Introduction to Lambda-Calculus

By John Lång, 20 February 2020

“Yön maku,
kun linnut syöksyy mustina siipinä tähtiä päin;
vaan mikä on nimesi nimi,
tähtesi salainen luku ja numero?”

– Arcana by CMX, from their album Discopolis

An Appetizer: The MIU-System

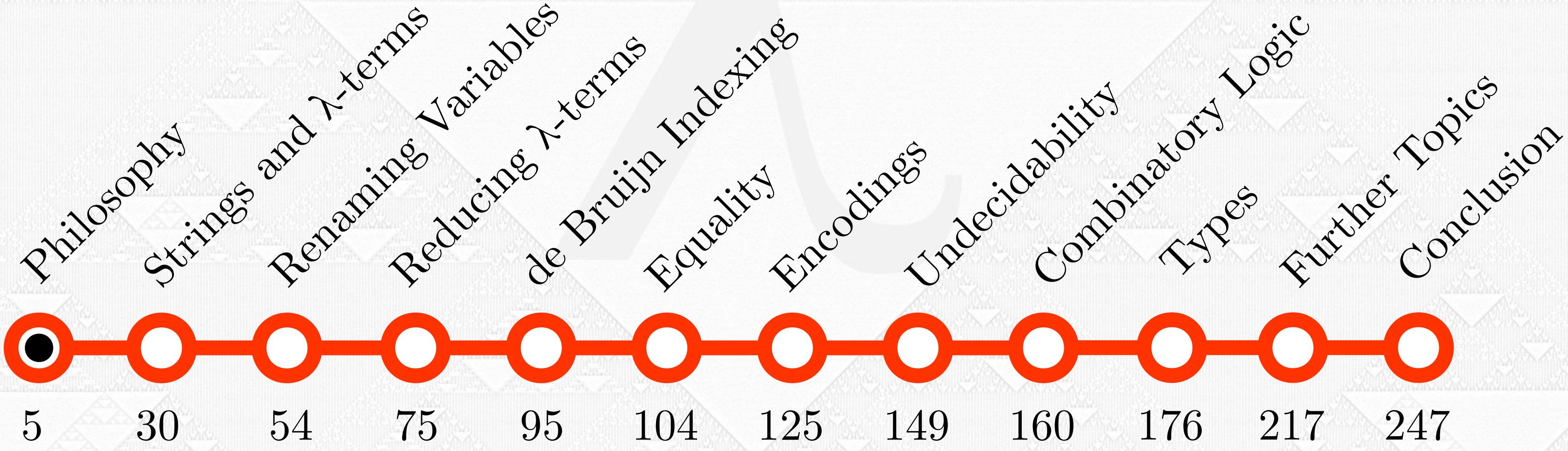
- Douglas Hofstadter presented the following puzzle in his famous book "Gödel, Escher, Bach: an Eternal Golden Braid":
- The MIU-system has the letters M, I, and U in its alphabet
- It has four production rules:

  1) $\alpha I \quad \rightarrow \quad \alpha IU$;
  2) $M\alpha \quad \rightarrow \quad M\alpha\alpha$;
  3) $\alpha III\beta \quad \rightarrow \quad \alpha U\beta$; and
  4) $\alpha UU\beta \quad \rightarrow \quad \alpha\beta$;

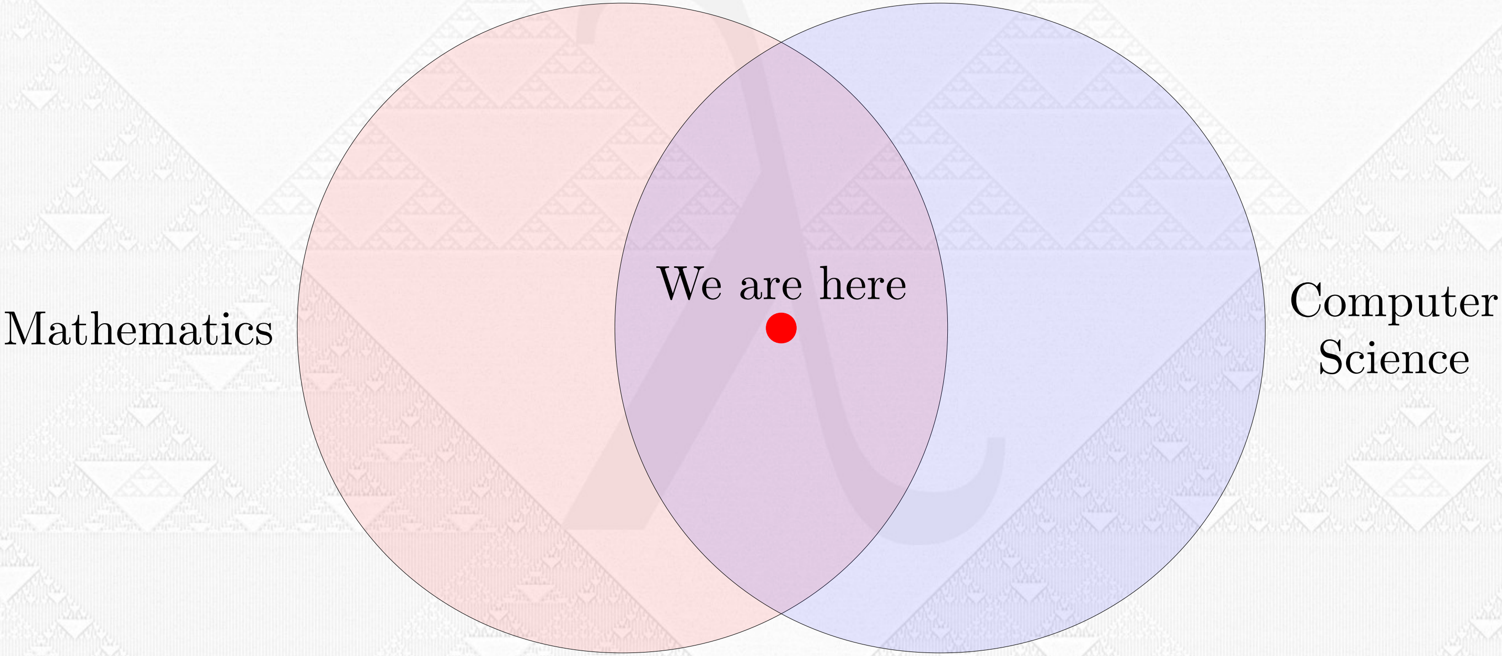
- The question is: Is MU derivable from MI?

Subway Map

- This slide was inspired by the West Metro, that connects Espoo and Helsinki (most of the time). The stops are the concepts worth learning. Page number is written beneath the stops



Map of the Universe



What Is a Function?

Formula

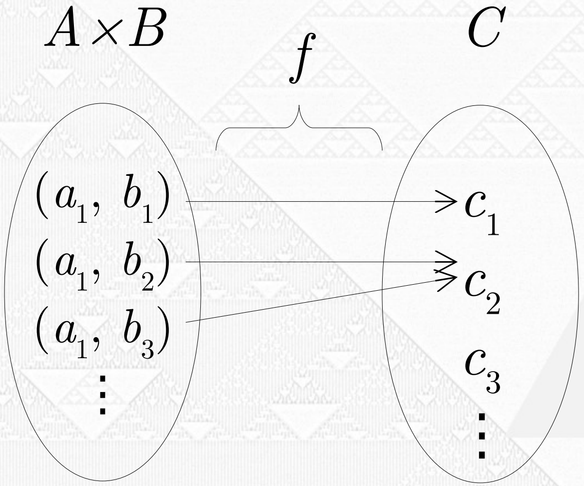
- $f(x) = 2x$
- $g(x,y) = 5$
- $h : X \to Y$
- $x \mapsto 3$
- $(a \circ b)(x) = a(b(x))$

Interpretation

- $f(x)$ yields $2x$
- g is a constant function
- h maps every X to a Y
- x is mapped to 3
- composition of a and b

First-Order vs. Higher-Order

First-Order



$$f = \{((a_1,b_1),c_1),\ ((a_1,b_2),c_2),\ \ldots\}$$

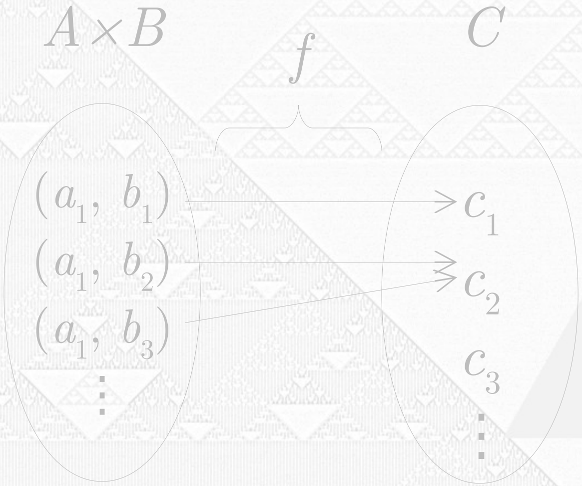
Higher-Order

$$\begin{tikzcd}
A \arrow[d, dashed, "\overline{f}"'] & A \times B \arrow[d, "\overline{f} \times 1_B"'] \arrow[dr, "f"] & \\
C^B & C^B \times B \arrow[r, "\epsilon"'] & C
\end{tikzcd}$$

$$f = \epsilon \circ \overline{f} \times 1_B$$

First-Order vs. Higher-Order

First-Order



$$f = \{((a_1,b_1),c_1),\ ((a_1,b_2),c_2),\ \ldots\}$$

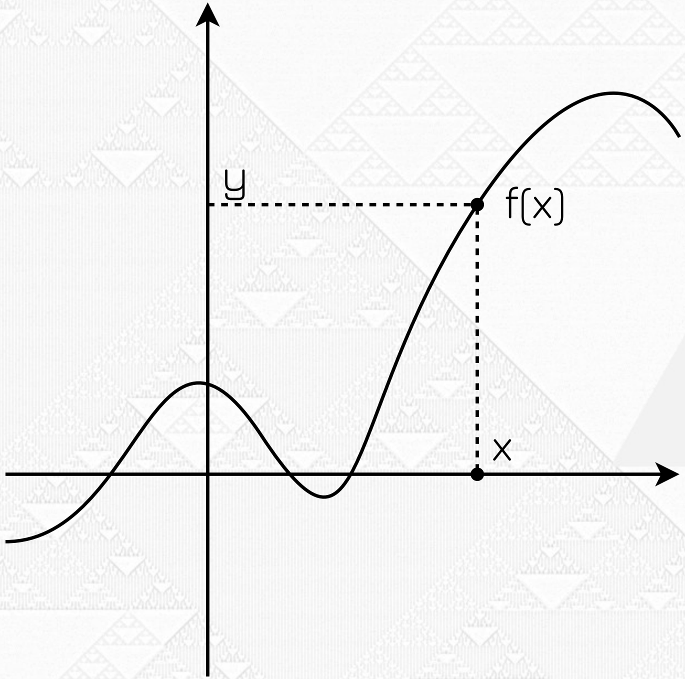
Higher-Order

$$\begin{tikzcd}
A \arrow[d, dashed, "\bar{f}"'] & A \times B \arrow[d, "\bar{f} \times 1_B"'] \arrow[dr, "f"] & \\
C^B & C^B \times B \arrow[r, "\epsilon"'] & C
\end{tikzcd}$$

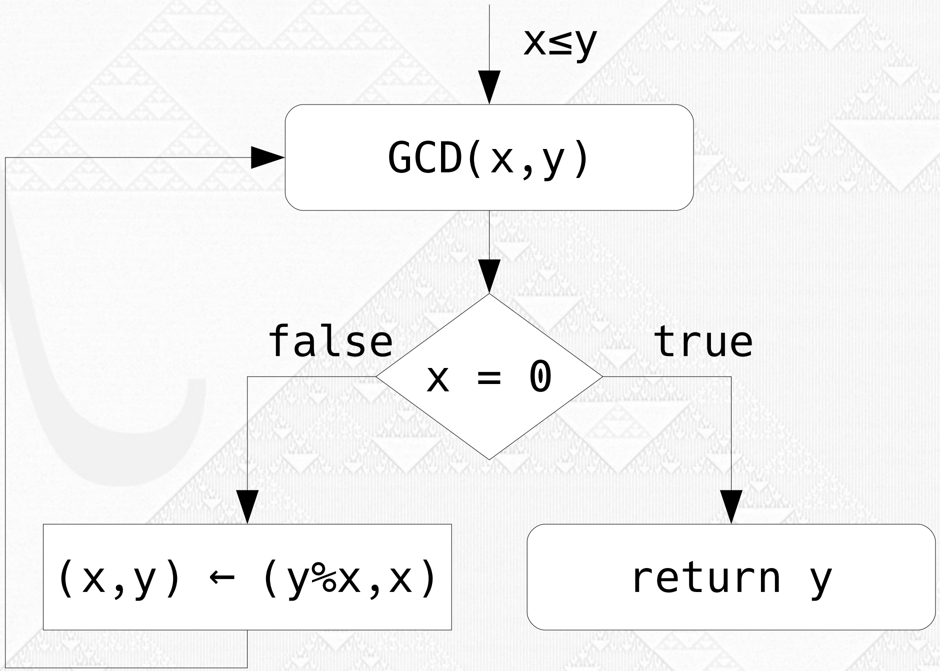
$$f = \epsilon \circ \bar{f} \times 1_B$$

Extensional vs. Intensional

Extensional

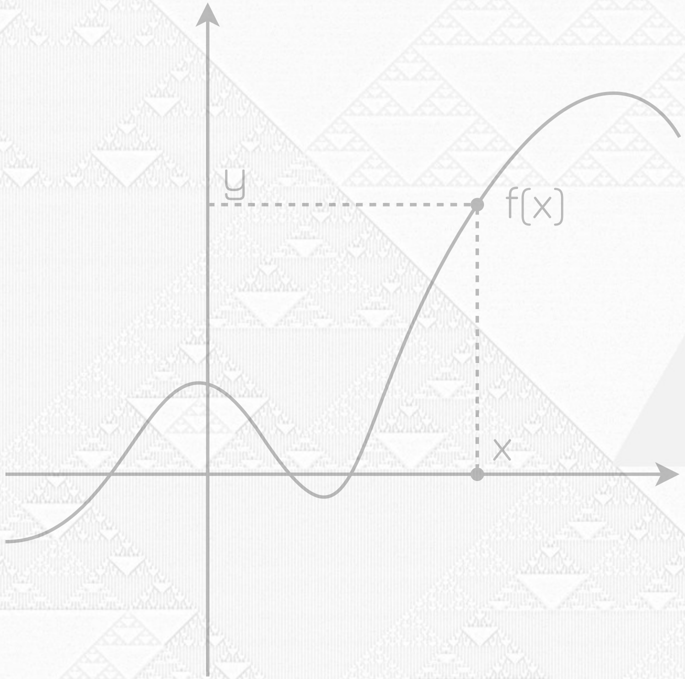


Intensional

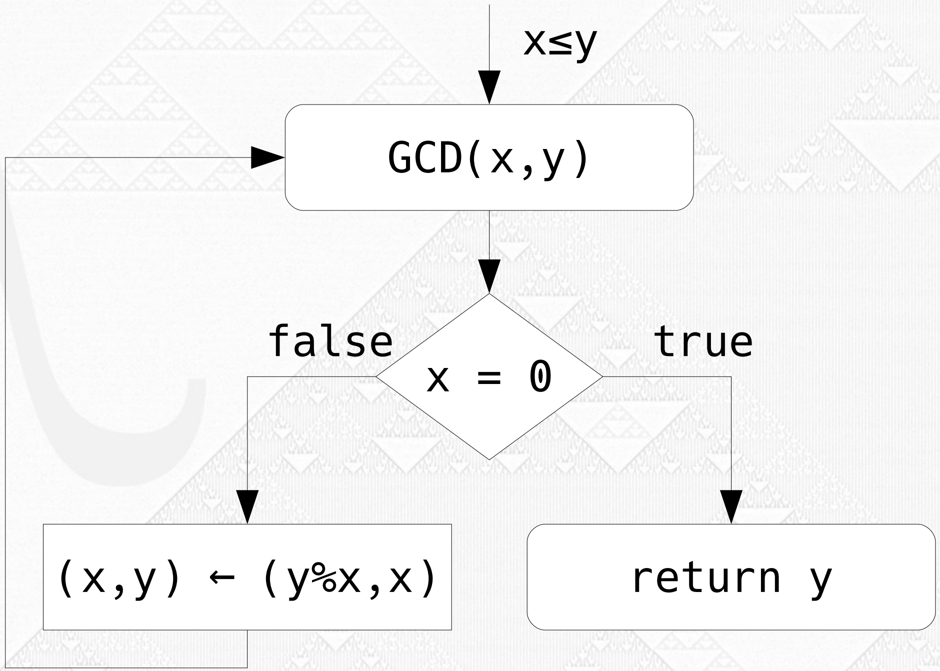


Extensional vs. Intensional

Extensional



Intensional



My Point of View

- According to Wiktionary (viewed some time in 2017):
  - The word "function" comes from a Latin word that means *performance* or *execution*
  - The Latin word "calculus" means *"a pebble or stone used for counting"*
- Does set theoretical mathematics do justice to these concepts?
  - Do you see pebbles in the real line?
- I think of functions as a *primitive* rather than a *derived* notion

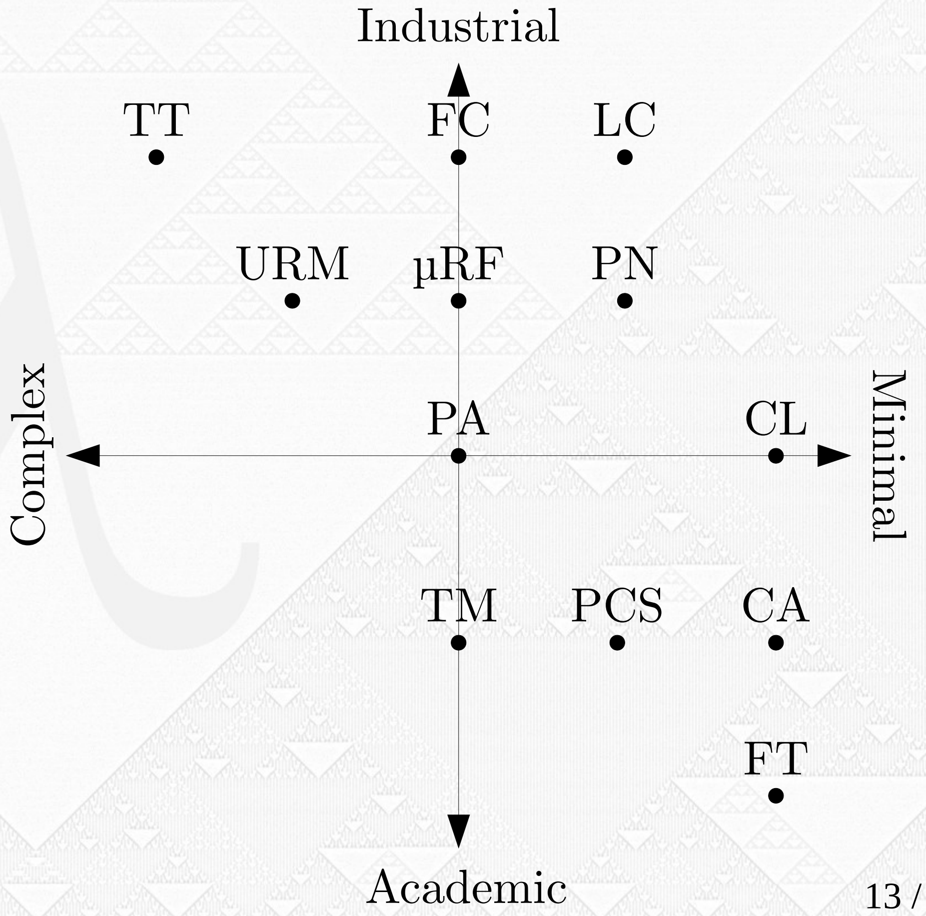
Some Terminology

- *Functions* are abstract dependencies between objects
- *Algorithms* are representations of functions in a formal system. Several algorithms per function; one function per algorithm
- *Strings* can represent functions as well as their arguments
- *Computation* is the art of algorithmic string manipulation
- A *model of computation* is a formal system that defines effectively computable functions in terms of algorithms

Some Models of Computation

Legend:

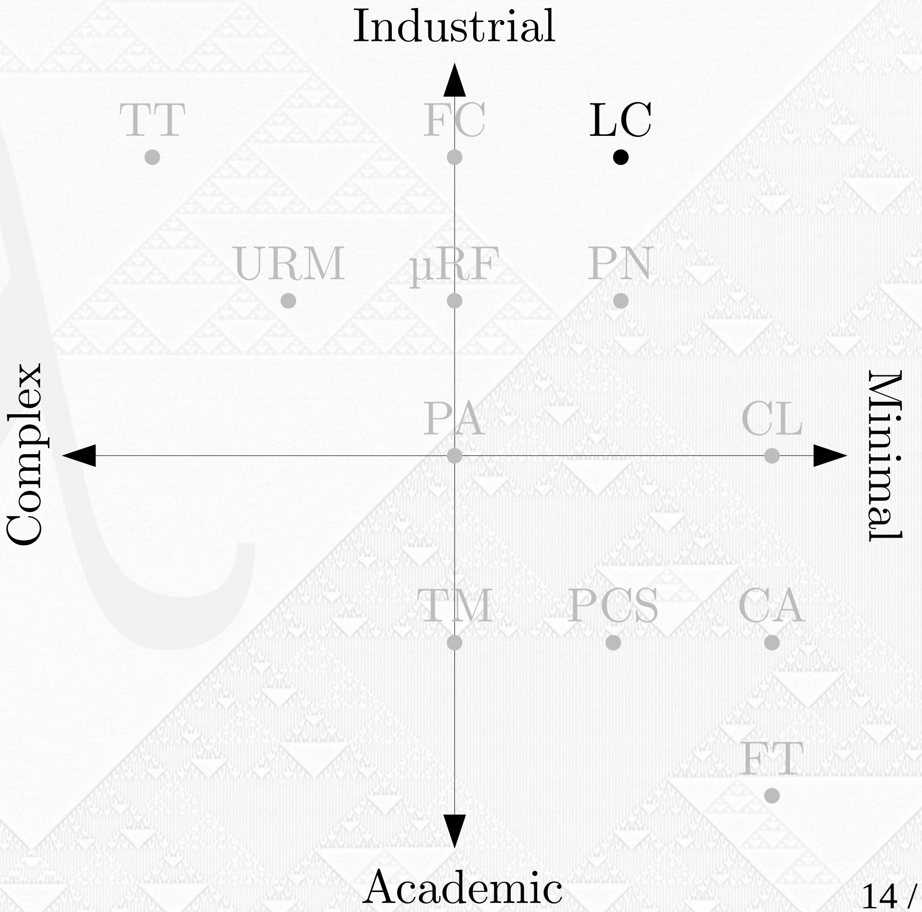
CA	=	Cellular Automata
CL	=	Combinatory Logic
FC	=	Flow Charts
FT	=	FRACTRAN
LC	=	Lambda-Calculus
µRF	=	Mu Recursive Functions
PA	=	Peano Arithmetic
PN	=	Petri Nets
PCS	=	Post Canonical Systems
TM	=	Turing Machines
TT	=	Type Theory
URM	=	Unlimited Register Machines



Some Models of Computation

Legend:

CA = Cellular Automata
CL = Combinatory Logic
FC = Flow Charts
FT = FRACTRAN
LC = Lambda-Calculus
μRF = Mu Recursive Functions
PA = Peano Arithmetic
PN = Petri Nets
PCS = Post Canonical Systems
TM = Turing Machines
TT = Type Theory
URM = Unlimited Register Machines



The Computational Universe

- Lambda-Calculus, Combinatory Logic, Mu Recursive Functions, Turing Machines, and others, represent *the strongest class of models of computation.* (Church-Turing Thesis)
- They're powerful enough to feature undecidability, i.e. the incompleteness of computational universe. (Thanks, Gödel!)
  - There are strange loops between object and meta languages
  - On the bright side, we can extend every formal system infinitely!
  - We'll see later how undecidability emerges in LC

Lambda-Calculus?

- *Lambda-Calculus* (LC) is the model (or language) of computation (i.e. programming) discussed in this presentation.
  - It is a system that expresses functions as strings of symbols
- A few common misconceptions need to be addressed:
  - It's *lambda* (the Greek letter Λ, λ), not "lambada" (the dance)
  - "Calculus" referes to proof calculus, not the differential/integral one
  - "Barendregt convention" was actually initiated by Thomas Ottmann
  - "Curry transformation" was actually discovered by Gottlob Frege

The Rough Idea

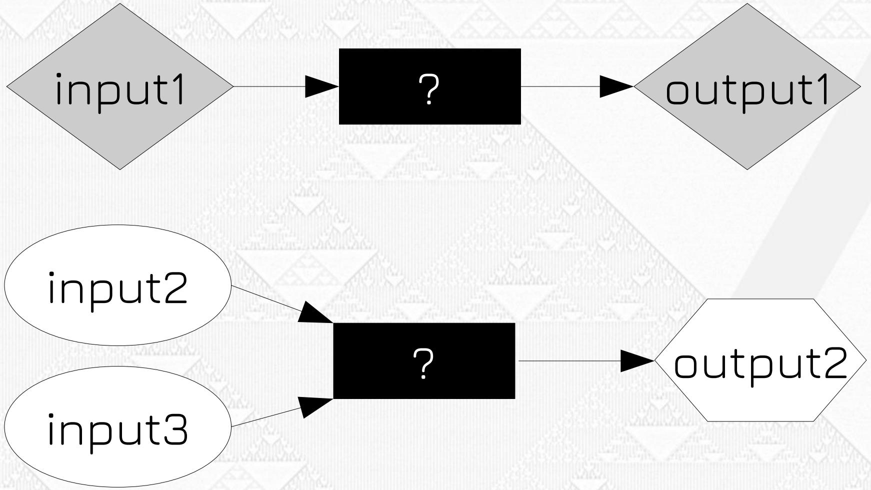
- Lambda-Calculus is about *anonymous functions*, called *lambda expressions* (λ-exprs)
- There are conversion and reduction rules that allow us to reason about (in)equality of λ-exprs. Reducing λ-exprs is like
  - running a computer program;
  - performing a series of algebraic simplifications; or
  - transforming graphs
- We'll see how this works in due time

Teaser: Building Functions

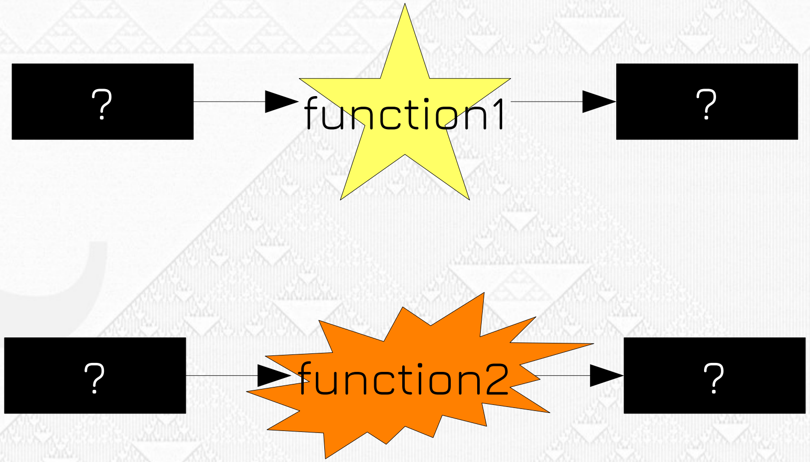
- $f(x) = 2x + c$ translates into $f \stackrel{\text{def}}{=} \lambda x.+\ ((\lambda t.\cdot\ 2\ \ t)x)\ \ c$ in LC
  - $2x$ can be seen as a function of t, $t \mapsto 2 \cdot t$, applied to x
  - In LC, we write $2 \cdot t$ in prefix notation as $\cdot\ 2\ t$
  - c is constant (or free variable); it stays fixed as x varies
  - We bind x as the *formal parameter*: $\lambda x.+\ ((\lambda t.\cdot\ 2\ \ t)x)\ \ c$
- $(\lambda t.\cdot\ 2\ \ t)x$ means evaluating $t \mapsto (\cdot\ 2\ \ t)$ with the argument x
- We can simplify: $f = \lambda x.+\ ((\lambda t.\cdot\ 2\ \ t)x)\ \ c = \lambda x.+\ (\cdot\ 2\ \ x)\ \ c$

The Black Box Analogy Revisited

- Traditionally, functions are seen as black boxes

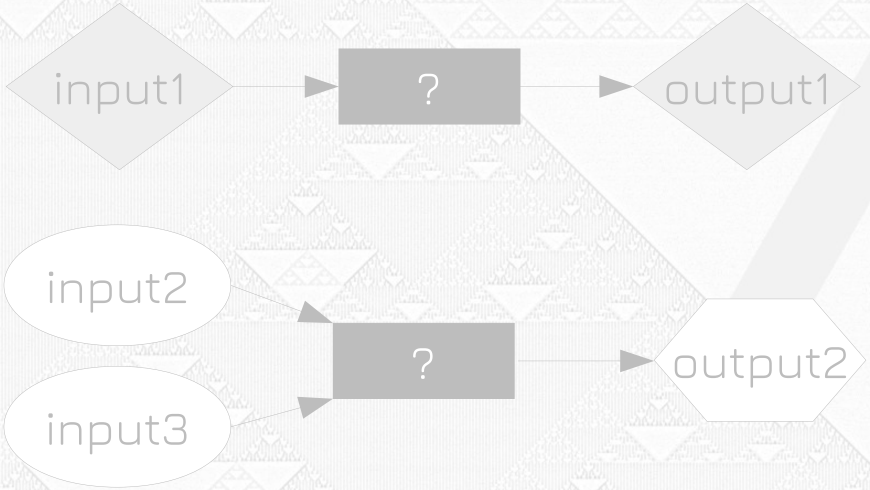


- In LC, functions are the stars of the show

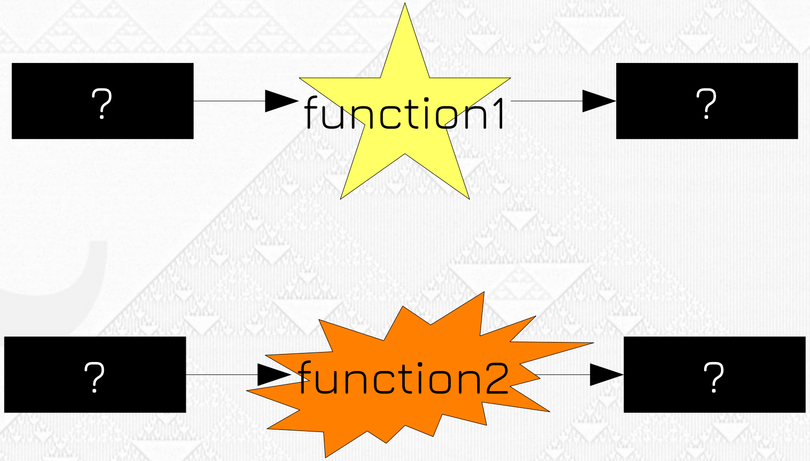


The Black Box Analogy Revisited

- Traditionally, functions are seen as black boxes



- In LC, functions are the stars of the show



Two Programming Paradigms

- *Imperative* paradigm operates algorithms like cooking recipes, on step-by-step basis:

```
0:  IN     R1,=KBD
1:  LOAD   R2,=1
2:  JZER   R1,6
3:  MUL    R2,R1
4:  SUB    R1,=1
5:  JUMP   2
6:  OUT    R2,=CRT
7:  SVC    R0,=HALT
```

- *Declarative* paradigm defines algorithms as compositions of computational primitives:

```
select name, email
    from students join staff
    where email like '%.fi'
```

- LC is declarative
- LC doesn't have implicit state or side effects

Two Programming Paradigms

- *Imperative* paradigm operates algorithms like cooking recipes, on step-by-step basis:

```
0:  IN     R1,=KBD
1:  LOAD   R2,=1
2:  JZER   R1,6
3:  MUL    R2,R1
4:  SUB    R1,=1
5:  JUMP   2
6:  OUT    R2,=CRT
7:  SVC    R0,=HALT
```

- *Declarative* paradigm defines algorithms as compositions of computational primitives:

```
select name, email
    from students join staff
    where email like '%.fi'
```

- LC is declarative
- LC doesn't have implicit state or side effects

Side-Effects Make Program Analysis Hard

- A "function" in C:

```c
int c = 0;
int f(int i)
{
    return i + c++;
}
f(3); // Returns 3
f(3); // Returns 4
```

- Thus, `f(3)` $\neq$ `f(3)`!

Lambda-Calculus Makes Effects Explicit

- A "function" in C:

```
int c = 0;
int f(int i)
{
    return i + c++;
}
f(3); // Returns 3
f(3); // Returns 4
```

- Thus, `f(3)` $\neq$ `f(3)`!

- Consider the following:

```
int g(int c, int i)
    {return i + (c + 1);}
int h(int i) {return g(0);}
```

- The state of `c` is now explicit
- In pseudo-LC, we'd declare

```
int g(int c)(int i);
int h(int i){return g(0)(i);}
```

- Now, `h(3)` = `h(3)`

Traditional Notation is Imprecise

- Take "$f(x) = y$" for instance
  - What does '=' mean?
  - Definition or assertion?
  - Did we apply f to x?
  - Perhaps we multiplied f by x?
  - Do we need the graph of f?
- Is '$\circ$' in $(f \circ g)$ a function?

Lambda-Calculus is Unambiguous

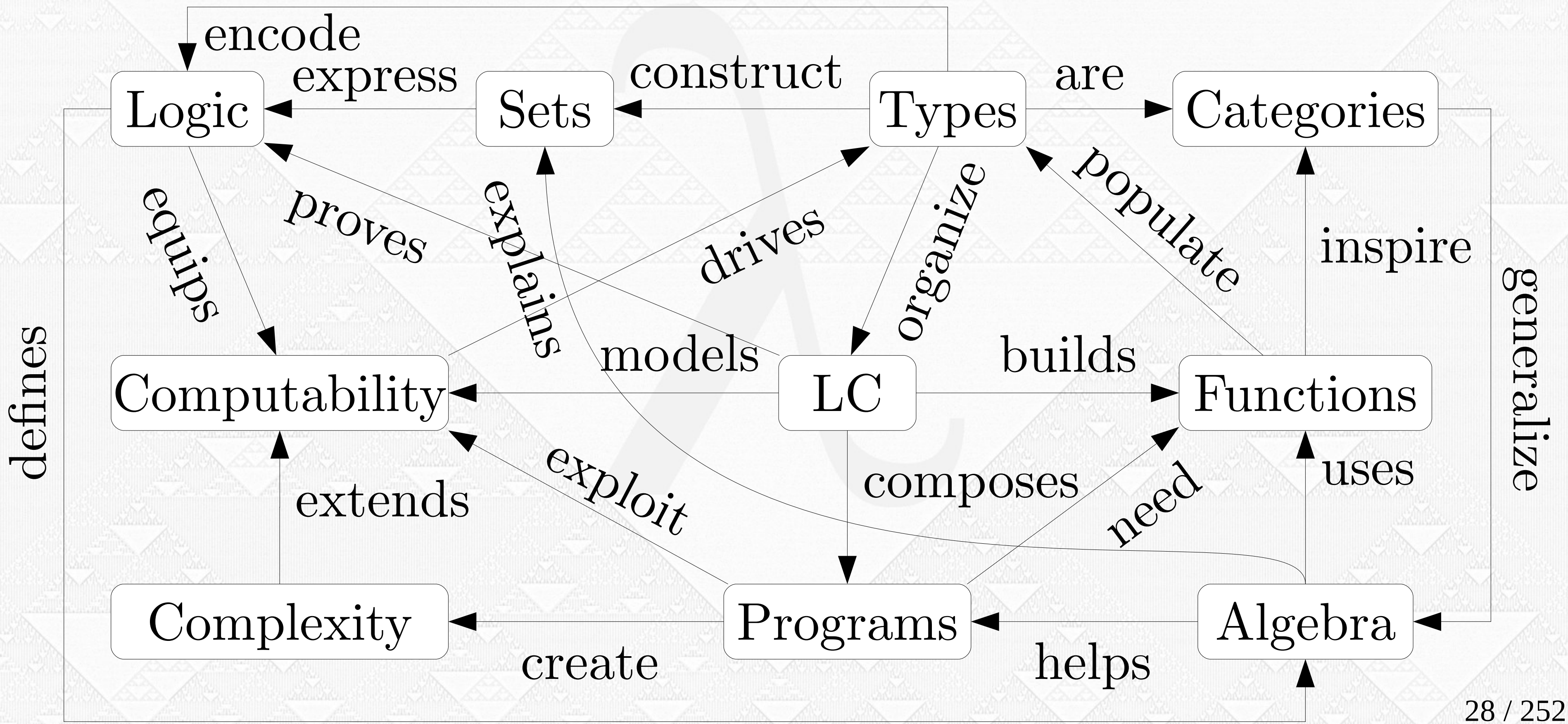
- Take "$f(x) = y$" for instance
  - What does '=' mean?
  - Definition or assertion?
  - Did we apply f to x?
  - Perhaps we multiplied f by x?
  - Do we need the graph of f?
- Is '$\circ$' in $(f \circ g)$ a function?

- In LC, $f(x) = y$ means "*f applied to x equals y*"
  - It is an assertion ("**==**"), not a definition ("**=**")
  - *No other interpretations*
  - '=' always has a direct proof
- $\circ$ is just another function:
  - $f{\circ}g \equiv (\lambda vwx.v(wx))fg$

Why Lambda-Calculus?

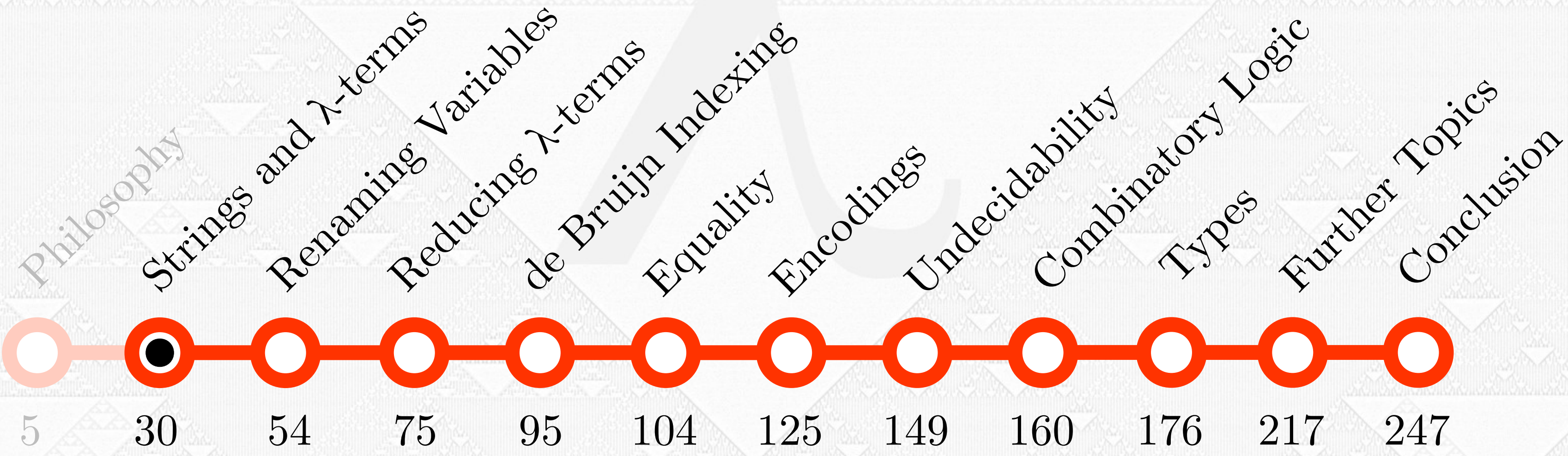
- Lambda-Calculus is capable of describing computable functions using a referentially transparent (i.e. compositional) language
- LC has an unambiguous syntax with very few special cases
- The computer of LC doesn't need soul, intuition, or magic
- LC doesn't burden us with boring and error-prone technical details, such as memory management or instruction sequencing
- LC gives the best of both mathematics and computer science

My Very Small World Map



Subway Map

- You're probably very eager to get started already, so let's get to business!



Before We Start

- Our *object language* is the language of lambda expressions
  - Words in the object language consist of *variable* and other *symbols*
- Our *meta language* (e.g. set/category/topos/type theory, etc.) defines rules for working with lambda expressions
  - Capital Latin letters (i.e. *meta variables*), indices, substitution notation, and various relational symbols belong to this language
- *Don't get confused with object and meta languages!*

Strings

- *Strings* are (finite) ordered sequences of symbols, i.e. objects
  - For example, "a string" is a string (of the English alphabet)
  - Quotation marks separate object and meta languages. They may (and usually will) be omitted, when there's no risk of ambiguity
- We refer to strings by identifying them with *meta variables*.
  - '$V \stackrel{\text{def}}{=}$ "value"' tells that we denote the string "value" with V;
  - '$V \stackrel{\text{def}}{=} W$' declares that V is a shorthand for W; and
  - A meta variable is *interchangeable* with the string it denotes

String Length and Equality

- The length of a string V, written as $|V|$, is the number of (possibly repeated) symbols in it
  - For example, $|$“”$| = 0$, $|$“x”$| = 1$, $|$“xx”$| = 2$, $|$“string”$| = 6$
- Strings V and W, are *equal*, written as $V \equiv W$, if and only if $|V| = |W|$ and they contain the same symbols in the same order
  - For example, “cat” $\equiv$ “cat”, but “cat” $\not\equiv$ “tac” and “cat” $\not\equiv$ “catch”
  - String equality is reflexive, symmetric, and transitive

String Catenation

- $V \star W$ or VW is a string, called *(con)catenation* of V and W. It contains every symbol of first V, then W.
  - E.g. "bat"$\star$"man" $\equiv$ "batman", but "bat"$\star$"man" $\not\equiv$ "man"$\star$"bat"
- For every string U, V, and W:

  1) ""$\star U \equiv U \star$ "" $\equiv U$ ("" is the neutral element of $\star$)

  2) $(U \star V) \star W \equiv U \star (V \star W)$ ($\star$ is associative)
- Catenation also satisfies the equation $|U \star V| = |U| + |V|$

Substrings

- V is a substring of W if and only if $|V| \leq |W|$ and V contains the same symbols as W in the same order, until the end of V
- For instance,
  - V and W (and their substrings) are substrings of VW;
  - "tba" is a substring of "cat"$\star$"bat", but not of "cat" or "bat"
  - "at" and "cat" are substrings of "cat"; "Cat" is not a substring of "cat" (as "C" $\not\equiv$ "c")

Alphabet of Lambda-Calculus

- The objects of study in LC are (non-empty) strings known as *lambda expressions* (λ-exprs, a.k.a. *lambda terms*). Their alphabet contains:
  1) 'λ' (lambda)
  2) '.' (dot)
  3) '(' (left parenthesis)
  4) ')' (right parenthesis)
  5) x_0, x_1, x_2, ... (variable symbols)

First Definition: Induction

- Let x be a variable symbol and M, N (meta!) λ-exprs. Then:
  1) x is a lambda expression (called *variable*);
  2) $(\lambda x.M)$ is a lambda expression (called *abstraction*);
  3) (MN) is a lambda expression (called *application*); and
  4) nothing else is a lambda expression.
- The definitions above may be applied *recursively*; e.g. since x is a λ-expr, (xx), $(\lambda x.(xx))$, $((\lambda x.(xx))(xx))$, etc. are also λ-exprs

Second Definition: BNF

- Alternatively, we can use *Backus-Naur Form* (BNF). Let

```
<λ-expr>   ::= <λ-var> | <λ-abstr> | <λ-app>
<λ-var>    ::= “(” <λ-symbol> “)”
<λ-abstr>  ::= “(λ” <λ-symbol> “.” <λ-expr> “)”
<λ-app>    ::= “(” <λ-expr> <λ-expr> “)”
```

  where `<λ-symbol>` can be replaced with an appropriate variable symbol from the alphabet

- We can construct every (*finitary*) λ-expr be applying these production rules *finitely many times*

Third Definition: Deduction Rules

- Yet another way to define lambda expressions is to use *formation rules*
- A string of symbols is a λ-expr iff it can be derived using the rules on the right
- Rules are useful, aren't they?

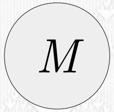
$$\frac{}{x : \lambda\text{-expr}}\ (\text{var})$$

$$\frac{x : \lambda\text{-expr} \qquad M : \lambda\text{-expr}}{\lambda x.M : \lambda\text{-expr}}\ (\text{abs})$$

$$\frac{M : \lambda\text{-expr} \qquad N : \lambda\text{-expr}}{MN : \lambda\text{-expr}}\ (\text{app})$$

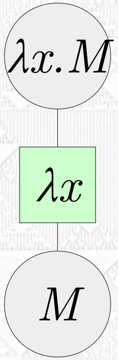
Fourth Definition: Parse Trees

Expression

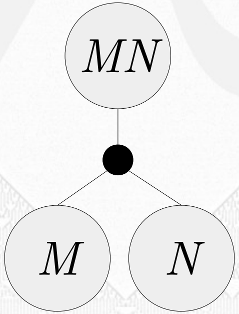


Variable

Abstraction



Application



- Subexpressions are below the bigger expressions
- Colours identify variables
- Trees can be nested
- More on parse trees shortly

Syntactic Sugar

- It shall be declared that:
  1) Outermost parentheses may be omitted;
  2) abstraction binds as far to the right as possible;
  3) $MNO \stackrel{\text{def}}{=} (MN)O$, so application is *left-associative*;
  4) $\lambda x.\lambda y.M \stackrel{\text{def}}{=} \lambda x.(\lambda y.M)$, so abstraction is *right-associative*; and
  5) $\lambda xy.M \stackrel{\text{def}}{=} \lambda x.\lambda y.M$ (in general, $\lambda x_0 x_1 x_2 \ldots x_n.M \stackrel{\text{def}}{=} \lambda x_0.\lambda x_1.\lambda x_2 \ldots \lambda x_n.M$).
- For example, $(\lambda x.x)y$ is different from $\lambda x.xy$ (i.e. $\lambda x.(xy)$)

Subexpressions

- A *subexpression* (or *subterm*) N of a λ-expr M is
  1) a substring of M; such that
  2) N is a λ-expr in its own right; and
  3) M can be formed from N using the syntax rules of LC.
- For example:
  - x is a subexpression of itself, $(\lambda x.x)$, and (xy); and
  - λw or $(\lambda x.z)$ are not subexpressions of x, (xy), or $(z(\lambda x.y)wv)$.

A Corner Case

- Consider $\lambda xz.xz$. One might think that it has $\lambda xz.x$ or $\lambda z.x$ as a subexpression...
- But it doesn't! Let's unroll the syntactic sugar:

$$\begin{aligned} \lambda xz.xz &\equiv \lambda x.\lambda z.xz \\ &\equiv \lambda x.(\lambda z.(xz)) \end{aligned}$$

- The subexpressions of $\lambda xz.xz$ are $\lambda xz.xz$, $\lambda z.xz$, xz, x, and z
- The lesson: *The definition applies to de-sugared expressions*

Did You Get The Syntax?

Are the following strings are λ-exprs? (Why?)

- λx
- $(\lambda x.x$
- $zx(\lambda x.y)$
- $\lambda xy.z$
- $(\lambda x.x)(\lambda x.x)$
- c
- (c)

Did You Get The Syntax?

Are the following strings are λ-exprs? (Why?)

- λx — No. The expression after "λx" is missing
- $(\lambda x.x$ — No. The right parenthesis is missing
- $zx(\lambda x.y)$ — Yes.
- $\lambda xy.z$ — Yes. Stands for $(\lambda x.\lambda y.z)$
- $(\lambda x.x)(\lambda x.x)$ — Yes, it's $(\lambda x.x)$ applied to itself
- c — Of course! (A variable expression)
- (c) — Yes, it's the same as c above

Are These Expressions Identical?

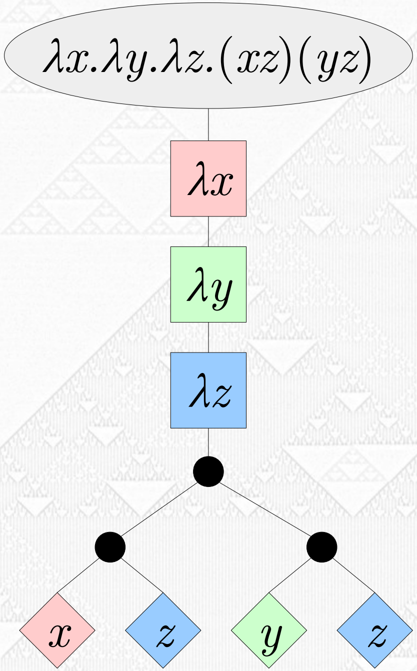
- Are the following pairs of lambda expressions identical?
  - (x) and x
  - $z(\lambda x.y)$ and $(\lambda x.y)z$
  - $x(\lambda x.y)z$ and $(x)((\lambda x.y)z)$
  - $wx(yz)$ and $(wx)(yz)$
  - $\lambda x.(\lambda y.x)z$ and $\lambda x.\lambda y.xz$
  - $\lambda wx.(\lambda y.z)$ and $\lambda wxy.z$
  - $\lambda xy.z$ and $\lambda x.yz$

Are These Expressions Identical?

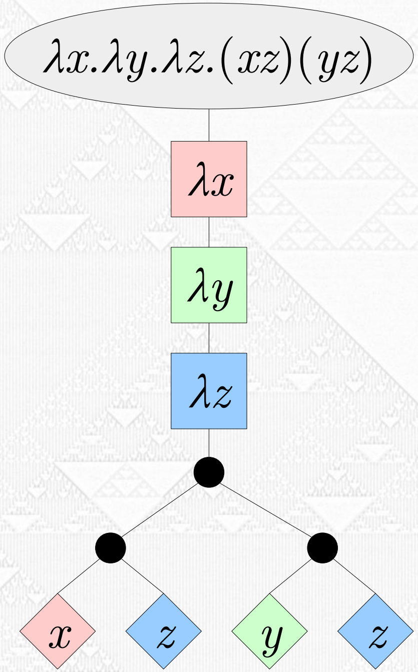
- Are the following pairs of lambda expressions identical?
  - (x) and x — Yes. Outer parentheses omitted
  - $z(\lambda x.y)$ and $(\lambda x.y)z$ — No, there's no commutativity
  - $x(\lambda x.y)z$ and $(x)((\lambda x.y)z)$ — No. Application is left-associative
  - $wx(yz)$ and $(wx)(yz)$ — Yes. Application is left-associative
  - $\lambda x.(\lambda y.x)z$ and $\lambda x.\lambda y.xz$ — No. $(\lambda y.x)z \not\equiv \lambda y.xz$
  - $\lambda wx.(\lambda y.z)$ and $\lambda wxy.z$ — Yes
  - $\lambda xy.z$ and $\lambda x.yz$ — No. $\lambda xy.z \equiv \lambda x.\lambda y.z \not\equiv \lambda x.yz$

Tree Analogy

- Lambda expressions can be combined into larger expressions, using the syntax rules given in p. 35
- The structure of a lambda expression can be visualized with a parse tree (or *abstract syntax tree*) (see p. 39)



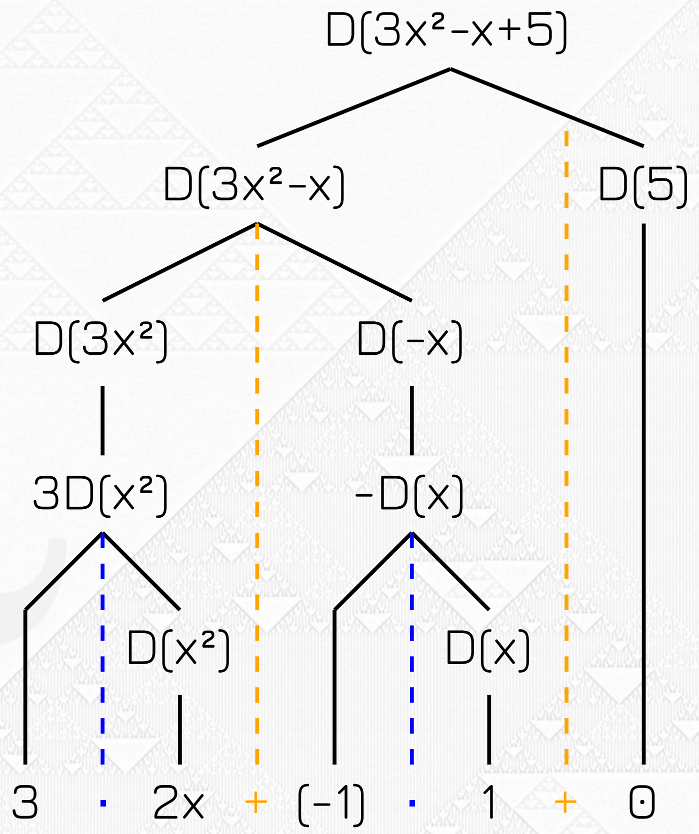
Side Note: Proof Trees



$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\overline{x}\qquad\overline{z}}{(xz)}\qquad\cfrac{\overline{y}\qquad\overline{z}}{(yz)}}{(xz)(yz)}}{\lambda z.(xz)(yz)}}{\lambda y.\lambda z.(xz)(yz)}}{\lambda x.\lambda y.\lambda z.(xz)(yz)}$$

The Other Calculus

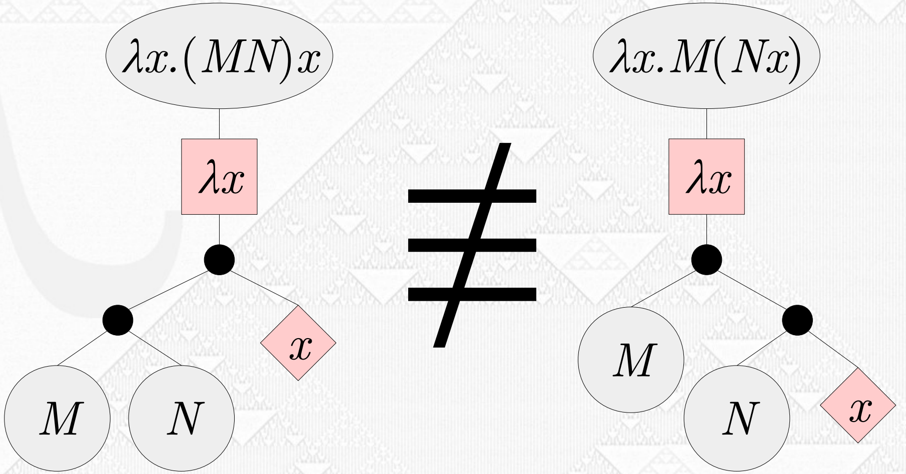
- Here's another tree analogy, representing the process of finding the derivative of a polynomial function
- Notice how structural it is
- No need to think about points or limits



The Syntax of LC Lacks Symmetry

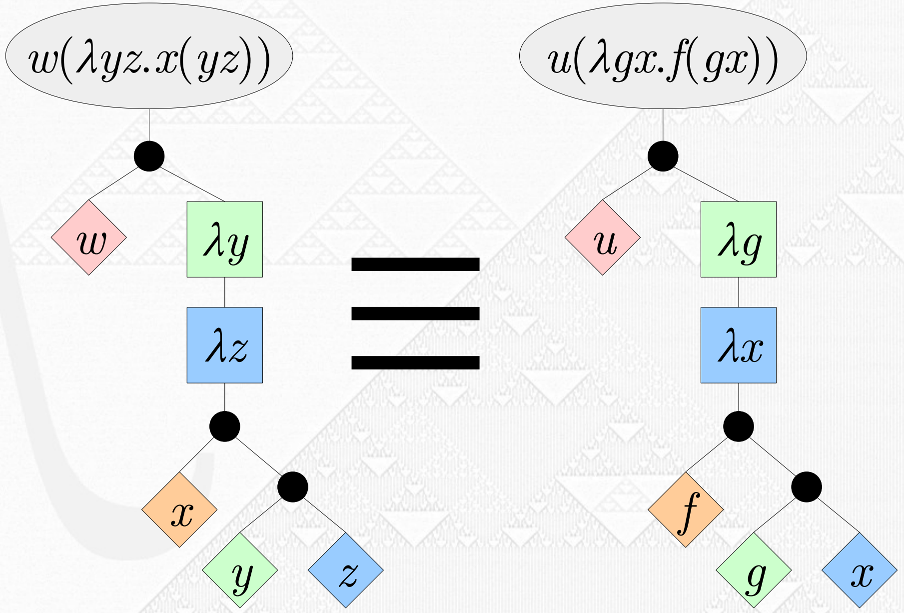
- Remember, that
  - $(MN)K \not\equiv M(NK)$
  - $MN \not\equiv NM$ (unless $M \equiv N$)
  - $(\lambda x.x)y \not\equiv \lambda x.(xy)$
- Cf. function composition:
  - Usually, $\quad -(x^{\mathbf{2}}) \quad \neq \quad (-x)^{\mathbf{2}}$
  - hence, $\quad -\circ^{\mathbf{2}} \quad \neq \quad {}^{\mathbf{2}}\circ-$

- No general *associativity*, *commutativity*, or *distributivity* in LC!



String Equality is Very Limiting

- Consider parse trees for $w(\lambda yz.x(yz))$ and $u(\lambda gx.f(gx))$
- They're essentially the same tree, with different labels
- We'll want to focus on their structure instead of concrete typography



From Lambdas to Calculus

- So far, we are only familiar with strings and lambda expressions. We don't have the "Calculus" part yet
- The idea in LC is to show that the dynamical behaviour of functions can be expressed in terms of a static language
- At this point, the only notion of equivalence between lambda expressions we have, is string equality (up to sugaring)
- In order to construct a more useful notion of equality, we're going to need more definitions...

Subway Map

- Now that we understand the basic structure of λ-exprs, it's time to start building the machinery that we'll need for meta theory. First, we need free/bound variables, renaming, and substitution

Free, Binding, And Bound Variables

- There are free, binding, and bound variables. Intuitively:
  - *Bound variables* occur as free variables in subexpressions of lambda abstractions. For example, x is bound in $\lambda x.xx$ but not in $\lambda y.x$
  - *Binding variables* are prefixed with lambdas, like x in $\lambda x.y$
  - *Free variables* are variables not bound, such as x in zxy or $(\lambda y.y)x$
  - The subexpressions of applications are handled recursively
- *A variable can be both free and bound, or neither*
  - For example, x in $x(\lambda x.x)$ is free, binding and bound

Free and Bound Variables, Formal Definition

- The free variables in a λ-expr, using the language of sets:

  1) $\text{free}(x) \stackrel{\text{def}}{=} \{x\}$
  2) $\text{free}(\lambda x.M) \stackrel{\text{def}}{=} \text{free}(M) \setminus \{x\}$
  3) $\text{free}(MN) \stackrel{\text{def}}{=} \text{free}(M) \cup \text{free}(N)$

- The bound variables, on the other hand, can be defined as:

  1) $\text{bound}(x) \stackrel{\text{def}}{=} \{\}$ (i.e. $\emptyset$)
  2) $\text{bound}(\lambda x.M) \stackrel{\text{def}}{=} \text{bound}(M) \cup \{x\}$
  3) $\text{bound}(MN) \stackrel{\text{def}}{=} \text{bound}(M) \cup \text{bound}(N)$

Example on Free Variables

$$
\begin{aligned}
\mathrm{free}(\lambda y.(\lambda x.zy)w) &= \mathrm{free}((\lambda x.zy)w) \setminus \{y\} \\
&= (\mathrm{free}(\lambda x.zy) \cup \mathrm{free}(w)) \setminus \{y\} \\
&= ((\mathrm{free}(zy) \setminus \{x\}) \cup \{w\}) \setminus \{y\} \\
&= (((\mathrm{free}(z) \cup \mathrm{free}(y)) \setminus \{x\}) \cup \{w\}) \setminus \{y\} \\
&= (((\{z\} \cup \{y\}) \setminus \{x\}) \cup \{w\}) \setminus \{y\} \\
&= ((\{z,\ y\} \setminus \{x\}) \cup \{w\}) \setminus \{y\} \\
&= (\{z,\ y\} \cup \{w\}) \setminus \{y\} \\
&= \{z,\ y,\ w\} \setminus \{y\} \\
&= \{z,\ w\}.
\end{aligned}
$$

Real World Examples on Variable Binding

- Consider the following examples:
  - $\exists x.(\varnothing \in x) \wedge (\forall y.(y \in x \rightarrow \{y, \{y\}\} \in x))$
  - $z + (\Sigma_{k \in \mathbb{N} \setminus \{0\}} 6k^{-2}) = z + \pi^2$
  - `int f(int i) {return i + c++;}`
- Variables x, y, k, and `i` are bound. They are parameters
- Variables z, π, and `c` are not bound. Their meaning depends on the context in which they are interpreted in

Free and Bound Variables Quiz

Which variables in the following expressions are free/bound/both/neither (in outermost context)?

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

Free and Bound Variables Quiz

Which variables in the following expressions are free/bound/both/neither (in outermost context)?

– $\lambda x.x$	x is bound and not free. Easy, wasn't it?
– $\lambda x.xx$	x is bound, not free. (y is neither free nor bound)
– $\lambda x.xy$	x is bound, not free. y is free and not bound
– $(\lambda x.y)(\lambda y.x)$	Both x and y have free and bound occurences
– $\lambda x.\lambda y.xy$	Both x and y are bound and not free
– $\lambda x.x(\lambda y.y)$	Both x and y are bound and not free
– $x(\lambda y.y)yz$	x and z are only free. y is both free and bound

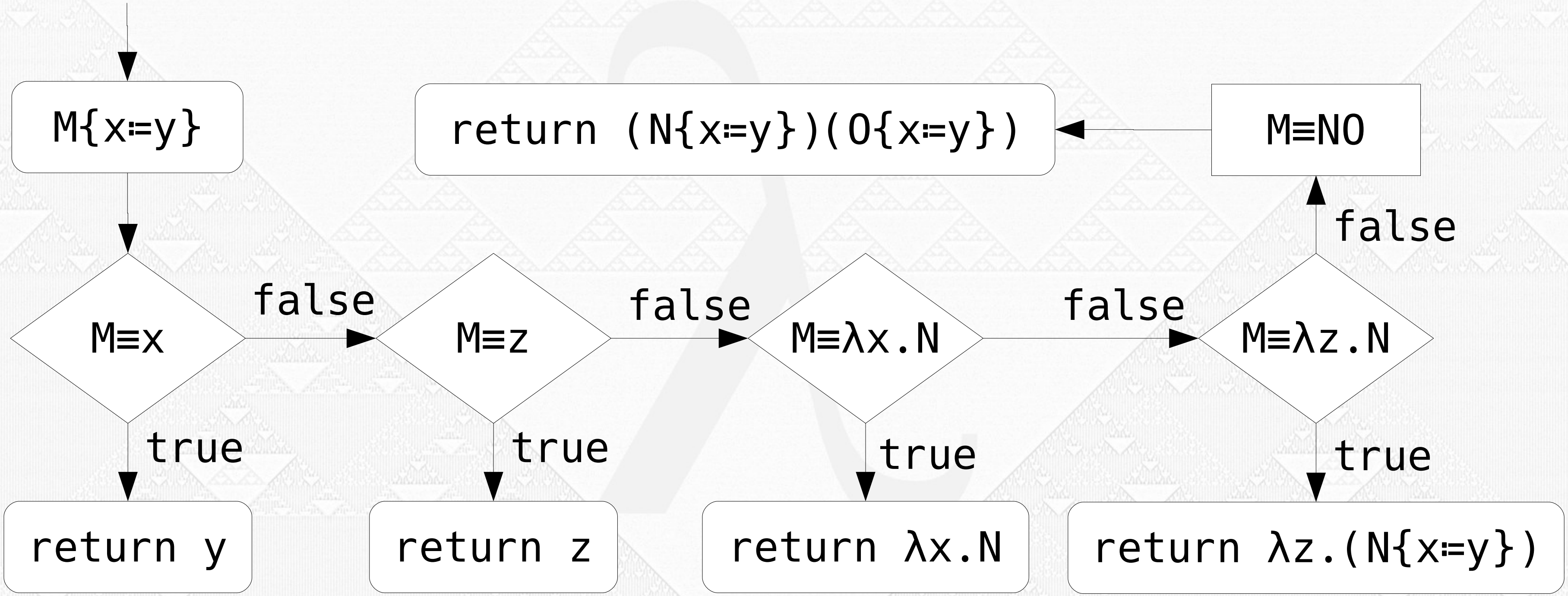
Renaming Variables

- Let M be a λ-expr and y be a variable symbol. Then,

  1) $x\{x{:=}y\} \stackrel{\text{def}}{=} y$
  2) $z\{x{:=}y\} \stackrel{\text{def}}{=} z$ (with $z \not\equiv x$)
  3) $(\lambda x.N)\{x{:=}y\} \stackrel{\text{def}}{=} \lambda x.N$
  4) $(\lambda z.N)\{x{:=}y\} \stackrel{\text{def}}{=} \lambda x.(N\{x{:=}y\})$ (with $z \not\equiv x$)
  5) $(NO)\{x{:=}y\} \stackrel{\text{def}}{=} (N\{x{:=}y\})(O\{x{:=}y\})$

- For example, $(\lambda z.xy(\lambda x.x)(\lambda y.x)z)\{x{:=}y\} \equiv \lambda z.yy(\lambda x.x)(\lambda y.y)z$

Renaming Flowchart



Renaming Quiz

- Are the following assertions correct or not?
  - $(\lambda x.x)\{x{:=}y\} \equiv \lambda y.y$
  - $(\lambda y.x)\{x{:=}y\} \equiv \lambda y.y$
  - $(\lambda y.y)\{x{:=}y\} \equiv \lambda y.y$
  - $(\lambda xy.z)\{x{:=}y\} \equiv \lambda xy.y$
  - $(\lambda xx.x)\{x{:=}y\} \equiv \lambda xy.y$
  - $(\lambda xy.x)\{x{:=}y\} \equiv \lambda xy.y$

Renaming Quiz

- Are the following assertions correct or not?
  - $(\lambda x.x)\{x{:=}y\} \equiv \lambda y.y$ No. x is bound.
  - $(\lambda y.x)\{x{:=}y\} \equiv \lambda y.y$ Yes. x is free.
  - $(\lambda y.y)\{x{:=}y\} \equiv \lambda y.y$ Yes. x does not occur in $\lambda y.y$.
  - $(\lambda xy.z)\{x{:=}y\} \equiv \lambda xy.y$ No. z is not being renamed.
  - $(\lambda xx.x)\{x{:=}y\} \equiv \lambda xy.y$ No. x is bound by the inner λ.
  - $(\lambda xy.x)\{x{:=}y\} \equiv \lambda xy.y$ No. x is bound by the outer λ.

Alpha Congruence

- A λ-expr $M \equiv \lambda x.N$ is *alpha congruent/convertible* with $M' \equiv \lambda y.N\{x{:=}y\}$, if and only if y does not occur (at all) in N
  - The assertion of alpha congruence is denoted with $M \equiv_\alpha M'$
  - We also consider expressions having congruent subexpressions to be congruent, i.e. if $M \equiv_\alpha M'$, then $aM\beta \equiv_\alpha aM'\beta$ (a or β may be “”)
- Following the custom in LC, we focus on λ-exprs *modulo alpha congruence* (i.e. as representatives of equivalence classes of $\equiv_\alpha$), meaning that if $M \equiv_\alpha N$, then we usually write just $M \equiv N$

Can You Convert These?

- Are the following pairs of expressions alpha congruent?
  - $\lambda x.x$ and $\lambda y.y$
  - $\lambda x.x$ and $\lambda x.y$
  - $\lambda x.x$ and $\lambda y.x$
  - $\lambda xy.x$ and $\lambda yx.x$
  - $\lambda xy.x$ and $\lambda yx.y$
  - $\lambda xy.xy$ and $\lambda zy.zy$
  - $\lambda xy.xy$ and $\lambda yx.yx$

Can You Convert These?

- Are the following pairs of expressions alpha congruent?
  - $\lambda x.x$ and $\lambda y.y$ Yes. $\lambda x.x \equiv_\alpha \lambda y.x[x{:=}y] \equiv_\alpha \lambda y.y$
  - $\lambda x.x$ and $\lambda x.y$ No. Different variable bindings
  - $\lambda x.x$ and $\lambda y.x$ No. Different variable bindings
  - $\lambda xy.x$ and $\lambda yx.x$ No. $\lambda xy.x \equiv \lambda x.\underline{\lambda y.x} \not\equiv \lambda y.\underline{\lambda x.x} \equiv \lambda yx.x$
  - $\lambda xy.x$ and $\lambda yx.y$ Yes. $\lambda xy.x \equiv \lambda zy.z \equiv \lambda zx.z \equiv \lambda y.xy$
  - $\lambda xy.xy$ and $\lambda zy.zy$ Yes. $\lambda xy.xy \equiv \lambda x.\lambda y.xy \equiv \lambda z.\lambda y.zy \equiv \lambda zy.zy$
  - $\lambda xy.xy$ and $\lambda yx.yx$ Yes. $\lambda xy.xy \equiv \lambda xz.xz \equiv \lambda yz.yz \equiv \lambda yx.yx$

Substitution of Expressions

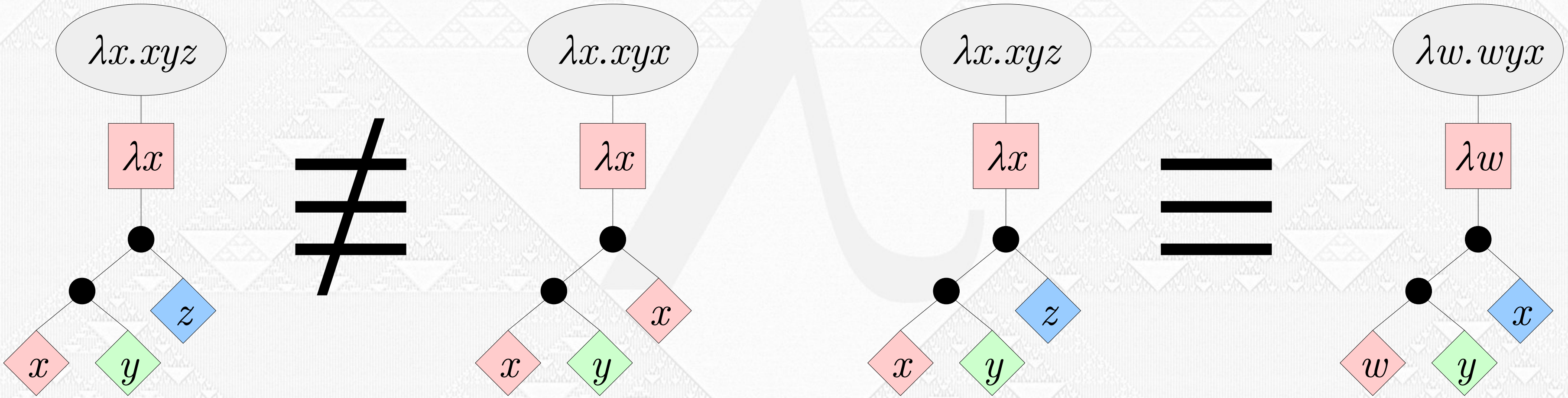
- $M[x{:=}N]$ denotes "*M with all free occurrences of x replaced with N, after renaming bound variables of M if necessary*"
- The formal definition of substitution is technical. Intuitively:
  1) free variables must remain free;
  2) bound variables must remain bound;
  3) same variables must remain same; and
  4) different variables must remain different.

Perils of Careless Substitution

- Substitution is surprisingly non-trivial business because variable capture needs to be avoided
- *Variable capture* is a violation of the rules on the previous slide
  - Note that y captured x on p. 60! That's why we have the extra condition on p. 67 demanding that we always pick *fresh* variables
- For example, if $(\lambda x.y)[x{:=}y]$ yielded $\lambda y.y$, the free variable y would become bound. (Thus, it gives $\lambda x.y$ back unchanged)
  - Other examples include $(\lambda x.y)[y{:=}x]$, $(\lambda x.yz)[x{:=}z]$, and $(\lambda x.yz)[y{:=}z]$

How to Avoid Variable Capture

- Wrong $(\lambda x.xyz)[z{:=}x]$:
  - Naïve substitution
- Correct $(\lambda x.xyz)[z{:=}x]$:
  - Rename x; Then substitute

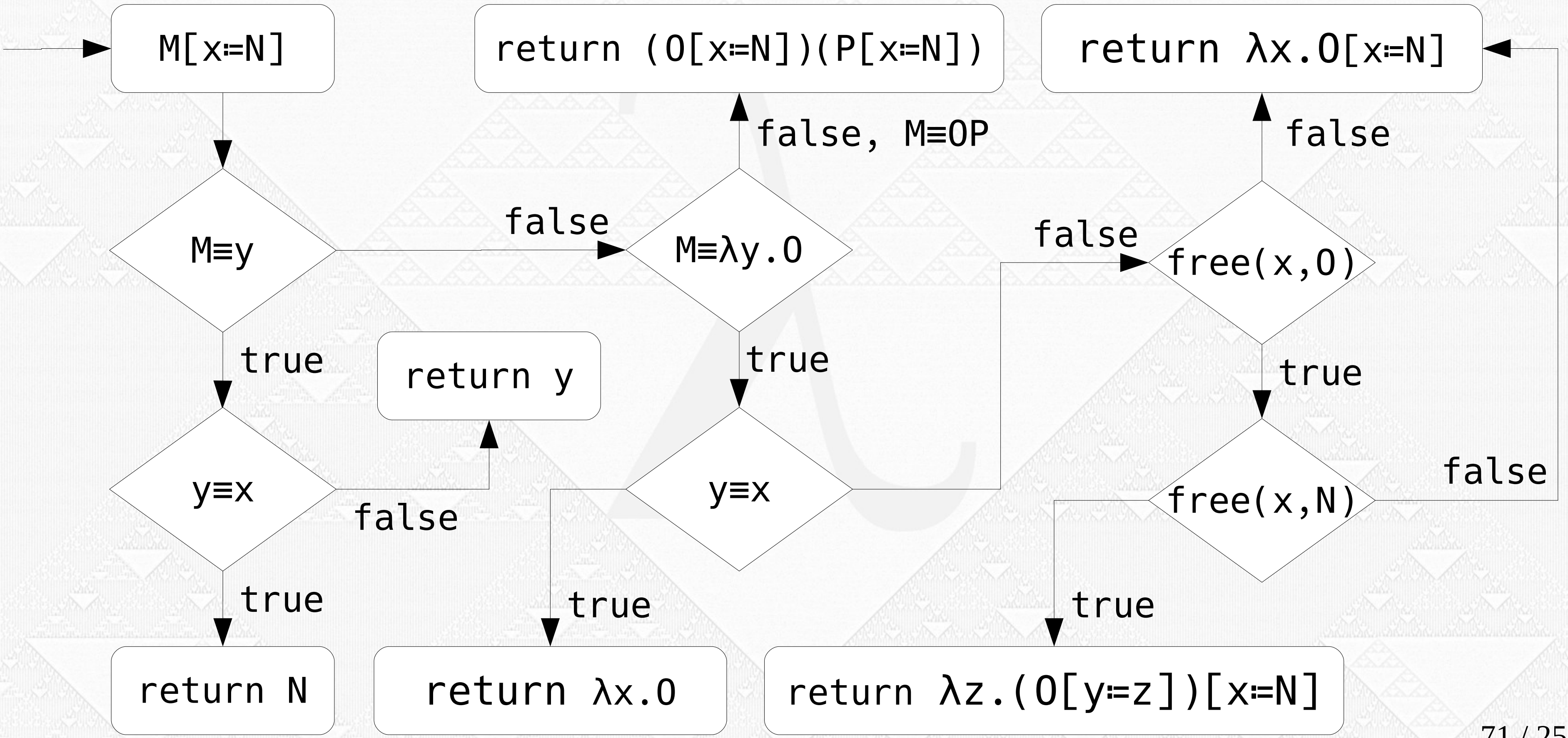


Formal Definition of Substitution

- Formally, we define $M[x{:=}N]$ by cases on M:

1a)	$x[x{:=}N]$	$\stackrel{\text{def}}{=} N$	
1b)	$y[x{:=}N]$	$\stackrel{\text{def}}{=} y$	(if $y \not\equiv x$, o/w case 1a applies)
2)	$(\lambda x.O)[x{:=}N]$	$\stackrel{\text{def}}{=} \lambda x.O$	(x isn't free in $\lambda x.O$)
3a)	$(\lambda y.O)[x{:=}N]$	$\stackrel{\text{def}}{=} \lambda y.O[x{:=}N]$	(if $x \notin \mathrm{free}(O)$ or $x \notin \mathrm{free}(N)$)
3b)	$(\lambda y.O)[x{:=}N]$	$\stackrel{\text{def}}{=} \lambda z.(O[y{:=}z])[x{:=}N]$	(with z being *fresh*)
4)	$(OP)[x{:=}N]$	$\stackrel{\text{def}}{=} O[x{:=}N]P[x{:=}N]$	(recursive case)

Substitution Flowchart



Substitution Quiz

What is the result of these substitutions? (Why?)

- $y[x{:=}y]$
- $x[x{:=}y]$
- $(xy)[x{:=}y]$
- $(\lambda y.x)[x{:=}y]$
- $(\lambda y.y)[x{:=}y]$
- $(\lambda y.x)[x{:=}(\lambda z.z)]$
- $(\lambda y.x)[x{:=}(\lambda x.x)]$

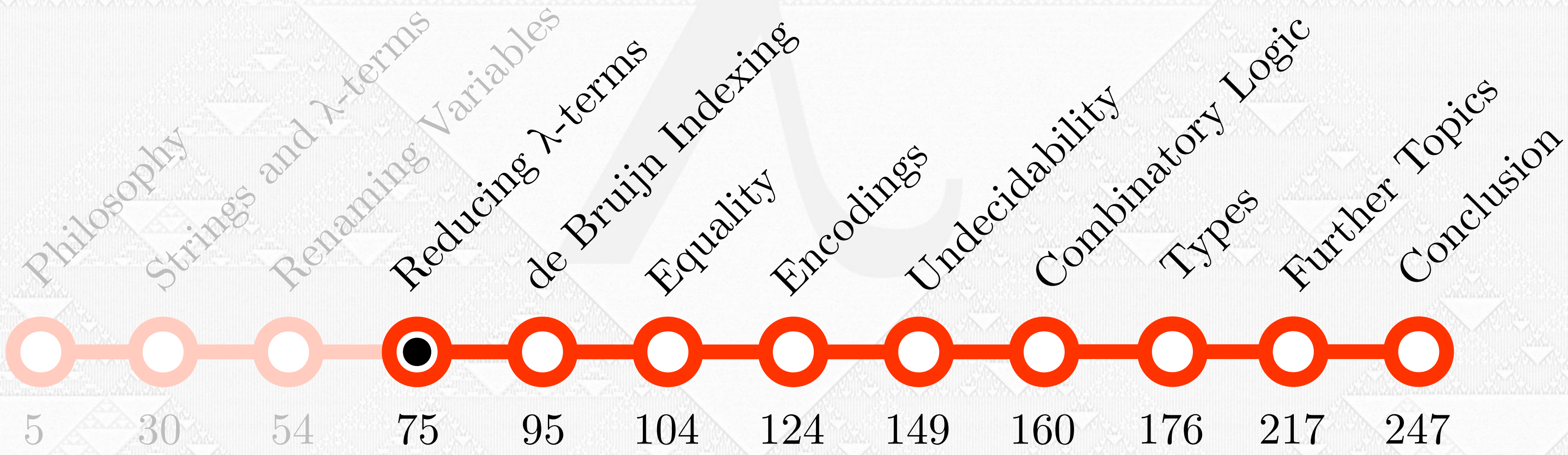
Substitution Quiz

What is the result of these substitutions? (Why?)

- $y[x{:=}y]$ — y. There's no x to be replaced
- $x[x{:=}y]$ — y. This is a basic substitution
- $(xy)[x{:=}y]$ — xy. Otherwise, y would capture x
- $(\lambda y.x)[x{:=}y]$ — $\lambda z.y$. The bound variable was renamed
- $(\lambda y.y)[x{:=}y]$ — $\lambda y.y$. No x present
- $(\lambda y.x)[x{:=}(\lambda z.z)]$ — $\lambda y.\lambda z.z$
- $(\lambda y.x)[x{:=}(\lambda x.x)]$ — $\lambda y.\lambda x.x$. x is not free in $\lambda x.x$

Subway Map

- Now that we've endured most of the gory technical details, we can take the next step towards defining equality of λ-exprs. Equality is one of the most interesting questions in LC and TT



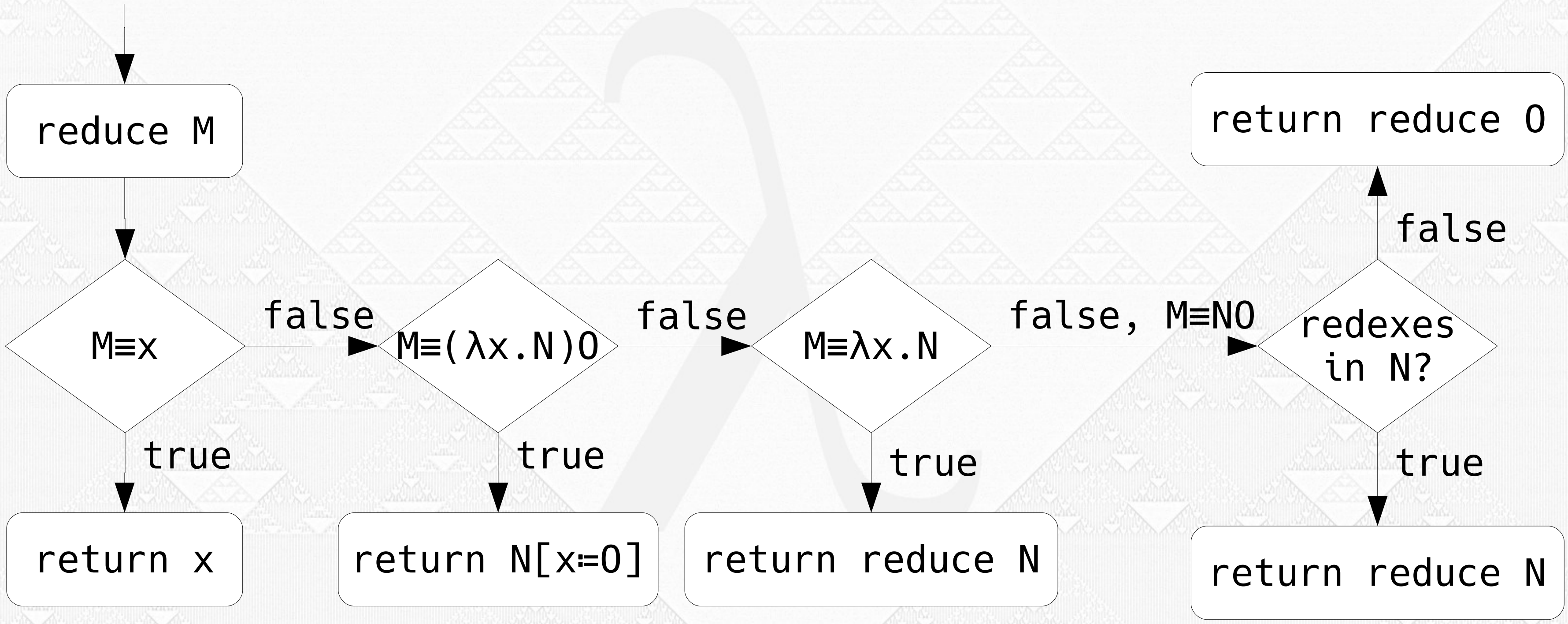
Beta Reducible Expressions

- From now on, were going to be naïve. We will only work with expressions for which substitution works easily
- A lambda expression is *beta reducible expression* (β-redex) if (and only if) it's of the following form:
  - $(\lambda x.M)N$ (where x is a variable symbol and M, N are λ-expr)
- A λ-expr is in *β-Normal Form* (β-NF) if and only if it doesn't contain any β-redexes.

Beta Reduction

- Let $M \equiv (\lambda x.N)\,O$ and $M' \equiv N[x{:=}O]$ be expressions. The rule of *(beta) reduction* says that:
  - *redex* M *reduces to reduct* M'. This is denoted with $M \to M'$
  - We also say that $\alpha M \beta$ reduces to $\alpha M' \beta$. (α or β can be empty.)
- This means, that the computer has to:
  - Take $(\lambda x.N)\,O$ (A β-reducible expression)
  - Give $[x{:=}O]N$ (Drop "$\lambda x.$"; Substitute free 'x's with Os)

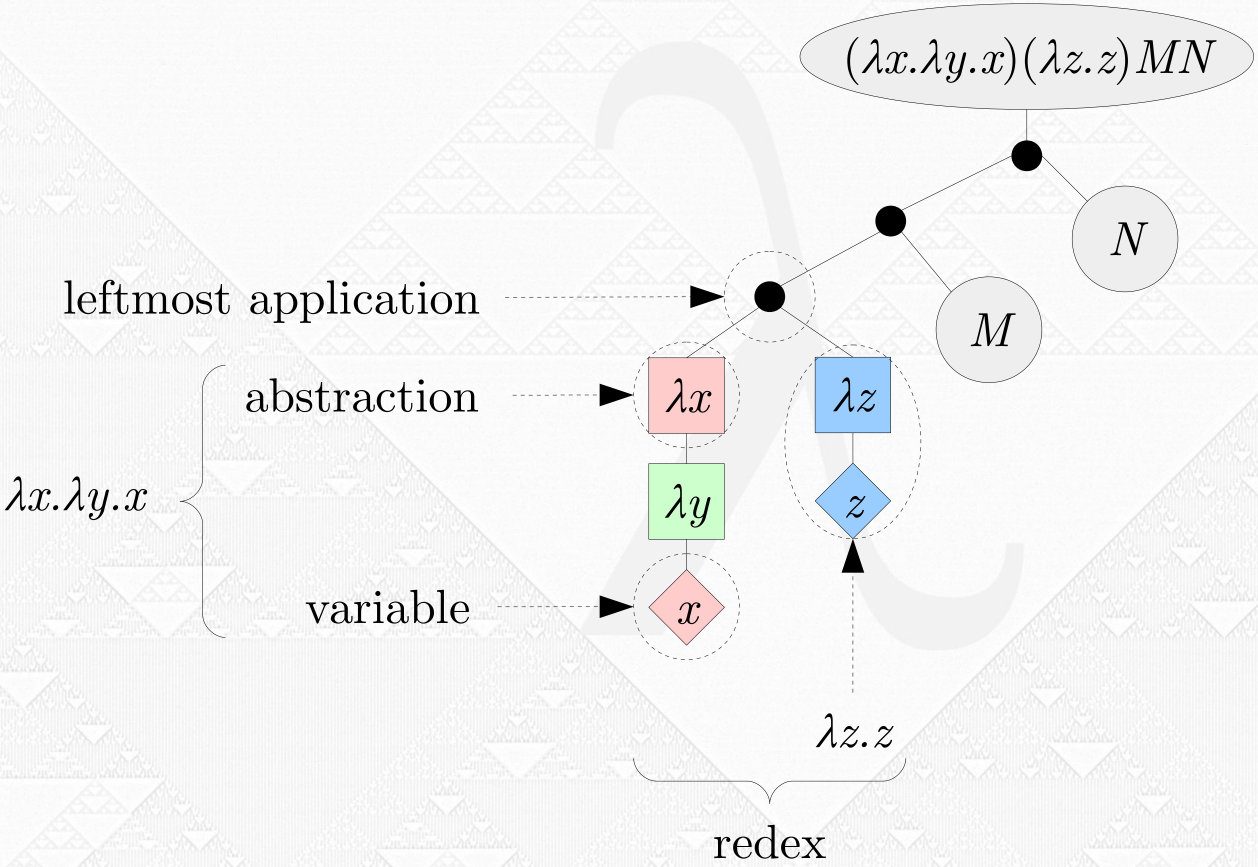
Reduction Flowchart (Single Step)



The Fun Is About to Begin

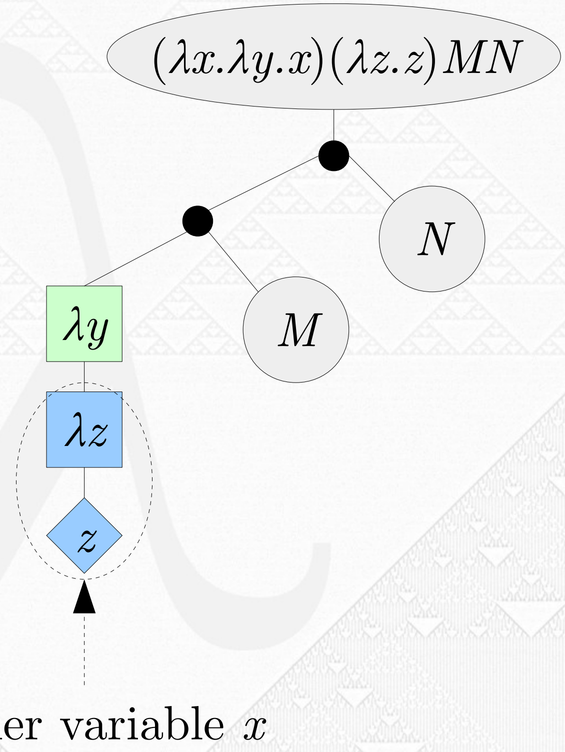
- Performing a single step of reduction is like performing one arithmetic operation, or one step of logical inference.
  - Usually, we need many steps to fully reduce long expressions
- Reducing complex expressions comes down to repeatedly applying beta reduction until a β-NF is reached
- If M reduces to N in zero or more steps, we write $M \twoheadrightarrow N$.
  - ($\twoheadrightarrow$ is the transitive-reflexive closure of $\rightarrow$)

Graph(ical) Example



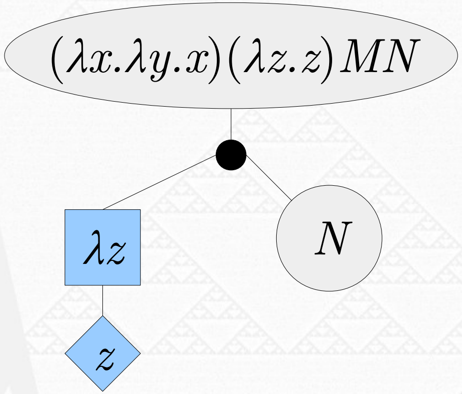
Graph(ical) Example

- "$\lambda x.$" was removed and $\lambda z.z$ replaced x.
- In programming jargon, the *formal parameter* x was *evaluated* with the *actual parameter* $\lambda z.z$.



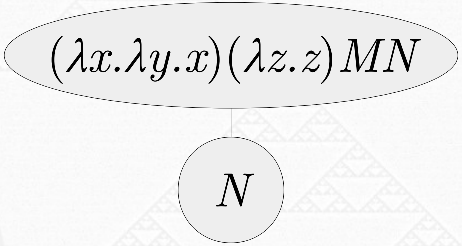
Graph(ical) Example

- Next, "$\lambda y.$" was removed.
- Because y wax not free in the subexpression $\lambda z.z$, the applicand M was thrown away during the reduction process.
- One more step to go.



Graph(ical) Example

- So, $(\lambda x.\lambda y.x)(\lambda z.z)MN \twoheadrightarrow N$.
- The result didn't depend on M or N in any way, because LC is referentially transparent.
  - Cf. the C language example in p. 22–23.



Formal Example

- Let's consider $M \stackrel{\text{def}}{=} (\lambda vw.v)xy$. It holds that $M \twoheadrightarrow x$:
- $$\begin{aligned} M &\equiv (\lambda vw.v)xy \\ &\equiv ((\lambda v.\lambda w.v)x)y \\ &\to ((\lambda w.v)[v{:=}x])y \\ &\equiv (\lambda w.x)y \\ &\to x[w{:=}y] \\ &\equiv x \end{aligned}$$
- Thus, $M \twoheadrightarrow x$ (in two steps).

Numerical Example

- Intuitively, we know what '+', '–', '·', '2', '3', '5', '7', "10", "35", "32", and "42" are.
- We'll learn how to define these things in a way that makes *even machines able operate on them.*
  - The trick is called *recursion.*

$$(\lambda x.\lambda y.2\cdot x+x\cdot y-3)\ 5\ 7$$

$$\to\ (\lambda y.2\cdot 5+5\cdot y-3)\ 7$$

$$\to\ (\lambda y.10+5\cdot y-3)\ 7$$

$$\to\ 10+5\cdot 7-3$$

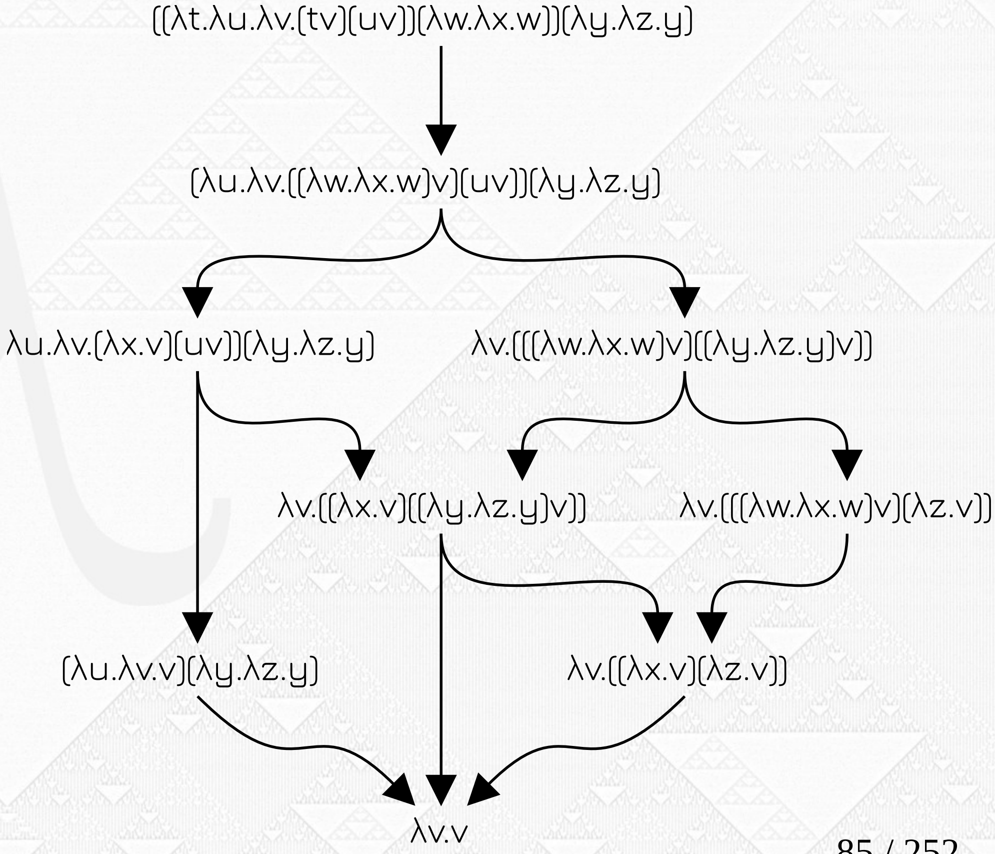
$$\to\ 10+35-3$$

$$\to\ 10+32$$

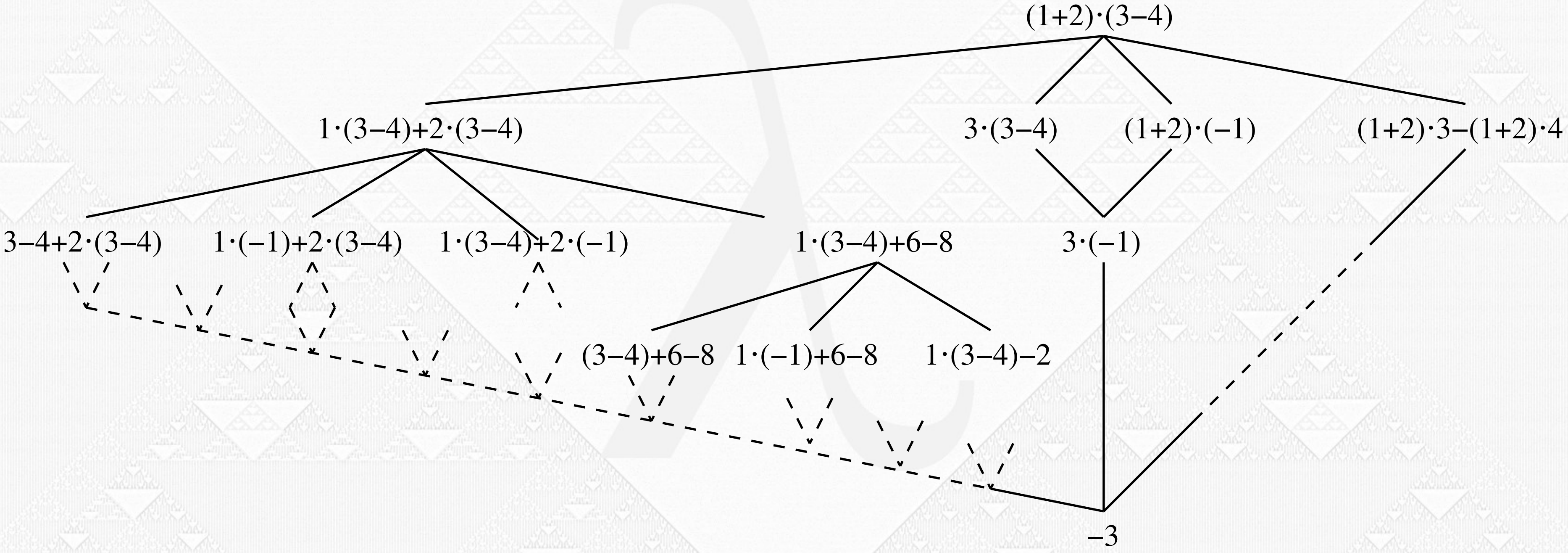
$$\to\ 42$$

More graphs

- The graph to the right shows all the possible reducts of one particular λ-expr.
- Note that every path ends to the same normal form ($\lambda v.v$).
  - This is not a coincidence!
  - (It follows from the Church-Rosser Theorem.)



Algebraic Analogy



Reduction Quiz, Part I

1) The β-NF of $(\lambda x.xx)y$ is...
   a) xx
   b) yy
   c) neither

2) The β-NF of $w(\lambda x.xz)y$ is...
   a) the expression itself
   b) wyz
   c) wzy

Reduction Quiz, Part I

1) The β-NF of $(\lambda x.xx)y$ is…

a) $\cancel{xx}$

b) yy

c) ~~neither~~

By definition,

$$(\lambda x.xx)y \to xx[x{:=}y] \equiv yy,$$

so we throw away "$\lambda x.$" and substitute both xs with y.

2) The β-NF of $w(\lambda x.xz)y$ is…

a) the expression itself

b) $\cancel{wyz}$

c) $\cancel{wzy}$

$w(\lambda x.xz)y \equiv (w(\lambda x.xz))y$, so $\lambda x.xz$ cannot be applied to y. On the other hand, w is just a variable. Thus, the expression is in β-NF.

Reduction Quiz, Part II

3) What is the β-NF of $(\lambda w.w)(\lambda x.z)(((\lambda x.x)y)(\lambda x.xz)yw)$?
   a) $(\lambda w.w)z$
   b) $(\lambda x.z)(((\lambda x.x)y)(\lambda x.xz)yw)$
   c) neither

4) The β-NF of $v(\lambda x.z)$ is...
   a) the expression itself
   b) z
   c) neither

Reduction Quiz, Part II

3) What is the β-NF of $(\lambda w.w)(\lambda x.z)(((\lambda x.x)y)(\lambda x.xz)yw)$?

a) $\cancel{(\lambda w.w)z}$

b) $\cancel{(\lambda x.z)(((\lambda x.x)y)(\lambda x.xz)yw)}$

c) neither

The first two functions from the left are identity and a constant function, so we get z in 2 steps.

4) The β-NF of $v(\lambda x.z)$ is...

a) the expression itself

b) $\cancel{z}$

c) ~~neither~~

$v(\lambda x.z)$ is in normal form, so it cannot be reduced to anything else. N.B. (b) is wrong, because *the operations don't commute*.

Reduction Quiz, Part III

5) The β-NF of $w(x(\lambda y.wz))$ is...

a) $w(x(\lambda x.wz))$

b) $w(x(\lambda w.wz))$

c) $w(x(\lambda z.wz))$

6) The β-NF of $(\lambda x.xx)(\lambda x.xx)$ is...

a) the expression itself

b) $(\lambda x.xx)$

c) neither

Reduction Quiz, Part III

5) The β-NF of $w(x(\lambda y.wz))$ is...

a) $w(x(\lambda x.wz))$

b) $\cancel{w(x(\lambda w.wz))}$

c) $\cancel{w(x(\lambda z.wz))}$

$w(x(\lambda y.wz)) \equiv_\alpha w(x(\lambda x.wz))$, so we consider them identical. There's no variable capture, since x, y are not free in wz.

6) The β-NF of $(\lambda x.xx)(\lambda x.xx)$ is...

a) ~~the expression itself~~

b) $\cancel{(\lambda x.xx)}$

c) neither

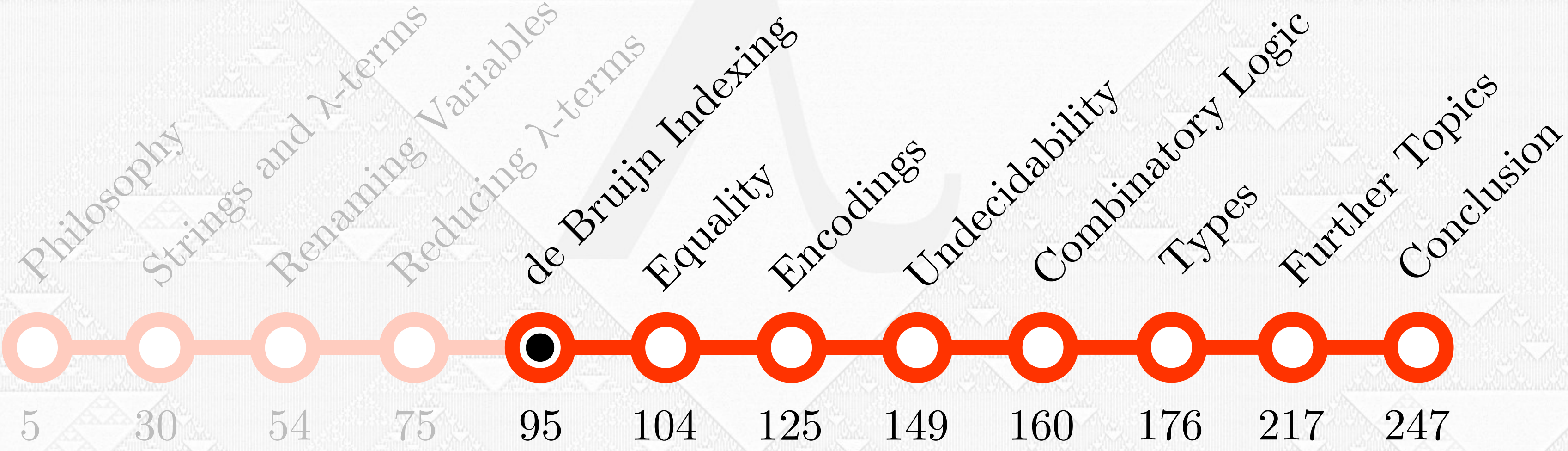
$(\lambda x.xx)(\lambda x.xx) \to (\lambda x.xx)(\lambda x.xx)$. Uh, oh! This expression reduces itself, so it's a β-redex that cannot be reduced!

A Quick Recapitulation

- Let's have a short recap on the (meta) notation:
  - $V \stackrel{\text{def}}{=} W$ defines a meta variable V that refers to W;
  - $V \stackrel{\text{def}}{=}$ "wzq" defines V to be a reference to the string literal "wzq";
  - $M[x{:=}N]$ the λ-expr obtained from M by substituting x with N;
  - $M \equiv N$ asserts that M and N refer to (alpha) congruent λ-exprs;
  - $M \to N$ asserts that M reduces to N in single step; and
  - $M \twoheadrightarrow N$ asserts that M reduces to N in any number of steps.

Subway Map

- We have seen that β-reduction, though simple on the surface, contains some complexity in the underlying machinery. We'll have a look at an arguably simpler alternative notation next



Alternative notations

- We’ve seen that named free and bound variables lead into some uncomfortable technicalities
- There are two alternatives to the ordinary LC notation that bypass some of these challenges, namely Combinatory Logic (CL) and de Bruijn Indexing
  - However, they come with their own limitations
  - We’ll discuss de Bruijn Indexing next

de Bruijn Indexing

- *de Bruijn Indexing* (dB-exprs) is an alternative syntax to LC
- Let n be a natural number and M, N be dB-exprs. Then,

1)	(n)	is a dB-expr;
2)	(MN)	is a dB-expr;
3)	(λM)	is a dB-expr; and
4)	nothing else	is a dB-expr;

- We'll apply syntactic sugar, e.g. $\lambda\lambda 0\ 2 \equiv (\lambda(\lambda((0)(2))))$

Syntactical Correspondence

- The idea of de Bruijn Indexing is that the natural numbers represent bound variables by expressing their distance to the binding lambda abstraction, with 0 meaning immediate binding
  - E.g. The λ-exprs $\lambda x.x$, $\lambda xy.x$, and $\lambda xyz.(\lambda w.w)xz(yz)$ would translate into the dB-exprs $\lambda 0$, $\lambda\lambda 1$, and $\lambda\lambda\lambda(\lambda 0)2\ 0(1\ 0)$ respectively. (2 0 is 2 applied to 0, 20 is number twenty)
- Free variables can be represented with sufficiently large numbers
  - E.g. $\lambda xy.z(\lambda w.w)$ translates into $\lambda\lambda 2(\lambda 0)$ (or e.g. $\lambda\lambda 7(\lambda 0)$)

de Bruijn Indexing Quiz

- Do the following pairs of expressions correspond?
  - $\lambda x.x$ and $\lambda 0$
  - $\lambda y.y$ and $\lambda 0$
  - $\lambda x.y$ and $\lambda 0$
  - $\lambda x.y$ and $\lambda 1$
  - $(\lambda xy.z)(\lambda v.w)$ and $(\lambda\lambda 7)(\lambda 1)$
  - $\lambda u.(\lambda xy.z)(\lambda v.w)$ and $\lambda(\lambda\lambda 7)(\lambda 1)$

de Bruijn Indexing Quiz

- Do the following pairs of expressions correspond?
  - $\lambda x.x$ and $\lambda 0$ Yes.
  - $\lambda y.y$ and $\lambda 0$ Yes.
  - $\lambda x.y$ and $\lambda 0$ No, y is free.
  - $\lambda x.y$ and $\lambda 1$ Yes.
  - $(\lambda xy.z)(\lambda v.w)$ and $(\lambda\lambda 7)(\lambda 1)$ Yes. (z could be also 2.)
  - $\lambda u.(\lambda xy.z)(\lambda v.w)$ and $\lambda(\lambda\lambda 7)(\lambda 1)$ No, w becomes bound.

Beta-Reduction With de Bruijn Indexing

- I paraphrase the definition of β-reduction of a dB-redex $(\lambda M)N$, given in https://en.wikipedia.org/wiki/De_Bruijn_index (viewed in 2020-01-24):
  1) Find the indices n_1, n_2, … n_k corresponding to the variables bound by the abstraction of the beta redex
  2) Decrement the indices of the free variables in M by one
  3) Substitute each n_i, with N_i, where N_i is N with the indices of free variables incremented suitably to avoid binding

de Bruijn Indexing in Action

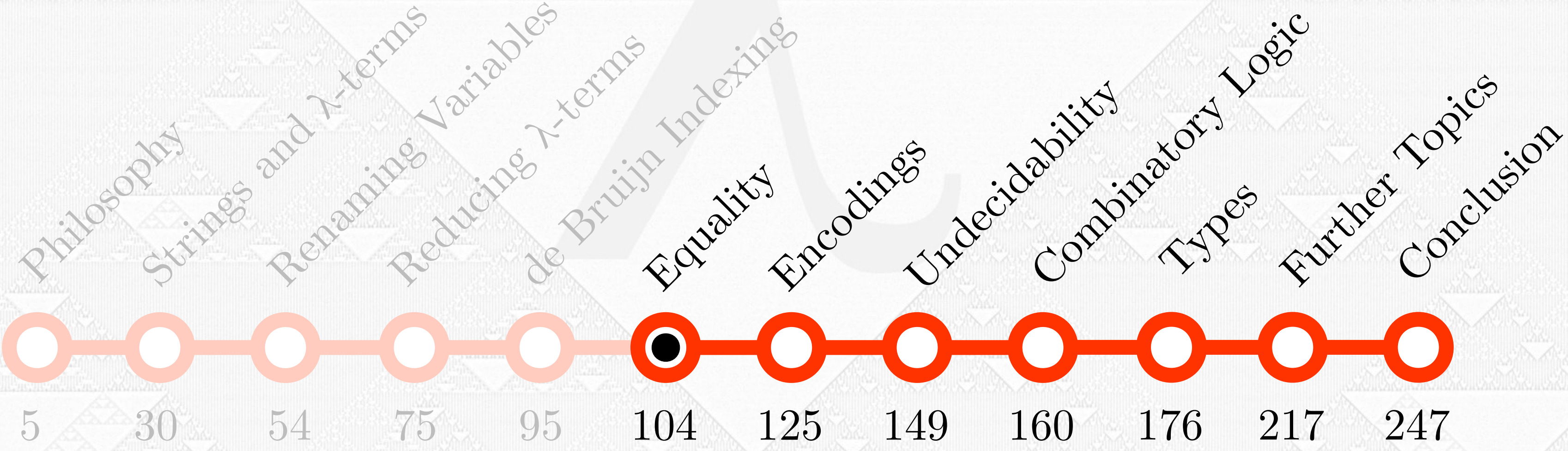
- Consider the example from Wikipedia (see previous slide):
  - $(\underline{\lambda x}.\lambda y.z\underline{x}(\lambda u.u\underline{x}))(\lambda x.wx)$; which is
  - $(\underline{\lambda}\ \lambda\ 3\ \underline{1}\ (\lambda\ 0\ \underline{2}))\ (\lambda\ 4\ 0)$ as a dB-expr
- We decrement the free variable, yielding $(\underline{\lambda}\ \lambda\ 2\ \underline{1}\ (\lambda\ 0\ \underline{2}))$
- We reduce the expression while increasing the index 4 by the number of λs in the new scope of the blue expression, yielding $(\lambda\ 3\ (\lambda\ 5\ 0)\ (\lambda\ 0\ (\lambda\ 6\ 0)))$, i.e. $(\lambda y.z(\lambda x.wx)(\lambda u.u(\lambda x.wx)))$

Pros and Cons of de Bruijn Indexing

- de Bruijn indexing may be less human-readable than the standard notation of LC
- On the other hand, de Bruijn indexing can be used to partition standard λ-exprs into α-congruence classes
- de Bruijn indexing can be useful for building interpreters or compilers of LC-like languages
- Then again, reducing dB-exprs may not be easier for humans

Subway Map

- We now move back to the classical LC. We're ready to define the equality of λ-exprs, and discuss the consequences



Checkpoint

- We have seen lots of different topics and alternative definitions
- Before moving beyond this point, we need to understand
  - λ-expressions;
  - α-conversion;
  - α-congruence;
  - β-reduction; and
  - β-normal forms

Equality of Lambda Expressions

- We're finally ready to formulate what is perhaps the main question in LC: *Equality* of lambda expressions
- Given any two λ-exprs M and N, if
  1) $M \equiv N$ (actually included in (2) and (3));
  2) $M \equiv_\alpha N$; or
  3) $M \twoheadrightarrow N$ or $N \twoheadrightarrow M$;

  then M and N are said to be *equal*, denoted with $M = N$.

Example Equation

- Consider $(\lambda xy.y)z$ and $(\lambda x.x)(\lambda w.w)$
  1) $(\lambda xy.y)z \to (\lambda y.y)$, so $(\lambda xy.y)z = (\lambda y.y)$
  2) $(\lambda y.y) \equiv_\alpha (\lambda w.w)$, so $(\lambda y.y) = (\lambda w.w)$
  3) $(\lambda x.x)(\lambda w.w) \to (\lambda w.w)$, so $(\lambda w.w) = (\lambda x.x)(\lambda w.w)$
  4) By transitivity (twice), $(\lambda xy.y)z = (\lambda x.x)(\lambda w.w)$
- However, $(\lambda xy.y) \neq (\lambda x.x)$ and $z \neq (\lambda w.w)$
- Thus, *equal expressions may have non-equal subexpressions*

Equality Quiz

- Are the following pairs of λ-exprs equal?
  - $\lambda x.x$ and $\lambda xy.xy$
  - $\lambda x.x$ and $\lambda x.xx$
  - $\lambda x.(\lambda x.x)x$ and $\lambda x.(\lambda y.y)x$
  - $\lambda x.\lambda x.xx$ and $\lambda x.\lambda y.yx$
  - $(\lambda x.xx)(\lambda y.y)$ and $\lambda y.y$
  - $\lambda y.(\lambda x.f(gx))(hy)$ and $\lambda y.f((\lambda x.g(hx))y)$

Equality Quiz

- Are the following pairs of λ-exprs equal?
  - $\lambda x.x$ and $\lambda xy.xy$ — No, not intensionally
  - $\lambda x.x$ and $\lambda x.xx$ — No
  - $\lambda x.(\lambda x.x)x$ and $\lambda x.(\lambda y.y)x$ — Yes, α-congruent
  - $\lambda x.\lambda x.xx$ and $\lambda x.\lambda y.yx$ — No
  - $(\lambda x.xx)(\lambda y.y)$ and $\lambda y.y$ — Yes, redex$\twoheadrightarrow$reduct
  - $\lambda y.(\lambda x.f(gx))(hy)$ and $\lambda y.f((\lambda x.g(hx))y)$ — Yes, same β-nf

A Semantical Analogy

- If f is a (possibly constant) function (or program) of x and its value can be interpreted as a λ-expr M; then
  1) $f(x)$ is $\lambda x.M$ (“take x as a parameter”); and
  2) $f(c)$ is $(\lambda x.M)\,c$ (“apply f to c”); so
  3) $\lambda x.M$ is like a program, which transforms x into M; and
  4) if $M[x{:=}c] \twoheadrightarrow N$ and N is a β-NF, then the program halts and produces N as its output
  5) if $M[x{:=}c]$ has no β-NF, then the program never halts

Referential Transparency

- A very important feature of LC is *compositionality*, a.k.a. *referential transparency* in computer science speak
  - The meaning of a lambda expression is entirely determined by the meanings of its subexpressions
  - In LC, *the whole is, no more, no less, than the sum of its parts*
  - $M = N$, if and only if $\alpha M \beta = \alpha N \beta$. This is *always* true in LC
- Compositionality is taken as granted in mathematics, but only the most elite functional programming languages can deliver it

Confluence of Beta Reduction

- Church-Rosser Theorem is perhaps the central result in LC:
  - If $M \twoheadrightarrow N$ and $M \twoheadrightarrow N'$, then there is O s.t. $N \twoheadrightarrow O$ and $N' \twoheadrightarrow O$
- This means that if a normal form exists, it is unique and reachable through (iterated) reduction.
  - For example, consider the graphs in p. 85–86
  - If the NF is reachable, then the order of reductions is irrelevant to the outcome
  - Cf. (1+2)·(3–4) = 3·(3–4) = (1+2)·(–1) = 3·(–1) = –3

Confluence as Diagrams

$$
\begin{tikzcd}
 & M \arrow[dl] \arrow[dr] & \\
N & & N'
\end{tikzcd}
\Rightarrow
\left(
\begin{tikzcd}
 & M \arrow[dl] \arrow[dr] & \\
N \arrow[dr, dashed] & & N' \arrow[dl, dashed] \\
 & O &
\end{tikzcd}
\land
\begin{tikzcd}
N \arrow[r, phantom, "="] \arrow[dr, dashed] & {} & N' \arrow[dl, dashed] \\
 & O &
\end{tikzcd}
\right)
$$

Side Note: Extensionality

- $(\lambda xy.xy) \neq (\lambda z.z)$, even though $(\lambda xy.xy)MN = MN = (\lambda z.z)MN$ for any M and N
- *Equality implies equal behaviour*
- The converse of the above claim is called *extensionality*
  - The rule of *η-conversion* is that $\lambda x.Mx =_\eta M$ when x is not free in M
  - Using this rule, we see that $\lambda xy.xy \equiv \lambda x.(\lambda y.xy) =_\eta \lambda x.x \equiv_\alpha \lambda z.z$
- We don't need extensionality in this presentation though

Side Note: Delta Conversion

- So far, the use of meta variables hasn't been formally explained
  - "$X \stackrel{\text{def}}{=} M$" translates to "$\Delta \rhd X \triangleq M$" where Δ is a *context*, X is an identifier (*definiendum*) and M is an expression (*definiens*)
  - E.g. $\rhd\ n^2 := n \cdot n$ ("In empty context, n^2 denotes $n \cdot n$")
- Switching between definiendum and definiens is known as *δ-conversion*. It is used in some type systems (e.g. Automath)
  - Hence $(n \cdot n) \cdot (n \cdot n)$ is δ-equal (always interchangeable) with $(n^2)^2$
- α, β, η, and δ-conversions together form *judgemental equality*

Connection Between Computation And Logic

- LC can be seen as a functional programming language
  - It can be used for developing and analyzing algorithms
  - It can be used as the foundational basis for more practical programming languages (e.g. Haskell, Agda, Idris, etc.)
- LC can be also seen as a formal proof system
  - The equivalence of computer programs and logical proofs is a deep mathematical fact, known as the *Curry-Howard Correspondence*
  - However, "truth" is not a concept in LC (but *provability* is)

Evaluation as Deduction

- We'll define `if-then-else` soon, but let's use intuition for now. Our example is the famous Aristotelian syllogism
- We know that every man is mortal, so
  - $P \stackrel{\text{def}}{=} (\lambda x.\texttt{if}\ (\mathbf{Man}\ x)\ \texttt{then}\ (\mathbf{Mortal}\ x)\ \texttt{else}\ \bot)$
- By assumption, Socrates is a man, i.e. **Man Socrates** holds
- Thus, $P(\mathbf{Socrates}) \twoheadrightarrow (\mathbf{Mortal\ Socrates})$, so *reduction is like deduction* using the *modus ponens* rule

Did That Even Make Sense?

- The key difference between LC and predicate logic is that in LC there's no notion of objective truth
  - Instead, LC investigates definability, provability, and solvability
- Also, λ-exprs don't quite seem like the same kind of functions than those encountered in logic or set theory
- Actually, LC does have models that make the connection to set theory clear, but they require rather advanced mathematics that is beyond our scope. Domain theory studies these models

The Notion of Consistency in LC

- For the logicians among the audience, here's the idea of consistency in LC:
  - Two expressions, M and N are *incompatible*, denoted with $M \# N$, if and only if it is possible to derive an arbitrary equation from $M = N$
  - Equivalently, $M \# N$ if $M = N$ implies $O = \lambda x.x$ for any O
  - A *theory*, i.e. an assortment of equations is *consistent* if and only if it doesn't contain an equation $M = N$ such that $M \# N$
  - Such an equation would collapse the universe into a singleton

Standard Combinators

- *Combinator* is a λ-expr without free variables. For example:
  - $\mathbf{I} \stackrel{\text{def}}{=} \lambda x.x$ (Thus, $\mathbf{I}M \twoheadrightarrow M$)
  - $\mathbf{K} \stackrel{\text{def}}{=} \lambda xy.x$ (Thus, $\mathbf{K}MN \twoheadrightarrow M$)
  - $\mathbf{S} \stackrel{\text{def}}{=} \lambda xyz.xz(yz)$ (Thus, $\mathbf{S}MNO \twoheadrightarrow MO(NO)$)
- Actually, **S** and **K** are sufficient for expressing all combinators
  - There is even a single combinator **X** that can express both **S** and **K**!
  - Likewise, there is a *single-instruction Turing-complete computer*!

Using Combinators to Express Others

- $$\begin{aligned} \mathbf{SKK} &\equiv (\lambda xyz.xz(yz))\mathbf{KK} \\ &\to (\lambda yz.\mathbf{K}z(yz))\mathbf{K} \\ &\to (\lambda z.\mathbf{K}z(\mathbf{K}z)) \\ &\equiv (\lambda z.(\lambda tu.t)z(\mathbf{K}z)) \\ &\to (\lambda z.(\lambda u.z)(\mathbf{K}z)) \\ &\to \lambda z.z \\ &\equiv \mathbf{I} \end{aligned}$$
- Because $\mathbf{SKK} \twoheadrightarrow \mathbf{I}$ (in 4 steps), it holds that $\mathbf{SKK} = \mathbf{I}$. ■

Observations On the Proof

- In the previous slide, $\mathbf{S}$ was *partially applied* (i.e. lacked some of its defined argument(s)), so we needed to recall its definition
- The instance of $\mathbf{K}$ that was reduced, was *fully applied*, so we could treat it as a black box, using the fact $\mathbf{K}MN \twoheadrightarrow M$
  - The second instance of $(\mathbf{K}z)$ was discarded completely!
- These kind of situations are common in LC
  - This has implications in lazy functional programming

An Inconsistent Theory

- Suppose that $\mathbf{K} = \mathbf{S}$. For an arbitrary λ-expr M we have

$$\mathbf{KI}(\mathbf{K}M)\mathbf{I} = \mathbf{SI}(\mathbf{K}M)\mathbf{I} \twoheadrightarrow \mathbf{II}(\mathbf{K}M\mathbf{I}) \twoheadrightarrow M, \text{ so } M = \mathbf{KI}(\mathbf{K}M)\mathbf{I}$$

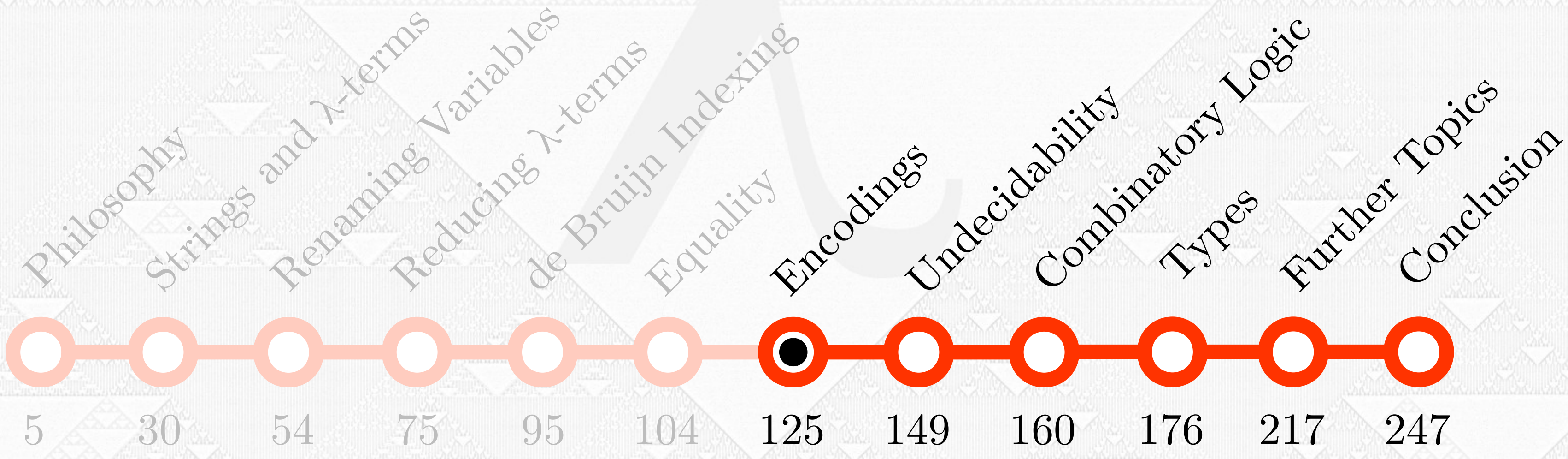
- On the other hand, $\mathbf{KI}(\mathbf{K}M)\mathbf{I} = \mathbf{I}$. By transitivity $M = \mathbf{I}$. (We could already stop here.)
- The previous steps can be repeated for another arbitrary λ-expr M', yielding $M' = \mathbf{I}$. By symmetry and transitivity, $M = M'$
- Therefore, $\mathbf{K} \# \mathbf{S}$, so the theory $\{\mathbf{K} = \mathbf{S}\}$ is inconsistent $\blacksquare$

Side Note: The Notion of Definedness

- A combinator M is *solvable* if and only if there are expressions N_0, N_1, ..., N_k such that $M\ N_0\ N_1\ ...\ N_k \quad = \quad \mathbf{I}$
- Unsolvable expressions can be safely identified. The symbol *bottom*, '$\bot$' is sometimes used to represent an unsolvable expression, or *undefined* value. E.g. $\mathbf{\Omega} \stackrel{\text{def}}{=} (\lambda x.xx)(\lambda x.xx) = \bot$
- Identifying a solvable term with an unsolvable term is inconsistent. E.g. the theory $\{\lambda x.x\mathbf{I}\mathbf{\Omega} = \bot\}$ proves anything!

Subway Map

- We'll investigate the (theoretical) computing aspects of LC next



Programming in LC

- Every computer program can be translated into LC
- LC offers *control structures* for
  1) *composition*;
  2) *decomposition* (or branching); and
  3) *recursion*.
- All (partial) recursive functions, i.e. effectively computable programs can be expressed using these three operations

Function Composition

- $\circ \stackrel{\text{def}}{=} \mathbf{S}(\mathbf{KS})\mathbf{K} \twoheadrightarrow \lambda fgx.f(gx)$, so $\circ MNO \twoheadrightarrow M(NO)$ for all M, N, O
- λ-exprs are *closed under composition*, as shown above
- $\circ(\circ MN)O = \circ M(\circ NO)$, so composition is *associative*
  - (This statement would be $(M\circ N)\circ O = M\circ(N\circ O)$ in infix notation)
- I is the *identity element* of composition
- (Thus, λ-exprs with the composition operation have the algebraic structure of a *monoid*)

Truth Values And Branching

- Truth values and branching as programming concepts can be expressed in terms of combinators
  - $\mathbf{T} \overset{\text{def}}{=} \mathbf{K} \twoheadrightarrow \lambda xy.x \quad (\text{Thus, } \mathbf{T}MN \twoheadrightarrow M)$
  - $\mathbf{F} \overset{\text{def}}{=} \mathbf{KI} \twoheadrightarrow \lambda xy.y \quad (\text{Thus, } \mathbf{F}MN \twoheadrightarrow N)$
- We define, that
  - `if` M `then` N `else` $O \overset{\text{def}}{=} MNO$;
  - provided $M \twoheadrightarrow \mathbf{T}$ or $M \twoheadrightarrow \mathbf{F}$.

Natural Numbers in LC

- For discussing natural numbers, we need the following λ-exprs:
  1) The λ-expr for representing the natural number *zero*: $\mathbf{Z^0}$
  2) For any λ-expr n (natural number), *successor of n*: $\mathbf{S^+}\ n$
  3) *Test for zero* function: $\mathbf{Zero}\ \mathbf{Z^0} = \mathbf{T}$ and $\mathbf{Zero}\ (\mathbf{S^+}\ n) = \mathbf{F}$
  4) The *predecessor* function: $\mathbf{P^-}\ \mathbf{Z^0} = \mathbf{Z^0}$, $\mathbf{P^-}\ (\mathbf{S^+}\ n) = n$
  5) (There is also the constant *zero function* $\mathbf{KZ^0} = \lambda x.\mathbf{Z^0}$)
  - There are at least two encodings for $\mathbf{Z^0}$, $\mathbf{Zero}$, $\mathbf{S^+}$ and $\mathbf{P^-}$

Recursive Function Definitions

- Consider factorial: $0! = 1$, $(n+1)! = (n+1)\cdot(n!)$, $\forall n \in \mathbb{N}$
  - This kind of explicitly recursive definition is not possible in LC
  - λ-exprs are anonymous, so they cannot refer to their own values
- *Fixpoint combinator* is a combinator F, s.t. $FM = M(FM)$ for an arbitrary λ-expr M. (F makes FM a *fixpoint* of M)
  - $\boldsymbol{\Theta}$ is a fixed point combinator
  - Fixed point combinators, together with lambda abstraction, introduce a backdoor that enables recursion...

Self-Application

- Consider $\mathbf{\Omega} \stackrel{\text{def}}{=} (\lambda x.xx)(\lambda x.xx)$
  - It's $\lambda x.xx$ applied to itself!
  - It's not in normal form
  - $\mathbf{\Omega}$ reduces to $\mathbf{\Omega}$!
- Self-application is not possible in Set Theory or most programming languages (for good reasons)

A Fixed Point Combinator

- $\mathbf{\Theta} \stackrel{\text{def}}{=} (\lambda xy.y(xxy))(\lambda xy.y(xxy))$ is another funny expression
- I's called *Turing's Theta Combinator* (after Alan Turing)
- For any lambda expression M, it holds that:

$$
\begin{aligned}
\mathbf{\Theta}\,M &\equiv ((\lambda xy.y(xxy))(\lambda xy.y(xxy)))\,M \\
&\equiv ((\lambda x.\lambda y.y(xxy))(\lambda xy.y(xxy)))\,M \\
&\to (\lambda y.y((\lambda xy.y(xxy))(\lambda xy.y(xxy))\,y))\,M \\
&\to M((\lambda xy.y(xxy))(\lambda xy.y(xxy))\,M) \\
&\equiv M(\mathbf{\Theta}\,M)
\end{aligned}
$$

Let's See a Replay

- $$\begin{aligned} \mathbf{\Theta} M &\equiv ((\lambda xy.y(xxy))(\lambda xy.y(xxy)))M \\ &\equiv ((\underline{\lambda x.}\lambda y.y(\underline{xxy}))\underline{(\lambda xy.y(xxy))})M \\ &\to (\underline{\lambda y.y}((\lambda xy.y(xxy))(\lambda xy.y(xxy))\underline{y}))\underline{M} \\ &\to M((\lambda xy.y(xxy))(\lambda xy.y(xxy))M) \\ &\equiv M(\mathbf{\Theta} M) \end{aligned}$$
- Of course, proofs can be refactored. For example, we could assign a name for the green part or get $\mathbf{\Theta}$ back earlier
  - Also, the first two pairs of blue parentheses were redundant

How To Hack The System To Get Recursion

- Let's say that we want to define function F recursively
  - Firstly, let $F \stackrel{\text{def}}{=} \mathbf{\Theta} E$, so $F \twoheadrightarrow E(\mathbf{\Theta} E) \equiv EF$
- In order to eventually reach β-NF, some condition P is needed:
  - $E \stackrel{\text{def}}{=} \lambda fx.\texttt{if}\ (Px)\ \texttt{then}\ (Gx)\ \texttt{else}\ (Hx(fx))$
  - (n-ary: $\lambda fxy_1 \ldots y_n.\texttt{if}\ (Px)\ \texttt{then}\ (Gxy_1 \ldots y_n)\ \texttt{else}\ (Hx(fxy_1 \ldots y_n)y_1 \ldots y_n)$)
  - We say that F *is defined by (primitive) recursion over* G *and* H
- Thus, $F \twoheadrightarrow \lambda x.\texttt{if}\ (Px)\ \texttt{then}\ (Gx)\ \texttt{else}\ (Hx(Fx))$

Arithmetics

- Let $\underline{0} \approx \mathbf{Z^0}$ and $\underline{n} \approx (\mathbf{S^+} \circ \mathbf{S^+} \circ \mathbf{S^+} \circ \ldots \circ \mathbf{S^+})\ \mathbf{Z^0}$ (with n repetitions of $\mathbf{S^+}$). (We used an infix '$\circ$' for readability.)
  - $\underline{1} \equiv \mathbf{S^+}\ \mathbf{Z^0}$, $\underline{2} \equiv \mathbf{S^+}\ (\mathbf{S^+}\ \mathbf{Z^0})$, $\underline{3} \equiv \mathbf{S^+}\ (\mathbf{S^+}\ (\mathbf{S^+}\ \mathbf{Z^0}))$, etc.
- We can now proceed with:
  - $+ \quad \stackrel{\text{def}}{=} \quad \mathbf{\Theta} X$;
  - $X \quad \stackrel{\text{def}}{=} \quad \lambda fnm.\texttt{if}\ (\mathbf{Zero}\ m)\ \texttt{then}\ n\ \texttt{else}\ R$; and
  - $R \quad \stackrel{\text{def}}{=} \quad (f\ (\mathbf{S^+}\ n)\ (\mathbf{P^-}\ m))$

One Plus One Equals Two

$$
\begin{aligned}
\mathbf{+}\ \underline{1}\ \underline{1} &\equiv \boldsymbol{\Theta} X\ \underline{1}\ \underline{1} \\
&\twoheadrightarrow X(\boldsymbol{\Theta} X)\ \underline{1}\ \underline{1} \\
&\equiv (\lambda fnm.\texttt{if}\ (\mathbf{Zero}\ m)\ \texttt{then}\ n\ \texttt{else}\ R)\ \mathbf{+}\ \underline{1}\ \underline{1} \\
&\twoheadrightarrow \texttt{if}\ (\mathbf{Zero}\ \underline{1})\ \texttt{then}\ \underline{1}\ \texttt{else}\ (\mathbf{+}\ (\mathbf{S}^{+}\ \underline{1})\ (\mathbf{P}^{-}\ \underline{1})) \\
&\twoheadrightarrow \mathbf{F}\ \underline{1}\ (\mathbf{+}\ (\mathbf{S}^{+}\ \underline{1})\ (\mathbf{P}^{-}\ \underline{1})) \\
&\twoheadrightarrow \mathbf{+}\ (\mathbf{S}^{+}\ \underline{1})\ \mathbf{Z}^{0} \\
&\twoheadrightarrow \texttt{if}\ (\mathbf{Zero}\ \mathbf{Z}^{0})\ \texttt{then}\ (\mathbf{S}^{+}\ \underline{1})\ \texttt{else}\ (\mathbf{+}\ (\mathbf{S}^{+}\ \underline{1})\ \mathbf{Z}^{0}) \\
&\twoheadrightarrow \mathbf{S}^{+}\ \underline{1} \equiv \underline{2}. \qquad \blacksquare
\end{aligned}
$$

Did That Look Complicated?

- Consider the following C-style programming example:

```
int fact(int n)
{
    if (n == 0) {
        return 1;
    } else {
        return n * fact(n-1);
    }
}
```

- The same idea can be expressed elegantly in LC:

$$
\begin{array}{l}
\mathtt{fix}\ (\lambda f n. \\
\quad \mathtt{if}\ (\mathbf{Zero}\ n) \\
\qquad \mathtt{then}\ \underline{1} \\
\qquad \mathtt{else}\ (\mathbf{S} \cdot (\circ\, f\, \mathbf{P}^{-})\ n))
\end{array}
$$

with $\mathtt{fix} \stackrel{\text{def}}{=} \mathbf{\Theta}$.

Tuples and Projections

- The idea of truth values has a generalisation: An *n-tuple*:
  - The *constructor*: $(x_0, x_1, \ldots, x_{n\text{-}1}) \stackrel{\text{def}}{=} \lambda x_0 x_1 \ldots x_{n\text{-}1}.\lambda z.z x_0 x_1 \ldots x_{n\text{-}1}$
  - *Projections*: $\mathsf{p}_{\mathrm{i}} \stackrel{\text{def}}{=} \lambda w.w(\lambda x_0 x_1 \ldots x_{n\text{-}1}.x_i)$, for every i s.t. $0 \leq i < n$
- For instance,
  - $(M, N, O) \equiv (\lambda x_0 x_1 x_2.\lambda z.z x_0 x_1 x_2) MNO \twoheadrightarrow \lambda z.zMNO$
  - Thus, $\mathsf{p}_1(M, N, O) \equiv (\lambda w.w(\lambda x_0 x_1 x_2.x_1))(\lambda z.zMNO) \twoheadrightarrow N$
- This construction is called the *Scott encoding*

Multivariate Composition

- Function composition can be generalised for n-ary functions
- If $f: Y^n \to Z$, and $g_1, g_2, \ldots, g_\mathrm{n} : X^m \to Y$, then

$$h(x_1, \ldots x_m) \stackrel{\text{def}}{=} f(g_1(x_1, \ldots\ x_m), g_2(x_1, \ldots x_m), \ldots, g_n(x_1, \ldots x_m))$$

  is the composed function $X^m \to Z$
- For λ-exprs F, G_1, G_2, … , G_n, we can define the composition as

$$\lambda x_1 \ldots\ x_m\ .\ F\ (G_1 x_1 \ldots x_m)\ (G_2 x_1 \ldots x_m)\ \ldots\ (G_1 x_1 \ldots x_m)$$

Unbounded Minimization

- For representing all effectively computable (μ-recursive, Turing complete, or whatever) functions, we need one more construct
- The *unbounded minimization* operator μ, when applied to a λ-expr M, returns *the least natural number* n *such that* $Pn = \mathbf{T}$ (if it exists, otherwise μ has no β-NF, i.e. evaluation goes forever)
- In other words, $\mu \stackrel{\text{def}}{=} \lambda p.\mathbf{\Theta} E \mathbf{Z}^{0}$, with
  $E \stackrel{\text{def}}{=} \lambda f x.\texttt{if}\ (px)\ \texttt{then}\ x\ \texttt{else}\ (f(\mathbf{S}^{+}x))$
  - Thus, $\mu P \twoheadrightarrow \texttt{if}\ (P\mathbf{Z}^{0})\ \texttt{then}\ \mathbf{Z}^{0}\ \texttt{else}\ (\mathbf{\Theta}(E[p{:=}P])(\mathbf{S}^{+}\mathbf{Z}^{0}))$

Let's Pause for a Minute

- We showed that LC can express the following technicalities:
  1) The *initial functions*: $\mathbf{KZ^0}$, projections, and $\mathbf{S^+}$;
  2) closure under (multivariate) composition;
  3) closure under primitive recursion; and
  4) closure under unbounded minimalization (the μ-operator)
- Thus, LC satisfies the axioms of μ-*Recursive Functions*
  - Put differently, *LC is Turing-complete*

Side Note: Currying

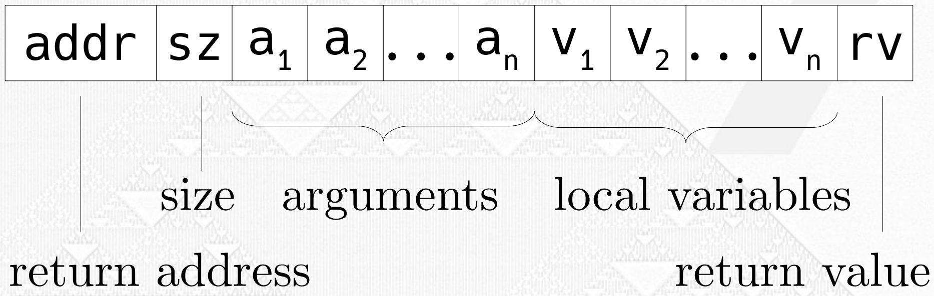
- Another powerful technique is called *currying*, (misattributed) after Haskell Curry, a pioneer in Combinatory Logic and LC
- Currying is the transformation of a $f : (X \times Y) \to Z$ (in a Closed Monoidal Category) to a $f' : X \to (Y \to Z)$
  - If $f(x,y) \equiv \phi(x,y)$, then $f'(x) \stackrel{\text{def}}{=} (y \mapsto \phi(x,y))$ (x is constant in RHS)
  - Thus, $f(a,b) \equiv f'(a)(b)$ (i.e. $\phi(a,b)$). Remember pseudocode on p. 24?
- We have been currying our functions all along... Currying also works also for multivariate (n-ary) functions, by the way

Side Note: Evaluation Strategies

- Consider what happens when reducing **KIΩ**
  - If we start from **K**, we obtain **I**, which is in normal form
  - If we start from **Ω**, we get **Ω** back, which is not in normal form
- An algorithm for choosing the redex to reduce, step by step, until a NF is reached, is called an *evaluation strategy*. E.g.:
  - "*Always choose the leftmost redex*" is guaranteed to always find the β-NF, if it exists (called *lazy evaluation* strategy in programming)
  - "*Reduce arguments before functions*" fails to reach NF with **KIΩ**

Side Note: Subroutines

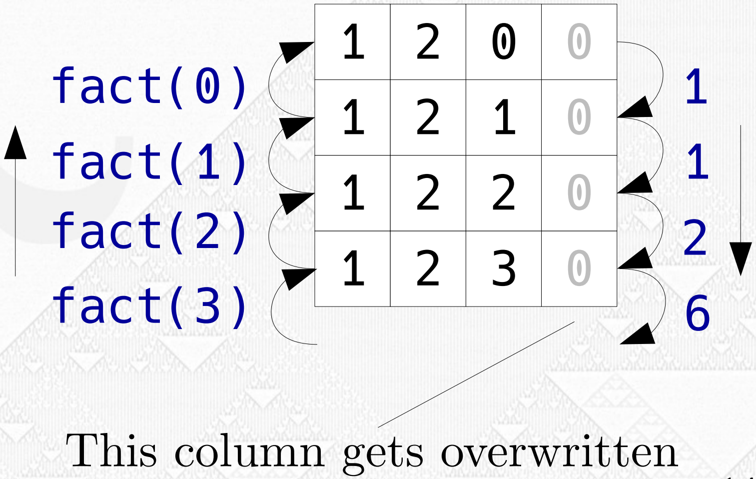
- You may think that heavy use of functions is inefficient
- *Subroutines* need a *call stack*
- The stack contains *frames*:



- Consider the invocation

`1: fact(3)` (see p. 136)

- It results in four *calls*:



Side Note: Tail Recursion

- *Pure functions don't need to be represented as subroutines*
- This (pure) Haskell function uses a helper function `fact'` with an *accumulator* `k`:

```
fact n =
  let fact' 0 k = k
      fact' n k = fact' (n-1)
                        (n*k)
  in  fact' n 1
```

```
fact 3 → fact' 3 1
       ↠ fact' 2 3
       ↠ fact' 1 6
       ↠ fact' 0 6
       ↠ 6
```

- The function is *tail recursive*. It compiles into a *loop*!
- *Pure functions are just rewrite rules*

Side Note: List Fusion

- In Haskell, we write $\circ$ as '`.`':
  `(f . g) x = g (f x)`
- Consider the function `map`:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- It has the following property:
  `map f . map g = map (f . g)`
- Since `f` and `g` are *pure*, properties like this exist

- Two list traversals:

```
(map (+1) . map (*2)) [0,1,2]
↠ map (+1) (map (*2) [0,1,2])
↠ map (+1) [0,2,4]
↠ [1,3,5]
```

- They can be turned into one:

```
map ((+1) . (*2)) [0,1,2]
↠ [0*2+1, 1*2+1, 2*2+1]
↠ [1, 3, 5]
```

Side Note: Encoding Natural Numbers

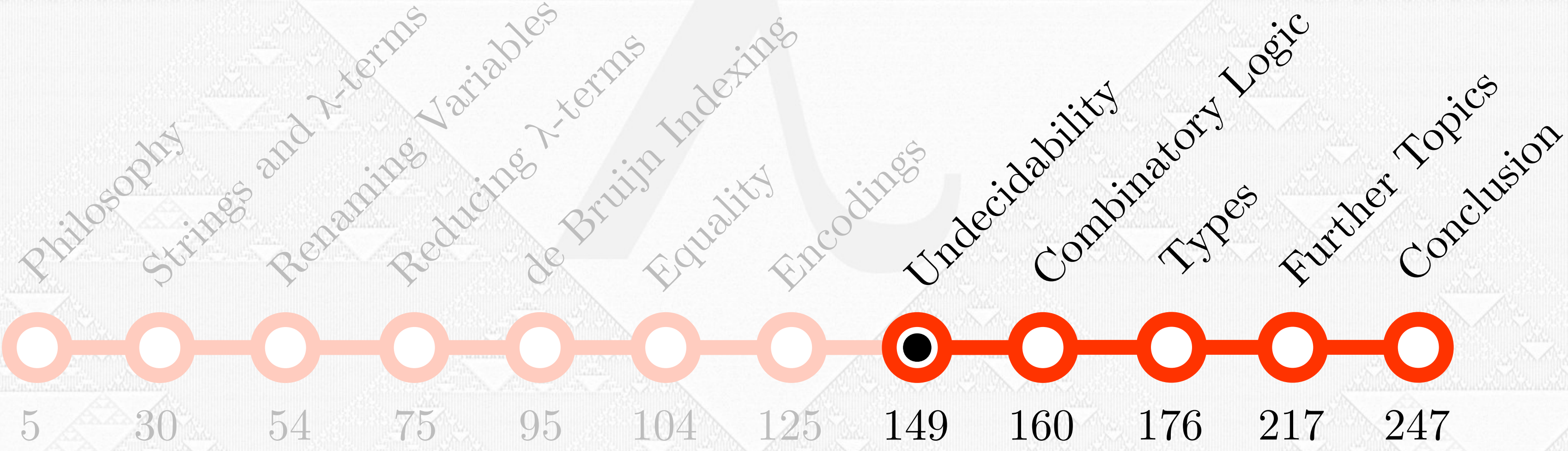
- In *Church encoding*, numerals (see p. 128) can be defined as
  $\underline{n} \stackrel{\text{def}}{=} \lambda fx.f^{\,n}x$,
  where $f^{0}x \stackrel{\text{def}}{=} x$, $f^{\,n+1}x \stackrel{\text{def}}{=} f(f^{\,n}x)$
- Thus, $\underline{n}$ means "apply f n times on x". Define
  - $\mathbf{Z}^{0} \stackrel{\text{def}}{=} \lambda fx.x$ (i.e. $\underline{0}$)
  - $\mathbf{S}^{+} \stackrel{\text{def}}{=} \lambda nfx.f(nfx)$ (i.e. $\lambda nfx.f^{\,n+1}x$)
- It can be verified that $\mathbf{S}^{+}\underline{n} = \underline{n+1} = (\mathbf{S}^{+})^{n}$

Side Note: Arithmetic in Church Encoding

- The structure of Church natural numbers can be exploited to define basic arithmetic operations without recursion
  - $\underline{n}+\underline{m} \overset{\text{def}}{=} \lambda fx.\underline{n}f(\underline{m}fx)$ (i.e. $\lambda fx.f^{\,n}(f^{m}x) = \lambda fx.f^{\,m+n}x$)
  - $\underline{n}\cdot\underline{m} \overset{\text{def}}{=} \lambda fx.\underline{n}(\underline{m}f)x$ (i.e. $\lambda fx.(\lambda y.f^{\,m}y)^{n}x =_{\eta} \lambda fx.f^{\,mn}x$)
  - $\underline{n}^{m} \overset{\text{def}}{=} \underline{mn}$ (i.e. m times multiplication with n)
- Actually, the recursion is there, but it's built in the structure of the numerals, in meta language

Subway Map

- Now that we know what can be done with LC, it's time to have a look at *what can't be done*



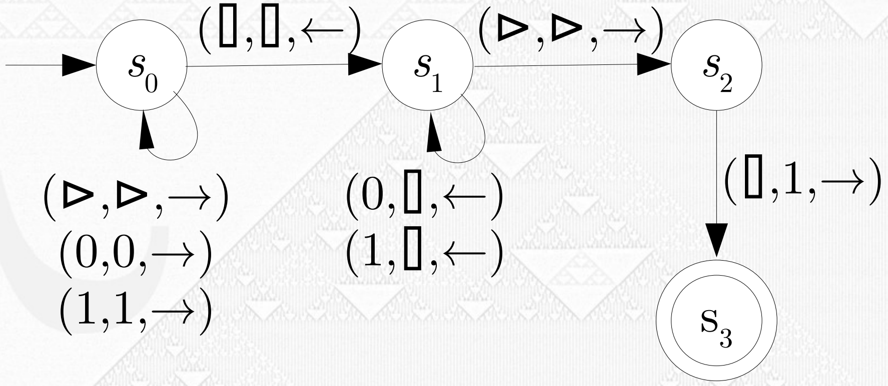
The Limits of Lambda-Calculus

- Using the techniques presented so far, it is possible to define each and every *(μ-)recursive function* in LC
- Now that we've seen what can be done in LC, it's time to ask: *what can't be done in LC?*
- It turns out to be impossible to predict whether or not an arbitrary λ-expr has a normal form
- We're going to prove this using combinators

(Semi)Decidability

- Does a λ-expr have a β-nf?
  - If so, what is it?
- Do two functions have equal graphs?
- Do we know when two real numbers are equal?
- We'll need more machinery!

- Does a Turing Machine halt and accept/reject an input?



A Little Bit More Machinery

- Let's define two more standard combinators
  - $\mathbf{B} \stackrel{\text{def}}{=} \mathbf{K}(\mathbf{SK})\mathbf{K}$, or equivalently $\mathbf{B} \stackrel{\text{def}}{=} \lambda xyz.x(yz)$
  - It follows that $\mathbf{B}MNO \twoheadrightarrow M(NO)$
  - $\mathbf{C} \stackrel{\text{def}}{=} \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S}))(\mathbf{KK})$, or equivalently $\mathbf{C} \stackrel{\text{def}}{=} \lambda xyz.xzy$
  - It follows that $\mathbf{C}MNO \twoheadrightarrow MON$

Why These Are the Standard Combinators?

Combinator	Explanation
$\mathbf{I}x = x$	The identity function
$\mathbf{K}x = \lambda y.x$	Constant function
$\mathbf{S}fg = \lambda x.fx(gx)$	Distributes x to f and g
$\mathbf{B}fg = \lambda x.f(gx)$	Function composition
$\mathbf{C}fgh = fhg$	Inverts `if-then-else`

Combinator Cheatsheet

Symbol	Definition	Redex*	Reduct*
$\mathbf{I}$	$\lambda x.x$	$\mathbf{I}M$	M
$\mathbf{K}$	$\lambda xy.x$	$\mathbf{K}MN$	MN
$\mathbf{T}$	$\lambda xy.x$	$\mathbf{T}MN$	M
$\mathbf{F}$	$\lambda xy.y$	$\mathbf{F}MN$	N
$\mathbf{S}$	$\lambda xyz.xz(yz)$	$\mathbf{S}MNO$	$MO(NO)$
$\mathbf{B}$	$\lambda xyz.x(yz)$	$\mathbf{B}MNO$	$M(NO)$
$\mathbf{C}$	$\lambda xyz.xzy$	$\mathbf{C}MNO$	MON
$\mathbf{\Theta}$	$(\lambda xy.y(xxy))(\lambda xy.y(xxy))$	$\mathbf{\Theta}M$	$M(\mathbf{\Theta}M)$
$\mathbf{\Omega}$	$(\lambda x.xx)(\lambda x.xx)$	$\mathbf{\Omega}$	$\mathbf{\Omega}$

* = When fully applied

Getting Ready For The Big Surprise

- We're about to reach the famous *Halting Problem*
  - The formulation in this presentation is taken from https://en.wikipedia.org/wiki/Combinatory_logic#Undecidability_of_combinatorial_calculus (viewed in 2020-02-08)
  - The problem is to decide whether an arbitrary λ-expr has a β-nf
- Let's assume that there is such a λ-expr N that for any F:
  - $NF \twoheadrightarrow \mathbf{T}$, if F has a normal form; and $NF \twoheadrightarrow \mathbf{F}$ otherwise
- Let $Z \stackrel{\text{def}}{=} \mathbf{C}(\mathbf{C}(\mathbf{B}N(\mathbf{SII}))\mathbf{\Omega})\mathbf{I}$

The Halting Problem

- $$\begin{aligned} ZZ \quad & \equiv (\mathbf{C}(\mathbf{C}(\mathbf{B}N(\mathbf{SII}))\mathbf{\Omega})\mathbf{I})Z \\ & \twoheadrightarrow \mathbf{C}(\mathbf{B}N(\mathbf{SII}))\mathbf{\Omega}Z\mathbf{I} \\ & \twoheadrightarrow \mathbf{B}N(\mathbf{SII})Z\mathbf{\Omega}\mathbf{I} \\ & \twoheadrightarrow N((\mathbf{SII})Z)\mathbf{\Omega}\mathbf{I} \\ & \twoheadrightarrow N(\mathbf{I}Z(\mathbf{I}Z))\mathbf{\Omega}\mathbf{I} \\ & \twoheadrightarrow N(ZZ)\mathbf{\Omega}\mathbf{I} \end{aligned}$$
- ZZ asks N whether ZZ itself has a normal form or not
  - A rather strange loop, isn't it?

The Halting Problem, Continued

- If $N(ZZ) \twoheadrightarrow \mathbf{T}$, then $ZZ = N(ZZ)\mathbf{\Omega I} \twoheadrightarrow \mathbf{T\Omega I} \twoheadrightarrow \mathbf{\Omega}$, which does not have a normal form
  - This contradicts the decision N made. N didn't see that coming!
- If $N(ZZ) \twoheadrightarrow \mathbf{F}$, then $ZZ = N(ZZ)\mathbf{\Omega I} \twoheadrightarrow \mathbf{F\Omega I} \twoheadrightarrow \mathbf{I}$, which does have a normal form
  - Again, N made a mistake
- Therefore, we must conclude that N *cannot exist* $\blacksquare$

So What?

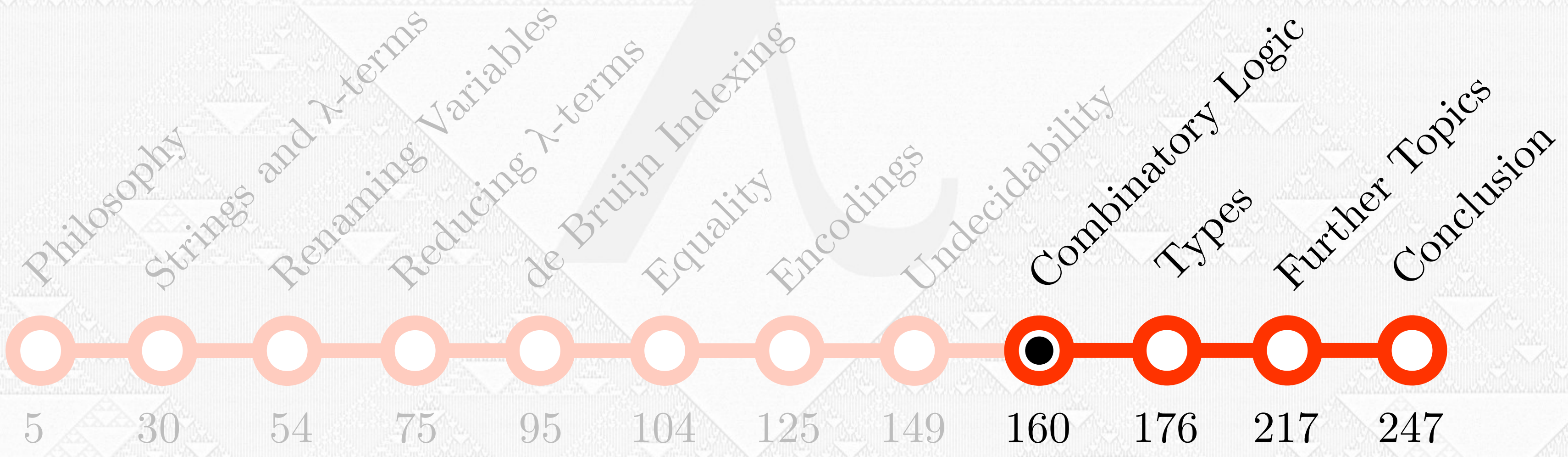
- It's important to understand both the potential and limitations of the system one's working with
- Undecidability of certain questions, e.g. existence of a β-NF for an arbitrary λ-expr, termination of μ-recursive functions, or the Rice's Theorem (saying that we can't decide anything non-trivial for an arbitrary μ-recursive function) doesn't mean that we should give up!
- *We just need to ask the right questions* (e.g. type checking, model checking, static/dynamic analysis, etc.) *to make things decidable*

Summary on LC

- The fundamental concept of LC is lambda expressions
  - Lambda expressions behave like anonymous functions
- Every λ-expr can be constructed using three rules
- Reducing lambda expressions is like performing arithmetics
- LC is the functional analogue to assembly language
  - It provides only the minimal set of building blocks
  - Recursion emerges in LC through self-application

Subway Map

- Now that the beef of pure untyped LC has been chewed, we can discuss topics that are built on top of this foundation



More About Combinators

- Abstraction is not necessary if we axiomatize the reducts for fully applied combinators (see the table on p. 153)
- Combinators can simulate other λ-exprs
  - (We know the converse already)
- If we restrict the rules to only allow the use of combinators and free variables, we don't need substitution at all
  - Such restriction is called *Combinatory Logic* (CL)

Removing Lambda Abstractions

1) $[x] \stackrel{\text{def}}{=} x;$

2) $[\lambda x.x] \stackrel{\text{def}}{=} \mathbf{I};$

3) $[\lambda x.M] \stackrel{\text{def}}{=} (\mathbf{K}[M])$, if x is not free in M;

4) $[\lambda x.\lambda y.M] \stackrel{\text{def}}{=} [\lambda x.[\lambda y.M]]$, if x is free in M;

5) $[\lambda x.MN] \stackrel{\text{def}}{=} (\mathbf{C}\ [\lambda x.M]\ [N])$, if x is free only in M;

6) $[\lambda x.MN] \stackrel{\text{def}}{=} (\mathbf{B}\ [M]\ [\lambda x.N])$, if x is free only in N;

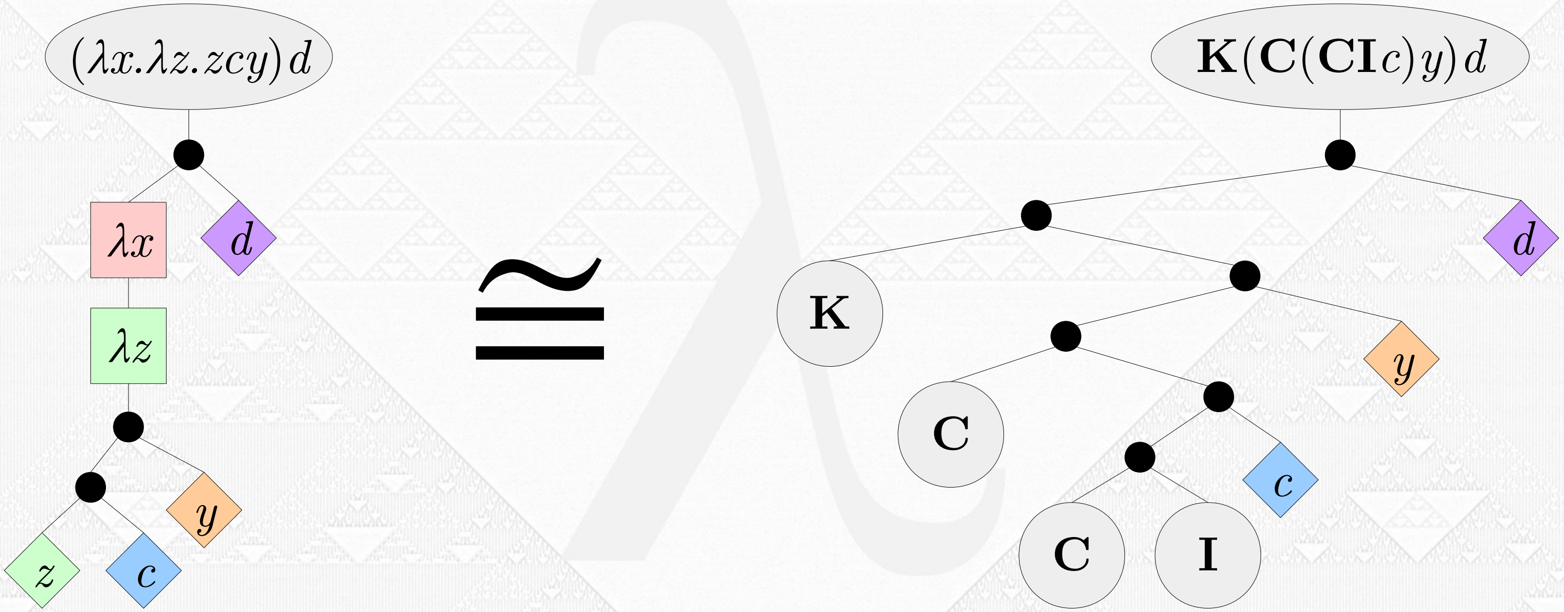
7) $[\lambda x.MN] \stackrel{\text{def}}{=} (\mathbf{S}\ [\lambda x.M]\ [\lambda x.N])$, if x is free in M and N;

8) $[MN] \stackrel{\text{def}}{=} ([M]\ [N])$

Example Abstraction Elimination

$$
\begin{aligned}
[(\lambda x.\lambda z.zcy)\,d] &= ([(\lambda x.\lambda z.zcy)][d]) && (8)\\
&= ([\lambda x.\lambda z.zcy])\,d && (1)\\
&= (\mathbf{K}[\lambda z.zcy])\,d && (3)\\
&= (\mathbf{K}(\mathbf{C}[\lambda z.zc][y]))\,d && (5)\\
&= (\mathbf{K}(\mathbf{C}(\mathbf{C}[\lambda z.z][c])[y]))\,d && (5)\\
&= (\mathbf{K}(\mathbf{C}(\mathbf{C}\mathbf{I}[c])[y]))\,d && (2)\\
&= (\mathbf{K}(\mathbf{C}(\mathbf{C}\mathbf{I}c)[y]))\,d && (1)\\
&= \mathbf{K}(\mathbf{C}(\mathbf{C}\mathbf{I}c)y)\,d && (1)
\end{aligned}
$$

The Last Trees



Formalising Combinatory Logic

- We've seen combinators in LC. Now we define CL independently
- The alphabet of CL contains parentheses, variable symbols (x_0, x_1, x_2, ...), and constants $\mathbf{K}$ and $\mathbf{S}$ (remember p. 119?)
- CL-terms are defined inductively as follows:
  1) $\mathbf{K}$, $\mathbf{S}$, and x (for any variable symbol x) are CL-terms
  2) If X and Y are CL-terms, then so is (XY) (with $XYZ \overset{\text{def}}{=} (XY)Z$)
  3) Nothing else is a CL-term

The Equational Theory CLw

- CLw formalises CL with the following *axiom schemes*:

$$
\begin{array}{lll}
(\mathbf{K}) & \mathbf{K}XY & = X \\
(\mathbf{S}) & \mathbf{S}XYZ & = XZ(YZ) \\
(\rho) & X & = X
\end{array}
$$

- These are templates for equations that are assumed to hold *a priori*

- Four rules of inference:

(σ) $X = Y$ implies $Y = X$

(τ) $X = Y$ and $Y = Z$ implies $Y = Z$

(μ) $X = Y$ implies $ZX = ZY$

(ν) $X = Y$ implies $XZ = YZ$

- These rules work for any CL expressions X, Y, and Z

Provability

- An equation $X = Y$ in an *equational theory* T (e.g. CLw) is *provable* if and only if it can be derived using axioms and inference rules. We denote this with $T \vdash X = Y$
- For example, $CLw \vdash \mathbf{SKS}x = x$:

1)	$\mathbf{SKS}x = \mathbf{K}x(\mathbf{S}x)$	$(\mathbf{S})$
2)	$\mathbf{K}x(\mathbf{S}x) = x$	$(\mathbf{K})$
3)	$\mathbf{SKS}x = x$	$(\tau, 1, 2)$

Sidenote: Simulating β-Equality

- β-equality can be simulated with the following five axioms:
  1) $\mathbf{K} = \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{K}))(\mathbf{K}(\mathbf{SKK}))$
  2) $\mathbf{S} = \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{KS})))(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{KK}))\mathbf{S})))(\mathbf{K}(\mathbf{K}(\mathbf{SKK})))$
  3) $\mathbf{S}(\mathbf{KK}) = \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})(\mathbf{SKK})))(\mathbf{K}(\mathbf{SKK}))$
  4) $\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})) = \mathbf{S}(\mathbf{KK})(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})(\mathbf{SKK})))(\mathbf{K}(\mathbf{SKK})))$
  5) $\mathbf{S}((\mathbf{K}(\mathbf{S}(\mathbf{KS})))(\mathbf{S}(\mathbf{KS}))) = \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{KS}))\mathbf{S}))))(\mathbf{KS})$

Sidenote: Simulating η-Equality

- $(\beta)\eta$-equality can be simulated with the following five axioms:

3) $\mathbf{S}(\mathbf{KK}) = \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})(\mathbf{SKK})))(\mathbf{K}(\mathbf{SKK}))$

4) $\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})) = \mathbf{S}(\mathbf{KK})(\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})(\mathbf{SKK})))(\mathbf{K}(\mathbf{SKK})))$

5) $\mathbf{S}((\mathbf{K}(\mathbf{S}(\mathbf{KS})))(\mathbf{S}(\mathbf{KS}))) =$
$$\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{KS})))\mathbf{S}))))(\mathbf{KS})$$

6) $\mathbf{SKK} = \mathbf{S}(\mathbf{S}(\mathbf{KS})\mathbf{K})(\mathbf{K}(\mathbf{SKK}))$

Applicative Structures

- A set D with at least two distinct elements and a binary operator $\bullet : D^2 \to D$ form an *applicative structure* $(D,\bullet)$ if
  - D is closed under $\bullet$, i.e. for any elements x and y of D, $x\bullet y$ is an element of D
  - $\bullet$ associates to left, i.e. for all elements x, y, and z of D, $(x\bullet y)\bullet z = x\bullet(y\bullet z)$
- An assignment f maps variables to elements of D

Combinatory Algebras

- *Combinatory algebra* is an applicative structure with two distinct elements, k and s, such that
  1) $k \bullet x \bullet y = x$
  2) $s \bullet x \bullet y \bullet z = x \bullet z \bullet (y \bullet z)$
- Looks familiar, doesn't it?

- The *interpretation* of an CL expression M is defined inductively as
  1) $[\![x]\!]_f \stackrel{\text{def}}{=} f(x)$
  2) $[\![\mathbf{K}]\!]_f \stackrel{\text{def}}{=} k$
  3) $[\![\mathbf{S}]\!]_f \stackrel{\text{def}}{=} s;$
  4) $[\![XY]\!]_f \stackrel{\text{def}}{=} [\![X]\!]_f \bullet [\![Y]\!]_f$

Term Models

- Let $[X] \stackrel{\text{def}}{=} \{\, Y \mid CLw \vdash X = Y\}$. In other words, [X] is the equivalence class of X w.r.t. provability in CLw
- The *term model* of CLw is a combinatory algebra:
  - $D = \{[X] \mid X \text{ is a CL-term}\}$
  - $[X] \bullet [Y] = [XY]$
  - $k = [\mathbf{K}]$
  - $s = [\mathbf{S}]$

Semantical Example

- Consider SKx. Let $f(x) \stackrel{\text{def}}{=} [\mathbf{K}]$. Thus,
  - $$\begin{aligned}[\![\mathbf{SK}x]\!]_f &= [\![\mathbf{SK}]\!]_f \bullet [\![x]\!]_f \\ &= ([\![\mathbf{S}]\!]_f \bullet [\![\mathbf{K}]\!]_f) \bullet [\![x]\!]_f \\ &= (s \bullet k) \bullet f(x) \\ &= ([\mathbf{S}] \bullet [\mathbf{K}]) \bullet [\mathbf{K}] \\ &= ([\mathbf{SK}]) \bullet [\mathbf{K}] \\ &= [\mathbf{SKK}]\end{aligned}$$
- This model is rather boring, isn't it?

Satisfaction

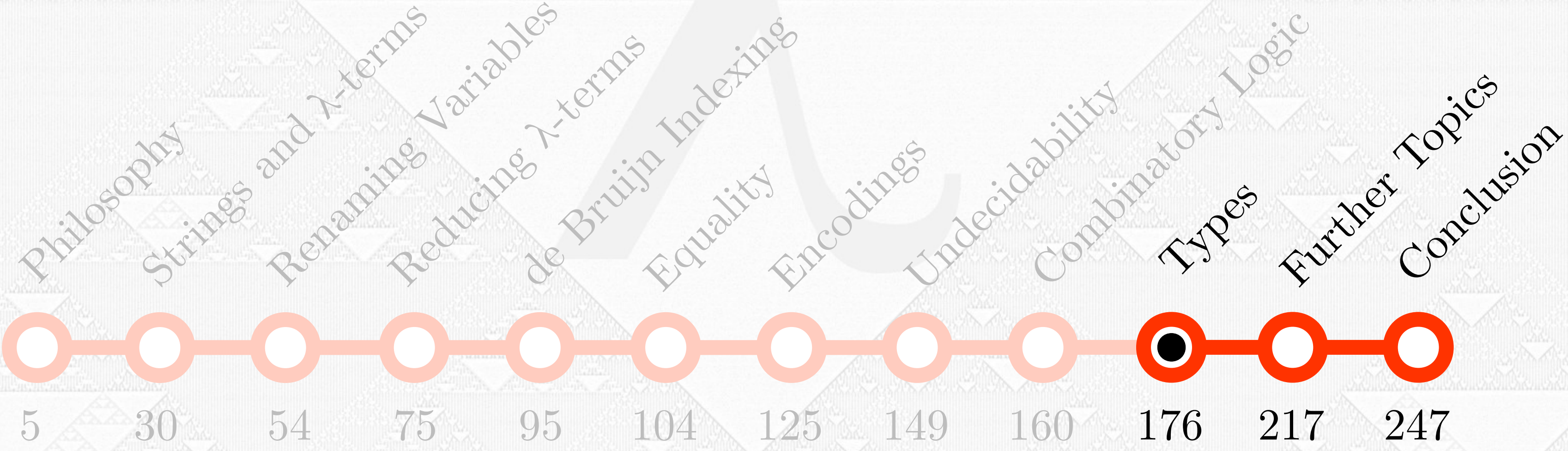
- A *model* M (e.g. a combinatory algebra) with assignment f *satisfies* the equation $X = Y$ if and only if $[\![X]\!]_f = [\![Y]\!]_f$ in M, denoted $M, f \models X = Y$. We omit f if the equation holds for all f
- We saw that the term model TM satisfied $\mathbf{SK}\mathrm{x} = \mathbf{SKK}$ with the assignment $f(x) \stackrel{\text{def}}{=} [\mathrm{K}]$. Hence, $TM, f \models \mathbf{SK}\mathrm{x} = \mathbf{SKK}$
- If $M, f \models \mathbf{S} = \mathbf{K}$, then the model breaks down as all its elements become equal. *Soundness* is built in the concept of a model

Combinatory Completeness

- A *combination* of variables x_1, x_2, ..., x_n is any combinatory term made of these variables, not containing **K** or **S**
  - E.g. $x_1(x_2x_1)x_4$ is a combination of x_1, x_2, x_3, and x_4
- An applicative structure D is *combinatory complete* iff for any combination X of x_1, x_2, ..., x_n, there are elements a, d_1, d_2, ..., d_n in D s.t. $a \bullet d_1 \bullet d_2 \bullet \ldots \bullet d_n = [\![X]\!]_{f[x1:=d1][x2:=d2]\ldots[xn:=dn]}$
- *An applicative structory is combinatory complete iff it's a combinatory algebra*

Subway Map

- We'll see later why combinatory algebras are a meaningful concept, and that there are non-trivial models of LC. But let's switch to type theory



Type Theory

- In untyped formal systems, self-application, self-reference, unbounded recursion, or undefined values often cause contradictions, paradoxes, infinite loops, crashes, or exploits
- Type theory (TT) is a foundational field of math and computer science that specialises in preventing these problems by finding necessary and sufficient restrictions to the object language
  - TT has origins in set theory, though these days it mostly uses LC
- Type theory adds new primitives to LC (e.g. tuples)

Why Types Matter in Programming?

- Non-sensible typing decisions in programming, such as representing strings as pointers to byte arrays, has often led into catastrophical failures and security vulnerabilities
- Debugging a program can be much harder than ensuring its type safety statically during compilation
- At its best, a type system can guide architectural and design choices and communication, and help finding canonical solutions
- Pre and postconditions can be often expressed as types

Type Theory in Programming

- Type theory is often used (unwittingly) by programmers that use statically typed programming languages
- In statically typed programming languages, such as Haskell, Java, or C/C++, a type system is used as a *partial formal verification tool* for program code
  - In these languages, untyped variable and function definitions are augmented with type signatures
  - In modern languages, the compiler can infer most signatures

Examples of Type Signatures

- Statically typed programming languages usually provide primitive types such as `bool`, `byte`, `int`, `float`, `double`, etc.
  - In Java or C, `int x = 0;` defines an integer variable, whereas the Haskell syntax for it would be `x = 0` (and optionally `x :: Int`)
- Many languages feature complex, often polymorphic and/or generic, types such as function, list, or record types
  - A list in C++ would be initiated with `list<int> xs{1,2,3}`, whereas the Haskell syntax is `xs = [1,2,3]` (which permits `xs :: [Int]`)

Algebraic Datatypes

- Scott encoding generalises into *Algebraic Data Types* (ADTs)
- In Haskell, the following ADTs can be defined:
  - `data Bool       = True   | False`,
  - `data Either a b = Left a | Right b`
  - `data List a     = Nil    | Cons a (List a)`
- In a `data` equation, the left hand side defines a type constructor, and the right hand side provides (value) constructors. Note the similarity to the BNF notation (p. 37)

Pattern Matching

- In Haskell, functions can be defined in parts in three ways
- Consider a case-expression:
  - `f e = case e of Left x -> g x; Right y -> h y`
- Pattern maching allows deconstructing a value `e` of type `Either` and handling the different possibilities as separate cases
- We may conclude that
  - `g :: a -> c; h :: b -> c; f :: Either a b -> c`

Type Theory in Mathematics

- Type theory is also used by mathematicians to prove theorems using computer programs known as *interactive proof assistants*.
- There are two main schools:
  - *Logic for Computable Functions* (LCF): Types ensure well-formedness of formulae, rules are implemented as functions
  - *Propositions as Types / Proofs as Terms* (PAT): Types encode propositions, for which values provide witnesses

Type Theory in Mathematics

- Type theory is also used by mathematicians to prove theorems using computer programs known as *interactive proof assistants*.
- There are two main schools:
  - *Logic for Computable Functions* (LCF): Types ensure well-formedness of formulae, rules are implemented as functions
  - *Propositions as Types / Proofs as Terms* (PAT): Types encode propositions, for which values provide witnesses

Judgements and Deductions

- The basic notion in type theory is that of a *typing judgement*:
  - $\Gamma \vdash M : A$ means "assuming Γ, the *value* M has *type* A"
  - Γ is a list (or ordered sequence) of hypotheses $x_i : A_i$
  - E.g. $[] \vdash 0 : \mathbb{N}$, $[] \vdash 1{+}1 : \mathbb{N}$, $[n : \mathbb{N}] \vdash n + 1 : \mathbb{N}$
- *Deductions* in type theory chain judgements together:

$$\frac{\mathit{premise}}{\mathit{conclusion}} \quad \text{, e.g.} \quad \frac{[n : \mathbb{N}] \vdash n + 1 : \mathbb{N}}{[n : \mathbb{N}] \vdash \mathbf{S}^{+}(n + 1) : \mathbb{N}}$$

Implication

- Recall the formation rules in slide 38

$$\frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Implication

- *(Intuitionistic) Implication* is basically the same as *logical consequence*

$$\frac{}{\Gamma, A \vdash A}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

Implication

- Recall the formation rules in slide 38
- *(Intuitionistic) Implication* is basically the same as *logical consequence*
- In intuitionistic logic, a proof of implication is a function of proofs of A to proofs of B

$$\frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \Rightarrow B}$$

$$\frac{\Gamma \vdash M : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Conjunction

- Pair (or generally tuple) types correspond with conjunctions of logical propositions
- Cf. Scott encoding, p. 137
- Proofs of conjunctions are pairs of proofs of the conjuncts

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \land B}$$

$$\frac{\Gamma \vdash M : A \land B}{\Gamma \vdash \pi_0 M : A} \qquad \frac{\Gamma \vdash M : A \land B}{\Gamma \vdash \pi_1 M : B}$$

Disjunction

- Disjunction is based on the concept of *disjoint union*
- Elements of type A or B can be injected into $A \vee B$
- $[N, O](\iota_0\, M) \to NM$; and $[N, O](\iota_1\, M') \to OM'$
- This is *pattern matching*

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \iota_0 M : A \vee B}$$

$$\frac{\Gamma \vdash M : B}{\Gamma \vdash \iota_1 M : A \vee B}$$

$$\frac{\Gamma \vdash N : A \Rightarrow C \qquad \Gamma \vdash O : B \Rightarrow C}{\Gamma \vdash [\mathrm{N,O}] : A \vee B \Rightarrow C}$$

Example Deduction

- Let $\Gamma \overset{\text{def}}{=} p{:}A\wedge(B\vee C)$, $\Gamma' \overset{\text{def}}{=} \Gamma, b{:}B$, $\Gamma'' \overset{\text{def}}{=} \Gamma, c{:}C$. Thus,

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \vdash p{:}A\wedge(B\vee C)}{\Gamma' \vdash \pi_0 p{:}A} \qquad \Gamma' \vdash b{:}B}{\Gamma' \vdash \langle \pi_0 p, b\rangle{:}(A\wedge B)}}{\Gamma' \vdash \iota_0\langle \pi_0 p, b\rangle{:}(A\wedge B)\vee(A\wedge C)}}{\Gamma \vdash \lambda b.\iota_0\langle \pi_0 p, b\rangle{:}B\Rightarrow(A\wedge B)\vee(A\wedge C)}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma'' \vdash p{:}A\wedge(B\vee C)}{\Gamma'' \vdash \pi_0 p{:}A} \qquad \Gamma'' \vdash c{:}C}{\Gamma'' \vdash \langle \pi_0 p, c\rangle{:}(A\wedge C)}}{\Gamma'' \vdash \iota_1\langle \pi_0 p, c\rangle{:}(A\wedge B)\vee(A\wedge C)}}{\Gamma \vdash \lambda c.\iota_1\langle \pi_0 p, c\rangle{:}C\Rightarrow(A\wedge B)\vee(A\wedge C)}$$

$$\dfrac{\Gamma \vdash \lambda b.\iota_0\langle \pi_0 p, b\rangle{:}B\Rightarrow(A\wedge B)\vee(A\wedge C) \qquad \Gamma \vdash \lambda c.\iota_1\langle \pi_0 p, c\rangle{:}C\Rightarrow(A\wedge B)\vee(A\wedge C)}{\Gamma \vdash [\lambda b.\iota_0\langle \pi_0 p, b\rangle, \lambda c.\iota_1\langle \pi_0 p, c\rangle]{:}(B\vee C)\Rightarrow(A\wedge B)\vee(A\wedge C)}$$

$$\dfrac{\Gamma \vdash p{:}A\wedge(B\vee C)}{\Gamma \vdash \pi_1 p{:}B\vee C}$$

$$\dfrac{\dfrac{\Gamma \vdash [\lambda b.\iota_0\langle \pi_0 p, b\rangle, \lambda c.\iota_1\langle \pi_0 p, c\rangle]{:}(B\vee C)\Rightarrow(A\wedge B)\vee(A\wedge C) \qquad \Gamma \vdash \pi_1 p{:}B\vee C}{\Gamma \vdash [\lambda b.\iota_0\langle \pi_0 p,\ b\rangle,\ \lambda c.\iota_1\langle \pi_0 p,\ c\rangle](\pi_1 p) : (A\wedge B)\vee(A\wedge C)}}{\vdash \lambda p.[\lambda b.\iota_0\langle \pi_0 p,\ b\rangle,\ \lambda c.\iota_1\langle \pi_0 p,\ c\rangle](\pi_1 p) : A\wedge(B\vee C)\Rightarrow(A\wedge B)\vee(A\wedge C)}$$

Products and Sums

- Conjunction and disjunction are also known as *product* and *sum* (or *coproduct*) types, respectively
- They're *dual* to each other in the following sense:
  - products: one rule for introduction, two for elimination
  - coproducts: two rules for introduction, one for elimination
- However, they can't be defined in terms of each other, because not all of de Morgan's laws are intuitionistically valid

Unit and Void Types

- Two special types, the *unit type* and the *void type*, are needed for logical and algebraic uses of type theory
- The unit type has only one constructor, $* : \top$
- The void type doesn't have any constructors!

$$\frac{}{\Gamma \vdash * : \top} \qquad \left(\frac{\Gamma \vdash M : \top \Rightarrow A}{\Gamma \vdash M* : A} \right)$$

$$\frac{}{\Gamma \vdash \lambda() : \bot \Rightarrow A} \qquad \frac{\Gamma \vdash M : A \Rightarrow \bot}{\Gamma \vdash M : \neg A}$$

Products are Commutative

- Consider $f \stackrel{\text{def}}{=} \lambda p.\langle \pi_1\, p,\ \pi_0\, p\rangle$
  - $$\begin{aligned} f \circ f &= \lambda x.f(fx) \\ &= \lambda x.\langle \pi_1\,(fx),\ \pi_0\,(fx)\rangle \\ &= \lambda x.\langle \pi_1\,\langle \pi_1\, x,\ \pi_0\, x\rangle,\ \pi_0\,\langle \pi_1\, x,\ \pi_0\, x\rangle\rangle \\ &= \lambda x.\langle \pi_0\, x,\ \pi_1\, x\rangle \end{aligned}$$
- $f : A \wedge B \Rightarrow B \wedge A$ for any types A and B! *f has infinitely many types*
- *$f : B \wedge A \Rightarrow A \wedge B$ is the inverse of $f : A \wedge B \Rightarrow B \wedge A$!*
- f is a *polymorphic* (or *natural*) *isomorphism* $A \wedge B \simeq B \wedge A$. $\blacksquare$

Unit is the Neutral Element of Product

- Let A be a type. Hence, $\pi_0 : \top \wedge A \Rightarrow A$ and $\lambda x.\langle *, x\rangle : A \Rightarrow \top \wedge A$.
  - $\pi_0 \circ \lambda x.\langle *, x\rangle \quad = \lambda y.\pi_0((\lambda x.\langle *, x\rangle)\,y)$
    $= \lambda y.\pi_0\langle *, y\rangle$
    $= \lambda y.y \qquad$ (i.e. identity $A \Rightarrow A$)
  - $\lambda x.\langle *, x\rangle \circ \pi_0 \quad = \lambda y.(\lambda x.\langle *, x\rangle)(\pi_0\, y)$
    $= \lambda y.\langle *, \pi_0\, y\rangle \qquad$ (identity $\top \wedge A \Rightarrow \top \wedge A$)
- We see that π_0 and $\lambda x.\langle *, x\rangle$ are inverses. Therefore, $\mathrm{A} \simeq \top \wedge A$. ■

Products Are Associative up to Isomorphism

- Let $f \stackrel{\text{def}}{=} \lambda p.\langle \pi_0\,(\pi_0\,p),\ \langle \pi_1\,(\pi_0\,p),\ \pi_1\,p\rangle\rangle$
  - $$\begin{aligned} f\,\langle\langle M,\,N\rangle,\,O\rangle &= \langle\ \pi_0\,(\pi_0\,\langle\langle M,\,N\rangle,\,O\rangle),\ \langle\ \pi_1\,(\pi_0\,\langle\langle M,\,N\rangle,\,O\rangle) \\ &\qquad ,\ \pi_1\,\langle\langle M,\,N\rangle,\,O\rangle\ \rangle\ \rangle \\ &= \langle \pi_0\,\langle M,\,N\rangle,\ \langle \pi_1\,\langle M,\,N\rangle,\ O\rangle\rangle \\ &= \langle M,\ \langle N,\ O\rangle\rangle \end{aligned}$$
- Thus, f maps any type $(A \wedge B) \wedge C$ to $A \wedge (B \wedge C)$
- It is to straightforward to show that f has an inverse. Therefore, $(A \wedge B) \wedge C \simeq A \wedge (B \wedge C)$ for any types A, B, and C. ■

Types Form a Commutative Semiring

- The product of any two types is a type. Products have the following properties (i.e. they form a *commutative monoid up to isomorphism*):
  - There is a neutral element;
  - product is associative; and
  - product is commutative
- The same also holds sum types. (Proof is as an exercise)
- Therefore, Types form a commutative semiring (cf. $\mathbb{N}$). $\blacksquare$

Law of the Excluded Middle

- Suppose that $p : \neg(A \vee \neg A)$, i.e. $p : (A \vee \neg A) \Rightarrow \bot$
  - For any $x : A$, $\iota_0\ x : A \vee \neg A$, so $p(\iota_0\ x) : \bot$, i.e. $\iota_0\ x$ contradicts p
  - $\lambda x.p(\iota_0\ x) : A \Rightarrow \bot$, i.e. $\lambda x.p(\iota_0\ x) : \neg A$
  - $\iota_0\,(\lambda x.p(\iota_0\ x)) : (A \vee \neg A)$
  - $p\ (\iota_0\,(\lambda x.p(\iota_0\ x))) : ((A \vee \neg A) \Rightarrow \bot)$
- Therefore, $\lambda p.p\ (\iota_0\,(\lambda x.p(\iota_0\ x))) : ((A \vee \neg A) \Rightarrow \bot) \Rightarrow \bot$, so the type $\neg\neg(A \vee \neg A)$ is inhabited, meaning that LEM is irrefutable. $\blacksquare$

Dependent Type Theory

- We now move on to a more advanced form of type theory, known as *Martin-Löf Type Theory* (MLTT), originally devised by Per Martin-Löf in 1970'ies
  - It is also called intuitionistic type theory or dependent type theory
- The PAT interpretation extends to predicate logic:
  - For any type A and value $x{:}A$ a *proposition* $C(x)$ is a type whose values are proofs (or witnesses) demonstrating that x satisfies C
  - Try not to get confused with the levels (types vs. values)!

The Type of Natural Numbers

- The type of natural numbers is an *inductive type*
- For using the elimination rule, we need a type C that *depends* on a natural number

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \mathbf{S}^{+} M : \mathbb{N}}$$

$$\frac{\Gamma \vdash p_0 : C(0) \qquad \Gamma,\ n{:}\mathbb{N},\ p_n : C(n) \vdash p_{\mathbf{S}^+} : C(\mathbf{S}^{+} n) \qquad \Gamma \vdash M{:}\mathbb{N}}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(C, p_0, p_{\mathbf{S}^+}, M) : C(M)}$$

Pi Types

- Basic function types can be generalised into *dependent product types*
- Using dependent products, universal quantification can be expressed as

$$\forall x.P(x) \stackrel{\text{def}}{=} \Pi x.P(x)$$

$$\frac{\Gamma, x : A \vdash M : B(x)}{\Gamma \vdash \lambda x.M : \Pi x{:}A.B(x)}$$

$$\frac{\Gamma \vdash M : \Pi x{:}A.B(x) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x{:=}A]}$$

Sigma Types

- Somewhat confusingly, basic products can be generalised into *dependent sum types*
  - (These are also sometimes called dependent products!)
- Existential quantification can be expressed as

$$\exists x. P(x) \overset{\text{def}}{=} \Sigma x. P(x)$$

$$\frac{\Gamma, x : A \vdash M : B(x)}{\Gamma \vdash \langle x, M \rangle : \Sigma x{:}A. B(x)}$$

$$\frac{\Gamma \vdash M : \Sigma x{:}A. B(x)}{\Gamma \vdash \pi_0 M : A}$$

$$\frac{\Gamma \vdash M : \Sigma x{:}A. B(x)}{\Gamma \vdash \pi_1 M : B(\pi_0 M)}$$

(Intuitionistic) Axiom of Choice

- As an example, consider the type

  $(\Pi x{:}A.\Sigma y{:}B.R(x,y)) \Rightarrow (\Sigma f{:}A{\Rightarrow}B.\Pi x{:}A.R(x,fx))$

- This type has a value:
  - Let $g : \Pi x{:}A.\Sigma y{:}B.R(x,y)$, so $ga : \Sigma y{:}B.R(x,y)$ for any $a : A$
  - $\pi_0(ga) : B$ for any $a : A$, so $\lambda a.\pi_0(ga) : A \Rightarrow B$
  - $\pi_1(ga) : R(a,\pi_0(ga))$ for all $a : A$, so $\lambda a.\pi_1(ga) : \Pi x{:}A.R(x,\pi_0(gx))$
  - $\langle \lambda a.\pi_0(ga), \lambda a.\pi_1(ga)\rangle : \Sigma f{:}A{\Rightarrow}B.\Pi x{:}A.R(x,fx)$ ∎

Propositional Equality

- So far, we've only discussed *judgemental equality*
- There are two equivalent notions of *propositional equality* used in type theories:
  - Leibniz equality (identity of indiscernibles, $\forall xy{:}A.(\forall P.Px \Rightarrow Py) \Rightarrow x \doteq y$)
  - Martin-Löf equality (shown on the next slide)
  - Given x,y of type A, the *proposition of equality of x and y is a type*
- *Propositional equality is what mathematicians usually want*

Identity Types

- The basic proof of equality says that x is equal to itself:

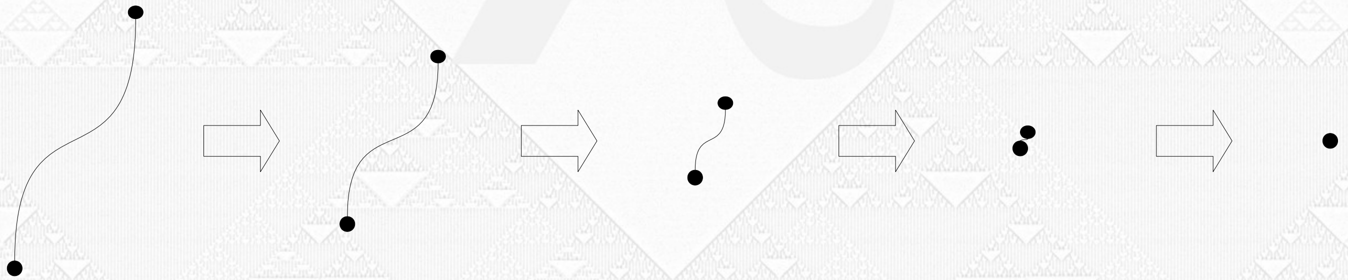
$$\frac{}{\Gamma,\ x : A\ \vdash \mathrm{refl} : \mathrm{Id}_A(x,x)}$$

- For using the elimination rule below, we need a type C that depends on two values of type A and their identity proof:

$$\frac{\Gamma,\ x{:}A \vdash P{:}C(x,x,\mathrm{refl}\ x) \qquad \Gamma \vdash M{:}A,\ N{:}A \qquad \Gamma \vdash Q{:}\mathrm{Id}_A(M,N)}{\Gamma, a{:}A, b{:}A \vdash \mathrm{J}(P,M,N,Q) : C(M,N,Q)}$$

Interpreting Identity Types

- There are at least two ways to think about identity types
  1) Two values are equal if and only if they're interchangeable in all circumstances (Leibniz equality)
  2) We can think of types as spaces, values as points, and equalities as contractible paths between points. We could even have paths between paths, paths between paths between paths, etc.!



Sidenote: Fitch-Style Notation

- Consider the following proof. Let $\Gamma \stackrel{\text{def}}{=} f{:}A{\wedge}B{\Rightarrow}C,\ x{:}A,\ y{:}B$

$$\dfrac{\Gamma \vdash f : A{\wedge}B{\Rightarrow}C \qquad \dfrac{\Gamma{\vdash}x{:}A \qquad \Gamma{\vdash}y{:}B}{\Gamma \vdash \langle x,y\rangle : A{\wedge}B}}{\dfrac{\Gamma \vdash f\langle x,y\rangle : C}{\dfrac{f : A{\wedge}B{\Rightarrow}C,\ x{:}A \vdash \lambda y.f\langle x,y\rangle : B{\Rightarrow}C}{f : A{\wedge}B{\Rightarrow}C \vdash \lambda xy.f\langle x,y\rangle : A{\Rightarrow}B{\Rightarrow}C}}}$$

- The same proof in flag style (with trivial steps omitted):

$$\begin{array}{|l}
\boxed{f : A{\wedge}B{\Rightarrow}C} \\
\quad \begin{array}{|l}
\boxed{x : A} \\
\quad \begin{array}{|l}
\boxed{y : B} \\
\langle x,y\rangle : A{\wedge}B \\
f\langle x,y\rangle : C
\end{array} \\
\lambda y.f\langle x,y\rangle : B{\Rightarrow}C
\end{array} \\
\lambda xy.f\langle x,y\rangle : A{\Rightarrow}B{\Rightarrow}C
\end{array}$$

Side Note: Kinds, Sorts, Rules

- This presentation on type theory hasn't been fully formal
  - Every type A has *kind* $*$, denoted $A : *$
  - Function types A ⇒ B have kind $* \Rightarrow *$, i.e. $A \Rightarrow B : * \Rightarrow *$
  - For technical reasons, also $*$ has *sort* $\square$, denoted $* : \square$
  - Some type systems even have an infinite hierarchy of *universes*
  - The stuff above is needed for specifying *type formation rules*
  - Types may also have *uniqueness* and *computation rules*
  - Even typing contexts can be given formation rules

Typical Type-Theoretic Questions

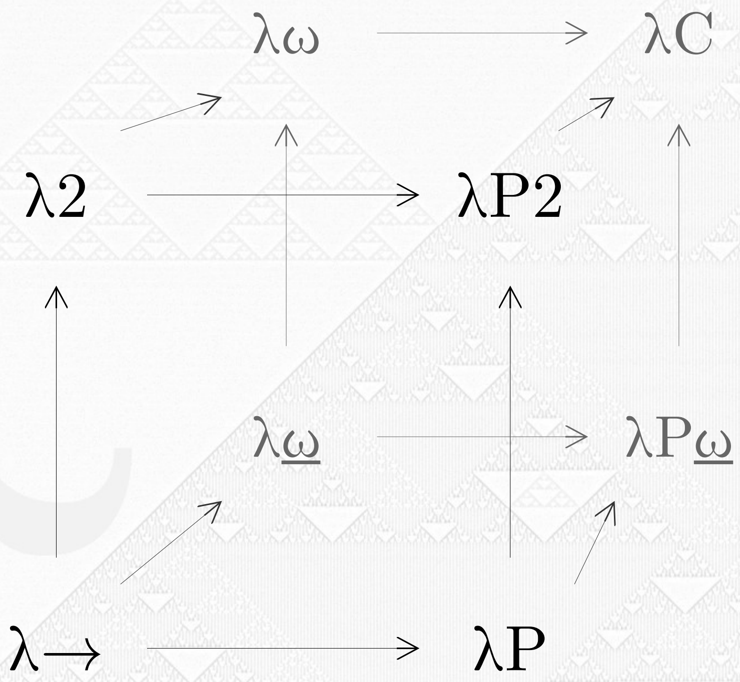
- There are three main types of questions in type theory:
  1) *Type checking*:
     Given Γ, M, A, does $\Gamma \vdash M : A$ hold?
  2) *Type inferece*:
     Given Γ, M, find type A such that $\Gamma \vdash M : A$
  3) *Type inhabitation*:
     Given Γ, A, find term M such that $\Gamma \vdash M : A$
- We say that M and A are *legal* whenever $\Gamma \vdash M : A$ for some Γ.

Select Meta-Theoretic Questions

- *Decidability/soundness* of type checking/inference/inhabitation
- *Weakening/strengthening*: Can we add/remove hypotheses freely?
- *Uniqueness of types*: Are types unique up to $\alpha\beta(\eta\Delta)$-conversion?
- *Weak normalisation*: Does every well-formed type and/or value have a β-NF? (Lazy evaluation always works if there's a β-NF)
  - *Strong normalisation*: Is it reachable with all evaluation strategies?
- *Confluence*: Does the Church-Rosser theorem hold?

The Lambda Cube

- There are eight so-called *pure type systems*, shown in the *Barendregt cube* to the right
- The arrows point from special to more general
- There are plenty of type systems not in the cube



Applied Type Systems

- There are many more type systems than those in the Barendregt cube, including but not limited to:
  - *Martin-Löf Type Theory* (based on λP)
  - *Calculus of (Inductive) Constructions* (based on λC)
  - *Girard-Reynolds Type Theory*, or *System F* (based on λ2)
  - *Hindley-Milner Type System* (slight generalisation of $\lambda\!\to$)
  - The type system of the programming language Haskell extends the Hindley-Milner type system with *type classes*

More Type Systems

- There's a whole zoo of type systems
  - *Linear types*, handy for expressing side-effects and tracking object lifetimes (dependent types also work for this in imperative languages)
  - *Temporal types*, which can be thought of as a typed alternative to macros
  - *Liquid (logically quantified) types* seem to be an alternative to dependent types with a different type inference algorithm

Why Type Theory Matters?

- A mathematician might wonder why bothering with all this constructive extra information in proofs
- Here are some points:
  - Type theory is more precise (richer!) than classical math
  - Efficient automation scales better than blackboard
  - A success story: *Homotopy Type Theory was developed in the Coq proof system first and only deformalised later*

Pros and Cons of Constructive Approach

Cons

- More restricted
- Proofs may become lengthy
- Can prove $p \Rightarrow \neg\neg\, p$, but not $\neg\neg p \Rightarrow p$ (broken symmetry)
- What kind of "calculus" doesn't have real numbers?

Pros

- More restricted
- No need to speculate about "truth" or "existence"
- Proofs produce evidence
- Proofs are programs
- Classical axioms optional

Pros and Cons of Constructive Approach

Cons

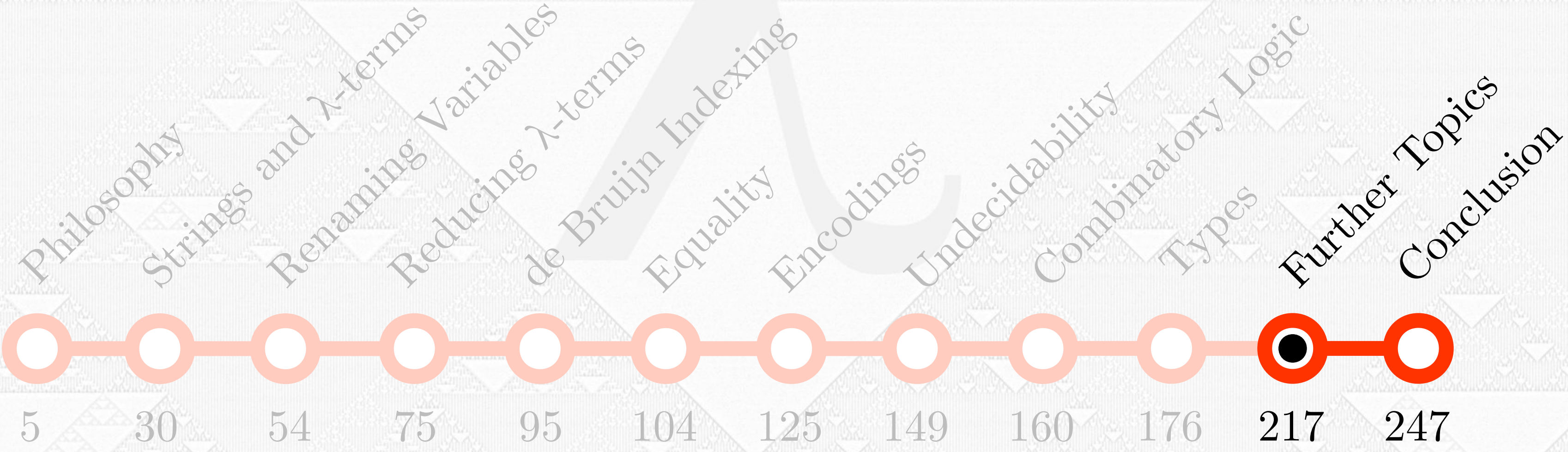
- More restricted
- Proofs may become lengthy
- Can prove $p \Rightarrow \neg\neg\, p$, but not $\neg\neg p \Rightarrow p$ (broken symmetry)
- What kind of "calculus" doesn't have real numbers?

Pros

- More restricted
- No need to speculate about "truth" or "existence"
- Proofs produce evidence
- Proofs are programs
- Classical axioms optional

Subway Map

- Our journey is nearing its end. It's time to flash some teasers on further topics



A Peek at the Curry-Howard Correspondence

Simply Typed LC	$\Longleftrightarrow$	Cartesian Closed Categories
Propositional Calculus	$\Longleftrightarrow$	Heyting Algebras
Dependent Type Theory	$\Longleftrightarrow$	Intuitionistic Logic
Monads	$\Longleftrightarrow$	Modal Logic
Continuations	$\Longleftrightarrow$	Gödel-Gentzen Translation
Programs	$\Longleftrightarrow$	Proofs

A Peek at the Curry-Howard Correspondence

Simply Typed LC	$\Longleftrightarrow$	Cartesian Closed Categories
Propositional Calculus	$\Longleftrightarrow$	Heyting Algebras
Dependent Type Theory	$\Longleftrightarrow$	Intuitionistic Logic
Monads	$\Longleftrightarrow$	Modal Logic
Continuations	$\Longleftrightarrow$	Gödel-Gentzen Translation
Programs	$\Longleftrightarrow$	Proofs

Categories

- A *category* (a type of multigraph that generalises monoids) has:
  1) *objects* and *morphisms*;
  2) *domain* and *codomain* objects x and y for each morphism f, denoted as $f : x \to y$;
  3) *composed morphisms* $g \circ f$ when $g : y \to z$ and $f : x \to y$;
  4) for each object x, the *identity morphism* 1_x s.t. $f \circ 1_x = f$ and $1_z \circ g = g$, when $f : y \to x$, $g : x \to z$; and
  5) *associative composition*: $(h \circ g) \circ f = h \circ (g \circ f)$ (when defined)

Products and Exponentials

$$
\begin{tikzcd}
 & C \arrow[dl, "f_0"'] \arrow[d, dashed, "\langle f_0, f_1 \rangle"] \arrow[dr, "f_1"] & \\
A & A \times B \arrow[l, "\pi_0"] \arrow[r, "\pi_1"'] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[d, dashed, "\bar{f}"'] \\
C^B
\end{tikzcd}
\qquad
\begin{tikzcd}
A \times B \arrow[d, dashed, "\bar{f} \times 1_B"'] \arrow[dr, "f"] & \\
C^B \times B \arrow[r, "\epsilon"'] & C
\end{tikzcd}
$$

$$
\begin{cases}
f_0 = \pi_0 \circ \langle f_0, f_1 \rangle \\
f_1 = \pi_1 \circ \langle f_0, f_1 \rangle
\end{cases}
\qquad
f = \epsilon \circ \langle \bar{f}, 1_B \rangle
$$

Cartesian Closed Categories

- The left-hand side diagram on the previous slide is the *limit* (we don't need the formal definition here) over an *index category* with two objects and zero non-identity morphisms
- The limit over the empty category is called a *terminal object*
  - It's an object that has *exactly one incoming morphism* from any other object in the category. It's like the empty product
- A category that has (finite) products (including a terminal object) exponentials is called *Cartesian closed*

STLC Forms a Cartesian Closed Category

- The *category of types* has:
  - (Non-dependent or *simple*) *types* as objects
  - *β-equivalence classes* $[M]_\beta \stackrel{\text{def}}{=} \{N | M =_\beta N\}$ *of combinators* as morphisms
  - Identities $[\lambda x.x]_\beta$ and compositions $[M]_\beta \circ [N]_\beta \stackrel{\text{def}}{=} [\lambda x.M(Nx)]_\beta$
  - Products (see p.184), exponentials $B^A \stackrel{\text{def}}{=} A \to B$, and terminal object $\top$
  - eval $\stackrel{\text{def}}{=} \lambda p.\pi_0 p(\pi_1 p) : B^A \times A \to B$
  - curry $\stackrel{\text{def}}{=} \lambda fxy.f\langle x,y\rangle : (A \times B \to C) \to A \to B \to C$

Lambda Models

- A λ-*model* is an applicative structure $(D,\bullet)$ with with mapping $[\![-]\!]$ such that for any assignments f and g:
  1) $[\![x]\!]_f = f(x)$
  2) $[\![MN]\!]_f = [\![M]\!]_f \bullet [\![N]\!]_f$
  3) $[\![\lambda x.M]\!]_f \bullet d = [\![M]\!]_{f[x:=d]}$ $\quad ((f[x:=d])(y) = d$ if $x = y$ and $f(y)$ o/w)
  4) $[\![M]\!]_f = [\![M]\!]_g$, $\quad$ if $f(x) = g(x)$ for every x in D
  5) $[\![\lambda x.M]\!]_f = [\![\lambda y.M[x:=y]]\!]_f$, $\quad$ if y is not free in M ($\equiv_\alpha$ implies $=$ in D)
  6) $[\![\lambda x.M]\!]_f = [\![\lambda x.N]\!]_f$, $\quad$ if $[\![M]\!]_{f[x:=d]} = [\![N]\!]_{f[x:=d]}$ for every d in D

Sidenote: Alternative Definition

- Alternatively, a λ-model can be defined as a combinatory algebra $(D,\bullet,k,s)$ such that
  - It satisfies Curry's axioms on p. 167
  - It satisfies the following property (*weak extensionality*): For any X and Y *s.t.* $XZ = YZ$ for all Z, $D \models \mathbf{S}(\mathbf{KI})X = \mathbf{S}(\mathbf{KI})Y$
  - In LC, this is expressed as the rule (ξ) $M = N \vdash \lambda x.M = \lambda x.M$
  - It's the same as rule number 6 on the previous slide
- This version can be useful for theoretical purposes

Challenges in Modelling

- Modelling λ-exprs naïvely as functions doesn't work. Take $\lambda x.xx$
  - $\lambda x.xx$ is not the same as $(\lambda x.x \circ x)$
  - E.g. $(\lambda x.xx)\ \mathbf{S}^{+} = \mathbf{S}^{+}\mathbf{S}^{+}$ is neither a natural number, nor a function. (The closest sensible alternatives would be $\mathbf{S}^{+}(\mathbf{S}^{+}\mathbf{Z}^{0})$ or $\mathbf{S}^{+} \circ \mathbf{S}^{+}$)
  - If the right x would be an element of a set A, then the left x would have to be an element of the function space $A^{A} = \{f \mid f : A \to A\}$
  - At least we would need $A \cong A^{A}$, but this is impossible as there are more functions $A \to A$ than elements in A...

How to Make A Non-Trivial λ-Model

- The trick is to add extra structure to the sets that we're working with and use only structure-preserving functions
- A set D with binary relation $\sqsubseteq$ is *partially ordered* if and only if
  - $x \sqsubseteq x$ for every x in D (reflexivity)
  - If $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$ (antisymmetry)
  - If $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$ (transitivity)
- We call $(D, \sqsubseteq)$ a *partially ordered* set, or *poset* among friends

Directed-Complete Partial Orders

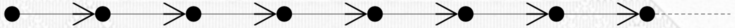
- A set $X \subseteq D$ is *directed* iff for every $x, y \in X$, there is $z \in D$ s.t. $x \sqsubseteq z$ and $y \sqsubseteq z$
  - An element $z \in D$ s.t. $x \sqsubseteq z$ for every $x \in X$ is called an *upper bound* of X
- The *least upper bound* of a set X, denoted $\sqcup X$, is an upper bound z of X such that $z \sqsubseteq w$ for any other upper bound w of X
- A poset $(D, \sqsubseteq)$ is called *(directed-)complete* if and only if
  - D has a *least element*, i.e. an element $\perp$ s.t. $\perp \sqsubseteq x$ for every x$\in D$
  - every directed subset X of D has a least upper bound $\sqcup X \in D$

Posets Over Natural Numbers

- The usual poset of the natural numbers is defined as

$$n \leq 0 \wedge (n \leq k \Rightarrow \mathbf{S}^{+}n \leq \mathbf{S}^{+}k)$$

- $(\mathbb{N},\leq)$ is not complete. For instance, there's no biggest prime number

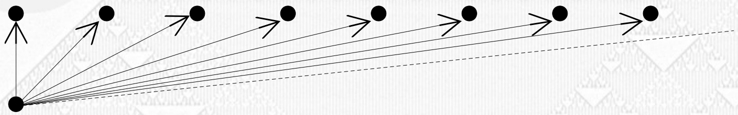


$0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq 3 \sqsubseteq 4 \sqsubseteq 5 \sqsubseteq \ldots$

- *Flat natural numbers* are the set $\mathbb{N}^{+} \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ ordered with

$$n \sqsubseteq k \Leftrightarrow (n = \perp \vee n = k)$$

- This poset is complete as every natural number n is a least upper bound of $\{n,\perp\}$



$\perp \sqsubseteq 0, \perp \sqsubseteq 1, \perp \sqsubseteq 2, \ldots$

Continuous Function Spaces

- Let $(D,\sqsubseteq)$ and $(E,\preccurlyeq)$ be DCPOs. Denote the set of continuous functions from D to E as $[D\rightarrow E]$. It's a proper subset of E^D
- $f{:}D\rightarrow E$ is *continuous* iff $f(\sqcup X) = \sqcup f[X]$, for every directed $X\subseteq D$
- $f{:}D\rightarrow E$ is *monotone (increasing)* iff $x \sqsubseteq y$ implies $f(x) \preccurlyeq f(y)$
- *Under Scott Topology, continuity implies monotonicity*
- $[D\rightarrow E]$ can be made a DCPO by defining

  $f \leq g$ if and only if $f(x) \preccurlyeq g(x)$ for every x in D

Why Flat Natural Numbers?

- The ordering $\sqsubseteq$ measures "definedness" of functions
  - e.g. $[\![\lambda n.\bot]\!]_f \sqsubset [\![\lambda n.1/n]\!]_f \sqsubset [\![\lambda n.n]\!]_f$ (consider $n = 0$, $n = \bot$)
- Bottom ($\bot$) represents an undefined/non-normalising value
- The *graph* F of a function f (representing some λ-expr) is:
  - $F \stackrel{\text{def}}{=} \bigcup_{n\in\mathbb{N}} F_n$, where $F_0 \stackrel{\text{def}}{=} \emptyset$ and $F_{n+1} \stackrel{\text{def}}{=} \{(n, f(n))\} \cup F_n$
  - $F_n \sqsubseteq F_{n+1}$ for every $n \in \mathbb{N}$, so the sequence (F_n) is monotone
  - F is the *least fixpoint* of the sequence (F_n), "$F_n \to F$ as $n \to \infty$"

Products, Application, and Abstraction

- Let $(D,\sqsubseteq)$ and $(E,\preccurlyeq)$ be DCPOs. Their Cartesian product $D\times E$ is a DCPO with the following ordering:

  $(x,y) \leq (x',y')$ if and only if $x \sqsubseteq x'$ and $y \preccurlyeq x'$

- Theorem: A function $D\times E\to F$ is continuous iff it's continuous w.r.t. its both arguments separately
- Let $\mathrm{eval}(f,x) \overset{\text{def}}{=} f(x)$. It's continuous.
- Let $\mathrm{curry}(g)(x)(y) \overset{\text{def}}{=} g(x,y)$ for $g{:}D\times E\to F$. It's also continuous.

DCPOs Form a Cartesian Closed Category

- For any $f \in [D \times E \to F]$, $x \in D$, and $y \in E$,

  $$\mathrm{eval}(\mathrm{curry}(f)(x), 1_E y) = \mathrm{curry}(f)(x)(y) = f(x,y),$$

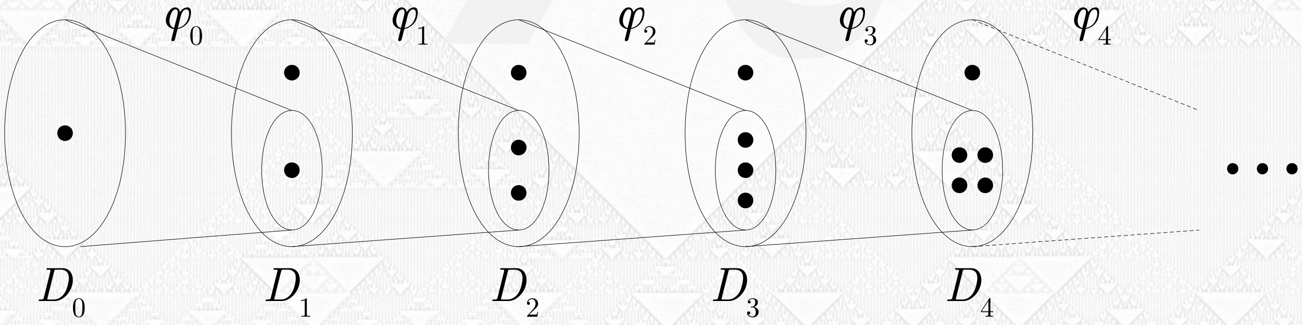
  so $F^E \stackrel{\mathrm{def}}{=} [E \to F]$ is an exponential
- A singleton DCPO D has exactly one incoming (continuous) function from any other DCPO, so D a terminal object
- The category of DCPOs and continuous functions is a *Cartesian closed Category* (CCC)

D_∞: The First Non-Trivial Model of Untyped LC

- The model D_∞ was invented by Dana Scott in early 1970'ies, almost 40 years after LC was first introduced!
  - Let $D_0 \stackrel{\text{def}}{=} \mathbb{N}^+$ and $D_{i+1} \stackrel{\text{def}}{=} [D_i \to D_i]$
  - D_∞ consists of infinite sequences $(x_0, x_1, x_2, \ldots)$ of $D_0, D_1, D_2, \ldots$
  - The rough idea is that if $[\![M]\!]_f = x$ and $[\![N]\!]_f = y$, then $[\![MN]\!]_f = x \bullet y = (x_1(y_0), x_2(y_1), x_3(y_2), \ldots)$
  - The interesting part is how to convert x_i to x_{i+1} and back as every D_{i+1} needs to be projected into D_i

Moving Between Levels

- Define
  - $\varphi_0(x) \stackrel{\text{def}}{=} f_x$, with $f_x(y) \stackrel{\text{def}}{=} x$;
  - $\psi_0(z) \stackrel{\text{def}}{=} z(\bot)$;
  - $\varphi_i(x) \stackrel{\text{def}}{=} \varphi_{i-1} \circ x \circ \psi_{i-1}$; and
  - $\psi_i(x) \stackrel{\text{def}}{=} \psi_{i-1} \circ x \circ \varphi_{i-1}$
- These properties hold:
  - $\varphi_i \in [D_i \to D_{i+1}]$
  - $\psi_i \in [D_{i+1} \to D_i]$
  - $\varphi_i \circ \psi_i \sqsubseteq 1_{Di+1}$
  - $\psi_i \circ \varphi_i = 1_{Di}$



More Machinery

- Define
  - $D_\infty \stackrel{\text{def}}{=} \{(x_0,\ x_1,\ x_2,\ \ldots) \in \Pi_{i\in\mathbb{N}} D_i \mid \psi_i(x_{i+1}) = x_i\}$
  - $\Phi_{i+p,i}(x) \stackrel{\text{def}}{=} (\psi_i \circ \psi_{i+1} \circ \ldots \circ \psi_{i+j-1})(x)$ $\quad(\Phi_{i+j,i} \in [D_{i+j} \to D_i])$
  - $\Phi_{i,i}(x) \stackrel{\text{def}}{=} x$ $\quad(\Phi_{i,i} \in [D_i \to D_i])$
  - $\Phi_{i,i+j}(x) \stackrel{\text{def}}{=} (\varphi_{i+j-1} \circ \varphi_{i+j} \circ \ldots \circ \varphi_i)(x)$ $\quad(\Phi_{i,i+j} \in [D_i \to D_{i+j}])$
  - $\Phi_{i,\infty}(x) \stackrel{\text{def}}{=} (\Phi_{i,0}(x),\ \Phi_{i,1}(x),\ \Phi_{i,2}(x),\ \ldots)$ $\quad(\Phi_{i,\infty} \in [D_i \to D_\infty])$
  - $\Phi_{\infty,i}(x) \stackrel{\text{def}}{=} x$ $\quad(\Phi_{\infty,i} \in [D_\infty \to D_i])$

Interpretation of LC in D_∞

- $D_\infty \cong [D_\infty \to D_\infty]$, which is witnessed by certain functions $F : D_\infty \to [D_\infty \to D_\infty]$ and $G : [D_\infty \to D_\infty] \to D_\infty$
- Now we can define application in D_∞ as $x \bullet y \overset{\text{def}}{=} \bigsqcup_{i \in \mathbb{N}} \Phi_{i,\infty}(x_{i+1}(y_i))$
- The interpretation with assingment f is the following:
  1) $[\![x]\!]_f \overset{\text{def}}{=} f(x)$
  2) $[\![MN]\!]_f \overset{\text{def}}{=} [\![M]\!]_f \bullet [\![N]\!]_f$
  3) $[\![\lambda x.M]\!]_f \overset{\text{def}}{=} G(d \mapsto [\![M]\!]_{f[x:=d]})$

Other Models

- Lambda-Calculus/Type theories have numerous models, such as
  - Graph models $P\omega$, D_A
  - Tree models $\mathcal{B}$, $T\omega$
  - Categorical Abstract Machine (CAM) models
  - Runtime systems of purely functional programming languages
- Generally, type theories are easier to model than pure untyped LC, because the problem with $A \cong A^A$ is avoided

Bicartesian Closed Categories

- A *Bicartesian Closed Category* is a Cartesian Closed Category which also has all finite coproducts
- *Coproduct* is a *colimit* of type • •, i.e. the dual of a product (cf. P 166), which is a *limit*

$$
\begin{tikzcd}
A \arrow[r, "\iota_0"] \arrow[dr, "f_0"'] & A+B \arrow[d, dashed, "{[f_0,f_1]}"] & B \arrow[l, "\iota_1"'] \arrow[dl, "f_1"] \\
 & C &
\end{tikzcd}
$$

$$
\begin{cases}
f_0 = [f_0,f_1] \circ \iota_0 \\
f_1 = [f_0,f_1] \circ \iota_1
\end{cases}
$$

Equalizers and Coequalizers

$$
\begin{tikzcd}
E \arrow[r, "e"] & A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \\
C \arrow[u, dashed, "u"] \arrow[ur, "z"'] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
& & C \\
B \arrow[r, shift left, "g"] \arrow[r, shift right, "f"'] & A \arrow[r, "e"'] \arrow[ur, "z"] & E \arrow[u, dashed, "u"']
\end{tikzcd}
$$

$$
\begin{cases}
f \circ e = g \circ e \\
z \quad = e \circ u
\end{cases}
\qquad
\begin{cases}
e \circ f = e \circ g \\
z \quad = u \circ e
\end{cases}
$$

Pullbacks and Pushouts

$$
\begin{tikzcd}
Y \arrow[dr, dashed, "u"] \arrow[drr, "k"] \arrow[ddr, "h"'] & & \\
& X \arrow[r, "p_B"'] \arrow[d, "p_A"] & B \arrow[d, "g"] \\
& A \arrow[r, "f"'] & C
\end{tikzcd}
\qquad
\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "g"'] & A \arrow[d, "p_A"'] \arrow[ddr, "h"] & \\
B \arrow[r, "p_B"] \arrow[drr, "k"'] & X \arrow[dr, dashed, "u"] & \\
& & Y
\end{tikzcd}
$$

$$
\begin{cases}
f \circ h = f \circ p_A \circ u \\
g \circ k = g \circ p_B \circ u
\end{cases}
\qquad
\begin{cases}
h \circ f = u \circ p_A \circ f \\
k \circ g = u \circ p_B \circ g
\end{cases}
$$

Topoi

$$
\begin{tikzcd}
A \arrow[r, tail, "f"] \arrow[d, "!"'] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "\chi_f"] \\
1 \arrow[r, "\top"'] & \Omega
\end{tikzcd}
$$

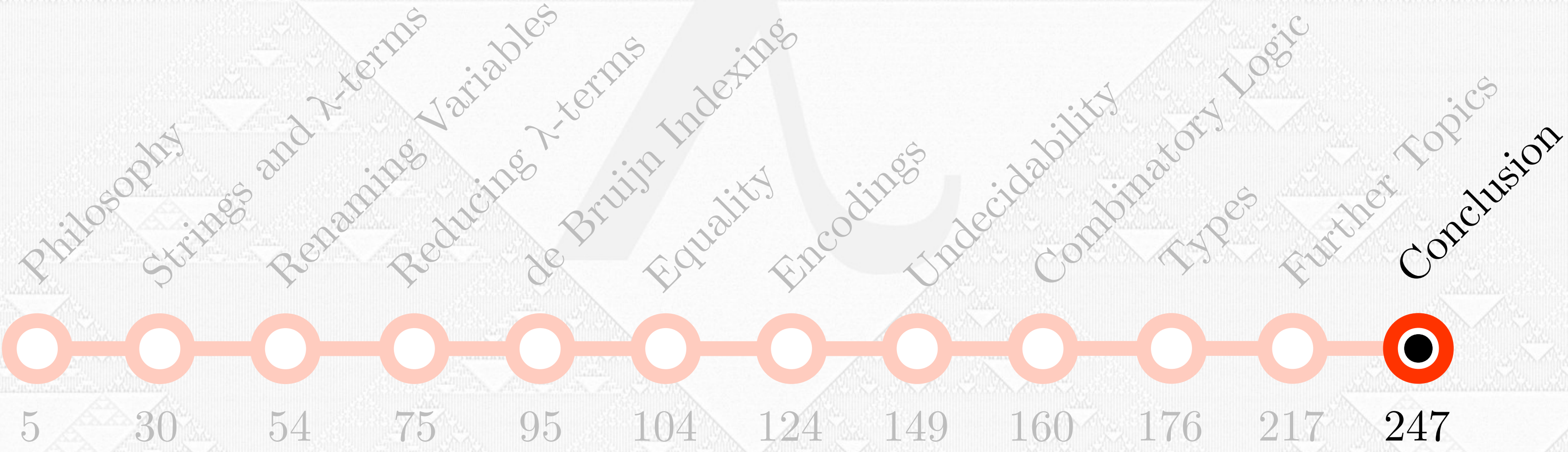
$$\chi_f \circ f = \top \circ !$$

$$
\begin{tikzcd}
 & N \arrow[loop right, "s"] \arrow[d, "h"] \\
A \arrow[ur, "z"] \arrow[dr, "x"'] & \\
 & N \arrow[loop right, "f"]
\end{tikzcd}
$$

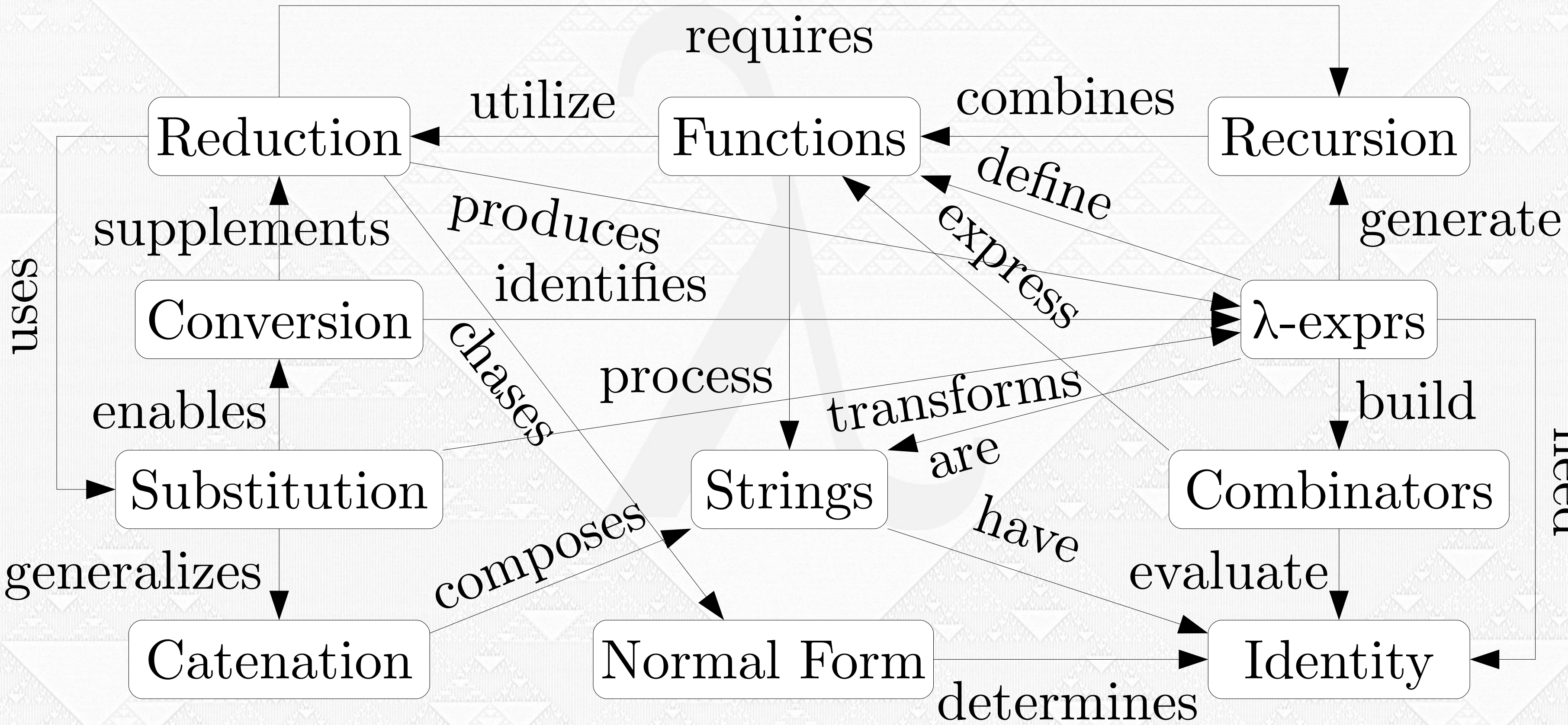
$$f^n \circ x = s^n \circ z$$

Subway Map

- It's time to wrap up



Local Map



requires
utilize
combines
Reduction
Functions
Recursion
define
express
produces
supplements
generate
identifies
uses
Conversion
λ-exprs
chases
process
transforms
enables
build
are
need
Substitution
Strings
Combinators
have
composes
generalizes
evaluate
Catenation
Normal Form
Identity
determines

Programming Terminology Inspired by LC

Concept	Definition
Higher-Order Function	A function that returns (or takes) another function.
Function Composition	A higher-order function that composes two functions.
Partial Application	Applying a single argument to a higher-order function.
Currying	Defining a higher-order function.
Recursion	Defining a function in terms of its own values.
Evaluation Strategy	Determines the order of reduction steps for an expression.
Pattern Matching	A generalization of the `if-then-else` construct.
Type	A proposition about the structure of an object.

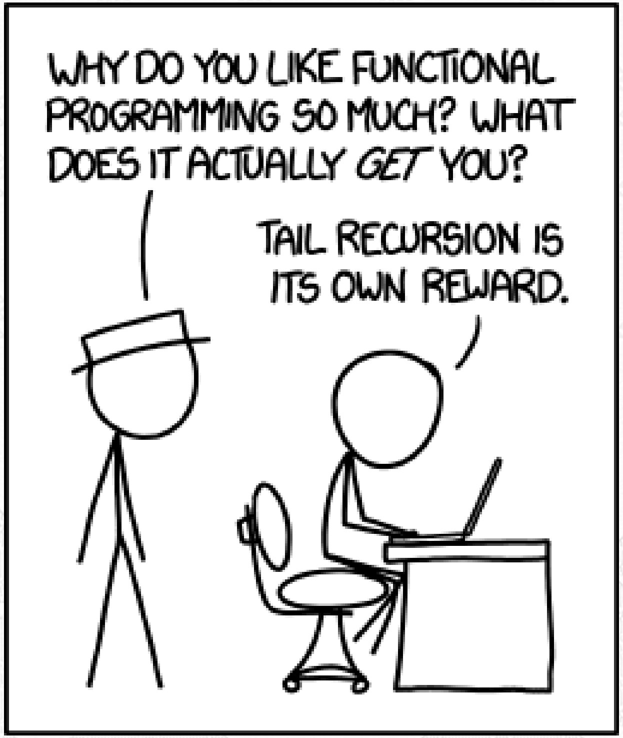
Conclusion

- We now (hopefully) understand the concept of a function from a computational (LC/CL) perspective.
  - Extensional equality of functions is undecidable in general case.
  - Halting problem is the computational analogue to the Gödel's incompleteness theorems.
- Lambda-Calculus is a deep field of mathematics with connections to many other disciplines.
  - This presentation barely scratched the surface.

Related Topics

- So, what next? We can extend LC into many directions. E.g.:
  - Q: When does an application make sense? A: Type Theory
  - Q: What kind of algebra is behind LC? A: Category Theory
  - Q: What kind of calculus is behind LC? A: Domain Theory
  - Q: How can we profit from LC in logic? A: Proof Theory
  - Q: What kind of programming is LC? A: Functional Programming
  - Q: Why is LC future-proof? A: Non-Classical Computing

Thank You!



https://xkcd.com/1270/

Warm-Up Excercises

1) Draw a parse tree (see p. 39 and 47–50) for $(\lambda x.\lambda y.\lambda z.zxw)cd\mathbf{I}$
2) Reduce this tree, step by step, to a normal form. (Cf. p. 79–82.)
3) Prove the claims about combinators on p. 118
4) Draw your own World/Local Map of LC/CL. (See p. 28, 234.)
5) Convert $(\mathbf{K}(\mathbf{C}(\mathbf{CI}c)y))d$ back to LC and find its normal form
6) $\mathbf{Y} \overset{\text{def}}{=} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ is another fixed point combinator
   Can you tell the subtle behavioral difference between $\mathbf{Y}$ and $\mathbf{\Theta}$?

Advanced Excercises

1) Develop an example that uses all the rules given in p. 161
2) Consider the Euclidean algorithm flowchart in p. 9 (and 10). Describe it using the techniques using provided in these slides
   - Hint: You may take elementary artihmetics as granted in LC
3) Can you encode the algorithm shown in p. 161 using LC/CL?
4) Can you analyze the MIU-system (see p. 3) in LC/CL?
5) Construct a Turing Machine or other interpreter for LC/CL

Type Theory Excercises

1) Consider the proofs on p. 193–197. Show that types and disjunction also form a commutative monoid
2) Show that sum types are associative and commutative.
3) Show that that $\perp$ is the neutral type. Hint: $\lambda()$, $[\_,\_]$
4) Define a lambda expression that is not polymorphic (i.e. *monomorphic*). Can you explain the difference between polymorphic and monomorphic functions?

Further Reading

- J. Roger Hindley and Jonathan P. Seldin: *Lambda-Calculus and Combinators: an Introduction*. Cambridge University Press,2008
- Nederpelt, Rob; Geuvers, Herman: *Type Theory and Formal Proof : An Introduction*, 2014
- Henk. P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*, Volume 40 of Studies in Logic: Mathematical Logic and Foundations. College Publications, 2012
- Steve Awodey: Category Theory. Second Edition. 2010.

Further Reading

- The Univalent Foundations Program: *Homotopy Type Theory.* Univalent Foundations of Mathematics, 2013. Available at: https://homotopytypetheory.org/book
- Wadler, Philip and Wen Kokke. *Programming Language Foundations in Agda.* Available at http://plfa.inf.ed.ac.uk. 2019.
- The Standford Encyclopaedia of Philosophy has excellent articles on LC and CL at https://plato.stanford.edu/
- The nLab wiki sketches some advanced topics at https://ncatlab.org