CDM

Register Machines

Klaus Sutner

Carnegie Mellon University

Fall 2024

Where Are We?

The examples from last lecture show that primitive recursive functions are not enough to serve as a definition of computability—even though they encompass a lot of functions that fail to be efficiently computable.

General Recursion Some intuitively computable functions, based on a general type of recursion, fail to be primitive recursive.

Evaluation Computability forces functions to be partial in general, we need to adjust our framework correspondingly.

Insane Growth Some computable total functions have stupendous growth rates that are essentially incomprehensible.

A Different Model

What now? We will turn our problems into a solution: concoct a model of computation that, by design, can handle Ackermann, Friedman's α (and other perverse examples of computable functions) and partial evaluation.

We will do this by using a machine model, another critical method to define computability and complexity classes. There are many plausible approaches, we will use a model that is slightly reminiscent of assembly language programming, only that our language is much, much simpler than real assembly languages.

Functions computed by these machines will turn out to be partial in general, so this might fix all our problems.

Legitimate Question: Why Not Turing Machines?

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.

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.

And they work extremely well in the context of complexity theory, unlike some of the other models. Since we are interested in abstract computability, this is not a central concern for us.

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 a Turing machine, on input n, can compute the nth prime.

Proofs in complexity theory using TMs are often incredibly tricky and use very clever and intricate constructions. The justification is typically: "clearly, one can construct a TM that does such-and-such . . . " Looking at these proofs, one often has the sense that the argument may well be correct, but things feel a bit iffy.

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

Nothing New

This is not a personal gripe of mine, the difficulties with TMs have been observed any number of times.

> . . . a Turing machine has a certain opacity . . . a Turing machine is slow in (hypothetical) operation and, usually, complicated. This makes it rather hard to design it, and even harder to investigate such matters as time or storage optimization or a comparison between efficiency of two algorithms.
>
> Z.A. Melzak 1961

We will use register machines instead.

Register Machine

Definition

A register machine (RM) consists of a finite number of registers and a control unit.

We write R_0, R_1, ... for the registers and $[R_i]$ for the content of the ith register: a single natural number.

Note: there is no bound on the size of the numbers stored in our registers, any number of bits is fine. This is where we break physics.

The control unit is capable of executing certain instructions that manipulate the register contents.

Instruction Set

Our instruction set is very, very primitive:

- `inc r k`
  increment register R_r, goto k.
- `dec r k l`
  if $[R_r] > 0$ decrement register R_r and goto k, otherwise goto l.
- `halt`
  well ...

The gotos refer to line numbers in the program; note that there is no indirect addressing. These machines are sometimes called counter machines.

Definition

A register machine program (RMP) is a sequence of RM instructions $P = I_0, I_1, \ldots, I_{\ell-1}$.

For example, the following program performs addition:

```
// addition    R0 R1 --> R2
  0:    dec 0   1   2
  1:    inc 2   0
  2:    dec 1   3   4
  3:    inc 2   2
  4:    halt
```

Hardware versus Software

Since we have no intentions of actually building a physical version of a register machine, this distinction between register machines and register machine programs is slightly silly.

Still, it's good mental hygiene: we can conceptually separate the physical hardware that supports some kind of computation from the programs that are executed on this hardware. For real digital computers this makes perfect sense. A similar problem arises in the distinction between the syntax and semantics of a programming language.

And, it leads to the juicy question: what is the relationship between physics and computation? We'll have more to say about this in a while.

Definition

A function is RM-computable if there is some RMP that implements the function.

This is a bit wishy-washy: we really need to fix

- a register machine program P,
- input registers I, and
- an output register O.

Then (P, I, O) determines a partial function $f : \mathbb{N}^k \nrightarrow \mathbb{N}$ where $k = |I|$.

A Reasonable I/O Convention

- Given input arguments $\boldsymbol{a} = (a_1, \ldots, a_k) \in \mathbb{N}^k$, set the input registers: $[R_i] = a_i$.
- All other registers are initialized to 0.
- Then run the program.
- If it terminates, read off the value: $f(\boldsymbol{a}) = [R_0]$.
- If the program does not terminate, $f(\boldsymbol{a})$ is undefined.

Run the Program?

To describe a computation of a RMP P we need to explain what a snapshot of a computation is, and how get from one snapshot to the next. Clearly, for RMPs we need two pieces of information:

- the current instruction, and
- the contents of all registers.

Definition

A configuration of P is a pair $C = (p, \boldsymbol{x}) \in \mathbb{N} \times \mathbb{N}^n$.

Steps in a Computation

Here is a very careful definition of what it means that a configuration $(p, \boldsymbol{x})$ evolves to the next configuration $(q, \boldsymbol{y})$ in one step under P:

- $I_p = \texttt{inc r k}$:
  $q = k$ and $\boldsymbol{y} = \boldsymbol{x}[x_r \mapsto x_r + 1]$

- $I_p = \texttt{dec r k l}$:
  $x_r > 0$, $q = k$ and $\boldsymbol{y} = \boldsymbol{x}[x_r \mapsto x_r - 1]$ or
  $x_r = 0$, $q = l$ and $\boldsymbol{y} = \boldsymbol{x}$

Notation: $(p, \boldsymbol{x}) \left|\frac{1}{P}\right. (q, \boldsymbol{y})$.

Note that if $(p, \boldsymbol{x})$ is halting (i.e. $I_p = \texttt{halt}$) there is no next configuration. Ditto for $p \geq \ell$, the length of the program.

Whole Computation

Define

$$(p, \boldsymbol{x}) \left|\frac{0}{P}\right. (q, \boldsymbol{y}) :\Leftrightarrow (p, \boldsymbol{x}) = (q, \boldsymbol{y})$$

$$(p, \boldsymbol{x}) \left|\frac{t}{P}\right. (q, \boldsymbol{y}) :\Leftrightarrow \exists\, q', \boldsymbol{y}' \ \left((p, \boldsymbol{x}) \left|\frac{t-1}{P}\right. (q', \boldsymbol{y}') \left|\frac{1}{P}\right. (q, \boldsymbol{y}) \right)$$

$$(p, \boldsymbol{x}) \left|\frac{}{P}\right. (q, \boldsymbol{y}) :\Leftrightarrow \exists\, t \ \left((p, \boldsymbol{x}) \left|\frac{t}{P}\right. (q, \boldsymbol{y}) \right)$$

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

Finite versus Infinite

Note that a computation may well be infinite: the program

```
0:  inc 0 0
```

has no terminating computations at all. More generally, for some particular input a computation on a machine may be finite, and infinite for other inputs.

Also, computations may get stuck. The program

```
0: inc 0 1
```

cannot execute the first instruction since there is no goto label 1.

Again, we have two kinds of computations: finite ones (that necessarily end in a halt instruction), and infinite ones. We will write

$$(C_i)_{i<n} \qquad \text{and} \qquad (C_i)_{i<\omega}$$

for finite versus infinite computations.

Here ω denotes the first infinite ordinal. If you don't like ordinals, replace ω by some meaningless but pretty symbol like ∞.

Computing a Function

Suppose P is an RMP of length ℓ where and $I_{\ell-1} = \texttt{halt}$. The initial configuration for input $\boldsymbol{a} \in \mathbb{N}^k$ is $E_{\boldsymbol{a}} = (0, (0, \boldsymbol{a}, \boldsymbol{0}))$. So the input is in registers $R_1, \ldots, R_k$, all others are zero; the initially state is 0.

Definition

A RMP P computes the partial function $f : \mathbb{N}^k \nrightarrow \mathbb{N}$ if for all $\boldsymbol{a} \in \mathbb{N}^k$:

- If f is defined on $\boldsymbol{a}$, then the computation of P on $C_0 = E_{\boldsymbol{a}}$ terminates in configuration $C_n = (\ell-1, (b, \boldsymbol{y}))$ where $f(\boldsymbol{a}) = b$.
- Otherwise, the computation of P on $E_{\boldsymbol{a}}$ fails to terminate.

Terminology

Since all the standard models of computation produce the same clone of functions one simply speaks about computable functions (unless there is a reason to point to some particular model).

Traditionally, computable functions are called

- Recursive functions
  computable functions that are total
- Partial recursive functions
  computable functions that may be partial

It's A Clone

Clearly we can generalize the notion of a clone from total functions to partial ones.

Proposition

Register machines computable functions form a clone, containing the clone of primitive recursive functions.

Exercise

Figure out the details.

Cleaning Up

We may safely assume that $P = I_0, I_1, \ldots, I_{\ell-1}$ uses only registers R_i, $i < \ell$. Similarly, we may assume that all the goto targets k lie in the range $0 \leq k < \ell$. Hence all numbers in the instructions are bounded by ℓ. Lastly, $I_{\ell-1}$ is a halt instruction, and there are no others.

We will call these programs regular register machines programs (RRMP).

It follows that RRMPs cannot get stuck, every computation either ends in halting, or is infinite. From now on, we will only deal with RRMs.

Exercise

Write a program transformer that converts an arbitrary RMP into an "equivalent" RRMP.

Named Registers

To make RMPs slightly easier to read we use names such as X, Y, Z and so forth for the registers.

This is just a bit of syntactic sugar, if you like you can always replace X by R_0, Y by R_1 and so forth (at least if you ignore our I/O conventions).

And we will be quite relaxed about distinguishing register X from its content $[X]$.

Example: Multiplication

Here is a program that multiplies registers X and Y, and places the product into Z. U is auxiliary.

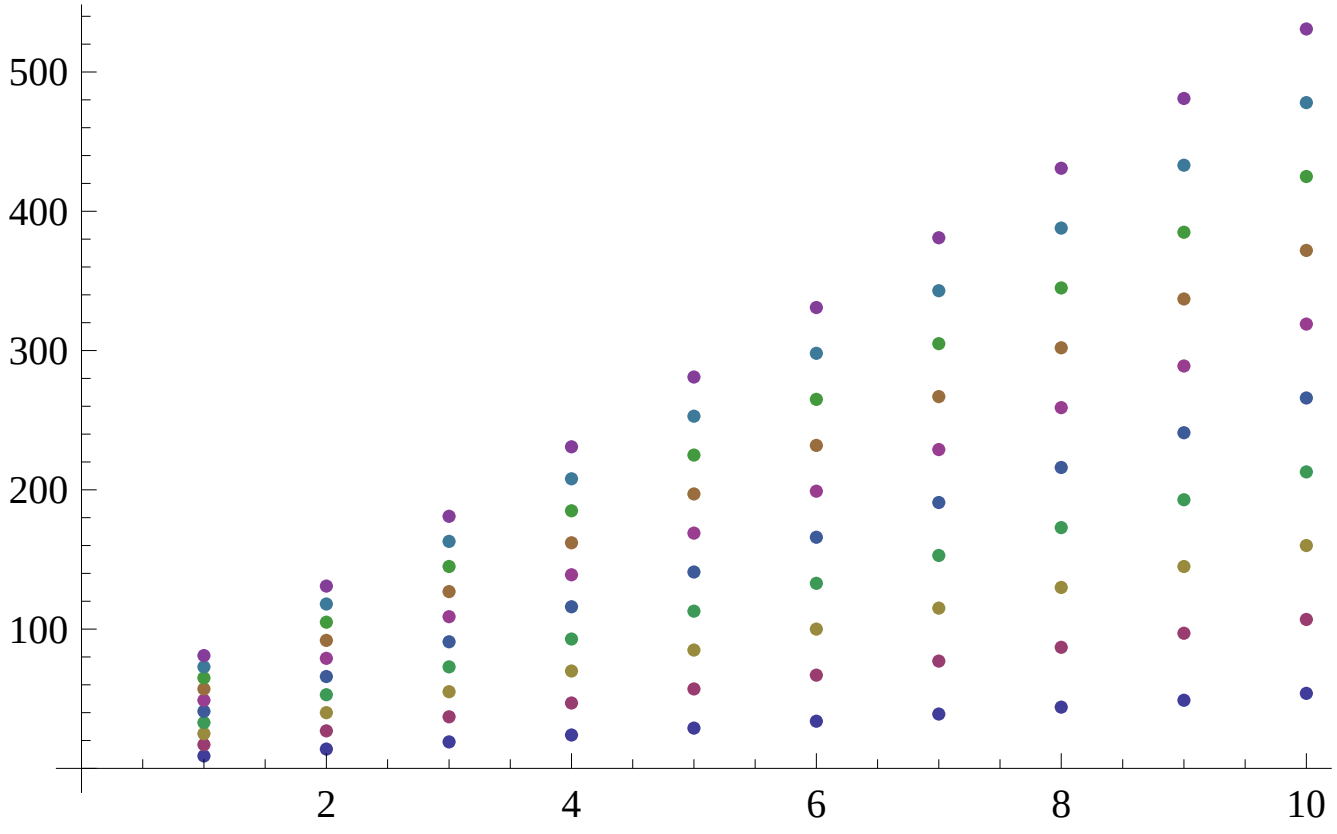
```
// multiplication   X Y --> Z
  0:   dec X  1  6
  1:   dec Y  2  4
  2:   inc Z  3
  3:   inc U  1
  4:   dec U  5  0
  5:   inc Y  4
  6:   halt
```

A Computation

0	$(2,2,0,0)$	1	$(0,2,2,0)$
1	$(1,2,0,0)$	2	$(0,1,2,0)$
2	$(1,1,0,0)$	3	$(0,1,3,0)$
3	$(1,1,1,0)$	1	$(0,1,3,1)$
1	$(1,1,1,1)$	2	$(0,0,3,1)$
2	$(1,0,1,1)$	3	$(0,0,4,1)$
3	$(1,0,2,1)$	1	$(0,0,4,2)$
1	$(1,0,2,2)$	4	$(0,0,4,2)$
4	$(1,0,2,2)$	5	$(0,0,4,1)$
5	$(1,0,2,1)$	4	$(0,1,4,1)$
4	$(1,1,2,1)$	5	$(0,1,4,0)$
5	$(1,1,2,0)$	4	$(0,2,4,0)$
4	$(1,2,2,0)$	0	$(0,2,4,0)$
0	$(1,2,2,0)$	6	$(0,2,4,0)$

```
// multiplication   X Y --> Z
  0:   dec X  1  6
  1:   dec Y  2  4
  2:   inc Z  3
  3:   inc U  1
  4:   dec U  5  0
  5:   inc Y  4
  6:   halt
```

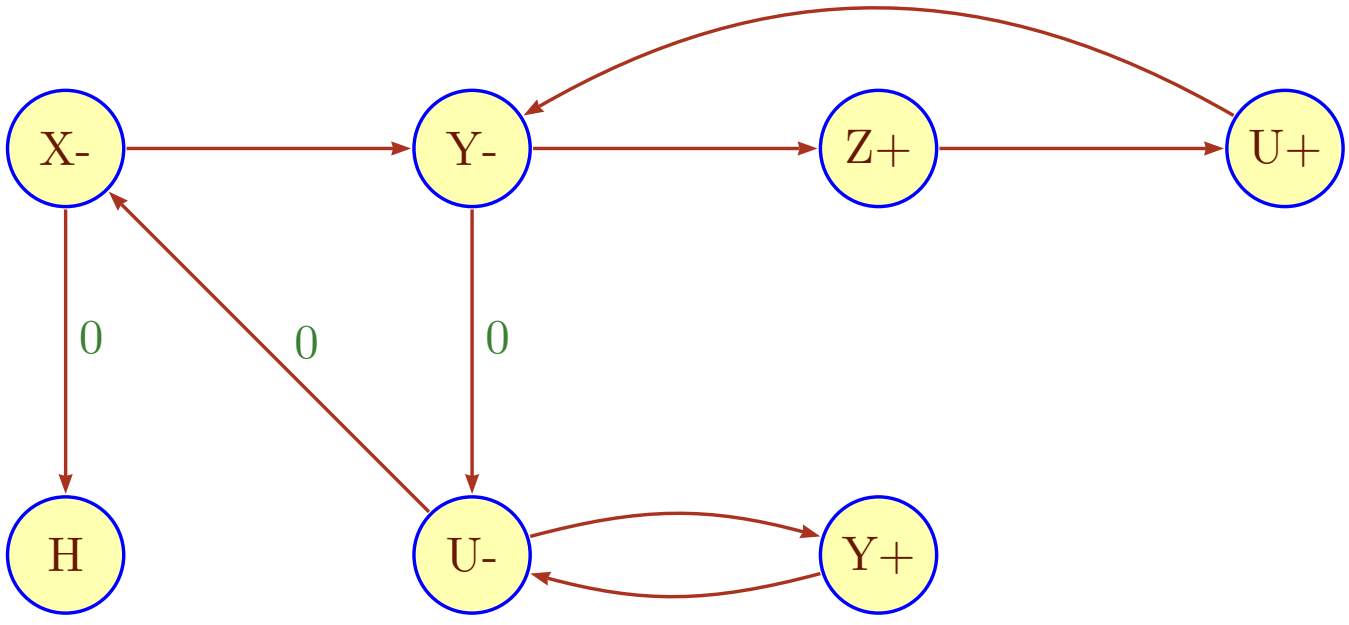
Time Complexity?



Exercise

Determine the time complexity of the multiplication RM.

Flowgraph for Multiplication

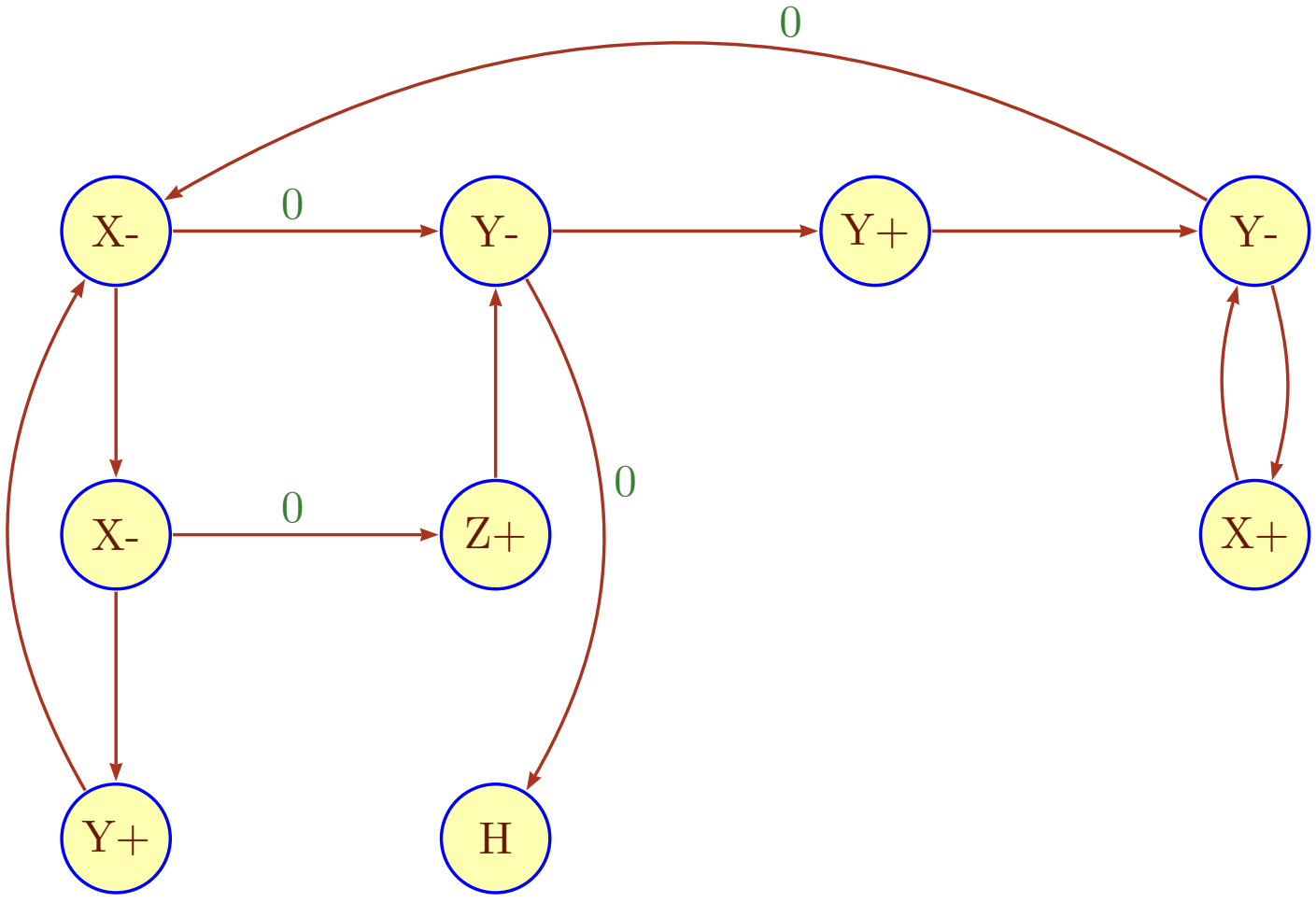


Example: Binary Digit Sum

The following RRMP computes the number of 1's in the binary expansion of X, the so-called binary digit sum of x.

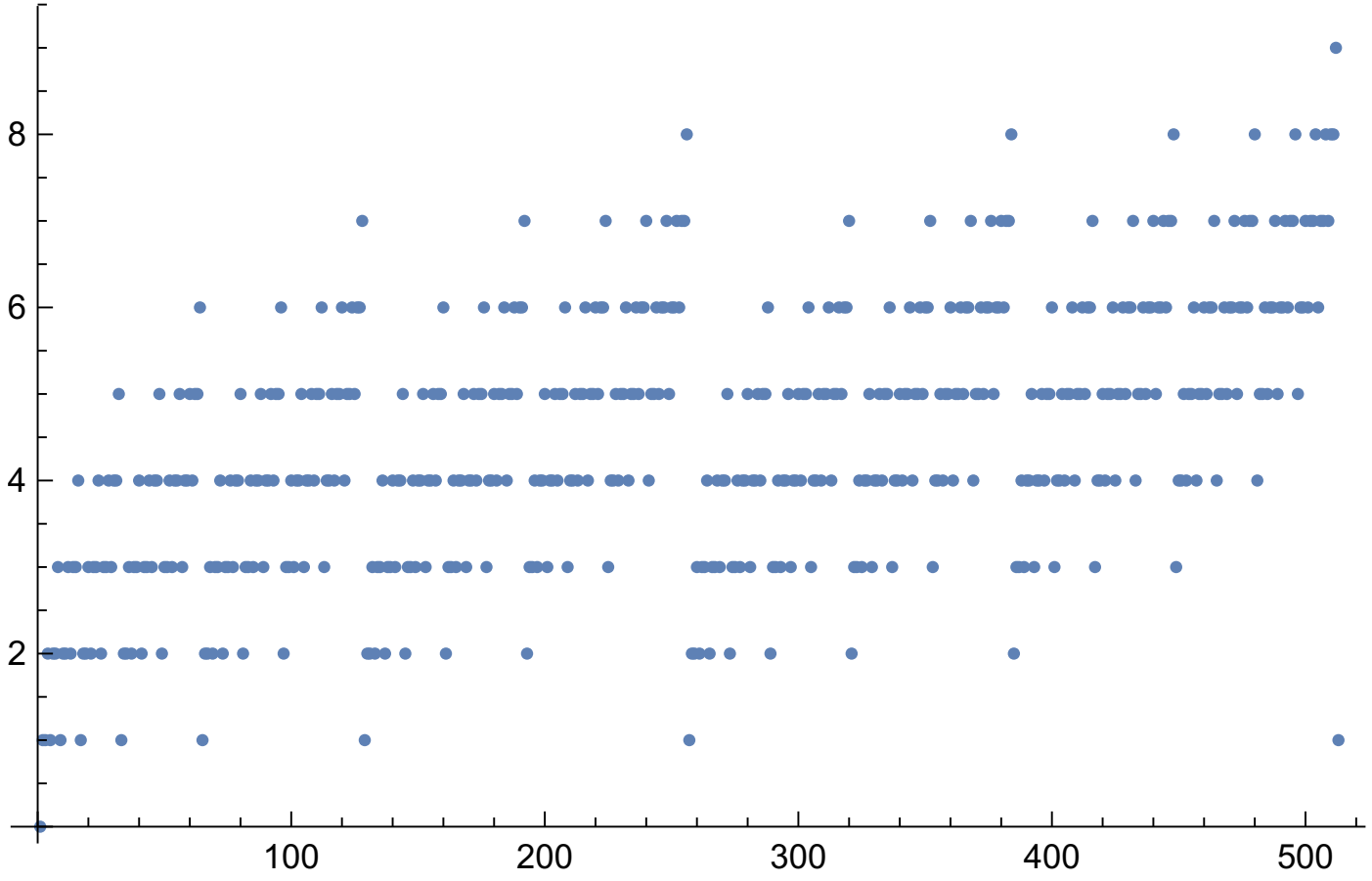
```
// binary digitsum of X --> Z
  0:   dec X  1  4
  1:   dec X  2  3
  2:   inc Y  0
  3:   inc Z  4
  4:   dec Y  5  8
  5:   inc Y  6
  6:   dec Y  7  0
  7:   inc X  6
  8:   halt
```

Flowgraph for DigitSum



Digit Sum

The (binary) digit sum is actually quite useful in some combinatorial arguments.



Exercises

Exercise

Show that every primitive recursive function can be computed by a register machine.

Exercise

Implement a primitive recursive to RM compiler.
How hard would it be to optimize the RM?

Exercise

Suppose some register machine M computes a total function f.
Why can we not conclude that f is primitive recursive?

Coding

To translate finite structures into (Gödel-) numbers, we need a coding system, consisting of three functions (see Coding for details).

$$\langle \ldots \rangle : \mathbb{N}^\star \to \mathbb{N}$$

$$\mathsf{dec} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$\mathsf{len} : \mathbb{N} \to \mathbb{N}$$

Here $\langle \ldots \rangle$ is multiadic and thus cannot be primitive recursive, but dec and len are typically primitive recursive (actually, even more basic than that).

Sequence Numbers

Write $\mathbb{N}^\star$ for the set of all finite sequences of natural numbers and nil for the empty sequence.

We want to express a sequence $a_0, a_1, \ldots, a_{n-1} \in \mathbb{N}^\star$ as a single sequence number $b = \langle a_0, a_1, \ldots, a_{n-1} \rangle$.

To decode, we need

$$\mathsf{len}(b) = n$$

$$\mathsf{dec}(b, i) = a_i \qquad 0 \leq i < n$$

Even/Odd Coding

A very natural system can be built around the even/odd pairing function

$$\pi(x, y) = 2^x(2y + 1)$$

For example

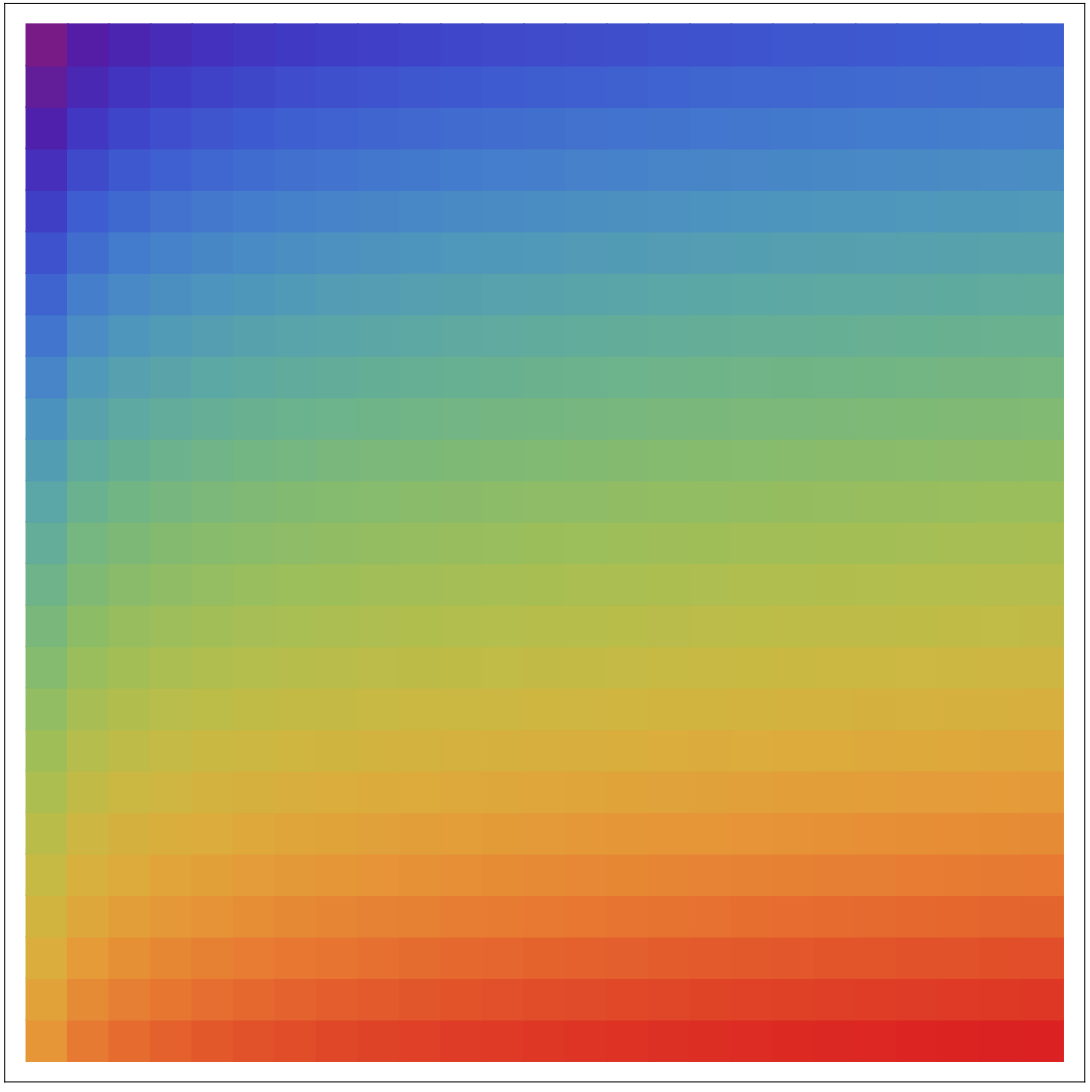
$$\pi(5, 27) = 32 \cdot 55 = 1760 = 110111\,00000_2$$

In general, the binary expansion of $\pi(x, y)$ looks like so:

$$y_k y_{k-1} \ldots y_0\, 1\, \underbrace{00\ldots0}_{x}$$

where $y_k y_{k-1} \ldots y_0$ is the standard binary expansion of y (y_k is the most significant digit).

Picture (log plot)



A Bijection

The range of π is $\mathbb{N}_+$, so we don't have a bijection. As it turns out, we can exploit this produce a rather elegant coding function:

$$\langle \mathsf{nil} \rangle := 0$$

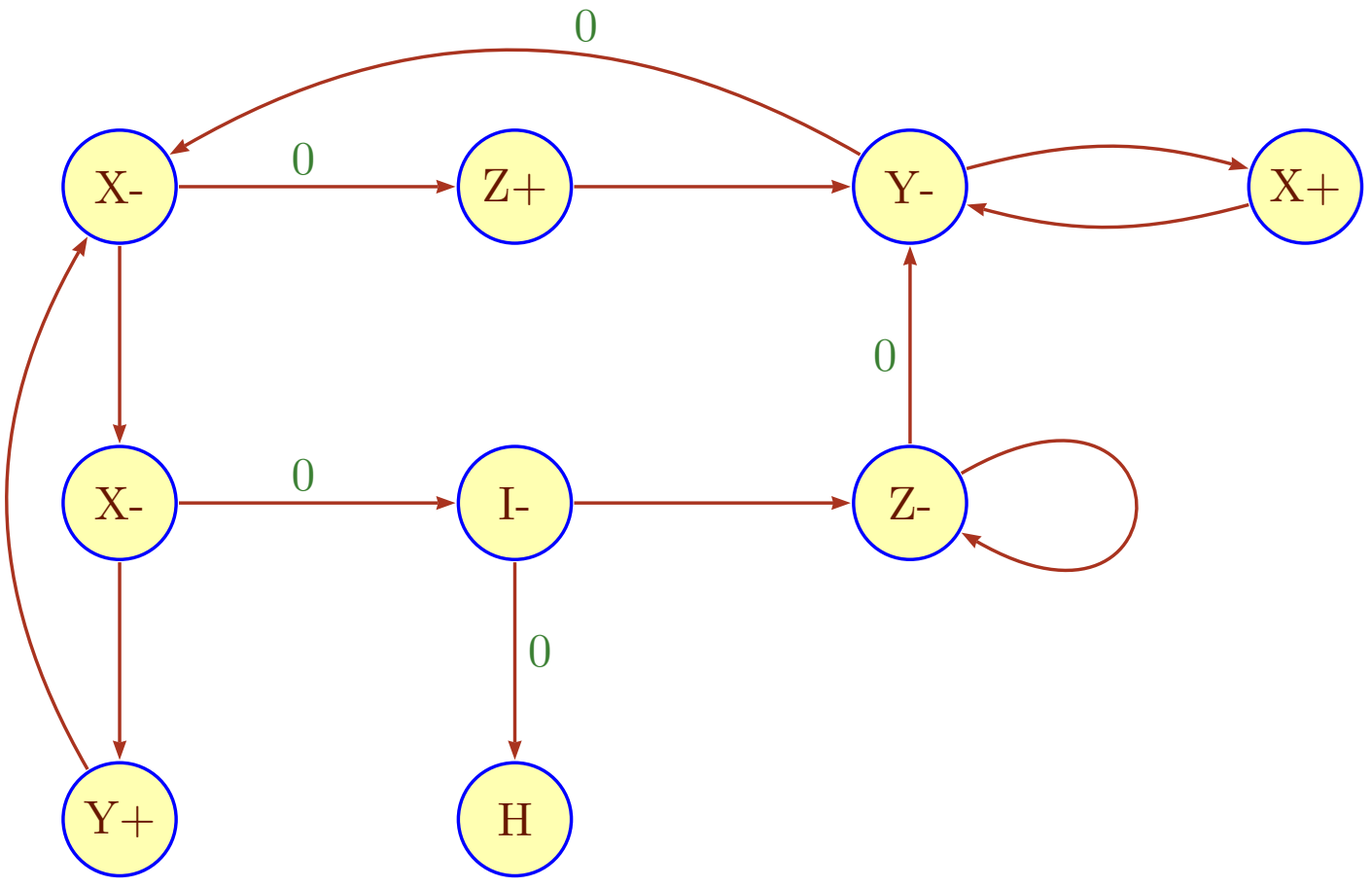
$$\langle a_1, \ldots, a_n \rangle := \pi(a_1, \langle a_2, \ldots, a_n \rangle)$$

Informally, it is easy to see that this coding function is indeed a bijection between $\mathbb{N}^\star$ and $\mathbb{N}$. The sequence numbers have a very simply structure in binary:

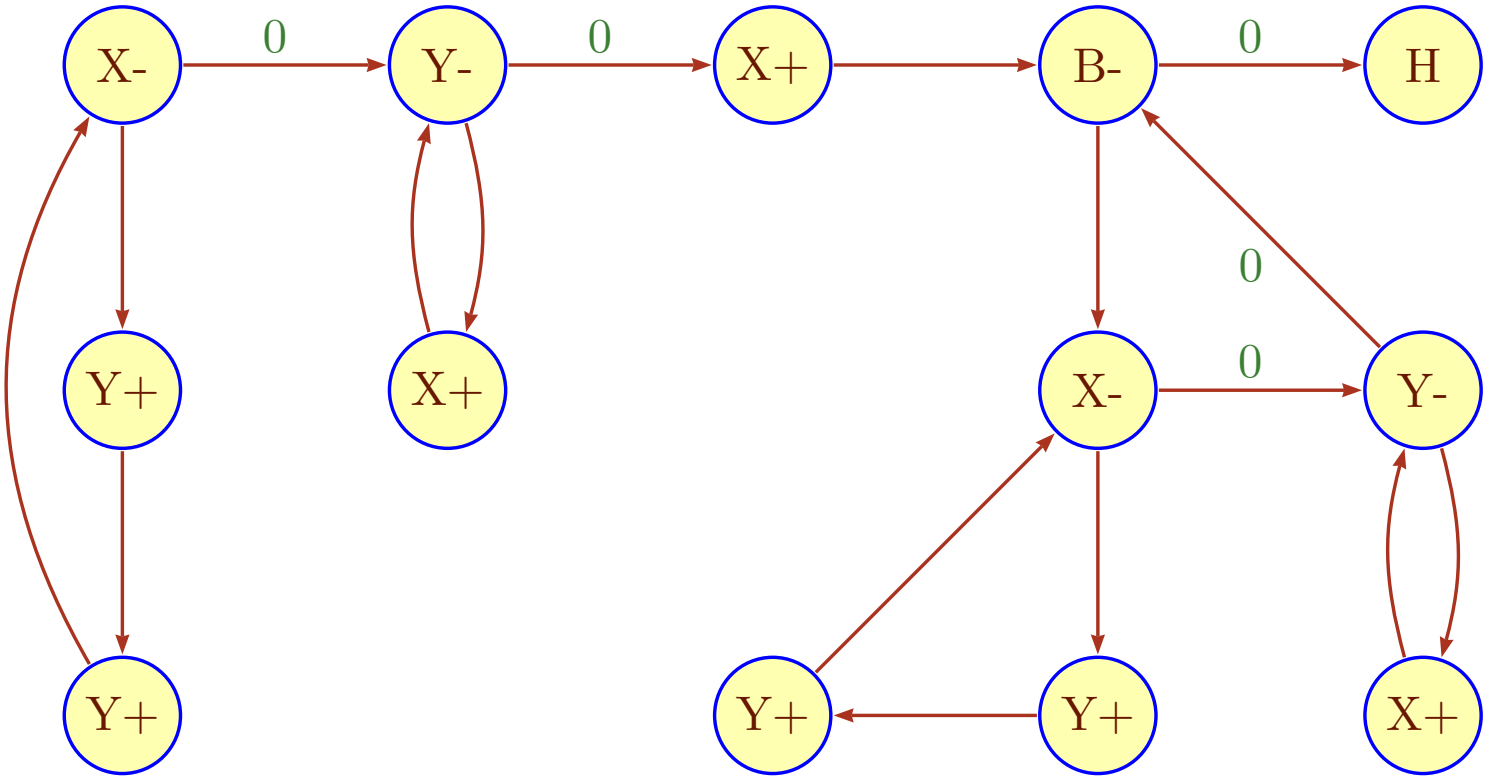
$$\begin{aligned} \langle 2, 3, 5, 1 \rangle &= 20548 \\ &= 1\underbrace{0}_{1}1\underbrace{00000}_{5}1\underbrace{000}_{3}1\underbrace{00}_{2} \end{aligned}$$

This makes it relatively easy to compute the decoding function $\mathrm{dec}(x, i)$.

Flowgraph dec(x, i)



Prepend b to x



Self-Reference

As Gödel has shown devastatingly in his incompleteness theorem, self-reference is an amazingly powerful tool.

On occasion, it wreaks plain havoc: the incompleteness theorem takes a wrecking ball to Hilbert's beautiful program (at least in its original form).

However, in the context of computation, self-reference turns into a genuine resource. We developed our coding machinery to show that standard discrete structures can be expressed as natural numbers and thus be used in an RPM. But an RPM is itself a discrete structure, so RPMs can compute with (representations of) RPMs.

This leads to the fundamental concept of **universality**.

Coding RMPs

A single instruction of an RMP can easily be coded as a sequence number:

- `halt` $\langle 0 \rangle$
- `inc r k` $\langle r, k \rangle$
- `dec r k l` $\langle r, k, l \rangle$

And a whole program can be coded as the sequence number of these numbers.

Example: Addition

For example, the simplified addition program

```
// addition   R0 + R1 --> R1
  0:   dec 0  1  2
  1:   inc 1  0
  2:   halt
```

has code number

$$\langle\langle 0, 1, 2\rangle, \langle 1, 0\rangle, \langle 0\rangle\rangle = 880983691755 52.$$

Note that this code number does not include I/O conventions, but it is not hard to tack these on if need be.

Coding RRMPs

One of the reasons RRMPs are slightly more useful than arbitrary RMPs is that the code number of a RRMP of length ℓ is bounded by a primitive recursive function of ℓ:

$$\langle P \rangle \leq f(\ell)$$

This is a nice property to have, e.g. when conducting searches over programs.

This fails completely and unnecessarily for plain RMPs.

Why?

Our coding function has the property that for $a_i \leq b_i$

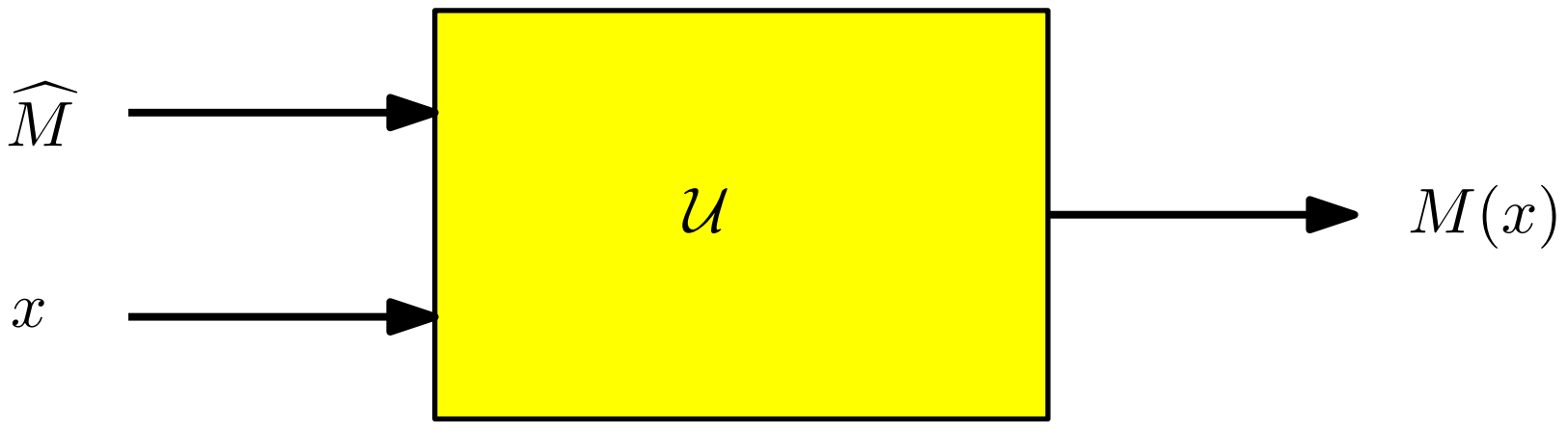
$$\langle a_1, a_2, \ldots, a_k \rangle \leq \langle b_1, b_2, \ldots, b_k \rangle$$

This is clear from the representation in binary.

Note that $\langle a, b, c \rangle = 2^a + 2^{a+b+1} + 2^{a+b+c+2}$, so $\langle a, a, a \rangle < 2^{3a+3}$. But then, for any RRMP of length ℓ, we have

$$\begin{aligned} \langle P \rangle &< \langle \langle \ell{-}1, \ell{-}1, \ell{-}1 \rangle, \ldots, \langle \ell{-}1, \ell{-}1, \ell{-}1 \rangle \rangle \\ &< \langle 2^{3\ell}, \ldots, 2^{3\ell} \rangle \\ &< 2^{2^{3\ell}} \end{aligned}$$

> This special property of digital computers, that they can mimic any discrete state machine, is described by saying that they are universal machines. The existence of machines with this property has the important consequence that, considerations of speed apart, it is unnecessary to design various machines to do various computing processes. They can all be done with one digital computer, suitably programmed for each case. It will be seen that as a consequence of this all digital computers are in a sense equivalent.
>
> Alan Turing (1950)


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

Turing 1936

Computational universality was established by Turing in 1936 as a purely theoretical concept.

Surprisingly, within just a few years, practical universal computers (at least in principle) were actually built and used:

1941 Konrad Zuse, Z3

1943 Tommy Flowers, Colossus

1944 Howard Aiken, Mark I

1946 Prosper Eckert and John Mauchley, ENIAC

Falling off a Cliff

Let's define the state complexity of a RMP to be its length, the number of instructions used in the program.

An RMP of complexity 1 is pretty boring, 2 is slightly better, 3 better yet; a dozen already produces some useful functions. With 1000 states we can do even more, let alone with 1000000, and so on.

Except that the "so on" is plain wrong: there is some magic number N such that every RMP can already by simulated by a RMP of state complexity just N: we can hide the complexity of the computation in one of the inputs. As far as state complexity is concerned, maximum power is already reached at N.

This is counterintuitive, to say the least.

How does one construct a universal computer? According to the last section, we can code a RMP $P = I_0, I_1, \ldots, I_{\ell-1}$ as an integer e, usually called an index for P in this context.

Moreover, we can access the instructions in the program by performing a bit of arithmetic on the index. Note that we can do this non-destructively by making copies of the original values.

So, if index e and some line number p (for program counter) are stored in registers we can retrieve instruction I_p and place it into register I.

Simulating a RM

Suppose we are given a sequence number e that is an index for some RMP P requiring one input x.

We claim that there is a universal register machine (URM) $\mathcal{U}$ that, on input e and x, simulates program P on x.

Alas, writing out $\mathcal{U}$ as a pure RMP is too messy, we need to use a few "macros" that shorten the program.

Of course, one has to check that all the macros can be removed and replaced by corresponding RMPs, but that is not very hard.

- `copy r s k`
  Non-destructively copy the contents of R_r to R_s, goto k.
- `zero r k l`
  Test if the content of R_r is 0; if so, goto k, otherwise goto l.
- `pop r s k`
  Interpret R_r as a sequence number $a = \langle b, c \rangle$; place b into R_s and c into R_r, goto k. If $R_r = 0$ both registers will be set to 0.
- `read r t s k`
  Interpret R_r as a sequence number and place the R_tth component into R_s, goto k. Halt if R_t is out of bounds.
- `write r t s k`
  Interpret R_r as a sequence number and replace the R_tth component by R_s, goto k. Halt if R_t is out of bounds.

The Pieces

Here are the registers used in $\mathcal{U}$:

- **x** input for the simulated program P
- **E** code number of P
- **R** register that simulates the registers of P
- **I** register for instructions of P
- **p** program counter

Hack: x is also used as an auxiliary variable to keep the whole program small.

Universal RM


```
 0:   copy   E  R  1          // R = E
 1:   write  R  p  x  2       // R[0] = x
 2:   read   E  p  I  3       // I = E[p]
 3:   pop    I  r  4          // r = pop(I)
 4:   zero   I 13  5          // if I was halt
 5:   pop    I  p  6          // p = pop(I)
 6:   read   R  r  x  7       // x = R[r]
 7:   zero   I  8  9          // check if I was inc/dec
 8:   inc    x 12             // x++; goto 12
 9:   zero   x 10 11          // if( x != 0 ) goto 11
10:   pop    I  p  2          // p = pop(I)
11:   dec    x 12 12          // x--; goto 12
12:   write  R  r  x  2       // R[r] = x; goto 2
13:   halt
```

WTF?


```
0:   copy   E  R  1          // R = E
1:   write  R  p  x  2       // R[0] = x
```

Let P be the program we want to simulate, ℓ its length and E its code.

$R = E$: this makes sure that R contains a sequence number of length ℓ.

Hence there is enough space to store all the registers of P.

It is now safe to place argument x into R_0 of P (p is initially 0).

```
2:   read   E  p  I  3        // I = E[p]
3:   pop    I  r  4          // r = pop(I)
4:   zero   I 13  5          // if I was halt
```

Fetch instruction I_p from E (p is initially 0), place into register I.

Pop the first component of sequence number I.

If I is now 0, we have found the halt instruction in P and halt, too.

```
5:   pop    I  p  6          // p = pop(I)
6:   read   R  r  x  7        // x = R[r]
7:   zero   I  8  9          // check if I was inc/dec
```

Otherwise, pop the next component of sequence number I and store the goto label in p.

In all cases, this is the number of the P register to work on, so we are writing the argument x into R_r.

If I is now 0, we have an increment instruction, otherwise a decrement; branch correspondingly.

```
 8:   inc    x 12              // x++; goto 12
 9:   zero   x 10 11           // if( x != 0 ) goto 11
10:   pop    I  p  2           // p = pop(I)
11:   dec    x 12 12           // x--; goto 12
12:   write  R  r  x  2        // R[r] = x; goto 2
```

In the increment case, increment x and write the result into R in position r; go back to the main loop in line 2.

Otherwise we have a decrement instruction and we check whether the register is 0. If so, we pop the second goto label, store it in p and continue with the main loop.

If the register is positive, we perform a decrement, write the result into R and go back to the main loop.

Size?

Of course, the 13 lines in this universal machine are a bit fraudulent, we really should expand all the macros. Still, the resulting honest register machine would not be terribly large.

And there are lots of ways to optimize.

Exercise

Give a reasonable bound for the size of the register machine obtained by expanding all macros.

Exercise

Try to build a smaller universal register machine.

Halting

If we define computability in terms of RMs, it follows that the Halting Problem for RMs is undecidable: there is no RM that takes an index e as input and determines whether the corresponding RM P_e halts (on all-zero registers).

Since RMs are perfectly general computational devices, this means that there is no algorithm to determine whether RM P_e halts; the Halting Problem is undecidable.

It should be intuitively clear that our universal machine, while easy to understand, is by no means optimal; there should be ways to build a URM with fewer instructions.

One problem in the study of small URMs is that there are a number of reasonable possible choices for the basic instruction set. Each choice is associated with a minimal URM, so the search is a bit messy. For our machines, we have the following result.

Theorem (Korec 1996)

There is a universal register machine with 22 instructions.

Take a look at Korec for the details of the construction.

And Turing Machines?

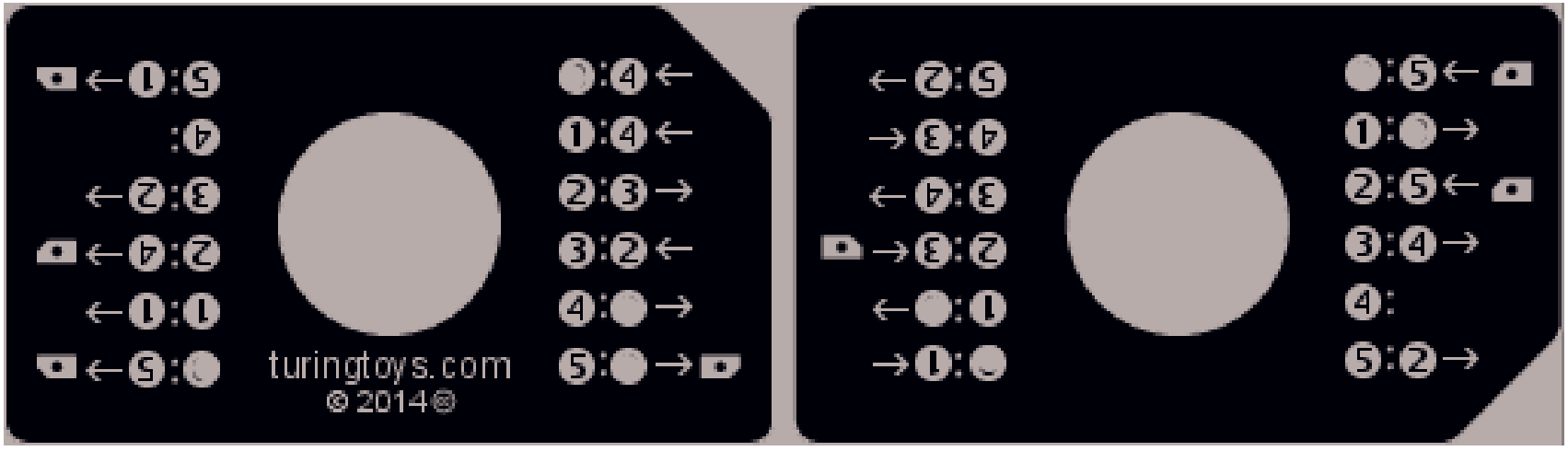
Define a (n, k)-Turing machine to be a TM that has n states and a tape alphabet of size k.

Here the instruction set is clear (except perhaps for head movement), but there are two parameters to work with. Intuitively, it should be clear that the smaller k is, the larger n, and vice versa.

Here are some combinations of n and k where universal machines are known to exist.

$$(24, 2), (10, 3), (7, 4), (5, 5), (4, 6), (3, 10), (2, 18), (2, 5)$$

A Universal Turing Machine



Exercise

Figure out what this picture means.

Exercise (Very Hard)

Prove that this is really a universal Turing machine.

No Coding

One very pleasant feature of register machines is that the do not require any input/output coding for arithmetic functions.

In general this is emphatically not the case. In particular Turing machines naturally operate on strings, so numbers have to be coded (say, using binary notation).

Things get worse if on looks at more exotic models of computation such as the λ-calculus or physics-like models such as cellular automata. The latter in particular tend to produce minor headaches when it comes to I/O conventions.

Who Cares?

By constructing more RMs, one can try to convince oneself that any "intuitively computable" function is already RM-computable. So the universal RM can compute all computable functions.

Or, if one prefers Turing machines, one can show that an arithmetic function is RM-computable iff it is TM-computable. Or λ-computable, or μ-computable, and so on and so forth.

For discrete computation, there is only one model (as opposed to computation on the reals).