The Lambda Calculus

A brief history of mathematical notation.
Our notation for numbers was introduced in the Western World in the Renaissance (around 1200) by people like Fibonacci. It is characterised by a small fixed set of digits, whose value varies with their position in a number. This place-value system was adopted from the Arabs who themselves credit the Indians. We do not know when and where in India it was invented. A notation for expressions and equations was not available until the 17th century, when Francois Viète started to make systematic use of placeholders for parameters and abbreviations for the arithmetic operations. Until then, a simple expression such as $3x^2$ had to be described by spelling out the actual computations which are necessary to obtain $3x^2$ from a value for x. It took another 250 years before Alonzo Church developed a notation for arbitrary functions. His notation is called λ-calculus ("lambda calculus"). Church introduced his formalism to give a functional foundation for mathematics but in the end mathematicians preferred (axiomatic) set theory. The λ-calculus was rediscovered as a versatile tool in computer science by people like McCarthy, Strachey, Landin, and Scott in the 1960s.
[Apparently, someone called Mervyn Pragnell appeared from nowhere and persuaded several fledgling computer scientists, among them Rod Burstall, Peter Landin and Christopher Strachey, to take an interest in lambda calculus by inviting him to his home, giving them a chapter each on Church's work. They came back the following week to discuss this work, and the rest is history.]
Incidentally, the history of programming languages mirrors that of mathematical notation, albeit in a time-condensed fashion. In the early days (1936-1950), computer engineers struggled with number representation and tried many different schemes, before the modern standard of 2-complement for integers and floating point for reals was generally adopted. Viète's notation for expressions was the main innovation in FORTRAN, the world's first high-level programming language (Backus 1953), thus liberating the programmer from writing out tedious sequences of assembly instructions. Not too long after this, 1960, McCarthy came out with his list processing language Lisp. McCarthy knew of the λ-calculus, and his language closely resembles it. This has led to the functional programming languages ML, Haskell and F$\sharp$.

Slide 1

The Lambda Calculus

Slide 2

Notions of Computability

- Church (1936): **λ-calculus**
- Turing (1936): Turing machines.

Turing showed that the two very different approaches determine the same class of computable functions. Hence:

Church-Turing Thesis. Every algorithm [in the intuitive sense of Lecture 1] can be realised as a Turing machine.

The λ-calculus notation can appear archaic. It is just notation! In mathematics, we write the function $f(x) = x + x$ to mean the function, called f, which gives back the answer $x + x$ for every x. The term $f(2)$, for a specific value of x, gives back the specific result 4.

Slide 3

Haskell Function

```
add :: Int → Int
add x = x+x
  ...
  > add 2       apply add to 2
        4       result
```

Slide 4

Another Haskell Function

```
h :: Int → Int → Int

        h x y =  g y
                 where  g z = x + z
```

> h 2 3	apply h to 2 and 3
g 3 where g z = 2 + z	apply g to 3
2 + 3	addition
5	result

In λ-notation, we have special notation which dispenses with the need to give a name to a function (as in f or add) and easily scales up to more complicated function definitions. We write $\lambda x. x + x$, or $\lambda x. (x + x)$ to be completely clear, for the function which takes in one argument x and returns the result $x + x$. The Greek letter λ ('lambda') has a role similar to the keyword 'function' in some programming languages. It alerts the reader to the fact that the variable which follows is not part of the term but a *formal parameter* of the function definition. The dot after the formal parameter introduces the function body.

A function which has been written in λ-notation can itself be used in an expression. For example, the application of the function $\lambda x. x + x$ to the argument 2 is written $(\lambda x. x + x)\, 2$. Although it is not strictly necessary, it will be convenient sometimes to introduce abbreviations for λ-terms. We write them in the same way that we always do in mathematics, using the equality symbol. So, we can abbreviate our function $\lambda x. x + x$ using a capital letter, say F:

$$F \overset{\textbf{def}}{=} \lambda x. x + x$$

We can then write $F2$ instead of $(\lambda x. x + x)2$.

Now suppose the body of a function consists of another function: for example,

$$H \stackrel{\text{def}}{=} \lambda x.(\lambda z.\, x + z)$$

We can apply this function to the value $\mathbf{2}$, written $H\mathbf{2}$, to obtain another function $\lambda z.\ \mathbf{2} + z$. However, we can also consider it as a function of *two* arguments, where we get a number back if we supply H with *two* arguments: the result of $(H\mathbf{2})\mathbf{3}$ is the answer $\mathbf{5}$. Both views are legitimate and completely consistent with each other. If we want to stress the first interpretation we can write the term with the brackets as above. If we want to see it as a function of two arguments then we can leave out the brackets:

$$\lambda x.\lambda z.\ x + z$$

Sometimes, we even lazily elide the second lambda:

$$\lambda xz.\ x + z$$

Likewise, in the application of H to arguments $\mathbf{2}$ and $\mathbf{3}$, we can use brackets to stress that $\mathbf{2}$ is to be used first, $(H\mathbf{2})\mathbf{3}$, or we can suggest simultaneaous application by the notation $H\,\mathbf{2}\,\mathbf{3}$. Whatever the intuition about H, the result will be the same.

We have already mentioned that, in a good definition of algorithm, an algorithm must be able to take an algorithm as input. With register machines and Turing machines, this is achieved by coding programs as numbers. With the lambda calculus, a function can directly have a function as input. For example, consider the λ-term

$$(\lambda z.\, z\,\mathbf{3})(\lambda y.\, \mathbf{2} + y)$$

The function $\lambda z.\, z\,\mathbf{3}$ take the function $\lambda y.\, \mathbf{2} + y$ as input, and returns the term $(\lambda y.\, \mathbf{2} + y)\mathbf{3}$, which can then be evaluated to $\mathbf{5}$.

Slide 5

λ-terms

$$\begin{aligned} F &\overset{\text{def}}{=} \lambda x.\, x + x \\ F\,2 &= (\lambda x.\, x + x)2 \rightarrow_\beta 2 + 2 \\ H &\overset{\text{def}}{=} \lambda x.(\lambda z.\, x + z) \\ H\,2\,3 &= (\lambda xz.\, x + z)2\,3 \rightarrow_\beta (\lambda z.\, 2 + z)3 \rightarrow_\beta 2 + 3 \end{aligned}$$

Higher-order funtion:

$$(\lambda z.\, z\,3)(\lambda y.\, 2 + y) \rightarrow_\beta (\lambda y.\, 2 + y)3 \rightarrow_\beta 2 + 3$$

For this course, we will work with the pure λ-calculus without constants (for example, $+$ is a constant): function formation and function application is all there is. We will sometimes use λ-terms with constants as intuitive examples. When developing a programming language based on the λ-calculus, we add many programming constructs including numbers and types, as you will have seen in the Haskell course. The point is that the pure λ-calculus (with or without constants) is as expressive as Turing machines, and so this is all we need!

Slide 6

Definition of λ-terms

The **λ-terms** are constructed, from a given, countable set of variables $x, y, z, \ldots \in Var$, by:

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

The term $\lambda x.M$ is called a **λ-abstraction** and MM an **application**.

With this definition, it is essential to use parenthesis to disambiguate.

This definition of λ-terms is ambiguous. The term $\lambda x.\, xy$ can be parsed as $\lambda x.\, (xy)$ or $(\lambda x.\, x)y$ In fact, we will establish a convention that $\lambda x.\, xy$ means $\lambda x.\, (xy)$, and use brackets to force the other interpretation.

Slide 7

Examples

$$\lambda x.\,(xy)$$
$$(\lambda x.\,x)y$$
$$\lambda x.\,(\lambda y.\,x)$$
$$\lambda x.\,(\lambda y.\,(\lambda z.\,(xz)(yz)))$$
$$(\lambda x.\,(xx))(\lambda x.\,(xx))$$
$$\lambda x.\,(x(\lambda y.\,(yx)))$$

Slide 8

Notation

- $\lambda x_1\, x_2 \ldots x_n.M$ means $\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.M) \ldots)$
- $MM_1\, M_2 \ldots M_n$ means $(\ldots(((M\, M_1)\, M_2)M_3)\ldots)M_n$: that is, application is left-associative
- Drop parentheses enclosing the body of a λ-abstraction.

Slide 9

Examples, simplified

$$\lambda x.\, xy$$
$$(\lambda x.\, x)y$$
$$\lambda xy.\, x$$
$$\lambda xyz.\, (xz)(yz)$$
$$(\lambda x.\, xx)(\lambda x.\, xx)$$
$$\lambda x.\, x(\lambda y.\, yx)$$

Computing λ-terms

λ-terms on their own would be a bit boring if we did not know how to *compute* with them as well. There is only one rule of computation, called β-*reduction*, and it concerns the replacement of the formal parameter by an actual argument. It can only occur if a λ-abstraction has been applied to some other term.

Slide 10

Computing λ-terms

$$(\lambda x.\, x + x)2 \rightarrow_\beta 2 + 2$$

$$(\lambda xz.\, x + z)2\,3 \rightarrow_\beta (\lambda z.\, 2 + z)3 \rightarrow_\beta 2 + 3$$

$$(\lambda z.\, z\,3)(\lambda y.\, 2 + y) \rightarrow_\beta (\lambda y.\, 2 + y)3 \rightarrow_\beta 2 + 3$$

Slide 11

Simplified β-reduction

β-reduction is a binary relation between λ-terms. These rules are instances of β-reduction for λ-terms satisfying **Barendregt's variable condition**. See later for the full definition.

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

$$\frac{M \rightarrow_\beta M'}{M\,N \rightarrow_\beta M'\,N} \qquad \frac{N \rightarrow_\beta N'}{M\,N \rightarrow_\beta M\,N'}$$

We omit the subscript β when the meaning is clear.

Here are some examples of β-reduction:

1. $(\lambda x.x)y \to y$
2. $(\lambda yz.z)u \to \lambda z.z$
3. $(\lambda x.xy)(\lambda x.x) \to (\lambda x.x)y \to y$
4. $(\lambda y.(\lambda xz.x)y)w \to (\lambda xz.x)w \to \lambda z.w$
5. $(\lambda x.xx)(\lambda x.xx) \to (\lambda x.xx)(\lambda x.xx) \to \ldots$

We see that reduction is nothing other than the textual replacement of a formal parameter in the body of a function by the actual parameter supplied. However, we need to do quite a bit of work to formalise what we mean by this. Consider the function $f(x) = x + x$: the xs in the expression $x + x$ are considered *free*; in $f(x) = x + x$, the x in the argument to the function f is a formal parameter that *binds* the xs in $x + x$; and the xs to the right of the equality $f(x) = x + x$ are considered *bound* by the formal parameter. From your mathematical and programming experience, you will understand these concepts intuitively, although you may not have named them before. Analogous definitions can be given for the λ-terms.

Slide 12

Free and bound variables

In $\lambda x.M$, we call x the **bound variable** and M the **body** of the λ-abstraction. An occurrence of x in a λ-term M is called

- a **binding** occurrence if x is in between λ and **.**
  (e.g. $(\lambda \mathbf{x}.y\, x)\, x$)
- **bound** if in the body of a binding occurrence of x
  (e.g. $(\lambda x.y\, \mathbf{x})\, x$)
- **free** if neither binding nor bound
  (e.g. $(\lambda x.y\, x)\mathbf{x}$).

Slide 13

Haskell Function: binding

```
h :: Int → Int → Int

          h x y =  g y
                   where g z = x + z
```

x, y binds everywhere under h.

z binds only in the where clause

Slide 14

Haskell Function: bound

$\mathtt{h} :: \mathtt{Int} \longrightarrow \mathtt{Int} \longrightarrow \mathtt{Int}$

$$\begin{aligned} \mathtt{h}\, x\, y = {} & \mathtt{g}\, y \\ & \mathtt{where}\ \ \mathtt{g}\, z = x + z \end{aligned}$$

x, y are bound everywhere under h.

z is bound only in the where clause

Slide 15

Haskell Function: free

$\mathtt{h} :: \mathtt{Int} \longrightarrow \mathtt{Int} \longrightarrow \mathtt{Int}$

$$\begin{aligned} \mathtt{h}\, x\, y = {} & \mathtt{g}\, y \\ & \mathtt{where}\ \ \mathtt{g}\, z = x + z \end{aligned}$$

x is free in g.

It is not free in h.

Slide 16

Sets of Free Variables

The set of **free** variables, $FV(M)$, is defined inductively by:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) - \{x\} \\ FV(M\,N) &= FV(M) \cup FV(N) \end{aligned}$$

If $FV(M) = \emptyset$, M is called a **closed term** or **combinator**.

Here are some examples of free and bound variables to check whether you understand

$$\begin{aligned} &FV(\lambda x.\,xy) = \{y\} \\ &FV((\lambda x.\,xy)x) = \{x, y\} \\ &FV(\lambda xy.\,x) = \{\ \} \\ &FV((\lambda x.\,xx)(\lambda x.\,xx)) = \{\ \} \\ &FV(\lambda x.\,x(\lambda y.\,yx)) = \{\ \} \end{aligned}$$

We do not care about the particular names of bound variables, just about the distinctions between them. For example, we know that $f(x) = x + x$ and $f(y) = y + y$ intuitively declare the same functions. Writing these as λ-terms, we think of $\lambda x.\,x + x$ and $\lambda y.\,y + y$ as equal, calling such λ-terms α-*equivalent*.

Slide 17

α-equivalence

α-equivalence is the binary relation on λ-terms defined by the rules:

$$\frac{}{x =_\alpha x} \qquad \frac{M[z/x] =_\alpha N[z/y] \qquad z \notin FV(M) \cup FV(N)}{\lambda x.M =_\alpha \lambda y.N}$$

$$\frac{M =_\alpha M' \qquad N =_\alpha N'}{M\,N =_\alpha M'\,N'}$$

Result $=_\alpha$ is an equivalence relation (reflexive, symmetric and transitive).

This definition depends on the definition of the λ-term $N[M/x]$, which is the **substitution** of λ-term M for variable x in λ-term N, and is defined on the next slide. As well as being used to define α-equivalence, substitution is essential for our definition of β-reduction: for example,

$$(\lambda x.\, x + x)2 \rightarrow_\beta (x + x)[2/x] = 2 + 2.$$

Slide 18

Substitution

The **substitution** of λ-term M for variable x in λ-term N, denoted $N[M/x]$, is defined by:

$$\begin{aligned}
x[M/x] &= M \\
y[M/x] &= y, \qquad y \neq x \\
(\lambda x.N')[M/x] &= \lambda x.N' \\
(\lambda y.N')[M/x] &= \lambda z.N'[z/y][M/x], \quad x \neq y, \\
& \qquad z \notin FV(M) \cup (FV(N') - \{y\}) \cup \{x\} \\
(N_1\, N_2)[M/x] &= N_1[M/x]\, N_2[M/x]
\end{aligned}$$

Slide 19

Examples of Substitution

1. $(\lambda y.\, x)[x/y] = \lambda y.x$
2. If $x \neq y \neq z$, then $(\lambda y.x)[y/x] = \lambda z.y.$
3. $(\lambda x.\, xy)[x/y]$ is $\lambda z.\, zx$ and $\lambda u.\, ux$, for $x \neq y$ and $z, u \notin \{x, y\}$.

Notice that the definition of substitution is a relation, not a function: for example, $(\lambda x.\, xy)[x/y]$ is defined to be both $\lambda z.\, zx$ and $\lambda u.\, ux$. It is a function if we work up to α-equivalence: for example, $\lambda z.\, zx =_\alpha \lambda u.ux$. There is a very simple way to avoid confusion with substitution. Given $N[M/x]$, first rename all bound variables in N using α-equivalence, to be different from x and the free variables in M and N, and obtain $N'[M/x]$. Now do the substitution $N'[M/x]$ which is straightforward as there are no variables clashes. The fact that this works is due to the following results.

Slide 20

Results

1. $M =_\alpha N$ implies $FV(M) = FV(N)$.
2. $FV(N[M/x]) \subseteq (FV(N) - \{x\}) \cup FV(M)$.
3. $N_1 =_\alpha N_2$ implies $N_1[M/x] =_\alpha N_2[M/x]$.
4. $M_1 =_\alpha M_2$ implies $N[M_1/x] =_\alpha N[M_2/x]$.

α-equivalence classes of λ-terms are more important than λ-terms themselves. Textbooks (and these lectures) suppress any notation for α-equivalence classes and refer to an equivalence class via a representative λ-term (look for phrases like “we identify terms up to α-equivalence” or “we work up to α-equivalence”). This means that for most of the time you can just work with λ-terms and not think about α-equivalence at all. For implementations and computer-assisted reasoning, there are various devices for picking canonical representatives of α-equivalence classes (e.g. de Bruijn indexes, graphical representations, . . .).

Slide 21

β-reduction

β-reduction is a binary relation between λ-terms defined by:

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

$$\frac{M \to_\beta M'}{M\,N \to_\beta M'\,N} \qquad \frac{N \to_\beta N'}{M\,N \to_\beta M\,N'}$$

$$\frac{N =_\alpha M \qquad M \to_\beta M' \qquad M' =_\alpha N'}{N \to_\beta N'}$$

We omit the subscript β when the meaning is clear.

Here are more examples of β-reduction:

- $(\lambda xy.\,x)y =_\alpha (\lambda xz.\,x)y \to \lambda z.\,y$
- $(\lambda x.\,xy)((\lambda yz.\,z)u) \to ((\lambda yz.\,z)u)y \to (\lambda z.\,z)y \to y$
- $(\lambda x.\,xy)((\lambda yz.\,z)u) \to (\lambda x.\,xy)(\lambda z.z) \to (\lambda z.z)y \to y$

Notice that the λ-calculus is *not* deterministic: that is, $(\lambda x.\,xy)((\lambda yz.\,z)u) \to ((\lambda yz.\,z)u)y$ and $(\lambda x.\,xy)((\lambda yz.\,z)u) \to (\lambda x.\,xy)(\lambda z.z)$. The λ-calculus *is* confluent.

Slide 22

Many Steps of β-reduction

Given a relation $\longrightarrow_\beta$, we define a new relation $\longrightarrow_\beta^*$ by:

$$\frac{M =_\alpha M'}{M \longrightarrow_\beta^* M'} \qquad \frac{M \longrightarrow_\beta M'' \quad M'' \longrightarrow_\beta^* M'}{M \longrightarrow_\beta^* M'}$$

This relation $\longrightarrow_\beta^*$ is called the reflexive transitive closure of $\longrightarrow_\beta$.

Again, we omit the subscript β when the meaning is clear.

Slide 23

Confluence

Church-Rosser Theorem

$\longrightarrow^*$ is **confluent**: that is, if $M \longrightarrow^* M_1$ and $M \longrightarrow^* M_2$ then there exists M' such that $M_1 \longrightarrow^* M'$ and $M_2 \longrightarrow^* M'$.

Slide 24

β-equivalence

β-equivalence (written $=_\beta$) is the smallest equivalence relation (i.e. reflexive, symmetric, and transitive) that contains $\rightarrow$.

Example. If $M_1 \rightarrow M_2 \leftarrow M_3 \leftarrow M_4 \rightarrow M_5$, then $M_1 =_\beta M_5$.

Lemma. $M_1 =_\beta M_2$ if and only if there exists M such that $M_1 \rightarrow^* M$ and $M_2 \rightarrow^* M$.

The definition of β-equivalence provides a good notion of equality: it is reflexive (in fact, all α-equivalent terms are β-equivalent), symmetric and transitive. The slide also gives a simple characterisation of β-equivalence, which is obtained as a straightforward corollary of the Church-Rosser theorem.

Slide 25

β-Normal Forms

Definition

A λ-term M is in **β-normal form** if it contains no β-redexes: that is, no subterms of the form $(\lambda x.M_1)M_2$.

M has **β-normal form** N if $M \longrightarrow_\beta^* N$ and N is in β-normal form.

Uniqueness of β-normal forms For all M, N_1, N_2, if $M \longrightarrow^* N_1$ and $M \longrightarrow^* N_2$ and N_1, N_2 are in β-normal form, then $N_1 =_\alpha N_2$.

To prove the uniqueness of β-normal forms, we know by the Church-Rosser property that there exists N with $N_1 \longrightarrow^* N$ and $N_2 \longrightarrow^* N$. Since N_1 is a β-normal form and $N_1 \longrightarrow^* N$, then it must be that $N_1 =_\alpha N$ (why?). Hence, $N_1 =_\alpha N_2$.

Slide 26

Non-termination

Some λ-terms have no β-normal form.

For example, $\Omega \triangleq (\lambda x.x\,x)(\lambda x.x\,x)$ satisfies

- $\Omega \rightarrow (x\,x)[(\lambda x.x\,x)/x] = \Omega$,
- $\Omega \rightarrow^{*} M$ implies $\Omega =_{\alpha} M$.

So there is no β-normal form N such that $\Omega =_{\beta} N$.

A term can possess both a β-normal form and infinite chains of reduction from it.

For example, $(\lambda x.y)\Omega \rightarrow y$, but also $(\lambda x.y)\Omega \rightarrow (\lambda x.y)\Omega \rightarrow \cdots$.

Slide 27

BONUS: Call-by-value Reduction

This is a deterministic strategy for β-reduction, in which redexes within function bodies are not reduced, and arguments are fully reduced before being substituted. It is written $\rightarrow_{\beta v}$ and defined by:

$$\frac{N \not\rightarrow_{\beta v}}{(\lambda x.M)N \rightarrow_{\beta v} M[N/x]} \qquad \frac{M \rightarrow_{\beta v} M'}{M\,N \rightarrow_{\beta v} M'\,N}$$

$$\frac{M \not\rightarrow_{\beta v} \qquad N \rightarrow_{\beta v} N'}{M\,N \rightarrow_{\beta v} M\,N'}$$

where $M \not\rightarrow_{\beta v}$ means $\neg\exists M'.\, M \rightarrow_{\beta v} M'$.

Slide 28

BONUS: Call-by-name Reduction

This is another deterministic strategy for β-reduction, in which redexes within function bodies are not reduced, and arguments are not reduced before being substituted. It is written $\rightarrow_{\beta n}$ and defined by:

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

To study more about evaluation strategies for the λ-calculus, read *Call-by name, Call-by-value and the λ-calculus* by Gordon Plotkin in the Journal of Theoretical Computer Science (just search Google). Plotkin invented the call-by-name and call-by-value reduction strategies to bridge the gap between the λ-calculus and Landin's SECD machine. The SECD machine is a highly influential abstract machine intended as a target for functional programming language compilers. The letters stand for Stack, Environment, Code, Dump, the internal registers of the machine. It is used to define the operational semantics of Landin's programming language ISWIM, based on call-by-value evaluation. The programming language ML uses call-by-value evaluation and Haskell uses call-by-need, a memoized[1] version of call-by-name.
The λ-calculus is as expressive as register machines and Turing machines. In this course, we cannot cover the full details, but we can see how to represent arithmetic in the λ-calculus. Here, we give the original encoding of numbers due to Church. Church numerals are functions which take two parameters: a successor function f and a variable x corresponding to $\mathbf{0}$. Numeric functions are represented by corresponding functions on Church numerals.

[1] Memoization is an optimization technique used primarily to speed up computer programs by having function calls avoid repeating the calculation of results for previously processed inputs.

These functions can be implemented in most functional programming languages (subject to type constraints) by direct translation of λ-terms

Slide 29

Church's numerals

$$\begin{aligned}
\underline{0} &\overset{\text{def}}{=} \lambda f\,x.x \\
\underline{1} &\overset{\text{def}}{=} \lambda f\,x.f\,x \\
\underline{2} &\overset{\text{def}}{=} \lambda f\,x.f(f\,x) \\
&\vdots \\
\underline{n} &\overset{\text{def}}{=} \lambda f\,x.\underbrace{f(\cdots(f}_{n\text{ times}}\ x)\cdots)
\end{aligned}$$

Notation: $M^0 N \overset{\text{def}}{=} N; M^1 N \overset{\text{def}}{=} M\,N; M^{n+1} N \overset{\text{def}}{=} M(M^n N)$

We can write $\underline{n}$ as $\lambda f\,x.f^n x$ and we have $\boxed{\underline{n}\,M\,N =_\beta M^n\,N}$.

Slide 30

λ-definable functions

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is **λ-definable** if there is a closed λ-term F that **represents** it: for all $(x_1, \ldots, x_n) \in \mathbb{N}^n$ and $y \in \mathbb{N}$

- if $f(x_1, \ldots, x_n) = y$, then $F\,\underline{x_1} \cdots \underline{x_n} =_\beta \underline{y}$
- if $f(x_1, \ldots, x_n)\uparrow$, then $F\,\underline{x_1} \cdots \underline{x_n}$ has no β-normal form.

Slide 31

Addition is λ-definable

Addition is represented by $P \overset{\text{def}}{=} \lambda x_1\, x_2.\lambda f\, x.\, x_1\, f(x_2\, f\, x)$:

$$\begin{aligned}
P\,\underline{m}\,\underline{n} &\to^* \lambda f\, x.\, \underline{m}\, f(\underline{n}\, f\, x) \\
&\to^* \lambda f\, x.\, \underline{m}\, f(f^n x) \\
&\to^* \lambda f\, x.\, f^m(f^n x) \\
&\overset{\text{def}}{=} \lambda f\, x. f^{m+n} x \\
&\overset{\text{def}}{=} \underline{m+n}
\end{aligned}$$

The other standard arithmetic functions are λ-definable. For example: the function $proj_i^n \in \mathbb{N}^n \to \mathbb{N}$ is represented by $\lambda x_1 \dots x_n . x_i$; the function $zero^n \in \mathbb{N}^n \to \mathbb{N}$ is represented by $\lambda x_1 \dots x_n . \underline{0}$; and the function $succ \in \mathbb{N} \to \mathbb{N}$ is represented by $\mathbf{Succ} \overset{\text{def}}{=} \lambda x_1\, f\, x . f(x_1\, f\, x)$ since

$$\begin{aligned}\mathbf{Succ}\,\underline{n} &\to^* \lambda f\, x.\, f(\underline{n}\, f\, x)\\ &\to^* \lambda f\, x.\, f(f^n\, x)\\ &\overset{\text{def}}{=} \lambda f\, x.\, f^{n+1}\, x\\ &\overset{\text{def}}{=} \underline{n+1}\end{aligned}$$

The definition of computable functions given by register machines or Turing machines is equivalent to the definition of λ-definable functions. Unfortunately, I have only been able to hint at how to provide the connection by showing how to represent numbers in the λ-calculus. I cannot give you the full story. It requires one more part of the jigsaw: *partial recursive functions*.

Slide 32

Computable functions = λ-definable functions

Theorem. A partial function is computable if and only if it is λ-definable.

We know that a function is Register Machine computable if and only if it is Turing computable. I have not told you about partial recursive functions which correspond to Turing-computable functions. However:

Result 1 Every partial recursive function is λ-definable.

Result 2 λ-definable functions are RM computable.

To show Fact 2: (1) code λ-terms as numbers (ensuring that operations for constructing/deconstructing terms are given by RM computable functions on codes); and (2) write a RM interpreter for (normal order) β-reduction.