

CM20167
Topic 9: Reduction, Evaluation, Equivalence

Guy McCusker

1W2.1

Before we go on . . .

We're about to embark on developing the core theory of the λ -calculus.

Before we continue, let's introduce a little helpful sloppiness, and an alternative (and perhaps easier to manage) definition of substitution.

From now on, we will *identify λ -terms up to α -equivalence*. That is, if two terms are α -equivalent, we'll consider them as being the same.

What we're really doing here is working with α -equivalence classes of terms, instead of terms themselves. The λ -terms we write down are really *representatives* of their equivalence classes.

Theoretically this is mildly tricky but we won't go into the problems. Suffice to say that we completely ignore differences in bound variables.

Variable convention

Since we're ignoring the difference between α -equivalent terms, when we write down λ -terms, we are free to choose and rename bound variables at will.

It makes things easier if we adopt the *variable convention*:

Variable convention

When considering a collection of λ -terms M_1, \dots, M_n , we always choose the bound variables to be different from all the free variables.

This convention was introduced by Barendregt in his authoritative book on the λ -calculus.

Alternative definition of substitution

Once we've adopted the variable convention, we can give an alternative, simpler-looking definition of substitution:

$$\begin{aligned}x[M/x] &= M \\y[M/x] &= y && \text{if } x \neq y \\(\lambda y.M)[N/x] &= \lambda y.(M[N/x]) \\(M_1 M_2)[N/x] &= (M_1[N/x])(M_2[N/x])\end{aligned}$$

The variable-capture problem has gone away! In the definition of substituting into an abstraction, thanks to the variable convention, we know that y is not free in N , so it cannot be accidentally captured.

Exercise

Another line has also vanished from our definition: the one that said $(\lambda x.M)[N/x] = \lambda x.M$. Why has it disappeared?

De Bruijn notation

There's a third approach to dealing with α -conversion and variable capture, which uses an alternative syntax to avoid the problems altogether.

The alternative syntax uses *de Bruijn indices*: instead of naming variables, we just use numbers to say "how many lambdas away" the variable we are referring to is.

For instance:

- ▶ $\lambda x.x$ becomes $\lambda.1$
- ▶ $\lambda x.\lambda y.x$ becomes $\lambda.\lambda.2$

and so on.

The syntax of the language in this presentation is:

$$M ::= n \mid MM \mid \lambda.M$$

where n ranges over the natural numbers.

de Bruijn notation

We won't use the de Bruijn notation very much in this course, but if you're intrigued, it's quite interesting to work out how it goes: essentially it is a programming problem.

Exercise

Define substitution, and hence β -reduction, in this notation. It's not so easy, but at least there's no α -conversion.

Multiple redexes

Now, back to the theory of the λ -calculus. Remember, we're using ordinary syntax, identifying terms up to α -equivalence, and adopting the variable convention.

The key computation step in the λ -calculus is the β -reduction

$$(\lambda x.M)N \rightarrow_{\beta} M[N/x].$$

The definition of the λ -calculus allows us to perform β -reductions anywhere we find a β -redex: any subterm of the correct form can be reduced "in place".

For example, in a term like

$$(\lambda x.\lambda y.x)((\lambda z.zz)(\lambda w.w))$$

there are two redexes: the $\lambda x \dots$ function can be reduced, as can the $\lambda z \dots$

Reduction inside terms, precisely

We can make precise what we mean by "reducing any subterm" using an *inductive definition*. We give a collection of *rules* of the form

$$\frac{\text{some hypotheses}}{\text{a conclusion}}$$

The way to read a rule like this is: if all the hypotheses hold, then the conclusion holds too.

We can use a collection of rules like this to define the relation of β -reduction precisely:

$$\frac{}{(\lambda x.M)N \rightarrow_{\beta} M[N/x]}$$

$$\frac{M \rightarrow_{\beta} N}{MP \rightarrow_{\beta} NP} \quad \frac{M \rightarrow_{\beta} N}{PM \rightarrow_{\beta} PN} \quad \frac{M \rightarrow_{\beta} N}{\lambda x.M \rightarrow_{\beta} \lambda x.N}$$

Interpreting these rules

The basic rule

$$\frac{}{(\lambda x.M)N \rightarrow_{\beta} M[N/x]}$$

is an *axiom*: because there's nothing above the line (no hypotheses), the conclusion always holds. So this rule says that the basic β -reductions we've defined are always available.

The rule

$$\frac{M \rightarrow_{\beta} N}{\lambda x.M \rightarrow_{\beta} \lambda x.N}$$

says that *if* you can β -reduce M to N , then you can reduce $\lambda x.M$ to $\lambda x.N$. This is what lets us perform reductions like

$$\lambda x.((\lambda y.y)M) \rightarrow_{\beta} \lambda x.M$$

by "reducing under the λ ".

Those are all the reductions

An inductive definition like this one says:

- ▶ some reductions can always take place (the axiom)
- ▶ if we know that certain reductions are valid, then certain other reductions are valid (apply the rules)
- ▶ *and no other reductions are valid.*

That is, if you can't produce a proof like the one above for the validity of a reduction, then it's not a valid reduction.

All we've really done, though, is to make precise the idea of "doing the basic beta step on subterms".

Other inductive definitions

Our definition of the syntax of the λ -calculus was an inductive definition too. Instead of writing

$$M ::= x \mid \lambda x.M \mid MM$$

we could have said: given a set Var of variables, the set Λ of λ -terms is inductively defined by the following rules

$$\frac{}{x \in \Lambda} \quad x \in \text{Var} \quad \frac{M \in \Lambda}{\lambda x.M \in \Lambda} \quad x \in \text{Var} \quad \frac{M \in \Lambda \quad N \in \Lambda}{MN \in \Lambda}$$

Defining \rightarrow_{β}^* out of \rightarrow_{β}

We can also use this inductive technique to define \rightarrow_{β}^* precisely:

$$\frac{M \rightarrow_{\beta} N}{M \rightarrow_{\beta}^* N} \quad \frac{}{M \rightarrow_{\beta}^* M} \quad \frac{M \rightarrow_{\beta}^* N \quad N \rightarrow_{\beta}^* P}{M \rightarrow_{\beta}^* P}$$

The first rule says that any single step of reduction is included in \rightarrow_{β}^* ; the second rule (which is an axiom) says that \rightarrow_{β}^* includes the "zero steps" reduction of a term to itself; and the final rule makes the relation transitive.

Exercise

Convince yourself that these rules mean that $M \rightarrow_{\beta}^* N$ if and only if there is a (possibly empty) finite sequence of \rightarrow_{β} steps leading from M to N .

Which redex should we choose?

Since we can reduce any subterm of a λ -term which happens to be a redex, there can be more than one reduction from a given λ -term.

This raises a question: when there are several possible redexes to reduce, which one should we work on?

Does it make any difference?

What does “make a difference” mean anyway?

Example

Let's look at the example term from a few slides back in more detail:

$$(\lambda x. \lambda y. x)((\lambda z. zz)(\lambda w. w)).$$

If we reduce the $\lambda x. \dots$ redex, we get

$$\lambda y. ((\lambda z. zz)(\lambda w. w))$$

Now there is only one redex, the $\lambda z. \dots$, which can be reduced to

$$\lambda y. ((\lambda w. w)(\lambda w. w))$$

Again there is one redex, the first $\lambda w. \dots$, and reducing it gives

$$\lambda y. \lambda w. w$$

What if we choose the other redex first?

Example

$$(\lambda x. \lambda y. x)((\lambda z. zz)(\lambda w. w))$$

If we reduce the $\lambda z. \dots$, we get

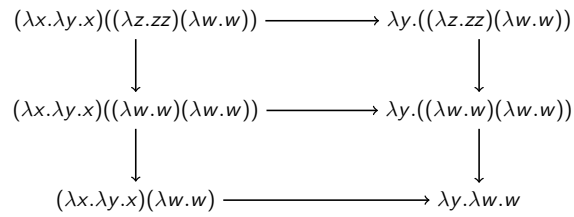
$$(\lambda x. \lambda y. x)((\lambda w. w)(\lambda w. w))$$

Again there are two redexes: the $\lambda x. \dots$ and the first $\lambda w. \dots$. This gives us two possible reduction paths:

$$\begin{array}{ccc} (\lambda x. \lambda y. x)((\lambda w. w)(\lambda w. w)) & \longrightarrow & (\lambda x. \lambda y. x)(\lambda w. w) \\ \downarrow & & \downarrow \\ \lambda y. ((\lambda w. w)(\lambda w. w)) & \longrightarrow & \lambda y. \lambda w. w \end{array}$$

The whole reduction graph

Here's a complete analysis of all possible reduction sequences starting from our example term.



Some things to notice

- ▶ Where we have a choice, the different paths always come back to the same place in the end
- ▶ In this case, every time there's a choice point, the paths come back together after one step.

The first of these is a general phenomenon: in the λ -calculus, choices of redex don't affect where you can get to.

The second is not generally true, as we will now see.

Evaluation order affects computation time

If we measure "time" as the number of β -steps taken to get from one λ -term to another, then the choice of which redex to reduce can have an impact on computation time.

Consider, for instance, the term

$$(\lambda x. xx)((\lambda y. y)(\lambda z. z))$$

If we first reduce the $\lambda y. \dots$ redex, we get

$$(\lambda x. xx)(\lambda z. z)$$

and now the only reduction path is

$$(\lambda x. xx)(\lambda z. z) \rightarrow_{\beta} (\lambda z. z)(\lambda z. z) \rightarrow_{\beta} \lambda z. z$$

Evaluation order affects computation time

Starting again at

$$(\lambda x.xx)((\lambda y.y)(\lambda z.z))$$

if we first reduce the $\lambda x \dots$ redex, we get

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

and one reduction path from here is

$$\begin{aligned} ((\lambda y.y)(\lambda z.z))((\lambda y.y)(\lambda z.z)) &\rightarrow_{\beta} (\lambda z.z)((\lambda y.y)(\lambda z.z)) \\ &\rightarrow_{\beta} (\lambda y.y)(\lambda z.z) \\ &\rightarrow_{\beta} \lambda z.z \end{aligned}$$

which takes one more step than the previous route.

Notice that we're reducing $(\lambda y.y)(\lambda z.z)$ twice in this computation.

What happened?

What happened there was that when we reduced the redex

$$(\lambda x.xx)(\text{something})$$

we duplicated the "something".

Since in this case the "something" was a term that required some evaluation, this duplication caused us to do more work.

When we instead reduced the "something" first, we did that work just once. By the time the term was duplicated, it needed no more evaluation; we duplicated the term, but we didn't duplicate the work.

An efficient reduction strategy?

Maybe we can learn a lesson from this.

Perhaps, when considering reducing a redex $(\lambda x.M)N$, we should always reduce N first if we can, so that we avoid duplicating work when we substitute N into M .

Would that give us the most efficient reduction path?

Sadly, no.

Doing unnecessary work

It is true that always evaluating the argument part (the N) first will avoid duplication of work. Unfortunately, it could also lead to us doing work that was not necessary at all.

For example, consider the term

$$(\lambda x. \lambda y. y)((\lambda z. zz)(\lambda w. w)).$$

As before we could reduce the argument, taking two β -steps to reach $\lambda w. w$, and then reduce the $\lambda x. \dots$. This would give $\lambda y. y$ in a total of three steps.

In this case we could have reached this in one step, if we'd simply reduced the $\lambda x. \dots$ redex first.

Key points about evaluation order

When faced with a choice of redexes to work on

- ▶ your choice will not “commit” you to anything: you’ll always be able to get to the same terms as you can from any other choice
- ▶ your choice may duplicate terms which require work to reduce, thus duplicating that work
- ▶ your choice may do work that is not necessary.

The theory of the λ -calculus

We’re now in a position to begin developing the theory of the λ -calculus properly.

Among other things, we’ll see that:

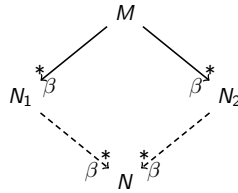
- ▶ the fact that you can always reconcile different choices is guaranteed by the *Church-Rosser theorem*
- ▶ thanks to this, β -reduction gives us a reasonable notion of *equivalence* of λ -terms
- ▶ there are various systematic methods of choosing redexes to reduce, called *reduction strategies*.

Church-Rosser property

The Church-Rosser property for β -reduction states that

Church-Rosser Property

if $M \rightarrow_{\beta}^* N_1$ and $M \rightarrow_{\beta}^* N_2$ then there exists some N such that $N_1 \rightarrow_{\beta}^* N$ and $N_2 \rightarrow_{\beta}^* N$.



If there's time, we might prove this fact later in the course.

Unique normal forms

One very helpful consequence of the Church-Rosser property is that a λ -term can have at most one normal form:

Theorem

If $M \rightarrow_{\beta}^* N_1$ and $M \rightarrow_{\beta}^* N_2$ and N_1 and N_2 are in normal form (that is, they contain no redexes) then $N_1 \equiv N_2$.

Proof By the Church-Rosser property, there is some N such that $N_1 \rightarrow_{\beta}^* N$ and $N_2 \rightarrow_{\beta}^* N$. But N_1 and N_2 are in normal form so $N_1 \equiv N \equiv N_2$ as required. \square

Equivalence of λ -terms

The uniqueness of normal forms means that we can speak of *the* normal form of a λ -term. Thus normal forms provide a decent-looking notion of "result" of a computation.

A natural thing to do is to consider terms "equal" if they give the same result, that is, if they have the same normal form.

Another natural thing to do is to consider terms "equal" if we can β -reduce one to the other.

Let's see how this can be done, and see how the two notions of equivalence of terms compare.

The equational theory $=_{\beta}$

Let's build a notion of equivalence on λ -terms based on β -reduction.

What we want is an equivalence relation (a reflexive, transitive, symmetric relation) on terms, which makes M equivalent to N if $M \rightarrow_{\beta} N$.

We can define this relation using another inductive definition.

$$\frac{M \rightarrow_{\beta}^* N}{M =_{\beta} N}$$
$$\frac{M =_{\beta} N}{N =_{\beta} M}$$
$$\frac{M =_{\beta} N \quad N =_{\beta} P}{M =_{\beta} P}$$

What does $=_{\beta}$ mean?

Suppose we can show, using these rules, that $M =_{\beta} N$. What does this mean?

By the final rule, it means that there is a sequence of $=_{\beta}$ equalities leading from M to N .

The first rule says that these equalities are generated by \rightarrow_{β}^* reductions.

The second rule says that we can "go backwards" with \rightarrow_{β}^* reductions too.

Theorem

$M =_{\beta} N$ iff there is a sequence M_1, \dots, M_k of terms such that

$$M \rightarrow_{\beta}^* M_1 \xrightarrow{\beta^*} M_2 \rightarrow_{\beta}^* M_3 \cdots \rightarrow_{\beta}^* M_k \xrightarrow{\beta^*} N.$$

What does $=_{\beta}$ mean

This is to say that terms are related by the equivalence relation $=_{\beta}$ if *one can be converted to the other* by applying β -steps forwards and backwards.

The *equational theory* $=_{\beta}$ is called the *theory of β -conversion*.

A slightly stronger statement of the Church-Rosser property, called the Church-Rosser *theorem*, lets us strengthen our understanding of $=_{\beta}$:

Theorem (Church-Rosser theorem)

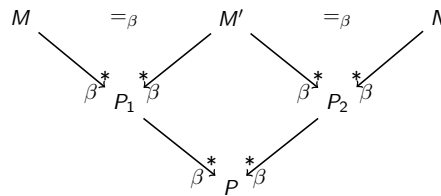
If $M =_{\beta} N$ then there is a term P such that $M \rightarrow_{\beta}^* P$ and $N \rightarrow_{\beta}^* P$.

Proof of the Church-Rosser theorem

Proof By induction on the number of proof rules used to establish $M =_{\beta} N$.

- ▶ If $M =_{\beta} N$ was proved from $M \rightarrow_{\beta}^* N$ using just one rule, then we can take $P \equiv N$ and then we have $M \rightarrow_{\beta}^* P$ and $N \rightarrow_{\beta}^* P$ as required.
- ▶ If $M =_{\beta} N$ because $N =_{\beta} M$ with a shorter proof, then by the inductive hypothesis we have P such that $N \rightarrow_{\beta}^* P$ and $M \rightarrow_{\beta}^* P$, which is what we needed.
- ▶ If $M =_{\beta} N$ because $M =_{\beta} M'$ and $M' =_{\beta} N$, both with shorter proofs, then the inductive hypothesis gives us P_1 such that $M \rightarrow_{\beta}^* P_1$ and $M' \rightarrow_{\beta}^* P_1$, and also gives us P_2 such that $M' \rightarrow_{\beta}^* P_2$ and $N \rightarrow_{\beta}^* P_2$. Then by the Church-Rosser property, there's a P such that $P_1 \rightarrow_{\beta}^* P$ and $P_2 \rightarrow_{\beta}^* P$. Hence $M \rightarrow_{\beta}^* P$ and $N \rightarrow_{\beta}^* P$ as required. \square

Illustration of the last case of the proof



The two top v-shapes are generated by the inductive hypothesis. The diamond in the middle is the Church-Rosser property.

$=_{\beta}$ and normal forms

We can now relate the theory $=_{\beta}$ to the idea that terms with the same normal form should be considered equal.

Suppose M can be reduced to a normal form N . That is, $M \rightarrow_{\beta}^* N$. Then by definition, $M =_{\beta} N$.

Conversely, suppose $M =_{\beta} N$ and N is in normal form. By the Church-Rosser theorem, there is a term P such that $M \rightarrow_{\beta}^* P$ and $N \rightarrow_{\beta}^* P$. But N is in normal form, so cannot reduce at all.

This means $N \equiv P$, so we know that $M \rightarrow_{\beta}^* N$. Thus:

If N is a normal form, then $M =_{\beta} N$ if and only if $M \rightarrow_{\beta}^* N$.

Consistency of $=_{\beta}$

We can now establish a fact which is very important to the theory of λ -calculus: the theory $=_{\beta}$ is *consistent*, meaning that it does not equate every pair of terms. Indeed, it does not equate any two distinct normal forms.

If $M =_{\beta} N$ and M and N are both normal forms, then by the Church-Rosser theorem we know $M \rightarrow_{\beta}^* N$, so $M \equiv N$ since M is a normal form.

Hence our notion of “final answer”, the normal form, is a good one: different final answers really are different!

Review

Let's review what we've got so far:

- ▶ a collection of “programs”, the λ -terms
- ▶ a notion of “computation step”, β -reduction
- ▶ a notion of “final answer”, the normal forms
- ▶ a notion of “equivalence of programs”, $=_{\beta}$, which (among other things) guarantees that equivalent programs produce the same final answer
- ▶ a non-deterministic approach to evaluating programs (choose any redex) which nevertheless does not cause us to make “mistakes” (thanks to the Church-Rosser property).

Reduction strategies

We know we can reduce any available redex in a term, and since reduction preserves $=_{\beta}$, thanks to the Church-Rosser theorem we will always be able to get to a normal form if one exists.

There is still the practical question of how to choose which redex to reduce.

We will look at three *reduction strategies*:

- ▶ normal-order reduction
- ▶ applicative-order reduction
- ▶ lazy reduction

and also consider what is done in practice in functional programming languages.

Normal-order reduction

Normal-order reduction, also known as *leftmost outermost* reduction and *standard reduction* is a valuable reduction strategy, because normal-order reduction always produces an answer, if there is one.

Theorem

If a term M has a normal form N , then reducing M with the normal-order strategy will eventually reach N .

As we will see, this is not the case for other reduction strategies.

However, normal-order reduction can involve a great deal of duplication of work.

Outermost reduction

In a λ -term, some redexes are inside other redexes. For instance, in

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

there are two redexes:

- ▶ the whole term
- ▶ $(\lambda y. yy)(\lambda z. z)$

Clearly the second redex is inside the first one, i.e. it forms a subterm of the first redex.

Outermost reduction strategies always reduce an outermost redex, i.e. a redex which is not inside any other redex.

Leftmost outermost reduction

There can be more than one outermost redex. For instance

$$((\lambda x. x)(\lambda y. y))((\lambda x. x)(\lambda y. y))$$

has exactly two redexes, both of the form

$$(\lambda x. x)(\lambda y. y)$$

and neither is inside the other.

Normal-order reduction is *leftmost outermost* reduction: find all the outermost redexes, pick the one furthest to the left, and reduce that.

Leftmost outermost reduction

Here's leftmost outermost (normal-order) reduction in a more algorithmic style:

- ▶ a term of the form x is a normal form. No need to reduce!
- ▶ given a term of the form $\lambda x.M$, look for a redex in M and reduce it.
- ▶ given a term of the form MN , consider the shape of M :
 - ▶ if M is $\lambda x.M'$, the whole term is $(\lambda x.M')N$. Reduce this redex.
 - ▶ otherwise, look for a redex in M (i.e. look on the left!) and reduce it if you find it. If there isn't one, look for a redex in N and reduce that.

Example

Let's apply normal-order reduction to the term

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

This is an application (remember application associates left), and the function part is not an abstraction, so we look for a redex in there.

The function part is

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

Again this is an application, but this time the function part is an abstraction, so we have found the leftmost outermost redex. Reduce it to get

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

If we were to keep going, our algorithm would locate the only redex in this term—the whole term itself—and reduce that to

$$\lambda x.xx.$$

Applicative order reduction

Applicative order reduction is the *leftmost innermost* reduction strategy.

That is to say, you look for all the innermost redexes, then pick the leftmost one, and reduce that.

Thus in the term

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

the applicative order strategy finds the redex

$$(\lambda x.xx)(\lambda x.xx)$$

and reduces that, because it's the innermost one.

Unfortunately, this term reduces to itself, so the applicative order strategy gets stuck in a loop.

Leftmost innermost reduction

Here's applicative order reduction as an algorithm:

- ▶ a term of the form x is in normal form; nothing to reduce.
- ▶ given a term of the form $\lambda x.M$, find a redex in M and reduce that.
- ▶ given a term of the form MN , look for a redex inside M and reduce that. If you don't find one, look inside N instead. If there isn't one there, perhaps M has the form $\lambda x.M'$, in which case the whole term is a redex; reduce that.

Because this is an *innermost* strategy, we only reduce a redex when we're sure there are no other redexes inside it.

This guarantees that in a function call MN , the term N is evaluated to normal form exactly once, before it is ever substituted into M .

That means we don't duplicate work, but as we've seen, it also means we sometimes do unnecessary work, with potentially fatal consequences.

Lazy reduction

Normal order reduction is good because it always finds a normal form if there is one.

However, it is not perfect because it can involve duplication of work.

Lazy reduction is an implementation of an approximation of normal-order reduction:

- ▶ find the redex to reduce using the leftmost-outermost strategy
- ▶ when you substitute a term in, keep a single copy of the term, and substitute *pointers* to that term instead
- ▶ if you later need to reduce one instance of the pointed-to terms, thanks to the pointer setup, *all* the instances are reduced at once.

This means that

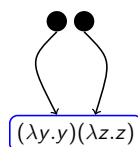
- ▶ we don't do unnecessary work (leftmost outermost)
- ▶ we don't duplicate work (pointer implementation).

Lazy reduction

Consider again the term

$$(\lambda x.xx)((\lambda y.y)(\lambda z.z))$$

When the lazy reduction implementation reduces its outermost redex, we get something like



Now when we come to reduce the "application" $\bullet \bullet$, our leftmost strategy tells us to reduce the function part.

This means we reduce the term inside the box, which simultaneously reduces the argument part.

Lazy reduction

After one step we have a situation like



To reduce this further, we have to unpack the pointer structure, copying the $\lambda z.z$ to reach

$$(\lambda z.z)(\lambda z.z)$$

which reduces to the normal form $\lambda z.z$ in one more step.

We avoided duplication of work, at the cost of some implementation overhead.

What functional languages really do

Real functional programming languages like Lisp, Haskell, ML etc. do not employ any of these reduction strategies; at least, not exactly as we've stated them.

All our reduction strategies work by reducing terms to *normal form*. In practice this is usually far too much work.

For instance, if a normal order or lazy reduction reaches a term of the form xM , where x is just a variable, the reduction strategy tells us to keep evaluating M .

There's no need to do this! If x is not bound to some function, there's just no point carrying on.

Similarly, if you reach a term of the form $\lambda x.M$, which is not applied to anything, you might as well stop there rather than reducing M .

Weak head normal form

In real functional languages, reduction always stops at *weak head normal form*. Weak head normal forms are λ -terms of the form

$$xM_1M_2 \dots M_n \quad \text{and} \quad \lambda x.M.$$

The reduction strategy of Lisp can be described as follows:

- ▶ given a term x or $\lambda x.M$, stop: it's already in weak head normal form
- ▶ given a term MN , reduce M until you reach weak head normal form.
 - ▶ If this is $xM_1 \dots M_n$, signal an error.
 - ▶ If it is $\lambda x.M'$, reduce N to weak head normal form N' , then reduce the redex $(\lambda x.M')N'$, and keep going.

This is essentially *applicative order reduction to weak head normal form*.

Lisp's evaluation strategy

Let's prove that Lisp does not evaluate under λ s, i.e. that it stops at weak head normal forms.

```
user> (lambda (x) (print 3))
#<Procedure #1f3528>
user> ((lambda (x) x) (lambda (y) (print 3)))
#<Procedure #1f303c>
user> (((lambda (x) x) (lambda (y) (print 3))) 5)
3 --- this is the 3 being printed
3 --- this is the 3 being returned
```

Note that in the first two cases, the 3 is not printed: the (print 3) is not evaluated because it is underneath a λ , and Lisp stops at weak head normal forms.

Head normal forms

Mid-way between normal forms and weak head normal forms are the *head normal forms*: terms of the form

$$\lambda x_1. \dots \lambda x_n. x M_1 \dots M_k.$$

In this case the variable x is called the *head variable* of the term.

Head normal forms play an important role in the theory of the λ -calculus, because it is possible to extend the equational theory $=_{\beta}$ with rules that say

any two terms which do not have a head normal form are equivalent

without losing the property of consistency.

Inconsistent theories

It is tempting to say that any two terms which do not produce a final answer are equivalent (since both are "useless" in some sense). But this leads to inconsistency.

Recall that $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ has no normal form. Therefore, neither do the terms

$$(\lambda x.\lambda y.\lambda z.y)\Omega M N \quad (\lambda x.\lambda y.\lambda z.z)\Omega M N$$

for any M and N . But if we decide that these terms are equivalent, then presumably reducing them gives equivalent things. But the first reduces to M and the second to N .

It follows that any two terms M and N are equivalent, so the theory is inconsistent.

Adding equations between all terms which lack a head normal form does not cause this kind of problem. We won't say more on this topic, though.