Intro to Analysis of Algorithms
Computational Foundations
Appendix
Chapter 9

Michael Soltys

CSU Channel Islands

[**Git** Date:(None) Hash:(None) Ed:3rd]

Part I
λ-calculus
(not in textbook)

The set Λ of *λ-terms* is the smallest set such that:

- $x, y, z \ldots \in \Lambda$ (*variables* are in Λ)
- if x is a variable and M is λ-term, then so is $(\lambda x.M)$ (*abstraction*)
- if M, N are λ-terms then so is (MN) (*application*)

$\mathrm{FV}(M)$ is the set of free variables of M. It is defined recursively as follows: $\mathrm{FV}(x) = \{x\}$, and $\mathrm{FV}(\lambda x.M) = \mathrm{FV}(M) - \{x\}$ and $\mathrm{FV}(MN) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$.

Terms without free variables are *closed terms* (also called *combinators*), i.e., M is closed iff $\mathrm{FV}(M) = \emptyset$.

$\mathrm{BV}(M)$ is the set of bounded variables of M. $\mathrm{BV}(x) = \emptyset$, $\mathrm{BV}(\lambda x.M) = \{x\} \cup \mathrm{BV}(M)$ and $\mathrm{BV}(M) \cup \mathrm{BV}(N)$.

Ex. $\lambda z.z$ is closed, and $\mathrm{FV}(z\lambda x.x) = \{z\}$ while $\mathrm{BV}(z\lambda x.x) = \{x\}$. On the other hand $\mathrm{BV}(x\lambda x.x) = \mathrm{BV}(x\lambda x.x) = \{x\}$.

It is important to realize that two formulas are essentially the same if they only differ in the names of bounded variables, e.g., $\lambda x.x$ and $\lambda y.y$ represent (in some sense) the same object. To make this concept precise, we introduce the notion of α-*equality*, denoted $=_\alpha$.

$M =_\alpha N$ if $M = N = x$

Note that the equalith on the right ($M = N = x$) is *syntactic* equality and x can be any variable.

$M =_\alpha N$ if $M = M_1 M_2$ and $N = N_1 N_2$ and $M_1 =_\alpha N_1$ and $M_2 =_\alpha N_2$.

Also, $M =_\alpha N$ if $M = \lambda x.M_1$ and $N = \lambda x.N_1$ and $M_1 =_\alpha N_1$.

Finally, $M =_\alpha N$ if $M = \lambda x.M_1$ and $N = \lambda y.N_1$ and there is a *new* variable z such that $M_1\{x \mapsto z\} =_\alpha N_1\{y \mapsto z\}$.

Here $M\{x \mapsto N\}$ denotes the λ-term M where every *free* instance of x has been replaced by the λ-term N, in such a way that no free variable u of N has been "caught" in the scope of some λu. If z is new, it will never be caught.

We shall soon give a formal definition of substitution.

But first: $=_\alpha$ is an equivalence relation.

Ex. $\lambda x.x =_\alpha \lambda y.y$, $\lambda x.\lambda y.xy =_\alpha \lambda z_1.\lambda z_2.z_1 z_2$ and $(\lambda x.x)z =_\alpha (\lambda y.y)z$.

Thus, we think of λ-terms in terms of their equivalence classes wrt $=_\alpha$ relation.

We now define the notion of computation: a *redex* is a term of the form $(\lambda x.M)N$. The idea is to apply the function $\lambda x.M$ to the argument N. We do this as follows:

$$(\lambda x.M)N \to_\beta M\{x \mapsto N\}$$

This is the so called *β-reduction* rule. We write $M \to_\beta M'$ to indicate that M reduces to M'.

Ex. $(\lambda x.x)y \to_\beta x\{x \mapsto y\} = y$

(again, note that the equality is a syntactic equality)

$(\lambda x.\lambda y.x)(\lambda x.x)u \to_\beta (\lambda y.\lambda z.z)u \to_\beta \lambda z.z$

(application associates to the left, i.e., $MNP = (MN)P$)

Ex. $(\lambda x.\lambda y.xy)(\lambda x.x) \to_\beta \lambda y.(\lambda x.x)y \to_\beta \lambda y.y$

The symbol $\to_\beta^*$ means zero or more applications of $\to_\beta$; from the previous example, $(\lambda x.\lambda y.xy)(\lambda x.x) \to_\beta^* \lambda y.y$.

We use the word *reduce* but this does not mean that the terms necessarily get simpler/smaller.

Ex. $(\lambda x.xx)(\lambda xyz.xz(yz)) \to_\beta (\lambda xyz.xz(yz))(\lambda xyz.xz(yz))$

(note that λxyz abbreviates $\lambda x.\lambda y.\lambda z$, and that abstractions associate to the right, i.e., $\lambda xyz.M$ is $\lambda x.(\lambda y.(\lambda z.M))$)

$$
\begin{aligned}
(\lambda x.xx)(\lambda y.yx)z &= ((\lambda x.xx)(\lambda y.yx))z \\
&\qquad\qquad\qquad \text{[application left associates]} \\
&\to_\beta ((xx)\{x \mapsto (\lambda y.yx)\})z \qquad \text{[substitution]} \\
&= ((\lambda y.yx)(\lambda y.yx))z \\
&\to_\beta ((yx)\{y \mapsto (\lambda y.yx)\})z \qquad \text{[substitution]} \\
&= ((\lambda y.yx)x)z \\
&\to_\beta ((yx)\{y \mapsto x\})z \qquad \text{[substitution]} \\
&= (xx)z = xxz \qquad \text{[application left associates]}
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.(\lambda y.(xy))y)z &\to_\beta (\lambda x.((xy)\{y \mapsto y\}))z \\
&= (\lambda x.(xy))z \\
&\to_\beta (xy)\{x \mapsto z\} = zy
\end{aligned}
$$

$$
\begin{aligned}
((\lambda x.xx)(\lambda y.y))(\lambda y.y) &\to_\beta ((xx)\{x \mapsto (\lambda y.y)\})(\lambda y.y) \\
&= ((\lambda y.y)(\lambda y.y))(\lambda y.y) \\
&\to_\beta (y\{y \mapsto (\lambda y.y)\})(\lambda y.y) \\
&= (\lambda y.y)(\lambda y.y) \\
&= (\lambda y.y) \quad \text{[just repeating previous line]}
\end{aligned}
$$

$$
\begin{aligned}
(((\lambda x.\lambda y(xy))(\lambda y.y))w) &= (((\lambda x.\lambda v.(xv))(\lambda y.y))w) \\
&\qquad \text{[use } =_\alpha \text{ so } y \text{ not ``caught'' by } \lambda y] \\
&\to_\beta ((\lambda v.(xv))\{x \mapsto (\lambda y.y)\})w \\
&= (\lambda v.((\lambda y.y)v))w \\
&\to_\beta (\lambda v.v)w \\
&\to_\beta w
\end{aligned}
$$

We now give a precise definition of *substitution* $M\{x \mapsto N\}$ by structural induction on M.

$x\{x \mapsto N\}N = N$

$y\{x \mapsto N\} = y$

$(PQ)\{x \mapsto N\} = (P\{x \mapsto N\})(Q\{x \mapsto N\})$

$(\lambda x.P)\{x \mapsto N\} = \lambda x.P$

$(\lambda y.P)\{x \mapsto N\} = \lambda y.(P\{x \mapsto N\})$ if $y \notin \mathrm{FV}(N)$ or $x \notin \mathrm{FV}(P)$

$(\lambda y.P)\{x \mapsto N\} = (\lambda z.P\{y \mapsto z\})\{x \mapsto N\}$ otherwise and z is a new variable

Ex.

$$
\begin{aligned}
(\lambda z.yz)\{y \mapsto z\} &=_\alpha (\lambda x.(yz)\{z \mapsto x\})\{y \mapsto z\} \\
&=_\alpha (\lambda x.((y\{z \mapsto x\})(z\{z \mapsto x\})))\{y \mapsto z\} \\
&=_\alpha (\lambda x.(yx))\{y \mapsto z\} \\
&=_\alpha \lambda x.(yx)\{y \mapsto z\} \\
&=_\alpha \lambda x.((y\{y \mapsto z\})(x\{y \mapsto z\})) \\
&=_\alpha \lambda x.(zy)
\end{aligned}
$$

Property: If $x \in \mathrm{FV}(P)$, then

$$(M\{x \mapsto N\})\{y \mapsto P\} =_\alpha (M\{y \mapsto P\})\{x \mapsto N\{y \mapsto P\}\}$$

A *normal form* is a term that does not contain any redexes.

A term that can be reduced to normal form is called *normalizable*.

Ex. $\lambda abc.((\lambda x.a(\lambda y.xy))bc) \to_\beta \lambda abc.(a(\lambda y.by)c)$ where the last term is in normal form (bec applications associate to the left)

Some terms are not normalizable, e.g., $(\lambda x.xx)(\lambda x.xx)$.

A term M is *strongly normalizable* (or *terminating*) if all reduction sequences starting from M are finite.

Weak head normal form: stop reducing when there are no redexe left, but without reducing under an abstraction.

Ex. $\lambda abc.((\lambda x.a(\lambda xy))bc)$ is in weak head normal form.

FACT: Our reduction relation $\rightarrow_\beta$ is *confluent* because whenever $M \rightarrow_\beta M_1$ and $M \rightarrow_\beta M_2$, then there exists a term M_3 such that $M_1 \rightarrow_\beta M_3$ and $M_2 \rightarrow_\beta M_3$.

Corollary: Each λ-term has at most one normal form.

Proof: Suppose that a term M has more than one normal form; i.e., $M \rightarrow_\beta^* M_1$ and $M \rightarrow_\beta^* M_2$, where M_1 and M_2 are in normal form. Then they should both be reducible to a common M_3 (by confluence), but if they are in normal form that cannot be done. Contradiction—hence there can be at most one normal form.

Church's numerals:

$$\bar{0} = \lambda x.\lambda y.y$$
$$\bar{1} = \lambda x.\lambda y.xy$$
$$\bar{2} = \lambda x.\lambda y.x(xy)$$
$$\bar{3} = \lambda x.\lambda y.x(x(xy))$$
$$\vdots$$
$$\bar{n} = \lambda x.\lambda y.\underbrace{x(x(x\ldots(x}_{n}\,y)\ldots))$$
$$\vdots$$

Alonzo Church

Consider $S := \lambda xyz.y(xyz)$

$$
\begin{aligned}
S\bar{n} &= S(\lambda xy.\underbrace{x(x(x\ldots(x}_{n}\,y)\ldots))) \\
&\to_\beta \lambda yz.y(\lambda xy.\underbrace{x(x(x\ldots(x}_{n}\,y)\ldots))yz) \\
&=_\alpha \lambda yz.y(\lambda xw.\underbrace{x(x(x\ldots(x}_{n}\,w)\ldots))yz) \\
&\to_\beta \lambda yz.y(\lambda w.\underbrace{y(y(y\ldots(y}_{n}\,w)\ldots))z) \\
&\to_\beta \lambda yz.\overbrace{y(\underbrace{y(y(y\ldots(y}_{n}\,z)\ldots)))}^{n+1} =_\alpha \overline{n+1}
\end{aligned}
$$

so $S(\bar{n}) = \overline{n+1}$, i.e., S is the successor fn.

Define ADD $:= \lambda xyab.(xa)(yab)$.

$$\begin{aligned}
\mathrm{ADD}\bar{n}\bar{m} &\to_\beta (\lambda yab.(\bar{n}a)(yab))\bar{m} \\
&\to_\beta \lambda ab.(\ \bar{n}a\ )(\ \bar{m}a\ b) \\
&\to_\beta \lambda ab.(\ \lambda y\ \underbrace{a(a(a\ldots(a}_{n}\,y)\ldots))\ )[(\ \lambda y\ \underbrace{a(a(a\ldots(a}_{m}\,y)\ldots))\ )b] \\
&\to_\beta \lambda ab.(\ \lambda y\ \underbrace{a(a(a\ldots(a}_{n}\,y)\ldots))\ )[\ \underbrace{a(a(a\ldots(a}_{m}\,b)\ldots))\ ] \\
&\to_\beta \lambda ab.(\underbrace{\underbrace{a(a(\ldots(a}_{n}\,\underbrace{(a(a\ldots(a}_{m}\,b)\ldots)))\ldots)))}_{n+m} \\
&=_\alpha \overline{n+m}
\end{aligned}$$

Part II
Recursive Functions
(not in textbook)

A *partial function* is a function

$$f : (\mathbb{N} \cup \{\infty\})^n \longrightarrow \mathbb{N} \cup \{\infty\}, \quad n \geq 0$$

such that $f(c_1, \ldots, c_n) = \infty$ if some $c_i = \infty$.

$\text{Domain}(f) = \{\vec{x} \in \mathbb{N}^n : f(\vec{x}) \neq \infty\}$ where $\vec{x} = (x_1, \ldots, x_n)$.

f is *total* if $\text{Domain}(f) = \mathbb{N}^n$, i.e., f is always defined if its arguments are defined.

A *Register Machine (RM)* is a computational model specified by a program $P = \langle c_0, c_1, \ldots, c_{h-1} \rangle$, consisting of a finite sequence of commands.

The commands operate on registers $R_1, R_2, R_3, \ldots$, each capable of storing an arbitrary natural number.

command	abbrev.	parameters
$R_i \leftarrow 0$	Z_i	$i = 1, 2, \ldots$
$R_i \leftarrow R_i + 1$	S_i	$i = 1, 2, \ldots$
goto k if $R_i = R_j$	J_{ijk}	$i, j = 1, 2, \ldots$ & $k = 0, 1, 2, \ldots h$

An example RM program that copies R_i into R_j:

c_0:	$R_j \leftarrow 0$	Z_j
c_1:	goto 4 if $R_i = R_j$	J_{ij4}
c_2:	$R_j \leftarrow R_j + 1$	S_j
c_3:	goto 1 if $R_1 = R_1$	J_{111}
c_4:		

Formally, the program is $\langle Z_j, J_{ij4}, S_j, J_{111} \rangle$.

Semantics of RM's

A *state* is an $m+1$-tuple

$$\langle K, R_1, \ldots, R_m \rangle$$

of natural numbers, where K is the instruction counter (i.e., the number of the next command to be executed), and $R_1, \ldots, R_m$ are the current values of the registers (m is the max register index referred to in the program).

Given a state $s = \langle K, R_1, \ldots, R_m \rangle$ and a program $P = \langle c_0, c_1, \ldots, c_{h-1} \rangle$, the next state, $s' = \text{Next}_P(s)$ is the state resulting when command c_K is applied to the register values given by s.

We say that s is a *halting state* if $K = h$, and in this case $s' = s$.

Suppose the state $s = \langle K, R_1, \ldots, R_m \rangle$ and the command c_k is S_j, where $1 \leq j \leq m$. Then,

$$\text{Next}_P(s) = \langle K + 1, R_1, \ldots, R_{j-1}, R_j + 1, R_{j+1}, \ldots, R_m \rangle$$

Ex. Give a formal definition of the function Next_P for the cases in which c_K is Z_i and J_{ijk}.

A computation of a program P is a finite or infinite sequence $s_0, s_1, \ldots$ of states such that $s_{i+1} = \text{Next}_P(s_i)$.

If the sequence is finite, then the last state must be a halting state, in which case that computation is halting—we say that P is halting starting in state s_0.

A program P computes a (partial) function $f(a_1, \ldots, a_n)$ as follows. Initially place $a_1, \ldots, a_n$ in $R_1, \ldots, R_n$ and set all other registers to 0. Start execution with c_0, i.e., the initial state is

$$s_0 = \langle 0, a_1, \ldots, a_n, 0, \ldots, 0 \rangle$$

If P halts in s_0, the final value of R_1 must be $f(a_1, \ldots, a_n)$ (which then must be defined). If P fails to halt, then $f(a_1, \ldots, a_n) = \infty$.

We say f is *RM-computable* (or just *computable*) if f is computed by some RM program.

Church's Thesis: Every algorithmically computable function is RM computable.

Ex. Show $P = \langle J_{234}, S_1, S_3, J_{110} \rangle$ computes $f(x, y) = x + y$.

Ex. Write RM programs that compute $f_1(x) = x \dot{-} 1$ and $f_2(x, y) = x \cdot y$. Be sure to respect the input/output conventions for RMs.

f is defined from g and h by *primitive recursion (pr)* if

$$f(\vec{x}, 0) = g(\vec{x})$$
$$f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}, y))$$

we allow $n = 0$ so $\vec{x}$ could be missing. The following high-level program computes f from g, h by pr:

$u \leftarrow g(\vec{x})$
for $z : 0 \ldots (y-1)$
    $u \leftarrow h(\vec{x}, z, u)$
end for

$f_+(x, y) = x + y$ can be define by pr as follows:

$$x + 0 = x$$
$$x + (y+1) = (x+y) + 1$$

In this case $g(x) = x$ and $h(x, y, z) = z + 1$.

f is defined from g and $h_1, \ldots, h_m$ by *composition* if $f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$, where $f, h_1, \ldots, h_m$ are each n-ary and g is m-ary.

Initial functions:

Z	0-ary constant function equal to 0
S	$S(x) = x + 1$
$\pi_{n,i}(x_1, \ldots, x_n) = x_i$	infinite class of *projection* functions

f is *primitive recursive (pr)* if f can be obtained from the initial functions by finitely many applications of primitive recursion and composition.

Proposition: Every pr function is total.

Theorem: Every pr function is RM-computable.

Proof: We show every pr f is computable by a program which upon halting leaves all registers 0 except R_1 (which contains the output). We do this by induction on the def of pr fns.

Base case: each initial fn is computable by such an RM program.

Z is just $\langle Z_1 \rangle$
$S(x) = x + 1$ is $\langle S_1 \rangle$
$\pi_{n,i}(x_1, \ldots, x_n)$ depends on whether $i = 1$ or $i \neq 1$. In the first case the program is $\langle Z_2, \ldots, Z_n \rangle$. In the second case it is $\langle \underbrace{Z_1, J_{i14}, S_1, J_{111}}_{\text{“Copy } R_i \text{ to } R_1\text{”}}, Z_2, \ldots, Z_n \rangle$.

Induction step: Composition: Assume that $g, h_1, \ldots, h_m$ are computable by programs $P_g, P_{h_1}, \ldots, P_{h_m}$, where these programs leave all registers zero except R_1.

We must show that f is computable by a program P_f where $f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$. At the start $\vec{x} = x_1, \ldots, x_n$ are in registers $R_1, \ldots, R_n$, with all other registers zero.

Program P_f must proceed (at a high level) as follows: it must move $\vec{x}$ out of the way, to some high-numbered registers. Then it must compute $h_i(\vec{x})$, for each i, by moving a $\vec{x}$ to $R_1, \ldots, R_n$, simulating P_{h_i}, and then moving the result from R_1 out of the way.

At the end it must move the value of $h_i(\vec{x})$ to R_i, for each i, and simulate P_g.

Primitive recursion: implement the high-level program given following the definition of pr.

Is the converse true? Is every computable fn pr?

No. Some computable fns are not total.

Is every total computable fn pr?

No. We can show this by a diagonal argument: each pr fn can be encoded as a number; let $f_1, f_2, f_3, \ldots$ be the list of all pr functions.

We are only interested in unary fns, so if f_i has arity greater than one, we replace it by S (the unary successor function). Let the new list be $g_1, g_2, g_3, \ldots$, where $g_i = f_i$ if f_i was unary, and $g_i = S$ otherwise.

Let $U(x, y) = g_x(y)$, so U is a total computable fn. However, U is not pr; for suppose that it is. Then so is $D(x) = S(U(x, x))$. If U were pr, so would be D.

But if D is pr, then $D = g_e$ for some e. This gives us a contradiction, since $g_e(e) = D(e) = g_e(e) + 1$.

We can in fact give a concrete example of a total computable fn, which is not primitive recursive.

The *Ackermann function* is defined as follows:

$$A_0(x) = \begin{cases} x+1 & \text{if } x = 0 \text{ or } x = 1 \\ x+2 & \text{otherwise} \end{cases}$$

and $A_{n+1}(0) = 1$ and $A_{n+1}(x+1) = A_n(A_{n+1}(x))$.

We can prove by induction on n that $A_n(x)$ is total for all n, and therefore so is $A(n,x) = A_n(x)$. Also, A is computable since it can be computed with an RM program following the recursion given above.

Note that $A_2(x) = 2^x$ while $A_3(x) = \left.2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\right\}$ of height x.

Lemma: For each n, A_n is pr.

Proof: By induction on n; the work is in the base case.

Fact: For every pr fn $h(\vec{x})$, there exists an n so that for sufficiently large B, if $\min\{\vec{x}\} > B$ then $h(\vec{x}) < A_n(\max\{\vec{x}\})$, i.e., A_n dominates h.

Then, if $A(n, x) = A_n(x)$, then A is not pr; in fact, $F(x) = A(x, x)$ is not pr, since A cannot dominate itself.

We let μ denote the least number operator. More precisely, $f(\vec{x}) = \mu y[g(\vec{x}, y) = 0]$ if

1. $f(\vec{x})$ is the least number b such that $g(\vec{x}, b) = 0$,
2. $g(\vec{x}, y) \neq \infty$ for $i < b$.

$f(\vec{x}) = \infty$ if no such b exists.

If g is computable and $f(\vec{x}) = \mu y[g(\vec{x}, y) = 0]$ then f is also computable:

for $y = 0 \ldots \infty$
    if $g(\vec{x}, y) = 0$ then
        output y and exit
    end if
end for

A function f is *recursive* if f can be obtained from the initial functions by finitely many applications of composition, primitive recursion, and minimization.

Theorem: Every recursive function is computable.

In the 1940s Kleene showed that the converse of the above theorem is also true: every computable function is recursive.

We next prove this converse: every computable fn is recursive.

First we assign a *Gödel number* $\#P$ to every program P:

command c	Z_i	S_i	J_{ijm}
code $\#c$	2^i	3^i	$5^i 7^j 11^m$

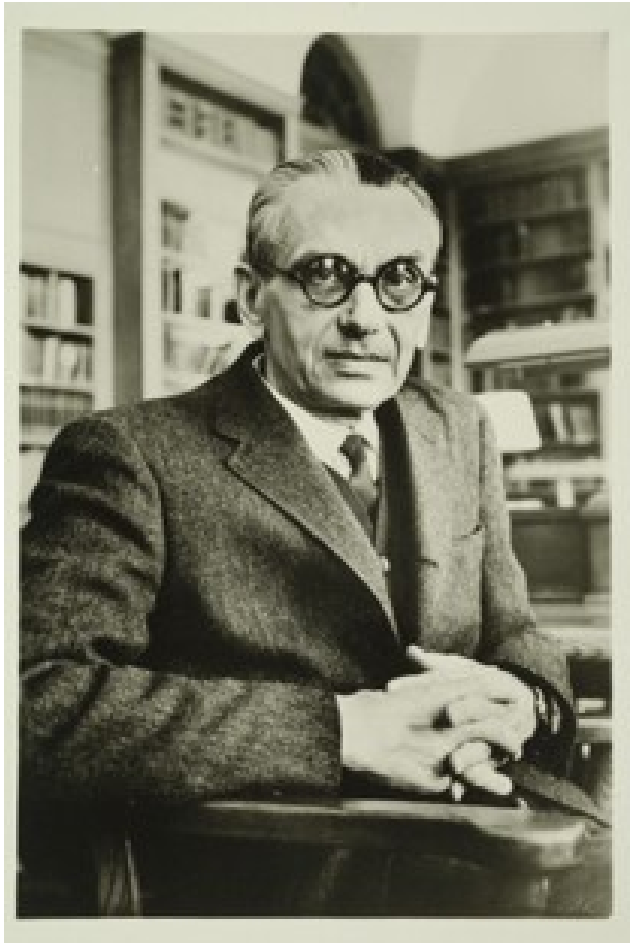
By the **Fundamental Theorem of Arithmetic** these codes are unique.

Let $p_0 < p_1 < p_2 < \cdots = 2 < 3 < 5 < \cdots$ be the list of all primes, in order. Then, if $P = \langle c_0, c_1, \ldots, c_{h-1} \rangle$,

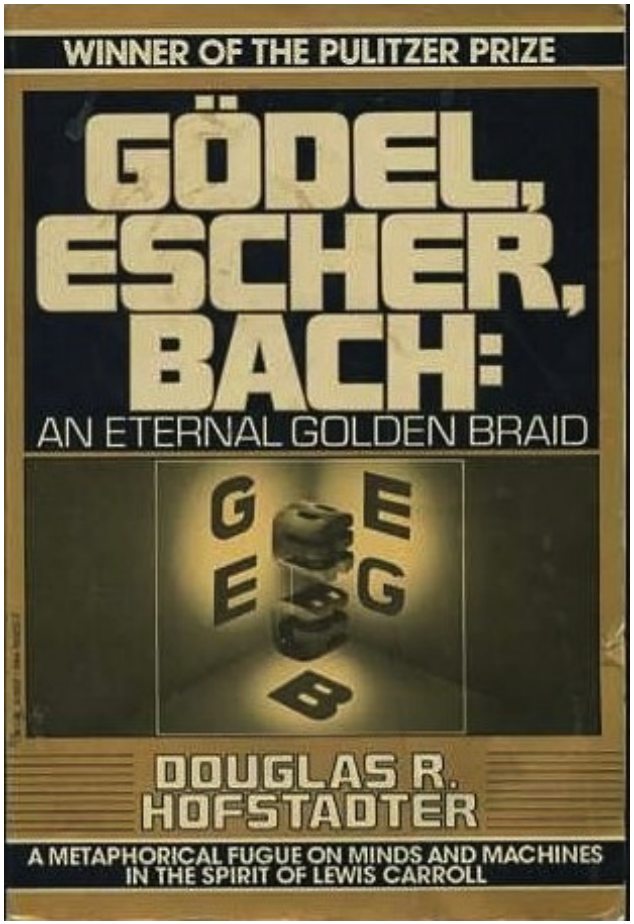
$$\#P = p_0^{\#c_0} p_1^{\#c_1} \cdots p_{h-1}^{\#c_{h-1}}$$

Encode the state s of a program as follows:

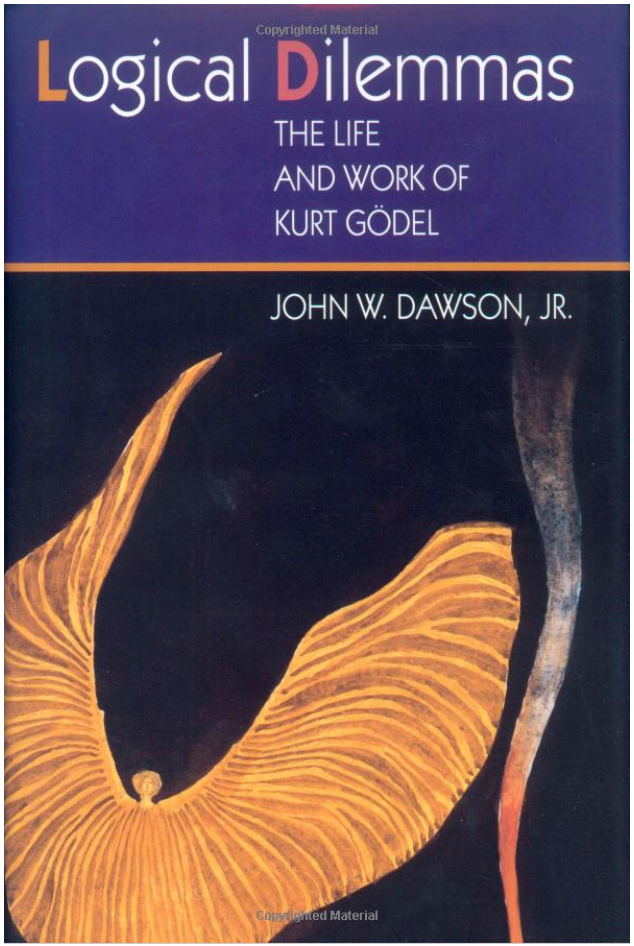
$$\#s = \#\langle K, R_1, \ldots, R_m \rangle = p_0^k p_1^{R_1} \cdots p_m^{R_m}$$



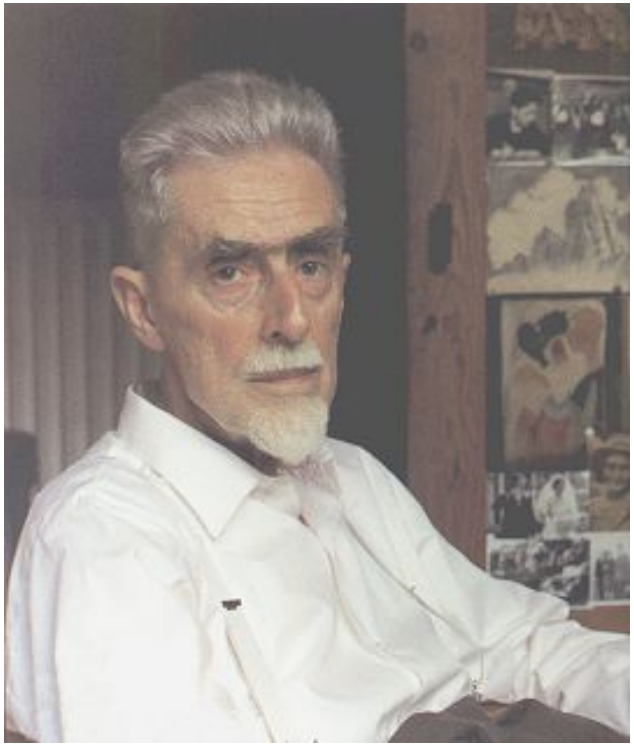
Kurt Gödel



Gödel, Escher, Bach

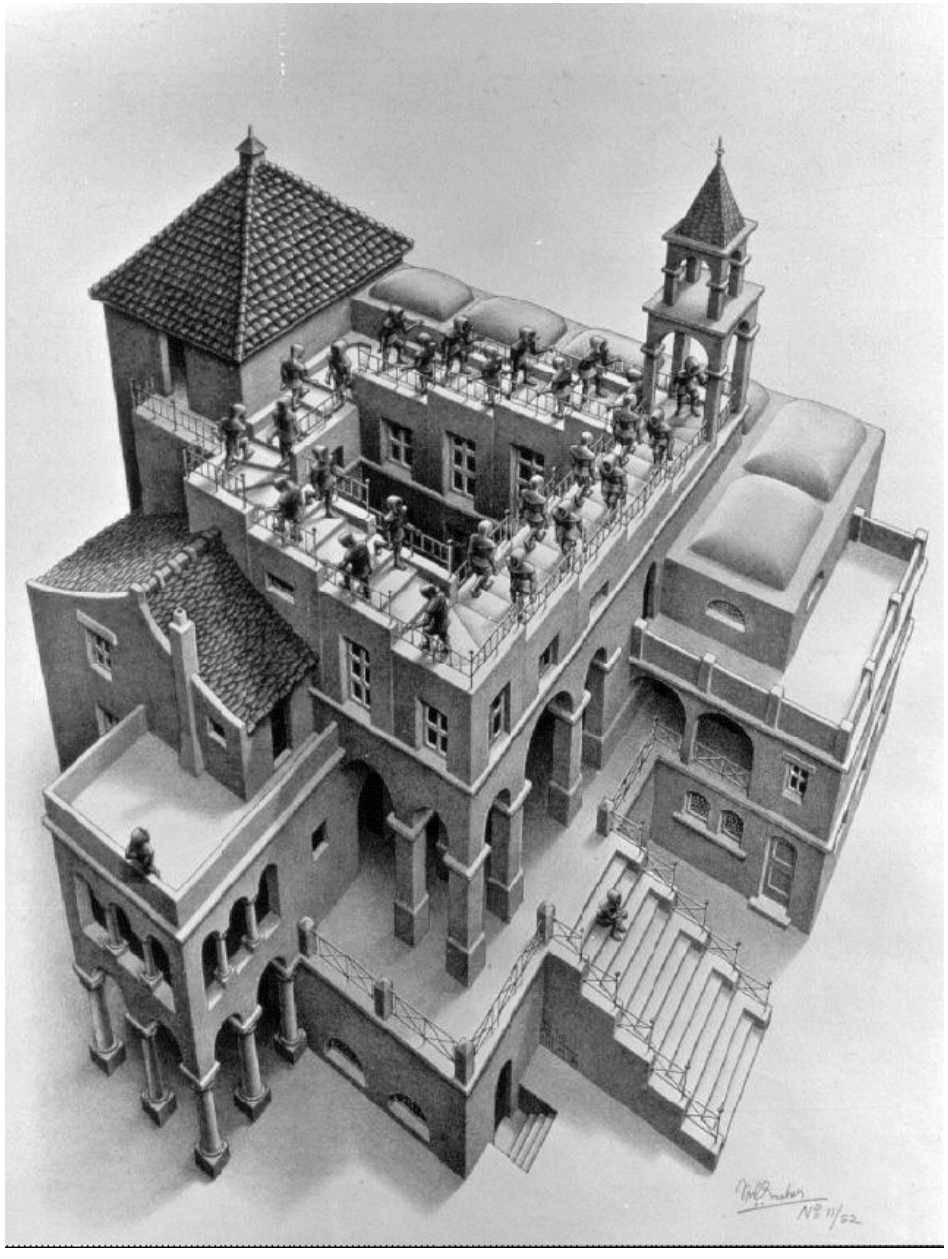


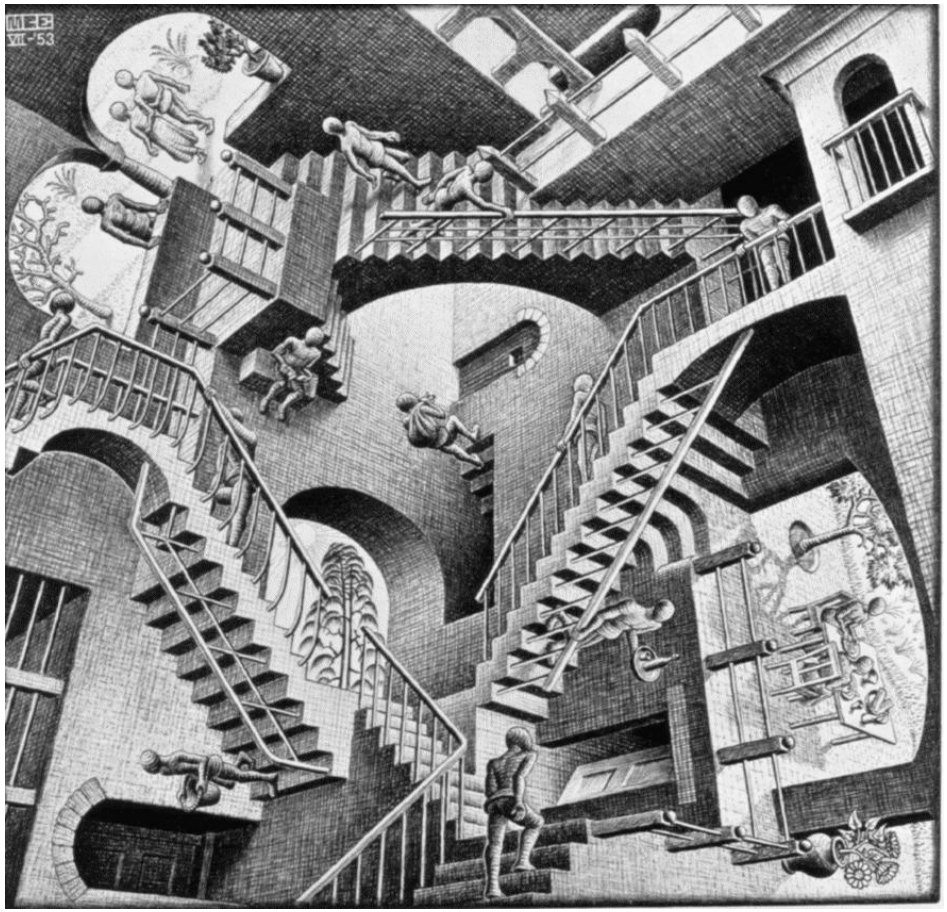
A serious study of Gödel

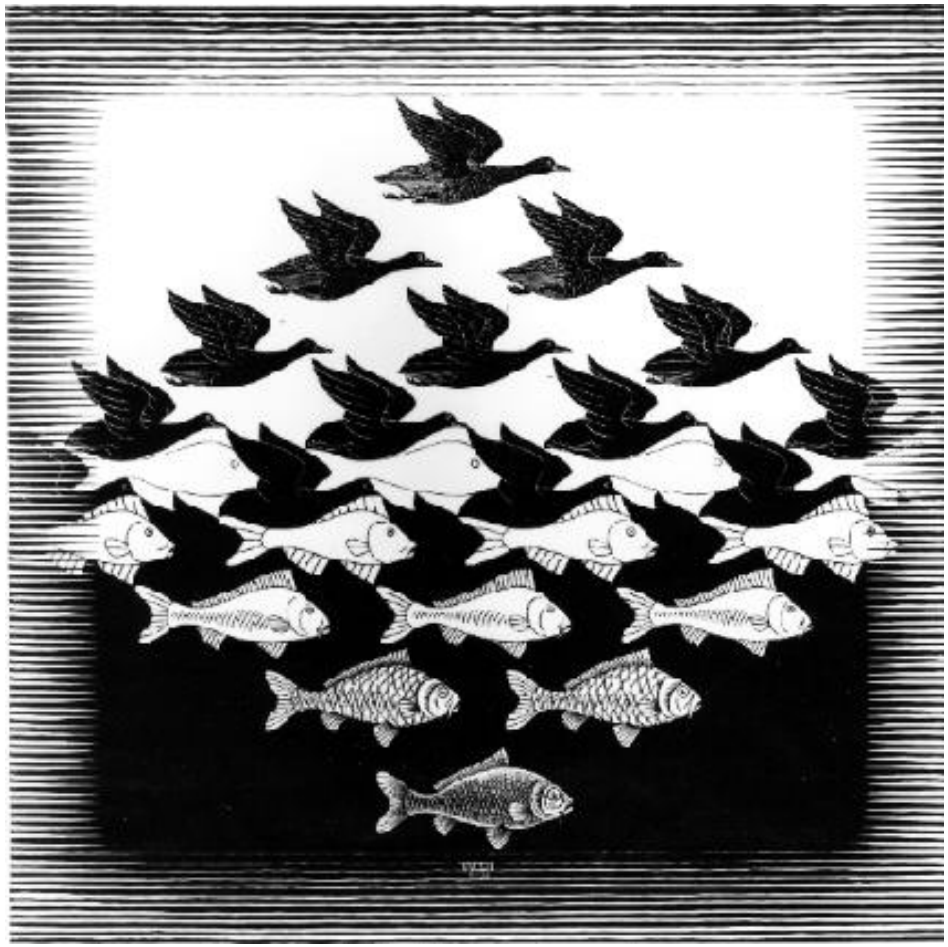


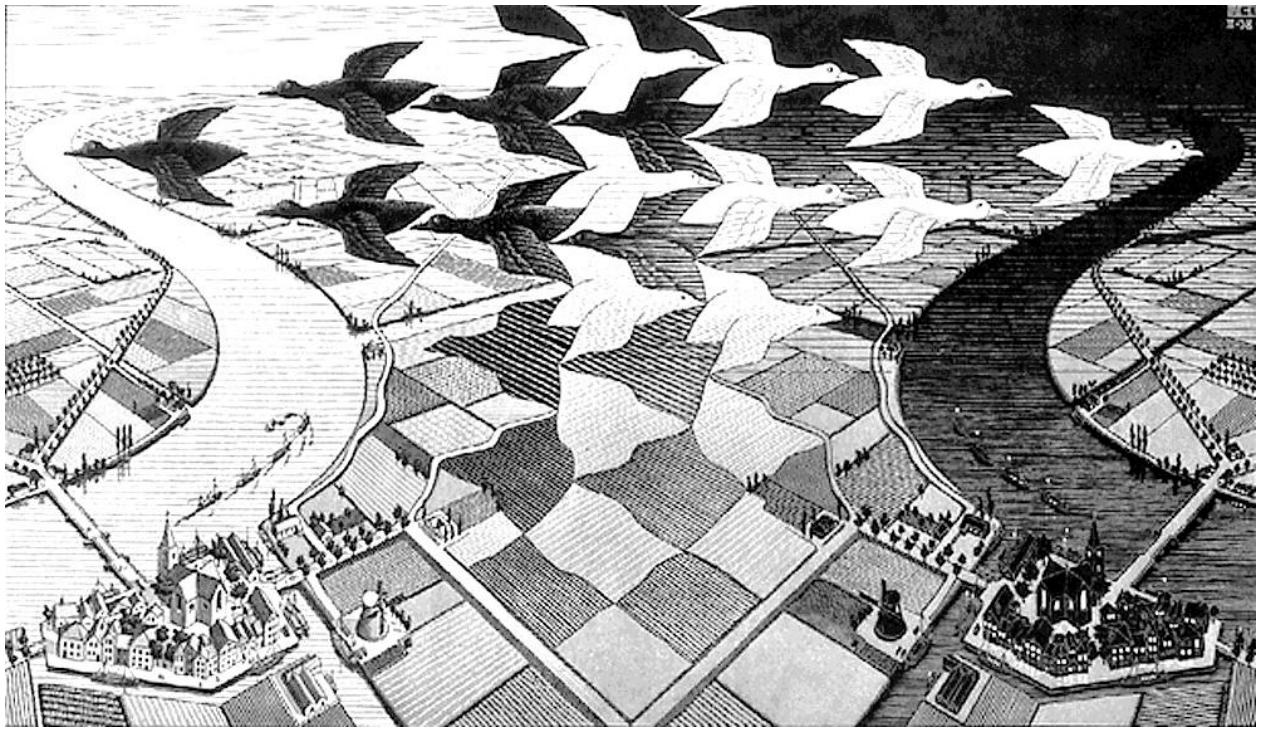
Maurits Escher


JOB




MCE
XII-'53





Ex.

$$\#S_1 = 3^1 = 3$$

$$\#\langle S_1 \rangle = 2^{\#S_3} = 2^3 = 8$$

$$\#\langle Z_1, S_1, J_{111} \rangle = 2^{\#Z_1} \cdot 3^{\#S_1} \cdot 5^{\#J_{111}} = 2^{2^1} \cdot 3^{3^1} \cdot 5^{(5^1 7^1 11^1)} = 4 \cdot 27 \cdot 5^{385}$$

Distinct programs get distinct codes, and given a code we can extract the (unique) program encoded by it
(or decide that it is not a code for any program).

Ex. Given the number 10871635968 we decompose it (uniquely) as a product of primes:

$$10871635968 = 2^{27} \cdot 3^4 = 2^{3^3} \cdot 3^{2^2} = 2^{\#S_3} \cdot 3^{\#Z_2} = \#\langle S_3, Z_2 \rangle$$

We let $\mathrm{Prog}(z)$ be a predicate that is true iff z is the code of some program P. $\mathrm{Prog}(z)$ is a pr predicate.

We let

$$\{z\} = \begin{cases} \text{program } P \text{ such that } z = \#P & \text{if } P \text{ exists} \\ \text{the empty program } \langle\rangle & \text{otherwise} \end{cases}$$

The function $\mathrm{Nex}(u, z) = u'$ is defined as follows: u' is the state resulting from a single step of $\{z\}$ on state u. Nex is pr.

If $u_0, u_1, \ldots, u_t$ is the sequence of codes for the successive states in a computation, then we code the entire computation by the number $y = p_0^{u_0} p_1^{u_1} \cdots p_t^{u_t}$.

Kleene T predicate: for each $n \geq 1$ we define the $n+2$-ary relation T_n as follows: $T_n(z, \vec{x}, y)$ is true iff y codes the computation of $\{z\}$ on input $\vec{x}$.

Theorem: For each $n \geq 1$, T_n is pr.

Let $\{z\}_n$ be the n-ary fn computed by program $\{z\}$.

Kleene Normal Form Theorem: There is a pr fn U such that

$$\forall n \geq 1, \qquad \{z\}_n(\vec{x}) = U(\mu y T_n(z, \vec{x}, y))$$

($U(y)$ extracts the contents of the first register in the last state of computation y.) Thus, every computable fn is recursive.

Part III
CONCLUSION

Church-Turing thesis: the following models of computation are all equivalent:

- Rewriting systems
- Turing machines
- λ-calculus
- Recursive functions
- Register machines
- **ZFC-computable**

Even more evidence that we have captures the notion of compation: ZFC is the Zarmelo-Fraenkel set theory together with the Axiom of Choice. All of mathematics can be formalized in ZFC.

A language L is ZFC-computable if there exists a formula $\alpha(x)$ such that if $w \in L \Rightarrow \text{ZFC} \vdash \alpha(w)$ and if $w \notin L \Rightarrow \text{ZFC} \vdash \neg\alpha(w)$.