CM20167: Programming III

Russell Bradford

1 Introduction

1.1 Woffle

This course is regarded by some as “difficult”. The principal reasoning behind this seems not to be the nature of the content, but rather the fact that you, the student, is expected to do some of the work for yourself.

This is not a course where you can attend the lectures, leave the coursework to the last moment and hope to understand what is going on. You are now a 2nd year University student and should not need handholding through every step of the learning process.

> You, the student, need to be involved in the learning process.

This is just newspeak for “you need to get down and do some work”.

I expect you to go away and do things for yourself, in particular the bulk of learning about Lisp, the language we shall be highlighting in the course. This is all part of your “learning how to learn”. After all, in a business environment, you will be given a problem to solve and then largely left to your own devices. Part of this course is for you to learn the skills you will need to do that.

1.2 What is the course about?

The gap between theory and practice. How does the stuff on computability and decidability tie in with real programming languages?

You have seen

- Assemblers (unstructured)
- C-like languages (procedural)
- Java or C++-like languages (object oriented)
- Perhaps Prolog (declarative languages)
- Event driven, such as used in User Interfaces

Next on the list is

- Lisp-like languages (functional)

There are four main aspects to this course:

- Practice: Lisp programming,

- Theory: Lambda Calculus (and more),
- The link between the above,
- Introduction to other functional languages.

No one style of programming is suitable for all jobs. Beware the person who says that "Java is the only language I need", as this is a sure sign of a bad computer scientist. Compare this with a plumber who says "A hammer is the only tool I need".

So:

> Use the right tool for the job.

One of the main points to this course is to introduce the functional style. This style is not suitable for every programming task, but it is another string to your bow that you can employ when appropriate. Even if you can't use a functional language for a project, many of the concepts transfer to other styles of programming and will improve your programming in that style.

This course is 25% coursework and 75% exam. The coursework will be a programming exercise in Lisp. **Don't leave the coursework until the last moment**. There is a lot of "culture shock" involved in learning Lisp (or any functional language) for people who have only encountered the OO or procedural styles. They spend more time trying to program Lisp in a procedural style than learning the new style and this just doesn't work.

> You will need to learn a new style of programming.

Trying to force the old style is counterproductive. The more open you are to new concepts, the easier you will find this course.

Some small easy exercises have been prepared for you to try: see `http://www.bath.ac.uk/~masrjb/CourseNotes/cm20167.html`

> Do these exercises. It will pay you back multifold in the long run, and not just in this course.

This web page also contains other items of interest regarding Lisp and Lambda Calculus. Do read some of the articles.

1.3 Books

For lambda calculus.

- H P Barendregt "The Lambda Calculus" Exhaustive, high level.
- J R Hindley and J P Seldin "Introduction to Combinators and λ-Calculus"
- The Web

For Lisp/Scheme.

- Abelson and Sussman "Structure and Interpretation of Computer Programs"
- a million books on Lisp

- `http://www.bath.ac.uk/~masrjb/Sources/eunotes.html`
- `http://www.bath.ac.uk/~masjap/TYL/`

> “ Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.”

Eric Raymond, ”How to Become a Hacker”.

2 Lisp and Scheme

2.1 A Brief History

Early history.

- Developed in 1956-1958 by John McCarthy. First “official” release 1959. Only Fortran and Algol 58 older.
- Symbolic processing oriented, not numerical.
- For the IBM 704. This had a 36 bit word and 15 bit registers in a 15 bit address space.
- List processor. 15 used to point to object, 15 bits used to point to rest of list. Used *address register* and *decrement register* respectively.

  > Note this the first hint of the functional style. A list is (a) empty, or (b) an object and the rest of the list. A recursive approach (or inductive, for the Mathematicians) is central.

- Wanted higher order functions, adopted Church’s lambda calculus to notate functions.
- Natural use of recursion. $n! = n \times (n-1)!$
- Simple syntax: program and data look the same “parenthesised prefix”
- So programs can read and manipulate programs
- The eval function takes a piece of data (i.e., a list data structure) and executes it as a program.
- Thus can write a Lisp interpreter in Lisp.
- Garbage collection, so no worries about memory allocation in lists, etc.

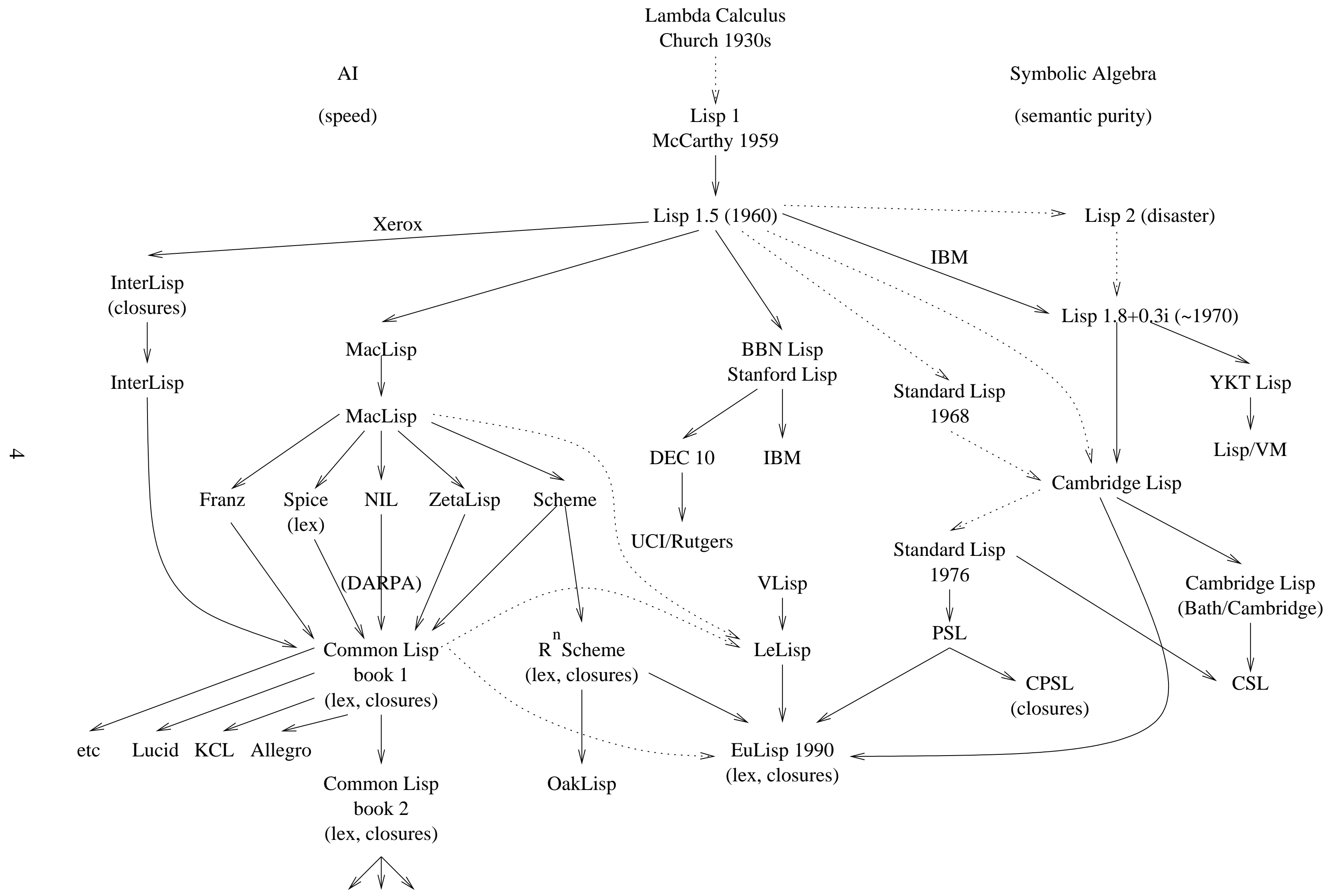
Lisp has flexibility. Many new ideas tested in Lisp before becoming integrated in other languages. E.g., object oriented techniques. All modern languages have something borrowed from Lisp.

Lisp diaspora. Common Lisp and Scheme. EuLisp. (ISLisp, Dylan, Java, Perl).

Lisp vs. Scheme.

2.2 Basic Introduction

BUCS: `drscheme`, `euscheme`.


Lambda Calculus
Church 1930s
AI
(speed)
Symbolic Algebra
(semantic purity)
Lisp 1
McCarthy 1959
Lisp 1.5 (1960)
Lisp 2 (disaster)
Xerox
IBM
InterLisp
(closures)
InterLisp
MacLisp
MacLisp
BBN Lisp
Stanford Lisp
Standard Lisp
1968
Lisp 1.8+0.3i (~1970)
YKT Lisp
Lisp/VM
DEC 10
IBM
UCI/Rutgers
Cambridge Lisp
Franz
Spice
(lex)
NIL
ZetaLisp
Scheme
(DARPA)
Standard Lisp
1976
Cambridge Lisp
(Bath/Cambridge)
VLisp
LeLisp
PSL
CPSL
(closures)
CSL
Common Lisp
book 1
(lex, closures)
R n Scheme
(lex, closures)
etc
Lucid
KCL
Allegro
EuLisp 1990
(lex, closures)
Common Lisp
book 2
(lex, closures)
OakLisp

2.3 The Functional Style

- Functions as objects (first class objects)
- Recursion rather than iteration (e.g., traversing a tree)
- Avoid assignment (`setq`), use binding (`let`)
- Datastructures "in the whole" (`map`)
- Avoid side effects (referential transparency)

3 Lambda Calculus

The theoretical basis for Lisp. First explored by Church in the 1930s as a way to investigate higher-order logic.

Motto: "Everything is a function"

We shall be describing a particular form of lambda calculus known as *pure* lambda calculus. Many variants exist, for example typed lambda calculus (see later), but we start with the simplest.

All formalisms like lambda calculus, and, indeed, set theory, start with a few basic symbols, rules on how to combine them into valid formulas, and rules on how formulas transform into each other. We shall not be too formal in showing proofs of things, though we ought to in order to treat the subject seriously.

3.1 Syntax

The syntax of lambda calculus is composed of λ-*terms*.

We have an inexhaustible supply of *variables*, e.g., x, y, z, and the special symbol λ.

Actually, using variables is a bad way to proceed and will cause us problems later. But better ways are very difficult to typeset and manipulate.

A λ-term is defined as follows.

- a variable is a λ-term
- if M and N are λ-terms, so is $(M)(N)$. This is called an *application*
- if M is a λ-term and v is a variable, then $\lambda v.(M)$ is a λ-term. This is called an *abstraction*. Here, M is the *body*, while v is the *formal argument*
- nothing else is a λ-term.

Examples.

$$\begin{array}{cccc} x & \lambda x.(x) & \lambda x.(z) & (\lambda x.(y))(z) \\ (x)(x) & (\lambda x.(x))(\lambda x.(x)) & (\lambda y.(y))(\lambda x.(x)) & (\lambda z.(z)(z))(\lambda z.(z)(z)) \end{array}$$

In that last example, the z in the right half is not the "same" as the z in the left half.

Note that we can apply something to itself: $(x)(x)$, this *does* make sense if x is, say, the identity function.

Also look at $\lambda x.(\lambda y.(z))$. This is a function (of x) that when you apply it to an argument it returns a function, namely $\lambda y.z$. Functions are valid values: in fact functions are the *only* values!

As is usual with such things, we drop parentheses when we can. The convention is that

- application binds more tightly than abstraction, e.g., $\lambda x.MN$ is $\lambda x.((M)(N))$, not $(\lambda x.(M))(N)$,
- application associates left-to-right, e.g., xyz is $((x)(y))(z)$.

Another notational convenience is collecting together λs: $\lambda xyz.M$ for $\lambda x.\lambda y.\lambda z.M$, which is

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

Very important note:$\lambda xyz.M$ is *not* a function of three variables, it is just a simple way of writing the nested lambda expression.

3.2 Free and Bound Variables

It is plain that the two λ-terms $\lambda x.x$ and $\lambda y.y$ are both ways of writing the "same" function (the identity, in this case). The fact we used an x in the first and a y in the second is somehow irrelevant. We had to use *some* variable, though.

On the other hand, $\lambda x.xy$ and $\lambda y.yy$ are definitely different, so we can't just swap names around at random. We must distinguish carefully between names that appear stuck to lambdas, and names that don't.

Compare with code

```
int f(int x)
{
  ... x + y ...
}
```

and

```
int f(int y)
{
  ... y + y ...
}
```

A variable in the body of an abstraction that is also the formal argument of that abstraction is called a *bound* variable (in that abstraction).

Non-bound variables are called *free* variables.

x	x is free
$\lambda x.x$	x is bound
$\lambda x.xy$	x is bound, y is free
$(\lambda x.x)x$	the x within the body is bound, while the one on the outside is free

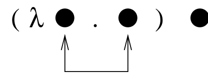


Figure 1: No names

This is a point of easy confusion: the x inside is "really" a different variable than the x outside, we have just been perverse. We could equally have written

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

to mean the same thing, and now the difference is clear. Again in

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

the inner xs are bound while the outer is free. Note that, as with computer languages, the inner binding of x shadows the outer binding. A more readable version could be

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

In a C-like language we can write

```
...                       ...
{ int x;                  { int y;
  ...                       ...
  { int x;                  { int z;
    ...                       ...
    use inner x               use z
    ...                       ...
  }                         }
  ...                       ...
  outer x again             use y
  ...                       ...
}                         }
```

This is bad coding style, but valid as a program.

It would be better not to have to use named variables for bound positions for this reason, however it makes writing down and understanding λ-terms that much harder.

For example, one alternative notation is to use λλ2 1 3 to mean $\lambda x.(\lambda y.xyz)$. And $\lambda y.(\lambda x.yxz)$. And $\lambda w.(\lambda x.wxz)$. And so on.

The integer indicates which λ the position should be bound to: 1 for the closest, 2 for the next, and so on. A number bigger than the depth of nesting of λs denotes a free variable. Different free variables get different values.

We do not want to distinguish between λ-terms that only differ in consistent renamings of bound variables. This is called α *renaming*. Thus $\lambda x.x$ "is the same as" $\lambda y.y$, and $(\lambda x.xx)z$ "is the same as" $(\lambda y.yy)z$, but $(\lambda x.xx)z$ "is not the same as" $(\lambda x.xx)w$, and $\lambda x.xy$ "is not the same as" $\lambda y.yy$. The last because the term on the lhs has a free variable (y), but the term on the rhs does not. This is called *name capture*, and is a thing to be avoided.

> Alpha renaming is *only* for bound variables.

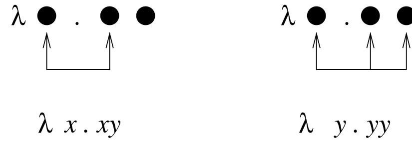


Figure 2: Name capture

Now, if expression A "is the same as" B then B "is the same as" A; if A "is the same as" B and B "is the same as" C then A "is the same as" C. *These statements are not entirely obvious and we really ought to prove them from our definitions by considering carefully what we mean by "is the same as".*

So when expression A "is the same as" B we can, up to a point, replace A whereever we see it by B (we are glossing over a *lot* here), and we are therefore justified to call A and B (in some sense) "equal".

So, the notation we use for "is the same as" is $A =_\alpha B$, or more commonly simply $A = B$.

Notice what we are doing here: we are *defining* a relationship between λ-terms that *behaves* like an equality.. If we risk confusion by using the familiar equals sign (=) we can use $=_\alpha$ to make things clear. There is a separate notion of *structural identity*, written $\equiv$, if we want to say two terms are completely identical. So $\lambda x.x \not\equiv \lambda y.y$, but $\lambda x.x = \lambda y.y$, or $\lambda x.x =_\alpha \lambda y.y$ if we are being fussy.

Reiterating an earlier point: we ought to prove at this point that $=_\alpha$ is worthy of being called an equality, namely that it has the properties we might expect from an equality. Some of these properties are

- Reflexive: a term is $=_\alpha$ to itself, $M =_\alpha M$ for all M. This is clear.
- Symmetric: if $M =_\alpha N$ then $N =_\alpha M$. This is also clear.
- Transitive: if $M =_\alpha N$ and $N =_\alpha P$ then $M =_\alpha P$. This says a succesion of renamings is a renaming, so this is OK.

There are other properties (namely substitionality, see later), but for now we can agree that calling $=_\alpha$ and equality makes sense.

Now, a term is *closed* if there are no free variables. We write $FV(M)$ for the collection of free variables of M.

Exercise: write down a formal definition of FV. You will need to refer back to the definition of a λ-term and go through each case of bound and unbound variables.

Answer:

$FV(x) = \{x\}$	for a variable x
$FV(MN) = FV(M) \cup FV(N)$	for an application
$FV(\lambda x.M) = FV(M) \setminus \{x\}$	for an abstraction

Thus $FV(\lambda x.xy) = FV(xy)\setminus\{x\} = (FV(x) \cup FV(y))\setminus\{x\} = (\{x\}\cup\{y\})\setminus\{x\} = \{x, y\}\setminus\{x\} = \{y\}$. Similarly, $FV((\lambda x.x)x) = FV(\lambda x.x) \cup FV(x) = (FV(x) \setminus \{x\}) \cup \{x\} = (\{x\} \setminus \{x\}) \cup \{x\} = \emptyset \cup \{x\} = \{x\}$.

Exercise: write down a formal definition of α equality of two λ-terms M and N.

Answer:

- M is a variable, x, say. Return true if $N \equiv M$ else return false.
- M is an application AB. Return true if N is an application CD with $A =_\alpha C$ and $B =_\alpha D$, else return false.

- M is an abstraction $\lambda x.A$. Return false if N is not an abstraction. So N is $\lambda y.B$, say.
  - if $x \equiv y$ return $A =_\alpha B$.
  - let z be some variable not appearing in AB. Replace all free xs in A by z giving A' and all free ys in B by z giving B'. Return $A' =_\alpha B'$

Note: this proof uses *structural recursion*.

Exercise: Prove that α equality has the behaviour you might expect from an equality, namely

- $A =_\alpha A$ for all λ-terms A
- if $A =_\alpha B$ then $B =_\alpha A$ for all λ-terms A and B
- if $A =_\alpha B$ and $B =_\alpha C$ then $A =_\alpha C$ for all λ-terms A, B and C

3.3 Substitution

Substitution captures the idea of replacing a variable with a value.

Given λ-terms M and N and a variable v, we can substitute N for each *free* occurrence of v in M, *provided we don't accidentally bind any of the free variables in* N. If there would be some name clashes, we can do some α renaming first.

We write this as $[N/v]M$. Note in the special case that v does not appear free in M we have $[N/v]M = M$ (in fact, $[N/v]M \equiv M$).

Substitution *only* happens for free variables.

Examples.

$[\lambda y.y/x]x = \lambda y.y$	
$[\lambda z.zz/x]\lambda x.xy = \lambda x.xy$	x is not free
$[y/x]\lambda z.(\lambda x.xy)x = \lambda z.(\lambda x.xy)z$	the outer x is free, but the inner is bound
$[y/x]\lambda y.xy \neq \lambda y.yy$	that unbound x became a bound y
$[\lambda z.x/y]\lambda x.xy \neq \lambda x.x(\lambda z.x)$	the x in the $\lambda z.x$ accidentally got bound
$[\lambda x.x/y]\lambda x.xy = \lambda x.x(\lambda x.x)$	the inner x is *rebound* and so is OK $[y/x]\lambda y.xy =_\alpha [y/x]\lambda z.xz = \lambda z.yz$

In the last, we renamed the variable in the abstraction from y to z to avoid a clash. The substitutions can be arbitrary terms for the free variable, e.g.,

$$[\lambda x.xz/x]\lambda y.xyx = \lambda y.(\lambda x.xz)y(\lambda x.xz).$$

Notationally, $[\,]$ has very low precedence, so that $[x/y]zw$ means $[x/y](zw)$. Also, the form $[N/x][M/y]P$ means $[N/x]([M/y]P)$

Just to get a flavour of these things, we show the formal definition of substitution. In general, we shan't be too formal.

The definition is in several cases:

- variable: $[N/x]x = N$
- variable: $[N/x]a = a$, for variables $a \neq x$

- application: $[N/x](PQ) = ([N/x]P)([N/x]Q)$
- abstraction: $[N/x](\lambda x.P) = \lambda x.P$
- abstraction: $[N/x](\lambda y.P) = \lambda y.[N/x]P$ if $y \neq x$ and $y \notin FV(N)$ or $x \notin FV(P)$ (y is not free in N so can't be accidentally bound, or x doesn't actually appear free in P)
- abstraction: $[N/x](\lambda y.P) = \lambda z.[N/x][z/y]P$ if $y \neq x$ and $y \in FV(N)$ and $x \in FV(P)$ (x really does appear this time and would bind a free y in N)

In the last, z is some variable not in $FV(NP)$. This is just α renaming in P to avoid capture of the free y in N. For example, $[y/x]\lambda y.x$. Note we only need to rename if (a) y appears free in N, and (b) x is free in P, i.e., some substitution actually happens. In $[y/x]\lambda y.y$ there is no free x.

Note this is an inductive definition: given a term M one of the above must apply (recall the definition of a λ-term), and then we get the substitution on M defined using the sub-terms of M.

Everything in lambda calculus is defined formally, but we shall focus on the informal interpretations. For example, in the last above, N has a variable y that clashes, so we α rename it to z first.

Exercise: From the above definition, prove that $[N/x]P \equiv P$ if $x \notin FV(P)$.

Answer: Do the cases!

1. variable
2. application
3. abstraction

3.4 Reduction

Reduction is the primary way of manipulating λ-terms. It is what makes the lambda calculus a model of computation: a reduction is like a computation.

A term of the form $(\lambda u.E)F$ is called a *redex* (*reducible expression*). Conceptually, this is a function being applied to an argument. The *reduction* $[F/u]E$ (together with any α renamings to make this safe) is sometimes called the *contractum*. Reduction is equivalent to execution of a function to produce a result.

We say: *M β-reduces to N in one step* if N results from M after a reduction of some subterm of M, and we write $M \succ_{1\beta} N$. Thus, for example, $(\lambda u.E)F \succ_{1\beta} [F/u]E$.

Examples.

$(\lambda x.y)z \succ_{1\beta} y$
$x((\lambda x.xy)z)y \succ_{1\beta} x(zy)y$
$(\lambda x.xy)(\lambda z.zz) \succ_{1\beta} (\lambda z.zz)y \succ_{1\beta} yy$
$(\lambda x.xx)(\lambda x.xx) \succ_{1\beta} (\lambda x.xx)(\lambda x.xx) \succ_{1\beta} \dots$
$x(\lambda y.y)z$ has no reduction!

We write $M \succ_{\beta} N$, or even $M \succ N$, if there is a finite sequence of zero or more single reductions

$$M \succ_{1\beta} M' \succ_{1\beta} M'' \succ_{1\beta} \dots \succ_{1\beta} N,$$

and say M β-reduces to N.

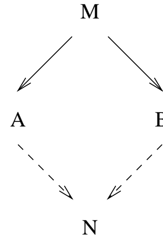


Figure 3: Church Rosser

We also allow any number of α renamings:

$$M \succ_{1\beta} M' \succ_{1\beta} M'' =_{\alpha} M''' \succ_{1\beta} \ldots \succ_{1\beta} N,$$

There can be zero reductions, so that $M \succ_{\beta} M$.

Note: in some sense "$2+2 \succ 4$". So $2+2$ and 4, while the same mathematically, differ by a reduction step in the lambda sense.

3.5 Normal Forms

M is in *normal form* if it contains no redexes. Thus x, xy, $x(\lambda y.y)z$ are in normal form. On the other hand, $(\lambda y.y)z$ and $z((\lambda y.y)z)\lambda x.x$ are not.

Clearly, a term in normal form cannot be β-reduced. A term not in normal form *can* be β-reduced. A normal form represents the end of a computation: the answer, if you like.

It would seem that to produce a normal form from a term we should just keep reducing it until we can go no further. Unfortunately, this doesn't work, as we have already seen $(\lambda x.xx)(\lambda x.xx)$ reduces forever. Of course, some computations never terminate either: think of infinite loops.

A further complication is that there is more than one way to reduce a term if it contains more than one redex.

$$(\lambda x.(\lambda y.xy)x)z \succ_{\beta} (\lambda y.zy)z$$

applying the outer λ, while

$$(\lambda x.(\lambda y.xy)x)z \succ_{\beta} (\lambda x.xx)z$$

applying the inner λ.

In this case, both further reduce to zz. In fact, this is generally true: there may be multiple ways to reduce, but you can always end up at the same place.

3.6 Church-Rosser

One of the cornerstones of the lambda calculus is this theorem of Church and Rosser:

> If M, A and B are any λ-terms with $M \succ_{\beta} A$ and $M \succ_{\beta} B$, then there is a λ-term N with $A \succ_{\beta} N$ and $B \succ_{\beta} N$.

This is sometimes called *confluence*, or the *diamond property* of reductions.

A corollary is:

> If M, A and B are any λ-terms with $M \succ_\beta A$ and $M \succ_\beta B$, and A and B are both in normal form, then $A = B$ (up to α renaming, so this is really $A =_\alpha B$).

Notice this does not guarantee the existence of a normal form: it just says if a normal form exists, it is unique. Nor does it guarantee a sequence of reductions will ultimately result in a normal form, even if that normal form exists.

For some terms we get the normal form regardless of the sequence of reductions, e.g., $(\lambda x.(\lambda y.xy)x)z$, above.

Some terms do not have a normal form, e.g., $(\lambda x.xx)(\lambda x.xx)$. This particular term is a favourite, and is often called Ω. Also, little $\omega = \lambda x.xx$, so that $\Omega = \omega\omega$.

Some terms have a normal form, but it depends on the order of reductions whether we reach it or not. Of course, the only way to fail to reach the normal form in this case is to have an infinite sequence of reductions. For example, $(\lambda x.y)\Omega = (\lambda x.y)((\lambda z.zz)(\lambda z.zz))$.

If we β-reduce the Ω, this term reduces to itself. If we reduce the first λ we get y, the normal form.

Church-Rosser does guarantee if the normal form exists, we can always get to it from wherever we are. There are no wrong paths, just some very long ones!

3.7 Applicative and Normal order

So what can we do about finding normal forms? Is there some mechanical way of always finding a normal form when it exists? There are two (important) ways of doing reductions, called *applicative* and *normal* orders.

Consider the evaluation of a function:

```
(defun double (x) (+ x x))

(double (+ 2 3))
```

There are conceivably two ways to proceed:

- textually substitute the actual argument for the formal argument on the body, then reduce that
- reduce the actual argument, then substitute and reduce

Of course, real implementations of Lisp do the second. As do most, but not all, other languages.

> Maple has a curious evaluation strategy. Algol 60 had call by name as well as call by value.

For the first way:

```
(double (+ 2 3)) ->
(+ (+ 2 3) (+ 2 3)) ->
(+ 5 5) ->
10
```

And the second

```
(double (+ 2 3)) ->
(double 5) ->
(+ 5 5) ->
10
```

We get the same answer. Note that we're not guaranteed this, as with Church-Rosser, since Lisp is not pure lambda calculus.

```
(setq n 0)
(defun inc ()
  (setq n (+ n 1))
  n)
```

What is `(double (inc))`?

With the first, called *normal order*, or *leftmost outermost*, or *call by name*, where we reduce the outermost lambda first (i.e., the `double`), we might get

```
(double (inc)) ->
(+ (inc) (inc)) ->
(+ 1 2) ->
3
```

With the second, called *applicative order*, or *leftmost innermost*, or *call by value*, where we reduce the innermost lambda first (i.e., the arguments), we get

```
(double (inc)) ->
(double 1) ->
(+ 1 1) ->
2
```

This divergence from Church-Rosser is very important, and we will return to it later when we talk about functional programming.

What is the value of the following?

```
(setq n 0)
(list (inc) (inc))
```

Back to lambda calculus. We know that it theoretically doesn't matter whether we use applicative or normal order reduction as a normal form is unique. But there is one big practical difference:

normal order reduction will converge to the normal form, if it exists.

In general, it is undecidable whether a term has a normal form, but normal order reduction gives us a *semi-decision* procedure. That is, a method that will find the normal form if it exists, but will never terminate it it doesn't.

On the other hand, applicative order reduction has no such guarantee of getting to the normal form.

Look at $(\lambda x.y)\Omega = (\lambda x.y)((\lambda z.zz)(\lambda z.zz))$.

Normal order reduces the first lambda:

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

Applicative order reduces the inner lambda (in Ω):

$$(\lambda x.y)\Omega \succ (\lambda x.y)\Omega \succ \dots$$

So the moral of the tale is: use normal order reduction in lambda calculus!

This effect is reflected in Lisp, too. Consider

```
(defun foo (x y)
  (if (zerop x) 0 y))

(defun bar () (bar))

(foo 0 (bar))
```

Under applicative order evaluation, this program never terminates. Of course, (most) programming languages use applicative order as it is more efficient: it evaluates the arguments just once, rather than each time a formal argument appears in the function body. Other languages, such as Haskell, approximate normal order reduction using *lazy evaluation*. See later.

Another example.

```
(defun try (a b)
  (if (= a 0) 1 b))

(try 0 (/ 1 0))
```

This generates an error in Lisp since the `(/ 1 0)` is evaluated before the `if`. With normal order reduction, the `if` is evaluated first, and the division by zero never happens.

In other words, in Lisp, and most other languages, there *are* parts that use normal order reduction: namely with `if`!

```
if (x == 0 || 1/x == 2.0) ...
```

3.7.1 A Test for Normal Forms?

Here is the first step in proving that the existence of normal forms is undecidable.

Theorem There is no λ-term N that detects whether a λ-term has a normal form. That is such that for any λ-term M,

$$\begin{array}{ll} NM \succ T & \text{if } M \text{ has a normal form} \\ NM \succ F & \text{otherwise} \end{array}$$

Here $T = \lambda xy.x$, $F = \lambda xy.y$.

Proof Let $\omega = \lambda x.xx$, $\Omega = \omega\omega$ and $I = \lambda x.x$. Note that Ω has no NF, while I is already in NF.

Let N be a λ-term that detects NFs, and set $Z = \lambda z.N(\omega z)\Omega I$.

Now $\omega Z \succ ZZ \succ N(\omega Z)\Omega I$, so ωZ has a NF if and only if $N(\omega Z)\Omega I$ does (by Church-Rosser).

1. Suppose ωZ has a NF, so $N(\omega Z) \succ T$. But then

   $$N(\omega Z)\Omega I \succ T\Omega I \succ \Omega$$

   which has no NF. Contradiction.

2. Suppose ωZ does not have a NF, so $N(\omega Z) \succ F$. But then

   $$N(\omega Z)\Omega I \succ F\Omega I \succ I$$

   which is a NF. Contradiction, again.

So such a λ-term N cannot exist.

Now if we assume something is computable if and only if there is a λ-term with a normal form that represents it, we can see that the existence of normal forms is not computable, i.e., undecidable.

3.7.2 Non-termination

Expressions that don't have normal forms can fail to terminate in several ways:

- Simple Loops:
  The λ-term Ω β-reduces to itself in a single step. A related example is

  $$(\lambda xy.yxy)(\lambda xy.yxy)(\lambda xy.yxy)$$

  which reduces to itself in two steps. And

  $$(\lambda xyz.zxyz)(\lambda xyz.zxyz)(\lambda xyz.zxyz)(\lambda xyz.zxyz)$$

  reduces to itself in three steps.
  We can have arbitrary long loops: let $L = \lambda x_1x_2\ldots x_n.x_nx_1x_2\ldots x_n$. Then

  $$L^{n+1} = \underbrace{LL\ldots L}_{n+1 \text{ times}}$$

  reduces to itself in n steps.

- A sequence of reductions, then a loop. A simple example could be $(\lambda x.x)(\lambda x.x)\Omega \succ_\beta (\lambda x.x)\Omega \succ_\beta \Omega \succ_\beta \Omega \ldots$. Clearly, this can be extended to an arbitrary number of steps before the loop.

- Non looping. This must necessarily be an expression that grows without limit. For example $(\lambda x.xxx)(\lambda x.xxx) \succ_\beta (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \succ_\beta (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \succ_\beta \ldots$. Or $(\lambda x.xxy)(\lambda x.xxy) \succ_\beta (\lambda x.xxy)(\lambda x.xxy)y \succ_\beta (\lambda x.xxy)(\lambda x.xxy)yy \succ_\beta \ldots$.

Note: there are λ-terms that grow arbitrarily before hitting a NF (see the exponential Church numeral, later) so we can't judge the existence of a NF on size alone.

A real answer to an exam question: "first try applicative; if that fails try normal". How many ways is that wrong?

3.8 Another Equality

We have seen various kinds of ways to transform one λ-term into another

- α renaming: changing bound variables
- β-reduction: application of λs
- substitution: $[M/x]N$

Sometimes we say things are equal, and use $=$. Other times we say things reduce, and use $\succ_\beta$. This is an important distinction, sometimes glossed over in other branches of mathematics. For example, we say $2 + 3 = 5$ in arithmetic, but would say $2 + 3$ *reduces* to 5 in the lambda world.

Why is this distinction made? Because reduction corresponds to *computation*. It is quite natural to replace $2 + 3$ by 5, but quite unusual to replace 5 by $2 + 3$. So this equality is not really symmetric.

In the physical world, it appears that reversible and non-reversible computations really are quite different, as the former necessarily consumes energy.

We shall now define another relationship between terms (adding to $\equiv$ and $=_\alpha$).

We define $\sim_\beta$ by $M \sim_\beta N$ if there is a sequence $M = M_0, M_1, M_2, \ldots, M_n = N$ with

$$M_i \succ_\beta M_{i+1} \text{ or } M_{i+1} \succ_\beta M_i \text{ or } M_{i+1} \text{ is an } \alpha \text{ renaming of } M_i$$

Now we claim that $\sim_\beta$ is another "equality like" relation between λ-terms. For this to be a reasonable definition of an equality, we have to prove it has the expected properties of an equality:

- $M \sim_\beta M$
- if $M \sim_\beta N$ then $N \sim_\beta M$
- $M \sim_\beta N$ and $N \sim_\beta P$ implies $M \sim_\beta P$
- if $M \sim_\beta N$ then substituting N for M in any expression leaves its value unchanged, i.e., the new expression is $\sim_\beta$ to the old (with the usual caveats about α renaming to avoid clashes during the substitution). This is called *substitionality*. In an abuse of notation we might write "if $M \sim_\beta N$ then $[N/M]P \sim_\beta P$".

In this case everything works well, for there is a corollary of Church-Rosser that says

> if $M \sim_\beta N$ then there is a T with $M \succ_\beta T$ and $N \succ_\beta T$.

- The proof of $M \sim_\beta M$ is easy: since $M \succ_\beta M$ with zero reductions.
- Similarly, if $M \sim_\beta N$ there is a sequence of reductions/renamings taking M to N. Just reversing the sequence takes us from N to M, showing $N \sim_\beta M$.

- If $M \sim_\beta N$ and $N \sim_\beta P$, there is a sequence from M to N and a sequence from N to p. Joining them together gives us a sequence from M to P, i.e., $M \sim_\beta P$.
- Here is an outline of a proof of substitionality: if $M \sim_\beta N$, then $M \succ_\beta T$ and $N \succ_\beta T$ for some T. Suppose M appears in some expression $\ldots M \ldots$. Now $\ldots M \ldots \succ_\beta \ldots T \ldots \prec_\beta \ldots N \ldots$, which is to say $\ldots M \ldots \sim_\beta \ldots N \ldots$. A real proof would have to worry about α substitutions to avoid name capture.

Thus it is reasonable to think of $\sim_\beta$ as an equality of λ-terms. Thus, the usual notation for this is actually $=_\beta$, or (again) simply $=$ if we are being lazy.

So here is the definition again, written as you would normally see it:

Define $=_\beta$ by $M =_\beta N$ if there is a sequence $M = M_0, M_1, M_2, \ldots, M_n = N$ with

$$M_i \succ_\beta M_{i+1} \text{ or } M_{i+1} \succ_\beta M_i \text{ or } M_{i+1} \text{ is an } \alpha \text{ renaming of } M_i$$

A lot of people get confused about what is happening here, so we need to think carefully about what we are doing: we have just *defined* what the symbol $=_\beta$ means. Just because it has some typographic similarity to the $=$ symbol some people expect that $=_\beta$ automatically has certain properties, for example if $M =_\beta M$ for every expression M. This may or may not be true: we have to *prove* that that equation is true, as we did above.

Saying $M =_\beta N$ is very different from saying $M \equiv N$, or even $M =_\alpha N$. These are *all* equalities, but they claim different collections of λ-terms are equal. It all depends on what you want to do at the time. Whether you want to talk about equality of structure ($\equiv$), or equality up to inessential names ($=_\alpha$), or computational equality ($=_\beta$). All are valid, all are useful in different circumstances.

We have $\equiv \Rightarrow =_\alpha \Rightarrow =_\beta$.

Examples:

$$(\lambda x.(\lambda y.xy))z \succ (\lambda y.zy)z \succ \lambda z.zz,$$

so $(\lambda x.(\lambda y.xy))z =_\beta \lambda z.zz$,.

$$(\lambda x.y)\Omega =_\beta y$$

$$(\lambda x.y)\Omega =_\beta (\lambda z.(\lambda z.y)z)\Omega$$

Exercise: Give examples of M and N such that

1. $M =_\beta N$ but $M \neq_\alpha N$ and $M \not\equiv N$.
2. $M =_\beta N$ and $M =_\alpha N$ but $M \not\equiv N$.

Answer: Nearly anything will do.

1. $(\lambda x.x)x =_\beta x$ but $(\lambda x.x)x \neq_\alpha x$ and $(\lambda x.x)x \not\equiv x$.
2. $\lambda x.x =_\alpha \lambda y.y$ but $\lambda x.x \not\equiv \lambda y.y$.

3.8.1 No Decision Procedure

Now, while it is easy to determine if two λ-terms are $\equiv$ or are $=_\alpha$, it is possible to show that there is no algorithm to determine if two λ-terms are $=_\beta$. This follows from the halting problem for Turing machines. There is, however, a *semi-decision procedure*.

A *semi-decision procedure* is something that, if it terminates, gives you the answer, but is not guaranteed to terminate. Recall that if something is to be called an algorithm, or a decision procedure, it *must* terminate on all inputs. Something that is not guaranteed to terminate is *not* an algorithm.

So the semi-decision procedure for β-equality is to simply reduce both to normal form (when they exist), and see if they are the same (namely, $=_\alpha$). Thus, sometimes we can show things to be equal, but we are not guaranteed to be able to do so.

In fact, we already know that determining if a λ-term even has a normal form is also undecidable.

We might be able to determine when some other terms without normal forms are $=_\beta$, for example we can easily see that $(\lambda x.\Omega)y =_\beta \Omega$, but in general there is no hope.

It may seem strange to be using an equality where we can't generally prove things to be equal, but in fact the same is also true of the old familiar equality of numbers: it is possible to write down two numerical expressions that we can't prove are equal or not equal (proof by Dan Richardson). Of course, these equalities are practical enough to be useful for many other things.

3.8.2 Extensionality

Another point to note is that to prove β-equality of two expressions, you can only use the definitions above. It would be tempting to have a proof that went

blah, blah, and so $MX =_\beta NX$ for all terms X. Therefore $M =_\beta N$.

This is called *extensionality*, and is an analogue of

blah, blah, and so $f(x) = g(x)$ for all x, thus $f = g$

in Set theory.

But extensionality does not hold for lambda calculus. For example, take $M = y$ and $N = \lambda x.yx$. Certainly $M \neq_\beta N$ (both are in normal form), but

$$NX = (\lambda x.yx)X =_\beta yX = MX$$

for all X.

Computationally speaking, these terms are certainly different: N needs one more function application than M. On the other hand, their outcomes when applied to an argument are always the same: N just seems a less efficient way of doing the same thing as M.

> To think about: if two programs produce the same outputs for the same inputs for all inputs, are they the same program?
>
> We could define a kind of equality on programs, namely $N = M$ if N and M produce the same outputs for the same inputs for all inputs, but it would be in general impossible to test two programs for equality due to the Halting Problem. This is called *extensional* equality.

To gain extensionality, something extra has to be added to lambda calculus. One way is using η-reduction.

3.8.3 η-reduction

η-reduction is

$$\lambda x.Mx \succ_\eta M \text{ whenever } x \text{ is not free in } M.$$

Note that $(\lambda x.Mx)X \succ_\beta MX$ for any X, but $\lambda x.Mx \not\succ_\beta M$.

Examples: $\lambda x.yx \succ_\eta y$; $\lambda x.(\lambda x.x)x \succ_\eta \lambda x.x$; but $\lambda x.xx \not\succ_\eta x$.

Note also: $\lambda x.(\lambda x.x)x \succ_\beta \lambda x.x$ by β-reducing the inner redex. Similarly, $\lambda x.(\lambda z.z)x \succ_\beta \lambda x.x$ while $\lambda x.(\lambda z.z)x \succ_\eta \lambda z.z$, but these are still α-equal.

This is acknowledging that M is independent of x, and the λ is not really doing anything for us. Notice that this is *not* a reduction we could do previously, even though whenever we saw this term *in some β redex context* we could have reduced it.

It turns out that η-reduction satisfies substitionality and the other properties expected of an equality, and so is "well behaved," but does alter the collection of things that can be proved in the lambda calculus. It is not a question of whether it is "right or wrong" to use η-reduction, you just get a different range of theorems to explore.

Imagine changing the rules of Chess so that a pawn could always move one or two squares forward. This would be a new game, not Chess, but would have substantial similarities to Chess. Some Chess strategies might be still relevant, but others now fail. And some new strategies might work.

Similarly, some theorems from the pure lambda calculus might still be true: others can now be false. Or things that were false previously might now be true (e.g., extensionality in the case of η reduction).

Indeed, we have to go back to anything we have proven previously and re-prove it (if possible) for this new calculus. For example, it turns out that Church-Rosser still holds in the presence of η-reduction:

> If M, A and B are any λ-terms with $M \succ_{\beta\eta} A$ and $M \succ_{\beta\eta} B$, then there is a λ-term N with $A \succ_{\beta\eta} N$ and $B \succ_{\beta\eta} N$.

Many popular theorems do remain true, but some of the deeper theorems break down, which lead some to avoid the use of η.

We use the notation $\succ_{\beta\eta}$ for a sequence of zero or more β and η (and α) reductions. Similarly, $=_{\beta\eta}$ for a sequence of zero or more β and η (and α) reductions or reverse reductions.

η-reductions do have these nice properties:

- a λ-term has a $\beta\eta$ normal form if and only if it has a β normal form.
- in a $\beta\eta$-reduction we can postpone the η-reductions to the end, that is if $M \succ_{\beta\eta} N$, then there is a P with $M \succ_\beta P \succ_\eta N$.

So in some sense η is a very benign reduction, but there is one big benefit: lambda calculus with η-reduction has extensionality. Since $MX =_{\beta\eta} NX$ for all X implies

$$\begin{array}{lll} M =_{\beta\eta} & \lambda x.Mx & \text{by reverse } \eta\text{, where } x \notin FV(MN) \\ =_{\beta\eta} & \lambda x.Nx & \text{by } Mx =_{\beta\eta} Nx \text{ and substitionality} \\ =_{\beta\eta} & N & \text{by } \eta \end{array}$$

Some liken η-reduction to program optimisation: you take a program and alter it to be more efficient while still giving the same answers. In this sense η does not model computation as does β.

We shall not be using η-reduction, but shall stick to the *pure* λ-calculus.

3.8.4 δ reduction

Another form of reduction is δ reduction. This does not apply to the pure lambda calculus as we have developed it, but rather to an extended form that contains *constants*. These are things like, for example, integers (which are distinct from variables), perhaps together with some operations, say addition.

We get terms like

$$(\lambda x.x + 1)2 \succ_\beta 2 + 1 \succ_\delta 3$$

We shall look at δ reduction in detail later, in section 6.

3.9 Currying

What of multi-argument functions? For example, the summation function takes two arguments.

```
(defun sum (x y) (+ x y))
```

so that the value of `sum` is

```
(lambda (x y) (+ x y))
```

and is called as

```
(sum 8 11) -> 19
```

Surely we can't represent `(lambda (x y) (+ x y))` as λ-terms only have one argument?

Now, we have already seen the notational convenience of $\lambda xy.fxy$ to mean $\lambda x.(\lambda y.fxy)$. Taking this as inspiration we can define `csum` as

```
(defun csum (x) (lambda (y) (+ x y)))
```

Now the value of `csum` is

```
(lambda (x) (lambda (y) (+ x y)))
```

namely a function of one argument that returns a function of one argument.

```
(csum 7) -> <function>
```

where the function takes one argument and returns that argument plus 7. So

```
((csum 7) 11) -> 18
```

This is called *Currying*, after Curry, who first thought of the idea.

In λ-terms:

$$(\lambda x_1 x_2 \ldots x_r.E) M_1 M_2 \ldots M_r \succ [M_r/x_r] \ldots [M_2/x_2][M_1/x_1]E.$$

In Lisp, we can define a function

```
(defun curry (f)
  (lambda (x)
    (lambda (y)
      (f x y))))
```

Now, `curry` is a function that takes a function of two arguments and returns that function curried.

```
(setq add7 ((curry sum) 7))

(add7 5) ->
12

(((curry sum) 7) 11) ->
18
```

In the opposite direction,

```
(defun uncurry (f)
   (lambda (x y) ((f x) y)))
```

Here, `uncurry` takes a function of one argument that returns a function of one argument and returns a function of two arguments.

```
(defun uncurry (f)
  (lambda (x y)
     ((f x) y)))

(setq csum (curry sum))
(setq add (uncurry csum))

(add 2 3) ->
5
```

Currying proves that by restricting functions to a single argument does not reduce the expressive power of the lambda calculus.

3.10 Y operator

When writing a recursive function in Lisp (or other language) we need a name for the function we are writing in order to refer back to itself:

```
(defun fact (n)
   (if (< n 2) 1 (* n (fact (- n 1)))))
```

Here the inner `fact` is a reference back to the function being defined. The problem with lambda calculus is that we can't do this as lambdas don't have names!

So we need some other way of doing things. Consider what happens in a simple recursive function.

```
(defun g (a) (h (g a)))
```

Here `h` would include some conditional so that the recursion can terminate.

Expanding

```
(g a) -> (h (g a))
      -> (h (h (g a)))
      -> (h (h (h (g a))))
      -> ...
```

When we apply `g` to some argument, it will only recurse to some finite depth (or else the function fails to return!), but we can't tell in advance how deep it will go. So what we would like is to define `g` as

```
(defun g (a) (h (h (h ... a))))
```

for an infinite depth. This may seem strange, but with normal order reduction, it isn't so bad.

Let's look at `f` where

```
(defun f (x) (lambda (a) (h (x a))))
```

Then

```
(f f) -> (lambda (a) (h (f a)))
(f (f f)) -> (lambda (a) (h ((lambda (a) (h (f a))) a)))
          -> (lambda (a) (h (h (f a))))
(f (f (f f))) -> (lambda (a) (h (h (h (f a)))))
```

and so on. Thus, a n-fold application of `f` to `f` matches a n-fold depth of recursion for `g`.

So again, nesting infinitely

```
(f (f ... f)) -> (lambda (a) (h (h (h ... a))))
```

which is the same as the function `g`, as we will only ever recurse to a finite depth in an actual computation of `g`.

Now, `F = (f (f ... f))` has the curious property that

```
(f F) = F
```

`F` is called a *fixed point* for `f`.

A *fixed point* for a function f is a value x such that $f(x) = x$. Thus the (numerical) function $f(x) = x^2 - 3x$ has fixed point 4 as $f(4) = 4^2 - 3 \times 4 = 16 - 12 = 4$. It also has another fixed point, namely $x = 0$. The function $f(x) = x + 1$ has no fixed points, while $f(x) = x$ has an infinite number of fixed points. A numerical function can have zero to infinity fixed points: in the λ calculus there is a significant difference.

Returning to our F, if we could define such an `F` by some proper means, then we would have `g = F` as our recursive function.

In lambda calculus terms, `g` is

$$\lambda a.h(h(\ldots a))$$

which is *not* a λ-term, as it is infinitely long(!), and `f` is

$$M = \lambda xa.h(xa)$$

which is a perfectly respectable term.

Now we have

$$\begin{array}{rcl} MM & = & (\lambda xa.h(xa))M \\ & \succ & \lambda a.h(Ma) \\ M(MM) & = & (\lambda xa.h(xa))(\lambda a.h(Ma)) \\ & \succ & \lambda a.h((\lambda a.h(Ma))a) \\ & \succ & \lambda a.h(h(Ma)) \end{array}$$

and so on. And then an infinite application $F = M(M(\ldots M))$ does what we want. Again, we have $MF =_\beta F$, i.e., F is a fixed point for the term M.

It may seem impossible to figure out how to find a reasonable definition for F, but it turns out that there is a theorem.

Fixed Point Theorem:

Given any λ-term M there is a term X (depending on M) such that

$$MX =_\beta X$$

Thus: *every λ-term has at least one fixed point*. This is different from normal, numerical, functions that might not have a fixed point.

Proof:

Let $W = \lambda x.M(xx)$, and $X = WW$. Then

$$X = WW = (\lambda x.M(xx))W \succ M(WW) = MX.$$

Examples.

$M = x$. We get $W = \lambda y.M(yy) = \lambda y.x(yy)$ and so $X = (\lambda y.x(yy))(\lambda y.x(yy))$. Checking this: $X \succ x((\lambda y.x(yy))(\lambda y.x(yy))) = xX$, i.e., $xX =_\beta X$, as required. Note that $X \succ xX \succ xxX \succ xxxX \ldots$.

$M = \lambda x.x$. Now $X = (\lambda y.(\lambda x.x)(yy))(\lambda y.(\lambda x.x)(yy)) \succ (\lambda y.yy)(\lambda y.yy) = \Omega$. As noted above, everything is a fixed point of the identity function, but here we compute a specific example, which happens to be Ω.

Even better than the fixed point theorem, there is a λ-term Y, a *a fixed point operator*, such that for any M, YM is a fixed point for M. In other words

$$M(YM) =_\beta YM.$$

(In terms numerical functions, this would be a "function" y such that $y(f)$ is a number that is a fixed point for f, i.e., $f(y(f)) = y(f)$. But no such "function" y exists.)

Hence, given M, YM is the term we seek, and implements the recursive function we are looking for.

Fixed points are not unique: consider $\lambda x.x$, for which *everything* is a fixed point. Similarly, fixed point operators are not unique. Two popular ones are due to Curry and Turing.

Curry:

$$Y = \lambda x.VV \text{ where } V = \lambda y.x(yy)$$

Turing:

$$Y = ZZ \text{ where } Z = \lambda zx.x(zzx)$$

Taking Curry, for example, we have

$$\begin{array}{rcl} YM & = & (\lambda x.(\lambda y.x(yy))(\lambda y.x(yy)))M \\ & \succ & (\lambda y.M(yy))(\lambda y.M(yy)) \\ & \succ & M((\lambda y.M(yy))(\lambda y.M(yy))) \\ & \prec & M((\lambda x.(\lambda y.x(yy))(\lambda y.x(yy)))M) \\ & = & M(YM) \end{array}$$

Reading this other way, YM is a fixed point of M: $M(YM) =_\beta YM$. Turing's Y is somewhat better in that $YM \succ M(YM)$ rather than just $=_\beta$:

$$\begin{array}{rcl} ZZ & = & (\lambda zx.x(zzx))Z \\ & \succ & \lambda x.x(ZZx) \end{array}$$

and so

$$YM = ZZM \succ (\lambda x.x(ZZx))M \succ M(ZZM) = M(YM).$$

In fact, fixed point operators are rather common. For example, if

$$A = \lambda abcdefghijklmnopqrstuvwxyz.r(thisisafixedpointoperator)$$

and

$$B = AAAAAAAAAAAAAAAAAAAAAAAAAA$$

(26 times), then B is a fixed point operator.

Example.

$M = x$. Then using Curry: $YM = Yx = (\lambda y.(\lambda z.y(zz))(\lambda z.y(zz)))x \succ (\lambda z.x(zz))(\lambda z.x(zz))$, as before. Using Turing: $Yx = ZZx = (\lambda zy.y(zzy))Zx \succ (\lambda y.y(ZZy))x \succ x(ZZx) = x(Yx)$. This doesn't simplify to anything particularly pretty.

The Y operator translates directly into Lisp. We can define

```
(defun Y (f) ((lambda (x) (f (lambda (y) ((x x) y))))
              (lambda (x) (f (lambda (y) ((x x) y))))))
```

This is Turing's operator. Curry's doesn't work with an applicative order Lisp. Then

```
(defun FACT (f) (lambda (n) (if (< n 2) 1 (* n (f (- n 1))))))
```

```
(setq fact (Y FACT))
```

```
(fact 5) ->
120
```

The point here is that the recursive `fact` has been defined, via `FACT`, without self-reference. The `Y` ties the loop.

In general,

```
(defun foo (x) ( ... (foo ... ) ... (foo ... ) ... )
```

can be defined as

```
(defun FOO (f) (lambda (x) ... (f ... ) ... (f ... ) ... )
```

```
(setq foo (Y FOO))
```

For example, instead of

```
(defun len (l) (if (null l) 0 (+ 1 (len (cdr l)))))
```

We could have

```
(defun LEN (f) (lambda (l) (if (null l) 0 (+ 1 (f (cdr l))))))
```

```
(setq len (Y LEN))
```

More succinctly,

```
(setq len (Y (lambda (f) (lambda (l) (if (null l) 0 (+ 1 (f (cdr l))))))))
```

3.10.1 More on Y

We can use Y to solve equations:

for any λ-term M the equation

$$xy_1 \dots y_n = M$$

has a solution for x. That is, there is an X such that

$$Xy_1 \dots y_n =_\beta [X/x]M$$

Proof:

Take $X = Y(\lambda x y_1 \ldots y_n.M)$. Then $(\lambda x y_1 \ldots y_n.M)X = X$, so

$$
\begin{array}{rcl}
X y_1 \ldots y_n & = & (Y(\lambda x y_1 \ldots y_n.M)) y_1 \ldots y_n \\
& = & (\lambda x y_1 \ldots y_n.M) X y_1 \ldots y_n \\
& \succ & (\lambda y_1 \ldots y_n.[X/x]M) y_1 \ldots y_n \\
& \succ & [X/x]M
\end{array}
$$

For example, find a solution x to

$$xyz = zy.$$

(Note this is the equation $(xy)z = zy$.) We take

$$
\begin{array}{rcl}
X & = & Y(\lambda xyz.zy) \\
& = & (\lambda x.(\lambda y.x(yy))(\lambda y.x(yy)))(\lambda xyz.zy) \\
& \succ & (\lambda w.(\lambda xyz.zy)(ww))(\lambda w.(\lambda xyz.zy)(ww)) \\
& \succ & (\lambda w.\lambda yz.zy)(\lambda w.\lambda yz.zy) \\
& \succ & \lambda yz.zy
\end{array}
$$

And, of course,

$$(\lambda uv.vu)yz \succ (\lambda v.vy)z \succ zy.$$

3.11 Relationship to Lisp

λ is `lambda`!

Lisp the language was first developed as a way of implementing the lambda calculus. Over the years it has developed into a powerful tool in its own right, which is a testament to its original design, and the flexibility of the lambda calculus as a means of expressing concepts.

Unfortunately (or fortunately, depending on how you look at it), Lisp diverges from the pure lambda calculus. There are minor differences, such a multiple argument functions, but we know this is not important because of currying. It *is* important in terms of efficiency, though. Multiple nested function calls are more expensive than multiple arguments. In fact, most of the divergence of Lisp from lambda calculus is driven by efficiency concerns. The of recursion instead of the Y operator is just the same.

There are major differences: Lisp has objects other than symbols; it has side effects; it can give values names; it uses applicative order evaluation.

Lisp is unusual for a computer language in that it has symbols, corresponding to lambda calculus variables. In common with other languages, though, it has numbers and lists (amongst other types), which do not appear in pure lambda calculus. Though such things are representable in pure lambda calculus (see, for example, Church numerals, later) it is somewhat involved, so for efficiency these are primitives in Lisp.

Side effects are like the function `inc` where

```
(setq n 0)
(defun inc ()
  (setq n (+ n 1))
  1)
```

Calling `inc` has the side effect of updating the variable `n`. Side effects are sometimes useful: for example, the `print` function has the side effect of updating the contents of the screen!

However, side effects break *referential transparency*. An expression is referentially transparent when its value is independent of its environment. Thus any part of a referentially transparent expression maybe replaced by something which is "equal", and the value of the expression is unchanged.

Thus, in `(+ 1 (+ 2 3))` we can replace the `(+ 2 3)` by `5` giving the equivalent `(+ 2 5)`. But we cannot replace `inc` in `(+ 1 (inc))` to get `(+ 1 1)`, even though `inc` always returns `1`. This is because `n` would not be updated.

The point of referential transparency is that

equals may be substituted for equals

That is, substitutionality.

This means analysing a program is much easier: the order of evaluation is unimportant and side effects do not complicate things. For example, what is the value of `(list (inc) (inc))`?

Quite often, referential transparency is violated by the use of global variables. This is part of the approach of the object oriented programming style: by keeping state in objects we are looking towards referential transparency, which means we can understand a program just by looking at local sections, not having to comprehend the thing as a whole.

When Lisp was first developed, programmers did not have the knowledge of how to program efficiently in a referentially transparent way, so the purity of Lisp was broken, as we see with the `inc` example. Nevertheless, even though we *can* write programs in such a way, we should try to avoid such a use of global state (the variable n in this case).

> Avoiding global state turns out to be a good thing generally, regardless of programming language. In fact, one of the big benefits of OO programming is that it naturally avoids global state, bringing OO languages closer to the referentially transparent ideal. This is tied up with the idea of software reuse: a module is self-contained and has no unexpected side-effects.

We have seen how applicative order evaluation means that some λ-terms do not translate well into Lisp. Of course, this is an efficiency issue again, but it would be reasonably straightforward to change the evaluation in Lisp to normal order.

4 Modelling Computation using Lambda Calculus

So what can we do with lambda calculus? We have seen how it can (impurely) be represented as Lisp, but there is also a lot of other things we can do with it. In particular, we can model various aspects of mathematics and computing using λs, and using these models we can prove things about mathematics and computing. Lambda calculus is a universal model of computation, that is, any computation that can be expressed using a Turing machine can also be expressed in the lambda calculus.

4.1 Church Numerals

You may have seen set theory being used as a foundation of mathematics. This starts by the definitions

$$\begin{array}{ccc} 0 & \stackrel{\text{def}}{=} & \emptyset \\ 1 & \stackrel{\text{def}}{=} & \{0\} \\ 2 & \stackrel{\text{def}}{=} & \{0,1\} \\ 3 & \stackrel{\text{def}}{=} & \{0,1,2\} \\ & \dots & \end{array}$$

The we can define a *successor* function: $S(n) = n \cup \{n\}$. Intuitively, this means $S(n) = n + 1$, but we haven't defined addition yet. Thus $S(2) = 2 \cup \{2\} = \{0, 1\} \cup \{2\} = \{0, 1, 2\} = 3$.

Addition is a defined inductively: $n + 0 = n$ and $n + S(m) = S(n + m)$. So $2 + 2 = 2 + S(1) = S(2 + 1) = S(2 + S(0)) = S(S(2 + 0)) = S(S(2)) = S(3) = 4$. We need to prove the various facts we expect of addition, e.g., $n+m = m+n$, $n+(m+n) = (n+m)+n$, and indeed $S(n) = n+1$ (since $n+1 = n+S(0) = S(n+0) = S(n)$). Next we can define other operations like multiplication and exponentiation. At each state we must prove that these operators have the properties we expect of them.

Pairs can be defined as $(a, b) = \{a, \{a, b\}\}$. This has the expected property that $(a, b) = (c, d)$ implies $a = c$ and $b = d$. For consider $\{a, \{a, b\}\} = \{c, \{c, d\}\}$ and suppose $a \neq c$. Then we must have $a = \{c, d\}$ and $c = \{a, b\}$. So $a = \{\{a, b\}, d\}$, which contradicts the axiom of foundation. So $a = c$. Then $\{a, b\} = \{c, d\} = \{a, d\}$, whence $b = d$.

Exercise: find and fix the hole in this proof.
Answer: What if $a = b$?

Using integers we can define rationals as (classes of) pairs of integers. For example, the pair $(1, 2)$ could be a representation of the fraction $1/2$. Again, we need to define addition and so on for our fractions: $(a, b)+(c, d) = (ad+bc, db)$. Real numbers can be defined as (classes of) sequences of rationals, then complex numbers as pairs of reals. Every time we must include proof that the definitions are working well.

We shall now that we can use lambda calculus in place of set theory. That is, rather than using sets and set axioms as primitives, we can use λs.

Definition:

$$\begin{array}{ccc} \overline{0} & \stackrel{\text{def}}{=} & \lambda xy.y \\ \overline{1} & \stackrel{\text{def}}{=} & \lambda xy.xy \\ \overline{2} & \stackrel{\text{def}}{=} & \lambda xy.x(xy) \\ \overline{3} & \stackrel{\text{def}}{=} & \lambda xy.x(x(xy)) \\ & \dots & \\ \overline{n} & \stackrel{\text{def}}{=} & \lambda xy.\underbrace{x(x\dots(x}_{n \text{ times}}\,y)\dots) \\ & = & \lambda xy.x^n y. \end{array}$$

We use overlines to remind ourselves that they are Church numerals rather than simple numbers.

> The alert philosopher should be worried that we seem to be using natural integers in the definitions of Church numerals. Be assured that things can be formulated without any reference to Set theory.
>
> In Set, "n" is "having n things", while in the Lambda Calculus it is "doing n things".

These numerals are all in normal form. For any terms M and M we have

$$\overline{n}MN \succ_\beta \underbrace{M(M\ldots(M}_{n \text{ times}}N)\ldots),$$

which is M applied to N n times. So the Church numeral $\overline{n}$ is an "apply n times" operator.

For a number $\overline{n}$ its *successor* $S\overline{n}$ is given by

$$S = \lambda n.\lambda xy.x(nxy),$$

(i.e., apply x n times, then one more time).

Example.

$$\begin{array}{rcll} S\overline{2} & = & (\lambda n.\lambda xy.x(nxy))\overline{2} & \\ & \succ & \lambda xy.x(\overline{2}xy) & \\ & \succ & \lambda xy.x(x(xy)) & \text{as } \overline{2}xy \succ x(xy) \\ & = & \overline{3}. & \end{array}$$

In general:

$$\begin{array}{rcll} S\overline{n} & = & (\lambda n.\lambda xy.x(nxy))\overline{n} & \\ & \succ & \lambda xy.x(\overline{n}xy) & \\ & \succ & \lambda xy.x(x^n y)) & \text{as } \overline{n}xy \succ x^n y \\ & = & \lambda xy.x^{n+1}y & \\ & = & \overline{n+1}. & \end{array}$$

That is

$$S\overline{n} \succ \overline{n+1}.$$

We can define addition in terms of S

$$A = \lambda mn.mSn,$$

(i.e., apply S m times to n; which means to add 1 m times) or directly as

$$A = \lambda mn.\lambda xy.mx(nxy),$$

(i.e., apply n times, then m more times) so that $A\overline{m}\,\overline{n} \succ \overline{m+n}$.

For example

$$\begin{array}{rcl} A\overline{2}\,\overline{3} & = & (\lambda mn.\lambda xy.mx(nxy))\overline{2}\,\overline{3} \\ & \succ & \lambda xy.\overline{2}x(\overline{3}xy) \\ & \succ & \lambda xy.x^2(x^3 y) \\ & = & \lambda xy.x^5 y \\ & = & \overline{5} \end{array}$$

Exercise: prove $A\overline{m}\,\overline{n} \succ \overline{m+n}$.

Similarly, multiplication is

$$M = \lambda mn.\lambda xy.m(nx)y,$$

(i.e., apply "apply n times" m times) so that $M\overline{m}\,\overline{n} \succ \overline{mn}$.

For example

$$\begin{array}{rcl} M\overline{2}\,\overline{3} & = & (\lambda mn.\lambda xy.m(nx)y)\overline{2}\,\overline{3} \\ & \succ & \lambda xy.\overline{2}(\overline{3}x)y \\ & \succ & \lambda xy.(\overline{3}x)(\overline{3}xy) \\ & \succ & \lambda xy.(\overline{3}x)(x^3y) \\ & \succ & \lambda xy.x^3(x^3y) \\ & = & \lambda xy.x^6y \\ & = & \overline{6} \end{array}$$

Exercise: prove $M\overline{m}\,\overline{n} \succ \overline{mn}$.

Now $\overline{m}\,\overline{n}M \succ \underbrace{\overline{n}(\overline{n}\ldots(\overline{n}}_{m \text{ times}}M)\ldots) = \overline{n}^m M$, so that exponentiation is

$$E = \lambda mn.\lambda xy.nmxy.$$

and $E\overline{m}\,\overline{n} \succ \overline{m^n}$.

Sometimes you will see the the definition $E = \lambda xy.yx$. This *almost* works with β reduction, but fails on $\overline{0}\overline{n} \succ_\beta \lambda z.z \neq_\beta \overline{1}$. On the other hand, we do have $\lambda z.z =_\eta \overline{1}$, so if we allow ourselves η reduction, we can use this simpler form for exponentiation.

Exercise: Prove, without using any property of the (normal) integers, that

for all n	$A\overline{n}\,\overline{0} = A\overline{0}\,\overline{n} = \overline{n}$	$n + 0 = 0 + n = n$
for all n and m	$A\overline{n}\,\overline{m} = A\overline{m}\,\overline{n}$	$n + m = m + n$
for all n, m and p	$A\overline{n}(A\overline{m}\,\overline{p}) = A(A\overline{n}\,\overline{m})\overline{p}$	$n + (m + p) = (n + m) + p$
for all n	$M\overline{n}\,\overline{0} = M\overline{0}\,\overline{n} = \overline{0}$	$n0 = 0n = 0$
for all n	$M\overline{n}\,\overline{1} = M\overline{1}\,\overline{n} = \overline{n}$	$n1 = 1n = n$
for all n and m	$M\overline{n}\,\overline{m} = M\overline{m}\,\overline{n}$	$nm = mn$
for all n, m and p	$M\overline{n}(M\overline{m}\,\overline{p}) = M(M\overline{n}\,\overline{m})\overline{p}$	$n(mp) = (nm)p$
for all n	$E\overline{n}\,\overline{0} = \overline{1}$	$n^0 = 1$
for all $n > 0$	$E\overline{0}\,\overline{n} = \overline{0}$	$0^n = 0$
for all n	$E\overline{n}\,\overline{1} = \overline{n}$	$n^1 = n$
for all n	$E\overline{1}\,\overline{n} = \overline{1}$	$1^n = 1$
for all n	$E\overline{n}\,\overline{2} = M\overline{n}\,\overline{n}$	$n^2 = n \times n$
for all n, m and p	$M\overline{n}(A\overline{m}\,\overline{p}) = A(M\overline{n}\,\overline{m})(M\overline{n}\,\overline{p})$	$n(m + p) = nm + np$
and so on.		

(Of course, those =s above are actually $=_\beta$s). The "cheating" proofs of these go like: $A\overline{n}\,\overline{0} = \overline{n+0} = \overline{n}$. However, we can prove all these equalities without reference to the natural integers.

All of this works very naturally in Lisp:

```
(defun int2church (n)
  (lambda (x)
    (lambda (y)
      (int2churchrec n x y))))

; apply x n times to y
(defun int2churchrec (n x y)
  (if (= n 0)
      y
      (x (int2churchrec (- n 1) x y))))
```

```
; apply ``add 1'' to 0 n times
(defun church2int (n)
  ((n (lambda (x) (+ x 1))) 0))

(defconstant zero (int2church 0))
(defconstant one (int2church 1))
(defconstant two (int2church 2))
(defconstant three (int2church 3))
```

And then

```
; successor
(defun S (n)
  (lambda (x)
    (lambda (y)
      (x ((n x) y)))))

; add
(defun A (m)
  (lambda (n)
    (lambda (x)
      (lambda (y)
        ((m x) ((n x) y))))))

; multiply
(defun M (m)
  (lambda (n)
    (lambda (x)
      (lambda (y)
       ((m (n x)) y)))))

; exponentiate
(defun E (m)
  (lambda (n)
    (lambda (x)
      (lambda (y)
        (((n m) x) y)))))

(church2int ((E two) three)) ->
8
```

A rather more elegant definition for `int2church` uses `S`

```
(defun int2church (n)
   (if (= n 0)
       (lambda (x) (lambda (y) y))          % viz 0
       (S (int2church (- n 1)))))
```

Church numerals display an interesting property of λ-terms, namely that their normal forms can be very large. Look at the sequence

$$\overline{2}, \quad \overline{2}\,\overline{2}, \quad \overline{2}(\overline{2}\,\overline{2}), \quad \overline{2}(\overline{2}(\overline{2}\,\overline{2})), \quad \ldots$$

which are λ-terms of lengths roughly

$$3, \quad 6, \quad 9, \quad 12, \quad \ldots$$

but whose normal forms (i.e., $\lambda xy.x(x(x \ldots (xy) \ldots)))$ are of lengths roughly

$$2, \quad 2^2, \quad 2^{2^2}, \quad 2^{2^{2^2}}, \quad \ldots$$

So, while the original lambda terms increase linearly in length their normal forms are super-exponential in length. This shows that λ-term can have a normal form that is arbitrarily longer than itself. Calling this "reduction" is perhaps a little misleading!

Subtraction comes later: it is surprisingly hard!

4.2 Cons, car, cdr as lambdas

We can make datastructures like lists using the lambda calculus.

Define the *pair*

$$<M, N> = \lambda v.vMN,$$

where v does not appear free in M or N.

Whenever we say "Let $A =$ blah" what we mean is "I am going to use A as a shorthand for blah". The A is *not* part of the λ-calculus, but just convenient notation so we don't have to write out the full expressions everywhere.

Then define

$$\begin{aligned} P &= \lambda xy.\lambda z.zxy \\ F &= \lambda p.p(\lambda xy.x) \\ R &= \lambda p.p(\lambda xy.y) \end{aligned}$$

We get

$$\begin{aligned} PMN &= (\lambda xy.\lambda z.zxy)MN \\ &\succ (\lambda y.\lambda z.zMy)N \\ &\succ \lambda z.zMN \\ &= <M, N> \end{aligned}$$

$$\begin{aligned} F<M, N> &= (\lambda p.p(\lambda xy.x))<M, N> \\ &\succ <M, N>(\lambda xy.x) \\ &= (\lambda v.vMN)(\lambda xy.x) \\ &\succ (\lambda xy.x)MN \\ &\succ (\lambda y.M)N \\ &\succ M \end{aligned}$$

Similarly,

$$R\langle M, N\rangle \succ N.$$

Thus we have P for pair (or cons), F for first (or car), and R for rest (or cdr).

Exercise: Prove $\langle A, B\rangle = \langle C, D\rangle$ if and only if $A = B$ and $C = D$.

From the positive integers (Church numerals) and pairs, we can define the negative integers; from these we can define rationals; and then reals and complexes. Lambda calculus can replace set theory as a foundation for arithmetic.

In Lisp:

```
; cons
(defun P (x)
  (lambda (y)
    (lambda (z)
      ((z x) y))))

; car
(defun F (p)
  (p (lambda (x) (lambda (y) x))))

; cdr
(defun R (p)
  (p (lambda (x) (lambda (y) y))))
```

4.3 Predecessor and Subtraction

Subtraction of Church numerals would be nice to have, as long as we take care not to end up with a negative number. This can be built from a predecessor function, where we define the predecessor of $\overline{0}$ to be $\overline{0}$.

First we define an operator U on pairs such that

$$U\langle\overline{m}, \overline{n}\rangle \succ \langle\overline{n}, \overline{n+1}\rangle$$

as the idea is to pair $\overline{n+1}$ with its predecessor $\overline{n}$.

We can define

$$U = \lambda p.P(Rp)(S(Rp)),$$

i.e, the pair consisting of the cdr of p and 1 plus the cdr of the pair p.

Next,

$$\underbrace{U(U(\ldots U}_{n>0 \text{ times}} \langle\overline{0}, \overline{0}\rangle)\ldots) \succ \langle\overline{n-1}, \overline{n}\rangle$$

But the operator $\overline{n}$ applies something n times, so this is

$$\overline{n}U\langle\overline{0}, \overline{0}\rangle \succ \langle\overline{n-1}, \overline{n}\rangle \qquad \text{if } n > 0$$

So if we define

$$\begin{array}{rcl} D & = & \lambda n.F(nU{<}\overline{0},\overline{0}{>}) \\ & = & \lambda n.(\lambda p.p(\lambda xy.x)) \\ & & (n(\lambda p.(\lambda xy.\lambda z.zxy)((\lambda p.p(\lambda xy.y))p)((\lambda n.\lambda xy.x(nxy))((\lambda p.p(\lambda xy.y))p))) \\ & & (\lambda v.v(\lambda xy.y)(\lambda xy.y))) \end{array}$$

when written out in full, we see that, for example,

$$\begin{array}{rcl} D\overline{3} & = & (\lambda n.F(nU{<}\overline{0},\overline{0}{>}))\overline{3} \\ & \succ & F(\overline{3}U{<}\overline{0},\overline{0}{>}) \\ & \succ & F({<}\overline{2},\overline{3}{>}) \\ & \succ & \overline{2} \end{array}$$

Now we can define a form of subtraction (*truncated subtraction*) by

$$T = \lambda mn.nDm,$$

then

$$T\overline{m}\,\overline{n} = \begin{cases} \overline{m-n} & \text{if } m \geq n \\ \overline{0} & \text{otherwise} \end{cases}$$

The toaster algorithm. How to make perfect toast when the toaster always overdoes it by 10 seconds. Get a second identical toaster. Put bread in the first, wait 10 seconds, then put bread in the second. When the first pops up with its overdone toast, extract the perfectly done toast from the second.

In Lisp:

```
(defun U (p)
  ((P (R p)) (S (R p))))

(defconstant zerozero ((P zero) zero))

; decrement
(defun D (n)
  (F ((n U) zerozero)))

; truncated subtraction
(defun T (m)
  (lambda (n)
    ((n D) m)))

(church2int ((T two) three)) ->
0

(church2int ((T three) two)) ->
1
```

4.4 Equality and Comparison

Now we can test for equality of Church Numerals. Define

$$Z = E\overline{0}$$

so that

$$\begin{array}{rcl} Z\overline{n} & = & E\overline{0}\,\overline{n} \\ & \succ & \begin{cases} \overline{1} & \text{if } n = 0 \\ \overline{0} & \text{otherwise} \end{cases} \end{array}$$

So Z is a test for zero, where we are using $\overline{1}$ to indicate true, and $\overline{0}$ to indicate false. Next,

$$L = \lambda mn.Z(Tmn)$$

has

$$\begin{array}{rcl} L\overline{m}\,\overline{n} & \succ & Z(T\overline{m}\,\overline{n}) \\ & \succ & \begin{cases} Z(\overline{m-n}) & \text{if } m > n \\ Z\overline{0} & \text{otherwise} \end{cases} \\ & \succ & \begin{cases} \overline{0} & \text{if } m > n \\ \overline{1} & \text{otherwise} \end{cases} \\ & = & \begin{cases} \overline{1} & \text{if } m \leq n \\ \overline{0} & \text{otherwise} \end{cases} \end{array}$$

Thus $L\overline{m}\,\overline{n}$ tests whether $m \leq n$.

Now,

$$Q = \lambda mn.M(Lmn)(Lnm)$$

has

$$\begin{array}{rcl} Q\overline{m}\,\overline{n} & \succ & M(L\overline{m}\,\overline{n})(L\overline{n}\,\overline{m}) \\ & \succ & \begin{cases} M\overline{1}\,\overline{0} & \text{if } m < n \\ M\overline{1}\,\overline{1} & \text{if } m = n \\ M\overline{0}\,\overline{1} & \text{if } m > n \end{cases} \\ & \succ & \begin{cases} \overline{1} & \text{if } m = n \\ \overline{0} & \text{otherwise} \end{cases} \end{array}$$

And so Q is a test for equality of Church numerals. Note that we can't test arbitrary λ-terms for equality! Since all Church numerals have normal forms, we could check that way: find normal form then check for α-equality. On the other hand, Q gives us a computed value without having to look at the structure of the terms.

In Lisp:

```
; = 0
(defconstant Z (E zero))

; <=
(defun L (m)
  (lambda (n)
    (Z ((T m) n))))

; =
(defun Q (m)
  (lambda (n)
```

```
    ((M ((L m) n)) ((L n) m))))

(church2int ((Q one) one)) ->
1

(church2int ((Q one) three)) ->
0
```

4.5 Booleans

The notion of Booleans can be taken further. For example, we may define an AND operator

$$AND = M,$$

as multiplication does the right thing on $\overline{0}$ and $\overline{1}$.

However, there is an alternative formulation of Booleans that tends to be more useful. This starts with

$$\begin{aligned} T &= \lambda xy.x \\ F &= \lambda xy.y \end{aligned}$$

T picks the first, while F picks the second: $TMN \succ M$, while $FMN \succ N$. Note these are the same as the car and cdr operators!

Then the Boolean operations are

$$\begin{aligned} AND &= \lambda mn.mnF \\ OR &= \lambda mn.mTn \\ NOT &= \lambda m.mFT \end{aligned}$$

This aligns more closely with the way we tend to think of the Boolean operations. If m is true then $m\,AND\,n$ is true exactly when n is true, else $m\,AND\,n$ is false: if m is true then mnF picks the n, otherwise it picks the F. This is the view that $m\,AND\,n$ is "if A then B else F".

Now we can prove the usual things about Boolean operators, like commutativity, associativity, idempotency, identities, distributivity, de Morgan, and so on.

Exercise: do this.

4.6 Other Datatypes

It is easy to see how we may construct other datatypes. For example, a string could be a list of numbers (where a list is a sequence of pairs, just as you might expect).

Aggregate types can be make out of simple ones using lists, and so on.

4.7 Control Structures

In Lisp there is no real distinction between program and data: they look alike. The same is true in the Lambda calculus: just as data can be modelled by λ-terms, so can program.

Here we model if-then-else:

$$IF = \lambda m.\lambda xy.mxy$$

That's all there is to it!

We find that

$$\begin{aligned} IF\,TMN &\succ (\lambda xy.Txy)MN \\ &\succ TMN \\ &\succ M \end{aligned}$$

and

$$\begin{aligned} IF\,FMN &\succ (\lambda xy.Fxy)MN \\ &\succ FMN \\ &\succ N \end{aligned}$$

Thus $IF\,BMN$ acts like the Lisp `(if B M N)`.

4.8 Summary

Thus we have built all the constructs we need in a computer language: numbers, comparisons, recursive functions and conditionals. This means that we can take an arbitrary algorithm and implement in the lambda calculus, and then use the tools of reduction to analyse the algorithm.

5 Combinators

The idea of taking operators and giving them names (e.g., the A and M operators on Church Numerals) can be taken a long way.

The purpose of *combinators* is to package up some useful operators and to do lambda calculus (initially) without variables. This was developed by Schönfinkel in the 1920s and later by Curry.

A lambda abstraction without any free variables is known as a *combinator*. For example,

$$S = \lambda xyz.xz(yz) \qquad K = \lambda xy.x \qquad I = \lambda x.x$$

These three are not independent, for consider

$$\begin{aligned} SK &= (\lambda xyz.xz(yz))(\lambda xy.x) \\ &\succ \lambda yz.(\lambda xy.x)z(yz) \\ &\succ \lambda yz.(\lambda y.z)(yz) \\ &\succ \lambda yz.z \end{aligned}$$

So

$$\begin{aligned} SKK &\succ (\lambda yz.z)(\lambda xy.x) \\ &\succ \lambda z.z \\ &= I \end{aligned}$$

In summary,

$$SKK =_\beta I$$

Rather than defining S and K in terms of lambdas, we can go in the opposite direction, and have S and K as primitives, forgetting, for a moment, the existence of the lambda calculus. The *combinator calculus* is defined as strings of Ss, Ks, and optionally Is, with parentheses for grouping.

A *combinator term* is

- an S or a K or (optionally) an I, or
- (ab) where a and b are combinator terms.

Thus, for example, I, (SK), $((SK)K)$, $((SS)(II))$ are combinator terms.

As usual, we drop parentheses and associate to the left, so SKK means $((SK)K)$.

There are a few reduction rules and nothing else:

$$\begin{array}{rcl} ((Ka)b) & \succ & a \\ (((Sa)b)c) & \succ & ((ac)(bc)) \\ (Ia) & \succ & a \end{array}$$

or, dropping parentheses:

$$\begin{array}{rcl} Kab & \succ & a \\ Sabc & \succ & ac(bc) \\ Ia & \succ & a \end{array}$$

where lower case letters represent arbitrary expressions. Note that these are *not* β reductions: this is *not* Lambda calculus, but the *Combinator calculus*.

Note that the word *combinator* now has two meanings: firstly, a λ-term with no free variables; and secondly a term as defined above. Usually, you can tell by the context, but it is important to know which you are talking about.

Note that

1. there are no variables;
2. there is no α renaming;
3. there is no substitution.

Just as in the lambda calculus, we use $=$ for a sequence of reductions or reverse reductions.

The reduction rules allow us to prove things like

$$SKKx \succ Kx(Kx) \succ x \prec Ix,$$

for all combinator terms x, thus

$$SKKx = Ix \qquad \forall x$$

so we could use SKK in place of I. In fact, most definitions of combinators just use S and K, and then define I as syntactic abbreviation for SKK. We can *either*

- choose to have S, K and I and note that SKK and I have the same properties but are not equal (no extensionality), *or*

- have just S and K and define I to be convenient abbreviation of SKK.

Another reduction is $SIIa \succ Ia(Ia) \succ aa$, so SII is like $\omega = \lambda x.xx$. And then $SII(SII) \succ SII(SII)$ is an infinite loop, just like $\Omega = \omega\omega$.

Notice that lambda calculus and combinators are not completely the same, since in lambda calculus we can *prove* that $SKK = I$, while the combinator rules don't let us do this.

If we have $X = \lambda x.xKSK$, then $XXX \succ K$ and $X(XX) \succ S$ so $X(XX)(XXX)(XXX) \succ I$. But there doesn't seem to be a nice combinator-style definition for X.

$XX = (\lambda x.xKSK)(\lambda x.xKSK) \succ (\lambda x.xKSK)KSK \succ (KKSK)SK \succ KKSK \succ KK$ so $XXX \succ KKX \succ K$ and $X(XX) \succ X(KK) \succ KKKSK \succ KSK \succ S$.

5.1 Other Properties

5.1.1 Normal Forms, Church-Rosser

Just as in lambda calculus we have normal forms. These are when we don't have a Kab or a $Sabc$ to reduce.

For example KS and SKK.

Next, we can prove a Church-Rosser theorem:

> if $u \succ x$ and $u \succ y$ then there is a z with $x \succ z$ and $y \succ z$.

So again, if a term has a normal form, it is unique. We see from $SII(SII)$ that normal forms need not exist.

5.1.2 Applicative and Normal Order Reduction

We can reduce inside-out (applicative order) or outside-in (normal order), just as before. Normal order reduction will reach a normal form, if it exists.

Example: $KI(SII(SII)) \succ KI(SII(SII)) \succ \ldots$, for applicative order, while $KI(SII(SII)) \succ I$ for normal order.

5.1.3 Church Numerals

We can define $\overline{0} = KI$, $\overline{1} = S(S(KS)K)(KI)$, $\overline{2} = S(S(KS)K)(S(S(KS)K)(KI))$, and so on. Here $S(S(KS)K)$ is a successor combinator.

More succinctly, let $B = S(KS)K$, then $\overline{n} = (SB)((SB)((SB)\ldots(KI))) = (SB)^n(KI)$.

Note that $Bxyz = S(KS)Kxyz = KSx(Kx)yz = S(Kx)yz = Kxz(yz) = x(yz)$.

Just as for lambda numerals, we find

$$\overline{n}Fx = F^n x.$$

Exercise: Define addition, multiplication, etc.

5.1.4 Datastructures

Exercise: Define pairs, car and cdr.

5.1.5 Booleans

Exercise: Define *true* and *false*, and the boolean operators.

Answer: $T = K$, $F = KI$.

5.1.6 Fixed Points

Let

$$V = S(S(KS)K)(K(SII))$$

and then

$$Y = SVV$$

has

$$Ym = m(Ym),$$

as before. This combinator is derived from Curry's Y. We can use this Y in the same way to define recursive functions and solve equations.

We find

$$\begin{array}{rcl} Vm & = & S(S(KS)K)(K(SII))m \\ & = & S(KS)Km(K(SII)m) \\ & = & KSm(Km)(SII) \\ & = & S(Km)(SII) \end{array}$$

And then

$$\begin{array}{rcl} Vmn & = & S(Km)(SII)n \\ & = & Kmn(SIIn) \\ & = & m(In(In)) \\ & = & m(nn) \end{array}$$

So

$$\begin{array}{rcl} Ym & = & SVVm \\ & = & Vm(Vm) \\ & = & m(Vm(Vm)) \\ & = & m(SVVm) \\ & = & m(Ym) \end{array}$$

5.1.7 Extensionality

The combinator calculus is not extensional. For example, $S(KK)Ix = KKx(Ix) = Kx$ for all x, but $S(KK)I \neq K$ (both sides are in normal form).

Adding extensionality to combinators is much harder than for the lambda calculus, and involves ideas that we don't have time for here.

5.2 Extended Combinators and Equivalence to Lambda Calculus

One of the major benefits of the combinator calculus is that it does not have variables: this makes everything nice and simple. We want to show that the Lambda Calculus and the Combinator Calculus are "equivalent," in the sense that they have equal expressive power. To do this we need to model combinators as λs and λs as combinators, which requires introducing variables to combinators.

Extending the combinator calculus with variables is easy: just include symbols like a, b, c alongside S and K. There are no bound variables and no substitution. The reduction rules are unchanged.

It is now possible to show that the combinator calculus using just S and K (or S, K and I) has equal expressive power to the lambda calculus.

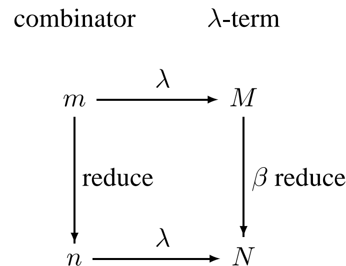
> Theorem: for every λ-term there is an equivalent (extended) combinator term.

Closed λ-terms are equivalent to pure combinators.

Translation of combinators to λ-terms is simple: use the lambda calculus definitions for S and K (and I) above:

$$\begin{array}{ccc} S & \rightarrow & \lambda xyz.xz(yz) \\ K & \rightarrow & \lambda xy.x \\ I & \rightarrow & \lambda x.x \end{array}$$

It is easy to see that the behaviour of the translations is the same as the behaviour of the combinators. After all, that's how we started.



Each reduction in the combinator calculus is reflected by a reduction in the Lambda calculus.

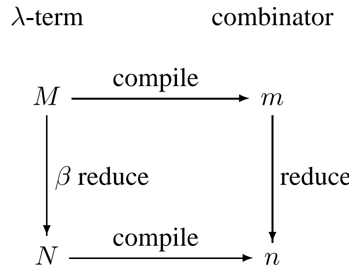
Translation of λ-terms to combinators is harder. Define C and A by

- $C(x) = x$ for a variable x
- $C(c) = c$ for a constant c (i.e., only Ss or Ks)
- $C(MN) = C(M)C(N)$ application
- $C(\lambda x.M) = A(x, C(M))$ abstraction
- $A(x, x) = I$
- $A(x, c) = Kc$ for a constant c (this is actually a special case of the next rule)
- $A(x, M) = K\,C(M)$ for $x \notin FV(M)$
- $A(x, MN) = S\,A(x, M)A(x, N)$

Note that $A(x, M)x = M$ for any combinator expression M:

- $A(x, S)x = K\,C(S)x = KSx = S$
- $A(x, K)x = K\,C(K)x = KKx = K$
- $A(x, x)x = Ix = x$
- $A(x, y)x = KC(y)x = Kyx = y$
- $A(x, MN)x = S\,A(x, M)A(x, N)x = A(x, M)x(A(x, N)x) = MN$, by structural induction

This is sometimes called *compiling* λ-terms. If we were proving things, we would now have to finish showing how the resulting λ-terms behave in the same way as the original combinators.



Each reduction in the Lambda calculus is reflected by a reduction in the combinator calculus.

Example. Compile $\lambda xy.y$.

$$\begin{aligned} C(\lambda xy.y) &= A(x, C(\lambda y.y)) \\ &= A(x, A(y, C(y))) \\ &= A(x, A(y, y)) \\ &= A(x, I) \\ &= KI \end{aligned}$$

And $KIxy = Iy = y$.

Example. Compile $\lambda x.y$. This has a free variable.

$$\begin{aligned} C(\lambda x.y) &= A(x, C(y)) \\ &= A(x, y) \\ &= Ky \end{aligned}$$

And $Kym \succ y$ as required.

Example. Compile $\lambda xy.x$.

$$\begin{aligned} C(\lambda xy.x) &= A(x, C(\lambda y.x)) \\ &= A(x, A(y, C(x))) \\ &= A(x, A(y, x)) \\ &= A(x, Kx) \\ &= S\,A(x, K)A(x, x) \\ &= S(KK)I \end{aligned}$$

We might have expected this to produce K, but we have $S(KK)I$ which does not reduce. Note, though, that $S(KK)Ix = KKx(Ix) = K(Ix) = Kx$ for all x. So, While $S(KK)I$ and K have equivalent behaviour, they are not actually equal. This is lack of extensionality again.

Example. Compile $\lambda x.Mx$, where $x \notin FV(M)$. This term was important regarding extensionality in the lambda calculus.

$$\begin{array}{rcl} C(\lambda x.Mx) & = & A(x, C(Mx)) \\ & = & A(x, C(M)C(x)) \\ & = & A(x, mx) \qquad \text{where } m = C(M) \\ & = & S\,A(x,m)A(x,x) \\ & = & S(Km)I \end{array}$$

And $S(Km)Iy = Kmy(Iy) = my$ for all y. We might hope that one possible way of adding extensionality to combinators would be to add the rule

$$S(Km)I \succ m$$

for all m. This new rule does allow us to prove

$$xm = xn \text{ for all } x \text{ implies } m = n$$

since $m = S(Km)I = S(Kn)I = n$. Though this is not quite what we want. And proving that adding this new rule is consistent is harder.

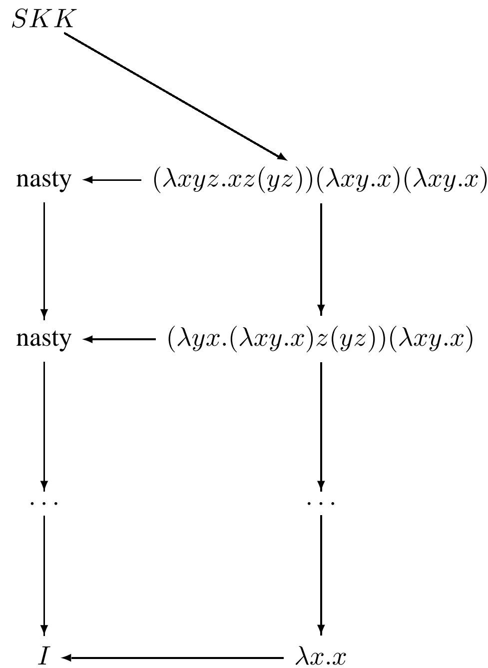
The above is just one way of compiling λ-terms to combinators. There have been many attempts to derive compilation rules that produce simpler combinators. Most schemes introduce new combinators, as discussed below.

How can we reconcile the fact that SKK is irreducible in the combinator calculus, while $(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda xy.x)$ simplifies in the lambda calculus with this equivalence?

$$\begin{array}{ccc} SKK & \xrightarrow{\lambda} & (\lambda xyz.xz(yz))(\lambda xy.x)(\lambda xy.x) \\ \Big\downarrow \text{reduce} & & \Big\downarrow \beta \text{ reduce} \\ ? & \xleftarrow{\text{compile}} & (\lambda yz.(\lambda xy.x)z(yz))(\lambda xy.x) \end{array}$$

If the λ-term on the right reduces, the equivalence says the combinator term on the left must reduce?

In reality, this is not a true picture. It is actually



The λ-term on the right compiles to something big and nasty on the left. Whatever it is, it can reduce to I. This nasty thing has the same behaviour as SKK, but is not equal to it.

In fact, a moment's reflection shows that there cannot be a compilation scheme that maps $(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda xy.x)$ to SKK, as the former reduces and the latter does not.

> We have equivalence, not an isomorphism: if $\mathcal{F}$: combinator $\rightarrow$ λ and $\mathcal{G}$: λ $\rightarrow$ combinator, then $\mathcal{GF} \neq$ identity, for example $\mathcal{GF}(K) = \mathcal{G}(\lambda xy.x) = S(KK)I \neq K$.
>
> However, we always find that $\mathcal{GF}(c) =$ something with equivalent behaviour to c.
>
> Note, also, that $\mathcal{FG}(L) \succ_{\beta\eta} L$ for all λ-terms L if we allow η reduction, as η reduction says "all λ-terms with the same behaviour reduce to the same λ-term".

5.3 Combinators for Computation

Turner 1979. Conversion of a program to combinators. Reduce the combinators to execute the program. Extend combinators with variables, integers, $+$, $=$, cond, Y, I, etc.

If we let

$$F = S(K(S((S((S(K\text{if})((S <)(K2))))(K1))))(S(K(S*))(S(S(KS)K)(K((S-)(K1)))))$$

then

$$f = YF$$

is the factorial function. In this, `if` would be the compilation of the λ-term for if-then-else; `2` the Church numeral; and `*` etc. the Church numeral arithmetic operators. Alternatively, we add these constructs as natives to the language.

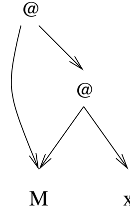


Figure 4: Shared Structure

5.3.1 Hardware Implementations

This idea behind combinator computation is that you can use parallel reduction to increase speed of execution. In an expression like

$$(II)(KII)$$

we can reduce the I simultaneously with the K as we know that there can be no interaction between the two parts, or any side effects to confuse things. Thus, if we have a computer with more than one processor, we can trivially increase the speed of execution by parallel reduction.

Similarly, in $S(SSSI)(SSSI)$ we know that any reduction of the first part can be mirrored in the second part, that is, we can reduce the first as far as we like, and simply copy the result over to the second. Computationally, we can go further: make both partss pointers to the *same structure*, so that reduction of the first *is* reduction of the second. This structure sharing reduces space and increases speed again.

This leads naturally to the idea of representing λ-terms as graphs: $M(Mx)$ becomes the graph in Figure 4.

Here @ means "apply".

There were several projects that aimed to implement these ideas in hardware. The vision was to gain speed over conventional architectures from parallel reduction.

ALICE (Applicative Language Idealised Computing Engine), designed by Darlington and Reeve at Imperial College in 1981. Intention to implement in VLSI, but only ever used discrete components.

The graph reducer GRIP (Graph Reduction in Parallel) was built from a distributed network of conventional processors.

SKIM (The SKI Machine). For reducing pure combinators.

But there is a problem that all these had, namely quadratic expression swell (in the number of abstractions). When we compile to combinators we find that the size of expressions grow immensely. For example

$$\begin{aligned} C(\lambda xyz.xz(yz)) \quad = \quad & S(KS)(S(K(KS))(S(KS)(S(S(KK)(S(KK)I)) \\ & (K(S(KS)(S(KI)(S(KS)(S(S(KK)I)(KI)))))))) \end{aligned}$$

rather than just S.

If a λ-term is $O(n)$ in size, then the compiled combinator is often $O(n^2)$.

Adding a new rule

- $A(x, Mx) = C(M)$, when $x \notin FV(M)$

can help sometimes. We know that $\lambda x.Mx$ has the same behaviour as M (they are extensionally equal) so we might as well compile to the simpler form. So adding this rule produces combinators with equivalent behaviour.

With this rule, we find that $C(\lambda xyz.xz(yz)) = S$, which is considerably better!

However, we now also need to allow η reduction in the λ-terms. For example:

$$\begin{array}{rcll} C(\lambda xy.xy) & = & A(x, A(y, xy)) & \\ & = & A(x, x) & \text{by application of the new rule} \\ & = & I & \end{array}$$

and $\lambda xy.xy = \lambda x.(\lambda y.xy) \succ_\eta \lambda x.x$. On the other hand, we get $(S((KS)(S((S((KK)I))(KI)))))$ without the new rule.

Exercise: Rework $\lambda xy.y$, $\lambda x.y$, $\lambda xy.x$, $\lambda x.Mx$ using this rule.

Answer: KI, Ky, K, m.

They needed more optimisations to be practical. So they added operators like $Babc = a(bc)$, $Cabc = (ac)b$ and rules

$$\begin{array}{l} S(Ka)(Kb) \to K(ab) \\ S(Ka)I \to a \\ S(Ka)b \to Bab \\ Sa(Kb) \to Cab \end{array}$$

Shared structure helps a little: we have

$$S(S(KS)(SK(K(SII))))(S(KS)(SK(K(SII)))) = SVV,$$

where $V = S(KS)(SK(K(SII))$, is about half the length again (as stored in memory), but still the expressions are larger than they really ought to be.

Another approach is to define *reaching combinators*:

- $S'pabc \succ p(Sabc) \succ p(ac(bc))$
- $K'pab \succ p(Kab) \succ pa$, and
- $I'pa \succ p(Ia) \succ pa$

It turns out that when we use reaching combinators, we only get linear expression swell, not quadratic.

5.3.2 Supercombinators and Lambda Lifting

Hardware reduction machines never really took off as they were rapidly superceded by new compilation techniques based around *supercombinators* (Hughes 1984). These are combinators that do a lot more work than simple combinators: in essence they are function definitions. For example, the program

```
(defun double (a) (+ a a))
(double 4)
```

We regard `double` as a (super)combinator with a reduction rule

```
(double n) -> (+ n n)
```

Thinking of combinators at this level is a whole lot easier than the low-level S and K, and is much easier to compile and optimise.

Compilation of supercombinators requires a technique known as *lambda lifting*. As a combinator has no free variables, we will have to manipulate code that contains free variables to eliminate them before we compile.

For example, in $H = \lambda x.y$ we want to rewrite this using supercombinators, but y is free inside the lambda and combinators don't have free variables. We can fix this by

$$\lambda x.y =_\beta (\lambda u.(\lambda x.u))y = (\lambda ux.u)y$$

So now we can now write $H = Gy$, where $G = \lambda ux.u$ is a supercombinator.

If $\lambda x.y =_\beta Gy$ appears inside some other lambda, $\lambda z.z(\lambda x.y) =_\beta \lambda z.z(Gy)$, say, we can lift y again

$$\lambda z.z(Gy) =_\beta (\lambda vz.z(Gv))y = Fy,$$

for the supercombinator $F = (\lambda vz.z(Gv))$. Clearly, y can be lifted as far as we need, until it is bound in some λ or we reach the top level of the program.

This idea is used in languages like Haskell and others and has developed into a technique used in compilers of standard languages.

6 Lambda Calculus with Constants

As previously mentioned, this takes the pure (or η) lambda calculus and adds new objects called *constants*. These differ from variables in that they cannot be bound. For example, we could have constants 0, 1, 2, $\ldots$, $+$. Note that we regard $+$ as a constant (after all, it is a constant function in the sense that its defintion never changes!). Another example is when we have constants **true**, **false**, **and**, **not**.

More formally, we could take our definition of the pure lambda calculus and add to it:
A λ-term with constants is defined as follows.

- a constant is a λ-term
- a variable is a λ-term
- if M and N are λ-terms, so is $(M)(N)$. This is called an *application*
- if M is a λ-term and v is a variable, then $\lambda v.(M)$ is a λ-term. This is called an *abstraction*. Here, M is the *body*, while v is the *formal argument*
- nothing else is a λ-term.

For readability, we shall use infix notation such as $1 + 2$ rather than $+1\,2$, though you will occasionally see people using the latter. This actually means, of course, $(+1)2$, or the constant $+$ being applied to the constant 1 which in turn is applied to the constant 2. So this is actually the Curried version of $+$.

Note that the rules for constructing λ-terms do not stop us writing down terms like $2+$ or $+1\,2$ even if we want infix notation.

When given a collection of constants we generally have some ulterior motive lurking about as to what they "really mean". For example, we like to think of 0, 1, and 2, as integers, and $+$ as addition of them. To capture this meaning we have δ-reduction rules. An example could be the rule

$$\underbrace{n+m}_{\text{lambda term}} \succ_\delta \underbrace{n+m}_{\text{integer sum}}$$

for constants n and m. Perhaps this would be easier to interpret if we wrote

$$(+n)m \succ_\delta n + m$$

or even

$$n + m \succ_\delta \text{the sum of } n \text{ and } m$$

The function over the integers is called an *external* function (i.e., external to the calculus), while the δ-reduction is the *internal* form, or its *internalisation*.

$$\underbrace{n+m}_{\text{internal}} \succ_\delta \underbrace{n+m}_{\text{external}}$$

This is actually an infinite number of reduction rules, one for each possible pairing of n and m. This is called a *rule schema*.

More generally, we can have multiple functions, and each need their own δ rule or rule schema. Strictly speaking, each function has a *different* δ, like δ_+, $\delta_\times$ with reductions $\succ_{\delta_+}$, $\succ_{\delta_\times}$ and so on.

Example. Constants **true**, **false**, **and**, **not**. Reductions

$$\begin{array}{rcl}
\textbf{not true} & \succ_\delta & \textbf{false} \\
\textbf{not false} & \succ_\delta & \textbf{true} \\
\textbf{false and false} & \succ_\delta & \textbf{false} \\
\textbf{false and true} & \succ_\delta & \textbf{false} \\
\textbf{true and false} & \succ_\delta & \textbf{false} \\
\textbf{true and true} & \succ_\delta & \textbf{true}
\end{array}$$

Again, "**false and true**" is a notational convenience for "((**and false**) **true**)". Combining with β-reduction we have $\succ_{\beta\delta}$. We find that (Mitschke, 1976)

> Let f be an external function on the constants. Then Church-Rosser holds for $\succ_{\beta\delta_f}$-reduction.

Now we can do all the usual stuff like normal forms, which can be found by normal order reduction when they exist.

More general forms of δ-reduction allow us to apply functions to closed λ-terms as well as constants. An early example of δ-reduction was used by Church:

$$\begin{array}{lll}
\mathcal{C}MN & \succ_{\delta_\mathcal{C}} & \textbf{true} \quad \text{if } M \equiv N \\
\mathcal{C}MN & \succ_{\delta_\mathcal{C}} & \textbf{false} \quad \text{if } M \not\equiv N
\end{array}$$

where M and N are closed λ-terms in normal form, and $\mathcal{C}$ a constant that models a conditional. This only works on closed normal forms, since

$$(\lambda xy.\mathcal{C}xy)II \succ_\beta \mathcal{C}II \succ_{\delta_\mathcal{C}} \textbf{true},$$

while

$$(\lambda xy.\mathcal{C}xy)II \succ_{\delta_\mathcal{C}} (\lambda xy.\textbf{false})II \succ_\beta \textbf{false}.$$

The power of lambda calculus with constants is that (a) we have the computational efficiency of the external function, and (b) we can mix things as we see fit. For example, have integers *and* booleans. Or have arithmetic with errors:

constants: n for each $n \in Z$, $+$, $-$, $*$, $/$, **error**.

δ-schema: as expected for $+$, $-$, $*$, but also

$$
\begin{array}{rcll}
n/m & \succ & \text{integer quotient} & \text{if } m \neq 0 \\
 & \succ & \textbf{error} & \text{if } m = 0 \\
n + \textbf{error} & \succ & \textbf{error} & \\
\textbf{error} + n & \succ & \textbf{error} & \\
n * \textbf{error} & \succ & \textbf{error} & \\
 & \ldots & &
\end{array}
$$

and so on. This all fits together naturally, unlike set theory where adding **error** is somewhat a kludge as it changes the domains and codomains of all our functions.

7 Typed Lambda Calculus

The lambda calculus, although very powerful, does not really reflect the commonly held view of a function as something with a specified domain and range and a rule to get from one to the other. Thus the functions $f : Z \to Z : n \mapsto n$ and $g : Z \to R : n \mapsto n$ are different, even though "they do the same thing". The functions

```
int f(int n)
{
  return n;
}
```

and

```
double g(int n)
{
  return (double)n;
}
```

are really quite different as `g` requires the conversion of an `int` (possibly a 32 bit signed integer) to a `double` (possibly a 64 bit IEEE).

Peculiarities like infinite reductions and functions being applied to themselves seem a bit strange, and do not fit well with ideas of set theory. Further, most modern computer languages are *typed*, i.e., there is a concept of a type of an object and this prevents some of the pure lambda calculus ideas being directly applicable. For example, applying a lambda to itself.

For some computer languages (like C, Java, etc.) the types are in the variables: the type of an object in inferred from the type of the variable that contains it. If you can squeeze an object into an `int` variable, then what you read of the the variable is an `int`. For others (like Lisp) the type is in the object itself and a variable can hold an object of any type.

There are *untyped* languages of course, assembler being one in point. More significantly, there are untyped *high level* languages, BCPL being the most visible example.

The *typed lambda calculus* attempts to regain some of the classical intuition of functions with types.

7.1 Types

Types capture the idea of domains and ranges.

We start with some collection of symbols that we shall call *atomic types*. These symbols are different from the ones we shall be using for variables.

A *type* is defined by

- an atomic type is a type, and
- if α and β are types, then $(\alpha \to \beta)$ is a type, called a *compound type*.

You may like to think of an atomic type as a set, and a compound type $(\alpha \to \beta)$ as the collection of functions from α to β. For example, the set $Z \to Q$ would be the collection of all functions from the integers to the rationals.

Alternatively, as some specific collection of functions, e.g., α and β are groups, and $\alpha \to \beta$ is the collection of all group homomorphisms from α to β; or α and β are vector spaces, and $\alpha \to \beta$ is the collection of all linear maps from α to β.

Or α and β are Booleans (true or false) and $\alpha \to \beta$ is an implication.

One point, to become important later, is that *all types are of finite length*, where α has length 1, $\alpha \to \beta$ length 2, and so on.

As always we drop parentheses:

- $\alpha \to \beta$ means $(\alpha \to \beta)$
- $\alpha \to \beta \to \gamma$ means $(\alpha \to (\beta \to \gamma))$

i.e., associate to the right.

Another common notation for compound types is β^α for $\alpha \to \beta$.

7.2 Typed λs

We now define the syntax of typed λs.

For each type α (atomic or compound) we have infinitely many variables, written v^α. The variable v^α is distinct from v^β if $\alpha \neq \beta$. In fact, to avoid confusion, we won't even use both v^α and v^β in the same equation. Occasionally we will use the notation $x : \alpha$ for x^α.

A typed λ-term is defined as follows

- each v^α is a typed λ-term of type α (atomic or compound)
- if $M^{\alpha\to\beta}$ and N^α are typed λ-terms of types $\alpha \to \beta$ and β respectively, then $(M^{\alpha\to\beta}N^\alpha)^\beta$ is a typed λ-term of type β (application)
- if x^α is a variable of type α, and M^β is a typed λ-term of type β, then $(\lambda x^\alpha.M^\beta)^{\alpha\to\beta}$ is a typed λ-term of type $\alpha \to \beta$ (abstraction)

- nothing else is a typed λ-term.

In the above definitions of typed λ-terms it is important to note that the αs and βs stand for arbitrary types, both atomic and compound. Thus $(x^{(\alpha\to\beta)\to(\gamma\to\delta)}y^{\alpha\to\beta})^{\gamma\to\delta}$ is a valid application. It is slightly confusing to be using, say, α both as an atomic type and as a type variable representing any type, but this is normal usage.

Notice that in the construction of the abstraction MN, the type of M *must* be a compound type $\alpha \to$ something, and the type of B *must* be exactly α. This corresponds to the intuition that a function that takes arguments of type α can only be applied to an object of type α. The form $M^{\alpha\to\beta}N^{\gamma}$ is not a valid typed λ-term if $\alpha \neq \gamma$.

The application rule takes N^{α} (if you like, something in the domain set α) and a function $M^{\alpha\to\beta}$ (one of the functions from $\alpha \to \beta$) and produces an object of type β (something in the range set β).

Similarly, abstraction creates a function of type $\alpha \to \beta$ which takes a value of type α and produces a value of type β.

To avoid things getting too complex, superscripts are often omitted. Other conventions, such as dropping parentheses, are as before.

Example. The identity function on α (any type, atomic or compound)

$$\begin{aligned} I_{\alpha} &= (\lambda x^{\alpha}.x^{\alpha})^{\alpha\to\alpha} \\ &= \lambda x^{\alpha}.x \end{aligned}$$

This has type $I_{\alpha} : \alpha \to \alpha$. I_{α} is *different* from I_{β} when $\alpha \neq \beta$.

Thus we can have $I_{\alpha\to\beta}(\lambda x^{\alpha}.y^{\beta})^{\alpha\to\beta} \succ_{\beta} \lambda x^{\alpha}.y^{\beta}$ Note that if $I_{\alpha}x^{\beta}$ is a valid typed λ-term only if $\alpha = \beta$.

Example.

$$\begin{aligned} K_{\alpha\beta} &= (\lambda x^{\alpha}.(\lambda y^{\beta}.x^{\alpha})^{(\beta\to\alpha)})^{(\alpha\to(\beta\to\alpha))} \\ &= (\lambda x^{\alpha}y^{\beta}.x^{\alpha})^{\alpha\to\beta\to\alpha} \\ &= \lambda x^{\alpha}y^{\beta}.x \end{aligned}$$

when abbreviated. We have $K_{\alpha\beta} : \alpha \to \beta \to \alpha = \alpha \to (\beta \to \alpha)$

Example.

$$S_{\alpha\beta\gamma} = \lambda x^{\alpha\to\beta\to\gamma}y^{\alpha\to\beta}z^{\alpha}.xz(yz)$$

$S_{\alpha\beta\gamma}$ has type $(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma = (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$.

In the abbreviated form we only drop as much annotation as to keep the meaning unambiguous. For example, $\lambda x^{\alpha}.x$ is OK, as we can deduce $\lambda x^{\alpha}.x = \lambda x^{\alpha}.x^{\alpha} = (\lambda x^{\alpha}.x^{\alpha})^{\alpha\to\alpha}$. On the other hand, $\lambda x.x$ gives us no starting point to deduce the type of x, so this is a bad abbreviation.

Similarly, $\lambda x^{\alpha}y^{\beta}.x = \lambda x^{\alpha}y^{\beta}.x^{\alpha} = \lambda x^{\alpha}(\lambda y^{\beta}.x^{\alpha})^{\beta\to\alpha} = (\lambda x^{\alpha}.(\lambda y^{\beta}.x^{\alpha})^{\beta\to\alpha})^{\alpha\to(\beta\to\alpha)}$. On the other hand $\lambda x^{\alpha}y^{\beta}.z$ is not enough.

The process of deducing incompletely specified types is called *type inference*, and is very important in computer languages, where the habit is to leave as much as possible unspecified and let the compiler figure things out.

Given:

- $M^{\alpha\to\beta}$ and N^{α} we can deduce that the application MN has type β
- $(MN^{\alpha})^{\beta}$ we can deduce that $M : \alpha \to \beta$
- $M^{\alpha\to\beta}N$ we can deduce that $N : \alpha$ and $MN : \beta$

- $M^{\alpha\to\beta}N^{\gamma}$ we can deduce that $\alpha = \gamma$
- M^{β} and x^{α} we can deduce the abstraction $\lambda x.M$ has type $\alpha \to \beta$
- $(\lambda x.M)^{\alpha\to\beta}$ we can deduce that $x : \alpha$ and $M : \beta$.

And so on.

Example. We have seen $I_\alpha = \lambda x^{\alpha}.x$. Step by step:

- The xs must match, so $I_\alpha = \lambda x^{\alpha}.x^{\alpha}$
- I_α takes x^{α} and returns x^{α}, so I_α has type $\alpha \to \alpha$, or $I_\alpha = (\lambda x^{\alpha}.x^{\alpha})^{\alpha\to\alpha}$.

A harder example: $S = \lambda xyz.xz(yz)$.

- As a completely unannotated term we need to make an initial assumption. So suppose z has type α
- Then y must be $\alpha \to \beta$ for some β as we have an application yz. So $yz : \beta$
- And x must be $\alpha \to \psi$ for some ψ as we have an application xz. So $xz : \psi$
- But, also, the application $xz(yz)$ tells us that $\psi = \beta \to \gamma$ for some γ. So $x : \alpha \to \beta \to \gamma$ and $xz(yz) : \gamma$

Thus $S = \lambda x^{\alpha\to\beta\to\gamma}y^{\alpha\to\beta}z^{\alpha}.xz(yz)$, and $S : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$.

7.3 Alpha Renaming

Just as before, but with one wrinkle: the destination variable must be of the same type as the source variable. So $\lambda x^{\alpha}.x =_{\alpha} \lambda y^{\alpha}$ is OK, but $\lambda x^{\alpha}.x =_{\alpha} \lambda y^{\beta}$ is not (when $\alpha \neq \beta$).

Here is more opportunity for confusion: the α type is not the same as the α in alpha equality!

7.4 Substitution

This is quite straightforward, and is defined in pretty much the same way as before. The only difference is the type of the substitution must match that of the variable:

$$[N^{\alpha}/x^{\alpha}](M^{\beta})$$

The result is of type β.

If the type of N differs from the type of x, then $[N/x]M$ is not defined. This never arises in practice, as we can't even construct a λ-term with a mis-typed redex.

7.5 Reduction

We can now define *redex*, *contractum* and *reduction* as before. Similarly, for β normal forms.

It can be shown that if $M^{\alpha} \succ_{\beta} N^{\gamma}$, then $\alpha = \gamma$, so that reduction does not change the type of a λ-term.

7.6 Typed Church-Rosser and Normal Forms

Church-Rosser still holds, so we still have unique normal forms. But better still, we find that

> in the typed lambda calculus there are no infinite β reductions.

This is the *strong normalisation theorem for typed λ-terms*.

So what about Ω? Look at $\omega = \lambda x.xx$. What, possibly, could be the type of ω? Suppose x has type α, so $\omega = \lambda x^\alpha.x^\alpha x^\alpha$. But the application xx tells us $\alpha = \alpha \to \beta$ for some β, as x is a function that takes an argument of type α. But then $\alpha = \alpha \to \beta = (\alpha \to \beta) \to \beta = ((\alpha \to \beta) \to \beta) \to \beta = \ldots$, which is not a valid type. The conclusion we are forced to make is: *ω is not a valid typed λ-term*. Thus the typed λ calculus simply outlaws ω, and therefore also Ω.

A corollary of the SNT and Church-Rosser is the *normalisation theorem*:

> every typed term has a unique normal form.

To imagine why the strong normalisation theorem might be true think of the types involved in a reduction:

$$M^{\alpha\to\beta}N^\alpha \succ_\beta P^\beta$$

Every reduction reduces the length of a compound type, e.g., $\alpha \to \beta$ becomes β, and a term can't have an infinite length compound type or an infinite number of finite length compound types by construction of typed λ-terms. So eventually we must stop reducing.

We can define an $=_\beta$ as before, and now we have an *algorithm* that will determine equality of terms: simply reduce both terms until we can go no further and see if the normal forms are the same. There is no non-termination to worry about.

And what's more, it doesn't matter if we use applicative or normal order reductions!

7.7 Church Numerals

Typed Church numerals are possible. For each type α we can define $\overline{n}_\alpha = \lambda x^{\alpha\to\alpha}y^\alpha.x^n y$, a term of type $(\alpha \to \alpha) \to \alpha \to \alpha = (\alpha \to \alpha) \to (\alpha \to \alpha)$. Then we can define (for each α) addition, multiplication and so on as before. Notice that we can't mix numerals of different types. Also, in $\overline{n}MN \succ M^n N$ we must have matching types for $\overline{n}$ and M, namely $\overline{n}_\alpha M^{\alpha\to\alpha}N^\alpha$.

7.8 Typed Combinators

The idea of types translates directly to combinators. For each α, β and γ define operators

$$K_{\alpha\beta} : \alpha \to (\beta \to \alpha) \qquad S_{\alpha\beta\gamma} : (\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$$

or

$$K_{\alpha\beta} : \alpha \to \beta \to \alpha \qquad S_{\alpha\beta\gamma} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

without the extra parentheses.

Notice that these are separate operators for each combination of α, β and γ, so we will have an infinite number of operators (one per type, atomic or compound). Reduction rules are

$$K_{\alpha\beta}MN \succ M \qquad S_{\alpha\beta\gamma}MNL \succ ML(NL)$$

where M, N and L are of appropriate types.

Exercise: Write these out in full with all their types.

Answer:

$$K_{\alpha\beta}^{\alpha\to\beta\to\alpha}M^\alpha N^\beta = M^\alpha$$
$$S_{\alpha\beta\gamma}^{(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}M^{\alpha\to\beta\to\gamma}N^{\alpha\to\beta}L^\alpha = M^{\alpha\to\beta\to\gamma}L^\alpha(N^{\alpha\to\beta}L^\alpha)$$

Just as for the lambda calculus, types restrict what combinators can do. For example, although KI is a simple untyped combinator with a reduction $KIx = I$, it is much more complicated when typed. In the application $K_{\alpha\beta}^{\alpha\to\beta\to\alpha}I_\gamma^{\gamma\to\gamma}$, we must have $\alpha = \gamma \to \gamma$ for the types to match. So this is actually

$$K^{(\gamma\to\gamma)\to\beta\to\gamma\to\gamma}I^{\gamma\to\gamma} : \beta \to \gamma \to \gamma$$

and this reduces as

$$K^{(\gamma\to\gamma)\to\beta\to\gamma\to\gamma}I^{\gamma\to\gamma}x^\beta = I^{\gamma\to\gamma}$$

Now, even though x^β is not "used" in the above, it *must* be of type β.

Notice that, as statements in logic, $\alpha \Rightarrow (\beta \Rightarrow \alpha)$ and $(\alpha \Rightarrow (\beta \Rightarrow \gamma)) \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \gamma))$ are tautologies, that is, for every possible true or false value of α and β the implictions are true:

α	β	$\beta \Rightarrow \alpha$	$\alpha \Rightarrow (\beta \Rightarrow \alpha)$
F	F	T	T
F	T	F	T
T	F	T	T
T	T	T	T

This is not a coincidence! There is a strong connection between lambda calculus and logic: the Curry-Howard isomorphism says that every valid proof of a theorem in logic corresponds to a λ-term reduction, and vice versa. Thus there is a link between logic and computation.

There is a closed λ-term with a particular type only if the type corresponds to a theorem of logic. So $\alpha \Rightarrow \beta \Rightarrow \alpha$ is a theorem, corresponding to K. On the other hand there is no closed λ-term of type $\alpha \to \beta$ as $\alpha \Rightarrow \beta$ is *not* a theorem. (Note that, conversely, given a theorem there is not necessarily a λ-term of that type, e.g., $((\alpha \Rightarrow \beta) \Rightarrow \alpha) \Rightarrow \alpha$ is a theorem, but there is no corresponding λ-term.)

We know that from $\alpha \Rightarrow \beta$ and α we can deduce β: this is just the reduction $M^{\alpha\to\beta}N^\alpha \succ P^\beta$!

7.9 Typed Lambda Calculus with Constants

We can give types to constants, too. For example, we can have some integer constants 0^Z, 1^Z, etc., and boolean constants $\mathbf{true}^B$ and $\mathbf{false}^B$. Types of δ-rules could be

$$\begin{array}{cc} \mathbf{not}^{B\to B} & \mathbf{and}^{B\to B\to B} \\ +^{Z\to Z\to Z} & *^{Z\to Z\to Z} \\ \mathbf{error}^{Z} & \mathbf{equal}^{Z\to Z\to B} \end{array}$$

And so on.

7.10 Polymorphic Lambda Calculus

Also called *second order typed* lambda calculus. This promotes types to first-class status, and allows them to be bound in λs.

The identity function I_α is defined

$$I_\alpha = \lambda x^\alpha .x$$

with type $\alpha \to \alpha$. Now, as always, $I_\alpha \neq I_\beta$ when $\alpha \neq \beta$, but there is no essential difference between all the Is: they are all an identity. The problem is that the type system is forcing us to define a separate I_α for each α.

Compare this with a similar situation in C (Note! The following is actually *overloading*, but we shall momentarily blur over this for the sake of an example)

```
int idint(int v)
{
  return v;
}

double iddouble(double v)
{
  return v;
}

...

n = idint(1);
x = iddouble(1.0);
```

There is no essential difference between `idint` and `iddouble`, but C's type system makes us write out the same function twice, and with different names. In C++ there is some help in this direction, as we can write

```
int id(int v)
{
  return v;
}

double id(double v)
{
```

```
  return v;
}
...

n = id(1);
x = id(1.0);
```

and the compiler can work out from the context by type inference which `id` we mean when we use it. But we still are writing the same code twice. So C++ introduced the idea of *templates*

```
template <class T>
T id(T v)
{
  return v;
}
...

n = id(1);
x = id(1.0);
```

Now `T` is a type variable (just like we have been using α) and the C++ compiler again works out what we mean from the context.

Java has a related thing called *generic types*, but with the usual kludges. Thus `List<Integer>` is OK, but `List<int>` is not.

Of course, Lisp programmers don't even realise there is an issue here and write

```
(defun id (v) v)
```

This last function is *polymorphic*. A polymorphic function doesn't care about the type of its arguments but does the same thing regardless. Thus, the `length` function is polymorphic for lists as it does not care what the types of the objects in the list are. Similarly, `cons` does not care what it is making a pair of.

The opposite of polymorphic is *monomorphic*.

Note that $+$ is *not* polymorphic even though we write $x + y$ regardless of the types of x and y. This is because if we were to look inside the definition of $+$, the code to add two integers (say as 32 bit representations) is quite different from the code to add two floating point numbers (say as 64 bit representations). This is called *overloading*. For example, the (non-template) `id` in the C++ above is actually overloaded as we could have different code bodies for each type. You can't properly do polymorphism in C/C++/Java without breaking the type system.

On the other hand, polymorphic functions just don't care about their arguments.

Back to lambda calculus. We would like our I_α to be polymorphic, but the standard typed lambda calculus won't let us. The *second order typed lambda calculus* introduces polymorphism.

We may define

$$I = \Lambda t.\lambda x^t.x$$

where t ranges over all possible types. The capital Λ indicates a type variable, while the small λ is for the usual variables. And now

$$Ix^\alpha \succ_\beta x^\alpha$$

whatever the α. The type of I is written

$$\forall t.t \to t$$

namely $t \to t$ for all types t.

We could at this point define the syntax for polymorphic typed lambda calculus, but instead we shall just use our intuition. In particular, in the application

$$(\Lambda t.\lambda x^t.x)M^\alpha$$

we see that we shall have to do substitutions $[\alpha/t]$ and $[M/x]$.

A polymorphic function can be *instantiated* at a type:

$$I[\alpha] = \lambda x^\alpha.x$$

and $I[\alpha] : \alpha \to \alpha$ is a monomorphic typed λ-term.

Example.

$$T_\alpha = \lambda f^{\alpha\to\alpha} x^\alpha.f(fx)$$

T_α has type $(\alpha \to \alpha) \to \alpha \to \alpha$, and takes a function f of type $\alpha \to \alpha$ and returns the function that is f applied twice. We can define a polymorphic

$$T = \Lambda t.\lambda f^{t\to t}.\lambda x^t.f(fx)$$

and t is a type variable, while f and x are normal variables. The type of T is

$$T : \forall t.(t \to t) \to t \to t$$

Example. We can define polymorphic Church numerals

$$\overline{n} = \Lambda t.\lambda x^{t\to t} y^t.x^n y$$

with type $\forall t.(t \to t) \to t \to t$, and polymorphic operators to add, multiply and so on. This is a more natural way to define integers than $\overline{n}_\alpha$ as there is only one kind of integer (not one per type), and the polymorphic $\overline{n}$ can be applied to any function

$$\overline{n} M^{\alpha\to\alpha} N^\alpha \succ M^n N$$

Example. Polymorphic combinators. We have seen I already.

$$K = \Lambda st.\lambda x^s y^t.x$$

with type $\forall st.s \to t \to s$, and

$$S = \Lambda rst.\lambda x^{r\to s\to t} y^{r\to s} z^r.xz(yz)$$

with type $\forall rst.(r \to s \to t) \to (r \to s) \to r \to t$.

Thus polymorphism regains us some of the flexibility that the untyped lambda calculus has.

The notion of polymorphism is widely used in computer languages as a means to reduce the amount of code you you have to write. An additional benefit is your compiled program should be smaller, too. In certain languages, e.g., Lisp, it happens naturally. In some, such as C and Pascal, it is either weakly or not supported. In others, such as C++ there are curious syntax wranglings to make it appear possible.

In fact, C++ cheats and uses overloading. If a C++ compiler comes across a definition like `id` above, and then a use `id(1)`, it effectively writes out the code

```
int id(int v)
{
  return v;
}
```

and compiles and uses that. Each time `id` is used on a new type it writes out the whole function again with the appropriate types inserted. This means that there is no code sharing going on at all. The programmer does get the benefit of writing the source just once, though.

7.11 Conclusion

We do lose some things, of course. We can't apply a function to itself: think of the type that such a function should have. Thus we lose things like the Y operator and guaranteed fixed points. Recursive functions are generally not possible: a recursive program could loop forever, but we can't have non-terminating reductions.

There are many extensions, for example *product types*. Given types α and β we can form a new type with name $\alpha \times \beta$ (just as previously given types α and β we formed a new type with name $\alpha \to \beta$) together with projection and pairing operations. This is the natural way of thinking about pairs of objects. For example $(2, 3.4)$ has type $\texttt{int} \times \texttt{double}$. A function of two objects can be expressed as a function of one pair: is $f(x, y)$ a function of both x and y or of the single (x, y)? Product types are available in C/C++ using `struct`:

```
struct prod {
  int a;
  double b;
}
```

and Java using `class`

```
class prod {
  int a;
  double b;
}
```

Currying can now be expressed using product types: $(\alpha \times \beta) \to \gamma$ vs $\alpha \to \beta \to \gamma$, or in a rather more suggestive notation: $\gamma^{\alpha \times \beta}$ vs $(\gamma^{\beta})^{\alpha}$.

There are also *sum* types, $\alpha + \beta$. This is something that is an α *or* a β (but not both). C/C++ has sum types using `union`

```
union sum {
  int a;
  double b;
}
```

An element of this type can store an `int` *or* a `double`, but not both at the same time.

Much fun can be had deciding whether equations like $\alpha \times (\beta + \gamma) \cong (\alpha \times \beta) + (\alpha \times \gamma)$ hold in your system for all types α, β and γ.

For example, is the type

```
struct T1 {
  char a;
  union {
    int b;
    double c;
  } u;
};
```

the "same" as the type

```
union T2 {
  struct {
    char a;
    int b;
  } s1;
  struct {
    char a;
    double c;
  } s2;
};
```

?

Of course, the answer depends on what you mean by "the same", i.e., what properties of the types you are considering.

Polymorphism is not the end of the story. Consider the function `if`. It certainly has type $B \to$ something, so why not regard it as polymorphic with type $\forall t.B \to t$? Well, consider the perfectly valid Lisp

```
(if (foo) 1 1.0)
```

This does not return an object of a fixed type. Sometimes it returns an `int`, sometimes a `double` and so is not of the type $\forall t.B \to t$. One solution (as used in ML and Haskell, below) requires the two branches of the `if` to be of the same type

```
if foo() then 1 else 1.0
```

would be *invalid*, uncompilable code. Perhaps sum types can be employed: `if` $: \forall s, t.B \to s + t$, but this gets unwieldy very quickly. The other solution, as adopted by Lisp, is beyond polymorphism.

The hierarchy does not stop with second order types as there are higher order: we can have *classes* of types, for example we can have `Num` which contains `char`, `int` and `double`. Then we can talk about things *for all numeric types*.

8 Functional Programming

Some people were unhappy with procedural languages since they are so hard to understand and compile. The presence of side effects means that certain optimisations cannot be made and compilers can't produce perfect code.

These people argued that if you disallowed side effects you could get a faster running program. Also, because of referential transparency, programs would be easier to write and debug though natural use of modularity. Thus was born the functional style.

The functional style of programming is one that emphasises the evaluation of expressions rather than the execution of commands. A functional programming language is one that supports such a style.

You can program, more or less, in a functional style whatever the language, but some languages actively help you. For example, higher order functions are pretty much necessary to program in a functional style.

This contrasts the C

```
sum = 0;
for (i = 0; i < 10; i++)
  sum += i;
```

with, say,

```
(reduce + (intlist 0 10))
```

where `intlist` would be a function that produces a list of integers, and `reduce` a function that repeatedly applies a function to a list.

Outline implementations (not complete!):

```
(defun intlist (a b)
   (if (< a b)
       (cons a (intlist (+ a 1) b))
       ()))

(defun reduce (f l)
  (if (= (length l) 1)
      (car l)
      (f (car l) (reduce f (cdr l)))))
```

Notice the compactness of the functional version, and also its use of higher order functions (`reduce`). The *pure* functional programming style also rejects the use of assignments, like `setq` or = in C. This is for referential transparency: when you see a variable (in a given block, or, in lambda calculus terms, within the scope of a lambda) it always has the same value.

In the functional style, variables do not vary!

We note at this point the difference between *binding* and *assignment*. Assignment is the traditional way of updating the value of a variable

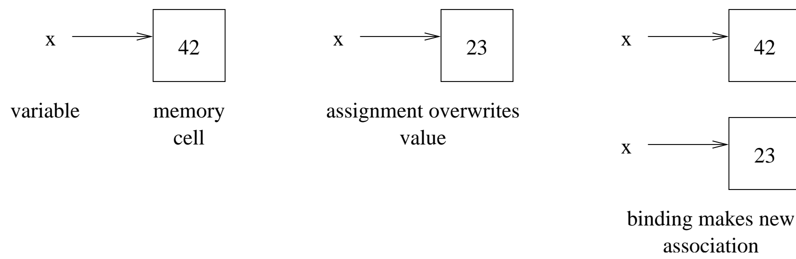


Figure 5: Assignment and Binding

```
(setq x 3)
```

and does not have a lambda calculus equivalent. On the other hand, binding is the direct equivalent of a bound variable in a λ:

```
(let ((x 3))
   ...)
```

The binding has the limited scope of the body, and the variable `x` is independent of any other variables presently named `x` (when using lexical scope).

In a C-like language this becomes:

```
x = 3
```

for assignment and

```
{
  int x = 3;

   ...
}
```

for binding. The syntax in C tends to obscure the difference.

In assignment, an existing memory location is updated. In binding a new memory location is associated with the variable name and it is initialised with a value: nothing is overwritten. This is the essence: assignment destroys the old value, while binding is non-destructive. With binding the old variable-value association can be retrieved.

Notice the relationship between `let` and `lambda`

```
((lambda (x)
   ...)
 3)
```

does exactly the same as the `let` above. In fact, some Lisps implement `let` in just this way!

It may seem weird to outlaw `setq`, but if you think truly in the functional style, you discover that you rarely need it. There are occasions when you do, of course, but these are usually due to efficiency concerns, or it makes your program just *too* contorted to avoid it.

Sometimes it is good to use `goto`, too!

Of course, Lisp was the first language that was extensively used to explore functional ideas, though other specifically functional language sprang up, e.g., ML, Miranda, Haskell.

The Haskell designers also wanted to support normal order reduction, but we know that isn't very efficient. So instead they use *lazy reduction*, also known as *call by need*. This is like normal order in that you do not evaluate arguments, but if we *do* evaluate an argument we remember the result and reuse it. In the absence of side effects this is identical to normal order reduction.

The opposite of lazy is *eager*. Closely related terms are *strict*, for languages that always evaluate arguments; and *non-strict* for those that don't.

Example. Suppose we had a Lisp with lazy evaluation. Then

```
(defun hi ()
  (print "hello")
  7)

(hi) ->
hello
7

(defun lazy (a b)
  a)

(lazy 9 (hi)) ->
9
```

The call to `hi` is not expanded.

```
(defun lazy2 (a)
  (list a a))

(lazy2 (hi)) ->
hello
(7 7)
```

The call to `hi` is expanded just once. In a pure normal order reduction the `(hi)` would be evaluated twice in the expansion `(list (hi) (hi))` and `hello` would be printed twice.

In

```
(defun loop () (loop))

(defun foo (n) 42)
```

The function call `(foo (loop))` *will* return 42 even though `loop` would never return if we called it. This is a direct equivalent of $(\lambda x.y)\Omega$.

Exercise: (difficult) Write down a Lisp expression that prints `applicative` if it is evaluated applicatively, `lazy` if evaluated lazily, and `normal` if evaluated in normal order.

Answer:

```
(setq n 0)

(defun inc ()
  (setq n (+ n 1))
  n)

(defun foo (a)
   (setq n 10)
   (if (= a 11)
       (if (= a a) "lazy" "normal")
       "applicative"))

(foo (inc))
```

The idea behind lazy evaluation is that you only execute those expressions that are actually needed. If an expression is never needed, you don't waste time executing it. You get the efficiency of single evaluation you get from applicative order, but you get the semantics of normal order.

In practice

- the overhead of lazy evaluation (carrying around expressions, etc.) is pretty high
- the overhead of checking an expression to see if it has already been evaluated is non-trivial
- most of the time you *do* require the value of most expressions so you don't gain as much as you might think from less re-evaluation.

So lazy evaluation does not gain you much in terms of speed. It does give you the semantics of normal order reduction, though.

It does allow you to program *infinite data structures* (note: this program won't work in normal Lisps!)

```
(defun intlist (n)
   (cons n (intlist (+ n 1))))

(setq l (intlist 0))

(car (cdr l)) ->
1
```

This is because the call to `intlist` in `(setq l (intlist 0))` is not evaluated until needed. When we do `(cdr l)` this expands into `(cdr (cons 0 (intlist 1)))` which evaluates to `(intlist 1)`. And then `car` of that expands again to `(car (cons 1 (intlist 2)))` which gives us 1.

Thus `(intlist 0)` has all the properties of the infinite list

```
(0 1 2 3 4 5 6 ... )
```

without, of course, taking up an infinite amount of storage.

8.1 ML

Roughly speaking, ML (for *meta language*) is a typed version of Lisp, though with a decidedly harder syntax. Similarly, it uses eager evaluation. You can run ML on the BUCS machines by typing `sml`

```
midge:1 % sml

Edinburgh Standard ML (core language)                (C) Edinburgh University


- 1+2;
> 3 : int
- ^D
ML exit
```

As with Lisp, this is a read-eval-print loop: type an expression, it reads, evaluates and prints the result. The `-` is a prompt, while the > introduces the result.

A very brief introduction.

- Variables

  ```
  - val x = 5;
  > val x = 5 : int
  - x;
  > 5 : int
  ```

  Everything gets annotated with its type.

- Functions

  ```
  - fun f x = 2*x;
  > val f = fn : int -> int
  ```

  Note that ML has inferred the type of `f` from the `2*x`. Here `x` must be of the same type as `2`, namely `int`. And then `f` returns an `int`.

  Arguments must have the right type

  ```
  - f(1.0);
  Type clash  in:  (f 1.0)
  Looking  for a:  int
  I have found a:  real
  ```

- Local Variables

  ```
  - let val y = x+1 in 2*y+3
  = end;
  > 15 : int
  ```

  The = is a continuation prompt.

- Type Inference

```
- fun add x y = x+y;
Type checking error in: (syntactic context unknown)
Unresolvable overloaded identifier:  +
Definition cannot be found for the type:  ('a * 'a) -> 'a
```

ML cannot work out what type `add` should be. It uses `'a` for α, `'b` for β, etc., as type variables. It can figure out that `x` and `y` must be of the same type, but no more than that.

```
- fun add (x:int) (y:int) = x+y;
> val add = fn : int -> (int -> int)
```

Annotating some types helps ML. We could actually do

```
- fun add (x:int) y =x+y;
> val add = fn : int -> (int -> int)
```

ML works out the type of `y` by type inference.

Notice that `add` is a function of one variable, returning a function of one variable.

```
- add;
> fn : int -> (int -> int)
- add 3;
> fn : int -> int
- add 3 4;
> 7 : int
```

Note the lack of parentheses in the call of `add`.

ML does allow functions of more than one argument

```
- fun add2(x:int,y:int) = x+y;
> val add2 = fn : (int * int) -> int
- add2(1,2);
> 3 : int
```

Actually, it doesn't: the parentheses `()` make a pair of their contents, so `(1,2)` is a single object of type pair of `int` or `int * int`. This is an example of a *product type*.

Note that writing `add(1,2)` is an error: you are passing an object of type `int * int` to `add`, which is a function of type `int -> (int -> int)`

```
- add(2,3);
Type clash  in:  (add (2,3))
Looking  for a:  int
I have found a:  int * int
```

- Lists

```
- [1,2,3];
> [1,2,3] : int list
```

All the elements in a list must be of the same type. We have `hd`, `tl`, `null` (test for `[]`), an infix `::` for cons

```
- 1 :: [2,3];
> [1,2,3] : int list
```

Notice the `list` here is acting like a function on types: it take a type and returns a new type that is lists of that type. This is called a *type constructor*: a fancy name for a function from types to types.

- Lambdas

  ```
  - fn x:int => x+1;
  > fn : int -> int
  ```

  As in Lisp, `fun f x ...` is the same as `val f = fn x => ...`.

- Conditionals

  ```
  - val x = if 1 = 1 then 2 else 3;
  > val x = 2 : int
  ```

  The values in each branch must have the same type

  ```
  - if 1 = 2 then 3 else 1.0;
  Type clash  in:  (if (1 = 2) then 3 else 1.0)
  Looking  for a:  int
  I have found a:  real
  ```

- Booleans `true` and `false`.

  ```
  - true;
  > true : bool
  - not true;
  > false : bool
  ```

- Recursion

  ```
  - fun fact n = if n < 2 then 1 else n*fact(n-1);
  > val fact = fn : int -> int
  - fact 10;
  > 3628800 : int
  ```

  No need for Y. The existence of recursion shows that ML is not a faithful rendition of the typed lambda calculus.

- Polymorphism.

  Functions like `hd`, `tl` and `::` are polymorphic, as they can be used on all list types

  ```
  - hd;
  > fn : ('a list) -> 'a
  ```

  The type of `hd` uses the type variable α. The empty list `[]` has type `'a list`, but it is also possible to have empty lists of any type, which causes some peculiarities:

```
- val nilint = tl [1];
> val nilint = [] : int list
- val nilreal = tl [1.0];
> val nilreal = [] : real list
- nilint = [];
> true : bool
- nilreal = [];
> true : bool
- nilreal = nilint;
Type clash  in:  (nilreal = nilint)
Looking  for a:  real
I have found a:  int
```

Now,

```
- fun id x = x;
> val id = fn : 'a -> 'a
```

defines a polymorphic identity. Further

```
- fun K x y = x;
> val K = fn : 'a -> ('b -> 'a)
```

defines a polymorphic K.

```
- K 7;
> fn : 'a -> int
```

ML always uses α as the first free type, then β, and so on, so we don't get `fn : 'b -> int` as we might have expected.

```
- K 7 8.0;
> 7 : int
```

You may define your own datatypes:

```
- datatype Bool = True | False;
> datatype Bool = False | True
  con True = True : Bool
  con False = False : Bool
- True;
> True : Bool
```

The symbols `True` and `False` are now constants of type `Bool`, and there are no others of type `Bool`. If it were legal code, we might have defined `int` as

```
data int ... | -2 | -1 | 0 | 1 | 2 | ...
```

Our types are just as good as built-in ones:

```
- fun Not b = if b = True then False else True;
> val Not = fn : Bool -> Bool
- Not False;
> True : Bool
```

Currying in ML:

```
- fun curry f = fn x => fn y => f(x, y);
> val curry = fn : (('a * 'b) -> 'c) -> ('a -> ('b -> 'c))
- fun sum(x:int,y:int) = x+y;
> val sum = fn : (int * int) -> int
- val csum = curry sum;
> val csum = fn : int -> (int -> int)
- csum 7;
> fn : int -> int
- csum 7 11;
> 18 : int
```

And back again:

```
- fun uncurry f = fn(x,y) => f x y;
> val uncurry = fn : ('a -> ('b -> 'c)) -> (('a * 'b) -> 'c)
- val add = uncurry csum;
> val add = fn : (int * int) -> int
- add(2,3);
> 5 : int
```

Most of the time we don't need to annotate types as ML can figure them out by *type analysis*, i.e., going through the expression and determining what types everything must be. On occasion, though, it needs a hint as to what we mean. For example, you can't deduce types for x and y in "$x + y$" as the $+$ operator is overloaded and can take arguments of many types. On the other hand, from "$x + 1$" we can deduce x must be an integer since the two arguments of $+$ in ML must be of the same type: there is no automatic coercion.

There's a lot more to ML than this, in particular, definition by pattern matching, abstract types, type constructors and exceptions.

8.2 Haskell

A typed language with a syntax not unlike ML, but with lazy evaluation. As previously mentioned, lazy evaluation is an attempt to combine the best features of both applicative (efficiency) and normal (good semantics) order reductions.

"Haskell" is named after Haskell Curry.

Run `hugs` (*Haskell User's Gofer System*), a Haskell interpreter

```
midge:22 % /u/ma/s/masrjb/Hugs/bin/hugs +t
__   __ __  __  ____   ___      _________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2001
||---||         ___||           World Wide Web: http://haskell.org/hugs
```

```
||    ||                             Report bugs to: hugs-bugs@haskell.org
||    || Version: February 2001      _________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Reading file "/u/ma/s/masrjb/Hugs/hugs/lib/Prelude.hs":

Hugs session for:
/u/ma/s/masrjb/Hugs/hugs/lib/Prelude.hs
Type :? for help
Prelude> 1+2
3 :: Integer
Prelude> :quit
[Leaving Hugs]
```

Hugs requires definitions to be in *modules*. In a file `Egs.hs` put

```
module Egs where
{- this is a comment -}
inc x = x+1
```

and load into Hugs by

```
> :load Egs.hs
```

The 'E' in `module Egs where` must be upper case.

You can reload a module after changing it by

```
> :reload Egs.hs
```

or simply

```
> :reload
```

will reload the last module again. Definitions must be in modules, but we can type expressions to be evaluated at the prompt. Below we shall mix definitions and evaluations, but you must separate them when actually using Hugs.

Haskell does everything ML does and more. Functions are defined by equations

```
inc x = x+1          -- definition in a module
> inc 3              -- typed in at prompt
= 4 :: Integer       -- result
```

which is short for

```
inc = \x -> x+1
```

with `\x` for λx. We can find the type of an object in Hugs by using `:t`

```
> :t inc
= inc :: Num a => a -> a
```

This can be read as $\forall\alpha \in \texttt{Num}.\alpha \to \alpha$, which is to say "for all numerical types ...". Haskell has *classes of types*, which are types of types, i.e., second order types. This is connected with object oriented ideas. The class `Num` contains the types `Integer`, `Float` and `Double` amongst others. These types are also in the class `Ord` of objects that support comparison, i.e., $<$. Use, e.g., `:info Ord` to see details of a class or any other object.

For

```
positive x = if x > 0 then True else False
```

or

```
positive x = x > 0
```

We get

```
> :t positive
= positive :: (Ord a, Num a) => a -> Bool
```

This is type $\forall\alpha \in \texttt{Ord} \cap \texttt{Num}.\alpha \to \texttt{Bool}$. So this works for any numeric type that also has comparison (recall that complex numbers don't have comparison).

This way of defining functions extends:

```
len :: [a] -> Integer
len [] = 0
len (x:xs) = 1 + len xs
```

The first line declares the type of the polymorphic `len`, while the others give the value of `len` of an empty list, and `len` of a cons (Haskell uses infix `:` for cons). This is an example of definition by pattern matching.

```
> :t len
= len :: [a] -> Integer
```

Here `[a]` is "list of α". As in ML, the empty list `[]` has type `[a]`, while `tail [1]` has type `[Integer]`, which is not the same as `tail [1.0]` of type `[Double]`. Sometimes.

Once given a value, a symbol cannot be reassigned (within a module)

```
x = 1
x = 2
ERROR haskell.hs:17 - "x" multiply defined
```

though it can be locally rebound

```
> let x = 1 in 2*x+1
= 3 :: Integer
```

The only way to change an assignment is to edit the module and reload it. This is so Haskell can have referential transparency.

One of the important differences between ML and Haskell is that Haskell is lazy:

```
from n = n : from(n+1)
> :t from
= from :: Num a => a -> [a]
```

This defines `from` as a function returning an infinite list of numbers starting from n.

```
ints = from 0
> :t ints
= ints :: [Integer]
> head ints
= 0 :: Integer
> head(tail ints)
= 1 :: Integer
```

The `map` function acts as in Lisp and ML, taking a function and a list and applying that function to the values in the list:

```
sqs = map (\x -> x*x) ints
> head(tail(tail sqs))
= 4 :: Integer
```

Infinite loops:

```
loopy :: a -> a
loopy n = loopy n
k x y = x
> :t k
= k :: a -> b -> a
```

Names starting with capital letters are special in Haskell. Now,

```
> k 1 (loopy 0)
= 1 :: Integer
```

while

```
> k (loopy 0) 1
=
```

goes into a busy loop. Hit `^C` to interrupt.

Again, there is a huge amount of Haskell we have omitted to describe: modules for structuring programs, *monads* (special structures that facilitate programming kinds of things that are traditionally difficult in pure functional languages, like state and I/O), object orientation and classes of types, and more. It is claimed that some compilers for Haskell produce code that is equal in speed to that from a C program even though you have the power of functional programming. It doesn't seem to be about to replace traditional languages, though.

8.3 SECD and FAM

Rather than make hardware run functional languages natively, SECD, CEK and FAM were attempts to give programming languages that would efficiently implement functional concepts.

These languages are quite low level, and can be though of as *virtual machines*. The idea is that just as von Neumann's 5 box model is a virtual machine for conventional languages to compile to, so should these should play the same role for functional languages. If you can efficiently compile to, say, FAM, and FAM is sufficiently low level that it can be efficiently implemented on a real machine, then your language will run efficiently on a real machine. FAM was designed to implement ML.

As a language, FAM is quite low level like an assembler, as it describes things like stacks, frame pointers and program counters. However, the objects it acts upon are things like closures and functions. Abstractly, the machine state is represented by a 7-tuple

$$(AS, RS, FR, PR, TS, ES, M)$$

for references to the

- argument stack
- return stack
- frame
- program
- trap stack
- environment stack
- memory

Various operations change the values of these in given ways, e.g., push a value on to a stack. Precise semantics are given for say, how to look up the value of a variable in the environment stack. Similarly, rules of how to compile ML statements into FAM are given.

SECD (*Store, Environment, Control, Dump*) was one of the first virtual machine designs (1964). The four variables S, E, C and D describe the state of the virtual machine. CEK (*Control, Environment, Kontinuation*)was designed in the 1980s as upgrade of SECD. It included *continuations*, that is an explicit notion of program control which unifies all the ideas of jumps, loops, recursive functions and so on.

See `http://www.cs.nott.ac.uk/~gmh/faq.html` for more information on functional languages.

9 Related Stuff

- type theory
- rewriting
- process calculi: CSP, CCS, π-calculus, ambient calculus. Where the subject of discourse is the interaction between processes
- logic
- category theory
- foundations