Traditional and Non Traditional lambda calculi

Fairouz Kamareddine

July 2009

- *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* Competence in lambda calculus, different variable techniques (de Bruijn indices, combinator variables), 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.

The syntax of the λ-calculus

- 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.
- 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* or an *abstraction* or an *application*.
- We let $A, B, C \cdots$ range over $\mathcal{M}$.

Examples

- $(\lambda x.x)$
- $(\lambda y.y)$
- $(\lambda x.(xx))$
- $(\lambda x.(\lambda y.x))$
- $(\lambda x.(\lambda y.(xy)))$
- $((\lambda x.x)(\lambda x.x))$.
- $((\lambda x.(xx))(\lambda x.(xx)))$.

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 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 are 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.

The semantic function

- Let D be a model of the λ-calculus, $a \in D$ and $\sigma \in$ ENV. We define $\sigma(a/v)(v') = \begin{cases} a & \text{if } v = v' \\ \sigma(v') & \text{if } v \neq v' \end{cases}$
- We define $\parallel . \parallel$: $\mathcal{M}$**times**ENV $\mapsto D$ as follows:
- $\parallel v \parallel_\sigma = \sigma(v)$.
- $\parallel (AB) \parallel_\sigma = \parallel A \parallel_\sigma (\parallel B \parallel_\sigma)$.
- $\parallel (\lambda v.A) \parallel_\sigma = f : D \mapsto D$ where $f(a) = \parallel A \parallel_{\sigma(a/v)}$.

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 $(\lambda x.y)z$.

Syntactic identity

- 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 $x \not\equiv y$, $\lambda x.x \not\equiv \lambda y.y$.
- 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: $(\lambda v.A)B \to_\beta A[v := B]$.
- However, one has to be careful.

The meaning of $\lambda xy.xy$

- Recall that $x \not\equiv y$.
- $\| \lambda xy.xy \|_\sigma = f$ where $f(a) = \| \lambda y.xy \|_{\sigma(a/x)}$.
- But, $\| \lambda y.xy \|_{\sigma(a/x)} = g$ where $g(b) = \| xy \|_{\sigma(a/x)(b/y)}$.
  But
  $\| xy \|_{\sigma(a/x)(b/y)} = \| x \|_{\sigma(a/x)(b/y)} (\| y \|_{\sigma(a/x)(b/y)}) = a(b)$.
- Hence, $\| \lambda xy.xy \|_\sigma = f$ where $f(a) = g$ where $g(b) = a(b)$.
- Hence, $\| \lambda xy.xy \|_\sigma = f$ which if given two arguments a and b produces $a(b)$.

The meaning of $\lambda xz.xz$

- Recall that $x \not\equiv z$.
- $\| \lambda xz.xz \|_\sigma = f$ where $f(a) = \| \lambda z.xz \|_{\sigma(a/x)}$.
- But, $\| \lambda z.xz \|_{\sigma(a/x)} = g$ where $g(b) = \| xz \|_{\sigma(a/x)(b/z)}$.
  But
  $\| xz \|_{\sigma(a/x)(b/z)} = \| x \|_{\sigma(a/x)(b/z)} (\| z \|_{\sigma(a/x)(b/z)}) = a(b)$.
- Hence, $\| \lambda xz.xz \|_\sigma = f$ where $f(a) = g$ where $g(b) = a(b)$.
- Hence, $\| \lambda xz.xz \|_\sigma = f$ which if given two arguments a and b produces $a(b)$.

- Hence, the meaning of $\lambda xy.xy$ is equal to 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 $\lambda y.yy$ and of $\lambda z.yz$

- Recall that $y \not\equiv z$.
- $\parallel \lambda y.yy \parallel_\sigma = f$ where $f(a) = \parallel yy \parallel_{\sigma(a/y)} = a(a)$.
- $\parallel \lambda z.yz \parallel_\sigma = g$ where $g(a) = \parallel yz \parallel_{\sigma(a/z)} = \parallel y \parallel_{\sigma(a/z)} (a) = \parallel y \parallel_\sigma (a)$.
- Since $f(a) = a(a)$ and $g(a) = \parallel y \parallel_\sigma (a)$, obviously, $f \neq g$.
- Hence, the meaning of $\parallel \lambda y.yy \parallel_\sigma \neq$ the meaning of $\parallel \lambda z.yz \parallel_\sigma$

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 define the notions of *free* and *bound* 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 variables

- Take the two expressions x and $\lambda x.x$.
- In the second expression, the variable x is bound, so that the whole expression would not depend on x.
- In fact we could replace x by any other variable everywhere and would still get an expression with the same meaning. $\lambda x.x$ has the same meaning as $\lambda y.y$.
- In the expression x however, x is free and cannot be replaced by 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:

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

- An occurrence of v in A is free if it is not within the scope of a λ, otherwise it is bound.
- For example, 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.

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

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

- A *closed term* is a λ-term in which all variables are bound.

- Free and bound variables are important in the λ-calculus:
- 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.
  - We will see more on this when we will introduce α-conversion.
- Substitution has to be handled with care due to the distinct roles played by bound and free variables.
  - After substitution, no free variable can become bound.
  - For example, $(\lambda y.xy)[x := y]$ must not return $\lambda y.yy$, but something like $\lambda z.yz$.
  - $\lambda y.yy$ and $\lambda z.yz$ have different meanings.
  - $\lambda z.yz$ is obtained by renaming the bound y in $\lambda y.xy$ to z, and then performing the substitution.
- There is no point in substituting for a bound variable.

Recalling Free and Bound variables

- Recall the definition of FV and BV.
- For example:

$FV(x) = \{x\}$	$BV(x) = \emptyset$
$FV(\lambda x.x) = \emptyset$	$BV(\lambda x.x) = \{x\}$
$FV(\lambda x.y) = \{y\}$	$BV(\lambda x.y) = \{x\}$
$FV(\lambda yx.y) = \emptyset$	$BV(\lambda yx.y) = \{x, y\}$
$FV((\lambda x.y)(\lambda y.y)) = \{y\}$	$BV((\lambda x.y)(\lambda y.y)) = \{x, y\}$
$FV((\lambda x.x)x) = \{x\}$	$BV((\lambda x.x)x) = \{x\}$

- Note that a variable v can be in both $FV(A)$ and $BV(A)$.
- For example, $x \in FV((\lambda x.x)x)$ and $x \in BV((\lambda x.x)x)$.

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:

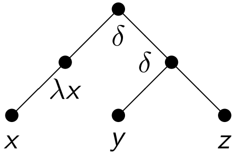


Figure: The tree of $(\lambda x.x)(yz)$

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

Scope and Occurrences

- We say that v is in the *scope* of λv in C if $\lambda v.A \in \mathrm{Subterms}(C)$ and $v \in FV(A)$.
- For example, take $\lambda xy.xy$.
  - y is in the scope of λy in $\lambda xy.xy$ because: $\lambda y.xy \in \mathrm{Subterms}(\lambda xy.xy)$ and $y \in FV(xy)$.
  - x is in the scope of λx in $\lambda xy.xy$ because: $\lambda xy.xy \mathrm{Subterms}(\lambda xy.xy)$ and $x \in FV(\lambda y.xy)$.
- We can talk about the *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 $(\lambda x.x)x$ but zero times in $\lambda x.y$.

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

Grafting

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

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

- Recall that taking $A[v := B]$ as grafting (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 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:

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

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 $\lambda y.(\lambda z.z)yy$.

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?

- 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.

- In the case when terms are identified modulo the names of their bound variables, then in the last clause of the above definition, 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 earlier definition of syntactic identity. 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: $(\lambda y.xy)[x := (\lambda z.z)y] \equiv \lambda x_3.(\lambda z.z)yx_3$.

Syntactic identity revised

- Now we review the definition of syntactic identity given in Lecture 2.
- We say that $A \equiv B$ 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.
- 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 $A' \equiv A[v := v']$.

Exercises

- 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. Simplify the following expressions:
    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. $x'(\lambda xyz.xz(yz))y'z'$
- 4. Write in SML, a recursive type of the expressions of the λ-calculus.
- 5. Write in SML, a function *free* which checks whether a variable is free in a λ-expression.
- 6. Write in SML, a function *freeVars* which finds the free variables of a λ-expression.

Exercises

- 7. 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. $(y(\lambda z.xz))[x := (\lambda y.zy)]$.
- 8. Write in SML, a function *subterms* which finds the subterms of a λ-expression.

- 9. Write in SML, a function *findme* which takes a variable v and a list l of variables and returns a new variable which is different from v and which does not occur in l.
- 10. Write in SML, a function *subs* which does substitution $A[v := B]$ as we defined it on unclean terms.
- 11. Run and test the SML functions we have written so far (i.e., free, freeVars, subterms, findme and subs).

- Recall the definition of terms: $\mathcal{M} ::= \mathcal{V} \mid (\lambda\mathcal{V}.\mathcal{M}) \mid (\mathcal{M}\mathcal{M})$.
- Recall our definition of FV and BV.
- Recall also that we take terms *modulo the name of bound variables.* I.e., $\lambda x.x \equiv \lambda y.y$.
- Now, BV does not make much sense anymore.
  - $BV(\lambda x.x) = \{x\}$ and $BV(\lambda y.y) = \{y\}$.
  - So, if $\lambda x.x \equiv \lambda y.y$, shouldn't $BV(\lambda x.x) = BV(\lambda y.y)$?
- Although $BV(A)$ does not make sense anymore, we can still speak of a bound occurrence of avariable.
- It is the occurrence that matter. So in $\lambda x.x^\circ$, the x° is bound.

Cleaning up terms

- Look at $(\lambda v'.A)[v := B]$.
  - If $v' \in FV(B)$, we can rename it to v''. We write $(\lambda v''.A[v' := v''])[v := B]$.
  - We can choose this v'' so that $v'' \notin FV(B)$.
  - For example, we can rename $(\lambda y.xy)[x := y]$ to $(\lambda z.xz)[x := y]$ where $z \notin FV(y)$.
  - Hence, in $(\lambda v'.A)[v := B]$, we can assume that $v' \notin FV(B)$.
  - We can even assume that in a substitution context $A[v := B]$, no variable occurs both free and bound.
- Also, in $(\lambda v'.A)[v := B]$, we can assume that $v \not\equiv v'$.
  - Otherwise, we can rename v' to another variable.
  - We can even assume that in $A[v := B]$, $\lambda v.C \notin \text{Subterms}(A)$.

Clean terms

- The above conventions for cleaning terms (are called the Barendregt convention).
- Cleaned up terms following the Barendregt convention are called *clean terms*.
- In clean terms, no variable is both free and bound.
- On clean terms, every substitution $A[x := B]$ is clean in that no variable occurs both free and bound.
- $(\lambda x.x)x$ is not clean because x occurs both as free and as bound.
- $(\lambda y.y)x$ is clean. No variable occurs both as free and as bound.

Substitution on clean terms

- On clean terms, we can simplify substitution.
- Clause 4 is no longer needed. We don't write $(\lambda v.A)[v := B]$.
- Clause 6 is no longer needed. Whenever we write $(\lambda v'.A)[v := B]$, we assume that $v \not\equiv v'$ and that $v' \not\in FV(B)$.
- *Now, substitution can be simplified (or cleaned) as follows:*
  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:

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

Why is the new clean definition of substitution correct?

- $(\lambda y.xy)[x := y]$ is not clean because y occurs as bound (in $\lambda y.xy$ and as free (in the y of $[x := y]$). We need to use instead the clean version $(\lambda z.xz)[x := y]$.
- With this clean version, we use clause 5' to substitute followed by clause 3.
  $(\lambda z.xz)[x := y] \equiv \lambda z.(xz)[x := y] \equiv \lambda z.yz$.
- Not only is clean substitution clearer and tidier, but it makes the proofs about the λ-calculus much simpler.

- So we have assumed that terms are equivalent up to the renaming of their bound variables. So, $\lambda x.x \equiv \lambda y.y$.
- If no further restrictions are imposed on our terms (i.e., variables can occur both free and bound in the same term), then we need to use the notion of substitution defined on the non-clean terms (clauses 1..6).
- If on the other hand, terms are assumed to be clean (as in the Barendregt convention) then substitution can be simplified so that clauses 4+5+6 are replaced by clause 5'.
- Note that in implementations, we cannot assume the terms are clean. There is no magic to automatically clean terms on a machine following the Barendregt convention.

The substitution Lemma

- We have an important lemma for substitution (which holds both for clean and unclean terms):
- *Lemma:* Let $A, B, C \in \mathcal{M},\ v,\ v', \in \mathcal{V}$.
  For $v \not\equiv v'$ and $v \not\in \mathtt{FV}(C)$, we have that:
  $A[v := B][v' := C] \equiv A[v' := C][v := B[v' := C]]$.
- The proof is by induction on the structure of A.
- Do this proof yourself and compare how easy it is if we use clean terms and check that it gets complicated if we don't use clean terms.
- For example: since $x \not\in FV((\lambda z.z)x_1)$ we have
  - $(xy)[x := \lambda z.yz][y := (\lambda z.z)x_1] \equiv (\lambda z.((\lambda z.z)x_1)z)((\lambda z.z)x_1)$

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*:
- $\dfrac{A \to_\alpha B}{AC \to_\alpha BC}$
- $\dfrac{A \to_\alpha B}{CA \to_\alpha CB}$
- $\dfrac{A \to_\alpha B}{\lambda x.A \to_\alpha \lambda x.B}$
- So $\lambda y.xy \to_\alpha \lambda z.xz$
- $(\lambda y.xy)y \to_\alpha (\lambda z.xz)y$
- $\lambda x.(\lambda y.xy)y \to_\alpha \lambda x.(\lambda z.xz)y$

Transitivity and reflexivity

- Now, look at $(\lambda y.xy)(\lambda z.z)$
- $(\lambda y.xy)(\lambda z.z) \to_\alpha (\lambda y_1.xy_1)(\lambda z.z)$
- $(\lambda y_1.xy_1)(\lambda z.z) \to_\alpha (\lambda y_1.xy_1)(\lambda z_1.z_1)$
- 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, we would like: $A \twoheadrightarrow_\alpha A$.

Alpha reduction

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

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

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

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

Beta reduction

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

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

- 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 $A, B, C, D \in \mathcal{M}$.
  1. If $C \rightarrow_\beta D$ then $A[x := C] \twoheadrightarrow_\beta A[x := D]$.
  2. If $A \rightarrow_\beta B$ then $A[x := C] \rightarrow_\beta B[x := C]$.
- Proof: 1. By induction on the structure of A.
  2. By induction on the derivation $A \rightarrow_\beta B$ using the substitution lemma.

Eta reduction

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

$$(\eta) \qquad \lambda v.Av \rightarrow_\eta A \qquad \text{for } v \not\in FV(A)$$

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

Let us summarize our reduction relations

- 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 $v \notin FV(A)$

- Let $r \in \{\beta, \alpha, \eta\}$. We said that: $\to_r$ 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.
  - $$\frac{A_1 \to_r A_2}{A \equiv A_1 C \to_r A_2 C \equiv B}$$
  - $$\frac{A_1 \to_r A_2}{}$$

Examples of $\rightarrow_r$ where r is β

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

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 compatibe:
    $\dfrac{A \twoheadrightarrow_r B}{AC \twoheadrightarrow_r BC}$ $\quad$ $\dfrac{A \twoheadrightarrow_r B}{CA \twoheadrightarrow_r CB}$ $\quad$ $\dfrac{A \twoheadrightarrow_r B}{\lambda x.A \twoheadrightarrow_r \lambda x.B}$

- 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 reflexive, transitive and symmetric relation which contains $\rightarrow_r$.
- $A =_r A \qquad \dfrac{A =_r B \quad B =_r C}{A =_r C} \qquad \dfrac{A =_r B}{B =_r A} \qquad \dfrac{A \rightarrow_r B}{A =_r B}$
- If $A =_r B$, we say that A and B are r-convertible.
- *Lemma:* $=_r$ is compatibe.
  - $\dfrac{A =_r B}{AC =_r BC} \qquad \dfrac{A =_r B}{CA =_r CB} \qquad \dfrac{A =_r B}{\lambda x.A =_r \lambda x.B}$
- 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 $A \to_\beta B$ 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.
- *Lemma [Extensionality]:* Assume $v \notin FV(A)$ and $v \notin FV(B)$. If $Av =_{\beta\eta} Bv$ then $A =_{\beta\eta} B$.
- Proof: Assume $v \notin FV(A)$, $v \notin FV(B)$ and $Av =_{\beta\eta} Bv$.
  - By compatibility, $\lambda v.Av =_{\beta\eta} \lambda v.Bv$.
  - $\lambda v.Av =_{\beta\eta} A$ by (η), since $v \notin FV(A)$
  - $\lambda v.Bv =_{\beta\eta} B$ by (η), since $v \notin FV(B)$
  - Hence, $A =_{\beta\eta} B$, since $=_{\beta\eta}$ is an equivalence relation.

In Normal Form

- We say that A is in β-normal form, if there are no β-redexes in A.
- $\lambda x.zx$ is in β-normal form.
- We say that A is in η-normal form, if 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, if 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

- Let $r \in \{\beta, \eta, \beta\eta\}$.
- We say that A has an r-normal form B if $A =_r B$ 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 and Strongly normalising terms

- A term A is *strongly r-normalising* if there are no infinite r-reduction sequences starting at A.
- $(\lambda x.xx)(\lambda x.xx)$ is not strongly β-normalising because: $(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx)$.......
- A term A is *weakly r-normalising* if there is a B in normal form such that $A \twoheadrightarrow_r B$.
- $(\lambda x.xx)(\lambda y.y)z$ is weakly β-normalising: $(\lambda x.xx)(\lambda y.y)z \twoheadrightarrow_\beta z$.
- Is $(\lambda z.y)((\lambda x.xx)(\lambda x.xx))$ weakly β-normalising?
- *Lemma:* If A is strongly r-normalising then A is weakly r-normalising and A has an r-normal form.

Exercises

- 1. For each of the following items, say whether it is clean or not. If not, say why not and give the clean version.
  1. $(\lambda xy.xy)(\lambda z.z)y$.
  2. $(\lambda xy.xy)(\lambda x.x)$.
  3. $(\lambda xy.xy)[y := z]$.
  4. $(\lambda z.yz)[y := z]$.
- 2. β-reduce the following until there are no more β-redexes:
  1. $(\lambda xyz.xyz)(\lambda x.xx)(\lambda x.x)x$
  2. $(\lambda xyz.xyz)(\lambda x.xx)(\lambda x.xx)x\ (\lambda x.xx)(\lambda x.xx)$
  3. $(\lambda x.z)((\lambda x.xx)(\lambda x.xx))$
- 3. Reduce $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x)$ until no β- or η-redexes remain.

- 4. Show that $\lambda zx.(\lambda y.y)x \rightarrow_\eta \lambda zy.y$.
- 5. Is $(\lambda z.y)((\lambda x.xx)(\lambda x.xx))$ weakly β-normalising? Is it strongly β-normalising? Explain your answer.
- 6. Write in SML a function beta_redex which checks whether a term is a β-redex.
- 7. Write in SML a function eta_redex which checks whether a term is a η-redex.
- 8. Write in SML a function hasbeta_redex which checks whether a term has a β-redex.
- 9. Write in SML a function haseta_redex which checks whether a term has a η-redex.

Propeties of terms

- Let $r \in \{\beta, \eta, \beta\eta\}$.
- Does every expression have a r-normal form?
  can we keep reducing an expression until we reach a normal form?
  Is every expression weakly r-normalising?
  Is every expression strongly r-normalising?
- Recall that an expression A is weakly r-normalising if $A \twoheadrightarrow_r B$ 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 normal forms 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 us to

- $\underline{(\lambda x.xx)(\lambda x.xx)}$ is not weakly β-normalising (and hence is not strongly β-normalising).
- We can reduce in different orders:
  $(\lambda y.\underline{(\lambda x.x)(\lambda z.z)})xy \to_\beta \underline{(\lambda y.\lambda z.z)x}y \to_\beta \underline{(\lambda z.z)y} \to_\beta y$ and
  $\underline{(\lambda y.(\lambda x.x)(\lambda z.z))x}y \to_\beta (\underline{(\lambda x.x)(\lambda z.z)})y \to_\beta \underline{(\lambda z.z)y} \to_\beta y$
- 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:
  $\underline{(\lambda y.z)((\lambda x.xx)(\lambda x.xx))} \to_\beta z$ yet
  $(\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_\beta (\lambda y.z)(\underline{(\lambda x.xx)(\lambda x.xx)}) \to_\beta \dots$
- A term may grow after reduction:

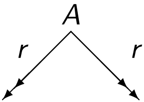
$$\begin{aligned}\underline{(\lambda x.xxx)(\lambda x.xxx)} &\to_\beta \underline{(\lambda x.xxx)(\lambda x.xxx)}(\lambda x.xxx)\\ &\to_\beta \underline{(\lambda x.xxx)(\lambda x.xxx)}(\lambda x.xxx)(\lambda x.xxx)\end{aligned}$$

- 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 (or normal form) 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, then those values, if they are in r-normal form 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.
- $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \to_\beta (\lambda yz.\underline{(\lambda x.x)z}(yz))(\lambda x.x) \to_\beta \underline{(\lambda yz.z(yz))(\lambda x.x)} \to_\beta \lambda z.z(\underline{(\lambda x.x)z}) \to_\beta \lambda z.zz$.
- $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \to_\beta \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)} \to_\beta \lambda z.\underline{(\lambda x.x)z}((\lambda x.x)z) \to_\beta \lambda z.z(\underline{(\lambda x.x)z}) \to_\beta \lambda z.zz$.
- $\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda x.x) \to_\beta \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)} \to_\beta \lambda z.(\lambda x.x)z(\underline{(\lambda x.x)z}) \to_\beta \lambda z.\underline{(\lambda x.x)z}z \to_\beta \lambda z.zz$.

Church-Rosser: 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: $(A \twoheadrightarrow_r B \wedge A \twoheadrightarrow_r C) \Rightarrow (B \twoheadrightarrow_r D \wedge C \twoheadrightarrow_r D)$.
- This theorem says that the results of reductions do not depend on the order in which they are done:



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

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

- In λ-calculus:

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

Corollaries

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

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

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

$$A \quad =_\beta \quad B$$

Corollaries 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 $A \equiv B$, 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. 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.
- Luckily we have a positive result to this question.

- That is, 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 which tells us that blind alleys in a reduction can be avoided by reducing the a kind of *leftmost* β-redex whose beginning λ is as far to the left as possible.
- 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.

Standardisation theorem

- A reduction path $A_0 \stackrel{R_0}{\to}_\beta A_1 \stackrel{R_1}{\to}_\beta A_2 \dots$ 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 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.
- A standard path, is a reduction path where one reduces from left to right.

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 if, 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

1. 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.
   1.1 $(\lambda x.xxx)(\lambda x.xx)(\lambda x.x)$
   1.2 $(\lambda x.xxx)(\lambda x.x)$
2. Now, it is urgent that you go to the lab and run and test all the SML functions you have written so far.

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

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

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:

Leftmost outermost leads to longer reductions paths than rightmost

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

Head β-normal forms

- A is in head β-normal form if and only if $A \equiv \lambda x_1 x_2..x_n.yA_1A_2..A_m$.
  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
- $\underline{(\lambda x.xxxx)((\lambda y.y)z)} \rightarrow_{\beta,h}$ $(\underline{(\lambda y.y)z})((\lambda y.y)z)((\lambda y.y)z)((\lambda y.y)z) \rightarrow_{\beta,h}$ $z((\lambda y.y)z)((\lambda y.y)z)((\lambda y.y)z)$

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:
  - $n(v) = \text{undefined}$
  - $n(\lambda v.A) = \text{undefined}$
  - $n(AB) = AB$ 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
- $(\lambda x.\lambda y.(\lambda z.z)xy)((\lambda x.x)x') \to_{\beta n}$

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

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

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
  - $rmv(\lambda v.A) =_{def} rmi(A)$
  - $rmv(AB) = rmv(B)$ 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

Exercises

- 1. 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. $(\lambda x.xxxx)((\lambda x.xxx)(\lambda x.xx))(\lambda x.x)$
    4. $(\lambda x.xxxx)((\lambda x.xxx)((\lambda x.xx)(\lambda x.x)))$

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 $\lambda(\lambda 1(\lambda 1))(\lambda 12)$.

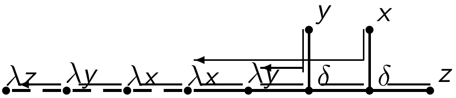
- Note that the above terms are all closed.
- What do we do if we had a term that has free variables?
- For example, how do we write $\lambda x.xz$ using de Bruijn's indices?
- In the presence of free variables, a *free variable list* which orders the variables must be assumed.
- For example, assume 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.

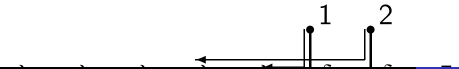
- $\lambda x.xz$ translates into $\lambda 14$.
- $(\lambda x.xz)y$ translates into $(\lambda 14)2$.
- $(\lambda x.xz)x$ translates into $(\lambda 14)1$.

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 $\lambda\lambda 521$





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 $(\lambda A)B \rightarrow_\beta ?$ 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 $\lambda 4$.

- 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:

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

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

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

- 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:

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

Exercises

- 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. $A \equiv (\lambda xy.xy)(\lambda z.zx)$.

- 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. $A \equiv (\lambda xy.xy)(\lambda z.zx)$.

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** $AB =_\beta B$

Representing pairing and projection in the λ-calculus

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

- 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 $1 \leq i \leq n$.

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

- $\mathbf{0} \equiv \lambda yx.x$
  $\mathbf{1} \equiv \lambda yx.yx$
  $\mathbf{2} \equiv \lambda yx.y(yx)$
  ...
  $\mathbf{n} \equiv \lambda yx.y^n x$ where $y^n x \equiv \underbrace{y(y(\dots(y}_{n \text{ times}}x)))$
  $\mathbf{succ} \equiv \lambda zyx.zy(yx)$
  $\mathbf{add} \equiv \lambda zz'yx.zy(z'yx)$
  $\mathbf{iszero} \equiv \lambda z.z(\lambda x.\mathbf{false})\mathbf{true}$
  $\mathbf{times} \equiv \lambda zyx.z(yx)$

Exercises

- 1. 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**
- 2. Show that $\textbf{posin}(\textbf{pair}\ A_1 \ldots (\textbf{pair}\ A_{n-1}\ A_n) \ldots)) =_\beta A_i$ for $1 \leq i \leq n$.

3.Show that:

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

4.Assume the following:

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

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

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

...

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

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

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

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

- But we should not depend on η.
- Can we define
  $\mathbf{mult} \equiv \lambda xy.\mathbf{cond}\ (\mathbf{iszero}\ x)\ \mathbf{0}\ (\mathbf{add}\ y\ (\mathbf{mult}\ (\mathbf{pre}\ x)\ y))$
- 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:
  $\mathrm{Fix}\,A =_\beta A(\mathrm{Fix}\,A)$.
- That is: $\mathrm{Fix}\,A$ is a fixed point of A.

- Find $\mathbf{mult} \equiv \lambda xy.\mathbf{cond}\,(\mathbf{iszero}\,x)\,\mathbf{0}\,(\mathbf{add}\,y\,(\mathbf{mult}\,(\mathbf{pre}\,x)\,y))$?
- 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 A, we have: $\mathrm{Fix}\,A =_\beta A(\mathrm{Fix}\,A)$. That is: $\mathrm{Fix}\,A$ is a fixed point of A.
- So, how do we use Fix to find **mult**?
- Define $\mathbf{multfn} \equiv \lambda zxy.\mathbf{cond}\,(\mathbf{iszero}\,x)\,\mathbf{0}\,(\mathbf{add}\,y\,(z\,(\mathbf{pre}\,x)\,y))$.
- Then, we define $\mathbf{mult} \equiv \mathrm{Fix}\,\mathbf{multfn}$.
- By Fixed point theorem, $\mathrm{Fix}\,\mathbf{multfn} =_\beta \mathbf{multfn}(\mathrm{Fix}\,\mathbf{multfn})$.
- Hence, $\mathbf{mult} \equiv \mathrm{Fix}\,\mathbf{multfn} =_\beta \mathbf{multfn}(\mathrm{Fix}\,\mathbf{multfn}) =_\beta \mathbf{multfn}(\mathbf{mult}) =_\beta \lambda xy.\mathbf{cond}(\mathbf{iszero}\,x)\mathbf{0}(\mathbf{add}\,y(\mathbf{mult}(\mathbf{pre}\,x)y))$
- Hence, $\mathbf{mult} =_\beta \lambda xy.\mathbf{cond}\,(\mathbf{iszero}\,x)\,\mathbf{0}\,(\mathbf{add}\,y\,(\mathbf{mult}\,(\mathbf{pre}\,x)\,y))$
- 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 $\mathbf{fact} \equiv \lambda x.\mathbf{cond}(\mathbf{iszero}\, x)\, \mathbf{1}\, (\mathbf{mult}\, x\, (\mathbf{fact}\, (\mathbf{pre}\, x)))$
- **fact** occurs on the left hand and right side of the equation.
- So, we are defining **fact** in terms of **fact**.
- **fact**, like **mult** must be defined by 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: $\mathrm{Fix}\, \mathbf{factfn} =_\beta \mathbf{factfn}(\mathrm{Fix}\, \mathbf{factfn})$.

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

Fix point Theorem

- 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 $xy_1y_2 \dots y_n =_\beta A$ (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 $By_1y_2 \dots y_n =_\beta A[x := B]$
- Proof: Let $B \equiv \mathrm{Fix}\,(\lambda xy_1y_2 \dots y_n.A)$.
  Hence $By_1y_2 \dots y_n \equiv (\mathrm{Fix}\,(\lambda xy_1y_2 \dots y_n.A))y_1y_2 \dots y_n =_\beta$
  $(\lambda xy_1y_2 \dots y_n.A)(\mathrm{Fix}\,(\lambda xy_1y_2 \dots y_n.A))y_1y_2 \dots y_n =_\beta$
  $A[x := \mathrm{Fix}\,(\lambda xy_1y_2 \dots y_n.A)][y_1 := y_1] \dots [y_n := y_n] \equiv$
  $A[x := B][y_1 := y_1] \dots [y_n := y_n] \equiv A[x := B]$.

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 $xy_1y_2 \dots y_n =_\beta A$ can be solved for x.
- There is a B such that $By_1y_2 \dots y_n =_\beta A[x := B]$
- *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:
    $By \equiv Y(\lambda xy.yx)y =_\beta^{\text{fixed point theorem}}$

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. $Y_{Klop}A \equiv \underbrace{\$}_{26 \text{ times}} A \equiv$
$\$ \underbrace{\$}_{25 \text{ times}} A \equiv$
$(\lambda \underbrace{abcdefghijklmnopqstuvwxyzr}_{26 \text{ arguments}}.r(\underbrace{thisisafixedpointcombinator}_{27 \text{ arguments}}))$

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.

- 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**.

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

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

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

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

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

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

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
  $\mathbf{append}\, x\, y =_\beta \mathbf{cond}\, (\mathbf{null}\, x)\, y\, (\mathbf{cons}\, (\mathbf{hd}\, x)(\mathbf{append}\, (\mathbf{tl}\, x)\, y))$.
- This is a recursive equation. Let **append** be a solution in z to the equation: $z\, x\, y =_\beta \mathbf{cond}\, (\mathbf{null}\, x)\, y\, (\mathbf{cons}\, (\mathbf{hd}\, x)(z\, (\mathbf{tl}\, x)\, y))$.
- **append** exists by the corollary of the fixed point theorem and $\mathbf{append}\, x\, y =_\beta \mathbf{cond}\, (\mathbf{null}\, x)\, y\, (\mathbf{cons}\, (\mathbf{hd}\, x)(\mathbf{append}\, (\mathbf{tl}\, x)\, y))$.

Undecidability of Having a normal Form

- There is no **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 $\text{foo} \equiv \lambda x.\textbf{cond}\ (\textbf{hasnf}\ x)\ \Omega\ I$.
- 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.

Exercises

- 1.Solve $zxy =_\beta z$ in z.
- 2. Construct a λ-term **eq** such that
  eq m n $=_\beta$ **cond** (**iszero m**) (**iszero n**)
  (**cond** (**iszero n**) **false** (**eq** (**pre m**) (**pre n**))).
- 3. 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.
- 4. Let $Y_{Turing} \equiv ZZ$ where $Z \equiv \lambda zx.x(zzx)$. Show that Y_{Turing} is a fixed point combinator.
- 5. Let $\$ \equiv$ $\lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$.

Exercises

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

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

Test One

Let $\mathbf{K} \equiv \lambda xy.x$, $\mathbf{S} \equiv \lambda xyz.xz(yz)$ and $\mathbf{B} \equiv \lambda xyz.x(yz)$. Simplify each of the following terms: i.e., for each N below, find the simplest possible M such that $N =_\beta M$.

- $\mathbf{B}XYZ$. [3]
  Solution: $\mathbf{B}XYZ \equiv (\lambda xyz.x(yz))XYZ =_\beta X(YZ)$.
- $\mathbf{SKSKSK}$. [4]
  Solution:
  $\mathbf{SKSKSK} \equiv (\lambda xyz.xz(yz))\mathbf{KSKSK} =_\beta \mathbf{KK(SK)SK} \equiv (\lambda xy.x)\mathbf{K(SK)SK} =_\beta \mathbf{KSK} \equiv (\lambda xy.x)\mathbf{SK} =_\beta \mathbf{S}$.

- Construct a λ-term F such that for any λ-terms M, N we have $FMN =_\beta M(NM)N$. [3]
  Solution: Let $F \equiv \lambda xy.x(yx)y$. Then $FMN =_\beta M(NM)N$.
- Construct a λ-term F such that for any λ-terms M, N and L, we have $FMNL =_\beta N(\lambda x.FM)(\lambda yz.yLM)$. [6]
  Solution: Let $E \equiv \lambda fmnl.n(\lambda x.fm)(\lambda yz.ylm)$. Then, take $F \equiv YE$ where Y is a fixed point operator. Hence, by fixed point theorem, $YE =_\beta E(YE)$. Hence, $F =_\beta EF$. Now, $FMNL =_\beta EFMNL \equiv (\lambda fmnl.n(\lambda x.fm)(\lambda yz.ylm))FMNL =_\beta N(\lambda x.FM)(\lambda yz.yLM)$.

- Let Y be a fixed point operator and let Y_1 be $Y(\lambda yf.f(yf))$. Show that Y_1 is a fixed point operator. [6]

  Solution: $Y_1 E \equiv (Y(\lambda yf.f(yf)))E \overset{F.P.theorem}{=_\beta}$
  $(\lambda yf.f(yf))(Y(\lambda yf.f(yf)))E =_\beta$
  $E((Y(\lambda yf.f(yf)))E) =_\beta E(Y_1 E)$.
- Is there a finite or an infinite number of fixed point operators? [3]

  Solution: Since by above we could take Y and make a new F.P. operator Y_1 which is different from Y, we can do the same with Y_1 to get Y_2 and the same with Y_2, etc. We can show that all these Y_i's are different. (I don't expect a formal proof of this as long as it is mentioned so the students are aware that they need to show the Y_i to be different).

- Explain what you understand by normal order reduction and by applicative order reduction. Compare these two reduction orders. [3]
  Solution: According to the call by value strategy, an argument is called only if it is a value (a normal form). According to the call by name strategy, an argument is called without first computing its value. Normal order reduction is guaranteed to reach a normal form if it exists. Applicative order however, might get stuck forever evaluating a term that is not strongly normalising (but may be normalising). For example, if normal order is used, $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$ will yield z; it will never terminate on the other hand, if applicative order is used. Applicative order however can reach a normal form faster than

- Reduce the following terms using first applicative order and then normal order. What can you deduce?
  1. $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$. [3]
     Solution:
     Applicative:
     $(\lambda y.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta (\lambda y.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta (\lambda y.z)((\lambda x.xx)(\lambda x.xx)) \dots$
     Normal: $(\lambda y.z)((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta z$.
  2. $(\lambda x.xx)((\lambda y.y)(\lambda z.z))$. [4]
     Solution:
     Applicative: $(\lambda x.xx)((\lambda y.y)(\lambda z.z)) \rightarrow_\beta (\lambda x.xx)(\lambda z.z) \rightarrow_\beta (\lambda z.z)(\lambda z.z) \rightarrow_\beta \lambda z.z$.
     Normal:
     $(\lambda x.xx)((\lambda y.y)(\lambda z.z)) \rightarrow_\beta ((\lambda y.y)(\lambda z.z))((\lambda y.y)(\lambda z.z)) \rightarrow_\beta (\lambda z.z)((\lambda y.y)(\lambda z.z)) \rightarrow_\beta (\lambda y.y)(\lambda z.z) \rightarrow_\beta \lambda z.z$.

- Let $\mathbf{I} \equiv \lambda x.x$. Simplify each of the following terms: i.e., for each N below, find the simplest possible M such that $N =_\beta M$.
  1. $(\lambda xyz.zyx)aa(\lambda pq.q)$. [3]
     Solution:
     $(\lambda xyz.zyx)aa(\lambda pq.q) =_\beta (\lambda pq.q)aa =_\beta a$.
  2. $(\lambda yz.zy)((\lambda x.xxx)(\lambda x.xxx))(\lambda w.\mathbf{I})$. [3]
     Solution:
     $(\lambda yz.zy)((\lambda x.xxx)(\lambda x.xxx))(\lambda w.\mathbf{I}) =_\beta$
     $(\lambda w.\mathbf{I})((\lambda x.xxx)(\lambda x.xxx)) =_\beta I$.

- The λ-calculus à la de Bruijn is given by: $\Lambda ::= \mathbb{N}|AB|\lambda A$ where numbers refer to the binding λ. For example,
  $\lambda 1$ represents $\lambda x.x$,
  $\lambda\lambda 12$ represents $\lambda x.\lambda y.yx$,
  $\lambda\lambda 21$ represents $\lambda x.\lambda y.xy$, etc.
  Write down what the following terms represent:
  1. $\lambda\lambda\lambda 123$ [2]
     Solution: $\lambda x.\lambda y.\lambda z.zyx$
  2. $\lambda\lambda 1$ [2]
     Solution: $\lambda x.\lambda y.y$
  3. $\lambda\lambda\lambda 13(23)$ [2]
     Solution: $\lambda x.\lambda y.\lambda z.zx(yx)$
  4. $\lambda\lambda\lambda 1(23)$. [2]
     Solution: $\lambda x.\lambda y.\lambda z.z(yx)$.
  5. Can you think of a way to deal with terms like $\lambda 23$ [1]

- Consider the following definitions:
  let $\mathbf{0} \equiv \lambda fx.x$
  let $\mathbf{1} \equiv \lambda fx.fx$
  let $\mathbf{2} \equiv \lambda fx.f(fx)$
  ...
  let $\mathbf{n} \equiv \lambda fx.f(f(\dots(fx)\dots))$ where f is applied n times to x
  let $\mathbf{succ} \equiv \lambda nfx.nf(fx)$
  let $\mathbf{true} \equiv \lambda xy.x$
  let $\mathbf{false} \equiv \lambda xy.y$
  let $\mathbf{iszero} \equiv \lambda n.n(\lambda x.\mathbf{false})\mathbf{true}$
  1. Show $\mathbf{iszero}\,\mathbf{0} =_\beta \mathbf{true}$ and $\mathbf{iszero}(\mathbf{succ}\,\mathbf{n}) =_\beta \mathbf{false}$. [6]
     Solution: $\mathbf{iszero}\,\mathbf{0} \equiv (\lambda n.n(\lambda x.\mathbf{false})\mathbf{true})\mathbf{0} =_\beta$ $\mathbf{0}(\lambda x.\mathbf{false})\mathbf{true} =_\beta (\lambda fx.x)(\lambda x.\mathbf{false})\mathbf{true} =_\beta \mathbf{true}$.
     And $\mathbf{iszero}(\mathbf{succ}\,\mathbf{n}) \equiv (\lambda n.n(\lambda x.\mathbf{false})\mathbf{true})(\mathbf{succ}\,\mathbf{n}) =_\beta$

2. Can you evaluate **iszero true** to either **true** or **false**? [3]
   Solution: No!
   iszero true $\equiv (\lambda n.n(\lambda x.\mathbf{false})\mathbf{true})\mathbf{true} =_\beta$
   $\mathbf{true}(\lambda x.\mathbf{false})\mathbf{true} \equiv$
   $(\lambda xy.x)(\lambda x.\mathbf{false})\mathbf{true} =_\beta \lambda x.\mathbf{false}$
   which is different from **true** and from **false**.
3. Let E be a λ-term. Can you evaluate **iszero** E to either **true** or **false**? [2]
   Solution: No not always. Take the above item as an example where $E \equiv$ **true**.

4. Show that we cannot define in the λ-calculus the λ-term **zero** such that:

$$\mathbf{zero}\ E =_\beta \begin{cases} \mathbf{true} & \text{if } E =_\beta \mathbf{0} \\ \mathbf{false} & \text{otherwise} \end{cases}$$

[5]

Solution: Assume **zero** is λ-definable.
Let $E \equiv \lambda x.\mathbf{cond}(\mathbf{zero}\ x)\mathbf{n}\ \mathbf{0}$ where $\mathbf{n} \neq_\beta \mathbf{0}$.
Let W be a solution for $x =_\beta Ex$. Hence,
$W =_\beta EW =_\beta \mathbf{cond}(\mathbf{zero}\ W)\mathbf{n}\ \mathbf{0}$.

- Case $W =_\beta \mathbf{0}$ then $W =_\beta \mathbf{cond\ true\ n\ 0} =_\beta \mathbf{n} \neq_\beta \mathbf{0}$. Absurd.
- Case $W \neq_\beta \mathbf{0}$ then $W =_\beta \mathbf{cond\ false\ n\ 0} =_\beta \mathbf{0}$. Absurd.

Hence **zero** is not λ-definable.

5. What is the difference between **zero** and **iszero**? [3]
   Solution: **zero** tests every element giving us only **true** or **false**.
   iszero only gives **true** or **false** for numbers.
6. Recall that the λ-term **cond** works as follows:
   cond true $E_1 E_2 =_\beta E_1$
   cond false $E_1 E_2 =_\beta E_2$
   Find a λ-term **eq** such that:
   eq $m\ n =_\beta$
   $\mathbf{cond}(\mathbf{iszero}\,m)(\mathbf{iszero}\,n)(\mathbf{cond}(\mathbf{iszero}\,n)\mathbf{false}(\mathbf{eq}(\mathbf{pre}\,m)(\mathbf{pre}\,n)))$. [6]

   Solution: Let $E \equiv$
   $\lambda fmn.\mathbf{cond}(\mathbf{iszero}\,m)(\mathbf{iszero}\,n)(\mathbf{cond}(\mathbf{iszero}\,n)\mathbf{false}(f(\mathbf{pre}\,m)(\mathbf{pre}\,n)))$

Test Two

- For each of the following terms, find its normal form if it exists. Otherwise, show that the term does not have a normal form.
  1. $(\lambda x.xxx)(\lambda x.xx)(\lambda x.x)$ [3]
     Solution: $\underline{(\lambda x.xxx)(\lambda x.xx)}(\lambda x.x) \to_\beta$
     $\underline{(\lambda x.xx)(\lambda x.xx)}(\lambda x.xx)(\lambda x.x) \to_\beta$
     $\underline{(\lambda x.xx)(\lambda x.xx)}(\lambda x.xx)(\lambda x.x) \ldots.$
     This is an infinite reduction. This is the only way we can reduce the term. Hence, the term does not have a normal form.
  2. $(\lambda x.xxx)(\lambda x.x)$ [2]
     Solution: $\underline{(\lambda x.xxx)(\lambda x.x)} \to_\beta$
     $(\lambda x.x)(\lambda x.x)(\lambda x.x) \to_\beta$

- For each of the following statements, give two terms M_1 and M_2 which satisfy it, explaining why each of M_1, M_2 and $M_1 M_2$ has (or does not have) a normal form.
  1. M_1 and M_2 have normal forms but $M_1 M_2$ does not. [3]
     Solution: $M_1 \equiv \lambda x.xx \equiv M_2$. Both M_1 and M_2 are in normal form since they have no β-redexes.
     $M_1 M_2 \rightarrow_\beta M_1 M_2 \rightarrow_\beta M_1 M_2 \rightarrow_\beta \ldots$. Since this is the only possible reduction sequence from $M_1 M_2$, we conclude that $M_1 M_2$ does not have a normal form.

2. $M_1 M_2$ has a normal form but M_1 does not. [3]
   Solution: Take $M_1 \equiv \lambda x.\mathrm{Cond}(\mathrm{iszero}\, x)1((\lambda x.xx)(\lambda x.xx))$ and $M_2 \equiv 0$. Obviously $M_1 M_2 \twoheadrightarrow_\beta 1$ and hence, has a normal form. But M_1 does not have a normal form since by the above item, $M_1 \to_\beta M_1 \to_\beta \dots$ and there are no reductions of M_1 which will contract $(\lambda x.xx)(\lambda x.xx)$ to a normal form or to contract $M_1 M_2$ to get rid of $(\lambda x.xx)(\lambda x.xx)$.
3. $M_1 M_2$ has a normal form but M_2 does not. [3]
   Solution: Take $M_1 \equiv \lambda x.1$ and $M_2 \equiv (\lambda x.xx)(\lambda x.xx)$. $M_1 M_2 \to_\beta 1$ in normal form, but M_2 does not have a normal form by 1. above.

- Give all the possible ways to reduce $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x)$ to normal form. [5]
  Solution:
  - $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x) \to_\beta (\lambda yz.(\lambda x.x)z(yz))(\lambda x.x) \to_\beta (\lambda yz.z(yz))(\lambda x.x) \to_\beta \lambda z.z((\lambda x.x)z) \to_\beta \lambda z.zz.$
  - $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x) \to_\beta (\lambda yz.(\lambda x.x)z(yz))(\lambda x.x) \to_\beta \lambda z.(\lambda x.x)z((\lambda x.x)z) \to_\beta \lambda z.z((\lambda x.x)z) \to_\beta \lambda z.zz.$
  - $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda x.x) \to_\beta (\lambda yz.(\lambda x.x)z(yz))(\lambda x.x) \to_\beta \lambda z.(\lambda x.x)z((\lambda x.x)z) \to_\beta \lambda z.(\lambda x.x)zz \to_\beta \lambda z.zz.$

- Let $I \equiv \lambda x.x$ and $S \equiv \lambda xyz.xz(yz)$.
  Find a λ-term M such that $MS =_\beta MISS$. [6]
  Solution: We solve $zy =_\beta zIyy$ in z. Let $E \equiv \lambda zy.zIyy$ and let $M \equiv \text{Fix}\, E$. Hence, by the fixed point operator,
  $MS =_\beta EMS \equiv (\lambda zy.zIyy)MS =_\beta MISS$.
- Explain the fixed point theorem and how it helps solve recursive equations.
  Solution: The fixed point theorem states that there is a fixed point operator Fix such that for any expression E, we have $\text{Fix}\, E =_\beta E(\text{Fix}\, E)$.
  To solve in x a recursive equation $xx_1 \dots x_n =_\beta \Phi$, we take the expression E to be $\lambda xx_1 \dots x_n.\Phi$ and then we know by the fixed point theorem that there is a fixed point X of E such

- Find an X such that $Xx = X$. [3]
  Solution: Let $E \equiv \lambda yx.y$ and let by the fixed point theorem, $X \equiv \mathrm{Fix}\, E$. Hence, by the fixed point operator, $X =_\beta EX$. Hence, $Xx =_\beta EXx \equiv (\lambda yx.y)Xx =_\beta X$.
- Solve in x the equation $xy =_\beta yx$. [5]
  Solution: Let $E \equiv \lambda xy.yx$ and let by the fixed point theorem, $X \equiv \mathrm{Fix}\, E$. Hence, by the fixed point operator, $X =_\beta EX$. Hence, $Xy =_\beta EXy \equiv (\lambda xy.yx)Xy =_\beta yX$.

- Take the Turing fixed point operator $\Theta \equiv (\lambda xy.y(xxy))(\lambda xy.y(xxy))$.
  Show that for Θ, we have indeed that $\Theta M \to_\beta M(\Theta M)$. [4]
  Solution: $\Theta M \equiv (\lambda xy.y(xxy))(\lambda xy.y(xxy))M \to_\beta$ $(\lambda y.y((\lambda xy.y(xxy))(\lambda xy.y(xxy))y))M \to_\beta$ $M((\lambda xy.y(xxy))(\lambda xy.y(xxy))M) \equiv M(\Theta M)$.
- Let $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. Is it the case that $YM \twoheadrightarrow_\beta M(YM)$? If yes, give the reduction steps from YM to $M(YM)$. If no, say why not. [2]
  Solution: No. $YM \to_\beta (\lambda x.M(xx))(\lambda x.M(xx)) \to_\beta$ $M((\lambda x.M(xx))(\lambda x.M(xx)))$. It is not clear how we can get from here by $\twoheadrightarrow_\beta$ to $M(YM)$. No formal proof is expected to be given here.

- Give the outermost redex and the innermost redex of each of the following (careful, you are asked for the outermost and not the leftmost outermost; also, you are asked for the innermost and not the rightmost innermost):
    1. $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$. [3]
    Solution: The outermost redex is $(\lambda y.z)((\lambda x.xx)(\lambda x.xx))$.
    The innermost redex is $((\lambda x.xx)(\lambda x.xx))$.
    2. $(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)$. [3]
    Solution: The outermost redex is $(\lambda yz.(\lambda x.x)z(yz))(\lambda x.x)$.
    The innermost redex is $(\lambda x.x)z$.

- De Bruijn wrote the λ-calculus in a different way. He wrote the argument before the function and used $[x]$ instead of λx. Here is the translation from the classical λ-calculus you studied into de Bruijn's notation via $\mathcal{I}$.

$$\mathcal{I}(v) =_{def} v,$$

$$\mathcal{I}(\lambda v.B) =_{def} [v]\mathcal{I}(B),$$

$$\mathcal{I}(AB) =_{def} (\mathcal{I}(B))\mathcal{I}(A)$$

De Bruijn called items of the form (A) and $[v]$ *applicator wagon* respectively *abstractor wagon*, or simply *wagon*.

- Translate the following terms in de Bruijn's notation giving for each translation the abstractor wagons as well as the applicator wagons.
  1. $(\lambda x.(\lambda y.xy))z$. [5]
     Solution: $(\lambda x.(\lambda y.xy))z$ translates to $(z)[x][y](y)x$.
     Abstractor wagons are: $[x]$ and $[y]$.
     Applicator wagons are: (z) and (y).
  2. $(\lambda x.(\lambda y.\lambda z.zD)C)BA$. [7]
     Solution: $(\lambda x.(\lambda y.\lambda z.zD)C)BA$ translates to $(A')(B')[x](C')[y][z](D')z$.
     Abstractor wagons are: $[x]$, $[y]$ and $[z]$.
     Applicator wagons are: (A'), (B'), (C'), (D') and those inside them.

- De Bruijn also wrote the substitution $A[v := B]$ as $[v := B]A$. In de Bruijn's notation, the β-rule $(\lambda v.A)B \to_\beta A[v := B]$ becomes: $(B)[v]A \to_\beta [v := B]A$
  1. Give the β-redexes in both the classical and in the de Bruijn's notation for each of the terms $(\lambda x.(\lambda y.xy))z$ and $(\lambda x.(\lambda y.\lambda z.zD)C)BA$. Say which redexes in the classical notation correspond to which redexes in de Bruijn's notation. [6]
     Solution:
     In $(\lambda x.(\lambda y.xy))z$, the only β-redex is:
     - in classical: $(\lambda x.(\lambda y.xy))z$.
     - in de Bruijn's: $(z)[x][y](y)x$.

     The redexes correspond to one-another.
     In $(\lambda x.(\lambda y.\lambda z.zD)C)BA$, the β-redexes are:
     - in classical: $(\lambda x.(\lambda y.\lambda z.zD)C)B$ and $(\lambda y.\lambda z.zD)C$.
     - in de Bruijn's: $(B')[x](C')[y][z](D')z$ and $(C')[y][z](D')z$

2. What do you notice about redexes in de Bruijn's notation? [3]
Solution: We notice that a redex is a lot clearer in de Bruijn's notation. Note for example how in $(\lambda x.(\lambda y.xy))z$ the z is away from its matching λx, whereas in $(z)[x][y](y)x$, the wagons (z) and $[x]$ are next to each other.

3. Assume A, B, C, D and AD are in normal forms. Reduce to normal forms each of the terms $(\lambda x.(\lambda y.\lambda z.zD)C)BA$ and $(A')(B')[x](C')[y][z](D')z$ in both the classical usual notation and in de Bruijn's notation using at each step, the outermost redex. [6]

   Solution:

Classical Notation	De Bruijn's Notation
$(\underline{\overset{\circ}{\lambda} x\,.(\overset{+}{\lambda} y\,.\,\overset{-}{\lambda} z\,.zD)\;\overset{+}{C})\;\overset{\circ}{B})}\;\overset{-}{A}$	$\overset{-}{(A')}\;\underline{\overset{\circ}{(B')}\overset{\circ}{[x]}}\;\overset{+}{(C')}\overset{+}{[y]}\overset{-}{[z]}\;(D')z$
$\downarrow_\beta$	$\downarrow_\beta$
$(\underline{(\overset{+}{\lambda} y\,.\,\overset{-}{\lambda} z\,.zD)\;\overset{+}{C}})\;\overset{-}{A}$	$\overset{-}{(A')}\;\underline{\overset{+}{(C')}\overset{+}{[y]}}\;\overset{-}{[z]}\;(D')z$
$\downarrow_\beta$	$\downarrow_\beta$

4. We define in de Bruijn's calculus, a segment to be a sequence (possibly empty) of wagons. For example, $(y)[x][z](A)$ is a segment.
   We say that a segment S is well-balanced if and only if either $S = \emptyset$ or $S = (M)S_1[v]S_2$ where S_1 and S_2 are well-balanced.
   For example, $(y)(z)(x)[y][z][x]$ is well-balanced.
   Give the well-balanced segments of the translations you gave above for $(\lambda x.(\lambda y.xy))z$ and $(\lambda x.(\lambda y.\lambda z.zD)C)BA$. [7]
   Solution: $(z)[x]$ is the only well-balanced segment in the translation of $(\lambda x.(\lambda y.xy))z$.
   $(A')(B')[x](C')[y][z]$, $(B')[x](C')[y]$, $(B')[x]$, $(C')[y]$ are the only well-balanced segments in the translation of $(\lambda x.(\lambda y.\lambda z.zD)C)BA$.