UCT

Turing Machines

Klaus Sutner

Carnegie Mellon University

Spring 2023

1 * Turing Machines

2 * Turing Computability

3 Enumeration and Diagonalization

4 The Busy Beaver Problem

5 Semidecidability

Alan Turing (1912–1954)

The Source

A. Turing

On Computable Numbers, with an Application to the Entscheidungsproblem

Proc.London Math.Soc., 2–42 (1936-7), pp. 230–265.

Turing called his now eponymous devices a-machines, a for automatic. These do not halt: their purpose is to write the infinite binary expansion of a real number on (part of) the tape.

Our description of a TM is the modern one due to Davis and Kleene. The main justification for TMs as the standard model of computation is that they work very well for complexity theory.

Before Turing

There were two models of computation in existence before Turing's seminal 1936 paper, both developed by logicians, and based on elementary ideas in math: equations and functional composition.

- Herbrand-Gödel equations.

  Those describe recursive functions in the most general sense (recursion on multiple variables; by contrast primitive recursive functions allow recursion only on one variable).

- Church's λ-computable functions.

  The λ-calculus is the abstract theory of functional composition. Very elegant, very hard to use for any concrete purpose (there are no data structures).

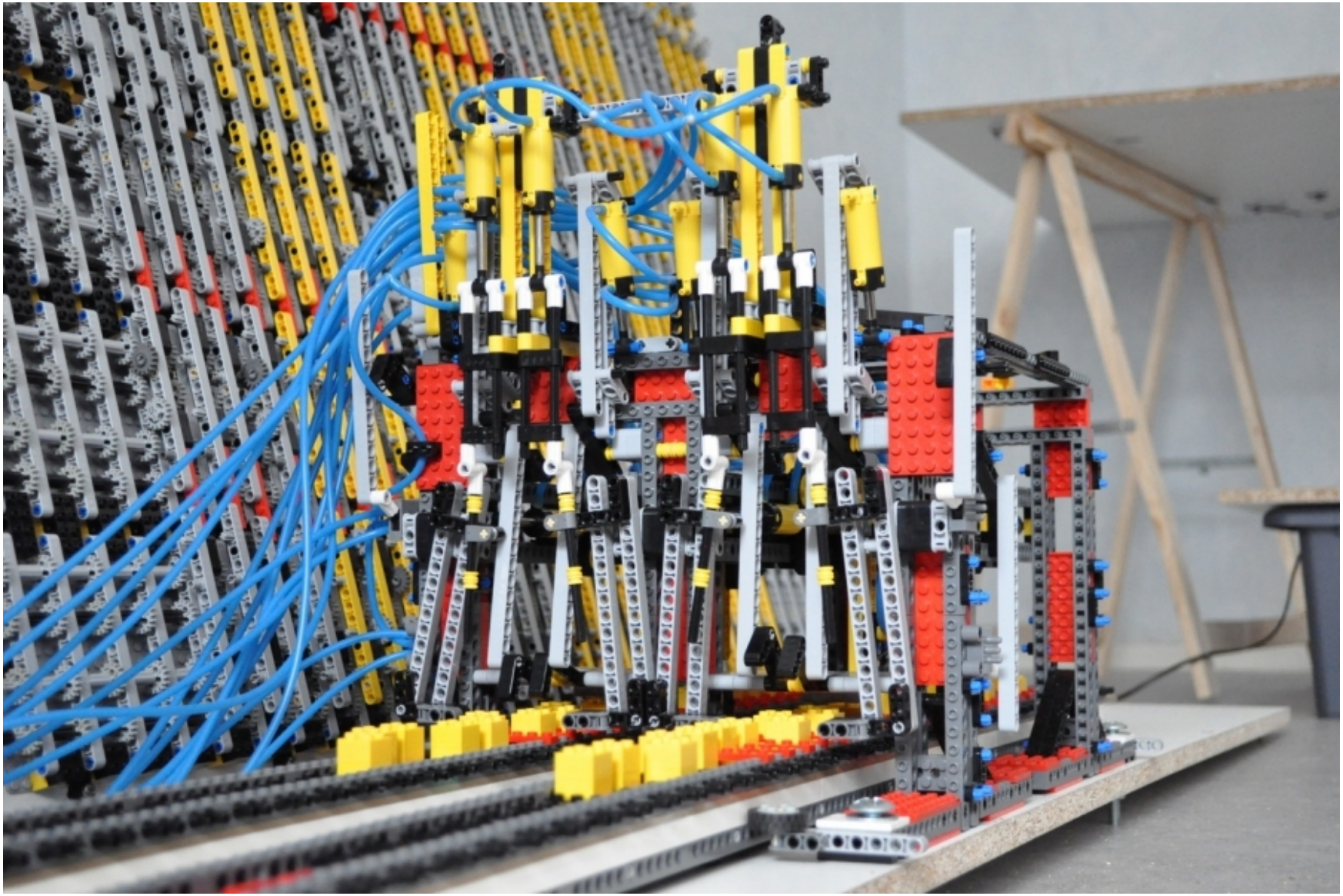
Turing's Machines

Brilliant Idea: Observe a human computor, then abstract away all the merely biological stuff and formalize what is left.

Everyone agrees that mathematicians compute (among other things such as drinking coffee or proving theorems). So we could try to define an abstract machine that can perform any calculation whatsoever that could be performed in principle by a mathematician, and only those.

Note the hedge "in principle": we will ignore "merely physical" constraints such as the computor dying of old age and decrepitude, or running out of scratch paper after using up the whole universe.

A Concrete Turing Machine



No Joke

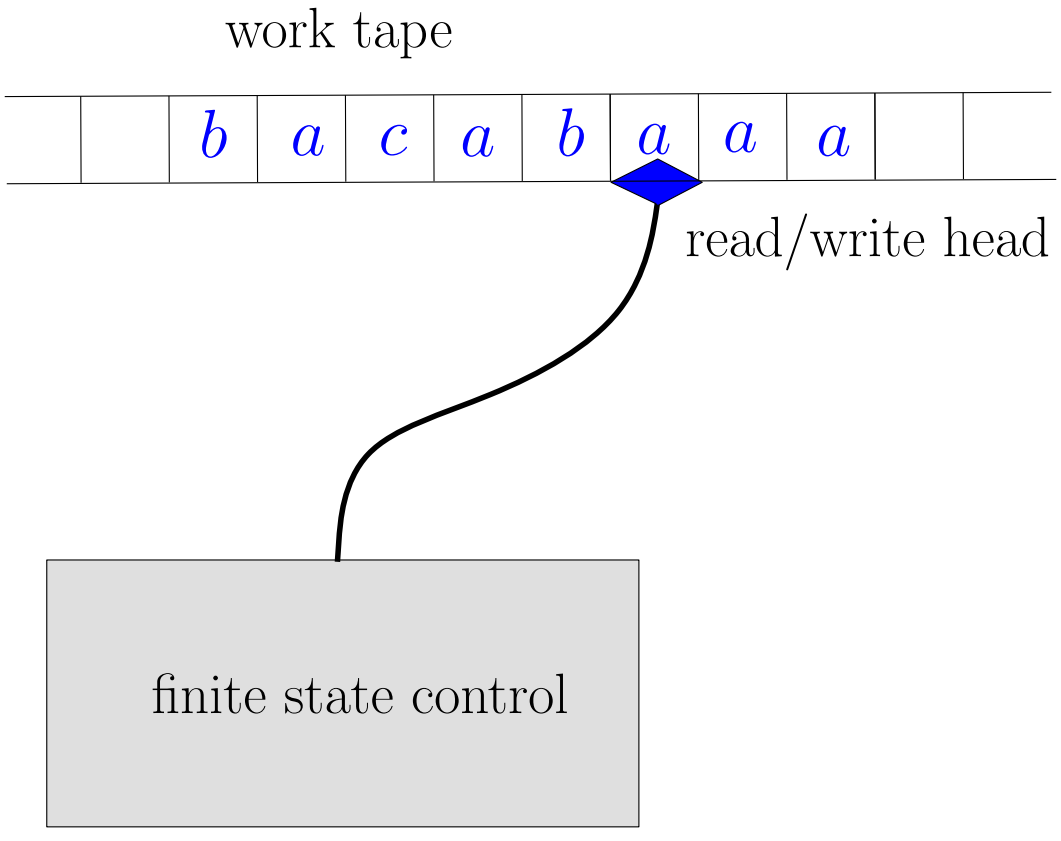
Obviously, whoever built the LEGO Turing machine has a lot of extra time on their hands.

But, the construction brings out a very important issue: Turing machines are clearly physically realizable. Our standard notion of computation is perfectly compatible with our physical universe.

This is not so clear for other mathematical tools: for example, do the reals[†] actually describe physical reality?

[†] If you think the answer is obviously Yes, note that a logician can easily come up with several versions of the reals. Which is the that one corresponds to actual physics?

An Abstract Turing Machine



The Pieces

- A tape: a bi-infinite strip of paper, subdivided into cells. Each cell contains a single letter; all but finitely many contain just a blank. So we have a tape inscription and we can represent it as a finite word over the tape alphabet (ignore the infinitely many blanks).
- A read/write head that is positioned at a particular cell. That head can move left and right.
- A finite state control that directs the head: symbols are read and written, the head is moved around and the internal state of the FSC changes.

Formalization

- alphabet Σ: finite set of symbols, special blank symbol $\sqcup$ in Σ
- state set Q: finite set of possible mind configurations
- $\delta : Q \times \Sigma \nrightarrow Q \times \Sigma \times \{-1, 0, +1\}$: partial transition function
- a special initial state $q_{\text{init}} \in Q$
- a special halting state $q_{\text{halt}} \in Q$.

Definition

A Turing machine[†] is a structure $\mathcal{M} = \langle Q, \Sigma, \delta, q_{\text{init}}, q_{\text{halt}} \rangle$.

[†] This is not Turing's original definition; he was interested in machines that do not halt and instead produce the infinite binary expansion of a real number. For us, halting is key.

Sanity Check

Let us agree on the following conventions:

- The tape alphabet contains at least two symbols. It is often convenient to let $\Sigma = \{0, 1\}$ and think of 0 as the blank symbol.
- $\delta(q_{\text{halt}}, a)$ is undefined for all $a \in \Sigma$.

  This will be used in a moment when we define what it means for a TM to halt.

Turing Machines are Brilliant

Of all the standard models of computation, Turing machines are most easily shown to capture precisely the intuitive notion of computability: arguably they correspond to the abilities of a human computor.

In addition, TMs are fairly simple, certainly much more palatable then Herbrand-Gödel equations or Church's λ-calculus, but not as nice as models that are closer to actual hardware such as register machines or random access machines, let alone programming languages.

And they work extremely well in the context of complexity theory, unlike some of the other models. For us, this reason alone settles the question of which model to use.

Turing's "Machines".

These machines are humans who calculate.

Turing Machines Suck

One substantial drawback of TMs is that it is hugely cumbersome to actually construct interesting examples. Say, a TM that computes multiplication of naturals given in binary. Or a universal machine that can be run on nice examples. Or try to prove that Turing machines can compute primitive recursive functions.

Similarly, even tiny TMs with single-digit number of states are often just about impossible to analyze (busy beaver problems).

Other models such as register machines or while-programs are better behaved in this regard. Fortunately one can usually get around the gory details and appeal to common sense: "clearly, one can construct a TM that does such-and-such . . . "

A Quibble

Arguably the most natural examples of computation come from arithmetic: operations like addition, multiplication, exponentiation, gcd, prime factorization and so on are clearly all computable.

These all involve natural numbers, they are referred to as arithmetic functions, maps $\mathbb{N}^k \to \mathbb{N}$. In fact, all of CRT was developed in terms of arithmetic functions. Computable arithmetic functions are historically called partial recursive functions. The ones that are in addition total are called (total) recursive functions.

But: Turing machines naturally operate on strings, not numbers. This is a blessing for complexity theory where we have to deal with data structures such as adjacency matrices as input, but for arithmetic it's a bit of a type mismatch.

There is a simple fix: we just code natural numbers as strings, typically either in unary or binary. No big deal, but certainly a misalignment.

On the other hand, sometimes it is very convenient not to have to worry about all the possible details of string encodings. So in this case, we think of strings simply as numbers: we simply interpret the letters of the alphabet as digits.

For example, we could think of some program P as a number $e \in \mathbb{N}$, called an index for P: think of the representation of P in memory as a number written in binary (maybe add a leading 1). Ditto for all other discrete data structures.

Lame Example: Successor Function

Tape alphabet $\Sigma = \{␣, |\}$

States $Q = \{0, 1, 2, 3\}$

Initial state $q_{\text{init}} = 0$, final state $q_{\text{halt}} = 3$.

The transition function δ is given be the following table:

p	σ	$\delta(p, \sigma)$		
0	␣	1	␣	$+1$
1	␣	2	|	-1
1	|	1	|	$+1$
2	␣	3	␣	0
2	|	2	|	-1

So the machine gets stuck if it is in state 0 but reads symbol 1. This won't matter during actual computations.

Why not Diagrams?

DFAs can be nicely represented by diagrams that have labeled edges of the form

$$p \xrightarrow{a} q$$

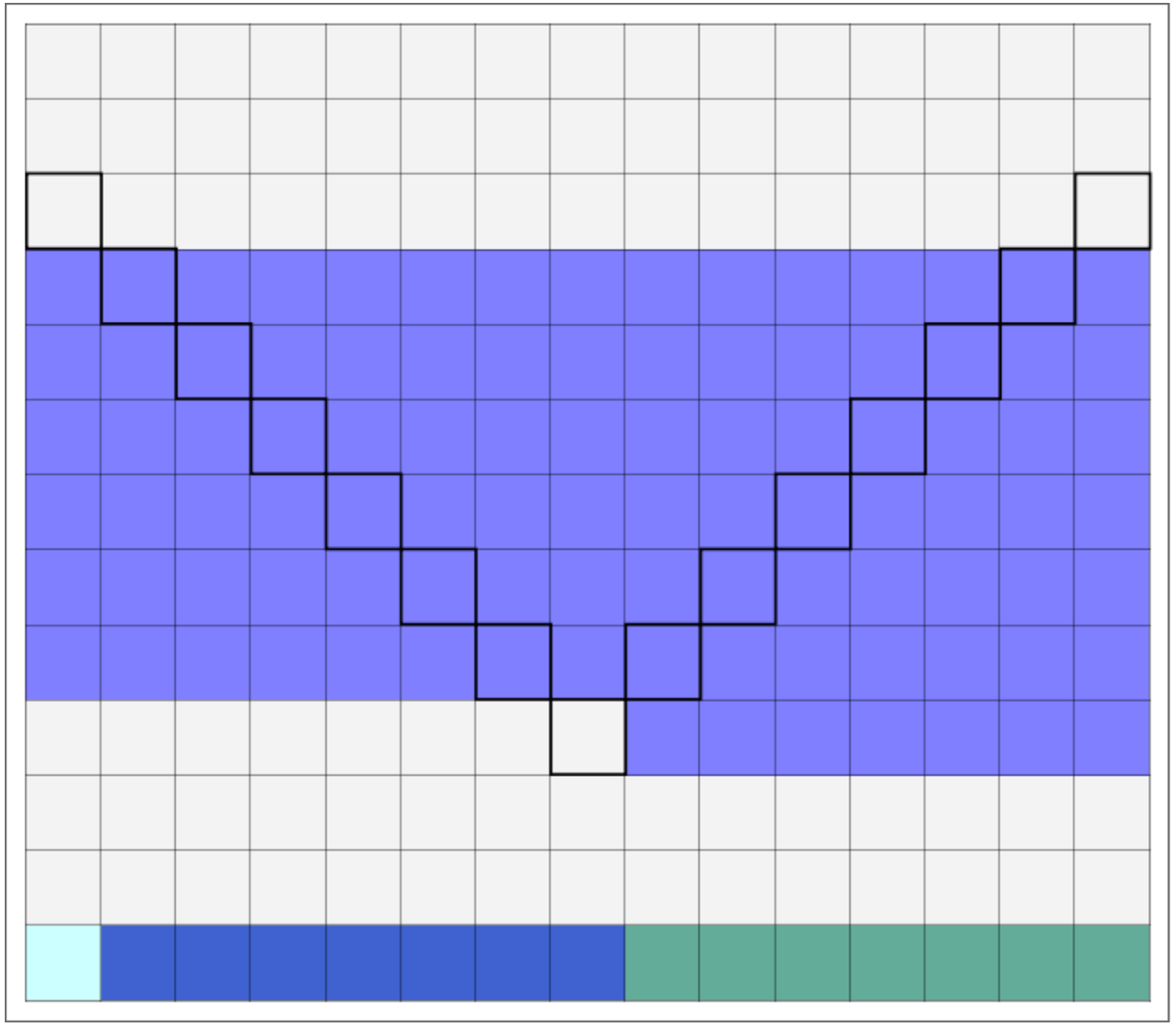
to indicate that $(p, a, q) \in \delta$.

While it is still possible to draw diagrams for Turing machines, the transitions now take the form

$$p \xrightarrow{a:b:d} q$$

to indicate that $\delta(p, a)=(q, b, d)$. This produces a lot of visual clutter and is much less useful: the tape content is missing.

Sample Run



Addition

We are using fat unary notation, n is represented by

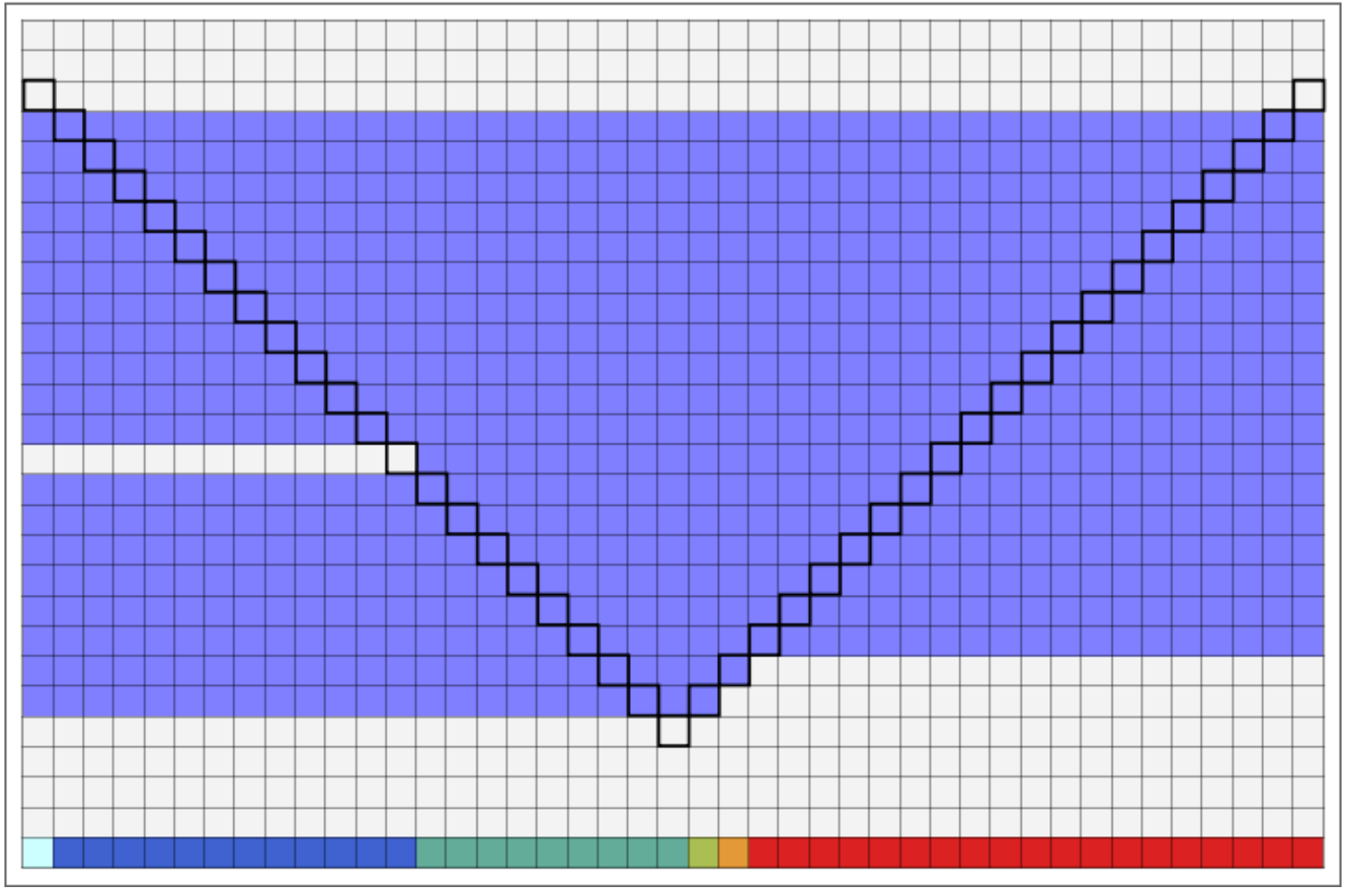
$$n \mapsto \underbrace{|||\ldots||}_{n+1}$$

Hence we have to erase two 1s at the end:

0	$\sqcup$	1	$\sqcup$	$+1$
1	$\sqcup$	2	$\vert$	$+1$
1	$\vert$	1	$\vert$	$+1$
2	$\sqcup$	3	$\sqcup$	-1
2	$\vert$	2	$\vert$	$+1$
3	$\vert$	4	$\sqcup$	-1
4	$\vert$	5	$\sqcup$	-1
5	$\sqcup$	6	$\sqcup$	0
5	$\vert$	5	$\vert$	-1

0 is the initial state, 6 is the final state

Sample Run



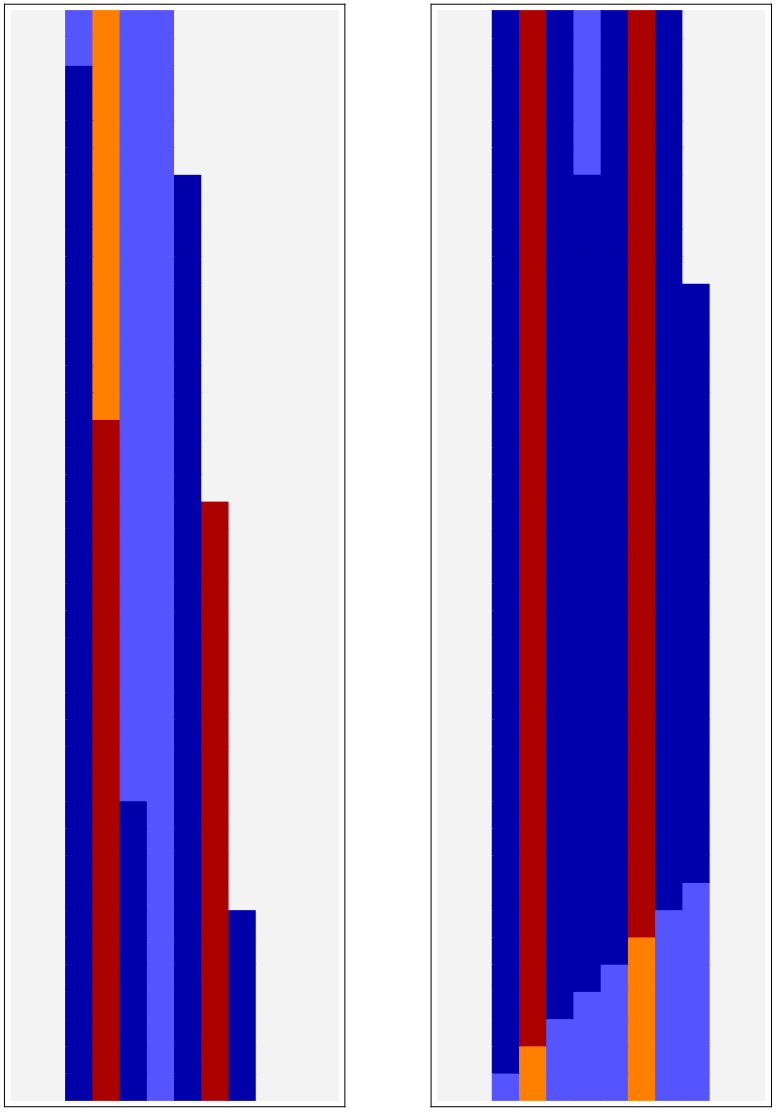
Copy

Here is a Turing machine[†] that copies its input: $x \mapsto xx$.

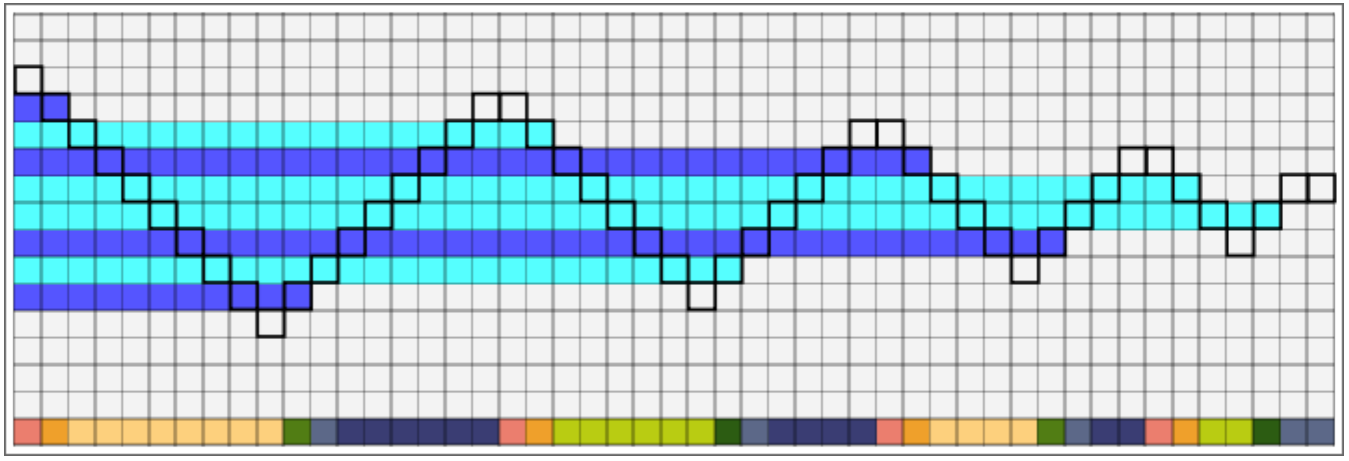
```
rules = {
    {0, 0} → {1, 0, 1},                          (* start *)
    {1, aa} → {1, aa, 1}, {1, bb} → {1, bb, 1},  (* look for unmarked letter *)
    {1, a} → {2, aa, 1}, {1, b} → {3, bb, 2},    (* change s to ss, remember *)
    {1, 0} → {6, 0, -1},
    {2, s_?Positive} → {2, s, 1},                (* seek right blank *)
    {2, 0} → {4, aa, 1},                         (* change to ss *)
    {3, s_?Positive} → {3, s, 1},                (* seek right blank *)
    {3, 0} → {4, bb, 1},                         (* change to ss *)
    {4, 0} → {5, 0, -1},
    {5, s_?Positive} → {5, s, -1},
    {5, 0} → {1, 0, 1},
    {6, s_?Positive} → {6, s, 1},                (* seek right blank *)
    {6, 0} → {7, 0, -1},
    {7, 3} → {7, 1, -1},                         (* seek left blank, unmark *)
    {7, 4} → {7, 2, -1}                          (* halt  *)
  } /. {a -> 1, b -> 2, aa -> 3, bb -> 4}
```

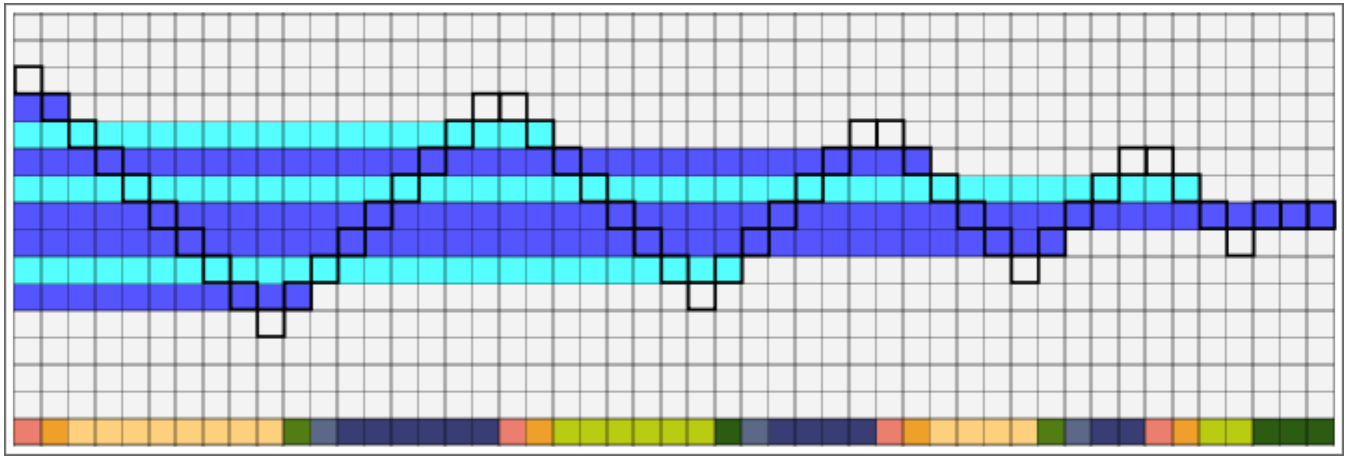
[†]Written in Mathematica, so it can actually be executed.

A Run



Palindromes





Exercise

What would a complete correctness proof for the Turing machine that performs unary addition look like? What is difficult about the proof?

Exercise

Construct a Turing machine that performs addition when the input is given in binary. What would the execution pictures look like in this case? How hard is a correctness proof?

Exercise

Figure out how the palindrome TM works and prove that it is correct (you need a convention for accepting versus rejecting computations).

Exercise (Hard)

Show that any one-tape Turing machine requires quadratic time to recognize palindromes.

This is Fraud

The pictures are stunningly beautiful, but we have not really explained what they are a picture of.

More precisely, we have a definition of a Turing machine, but we appeal to intuition when it comes to computations of these machines. As far as intuition is concerned, this is fine, but to prove anything we also need clear, formal definitions.

The Approach

Given any type of machine $\mathcal{M}$, to define computations of $\mathcal{M}$, it is a good idea to think about interrupting a computation.

What information is minimally needed to resume the computation later? This is called a configuration or instantaneous description (ID).

Then we explain how $\mathcal{M}$ moves from one configuration to the next, the one-step relation. Chaining together single steps by induction, we get a many-step relation that formalizes computations of arbitrary length.

We have to add input/output conventions, and then we're done.

Configurations here consist of:

- the current state
- the current tape inscription (non-blank part only)
- the current head position

So we have three pieces of information that we need to keep track of. They are all finite objects, so there is no problem in describing them by some explicit datatype. Something like a triple in $Q \times \Gamma^\star \times \mathbb{N}$. Make sure to figure out the details.

As it turns out, the following, less obvious, representation works very well in practice.

String Representation

We may safely assume that $Q \cap \Sigma = \emptyset$. We will use $Q \cup \Sigma$ as an alphabet and use strings of the form $\Sigma^+ \, Q \, \Sigma^+$ to encode configurations.

Definition

A configuration or instantaneous description (ID) is a word $y\,p\,x$ where $x, y \in \Sigma^+$ and $p \in Q$:

$$y_m y_{m-1} \ldots y_1 \, p \, x_1 x_2 \ldots x_n$$

means that the read/write head is positioned at x_1 and the tape inscription is $y_m \ldots y_1 x_1 \ldots x_n$.

One of the reasons this formalization is particularly useful is that one can use the theory of finite state machines to explain what it means to perform one step in a computation (a transducer can handle this task).

Details

The last description really captures one particular type of Turing machine: a

> TM with a single, two-way-infinite tape

with inscriptions of the form

$\ldots$	$\sqcup$	$\sqcup$	$\sqcup$	y_m	$\ldots$	y_1	$\mathbf{x_1}$	$\ldots$	x_n	$\sqcup$	$\sqcup$	$\sqcup$	$\ldots$

where the head is at x_1 and the machine is in state p.

Alas, this is just the tip of the iceberg: it is also useful to consider machines that have multiple tapes (each with a separate read/write head) and/or one-way-infinite tapes. In addition, later we will use separate read-only input tapes, and write-only output-tapes (these are always one-way-infinite).

More Details

We may safely assume that $n, m \geq 1$ since we can always let y_1 and x_1 be the blank symbol.

It also makes sense to choose n and m to be minimal such that $\boldsymbol{yx}$ captures all the non-blank symbols on the tape. We won't bother to make this part of the definition, though, it really does not matter much.

This is subtly different from a model where are tape inscription is a map $\mathbb{Z} \to \Gamma$ with finite support: our approach is coordinate free. Usually that is better, but sometimes it is preferable to keep track of the absolute position on the tape.

One Step

Next we need to explain a single step in a computation:

$$ypx \;\Big|\!\!\frac{1}{\mathcal{M}}\; y'qx'$$

Recall that we assume x and y to be non-empty (otherwise set $y_1 = x_1 = \text{␣}$). Now let $\delta(p, x_1) = (q, a, \Delta)$. Then the next configuration is defined by

$$\begin{array}{lll} y_m \ldots y_1\, p\, x_1 x_2 \ldots x_n & \rightsquigarrow & \\ y_m \ldots y_2\, q\, y_1 a x_2 \ldots x_n & & \Delta = -1 \\ y_m \ldots y_1\, q\, a\, x_2 \ldots x_n & & \Delta = 0 \\ y_m \ldots y_1 a\, q\, x_2 \ldots x_n & & \Delta = +1 \end{array}$$

Note that there is no next configuration whenever $\delta(p, x_1)$ is undefined.

Multiple Steps

Now we extend the "one-step" relation to multiple steps by induction:

- one step
  $C \left|\frac{1}{\mathcal{M}}\right. C' \iff$ as on the last slide
- exactly t steps
  $C \left|\frac{t}{\mathcal{M}}\right. C' \iff \exists\, D \left(C \left|\frac{t-1}{\mathcal{M}}\right. D \wedge D \left|\frac{1}{\mathcal{M}}\right. C'\right)$
- any finite number of steps
  $C \left|\frac{}{\mathcal{M}}\right. C' \iff \exists\, t \left(C \left|\frac{t}{\mathcal{M}}\right. C'\right)$

Lemma

The relation $\vdash_{\mathcal{M}}^{t}$ is primitive recursive, uniformly in t.

Meaning that there is a primitive recursive relation $R \subseteq \mathbb{N}^3$ such that

$$R(t, C, C') \iff C \vdash_{\mathcal{M}}^{t} C'$$

Here the configurations are coded as naturals simply by using sequence numbers. Say, let $\Sigma = [n]$ and $Q = [n{+}1, m]$, set

$$\langle C \rangle = \langle y_k, \ldots, y_1, q, x_1, \ldots, x_\ell \rangle$$

Look at the website if you want to know details about primitive recursive functions and coding tricks such as sequence numbers.

Warning

In the context of abstract computation, the relation $\vdash^{t}_{\mathcal{M}}$ is easy. But the relation $\vdash_{\mathcal{M}}$ most emphatically is not.

In fact, if we can compute the length of a computation in a primitive recursive manner, then the whole function is already primitive recursive: we just run the Turing machine an appropriate number of steps. If not, we are sunk: we just have to run the machine without any idea if and when it might stop.

So the difference between primitive recursive and Turing computable is just one unbounded search, one existential quantifier. As we will see, this makes a world of difference.

Input and Output

Given any input $x = x_1 x_2 \ldots x_n \in \Sigma^\star$, the initial configuration for x is

$$C_x^{\mathsf{init}} = q_{\mathrm{init}} \sqcup x_1 x_2 \ldots x_n$$

A halting configuration is of the form $y\, p\, x$ where $\delta(p, x_1)$ is undefined. Thus, a halting configuration has no next configuration.

An output configuration is of the form

$$C_y^{\mathsf{halt}} = q_{\mathrm{halt}} \sqcup y_1 y_2 \ldots y_m$$

Thus an output configuration is in particular halting, but there may well be other halting configurations: the machine may get stuck.

Conventions

For the initial configuration, we have chosen to place the head at the last blank to the left of the input symbol, there are lots of other possibilities (e.g, $q_{\text{init}}x_1 \dots x_n$).

For a halting configuration to be an output configuration, we require our machines to erase the tape except for the output, and to position the tape head properly before halting. This makes it fairly easy to compose two machines sequentially. Of course, it makes it harder to design the machine in the first place.

It is often convenient to assume that the input $\boldsymbol{x}$ contains no blanks (this makes it easy to go to the end). Alternatively, one can allow single blanks to separate parts of the input (a double blank would indicate the end of the input).

Exercises

Exercise

Explain in detail how to deal with blanks in the input.

Exercise

Come up with a way to associate output with arbitrary halting configurations. Make sure that it is computationally easy to read off the output.

Exercise

Can your machines be simulated by machines conforming to our definitions?

Halting and Output

We say that machine $\mathcal{M}$ halts on input x if

$$C_x^{\mathsf{init}} \vdash_{\mathcal{M}} C$$

where C is some halting configuration.

We say that $y \in \Sigma^\star$ is the output of the computation of machine $\mathcal{M}$ on input $x \in \Sigma^\star$ if

$$C_x^{\mathsf{init}} \vdash_{\mathcal{M}} C_y^{\mathsf{halt}}$$

Note that is trivial to read off the actual output string, given the corresponding output configuration.

Computing a Function

Similarly, $\mathcal{M}$ computes the partial function $f : \Sigma^\star \rightharpoonup \Sigma^\star$ if, for all $x \in \Sigma^\star$,

- If f is defined on x, then $C_x^{\mathsf{init}} \vdash_{\mathcal{M}} C_y^{\mathsf{halt}}$ and $f(x) = y$.
- If f is undefined on x, the computation of $\mathcal{M}$ on x does not halt.

With a view towards algorithms, one might object that total functions are more appropriate, we want all computations to be finite. As we will see shortly, partial functions are baked into the foundations of computability, there is no way to avoid them.

Computability Defined

Definition

A partial function $f : \Sigma^\star \nrightarrow \Sigma^\star$ is (Turing) computable if there is a Turing machine $\mathcal{M}$ that computes f.

There is a mountain of evidence that, for any reasonable model of computation $\mathfrak{M}$, it turns out that $\mathfrak{M}$-computable is equivalent to Turing computable, so it makes sense to simple say computable, without reference to any particular model.

Halting

To hammer this home: Turing machines naturally produce partial functions, this is a feature that is hardwired in the definitions. Of course, some machines halt on all inputs and produce a total function, but as we will see, this is not a property that is easy to check.

We can construct machines that are guaranteed to halt, e.g., by attaching a clock that simply interrupts the computation at some point (and returns some default value if it has not finished by then). But in general, all we get is partial functions.

As it turns out, this is actually a lifesaver: without partial functions we are automatically sunk.

Why Not Total?

Suppose we have some definition of computability that uses only total functions. For simplicity, say we are dealing with arithmetic functions $\mathbb{N}^k \to \mathbb{N}$.

Clearly we can express our computable functions by an index, a natural number e that encodes the details of the definition. This is not hard to do for any known model of computation.

But then a function that takes as input an index e and an argument x, and evaluates the corresponding function on x, must also be computable.

This is just what any interpreter for a real programming language does. What could go wrong?

Disaster Strikes

Let's call our interpreter eval, and let's define a new function f as follows:

$$f(x) := \mathsf{eval}(x, x) + 1$$

Since we assume we have only total functions, f is also total. And f is clearly computable since eval is. OK, but then f has some index e. Disaster strikes via diagonalization:

$$f(e) = \mathsf{eval}(e, e) + 1 = f(e) + 1$$

So under very reasonable assumptions, we are forced to allow some of our computable functions to be partial.

Dealing with Divergence

Since we cannot avoid partial functions, it is helpful to adjust notation a bit. We write

$$f(x)\downarrow \qquad f(x)\uparrow$$

to indicated that partial function f on input x is defined/undefined. We'll also say f converges/diverges.

Given expressions α, β involving partial functions, we use Kleene equality:

$$\alpha \simeq \beta$$

to indicate that either

- both α and β are defined (the computations involved all terminate) and have the same value, or
- both α and β are undefined (some computation diverges).

A Typical Glitch

Suppose $\mathcal{M}$ is some arbitrary Turing machine.
Does $\mathcal{M}$ always compute some partial function f?

Sadly, no. $\mathcal{M}$ might halt on some input, but not in an output configuration (in other words, the machine gets stuck somewhere). But we can fix this: we can construct a new machine $\mathcal{M}'$ such that

- if $\mathcal{M}$ halts on x with output y, then $\mathcal{M}'$ does the same,
- for all other x, $\mathcal{M}'$ does not halt on x.

So $\mathcal{M}'$ computes the partial function that $\mathcal{M}$ meant to compute.

Exercise

Show how to construct $\mathcal{M}'$ from $\mathcal{M}$.

Convention

From now on, we assume that all our Turing machines compute partial functions.

This is entirely reasonable, since we easily construct the nice machined $\mathcal{M}'$ from the original machine $\mathcal{M}$ (the construction is easily primitive recursive). By contrast, assuming that all machines compute total functions is utterly unreasonable and wrecks any further discussion.

Turing Decidability

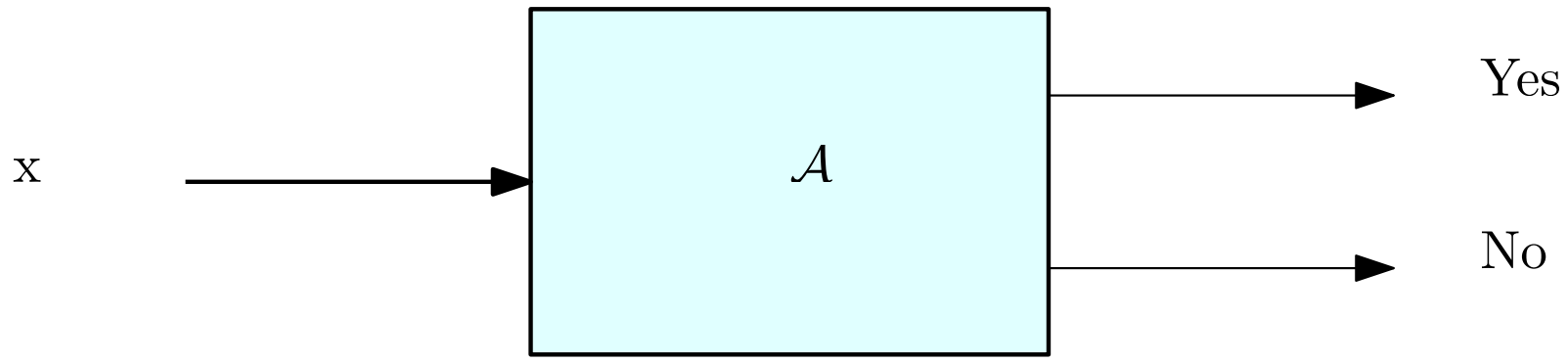
So we have notion of computability for functions, and we can use the standard trick of characteristic functions to translate it to sets and relations.

$$\text{char}_A(\boldsymbol{x}) = \begin{cases} 1 & \boldsymbol{x} \in A \\ 0 & \text{otherwise.} \end{cases}$$

Definition

A set (relation) $A \subseteq \Sigma^\star$ is (Turing) decidable if its characteristic function is Turing computable.

We use the same terminology for $A \subseteq \mathbb{N}^k$ via encoding.



The machine $\mathcal{A}$ always halts.

Acceptors versus Transducers

Informally, a problem is decidable if there is a decision algorithm $\mathcal{A}$ that returns Yes or No depending on whether the input has the property in question.

Again, these Turing machines must always halt, and return a one-bit output. They are often called acceptors.

By contrast, a Turing machine that computes some arbitrary (and possibly partial function) is called a transducer[‡].

[‡]The same terminology is also used for finite state machines.

Lemma

The decidable sets are closed under intersection, union and complement. In other words, the decidable sets form a Boolean algebra.

Proof.

Consider two decidable sets $A, B \subseteq \mathbb{N}$. We have two TMs $\mathcal{M}_A$ and $\mathcal{M}_B$ that decide membership.

Since we are dealing with Turing machines, we can simply run both $\mathcal{M}_A$ and $\mathcal{M}_B$ on input x sequentially.

It is straightforward to process the output of the two runs and return the correct yes/no answer.

□

Rant

The last proof is a perfect example of why Turing machines are a royal pain to use. After exploring them for a little bit, one may easily become convinced that one could, *in principle*, construct a master Turing machine that handles the sequential execution of the two serf machines, and produces the right output.

But no one has ever done this in the sense of actually writing down a transition table, simply because the details are far too messy and no relevant insights could be gained from this exercise.

By contrast, the analogous result for finite state machine constructs the machines directly, and this construction is useful in a number of algorithms (text searching and model checking).

Turing Computability and Robustness

So, we have a notion of a function being computable by a Turing machine that seems to conform very well to our intuition about computability..

This notion does **not change** if we modify our definitions slightly:

- one-way infinite tapes
- multiple tapes
- different head movements
- multiple heads
- different input/output conventions
- different coding conventions

Note that without this kind of robustness our model would be of rather dubious value: each variant would produce a different notion of computability.

Exercises

Exercise

Show how to modify our definitions so that Turing machines have total transition functions, but produce the same class of computable functions.

Exercise

Prove that some of the modifications on the last slide similarly yield a type of machine that produces the same class of computable functions as our original Turing machines.

"Prove" here means: think about it for long enough so that you become convinced that an actual proof could be constructed if one really needed a detailed argument.

We can specify a model $\mathfrak{M}$ of computation by defining

- a space $\mathcal{C}$ of possible configurations (snapshots),
- a "one-step" relation,
- an input and output convention,
- a coding convention (if needed).

The details vary greatly, but we always have the same pattern.

Major Warning: Minute details about input/output/coding conventions become really important in low complexity classes; higher up they are mostly interchangeable.

Whole Computation

Given a one-step relation $C \left|\frac{1}{\mathfrak{M}}\right. C'$, multiple steps and whole computations are defined in the obvious way:

$$C \left|\frac{0}{\mathfrak{M}}\right. C' :\Leftrightarrow C = C'$$

$$C \left|\frac{t}{\mathfrak{M}}\right. C' :\Leftrightarrow \exists\, C''\ \ C \left|\frac{t-1}{\mathfrak{M}}\right. C'' \wedge C'' \left|\frac{1}{\mathfrak{M}}\right. C'$$

$$C \left|\frac{}{\mathfrak{M}}\right. C' :\Leftrightarrow \exists\, t\ \ C \left|\frac{t}{\mathfrak{M}}\right. C'$$

A computation (or a run) of $\mathfrak{M}$ is a sequence of configurations C_0, C_1, C_2, ... where $C_i \left|\frac{}{\mathfrak{M}}\right. C_{i+1}$.

Input/Output

One needs a way to provide input from some set X, a map

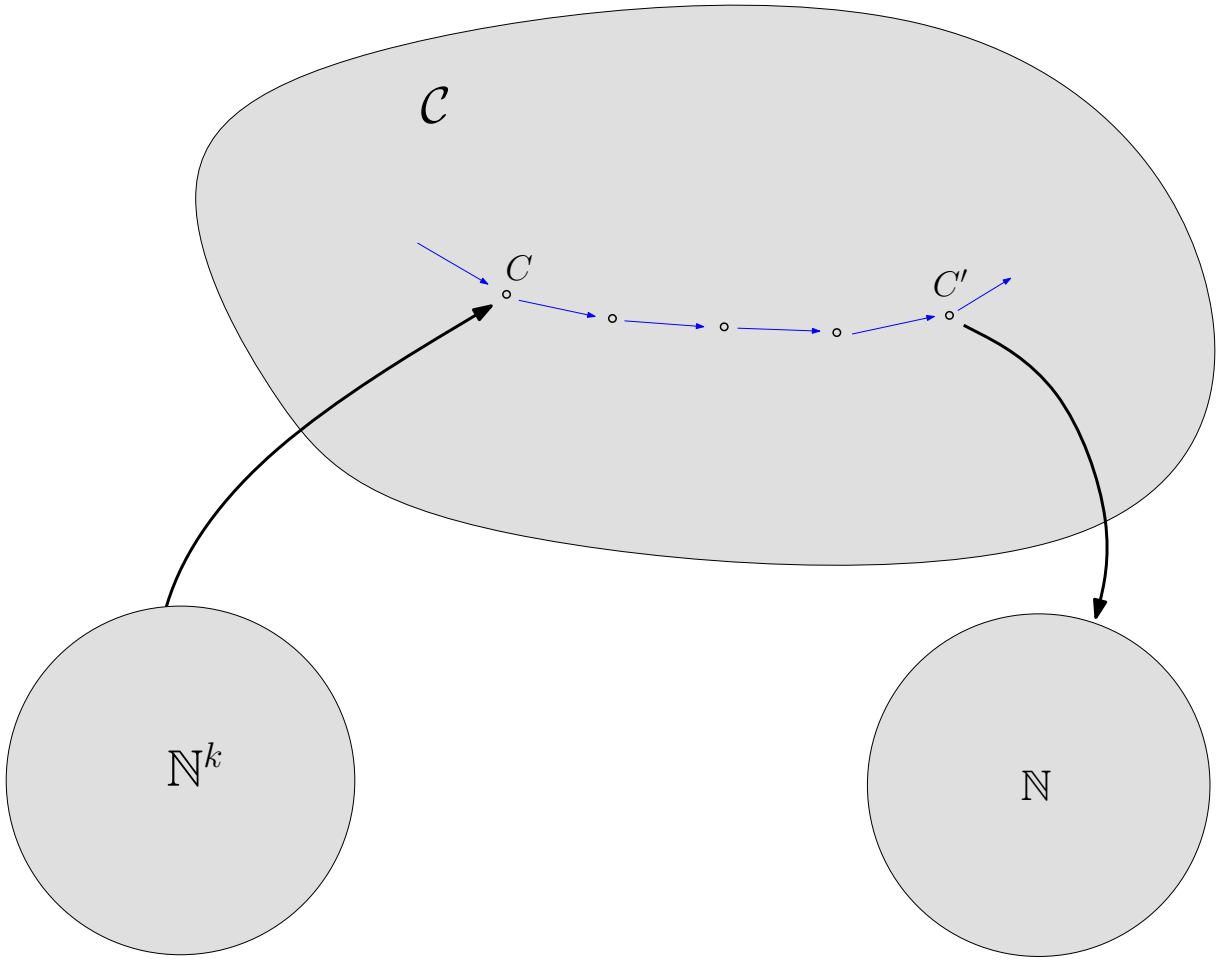
$$\mathsf{inp} : X \to \mathcal{C}$$

as well as an output map to some set Y:

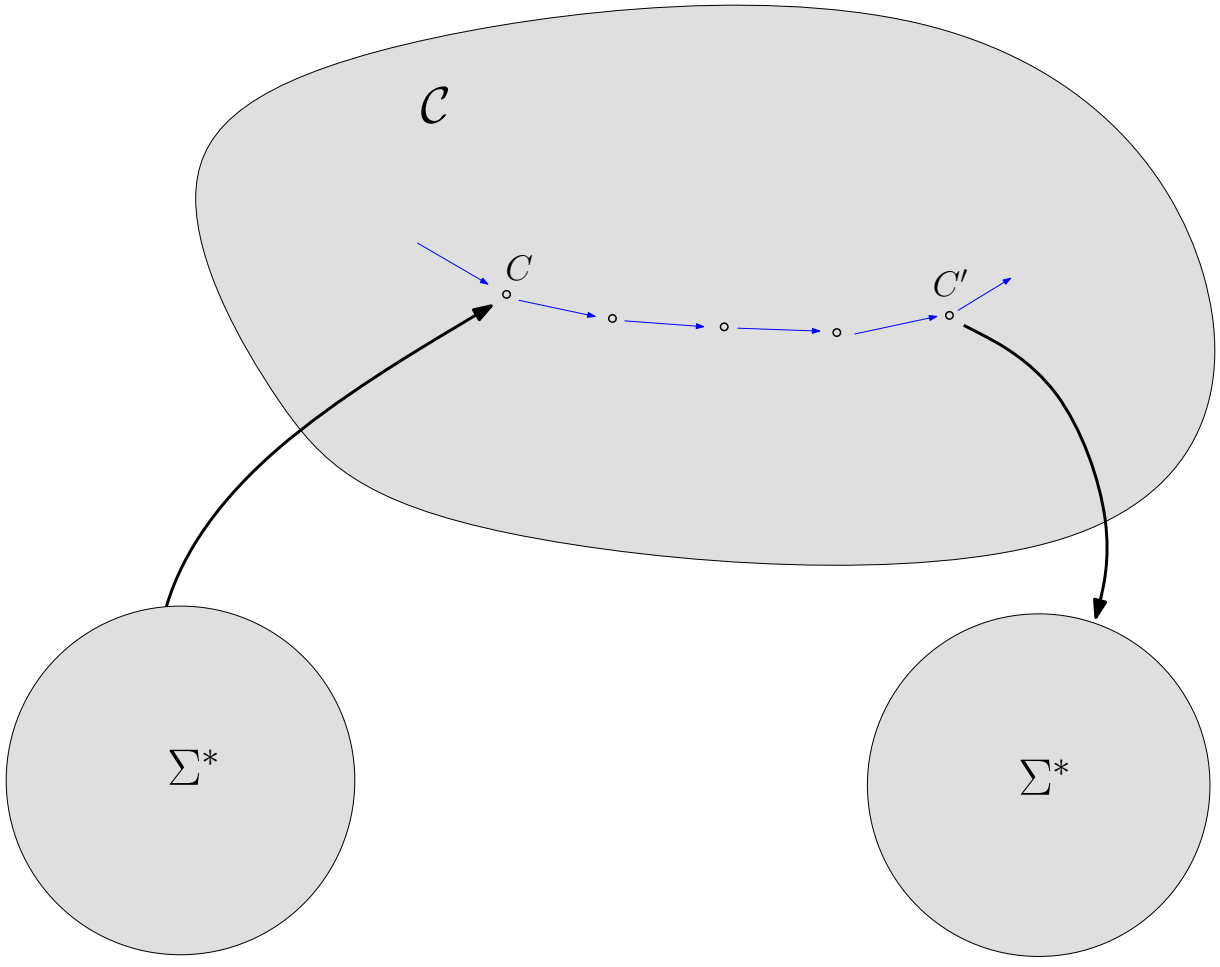
$$\mathsf{outp} : \mathcal{C} \nrightarrow Y$$

Both maps are very typically very simple, essentially just a bit of re-formatting (they do not contribute to the complexity of the computation).

A minor technical issue: the output map may only be defined on some of the configurations (the "halting" configurations).



The number-theoretic scenario: input and output are natural numbers.



The string scenario: input and output are words over some alphabet.

TMs and Strings

Recall that Turing machines are glorified type writers, they operate on strings, not the naturals.

When a TM computes an arithmetical function, the input/output conventions are typically unary or binary. OCCTM[†] that converts between the two formats, so, as far as general computability goes, it doesn't matter.

But it matters greatly in low complexity classes where we cannot tolerate an exponential time precomputation.

[†]OCCTM: one can construct a TM

Enumerations

Turing realized that the usual standard distinction between data and code is actually an illusion.

Nowadays, this observation is a snoozer, but a century ago it was a bit a a leap. No one thought of the Euclidean algorithm as being the same kind of bird as the numbers it consumes as input.

At any rate, using standard coding machinery, we can express a Turing machine $\mathcal{M}$ as a natural number $\widehat{\mathcal{M}}$, the (Turing) index for $\mathcal{M}$.

As a consequence, we can get an enumeration $(\mathcal{M}_e)_{e\geq 0}$ of all TMs.

Thinking of an index as a natural number is the arithmetic approach to life, hugely popular since Gödel's use of arithmetization in his incompleteness theorem (his infamous Gödel numbers).

Alternatively, we can think of e as a string over some suitable alphabet that encodes the machine. Of course, the binary alphabet $\mathbf{2}$ is good enough. Again we get an enumeration $(\mathcal{M}_e)_{e\in\mathbf{2}^\star}$ but this time the list is organized in length-lex order[†]. Or we could think of the whole TM as being written down as a string, it's easy to come up with some reasonable convention.

Strings or numbers, there is not much difference between the two scenarios. I will keep talking about indices as natural numbers, but you can just think of them as binary strings.

[†]Lexicographic order is a bad idea, it is not a well order.

Enumerating Computable Functions

Recall that all our machines compute partial functions, so once we have an enumeration for the machines, we also have an enumeration for all partial computable functions. Classically this was often written

$$\varphi_e \qquad \text{or} \qquad \{e\}$$

So $(\{e\})_{e\geq 0}$ is a complete listing of all these functions.

We won't deal with arity issues here, they are purely technical. So we'll happily write things like $\{e\}(x, y)$ and so on.

Keeping notation simple is better than going overboard on precision. Read Hartley Rogers' book if you think otherwise.

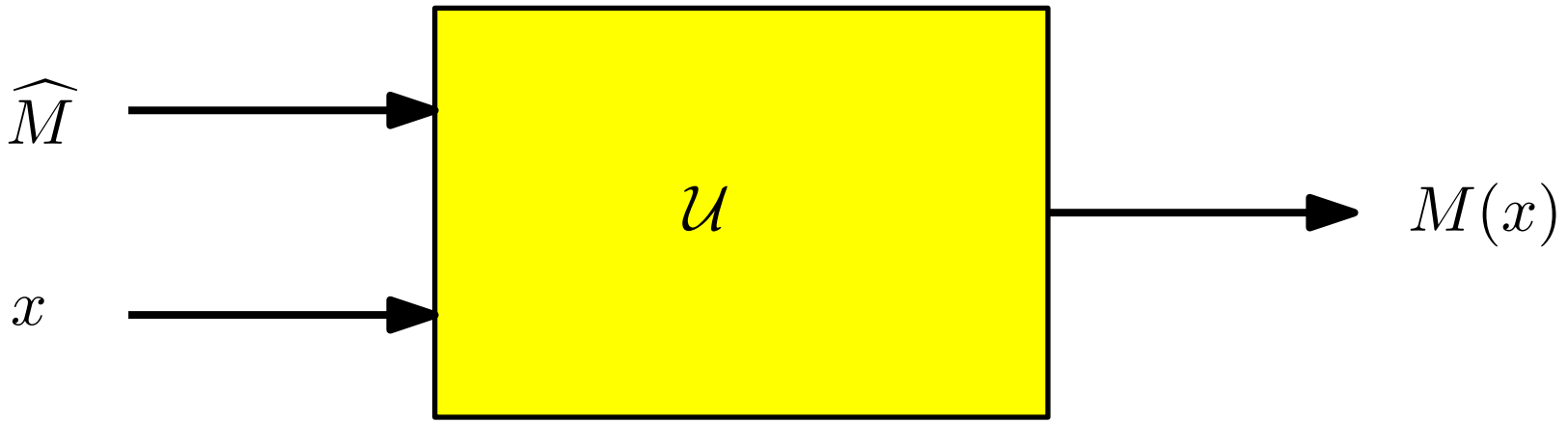
Universal Turing Machines

The fact that we can code a Turing machine as integers, and we can think of them as defining functions on integers, has an interesting consequence, first realized by Turing himself:

> OCCTM $\mathcal{U}$ that takes as input numbers e and x, interprets e as the index $\widehat{\mathcal{M}}$ of a Turing machine $\mathcal{M}$, and then simulates $\mathcal{M}$ on input x.
>
> $\mathcal{U}$ will halt on e and x iff $\mathcal{M}$ halts on x; moreover, $\mathcal{U}$ will return the same output in this case.

Any such machine $\mathcal{U}$ is called a universal Turing machine (UTM).

The details of the construction of $\mathcal{U}$ are very tedious, but it's "clear" that this can be done, see Turing's 1936 paper.



$$\mathcal{U}(\widehat{\mathcal{M}}, x) \simeq \mathcal{M}(x)$$

Comments

- If $\mathcal{U}$ is given a number/string e as input that fails to code a Turing machine we assume that $\mathcal{U}$ fails to halt. Just a convention, $\mathcal{U}$ could halt (say, with output 0).
- Note that $\mathcal{U}$ is by no means uniquely determined, there are lots and lots of choices. One has to make sure these choices do not affect the result in question.
- A very interesting question is how large a universal Turing machine needs to be. Amazingly, there is a 2-state, 5-symbol UTM. And a flaky 2-state, 3-symbol UTM.

The Key Observation

In classical computability theory, it is entirely irrelevant how the simulation is organized and how efficient it is. No problem if the running time of the simulator is exponentially larger.

This does not work in complexity theory! Fortunately, it turns out that one can make the universal machine quite fast, easily polynomial time, more later.

A beautiful and non-trivial example of OCCTM.

The Halting Problem

Problem:	**(Full) Halting Problem**
Instance:	An index e, an input x.
Question:	Does $\mathcal{M}_e$ on input x halt?

Theorem

The Halting Problem is undecidable.

There are other versions of the Halting Problem, see below, whence the qualifier "full."

Diagonalization

Suppose we have a halting tester TM $\mathcal{H}$ that checks whether $\mathcal{M}_e$ on x halts. Define a new machine $\mathcal{D}$, that, on input e, does the following:

if $\mathcal{H}(e, e) = 1$
then
    diverge
else
    halt

Let d be the index of this machine. But then $\mathcal{D}$ halts on input d iff it fails to halt, a contradiction. □

Again, OCCTM ...

Function Proof

We can express the issue also in terms of partial recursive functions.

If Halting were decidable we could define a function g by

$$g(x) \simeq \begin{cases} \{x\}(x) + 1 & \text{if } \{x\}(x) \downarrow, \\ 0 & \text{otherwise.} \end{cases}$$

Thus g is a computable, total function, so $g \simeq \{e\}$ for some e. We get a contradiction by **diagonalizing** via input e:

If $\{e\}(e) \simeq y$, then $g(e) = y + 1 = g(e) + 1$.

If $\{e\}(e) \uparrow$, then $g(e) = 0$.

Other Halting Problems

Here are two variants of our Full Halting Problem:

Problem:	**Halting Problem**
Instance:	Index e.
Question:	Does Turing machine $\mathcal{M}_e$ halt on input e?

Problem:	**Pure Halting**
Instance:	Index e.
Question:	Does Turing machine $\mathcal{M}_e$ halt on empty tape?

Pure Halting makes little sense from the perspective of computable functions, but it's perfectly fine as a question about TMs.

They Are All Undecidable

We know that Full Halting is undecidable, but the proof shows that ordinary Halting is already undecidable.

For Pure Halting, suppose we have some index e. Build a machine

- first writes e on the tape, then
- simulate $\mathcal{M}_e$ on input e.

Use a reduction:

The new machine has some index e' and we can compute e' easily from e (primitive recursive). But then a solution to Pure Halting would produce a solution to Halting, so the former must also be undecidable.

A Closer Look

We are given index e and need to decide Halting for $\mathcal{M}_e$.

There is a TM $\mathcal{T}$ that computes a new index e' for a machine $\mathcal{M}_{e'}$ takes simulates the computation of $\mathcal{M}_e$ on input e.

Then e is a yes-instance of Full Halting iff e' is a yes-instance of Pure Halting. Since e' is computable from e, Pure Halting must also be undecidable.

OCCTM, OCCTM, OCCTM, ...

Busy Beaver Problem

In 1962, Tibor Rado described a now famous problem in computability. Consider Turing machines on tape alphabet $\Sigma = \{0, 1\}$ (where 0 is the blank symbol) and n states.

> **Question:** What is the largest number of 1's any such machine can write on an initially blank tape, and then halt?

Halting is crucial, otherwise we could trivially write infinitely many 1's.

Other Variants

Rado's original question is actually slightly arbitrary, here are two versions more firmly rooted in computability theory.

Time Complexity What is the largest number of moves a halting n-state machine can make?

Space Complexity What is the largest number of tape cells a halting n-state machine can use?

Incidentally, it is standard practice to ignore the halting state in the count, so n means "n ordinary states plus one halting state." Also, one insists that that the tape head always moves.

Hierarchy

We write $\mathrm{BB}_{\mathrm{H}}(n)$ for largest time complexity of any halting n-state machine and refer to BB_{H} as the Busy Beaver function.

We will also consider the original version of the problem and write $\mathrm{BB}_{\mathrm{W}}(n)$ for the largest number of 1's written on the tape when an n-state machine halts.

Clearly, $\mathrm{BB}_{\mathrm{H}}(n) \geq \mathrm{BB}_{\mathrm{W}}(n)$, but the former has the advantage of relating more directly to the Halting Problem, which one would suspect to be the central issue with busy beaver functions.

Busy Beaver $n = 1$

$\mathrm{BB}_{\mathrm{H}}(1) = 1$

To see this, note that any attempt to make a second move would already lead to an infinite loop.

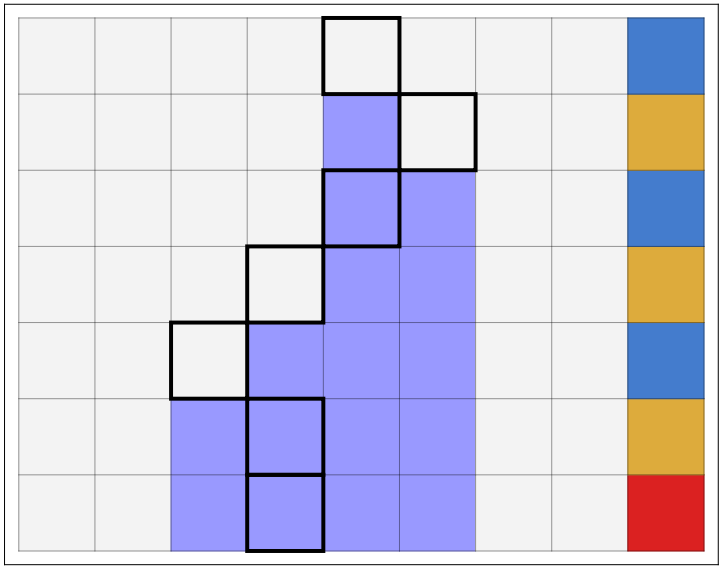
Similarly, $\mathrm{BB}_{\mathrm{W}}(1) = 1$.

Amazingly, the answer is no longer obvious: $\mathrm{BB}_{\mathrm{W}}(2) = 4$ and $\mathrm{BB}_{\mathrm{H}}(2) = 6$ with the same champion.

	0	1
p	(q,1,R)	(q,1,L)
q	(p,1,L)	halt

$$p0 \vdash 1q0 \vdash p11 \vdash q011 \vdash p0111 \vdash 1q111$$

Orbit



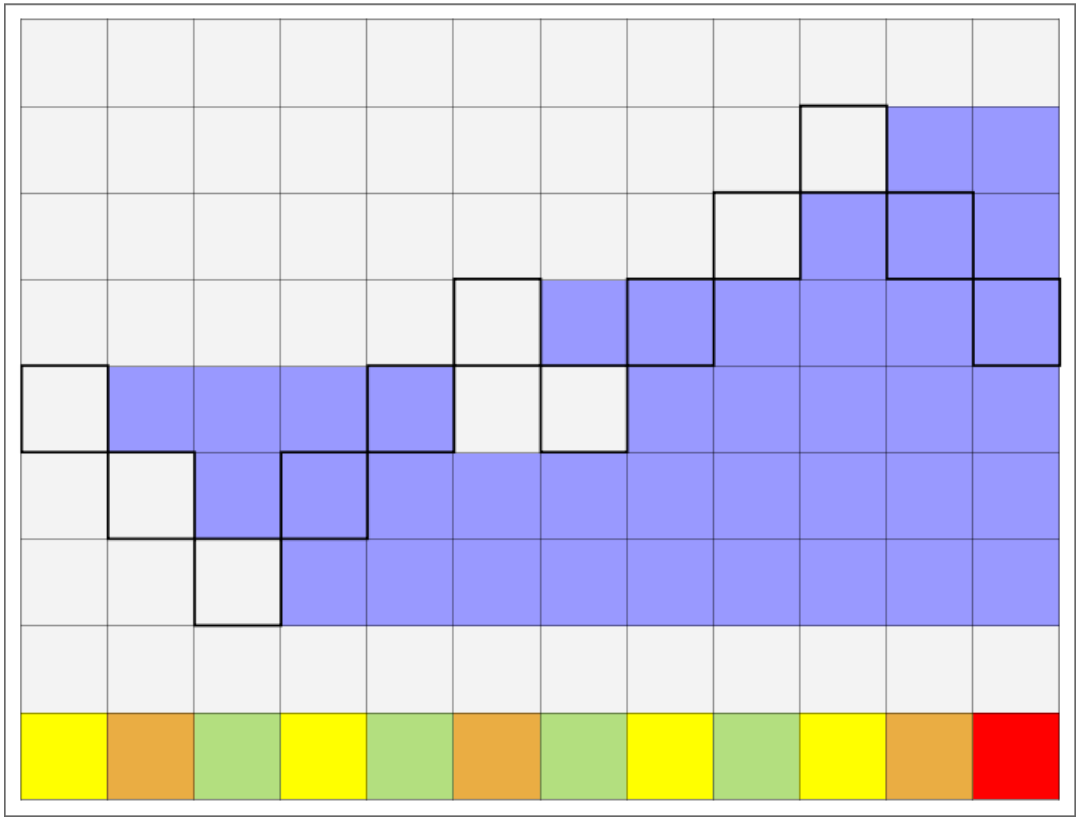
Busy Beaver $n = 3$

Here things start to get messy: there are $4\,826\,809$ Turing machines to consider.

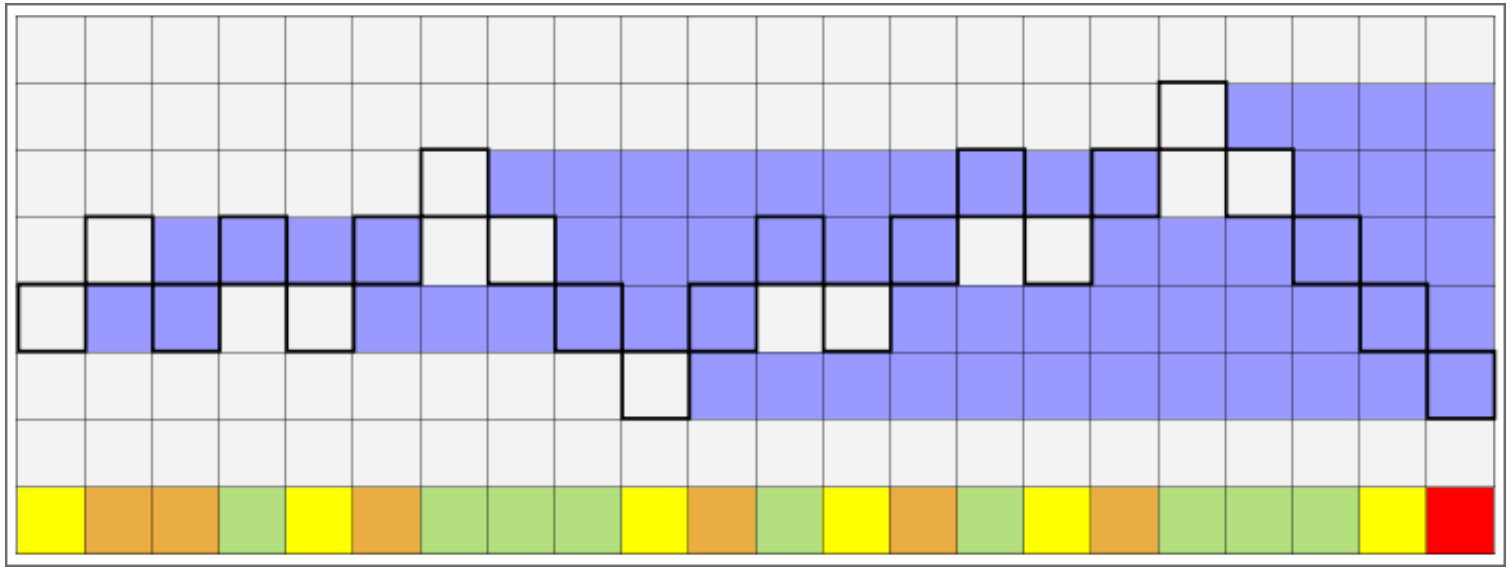
Exploiting isomorphisms, filtering out machines where all 4 states are reachable (in the diagram, not necessarily the computation on empty tape), and checking for halting we get down to $405\,072$.

From the last group we can pick out the champions.

Write-Champion



Halt-Champion



How bad can it be?

The number of machines quickly becomes very difficult to manage:

n	#machines
4	$6\,975\,757\,441$
5	$16\,679\,880\,978\,201$

As usual, the problem is not isomorph-rejection (which requires constructing all machines first), but to only build non-isomorphic ones to begin with. And, given these numbers, it won't make much of dent no matter what.

6-State Busy Beaver

For $n = 6$ all hell breaks loose, we have only lower bounds. For example, it has been known for a while that

$$\mathrm{BB}_{\mathrm{W}}(6) \geq 4.6 \times 10^{1439}$$

The raw search space here has size $59\,604\,644\,775\,390\,625$, but this can be improved a bit exploiting symmetries and reachability.

Halting gets very messy here, though: there is no good heuristic as to when the run should be truncated (leading to the conclusion that the machine is not halting).

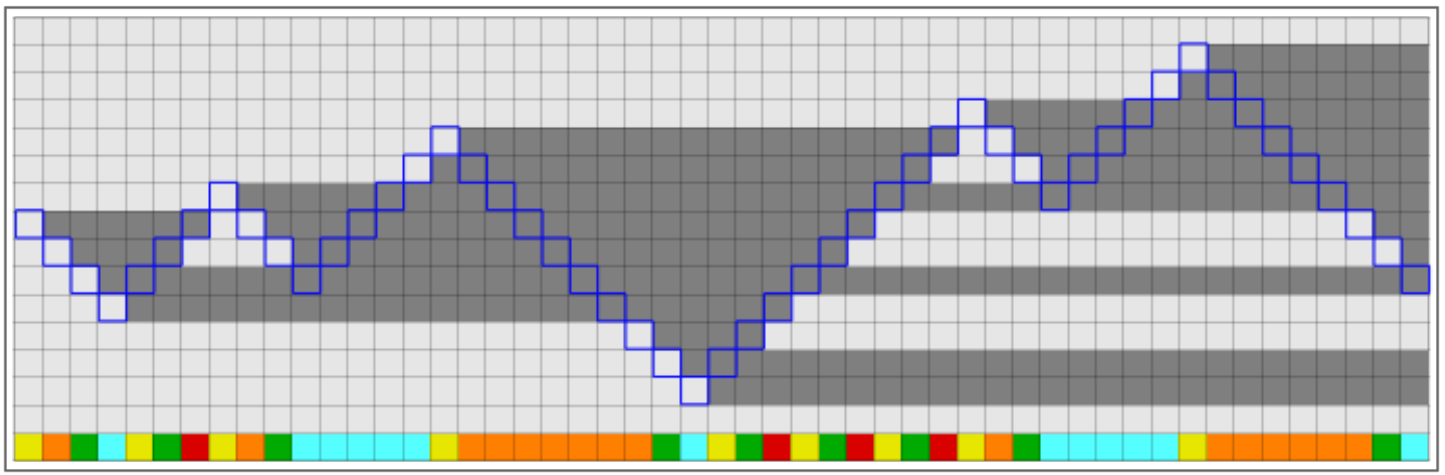
The Marxen-Buntrock Machine

The current 5-state champion was found by Marxen and Buntrock, and its discovery is a small miracle. Here is the table of the machine. Clearly all 5 states plus the halt state are reachable in the diagram.

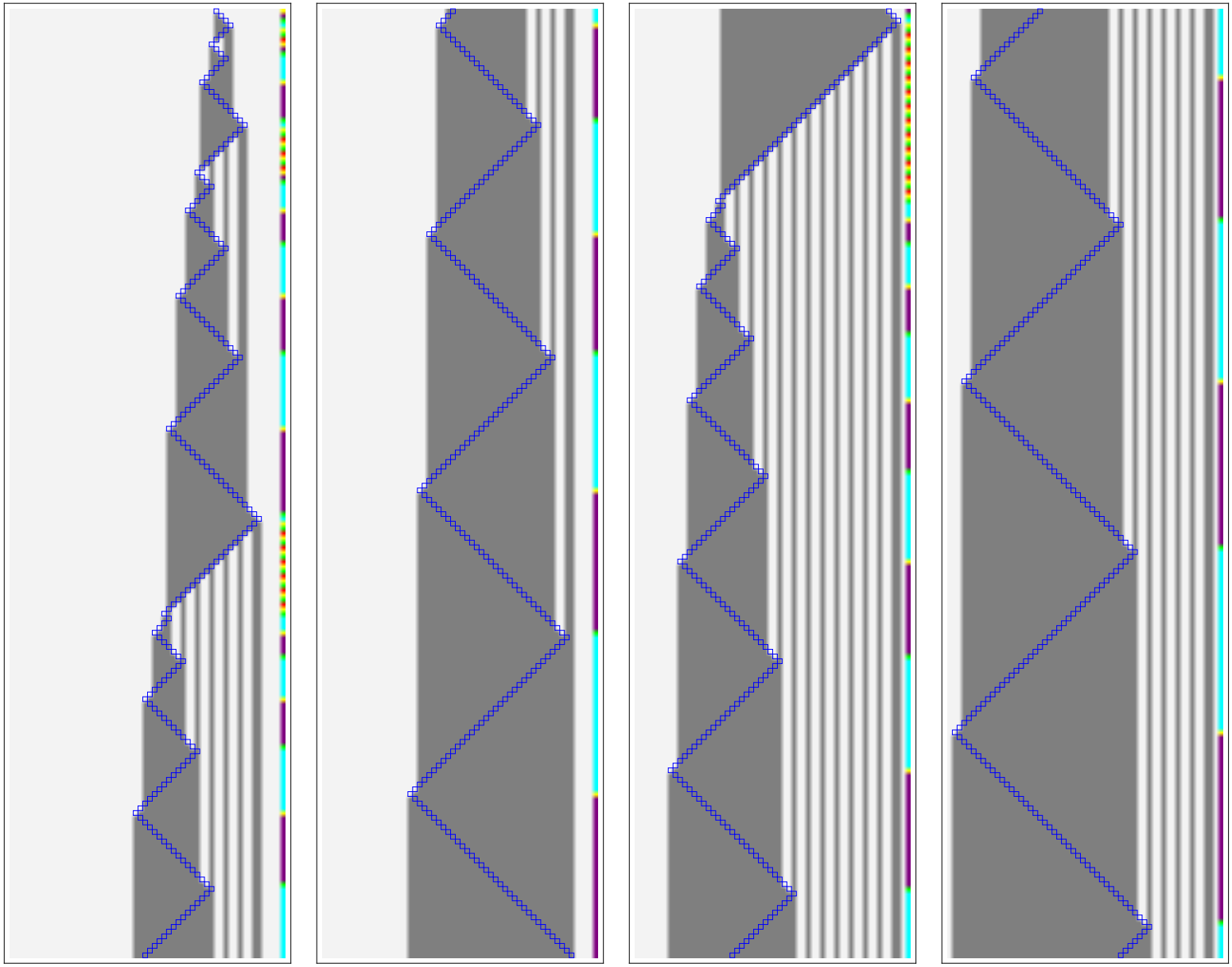
	0	1
1	(2,1,R)	(3,1,L)
2	(3,1,R)	(2,1,R)
3	(4,1,R)	(5,0,L)
4	(1,1,L)	(4,1,L)
5	halt	(1,0,L)

Of course, that's nowhere near enough: they need to appear in the computation on empty tape.

50 Steps



400 Steps



Misleading Pictures

Looking at a run of the Marxen-Buntrock machine for a few hundred or even a few thousand steps one invariably becomes convinced that the machine never halts: the machine zig-zags back and forth, sometimes building solid blocks of 1's, sometimes a striped pattern.

Whatever the details, the machine seems to be in a "loop" (not a an easy concept to clarify for Turing machines). Bear in mind: there are only 5 states, there is no obvious method to code an instruction such as "do some zig-zag move 1 million times, then stop".

And yet, this machine

> stops after $47\ 176\ 870$ steps
> writes $10(100)^{4097}$

Why is this Hard?

There are several fundamental obstructions to computing busy beaver numbers, in increasing levels of depth.

- Brute-force search quickly becomes infeasible, even for single-digit values of n.
- The Halting conundrum: Even if we could somehow deal with combinatorial explosion, there is the problem that we don't know if a machine will ever halt – it might just keep running forever.
- Reasoning about the behavior of Turing machines in a formal system like Peano arithmetic or Zermelo-Fraenkel set theory is necessarily of limited use.

State of the Art

n	$\mathrm{BB}_{\mathrm{H}}(n)$	$\mathrm{BB}_{\mathrm{W}}(n)$
1	1	1
2	6	4
3	21	6
4	107	13
5	$\geq 47\,176\,870$	≥ 4098
6	$> 7.4 \times 10^{36\,534}$	$> 3.5 \times 10^{18\,267}$

Concrete values are only available for $n \leq 4$, beyond that, we only have bounds. And these bounds soon get ridiculous:

$$\mathrm{BB}_{\mathrm{H}}(7) > 10^{2 \cdot 10^{10^{10^{18\,705\,353}}}}$$

Alas, these results are not as robust as one would like them to be, see Harland 2016 for a critique.

Semidecidability

Halting is undecidable, so it is natural to ask whether there is a reasonable description of the level of difficulty associated with Halting. In more modern terms: is there a complexity class that Halting naturally lives in?

Definition

A set $A \subseteq \mathbb{N}^k$ is (Turing) semidecidable if there is a Turing machine $\mathcal{M}$ that halts precisely on all $\boldsymbol{x}$ such that $\boldsymbol{x} \in A$.

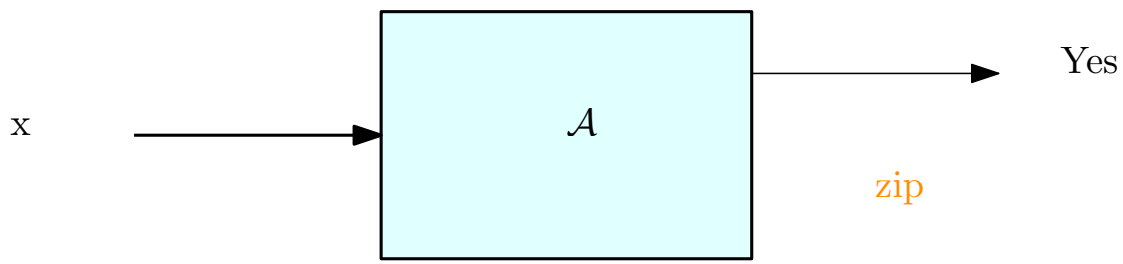
Proposition

Halting is semidecidable.

Quoi?

In terms of algorithms, this is quite weird: we have a "semi-decision algorithm" for A, a broken decision algorithm that

- correctly returns the answer Yes, if the answer is indeed Yes, but
- diverges and runs forever if the answer is No.



Who Cares?

Semidecidability may appear to be a rather useless idea in the world of real algorithms. Alas, it is arguably more fundamental than decidability, trust me. Here is the key connection.

Lemma

$A \subseteq \mathbb{N}$ is decidable iff A and $\mathbb{N} - A$ are both semidecidable.

Note that it follows from the lemma that not-Halting, the complement of Halting, is not even semidecidable The complements of semidecidable sets are called co-semidecidable.

This is the tip of an iceberg: decidable, semidecidable and co-semidecidable problems are just the very bottom of an infinite hierarchy of increasingly complicated and highly non-computable problems.

Proof

Left to right is obvious: we can turn a decision algorithm for A into semi-decision algorithms for A and for $\mathbb{N} - A$.

For the opposite direction, suppose we have two TMs $\mathcal{M}$ and $\mathcal{M}'$ that semidecide membership in A and $\mathbb{N} - A$, respectively.

Note that we cannot run the two machines sequentially: the first one might diverge, when the second one would converge.

But, we can run them in parallel: alternate between $\mathcal{M}$ and $\mathcal{M}'$, one step at a time. Since $\mathbb{N} = A \dot{\cup} (\mathbb{N} - A)$, exactly one of them will halt, producing the correct answer.

□

Lemma

The semidecidable sets are closed under intersection and union.

Proof.

Consider two semidecidable sets $A, B \subseteq \mathbb{N}$. We have two TMs $\mathcal{M}_A$ and $\mathcal{M}_B$ that semidecide membership.

For intersection, we can run the machines sequentially: if both terminate we halt.

For union, we run the machines in parallel, and halt as soon as one of them terminates.

This produces the right behavior: halting if the condition is satisfied, divergence otherwise.

□

One traditionally codes problems as sets $A \subseteq \mathbb{N}$ and writes

Δ_1	decidable sets
Σ_1	semidecidable sets
Π_1	co-semidecidable sets

Theorem

The classes Δ_1, Σ_1 and Π_1 are all distinct.

There is another way to look at semidecidable sets: one can generate them in a computable manner.

Definition

$A \subseteq \mathbb{N}$ recursively enumerable (r.e.) if there is a partial Turing computable function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ such that A is the range of f.

The special case $A = \emptyset$ is a nuisance here, some authors prefer to keep it separate, and insist that f is total, otherwise. In general that would mean that a particular element in A can be enumerated repeatedly, even infinitely often (that is actually useful on occasion).

Cleaning Up

As defined, the support of the enumeration function f could be anything. That's often inconvenient, we will show that the support of f can always be chosen to be an initial segment of $\mathbb{N}$.

Lemma

We may assume without loss of generality that the support of f is $\mathbb{N}$ or $\{0, 1, \ldots, n-1\}$ for some n, and that f is injective.

In essence, f is just a table (finite or infinite), containing one element of A in each row, with every element appearing exactly once. This is trivial in the la-la land of set theory, but we want the function to be computable.

A few preliminaries before the proof.

Two Approaches

There are two basic approaches towards studying the complexity of a collection A of objects:

Recognition Develop an algorithm that, given an object, determines whether it belongs to A.

Generation Develop a method that allows one to generate all objects in A in some systematic fashion.

Obviously these tasks are closely connected, but not quite the same: recognition is more difficult in general.

Connection

Suppose $A \subseteq \mathcal{S}$ where $\mathcal{S}$ is some ambient set. For us right now, $\mathcal{S} = \mathbb{N}$.

- Recognition to Generation
  Run through the elements of $\mathcal{S}$ in some systematic fashion. For each object x, check if x is in A. If so, output x; otherwise just keep going.

- Generation to Recognition
  Given an object x, start generating A. If x ever pops up, return Yes.

The step from Generating to Recognition only produces a semidecision procedure. We need some additional information to allow us to truncate the search. For example, we might know that after a while, only objects "larger" than x are generated, so at that point we can give up.

Interrupting a Computation

Suppose we have a Turing machine $\mathcal{M}$ that computes some function f. A key idea is that we can interrupt a computation at σ steps. We refer to σ as a stage in the computation.

$$f_\sigma(x) \simeq \begin{cases} y & \text{if } C_x^{\mathsf{init}} \overset{<\sigma}{\underset{\mathcal{M}}{\vdash}} C_y^{\mathsf{halt}} \\ \sigma & \text{otherwise.} \end{cases}$$

So $f_\sigma(x) = \sigma$ signals that the computation has not converged so far, though it might converge later.

We may safely assume that

$$f_\sigma(x) = y \quad \text{implies} \quad x, y < \sigma$$

Claim: $f_\sigma(x)$ is easy to compute (say, primitive recursive).

If $\mathcal{M}$ halts on x, then $f_\sigma(x) \simeq y$ for some y and all stages $\sigma \geq \sigma_0$.

Otherwise, $f_\sigma(x) = \sigma$ for all σ.

One could think of f as being the limit of the f_σ:

$$f(x) \simeq \lim_{\sigma \to \infty} f_\sigma(x)$$

Definitions in Stages

It is often convenient to define computable functions in stages: one explains what to do for each stage $\sigma \geq 0$.

> **Stage σ:**
> Perform some operations depending on σ and previous stages.
> Possibly define $f(x) = y$ for some $x, y < \sigma$.

The operations at stage σ are "trivial" (say, primitive recursive), they are guaranteed to take only finitely many steps.

We are allowed to define $f(x)$ only once, we cannot change our mind at a later stage.

Dovetailing

Using stages, one can in effect run infinitely many computations in parallel. This method is often referred to as dovetailing:

Problem:	**Halts Somewhere**
Instance:	A partial computable function f.
Question:	Is there some input x such that f on x converges?

Obviously we cannot simply try f on $0, 1, \ldots$ sequentially. But we can use stages:

> **Stage σ:**
> Compute $f_\sigma(0), f_\sigma(1), \ldots, f_\sigma(\sigma{-}1)$.

We stop as soon as one of these truncated computations halts. So Halts Somewhere is semidecidable.

Proof of Lemma

Given an arbitrary enumeration f, we construct a new well-behaved enumeration function g as follows.

We proceed in stages $\sigma \geq 0$. Initially, set a counter z to 0.

> **Stage σ:**
>
> Compute $f_\sigma(0), \ldots, f_\sigma(\sigma{-}1)$.
>
> If a new value y appears, set $g(z) \simeq y$ and increment z.

Then g is computable, injective, has the same range as f and its support[†] spt f is an initial segment of $\mathbb{N}$, as required.

□

[†]The support of a partial function is the set of inputs for which it is defined; unfortunately called domain (of definition) by some.

Lemma

A set $A \subseteq \mathbb{N}$ is semidecidable iff it is recursively enumerable.

Proof. Suppose A is semidecidable, say $A = \operatorname{spt} f$ for some computable function f.

We construct a computable function g such that $A = \operatorname{rng} g$ in stages:

Stage σ:

Compute $f_\sigma(0), \ldots, f_\sigma(\sigma{-}1)$.

If a new value $x < \sigma$ such that $f(x) \downarrow$ appears, set $g(z) \simeq x$ and increment z.

Opposite Direction

Suppose $A = \operatorname{rng} g$.

We construct a computable function f such that $A = \operatorname{spt} f$ in stages:

> **Stage σ:**
>
> Compute $g_\sigma(0), \ldots, g_\sigma(\sigma{-}1)$.
>
> If a new value $g(x) \simeq y < \sigma$ appears, set $f(y) \simeq 0$.

By construction, f is computable and is defined exactly on all the numbers in the range of g.

□

Aside: Hilbert's Dream

Recall Hilbert's dream of finding a simple axiom system $\mathcal{H}$ that is consistent and complete, and suffices to express all of math (actually, plain arithmetic is enough of a challenge)?

It is easy to see that the theorems provable in any formal system are recursively enumerable:

- enumerate all possible sequences of formulae in length-lex order
- filter out the ones that are actual proofs, and output their results.

The first step is purely combinatorial, and the second one exploits the fact that being a valid proof is a purely syntactical property. No problem for a Turing machine.

Completeness Kills

Now suppose you have some conjecture φ, a potential theorem.

Since Hilbert wanted $\mathcal{H}$ to be consistent and complete, the enumeration of theorems from the last slide must always produce either φ or $\neg\varphi$.

Et voila, we can solve the Entscheidungsproblem for $\mathcal{H}$.

To be clear, this works fine for some limited systems like Presburger arithmetic (addition only) or Abelian groups, but in general we have a catastrophic failure: a lot of areas of math are inherently undecidable.

Upcoming Attractions

The relationship between

$$\Delta_1 \text{ vs } \Sigma_1$$

is entirely analogous to

$$\mathbb{P} \text{ vs } \mathbb{NP}$$

Alas, we know how to use diagonalizing to separate Δ_1 from Σ_1, but we currently have no idea how to separate $\mathbb{P}$ from $\mathbb{NP}$.

In fact, some noteable researchers (Knuth, Levin) think the last two classes might be the same.