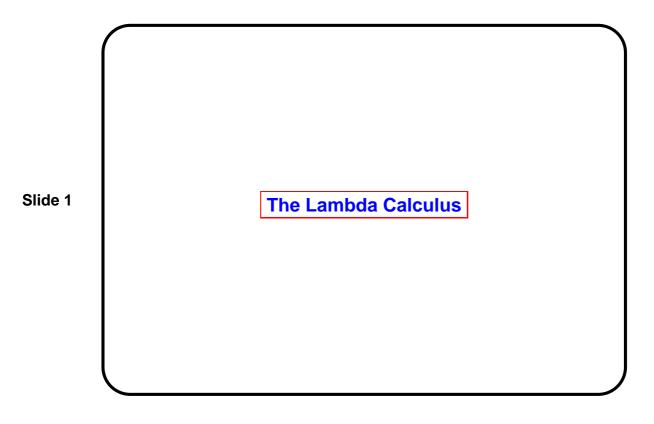
The Lambda Calculus

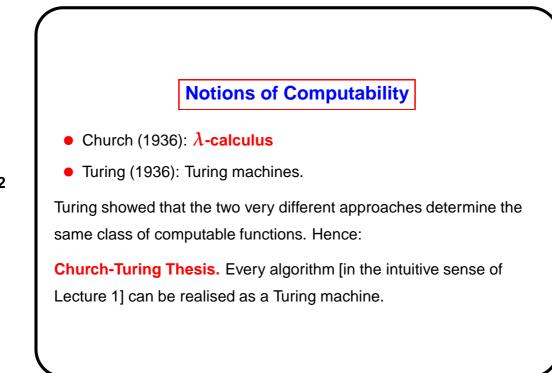
A brief history of mathematical notation.

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.]

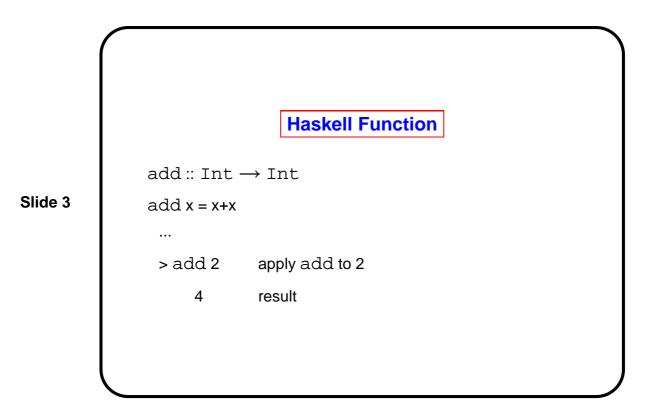
Incidentally, the history of programming languages mirrors that of mathematical notation, albeit in a time-condensed fashion. In the early days (1936-1950), computer engineers struggled with number representation and tried many different schemes, before the modern standard of 2-complement for integers 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. Mc-Carthy knew of the λ -calculus, and his language closely resembles it. This has led to the functional programming languages ML, Haskell and F \sharp .





Models of Computation, 2010

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.



3



```
h:: Int \rightarrow Int \rightarrow 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 $\lambda x \cdot x + x$, or $\lambda x \cdot (x + x)$ 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 $\lambda x \cdot x + x$ to the argument 2 is written $(\lambda x \cdot x + x)$ 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 $\lambda x \cdot x + x$ using a capital letter, say *F*:

 $F \stackrel{\mathrm{def}}{=} \lambda x. x + x$

We can then write *F***2** 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 2, written H2, to obtain another function λz . 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 (H2)3 is the answer 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:

$$\lambda x \cdot \lambda z \cdot x + z$$

Sometimes, we even lazily elide the second lambda:

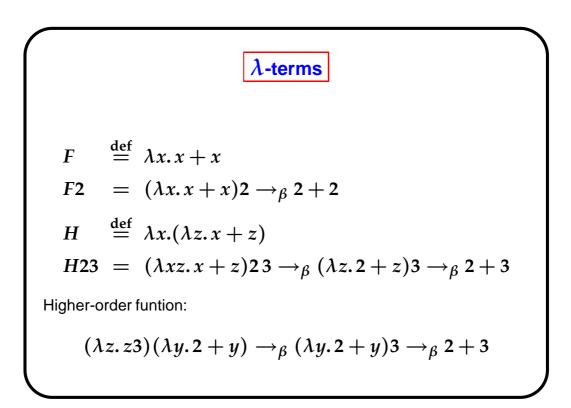
$$\lambda xz. x + z$$

Likewise, in the application of H to arguments 2 and 3, we can use brackets to stress that 2 is to be used first, (H2)3, or we can suggest simultaneaous application by the notation H23. Whatever the intuition about H, the result will be the same.

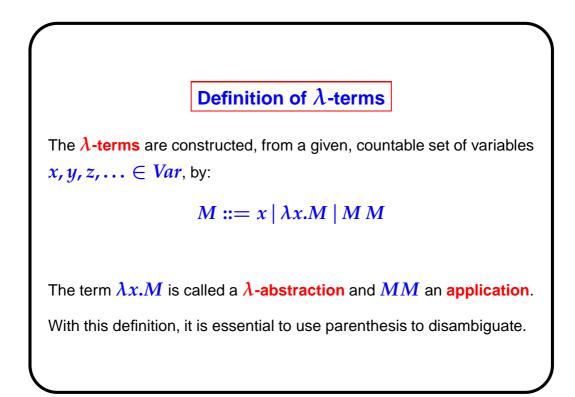
We have already mentioned that, in a good definition of algorithm, an algorithm must be able to take an algorithm as input. With register machines and Turing machines, this is achieved by coding programs as numbers. With the lambda calculus, a function can directly have a function as input. For example,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

$$(\lambda z. z3)(\lambda y. 2 + y)$$

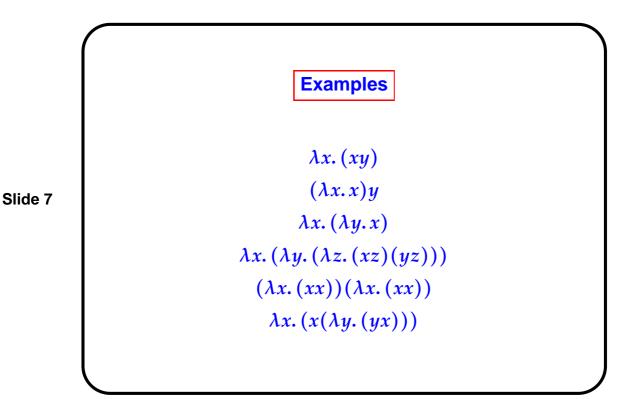
The function $\lambda z. z3$ take the function $\lambda y. 2 + y$ as input, and returns the term $(\lambda y. 2 + y)3$, which can then be evaluated to 5.

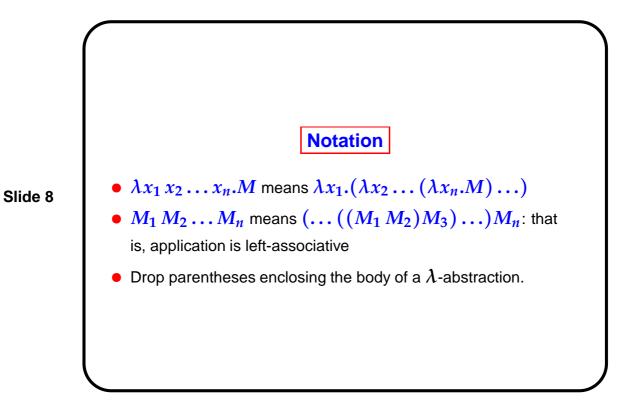


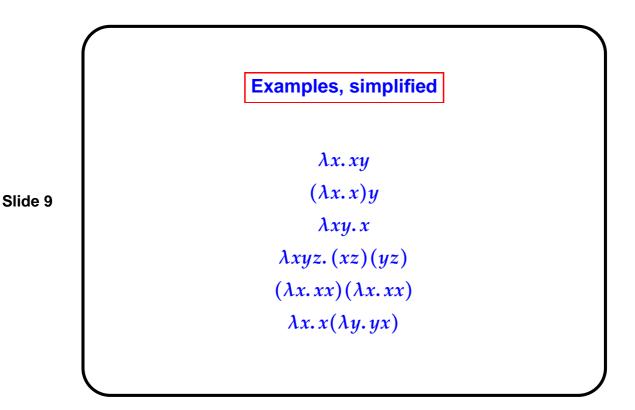
For this course, we will work with the pure λ -calculus without constants (for example, + is a constant): function formation and function application is all there is. We will sometimes use λ -terms with constants as intuitive examples. When developing a programming language based on the λ -calculus, 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!



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

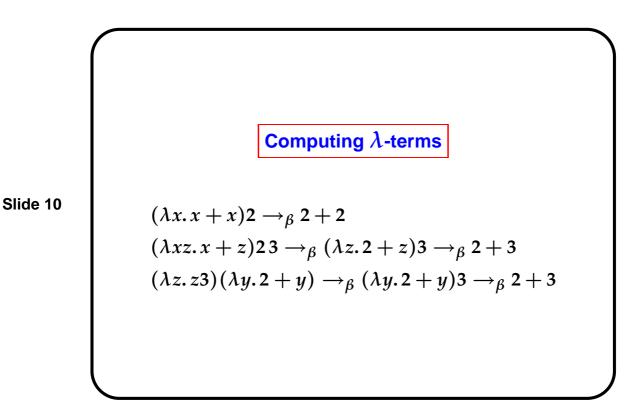




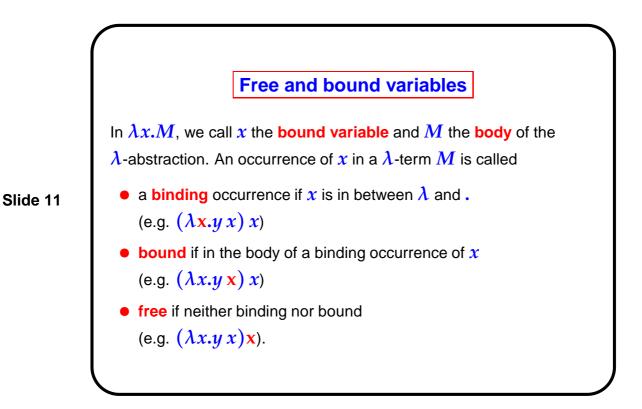


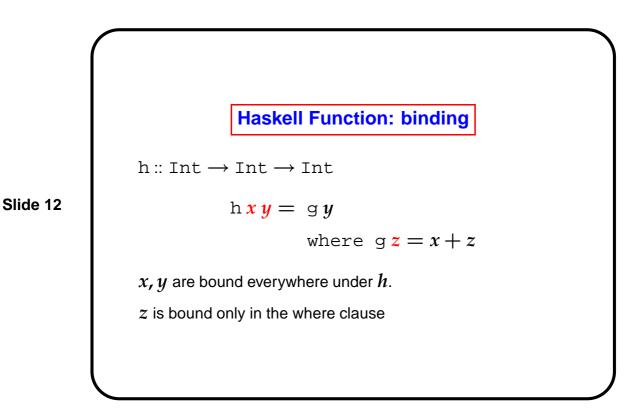
Computing λ -terms

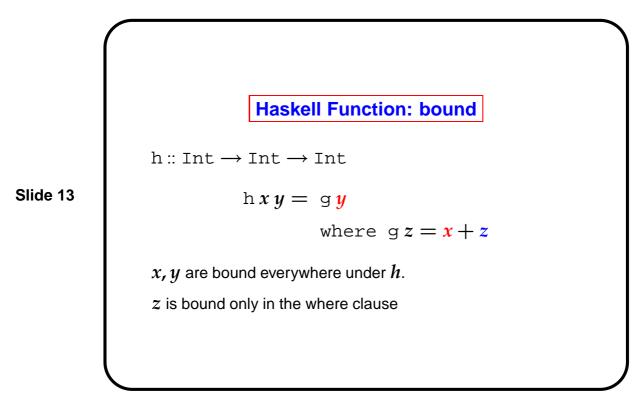
 λ -terms on their own would be a bit boring if we did not know how to *compute* with them as well. There is only one rule of computation, called β -reduction, and it concerns the replacement of the formal parameter by an actual argument. It can only occur if a λ -abstraction has been applied to some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

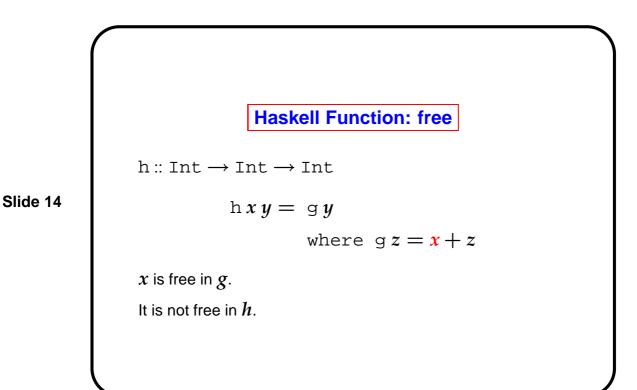


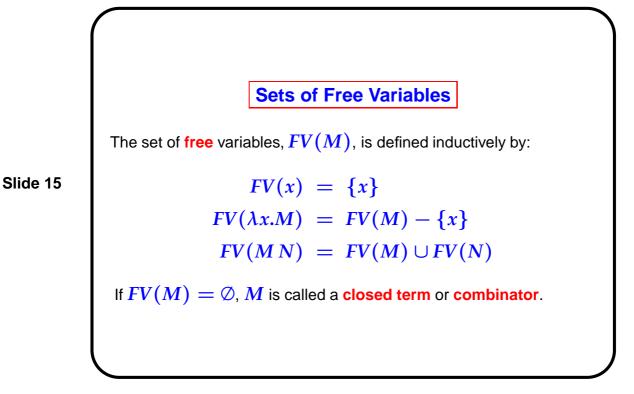
We see that reduction is nothing other than the textual replacement of a formal parameter in the body of a function by the actual parameter supplied. However, we need to do quite a bit of work to formalise what we mean by this. Consider the function f(x) = x + x: the *x*s in the expression x + xare considered *free*; in f(x) = x + x, the *x* in the argument to the function *f* is a formal parameter that *binds* the *x*s in x + x; and the *x*s 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.











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

$$FV(\lambda x. xy) = \{y\}$$

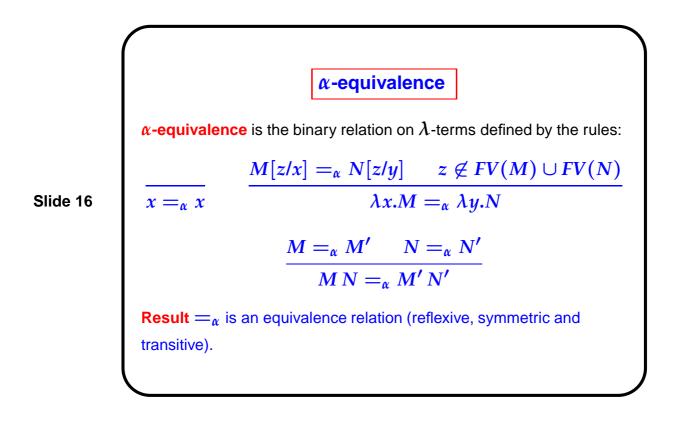
$$FV((\lambda x. xy)x) = \{x, y\}$$

$$FV(\lambda xy. x) = \{\}$$

$$FV((\lambda x. xx)(\lambda x. xx)) = \{\}$$

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

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 $\lambda x \cdot x + x$ and $\lambda y \cdot y + y$ as equal, calling such λ -terms α -equivalent.



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,

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

Substitution

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

$$x[M/x] = M$$

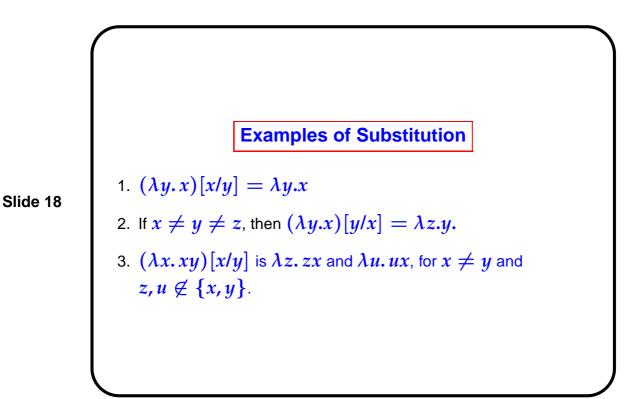
$$y[M/x] = y, \quad y \neq x$$

$$(\lambda x.N')[M/x] = \lambda x.N'$$

$$(\lambda y.N')[M/x] = \lambda z.N'[z/y][M/x], \quad x \neq y,$$

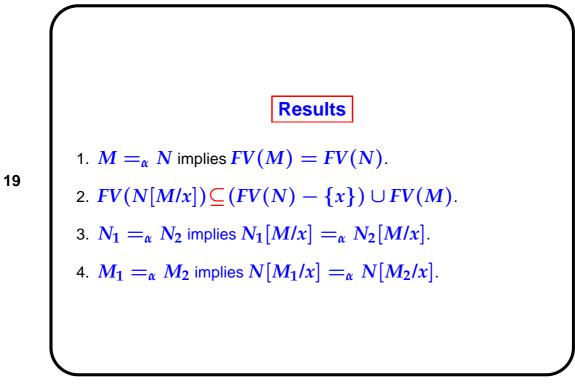
$$z \notin FV(M) \cup (FV(N') - \{y\}) \cup \{x\}$$

$$(N_1 N_2)[M/x] = N_1[M/x] N_2[M/x]$$

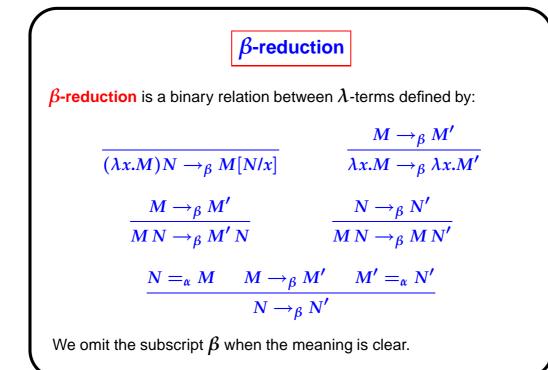


Models of Computation, 2010

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.



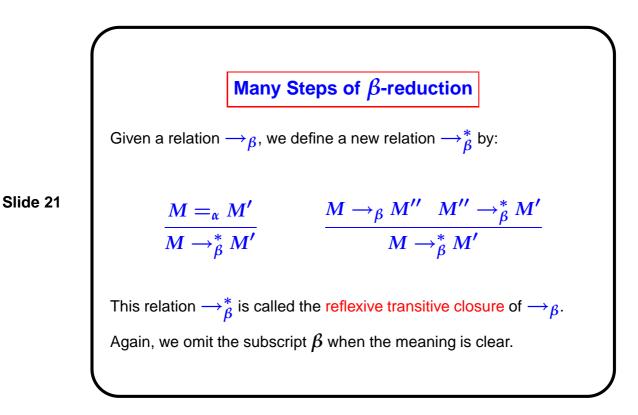
 α -equivalence classes of λ -terms are more important than λ -terms themselves. Textbooks (and these lectures) suppress any notation for α equivalence classes and refer to an equivalence class via a representative λ -term (look for phrases like "we identify terms up to α -equivalence" or "we work up 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, ...).

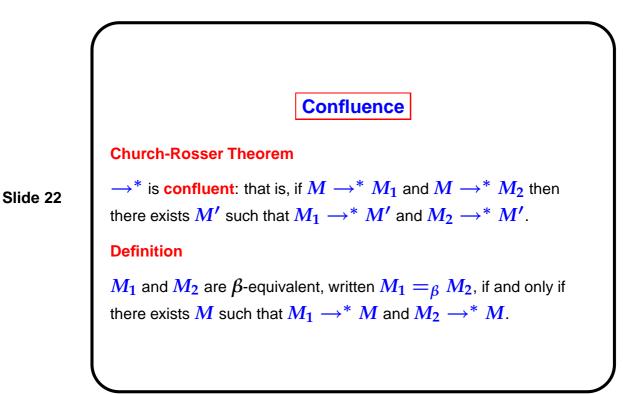


Here are some examples of β -reduction:

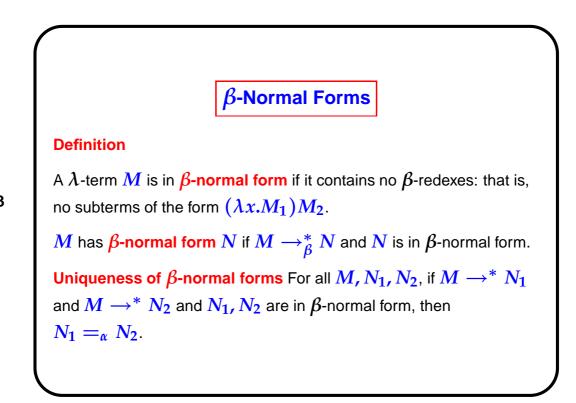
- 1. $(\lambda xy. x)y =_{\alpha} (\lambda xz. x)y \rightarrow \lambda z. y$
- 2. $(\lambda yz.z)u \rightarrow \lambda z.z$
- 3. $(\lambda x. xy)(\lambda x. x) \rightarrow (\lambda x. x)y \rightarrow y$
- 4. $(\lambda x. xy)((\lambda yz. z)u) \rightarrow ((\lambda yz. z)u)y \rightarrow (\lambda z. z)y \rightarrow y$

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

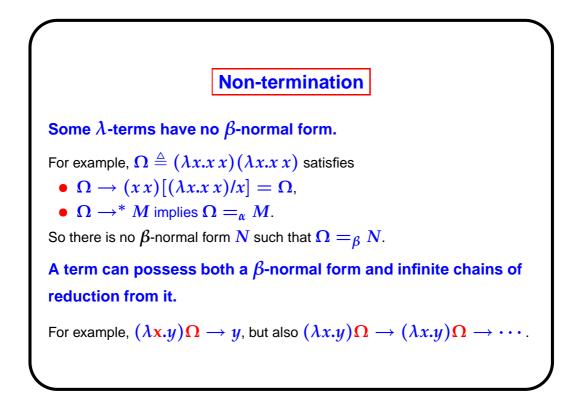


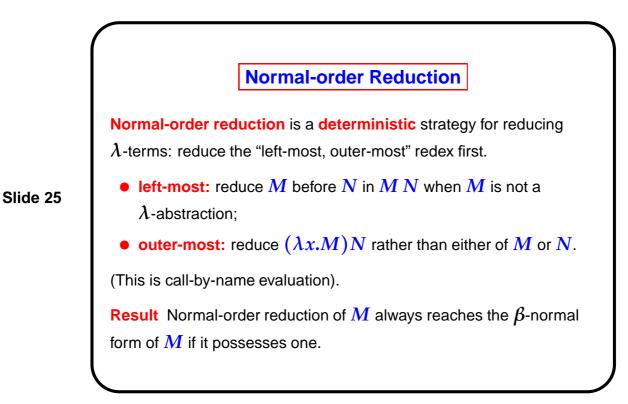


The definition of β -equivalence provides a good notion of equality: it is reflexive (in fact, all α -equivalent terms are β -equivalent), symmetric and, by the Church-Rosser theorem, transitive.



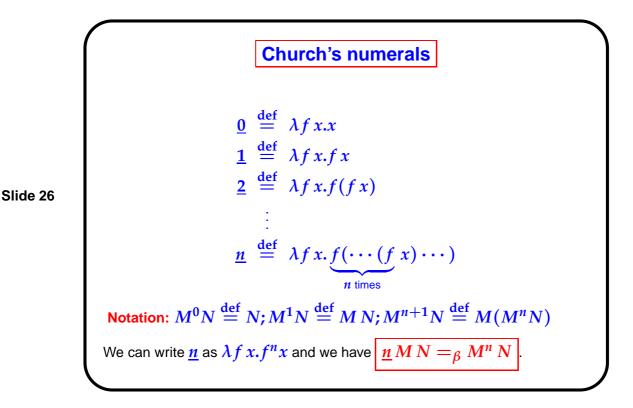
To prove the uniqueness of β -normal forms, we know by the Church-Rosser property that there exists N with $N_1 \rightarrow^* N$ and $N_2 \rightarrow^* N$. Since N_1 is a β -normal form and $N_1 \rightarrow^* N$, then it must be that $N_1 =_{\alpha} N$ (why?). Hence, $N_1 =_{\alpha} N_2$.



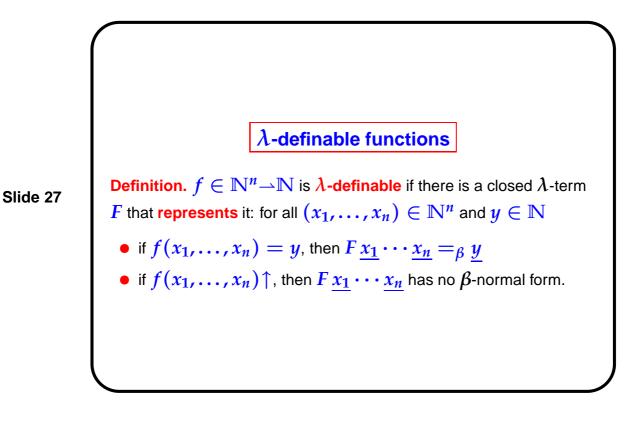


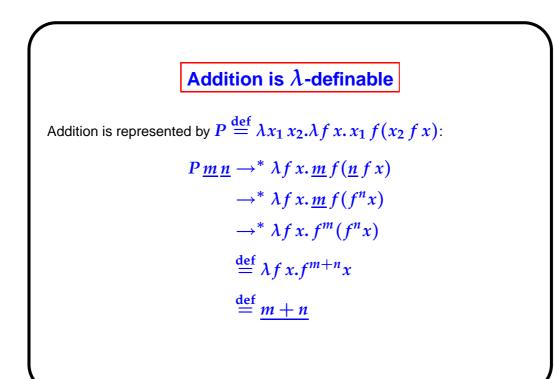
To study more about evaluation strategies for the λ -calculus, read *Call-by* name, *Call-by-value and the* λ -calculus by Gordon Plotkin in the Journal of Theoretical Computer Science (just search Google). Plotkin invented the call-by-name and call-by-value reduction strategies to bridge the gap between 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 memoized¹ 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 0. Numeric functions are represented by corresponding functions on Church numerals. These functions can be implemented in most functional programming languages (subject to type constraints) by direct translation of λ -terms



¹Memoization 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.



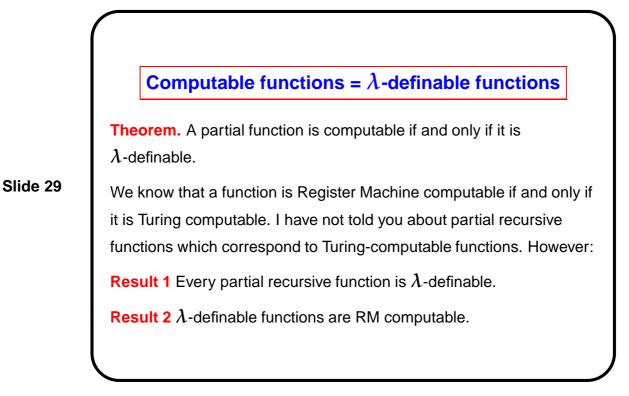


Models of Computation, 2010

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 $\lambda x_1 \dots x_n \cdot x_i$; the function $zero^n \in \mathbb{N}^n \to \mathbb{N}$ is represented by $\lambda x_1 \dots x_n \cdot \underline{0}$; and the function $succ \in \mathbb{N} \to \mathbb{N}$ is represented by Succ $\stackrel{\text{def}}{=} \lambda x_1 f x \cdot f(x_1 f x)$ since

Succ
$$\underline{n} \to^* \lambda f x$$
. $f(\underline{n} f x)$
 $\to^* \lambda f x$. $f(f^n x)$
 $\stackrel{\text{def}}{=} \lambda f x$. $f^{n+1} x$
 $\stackrel{\text{def}}{=} n+1$

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*.



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.

23