP (IAPP(e1,e2)) =
  (print "<"; printIEXP e1; print ">"; printIEXP e2);
- val Ivx = (IID "x");
  val Ivy = (IID "y");
  val Ivz = (IID "z");
  val It1 = (ILAM("x",Ivx));
  val It2 = (ILAM("y",Ivx));

More lambda terms and SML

- You learned $\mathcal{M}$ and its implementation in SML very well (our lectures and our lab, and all the SML functions I have given you).
- You can print terms of $\mathcal{M}$ in SML, you can reduce in SML, you can check subterms, free variables, substitutions, etc in SML. You are expert.
- You learned $\mathcal{M}'$ in SML and how to print it.

More lambda terms and SML

```
- lt1;
val it = lLAM ("x",lID "x") : lEXP
- printlEXP lt1;
[x]x val it = () : unit
- lt2;
val it = lLAM ("y",lID "x") : lEXP
- printlEXP lt2;
[y]x val it = () : unit
- lAPP(lt1,lt2);
val it = lAPP (lLAM ("x",lID #),lLAM ("y",lID #)) : lEXP
- printlEXP (lAPP(lt1,lt2));
⟨[x ] x ⟩[ y]x val it = () : unit
```

Translating from Classical to item notation $V : \mathcal{M} \mapsto \mathcal{M}'$

- Recall that for any function, we have to say what its domain is and what its range is.
- So, $V : \mathcal{M} \mapsto \mathcal{M}'$ has $\mathcal{M}$ for domain and $\mathcal{M}'$ for range. It takes elements from $\mathcal{M}$ and returns elements in $\mathcal{M}'$.
- Since $\mathcal{M} ::= \mathcal{V} \mid (\mathcal{M}\mathcal{M}) \mid (\lambda\mathcal{V}.\mathcal{M})$ then V needs to be defined for each case of $\mathcal{M}$. This is done as follows:
  - $V(v) = v$.
  - $V(AB) = \langle V(B)\rangle V(A)$.
  - $V(\lambda v.A) = [v]V(A)$.
- Note that for any $A \in \mathcal{M}$, $V(A) \in \mathcal{M}'$.
  $V(v) = v \in \mathcal{M}'$.
  $V(AB) = \langle V(B)\rangle V(A)$ and since $V(A) \in \mathcal{M}'$ and $V(B) \in \mathcal{M}'$ then $V(AB) = \langle V(B)\rangle V(A) \in \mathcal{M}'$.
  $V(\lambda v.A) = [v]V(A)$ and since $V(A) \in \mathcal{M}'$ then $V(\lambda v.A) = [v]V(A) \in \mathcal{M}'$.

- For example, $V((\lambda x.(\lambda y.xy))z) \equiv \langle z\rangle[x][y]\langle y\rangle x$. The wagons (or items) are $\langle z\rangle$, $[x]$, $[y]$ and $\langle y\rangle$.
- *applicator wagon* $\langle z\rangle$ and *abstractor wagon* $[x]$ occur NEXT to each other.
- Note that any term is a wagon followed by a term. So, $\langle z\rangle[x][y]\langle y\rangle x$ is the wagon $\langle z\rangle$ followed by the term $[x][y]\langle y\rangle x$.
- This continues so in the end, a term is a sequence of wagons followed by a variable which we call the *heart* of the term.
- So, $\langle z\rangle[x][y]\langle y\rangle x$ is the sequence of wagons $\langle z\rangle$, $[x]$, $[y]$, $\langle y\rangle$ followed by the heart x.

How to implement $V : \mathcal{M} \mapsto \mathcal{M}'$?

- We implemented $\mathcal{M}$ by datatype LEXP = APP of LEXP * LEXP | LAM of string * LEXP | ID of string;
- We implemented $\mathcal{M}'$ by datatype IEXP = IAPP of IEXP * IEXP | ILAM of string * IEXP | IID of string;
- So, the implementation of V (say Itran) should take an LEXP and return IEXP. That is:
  Itran : LEXP $\mapsto$ IEXP.
- So, you need to write an SML function (say Itran) which takes every case of LEXP and returns an IEXP. The skeleton is as follows:
- fun Itran (ID id) = ?fill corresponding IEXP term
  | Itran (APP(e1,e2)) = ??fill corresponding IEXP term
  | Itran (LAM(id, e)) = ???fill corresponding IEXP term;

Some Examples

Assume you implement V and its inverse as *ltran* and *tran*

- printLEXP t6;

(\x.(\y.(\z.((x z) (y z)))))val it = () : unit

- ltran t6;

val it = ILAM ("x",ILAM ("y",ILAM #)) : IEXP

- printIEXP (ltran t6);

[x][y][z]⟨⟨z⟩ y⟩⟨z⟩x val it = () : unit

- printLEXP (ltran(tran t6));

(\x.(\y.(\z.((x z) (y z)))))val it = () : unit

- ltran t1;

val it = ILAM ("x",IID "x") : IEXP

- printIEXP (ltran t1);

[x]xval it = () : unit

- printLEXP t9;

((\z.(z ((\ x.x) z))) (((\x.x) (\y.x)) z))val it = () : unit

- printIEXP (ltran t9);

⟨⟨z⟩⟨[y]x⟩[x]x⟩[z]⟨⟨z⟩[x]x⟩zval it = () : unit

You can also implement in SML leftmost reduction in item notation in a similar way to how I implemented it for classical notation. If you do so, then:

```
- printlreduce t6;
(\x.(\y.(\z.((x z) (y z)))))val it = () : unit
- printIloreduce (Itran t6);
[x][y][z]⟨⟨z⟩y⟩⟨z⟩x val it = () : unit
-printLEXP (APP(t6,t6));
((\x.(\y.(\z.((x z) (y z))))) (\x.(\y.(\z.((x z) (y z))))))val it = () :
unit
-printloreduce (APP(t6,t6));
((\x.(\y.(\z.((x z) (y z))))) (\x.(\y.(\z.((x z) (y z))))))→
(\y.(\z.(((\x.(\ y.(\ z.((x z) (y z))))) z) (y z))))→
(\y.(\z.((\y.(\x1.((z x1) (y x1)))) (y z))))→
(\y.(\z.(\x1.((z x1) ((y z) x1)))))val it = () : unit
```

-printIEXP (Itran(APP(t6,t6)));
$\langle[x][y][z]\langle\langle z\rangle y\rangle\langle z\rangle x\rangle[x][y][z]\langle\langle z\rangle y\rangle\langle z\rangle x$ val it = () : unit
-printIloreduce(Itran(APP(t6,t6)));
$\langle[x][y][z]\langle\langle z\rangle y\rangle\langle z\rangle x\rangle[x][y][z]\langle\langle z\rangle y\rangle\langle z\rangle x\longrightarrow$
$[y][z]\langle\langle z\rangle y\rangle\langle z\rangle[x][y][z]\langle\langle z\rangle y\rangle\langle z\rangle x\longrightarrow$
$[y][z]\langle\langle z\rangle y\rangle[y][x1]\langle\langle x1\rangle y\rangle\langle x1\rangle z\longrightarrow$
$[y][z][x1]\langle\langle x1\rangle\langle z\rangle y\rangle\langle x1\rangle z$val it = () : unit

```
- printLEXP t9;
((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))val it = () : unit
- printloreduce t9;
((\z.(z ((\x.x) z))) (((\x.x) (\y.x)) z))-->
((((\x.x) (\y.x)) z) ((\x.x) (((\x.x) (\y.x)) z)))-->
(((\y.x) z) ((\x.x) (((\x.x) (\y.x)) z)))-->
(x ((\x.x) (((\x.x) (\y.x)) z)))-->
(x (((\x.x) (\y.x)) z))-->
(x ((\y.x) z))-->
(x x)val it = () : unit
```

- printIEXP (Itran t9);

$<< z >< [y]x > [x]x > [z] << z > [x]x > z$val it = () : unit

- printIloreduce (Itran t9);

$<< z >< [y]x > [x]x > [z] << z > [x]x > z \rightarrow$

$<<< z >< [y]x > [x]x > [x]x >< z >< [y]x > [x]x \rightarrow$

$<<< z >< [y]x > [x]x > [x]x >< z > [y]x \rightarrow$

$<<< z >< [y]x > [x]x > [x]x > x \rightarrow$

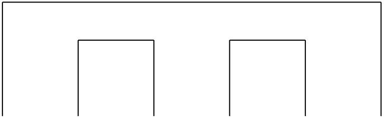
$<< z >< [y]x > [x]x > x \rightarrow$

$<< z > [y]x > x \rightarrow$

$< x > x$val it = () : unit

Generalising Redexes

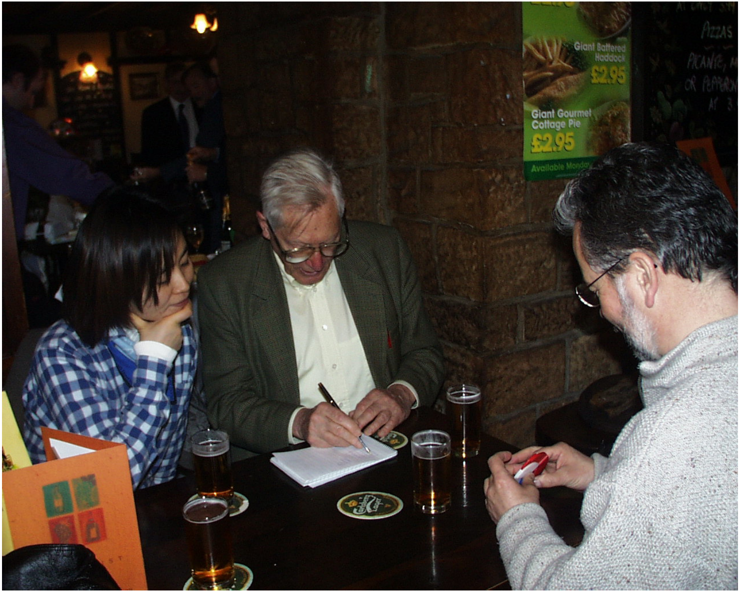
Classical Notation	Item Notation
$((\lambda_x.(\lambda_y.\lambda_z.zd)c)b)a$	$\langle a\rangle\langle b\rangle[x]\langle c\rangle[y][z]\langle d\rangle z$
$\downarrow_\beta$	$\downarrow_\beta$
$((\lambda_y.\lambda_z.zd)c)a$	$\langle a\rangle\langle c\rangle[y][z]\langle d\rangle z$
$\downarrow_\beta$	$\downarrow_\beta$
$(\lambda_z.zd)a$	$\langle a\rangle[z]\langle d\rangle z$
$\downarrow_\beta$	$\downarrow_\beta$
ad	$\langle d\rangle a$



$\langle a\rangle \quad \langle b\rangle \quad [x] \quad \langle c\rangle \quad [y] \quad [z] \quad \langle d\rangle \quad z$

- This maks it easy to introduce local/global/mini/lazy reductions into the λ-calculus.

De Bruijn: An impressive thinker and entertainer



- Insert the full amount of parenthesis in every lambda expression you have seen.
- Remove as many parenthesis as possible from any lambda expression you have seen.
- Give the subterms of every lambda expression you have seen.
- Consider any two lambda expressions you have seen and state whether one is a subterm of the other explaining your reasons.
- For any lambda term, give the free and bound occurrences of variables in it.
- Consider any two lambda expressions you have seen and state whether they are syntactically equivalent or not.
- For any lambda term, practice r-reducing it where $r \in \{\beta, \eta, \alpha, \beta\eta\}$.

- For any lambda term you know, state whether it has an r-normal form for $r \in \{\beta, \eta, \beta\eta\}$. Justify your reasons.
- For any lambda term you know, state whether it is in r-normal form for $r \in \{\beta, \eta, \beta\eta\}$. Justify your reasons.
- For any lambda term you know, state whether it is weakly β-normalising. Justify your reasons.
- For any lambda term you know, state whether it is strongly β-normalising. Justify your reasons.
- For any reduction path you have seen, state whether it is normal or standard or neither. Justify your reasons.
- Study the implementation of the lambda calculus in SML, write lambda terms in SML, run SML functions to find free variables, substitutions, subterms, etc.

Foundations 1: F29FA1
Lecture 16

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

How do we do β-reduction?

- Note that $(\lambda x.\lambda y.zxy)(\lambda x.yx)$ translates to $(\lambda\lambda 521)(\lambda 31)$
- Note that $\lambda y'.z(\lambda x.yx)y'$ translates to $\lambda 4(\lambda 41)1$.
- Since $\underline{(\lambda x\lambda y.zxy)(\lambda x.yx)} \to_\beta \lambda y'.z(\lambda x.yx)y'$, we want that $\underline{(\lambda\lambda 521)(\lambda 31)} \to_\beta \lambda 4(\lambda 41)1$.
- The body of $\lambda\lambda 521$ is $\lambda 521$ and the variable bound by the first λ of $\lambda\lambda 521$ is the 2.
- But $(\lambda 521)[2 := \lambda 31]$ does not give $\lambda 4(\lambda 41)1$.
- What is $(\lambda 521)[2 := \lambda 31]$? Is it $\lambda 5(\lambda 31)1$?

In order to define β-reduction $(\lambda A)B \to_\beta ?$ using de Bruijn indices. We must:

- find in A the occurrences $n_1, \ldots n_k$ of the variable bound by the λ of λA.
  For example, in $\lambda 1(\lambda 2(\lambda 3))$, all of 1, 2 and 3 are bound by the first λ. In normal notation this is: $\lambda x.x(\lambda y.x(\lambda z.x))$.
- decrease the variables of A to reflect the disappearance of the λ from λA.
  For example, $(\lambda 12)3$ must return $3\ 1$.
  I.e., $(\lambda y.yx)z$ must return zx.
- replace the occurrences $n_1, \ldots n_k$ in A by updated versions of B which take into account that variables in B may appear within the scope of extra λs in A.
  For example, $(\lambda\lambda 2)3$ must return $\lambda 4$.
  I.e., $(\lambda x.\lambda y.x)z$ must return $\lambda y.z$.

- Let us, in order to simplify things say that the β-rule is $(\lambda A)B \to_\beta A\{\{1 \leftarrow B\}\}$ and let us define $A\{\{1 \leftarrow B\}\}$ in a way that all the work is carried out.
- The *meta-updating functions* $U_k^i : \Lambda \to \Lambda$ for $k \geq 0$ and $i \geq 1$ are defined inductively as follows:

$$U_k^i(AB) \equiv U_k^i(A)\, U_k^i(B)$$

$$U_k^i(\lambda A) \equiv \lambda(U_{k+1}^i(A))$$

$$U_k^i(\mathtt{n}) \equiv \begin{cases} \mathtt{n}+\mathtt{i}-1 & \text{if} \quad n > k \\ \mathtt{n} & \text{if} \quad n \leq k \,. \end{cases}$$

- The intuition behind U_k^i is the following: k tests for free variables and $i-1$ is the value by which a variable, if free, must be incremented.

The *meta-substitutions at level* i, for $i \geq 1$, of a term $B \in \Lambda$ in a term $A \in \Lambda$, denoted $A\{\!\{\mathtt{i} \leftarrow B\}\!\}$, is defined inductively on A as follows:

- $$\begin{array}{lcl} (A_1 A_2)\{\!\{\mathtt{i} \leftarrow B\}\!\} & \equiv & (A_1\{\!\{\mathtt{i} \leftarrow B\}\!\})\,(A_2\{\!\{\mathtt{i} \leftarrow B\}\!\}) \\ (\lambda A)\{\!\{\mathtt{i} \leftarrow B\}\!\} & \equiv & \lambda(A\{\!\{\mathtt{i}+\mathtt{1} \leftarrow B\}\!\}) \end{array}$$
  $$\mathtt{n}\{\!\{\mathtt{i} \leftarrow B\}\!\} \equiv \begin{cases} \mathtt{n}-\mathtt{1} & \text{if } n > i \\ U_0^i(B) & \text{if } n = i \\ \mathtt{n} & \text{if } n < i\,. \end{cases}$$
- For example $(\lambda 521)\{\!\{1 \leftarrow (\lambda 31)\}\!\} \equiv \lambda 4(\lambda 41)1$
- Hence $(\lambda\lambda 521)(\lambda 31) \rightarrow_\beta \lambda 4(\lambda 41)1$.

Representing propositional logic in the λ-calculus

- **true** $\equiv \lambda xy.x$
  false $\equiv \lambda xy.y$
  not $\equiv \lambda x.x$ **false true**
  cond $\equiv \lambda xyz.xyz$
  and $\equiv \lambda xy.$**cond** x y**false**
  or $\equiv \lambda xy.$**cond** x **true** y
- We show that **not true** $=_\beta$ **false**:
  not true $\equiv$ $(\lambda x.x$ **false true**$)$**true** $\to_\beta$ **true false true** $\equiv$ $(\lambda xy.x)$ **false true** $\to_\beta$ $(\lambda y.$**false**$)$**true** $\to_\beta$ **false**.
- As an exercise, show that: **not false** $=_\beta$ **true**

cond true $AB =_\beta A$	**cond false** $AB =_\beta B$
and true false $=_\beta$ **false**	**and true true** $=_\beta$ **true**
and false false $=_\beta$ **false**	**and false true** $=_\beta$ **false**
or true false $=_\beta$ **true**	**or true true** $=_\beta$ **true**
or false false $=_\beta$ **false**	**or false true** $=_\beta$ **true**

Representing pairing and projection in the λ-calculus

- **pair** $\equiv \lambda xyz.zxy$
  fst $\equiv \lambda x.x$ **true**
  snd $\equiv \lambda x.x$ **false**
  n-tuple $\equiv \lambda x_1, x_2 \ldots x_n.$**pair** x_1 (**pair** x_2 $\ldots$ (**pair** x_{n-1} $x_n) \ldots)$
  pos1n $\equiv \lambda x.$**fst** x
  pos2n $\equiv \lambda x.$**fst**(**snd** x)
  posin $\equiv \lambda x.\mathbf{fst}(\underbrace{\mathbf{snd}(\ldots(\mathbf{snd}}_{i-1 \text{ times}}\ x)\ldots))$ for $i < n$
  posnn $\equiv \lambda x.\underbrace{\mathbf{snd}(\mathbf{snd}(\ldots(\mathbf{snd}}_{n-1 \text{ times}}\ x)\ldots))$

- We show that $\mathbf{fst}(\mathbf{pair}\ A\ B) =_\beta A$:
$\mathbf{fst}(\mathbf{pair}\ A\ B) \equiv (\lambda x.x\ \mathbf{true})(\mathbf{pair}\ A\ B) \to_\beta (\mathbf{pair}\ A\ B)\mathbf{true} \equiv$
$((\lambda xyz.zxy)A\ B)\mathbf{true} \to_\beta ((\lambda yz.zAy)\ B)\mathbf{true} \to_\beta$
$(\lambda z.zAB)\mathbf{true} \to_\beta \mathbf{true}\,AB \equiv (\lambda xy.x)AB \to_\beta (\lambda y.A)B \to_\beta A$.
- We show that $\mathbf{snd}(\mathbf{pair}\ A\ B) =_\beta B$:
$\mathbf{snd}(\mathbf{pair}\ A\ B) \equiv (\lambda x.x\ \mathbf{false})(\mathbf{pair}\ A\ B) \to_\beta (\mathbf{pair}\ A\ B)\mathbf{false} \equiv$
$((\lambda xyz.zxy)A\ B)\mathbf{false} \to_\beta ((\lambda yz.zAy)\ B)\mathbf{false} \to_\beta$
$(\lambda z.zAB)\mathbf{false} \to_\beta \mathbf{false}\,AB \equiv (\lambda xy.y)AB \to_\beta (\lambda y.y)B \to_\beta B$
- Show that $\mathbf{posin}(\mathbf{pair}\ A_1 \ldots (\mathbf{pair}\ A_{n-1}\ A_n) \ldots)) =_\beta A_i$ for $1 \leq i \leq n$.

Exercise 1

- For each of the terms A below do the following:
  Translate A to a term A' using de Bruijn indices. β-reduce A to a β-normal form B. β-reduce A' to a β-normal form B'. Translate B to a term B'' using de Bruijn indices. Note that $B' \equiv B''$.
  1. $A \equiv (\lambda x.x)y$.
  2. $A \equiv (\lambda xy.xy)y$.
  3. $A \equiv (\lambda xy.xy)(\lambda z.zx)$.

Exercise 2

- For each of the terms A below do the following:
  Translate A to a term A' using de Bruijn indices. β-reduce A to a β-normal form B. β-reduce A' to a β-normal form B'. Translate B to a term B'' using de Bruijn indices. Note that $B' \equiv B''$.
  1. $A \equiv (\lambda x.y)x$.
  2. $A \equiv (\lambda xy.yx)(\lambda x.x)$.
  3. $A \equiv (\lambda xy.xy)(\lambda z.zx)$.

- *Exercise 3* Show that
  1. **not false** $=_\beta$ **true**
  2. **cond true** $AB =_\beta A$
  3. **cond false** $AB =_\beta B$
  4. **and true false** $=_\beta$ **false**
  5. **and true true** $=_\beta$ **true**
  6. **and false false** $=_\beta$ **false**
  7. **and false true** $=_\beta$ **false**
  8. **or true false** $=_\beta$ **true**
  9. **or true true** $=_\beta$ **true**
  10. **or false false** $=_\beta$ **false**
  11. **or false true** $=_\beta$ **true**
- *Exercise 4* Show that
  $\mathbf{posin}(\mathbf{pair}\ A_1 \ldots (\mathbf{pair}\ A_{n-1}\ A_n) \ldots)) =_\beta A_i$ for $1 \le i \le n$.

Foundations 1: F29FA1
Lecture 17

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

Representing Church's numerals and arithmetic in the λ-calculus

- $\mathbf{0} \equiv \lambda yx.x$
  $\mathbf{1} \equiv \lambda yx.yx$
  $\mathbf{2} \equiv \lambda yx.y(yx)$
  $\ldots$
  $\mathbf{n} \equiv \lambda yx.y^n x$ where $y^n x \equiv \underbrace{y(y(\ldots(y}_{n \text{ times}} x)))$

  $\mathbf{succ} \equiv \lambda zyx.zy(yx)$
  $\mathbf{add} \equiv \lambda zz'yx.zy(z'yx)$
  $\mathbf{iszero} \equiv \lambda z.z(\lambda x.\mathbf{false})\mathbf{true}$
  $\mathbf{times} \equiv \lambda zyx.z(yx)$
  $\mathbf{prefn} \equiv \lambda yz.\mathbf{pair\ false}(\mathbf{cond}(\mathbf{fst}\ z)(\mathbf{snd}\ z)(y(\mathbf{snd}\ z)))$
  $\mathbf{pre} \equiv \lambda zyx.\mathbf{snd}(z(\mathbf{prefn}\ y)(\mathbf{pair\ true}\ x))$

- We can prove that: $(x^m)^n =_\beta x^{nxm}$.
- Recall again that **times** $\equiv \lambda zyx.z(yx)$ and take the following proof that **times n m** $=_{\beta\eta}$ **nxm**:

$$\begin{aligned}
\textbf{times n m} &\equiv (\lambda zyx.z(yx))\textbf{n m}\\
&\twoheadrightarrow_\beta \lambda x.\textbf{n}(\textbf{m}\, x)\\
&\equiv \lambda x.\textbf{n}((\lambda zy.z^m y)x)\\
&\rightarrow_\beta \lambda x.\textbf{n}(\lambda y.x^m y)\\
&\rightarrow_\eta \lambda x.\textbf{n}(x^m)\\
&\equiv \lambda x.(\lambda zy.z^n y)(x^m)\\
&\rightarrow_\beta \lambda x.(\lambda y.(x^m)^n y)\\
&\rightarrow_\eta \lambda x.(x^m)^n\\
&=_\beta \lambda x.x^{nxm}\\
&=_\eta \lambda x.\lambda y.x^{nxm} y\\
&\equiv \textbf{nxm}
\end{aligned}$$

- But we should not depend on η.
- Can we define
  $\textbf{mult} \equiv \lambda xy.\textbf{cond}\ (\textbf{iszero}\ x)\ \textbf{0}\ (\textbf{add}\ y\ (\textbf{mult}\ (\textbf{pre}\ x)\ y))$
- But this means that mult is defined in terms of mult. How can this be done?
- The solution comes from the fixed-point theorem: In the lambda calculus, we have fixed point finders.
- These are λ-expressions (say Fix) such that for any expression A, we have:
  $\text{Fix}\, A =_\beta A(\text{Fix}\, A)$.
- That is: Fix A is a fixed point of A.

- Can we define
  $\textbf{mult} \equiv \lambda xy.\textbf{cond}\,(\textbf{iszero}\,x)\,\textbf{0}\,(\textbf{add}\,y\,(\textbf{mult}\,(\textbf{pre}\,x)\,y))$?
- The solution comes from the fixed-point theorem: In the lambda calculus, we have fixed point finders.
- These are λ-expressions (say Fix) such that for any expression A, we have: Fix $A =_\beta A(\text{Fix } A)$.
- That is: Fix A is a fixed point of A.
- So, how do we use Fix to find **mult**?
- First, we define
  $\textbf{multfn} \equiv \lambda zxy.\textbf{cond}\,(\textbf{iszero}\,x)\,\textbf{0}\,(\textbf{add}\,y\,(z\,(\textbf{pre}\,x)\,y))$.
- Then, we define $\textbf{mult} \equiv \text{Fix}\,\textbf{multfn}$.
- By Fixed point theorem, $\text{Fix}\,\textbf{multfn} =_\beta \textbf{multfn}(\text{Fix}\,\textbf{multfn})$.
- Hence, $\textbf{mult} \equiv \text{Fix}\,\textbf{multfn} =_\beta \textbf{multfn}(\text{Fix}\,\textbf{multfn}) =_\beta \textbf{multfn}(\textbf{mult}) =_\beta \lambda xy.\textbf{cond}\,(\textbf{iszero}\,x)\,\textbf{0}\,(\textbf{add}\,y\,(\textbf{mult}\,(\textbf{pre}\,x)\,y))$
- Hence, $\textbf{mult} =_\beta \lambda xy.\textbf{cond}\,(\textbf{iszero}\,x)\,\textbf{0}\,(\textbf{add}\,y\,(\textbf{mult}\,(\textbf{pre}\,x)\,y))$
- And we have **mult** which really works like multiplication.

- One might still think that we could have kept to times and forget completely about **mult**.
- But then take **fact** which we intend to work as follows: $\mathbf{fact}\, x =_\beta \mathbf{cond}(\mathbf{iszero}\, x)\, \mathbf{1}\, (\mathbf{mult}\, x\, (\mathbf{fact}\, (\mathbf{pre}\, x)))$
- Assume $\mathbf{fact} \equiv \lambda x.\mathbf{cond}(\mathbf{iszero}\, x)\, \mathbf{1}\, (\mathbf{mult}\, x\, (\mathbf{fact}\, (\mathbf{pre}\, x)))$
- We see again that **fact** occurs on the left hand side and the right hand side of the equation.
- So, we are defining **fact** in terms of **fact**.
- So, again **fact**, like **mult** must be defined in terms of a fixed point operator.
- We define $\mathbf{factfn} \equiv \lambda zx.\mathbf{cond}(\mathbf{iszero}\, x)\, \mathbf{1}\, (\mathbf{mult}\, x\, (z\, (\mathbf{pre}\, x)))$
- So, we take $\mathbf{fact} \equiv \mathrm{Fix}\, \mathbf{factfn}$.
- By fixed point theorem we have: $\mathrm{Fix}\, \mathbf{factfn} =_\beta \mathbf{factfn}(\mathrm{Fix}\, \mathbf{factfn})$.

- $\mathbf{fact} \equiv \mathsf{Fix}\,\mathbf{factfn} =_\beta \mathbf{factfn}(\mathsf{Fix}\,\mathbf{factfn}) =_\beta \mathbf{factfn}(\mathbf{fact})$
- Hence, $\mathbf{fact} =_\beta \mathbf{factfn}(\mathbf{fact}) \equiv$ $(\lambda zx.\mathbf{cond}(\mathbf{iszero}\,x)\,\mathbf{1}\,(\mathbf{mult}\,x\,(z\,(\mathbf{pre}\,x))))(\mathbf{fact}) =_\beta$ $\lambda x.\mathbf{cond}(\mathbf{iszero}\,x)\,\mathbf{1}\,(\mathbf{mult}\,x\,(\mathbf{fact}\,(\mathbf{pre}\,x)))$
- So: $\mathbf{fact}\,x =_\beta \mathbf{cond}(\mathbf{iszero}\,x)\,\mathbf{1}\,(\mathbf{mult}\,x\,(\mathbf{fact}\,(\mathbf{pre}\,x)))$

- What is Fix? Is it unique? The answer is no. Fix is not unique.
- There are infinitely many fixed point operators.
- $Y_{Curry} \equiv \lambda x.(\lambda y.x(yy))(\lambda y.x(yy))$.
- Theorem: Y_{Curry} is a fixed point finder.
- Proof: $Y_{Curry}A \equiv (\lambda x.(\lambda y.x(yy))(\lambda y.x(yy)))A =_\beta$ $(\lambda y.A(yy))(\lambda y.A(yy)) =_\beta A((\lambda y.A(yy))(\lambda y.A(yy))) =_\beta$ $A(Y_{Curry}A)$.
- Hence Y_{Curry} is a fixed point operator.
- We also say that Y_{Curry} is a fixed point finder.
- We also say that Y_{Curry} is a fixed point combinator.

Foundations 1: F29FA1
Lecture 18

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

- What is Fix? Is it unique? The answer is no. Fix is not unique.
- There are infinitely many fixed point operators.
- $Y_{Curry} \equiv \lambda x.(\lambda y.x(yy))(\lambda y.x(yy))$.
- Theorem: Y_{Curry} is a fixed point finder.
- Proof: $Y_{Curry}A \equiv (\lambda x.(\lambda y.x(yy))(\lambda y.x(yy)))A =_\beta (\lambda y.A(yy))(\lambda y.A(yy)) =_\beta A((\lambda y.A(yy))(\lambda y.A(yy))) =_\beta A(Y_{Curry}A)$.
- Hence Y_{Curry} is a fixed point operator.
- We also say that Y_{Curry} is a fixed point finder.
- We also say that Y_{Curry} is a fixed point combinator.

- *Fixed point theorem:* In the λ-calculus, every λ-expression A has a fixed point A' such that $AA' =_\beta A'$
- The fixed point is found by a fixed point operator (say Fix) such that for any A, the fixed point of A is Fix A.
- Fix can be Y_{Curry}, or any other one of an infinite number of fixed point combinators.

- The fixed point theorem is powerful for recursive functions and equations.
- *Theorem:* In the λ-calculus, for any λ-expression A and for any $n \geq 0$, the equation $xy_1y_2\ldots y_n =_\beta A$ (where $y_i \not\equiv x$ for $1 \leq i \leq n$) can be solved for x.
- I.e., there is a B such that $By_1y_2\ldots y_n =_\beta A[x := B]$
- Proof: Let $B \equiv \mathrm{Fix}\,(\lambda xy_1y_2\ldots y_n.A)$.
  Hence $By_1y_2\ldots y_n \equiv (\mathrm{Fix}\,(\lambda xy_1y_2\ldots y_n.A))y_1y_2\ldots y_n =_\beta$
  $(\lambda xy_1y_2\ldots y_n.A)(\mathrm{Fix}\,(\lambda xy_1y_2\ldots y_n.A))y_1y_2\ldots y_n =_\beta$
  $A[x := \mathrm{Fix}\,(\lambda xy_1y_2\ldots y_n.A)][y_1 := y_1]\ldots[y_n := y_n] \equiv$
  $A[x := B][y_1 := y_1]\ldots[y_n := y_n] \equiv A[x := B]$.

Examples

- *Solve $xy =_\beta x$ in x.*
- Solution: Let $B \equiv Fix(\lambda xy.x)$.
- Now we prove that $By =_\beta B$ as follows:
  $By \equiv Fix(\lambda xy.x)y =_\beta^{\text{fixed point theorem}}$
  $(\lambda xy.x)(Fix(\lambda xy.x))y =_\beta Fix(\lambda xy.x) \equiv B$

Examples

- *Solve $xy =_\beta yx$ in x.*
- Solution: Let $B \equiv Fix(\lambda xy.yx)$.
- Now we prove that $By =_\beta yB$ as follows:
  $By \equiv Fix(\lambda xy.yx)y =_\beta^{\text{fixed point theorem}} (\lambda xy.yx)(Fix(\lambda xy.yx))y =_\beta y(Fix(\lambda xy.yx)) \equiv yB$.
- *Solve $zxy =_\beta xyz$ in z.*
- Solution: Let $B \equiv Fix(\lambda zxy.xyz)$.
- Now we prove that $Bxy =_\beta xyB$ as follows:
  $Bxy \equiv Fix(\lambda zxy.xyz)xy =_\beta^{\text{fixed point theorem}} (\lambda zxy.xyz)(Fix(\lambda zxy.xyz))xy =_\beta xy(Fix(\lambda zxy.xyz)) \equiv xyB$.

The fixed point theorem

- *Fixed point theorem:* In the λ-calculus, every λ-expression A has a fixed point A' such that $AA' =_\beta A'$
- The fixed point is found by a fixed point operator (say Fix) such that for any A, the fixed point of A is Fix A.
- Fix can be any one of an infinite number of fixed point combinators.
- $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ is a fixed point combinator
  - $YA \equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))A) =_\beta (\lambda x.A(xx))(\lambda x.A(xx)) =_\beta A((\lambda x.A(xx))(\lambda x.A(xx))) =_\beta A(YA)$.

- The fixed point theorem is powerful for recursion.
- *Corollary/Theorem:* In the λ-calculus, for any λ-expression A and for any $n \geq 0$, the equation $xy_1y_2\ldots y_n =_\beta A$ can be solved for x.
- There is a B such that $By_1y_2\ldots y_n =_\beta A[x := B]$
- *Example:Solve $xy =_\beta x$ in x.*
  - Solution: Let $B \equiv Y(\lambda xy.x)$.
  - Now we prove that $By =_\beta B$ as follows:

    $By \equiv Y(\lambda xy.x)y =_\beta^{\text{fixed point theorem}}$
    $(\lambda xy.x)(Y(\lambda xy.x))y =_\beta Y(\lambda xy.x) \equiv B$
- *Example: Solve $xy =_\beta yx$ in x.*
  - Solution: Let $B \equiv Y(\lambda xy.yx)$.
  - Now we prove that $By =_\beta yB$ as follows:

    $By \equiv Y(\lambda xy.yx)y =_\beta^{\text{fixed point theorem}}$
    $(\lambda xy.yx)(Y(\lambda xy.yx))y =_\beta y(Y(\lambda xy.yx)) \equiv yB$.

Y_{Klop} is a fixed point finder: $Y_{Klop}A \equiv A(Y_{Klop}A)$

1. Let $Y_{Klop} \equiv \underbrace{\$}_{26 \text{ times}}$ where $\$ \equiv$
   $\lambda \underbrace{abcdefghijklmnopqstuvwxyzr}_{26 \text{ arguments}} .r(\underbrace{thisisafixedpointcombinator}_{27 \text{ arguments}})$
2. $Y_{Klop}A \equiv \underbrace{\$}_{26 \text{ times}} A \equiv$
   $\$ \underbrace{\$}_{25 \text{ times}} A \equiv$
   $(\lambda \underbrace{abcdefghijklmnopqstuvwxyzr}_{26 \text{ arguments}} .r(\underbrace{thisisafixedpointcombinator}_{27 \text{ arguments}}))$
   $\underbrace{\$}_{25 \text{ times}} A$
   $=_\beta A(\underbrace{\$}_{26 \text{ times}} A) \equiv A(Y_{Klop}A).$

Exercises 5

Show that:

1. $\textbf{succ}\ \textbf{n} =_\beta \textbf{n} + \textbf{1}$
2. $\textbf{iszero}\ \textbf{0} =_\beta \textbf{true}$
3. $\textbf{iszero}(\textbf{succ}\ \textbf{n}) =_\beta \textbf{false}$
4. $\textbf{add}\ \textbf{n}\ \textbf{m} =_\beta \textbf{n} + \textbf{m}$
5. $\textbf{times}\ \textbf{n}\ \textbf{m} =_{\beta\eta} \textbf{nxm}$
6. $\textbf{prefn}\ y(\textbf{pair}\ z\ x) =_\beta \textbf{pair}\ \textbf{false}(\textbf{cond}\ z\ x(y\ x))$
7. $\textbf{prefn}\ y(\textbf{pair}\ \textbf{true}\ x) =_\beta \textbf{pair}\ \textbf{false}\ x$
8. $\textbf{prefn}\ y(\textbf{pair}\ \textbf{false}\ x) =_\beta \textbf{pair}\ \textbf{false}\ (y\ x)$
9. $(\textbf{prefn}\ y)^n(\textbf{pair}\ \textbf{false}\ x) =_\beta \textbf{pair}\ \textbf{false}\ (y^n x)$
10. $(\textbf{prefn}\ y)^n(\textbf{pair}\ \textbf{true}\ x) =_\beta \textbf{pair}\ \textbf{false}\ (y^{n-1} x)$ if $n > 0$
11. $\textbf{pre}(\textbf{succ}\ \textbf{n}) =_\beta \textbf{n}$
12. $\textbf{pre}\ \textbf{0} =_\beta \textbf{0}$

Exercise 6

Assume the following:

$\mathbf{0'} \equiv \lambda x.x$

$\mathbf{1'} \equiv \textbf{pair false } \mathbf{0'}$

$\mathbf{2'} \equiv \textbf{pair false } \mathbf{1'}$

...

$(\mathbf{n}+\mathbf{1})' \equiv \textbf{pair false } \mathbf{n'}$

1. Define **succ**′, **iszero**′, **pre**′ such that:
2. $\mathbf{succ'}\ \mathbf{n'} =_\beta (\mathbf{n}+\mathbf{1})'$
3. $\mathbf{iszero'}\ \mathbf{0'} =_\beta \textbf{true}$
4. $\mathbf{iszero'}(\mathbf{succ'}\ \mathbf{n'}) =_\beta \textbf{false}$
5. $\mathbf{pre'}(\mathbf{succ'}\ \mathbf{n'}) =_\beta \mathbf{n'}$.

- *Exercise 7* Solve $zxy =_\beta z$ in z.
- *Exercise 8* Construct a λ-term **eq** such that
  $$\mathbf{eq\ m\ n} =_\beta \mathbf{cond\ (iszero\ m)\ (iszero\ n)}$$
  $$\mathbf{(cond\ (iszero\ n)\ false\ (eq\ (pre\ m)\ (pre\ n)))}.$$
- *Exercise 9* Let Y be Y_{Curry} where
  $Y_{Curry} \equiv \lambda z.(\lambda x.z(xx))(\lambda x.z(xx))$ is a fixed point operator.
  Show that $Y_1 \equiv Y(\lambda yz.z(yz))$ is a fixed point operator.
- *Exercise 10* Let $Y_{Turing} \equiv ZZ$ where $Z \equiv \lambda zx.x(zzx)$. Show that Y_{Turing} is a fixed point combinator.
- *Exercise 11* Let $\$ \equiv$
  $\lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$.
  Let $Y_{Klop} \equiv \$$ (i.e., Y_{Klop} is a sequence of 26 $\$$s).
  Show that Y_{Klop} is a fixed point finder.

Foundations 1: F29FA1
Lecture 19

Lecturers:
Fairouz Kamareddine Edinburgh
and Adrian Turcanu Dubai

Lists

- Let us define lists as λ-expressions where $[\,]$ is the empty list.
- There does not exist a λ-expression null such that

$$\text{null } A =_\beta \begin{cases} \textbf{true} & \text{if } A =_\beta [\,] \\ \textbf{false} & \text{otherwise} \end{cases}$$

- *Proof* Assume null existed.
- Let $[\,]$ be the empty list and let l be a list such that $l \neq_\beta [\,]$.
- Let $\text{foo} \equiv \lambda x.\textbf{cond}\,(\text{null}\,x)\,l\,[\,]$.
- Let W be a solution in x of $x =_\beta \text{foo}\,x$.
- W exists by the corollary of the fixed point theorem.
- $W =_\beta \text{foo}\,W =_\beta \textbf{cond}\,(\text{null}\,W)\,l\,[\,]$.
- Case $W =_\beta [\,]$ then $(\text{null}\,W) =_\beta \textbf{true}$ and $W =_\beta l$. Absurd.
- Case $W \neq_\beta [\,]$ then $(\text{null}\,W) =_\beta \textbf{false}$ and $W =_\beta [\,]$. Absurd.

- Because null does not exist, we have to find a way to represent lists in a way which accommodates information of nullity in it.
- Let **null** $\equiv$ **fst**.
- Let $\perp$ be a solution to $xy =_\beta x$ in x.
- Let $[\,] \equiv$ **pair true** $\perp$
- Let $[E] \equiv$ **pair false** (**pair** E $[\,]$)
- Let $[E_1, E_2, \ldots, E_n] \equiv$ **pair false** (**pair** E_1 $[E_2, \ldots, E_n]$)
- Let **hd** $\equiv \lambda x.$**cond** (**null** x) $\perp$ (**fst**(**snd** x))
- Let **tl** $\equiv \lambda x.$**cond** (**null** x) $\perp$ (**snd**(**snd** x))
- Let **cons** $\equiv \lambda xy.$**pair false** (**pair** x y)
- Note that we did not use recursion for **cons**.

null $[\,] =_\beta$ true and null (cons x l) $=_\beta$ false

- null $[\,] \equiv$ fst $[\,] \equiv$ fst (pair true $\perp$) $=_\beta$ true.
- null (cons x l) $\equiv$ fst (cons x l) $\equiv$
  fst $((\lambda xy.$pair false (pair x y)$)x\ l) =_\beta$
  fst (pair false (pair x l)) $=_\beta$ false.

$\textbf{hd}\,(\textbf{cons}\; x\; l) =_\beta x$

- $\textbf{hd}\,(\textbf{cons}\, x\, l) \equiv (\lambda x.\textbf{cond}\,(\textbf{null}\, x) \perp (\textbf{fst}(\textbf{snd}\, x)))(\textbf{cons}\, x\, l) =_\beta$
  $\textbf{cond}\,(\textbf{null}\,(\textbf{cons}\, x\, l)) \perp (\textbf{fst}(\textbf{snd}\,(\textbf{cons}\, x\, l))) =_\beta$
  $\textbf{cond false} \perp (\textbf{fst}(\textbf{snd}\,(\textbf{cons}\, x\, l))) =_\beta \textbf{fst}(\textbf{snd}\,(\textbf{cons}\, x\, l)) \equiv$
  $\textbf{fst}(\textbf{snd}\,((\lambda xy.\textbf{pair false}\,(\textbf{pair}\, x\, y))\, x\, l)) =_\beta$
  $\textbf{fst}(\textbf{snd}\,(\textbf{pair false}\,(\textbf{pair}\, x\, l))) =_\beta \textbf{fst}(\textbf{pair}\, x\, l) =_\beta x.$

$\mathbf{tl}\ (\mathbf{cons}\ x\ l) =_\beta l$

- $\mathbf{tl}\,(\mathbf{cons}\,x\,l) \equiv (\lambda x.\mathbf{cond}\,(\mathbf{null}\,x)\perp(\mathbf{snd}(\mathbf{snd}\,x)))(\mathbf{cons}\,x\,l) =_\beta$
  $\mathbf{cond}\,(\mathbf{null}\,(\mathbf{cons}\,x\,l))\perp(\mathbf{snd}(\mathbf{snd}\,(\mathbf{cons}\,x\,l))) =_\beta$
  $\mathbf{cond}\,\mathbf{false}\perp(\mathbf{snd}(\mathbf{snd}\,(\mathbf{cons}\,x\,l))) =_\beta \mathbf{snd}(\mathbf{snd}\,(\mathbf{cons}\,x\,l)) \equiv$
  $\mathbf{snd}(\mathbf{snd}\,((\lambda xy.\mathbf{pair}\,\mathbf{false}\,(\mathbf{pair}\,x\,y))\,x\,l)) =_\beta$
  $\mathbf{snd}(\mathbf{snd}\,(\mathbf{pair}\,\mathbf{false}\,(\mathbf{pair}\,x\,l))) =_\beta \mathbf{snd}(\mathbf{pair}\,x\,l) =_\beta l.$

Append

- Define **append** which takes two lists and appends them together.
- For example, **append** $[1, 2]\,[3, 4] =_\beta [1, 2, 3, 4]$
- We want $\textbf{append}\,x\,y =_\beta \textbf{cond}\,(\textbf{null}\,x)\,y\,(\textbf{cons}\,(\textbf{hd}\,x)(\textbf{append}\,(\textbf{tl}\,x)\,y))$.
- This is a recursive equation. Let **append** be a solution in z to the equation: $z\,x\,y =_\beta \textbf{cond}\,(\textbf{null}\,x)\,y\,(\textbf{cons}\,(\textbf{hd}\,x)(z\,(\textbf{tl}\,x)\,y))$.
- **append** exists by the corollary of the fixed point theorem and $\textbf{append}\,x\,y =_\beta \textbf{cond}\,(\textbf{null}\,x)\,y\,(\textbf{cons}\,(\textbf{hd}\,x)(\textbf{append}\,(\textbf{tl}\,x)\,y))$.

Undecidability of Having a normal Form

- There is no λ-expression **hasnf** such that

$$\textbf{hasnf}\ A =_\beta \begin{cases} \textbf{true} & \text{if } A \text{ has a normal form} \\ \textbf{false} & \text{otherwise} \end{cases}$$

- *Proof:* Assume **hasnf** exists.
- Let $I \equiv \lambda x.x$ and $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$.
- I has a normal form and Ω does not have a normal form.
- By Church-Rosser, if $A =_\beta B$ then either both A and B have a normal form, or none of them has a anormal form.
- Let foo $\equiv \lambda x.\textbf{cond}\,(\textbf{hasnf}\ x)\,\Omega\,I$.
- Let W be a solution in z of $z = \text{foo}\,z$.
- W exists by the corollary of the fixed point theorems.
- $W =_\beta \text{foo}\,W =_\beta \textbf{cond}\,(\textbf{hasnf}\ W)\,\Omega\,I$.
- If $\textbf{hasnf}\,W =_\beta \textbf{true}$ then $W =_\beta \Omega$. Absurd by Church-Rosser.
- If $\textbf{hasnf}\,W =_\beta \textbf{false}$ then $W =_\beta I$. Absurd by Church-Rosser.

Undecidability of Halting

- Remember that A halts iff A has a normal form.
- Hence, there is no λ-expression **halts** such that

$$\textbf{halts}\ A =_\beta \begin{cases} \textbf{true} & \text{if } A \text{ halts} \\ \textbf{false} & \text{otherwise} \end{cases}$$

- Otherwise **halts** would be **hasnf** and we said that **hasnf** is not definable in the λ-calculus.
- Hence the λ-calculus does not allow the representation of the non-computable function **halts**.
- In fact, the λ-calculus only allows representing functions which are computable.

- *Exercise 12* Define **reverse** which takes a list and reverses the order of its elements. For example: **reverse** $[1, 2, 3] =_\beta [3, 2, 1]$.
- *Exercise 13* Show that the function **equal** below is undefinable as a λ-expression:

$$\textbf{equal}\ E_1\ E_2 =_\beta \begin{cases} \textbf{true} & \text{if } E_1 =_\beta E_2 \\ \textbf{false} & \text{otherwise} \end{cases}$$