UCT

Program Size Complexity II

Klaus Sutner

Carnegie Mellon University

Spring 2024

1 Prefix Complexity

2 Incompleteness

3 Solovay's Theorem

- Kolmogorov-Chaitin algorithmic information theory provides a measure for the “complexity” of a bit string (or any other finite object). This is in contrast to language based models that only differentiate between infinite collections.

- Since the definition is closely connected to Halting, the complexity function $K(x)$ fails to be computable, but it provides an elegant theoretical tool and can be used in lower bound arguments.

- And it absolutely critical in the context of randomness; more later.

Recall that our model of computation used in Kolmogorov-Chaitin complexity is a universal, one-tape Turing machine over the tape alphabet $\Gamma = \{0, 1, b\}$, with binary input and output.

As we have seen, this causes a number of problems because it is difficult to decode an input string of the form

$$p = q\,z$$

into an instruction part q and a data part z, with the intended semantics: run program q on input z.

Of course, this kind of problem would not surface if we used real programs instead of just binary strings: it is clear where a real program ends.

We should try to eliminate this issue in our setting, too.

M. Li, P. Vitányi

An Introduction to Kolmogorov Complexity and its Applications

Springer, 1993 (3ed 2009)

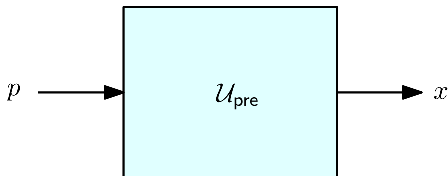
Encyclopedic treatment, some 850 references.

In language theory, a language $L \subseteq \Sigma^\star$ is said to be prefix if no string in L is a prefix of another.

Yes, this should be called prefix-free (some authors prefer this saner version), but the traditional form is prefix. Grin and bear it.

Always remember, obscure terminology keeps the uninitiated at bay.
Works great in job interviews.

The key idea is to restrict our universal machine a little bit.



We require that $P \subseteq \mathbf{2}^\star$, the collection of all syntactically correct programs for $\mathcal{U}_{\mathrm{pre}}$, is a prefix set: no valid program is a proper prefix of another.

Note that this condition trivially holds for most ordinary programming languages (at least in spirit; in the real world no one combines code and data in a single file).

Call a Turing machine $\mathcal{M}$ prefix if its halting set $\{\, p \in \mathbf{2}^\star \mid \mathcal{M}(p) \downarrow \}$ is prefix.

In order to compute $\mathcal{U}(p)$, machine $\mathcal{U}$ tries to parse the string p from left to right. Think about the machine tracing a branch in the tree $\mathbf{2}^\star$. If p turns out to be syntactically correct, it is executed. If not, $\mathcal{U}$ diverges.

This convention is perfect for the code/data model: since no program is an extension of another, there cannot be two different instruction parts q and q' such that $q\,z = q'\,z'$.

Simulations are also particularly simple for prefix machines: to simulate $\mathcal{M}$ on $\mathcal{M}'$ we can set up a header h such that

$$\mathcal{M}'(hp) \simeq \mathcal{M}(p)$$

for all $\mathcal{M}$-admissible programs p: $\mathcal{M}'$ can uniquely parse h, and then run $\mathcal{M}$ on the remaining string p.

This means that the difference in program-size complexity between various universal machines is rather small (unless the machines are of the cheating type that artificially encode a few strings with astronomical complexity by just a few bits).

We have to make sure that prefix machines exist and can do the same as ordinary ones.

Lemma

For any Turing machine $\mathcal{M}$, we can effectively construct a prefix Turing machine $\mathcal{M}'$ such that

- $\forall\, p \in \mathbf{2}^\star \left(\mathcal{M}'(p)\downarrow \;\Rightarrow \mathcal{M}(p) \simeq \mathcal{M}'(p)\right)$
- $\mathcal{M}$ *prefix* $\Rightarrow \forall\, p \in \mathbf{2}^\star \left(\mathcal{M}(p) \simeq \mathcal{M}'(p)\right)$

Of course, in general $\mathcal{M}'$ will halt on fewer inputs and the two machines are by no means equivalent (just think what happens to a machine with halting set contained in $0^\star$).

As a consequence, we can effectively enumerate all prefix functions $\{e\}_{\mathsf{pre}}$ just as we can effectively enumerate ordinary computable functions.

Suppose we have an ordinary machine $\mathcal{M}$ and some input $p \in \mathbf{2}^\star$. Recall that the halting set of $\mathcal{M}$ is semidecidable, and thus recursively enumerable. $\mathcal{M}'$ computes on p as follows:

- Enumerate the halting set of $\mathcal{M}$: $q_0, q_1, q_2, \ldots \in \mathbf{2}^\star$.
- If $q_i = p$, return $\mathcal{M}(p)$.
- If q_i is a proper prefix of p, or conversely, diverge.

Machine $\mathcal{M}$ also diverges if p is unrelated to any of the q_i. Essentially, we use the order of the enumeration to resolve prefix conflicts.

It is easy to check that $\mathcal{M}'$ is prefix and will define the same function as $\mathcal{M}$, provided $\mathcal{M}$ itself is already prefix.

□

Since the support in our construction typically shrinks (potentially by a lot), we have to make sure that there is a universal prefix machine. Programming languages suggest that should work, but we need make sure we can handle this for Turing machines.

No problem, really, we can control the syntax of correct input strings. Suppose $\mathcal{U}$ is a universal machine with two separate program/data tapes. Then define $\mathcal{U}'$ so that it checks for inputs of the form

$$p = u_1 0 u_2 0 \ldots 0 u_r 1\, x$$

Thus, if the input has the form $(\mathbf{2}0)^\star \mathbf{2}1\mathbf{2}^\star$, then $\mathcal{U}'$ runs $\mathcal{U}(u_1 \ldots u_r, x)$.

Otherwise it simply diverges.

Definition

Let $\mathcal{U}_{\rm pre}$ be a universal prefix Turing machine. Define the prefix Kolmogorov-Chaitin complexity of a string x by

$$C(x) = \min\big(\, |p| \mid \mathcal{U}_{\rm pre}(p) \simeq x \,\big)$$

Note that in general $C(x) > K(x)$: there are fewer programs available, so in general the shortest program for a fixed string will be longer than in the unconstrained case. Counting arguments become easier in this context.

Of course, $C(x)$ is again not computable, for essentially the same reasons.

The following mutual bounds are due to Robert Solovay:

$$C(x) \leq K(x) + K(K(x)) + O(K(K(K(x))))$$

$$K(x) \leq C(x) - C(C(x)) - O(C(C(C(x))))$$

This pins down the cost of dealing with prefix programs as opposed to arbitrary ones. So the difference between $K(x)$ and $C(x)$ is not too large, we expect the non-leading terms on the right-hand side to be fairly small.

Still, $C(x)$ is much better behaved than $K(x)$ in many ways.

Recall that for ordinary Kolmogorov-Chaitin complexity it is easy to get an upper bound for $K(x)$: the informal program

print "$x_1 x_2 \dots x_n$"

certainly does the job: this programming language is prefix. Alas, we need to worry about prefix TMs.

We could use delimiters around x as in the informal code snippet above, but remember that our input and output alphabet is fixed to be $\mathbf{2} = \{0, 1\}$.

We could add symbols to our base alphabet, but that does not solve the problem.

In the absence of delimiters, we can return to our old idea of self-delimiting programs. Informally, we could write

print next n **bits** $x_1 x_2 \dots x_n$

In pseudo-code this is fine, the only place where the prefix property could be violated is in the x part, but n fixes this problem. We obtain a complexity of at most $n + \log n + c$.

Again, we need to deal with prefix Turing machines, and that leads straight to cumbersome technical coding details.

To obtain a reliable bound on $C(x)$, there is no way around actually spelling out the coding details, at least in some detail.

We already know one way to satisfy the prefix condition: code $x \in \mathbf{2}^\star$ as

$$E(x_1 \ldots x_n) = x_1 0\, x_2 0\, \ldots\, x_{n-1} 0\, x_n 1$$

so that $|E(x)| = 2|x|$.

Again, there are other obvious solutions such as $0^{|x|}1x$.

Both approaches double the length of the string, which doubling would lead to a rather crude upper bound $2n + O(1)$ for the prefix complexity of a string via the program

print $E(x)$

We really would like $n + O(1)$. Can we get there? Or at least closer?

We have used the notation $|x|$ for a string x to denote its length. So

$$|.| : \mathbf{2}^\star \longrightarrow \mathbb{N}$$

In the following, it will be convenient to have another function

$$\begin{array}{rccc} \text{blen} : & \mathbf{2}^\star & \longrightarrow & \mathbf{2}^\star \\ & x & \longmapsto & \text{bin}(|x|) \end{array}$$

where the length is expressed as a number written in binary (say, MSD first, no leading zeros). We can think of blen as a kind of discrete logarithm.

How about leaving $x = x_1 x_2 \ldots x_n$ unchanged, but using E to code $n = |x|$, the length of x:

$$E(\mathsf{blen}\,x)\,x$$

Note that this still is a prefix code and we now only use some $2\log n + n$ bits to code x. But why stop here? We can also use

$$E(\mathsf{blen}^2\,x)\ \mathsf{blen}\,x\ x$$

This requires only some $2\log^2 n + \log n + n$ bits. And so on and so forth ...

$$E(\mathsf{blen}^3\,x)\ \mathsf{blen}^2\,x\ \mathsf{blen}\,x\ x$$

In fact, we can iterate this step of pushing the costly E encoding down to shorter and shorter strings. Let

$$E_0(x) := E(x)$$

$$E_{i+1}(x) := E_i(\mathsf{blen}\,x)\,x$$

Since the prefix $E_i(\mathsf{blen}\,x)$ is uniquely decodable, we can unravel the remainder of the string from there.

Exercise

Show that all the E_i are prefix codes.

Here is an example, using the string "abc . . . xyz" to represent any bit sequence of length 26.

0	a0b0c0d0e0f0g0h0i0j0k0l0m0n0o0p0q0r0s0t0u0v0w0x0y0z1	52
1	1010001001 | abcdefghijklmnopqrstuvwxyz	36
2	100011 | 11010abcdefghijklmnopqrstuvwxyz	37
3	1011 | 10111010abcdefghijklmnopqrstuvwxyz	38
4	1001 | 1110111010abcdefghijklmnopqrstuvwxyz	40
5	1001 | 101110111010abcdefghijklmnopqrstuvwxyz	42

The vertical bar is just for legibility, it ain't there in the actual code.

Note that E_1 is optimal in this particular case.

So there is a little problem: how do we choose the encoding level k as a function of the length of x?

There are at least two natural ways to do this:

- Iterate discrete logarithms to get below a fixed threshold (like length 2).
- Let k be minimal such that $|E_k(x)| \leq |E_\ell(x)|$ for all ℓ.

The second method is arguably more elegant, since it may produce better results. Alas, it leads to a few unpleasant surprises (non-monotonicity). Also note that there may well be better alternatives, these two just seem to be the most obvious choices (at least to me).

We'll stick with the first method. To this end, define $\mathsf{blen}^\star x$, a discrete version of an iterated logarithm $\log^*$. To avoid pesky edge cases, only consider $|x| \geq 2$.

$$\mathsf{blen}^\star : \mathbf{2}^\star \to \mathbb{N} \qquad x \mapsto \min\big(k \mid \mathsf{blen}^k x \in \{10, 11\} \big)$$

For example, for $x = \mathsf{bin}(10^{100})$ we get the following sequence of values, in binary and in decimal:

binary	x	101001101	1001	100	11	10	10	...
decimal	10^{100}	333	9	4	3	2	2	...

So $\mathsf{blen}^\star x = 4$.

Note that the numerical value of x does not really matter, it's just the number of bits the determine $\mathsf{blen}^\star x$.

We can use this iterated logarithm to determine the encoding level, for any binary string x:

$$E_\infty(x) = E(k)\, E_k(x) \qquad k = \mathsf{blen}^\star(x)$$

As written, this is a bit clumsy, we are using our basic prefix code E in two places; figure out how to get rid of this feature.

Example

Suppose we have a bit-string x of length 125000. Iterating , we get the following sequence of binary strings (lengths in decimal).

0	x	125000
1	11000011010100000	17
2	10001	5
3	101	3
4	11	2

So our method as stated would use $k = 4$ and thus add

$$17 + 5 + 3 + 2 \times 4 = 33$$

bits to the length of string, less that the typical 64-bit word length. A non-issue compared to the 125000 bits in x.

How much do we have to pay for a prefix version of x? Essentially a sum of iterated logs (we are shamelessly disregarding the necessary floors and/or ceilings).

Lemma

$$|E_\infty(x)| = n + \log n + \log^2 n + \log^3 n \ldots + \log^*(n) + O(1)$$

So this is an upper bound on $C(x)$.

Of course, some other coding scheme might produce even better results.

In the real world, a good rough approximation to $C(x)$ is $n + \log n$, in perfect keeping with our intuition about

> **print next** n **bits** $x_1 x_2 \ldots x_n$

It's clear that prefix complexity is a bit harder to deal with than ordinary Kolmogorov-Chaitin complexity. What are the payoffs?

For one thing, it is much easier to combine programs. This is useful, say, for concatenation: we want to generate xy. Suppose we have prefix programs p and q that produce x and y, respectively.

But then pq is uniquely parsable, and we can easily find a header program h such that

$$h\,p\,q$$

is an admissible program for $\mathcal{U}_{\mathrm{pre}}$ that executes p and q to obtain xy.

Thus

$$C(xy) \leq C(x) + C(y) + O(1)$$

Define $C(x, y)$ to be the length of the shortest program that writes $x\,b\,y$ on the tape (recall that our tape alphabet is $\{0, 1, b\}$). So this is essentially the pair (x, y), expressed as a string.

Note that $C(xy) \leq C(x, y) + O(1)$, but getting a bound in the opposite direction is tricky (think about $x, y \in 0^\star$).

At any rate, the last argument shows that $C(.)$ is subadditive:

$$C(x, y) \leq C(x) + C(y) + O(1)$$

This property simply fails for ordinary non-prefix complexity.

A higher level complaint is that plain non-prefix KC complexity does not help much when applied to the problem of infinite random sequences. To be sure, many arguments still work out fine, but there is a sense that the theory could be improved.

Sure enough, here is the killer app for prefix complexity.

Definition

The total halting probability of any prefix program is defined to be

$$\Omega = \sum_{\mathcal{U}_{\mathrm{pre}}(p)\downarrow} 2^{-|p|}$$

Ignoring the motivation behind this for a moment, note that this definition works because of the following bound.

Lemma (Kraft Inequality)

Let $S \subseteq \mathbf{2}^\star$ be a prefix set. Then $\sum_{x \in S} 2^{-|x|} \leq 1$.

We can define the halting probability for a single target string x to be

$$P(x) = \sum_{\mathcal{U}_{\mathrm{pre}}(p) \simeq x} 2^{-|p|}$$

and extend this to sets of strings via additivity: $P(S) = \sum_{x \in S} P(x)$.

Then $\Omega = P(\mathbf{2}^\star)$.

Note that Ω depends quite heavily on the choice of $\mathcal{U}_{\mathrm{pre}}$, so one could write $\Omega(\mathcal{U}_{\mathrm{pre}})$ or some such for emphasis.

Proposition

Ω is a real number and $0 < \Omega < 1$.

In fact, for one particular $\mathcal{U}_{\text{pre}}$, one can show with quite some pain that

$$0.00106502 < \Omega(\mathcal{U}_{\text{pre}}) < 0.217643$$

To establish this result one needs to get down into the weeds and, following the details of the definition of $\mathcal{U}_{\text{pre}}$, produce a

Lower Bound: show that some specific, short programs really converge.

Upper Bound: show that some specific, short programs really diverge.

So, this is unpleasantly close to Halting and rather messy in actuality–recall the Busy Beaver problem. For slightly longer programs, this type of analysis becomes quickly unmanageable.

Proposition

Ω is incompressible in the sense that $K(\Omega[n]) \geq n - c$, for all n.

Here $\Omega([n])$ denotes the first n bits of Ω. As a consequence, Ω is Martin-Löf random (nowadays the standard definition of randomness for infinite sequences).

This may seem a bit odd since we have a perfectly good definition of Ω in terms of a converging infinite series. But remember, the Halting Problem is lurking in the summation – from a strictly constructivist point of view Ω is in fact quite poorly defined.

Random strings are useful for certain algorithms, but one would think intuitively that they are quite useless as a direct source of information (compared to Wiles' proof of FLT, say).

Lemma

Consider $q \in \mathbf{2}^{\leq n}$. Given $\Omega[n]$, it is decidable whether $\mathcal{U}_{\rm pre}$ halts on input q.

Proof.

Start with a lower bound $\Omega_0 = 0$.

Dovetail computations of $\mathcal{U}_{\rm pre}$ on all inputs using the standard approach.

Whenever convergence occurs on some program p, update the approximation: $\Omega_0 := \Omega_0 + 2^{-|p|}$.

Stop as soon as $\Omega_0 \geq \Omega[n]$. Then we have the following lower and upper bounds:

$$\Omega[n] \leq \Omega_0 < \Omega < \Omega[n] + 2^{-n}.$$

But then no program of length at most n can converge at any later stage; we just have to check whenever q has already terminated. □

For $n \approx 10000$, knowledge of $\Omega[n]$ would settle, at least in principle, several major open problems in mathematics such as the Goldbach Conjecture or even the Riemann Hypothesis.

As mentioned some time ago, the Riemann Hypothesis can be checked by a Turing machine that fails to halt iff the RH is true.

Scott Aaronson has shown that the machine would need at most 744 states, and possibly far fewer. So a corresponding prefix program could probably be written in 10000 bits (just eyeballing things, I have not checked anything).

Goldbach could be handled this way for sure, supposedly there is a 27-state Turing machine that halts iff the conjecture is false.

Of course, we don't have the first 10000 bits of Ω, nor will we ever.

In fact, things are much, much worse than that.

Suppose some dæmon gave you these bits. It would take a long time to exploit this information: the running time of the oracle algorithm above is not bounded by any recursive function of n.

All the answers would be staring at us, but we could not pull them out.

1 Prefix Complexity

2 Incompleteness

3 Solovay's Theorem

David Hilbert wanted to crown 2000+ years of development in math by constructing an axiomatic system that is

- consistent
- complete
- decidable

Alas . . .

Theorem (Gödel 1931)

Every consistent reasonable theory of mathematics is incomplete.

Theorem (Turing 1936)

Every consistent reasonable theory of mathematics is undecidable.

Reasonable here just means: at least as strong as basic arithmetic, and the axioms are decidable.

This is good news for anyone interested in foundations, who would want to live in a boring world?

Gödel's argument is a very careful elaboration and formalization of the old Liar's Paradox:

> This here sentence is false.

Turing uses classical Cantor-style diagonalization applied to computable reals.

Both arguments are perfectly correct, but they seem a bit ephemeral; they don't quite have the devastating bite one might expect.

For example, Gödel's version of the Liar is an arithmetic statement φ that says: this sentence is not provable in the given system.

By consistency, φ must indeed fail to be provable in the chosen system, hence the sentence is true—so our theory is incomplete.

The problem is that the sentence in question is logical in nature, rather than just pure arithmetic. A lot of work has since gone into finding true but unprovable statements that are more mathematical in nature. Still, they are not totally compelling. Look at Harvey Friedman's work for several examples.

Ω can help to make the limitations of the formalist/axiomatic approach much more concrete. First a warm-up.

Émile Borel defined a normal number in base B to be a real r with the property that all digits in the base B expansion of r appear with limiting frequency $1/B$.

Theorem (Borel)

With probability 1, a randomly chosen real is normal in any base.

Alright, but how about concrete examples? It seems that $\sqrt{2}$, π and e are normal (billions of digits have been computed), but no one currently has a proof.

$$C = 0.12345678910111213141516171819202122\ldots$$

Champernowne showed that this number is normal in base 10 (and powers thereof), the proof is not difficult. The number is transcendental; it is open whether it is normal in any other base.

Proposition

Ω is normal in any base.

Of course, there is a trade-off: we don't know much about the individual digits of Ω. In fact, we basically know nothing, see the next section.

Time to get serious. Fix some cutoff n. Suppose we want to prove all correct "theorems" of the form

$$C(x) = m < n \qquad \text{or} \qquad C(x) \geq n$$

In other words, we want to prove a lower bound N for some concrete string x, or pin down the complexity exactly.

How much logic strength would we need to do this? Clearly, we have to contend with Halting, but we can use a powerful theory, say, Zermelo-Fraenkel with Choice. It should not be too hard to reason about Halting in that sort of environment, right?

To establish assertions like $C(x) = m < n$ or $C(x) \geq n$ in the obvious way, we need the maximal halting time τ of all programs of length at most $n - 1$. Then we can simply run all the relevant programs and see what happens.

It is not hard to see that

$$C(\tau) = n + O(1)$$

Essentially, nothing less will do.

Alas, this is the obvious, computational approach, we want to reason about theorem proving. Who knows, maybe there is some clever argument that does not require τ?

In the following we assume that $\mathcal{T}$ is some axiomatic theory of mathematics that includes arithmetic (first-order logic is fine).

Think of $\mathcal{T}$ as Dedekind-Peano Arithmetic, though stronger systems such as Zermelo-Fraenkel with Choice is perfectly fine, too (some technical details get a bit more complicated; we have to interpret arithmetic within the stronger theory).

Since we have arithmetic, we can certainly formalize assertions like $C(x) = m$ and $C(x) \geq n$ in $\mathcal{T}$. We need to figure out how easily these "theorems" might be provable in $\mathcal{T}$.

We need $\mathcal{T}$ to be consistent: it must not prove wrong assertions. This is strictly analogous to the situation in Gödel's theorem: inconsistent theories have no trouble proving any assertion whatsoever.

Actually, technically all we need is Σ_1 consistency: any theorem of the following simple form, provable in $\mathcal{T}$, must be true:

$$\exists\, x\, \varphi(x)$$

where φ is primitive recursive (defines a primitive recursive property in $\mathcal{T}$) and the existential quantifier is arithmetic.

We assume that suitable rules of inference for first-order logic are fixed, once and for all. So the theory $\mathcal{T}$ comes down to its set of axioms.

If there only finitely many axioms, we can think of their conjunction as a single string and define $C(\mathcal{T})$ accordingly.

If there are infinitely many axioms (as in DPA, think about induction), the set of all axioms is still decidable and we can define $C(\mathcal{T})$ as the complexity of the corresponding decision algorithm.

Note that this approach totally clobbers anything resembling semantics: it does not matter how clever and/or elegant the axioms are, just how large a program is needed to specify them.

Theorem (Chaitin 1974/75)

If $\mathcal{T}$ proves the assertion $C(x) \geq n$, then $n \leq C(\mathcal{T}) + O(1)$.

Proof.

Enumerate all theorems of $\mathcal{T}$, looking for statements of the form $C(x) \geq n$.

For any $k \geq 0$, let x_k be the string in the first theorem so discovered where $n > C(\mathcal{T}) + k$, if it exists. We have to make sure there are only finitely many such k.

By consistency, we have

$$C(\mathcal{T}) + k < C(x_k)$$

By construction and subadditivity we get

$$\begin{aligned} C(x_k) &\leq C(\mathcal{T}, C(\mathcal{T}), k) + O(1) \\ &\leq C(\mathcal{T}) + C(k) + O(1) \end{aligned}$$

But then it follows immediately that

$$k < C(k) + O(1) \leq \log k + O(1)$$

and thus $k \leq k_0$ for some fixed k_0. □

Similarly one can prove that no consistent theory can determine more than

$$C(\mathcal{T}) + O(1)$$

bits of Ω.

We have a perfectly well-defined real, but, in the context of any formal theory mathematics, we can only figure out a the first few of its digits.

Here is an application of Ω in number theory. Recall

Theorem (Y. Matiyasevic, 1970)

It is undecidable whether a Diophantine equation has a solution in the integers.

One important step towards the proof was to show that any semidecidable set can be expressed in terms of a exponential Diophantine equation:

$$a \in A \iff \exists\, x_1, \ldots, x_n \; E(a, x_1, x_2, \ldots, x_n) = 0$$

Of course, exponential Diophantine equations are scary in general, even over $\mathbb{N}$:

$$(x+1)^{n+3} + (y+1)^{n+3} = (z+1)^{n+3}$$

Incidentally, if $E(a, x_1, x_2, \ldots, x_n) = 0$ has α many solutions, then there is a program of size

$$C(\alpha) + O(1)$$

to find them: brute-force search until they are all discovered.

But how about infinitely many solutions?

Theorem (Chaitin)

There is an exponential Diophantine equation $E(k, \boldsymbol{x}) = 0$ such that: there are infinitely many solutions $\boldsymbol{x}$ iff the kth bit of Ω is 1.

Loosely speaking: randomness and chaos lurks even within integer polynomials–not a place where one would usually be looking for these scary things.

1 Prefix Complexity

2 Incompleteness

3 Solovay's Theorem

One can sharpen Chaitin's theorem to a point where it almost seems absurd:

Theorem

Let $\mathcal{T}$ be as before. Then there is a universal prefix machine $\mathcal{U}_{\mathrm{pre}}$ such that

- *Peano Arithmetic proves that $\mathcal{U}_{\mathrm{pre}}$ is indeed universal.*
- *$\mathcal{T}$ cannot determine a single digit of $\Omega = \Omega(\mathcal{U}_{\mathrm{pre}})$.*

This result is rather counter-intuitive, one might think that the standard approach towards identifying a few digits of Ω should work just fine. The proof depends on a very clever construction of a particular universal prefix machine and uses Kleene's recursion theorem.

Start with any old universal prefix machine $\mathcal{V}$ whose universality can be proven in DPA (any standard machine will do).

Define $\mathcal{U}_{\mathrm{pre}}$ in three cases as follows:

- $\mathcal{U}_{\mathrm{pre}}(\varepsilon) \uparrow$
- $\mathcal{U}_{\mathrm{pre}}(0p) \simeq \mathcal{V}(p)$
- $\mathcal{U}_{\mathrm{pre}}(1p)$: batten down the hatches

Note that $\mathcal{U}_{\mathrm{pre}}$ is already guaranteed to be universal, provably in DPA. We will use the missing inputs $1p$ to destroy any semblance of predictability of Ω.

Let $|p| = n$. We will not use p as an actual program (unlike in the $0p$ case) but only as a yardstick to determine n, and later to determine a rational number.

Enumerate the theorems of $\mathcal{T}$ of the form

$$\Theta(n, b) = \text{the } n\text{th bit of } \Omega \text{ is } b.$$

Here $b \in \mathbf{2}$ is a single bit.

Wait, we are in the middle of the definition of $\mathcal{U}_{\mathrm{pre}}$, but the formalization of $\Theta(n, b)$ requires knowledge of $\mathcal{U}_{\mathrm{pre}}$. This is when the Recursion Theorem strikes: in the definition of $\mathcal{U}_{\mathrm{pre}}$, we may safely assume that we have access to an index for $\mathcal{U}_{\mathrm{pre}}$.

We ignore all messy details.

Suppose we discover $\Theta(n, b)$. Now do the following:

For any program p of length n, let r_0 be the rational with dyadic expansion $pb \in \mathbf{2}^{n+1}$ and let $r_1 = r_0 + 2^{-(n+1)}$. Note that for the right p we must have $r_0 < \Omega < r_1$.

Now search for a stage $\sigma \geq 0$ where we already have

$$r_0 < \Omega_\sigma < r_1$$

for the standard, computable approximation Ω_σ. If the search succeeds, let $\mathcal{U}_{\mathrm{pre}}(1p)$ halt, adding a bit to Ω and promptly kicking us out of the interval.

But if $\mathcal{T}$ correctly determines the nth bit of Ω (i.e., proves a correct theorem $\Theta(n, b)$), then Ω will be in this interval, for the right program p of length n.

Contradiction.