CDM

Register Machines

Klaus Sutner

Carnegie Mellon University

Spring 2021

Where Are We?

Recall: we are looking for a formal definition of computability. So far we have seen primitive recursive functions, a huge collection of intuitively computable functions. So could this be the final answer? Sadly, no. Here are the main obstructions:

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 functions have stupendous growth rates, faster than primitive recursive ones.

Ackermann's Function (1928)

We have seen that course-of-value recursion does not push us out of the realm of primitive recursive functions. Similarly, double recursion won't hurt (try it). And yet, there is more to recursion than this. Here is a classical example: the Ackermann function $A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by double recursion. We write x^+ instead of $x + 1$.

$$A(0, y) = y^+$$
$$A(x^+, 0) = A(x, 1)$$
$$A(x^+, y^+) = A(x, A(x^+, y))$$

On the surface, this looks more complicated than primitive recursion. Of course, it is not at all clear that Ackermann could not somehow be turned into a p.r. function.

Family Perspective–Currying

It is useful to think of Ackermann's function as a family of unary functions $(A_x)_{x \geq 0}$ where $A_x(y) = A(x, y)$ ("level x of the Ackermann hierarchy").

The definition then looks like so:

$$A_0(y) = y^+$$

$$A_{x^+}(0) = A_x(1)$$
$$A_{x^+}(y^+) = A_x(A_{x^+}(y))$$

From this it follows easily by induction that

Lemma

Each of the functions A_x is primitive recursive (and hence total).

The Bottom Hierarchy

$$A(0, y) = y^+$$
$$A(1, y) = y^{++}$$
$$A(2, y) = 2y + 3$$
$$A(3, y) = 2^{y+3} - 3$$
$$A(4, y) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} - 3$$

The first 4 levels of the Ackermann hierarchy are easy to understand, though A_4 starts causing problems: the stack of 2's in the exponentiation has height $y + 3$.

Tetration

A_4 is usually called super-exponentiation or tetration and often written ${}^n a$ or $a\uparrow\uparrow n$.

$$a\uparrow\uparrow n = \begin{cases} 1 & \text{if } n = 0, \\ a^{a\uparrow\uparrow(n-1)} & \text{otherwise.} \end{cases}$$

For example,

$$A(4,3) = 2\uparrow\uparrow 6 - 3 = 2^{2^{65536}} - 3$$

an insanely large number.

The Mystery of $A(6,6)$

Alas, if we continue just a few more levels, darkness befalls.

$A(5,y) \approx$ super-super exponentiation

$A(6,y) \approx$ an unspeakable horror

$A(7,y) \approx$ speechlessness

For level 5, one can get some vague understanding of iterated super-exponentiation, $A(5,y) = (\lambda z.{}^{y+3}z)^{y+3}(1) - 3$ but things start to get murky.

At level 6, we iterate over the already nebulous level 5 function, and things really start to fall apart.

At level 7, Wittgenstein comes to mind: "Wovon man nicht sprechen kann, darüber muss man schweigen."[1]

[1]Whereof one cannot speak, thereof one must be silent. *Tractatus Logico-Philosophicus*

Ackermann vs. PR

Theorem

The Ackermann function dominates every primitive recursive function f in the sense that there is a k such that

$$f(\boldsymbol{x}) < A(k, \max \boldsymbol{x}).$$

Hence A is not primitive recursive.

Sketch of proof.

Since we are dealing with a rectype, we can argue by induction on the buildup of f.

The atomic functions are easy to deal with.

The interesting part is to show that the property is preserved during an application of composition and of primitive recursion. Alas, the details are rather tedious.

□

Ackermann and Union/Find

One might think that the only purpose of the Ackermann function is to refute the claim that computable is the same as p.r. Surprisingly, the function pops up in the analysis of the Union/Find algorithm (with ranking and path compression).

The running time of Union/Find differs from linear only by a minuscule amount, which is something like the inverse of the Ackermann function.

But in general anything beyond level 3.5 of the Ackermann hierarchy is irrelevant for practical computation.

Exercise

Read an algorithms text that analyzes the run time of the Union/Find method.

But Is It Computable?

Here is an entirely heuristic argument: we can write a tiny bit of C code that implements the Ackermann function (assuming that we have infinite precision integers).

```
int acker(int x, int y)
{
  return( x ? (acker(x-1, y ? acker(x, y-1) : 1)) : y+1 );
}
```

All the work of organizing the nested recursion is easily handled by the compiler and the execution stack. So this provides overwhelming evidence that the Ackermann function is intuitively computable.

Proofs by Hashing

We could memoize the values that are computed during a call to $A(a,b)$: build a hash table H such that $H[x,y] = z$ whenever an intermediate result $A(x,y) = z$ is discovered during the computation.

In practice, this helps in computing a few small values of A, but does not go very far.

More interesting is the following: suppose we call $A(a,b)$ and obtain result c, producing a hash table H as a side effect.

Claim: H provides a proof that $A(a,b) = c$.

Not a proof in the classical sense, but an object that makes it possible to perform a simple coherence check and conclude that the value c is indeed correct.

Checking

We have to check the following properties everywhere in H:

$$\begin{aligned} H[0,y] = z \quad &\text{implies} \quad z = y+1 \\ H[x^+,0] = z \quad &\text{implies} \quad H[x,1] = z \\ H[x^+,y^+] = z \quad &\text{implies} \quad H[x,z'] = z \text{ where } z' = H[x^+,y] \end{aligned}$$

This comes down to performing $O(N)$ table lookups where N is the size of H.

Once the table is verified, we check $H[a,b] = c$. Done.

PR versus Computable

Obvious Question: how much do we have to add to primitive recursion to capture the Ackermann function?

As it turns out, we need just one modification: we have to allow unbounded search: a type of search where the property we are looking for is still primitive recursive, but we don't know ahead of time how far we have to go.

Unbounded Search vs. Ackermann

Proposition

There is a primitive recursive relation R such that

$$A(a,b) = \big(\min\big(\, z \mid R(a,b,z)\,\big)\big)_0$$

Recall that $(s)_0$ is just the first element of a coded sequence s.

Sketch of proof. Think of z as a pair $\langle c,h\rangle$ where h encodes the hash table H from above, and $c = H[a,b]$.

R performs the coherence test described above and is clearly primitive recursive. □

Here is a more direct, computational description.

Ackermann and Lists

The computation of, say, $A(2,1)$ can be handled in a very systematic fashion: always unfold the rightmost subexpression.

$$A(2,1) = A(1,A(2,0)) = A(1,A(1,1)) = A(1,A(0,A(1,0))) = \dots$$

Note that the A's and parens are just syntactic sugar, a better description would be

$$2,1 \rightsquigarrow 1,2,0 \rightsquigarrow 1,1,1 \rightsquigarrow 1,0,1,0 \rightsquigarrow 1,0,0,1 \rightsquigarrow 1,0,2 \rightsquigarrow 1,3 \rightsquigarrow 0,1,2$$
$$\rightsquigarrow 0,0,1,1 \rightsquigarrow 0,0,0,1,0 \rightsquigarrow 0,0,0,0,1 \rightsquigarrow 0,0,0,2 \rightsquigarrow 0,0,3 \rightsquigarrow 0,4 \rightsquigarrow 5$$

We can model these steps by a list function Δ defined on sequences of naturals (or, we could use a stack).

List Operation

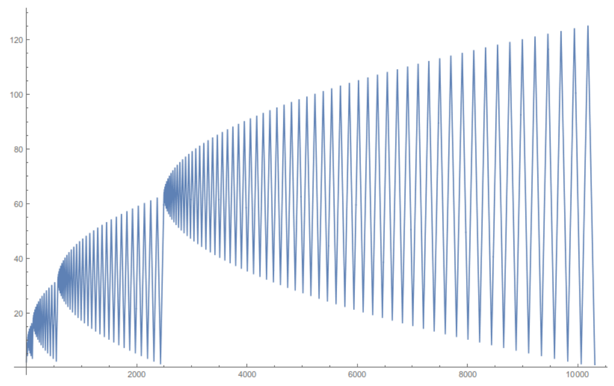
$$\Delta(\dots,0,y) = (\dots,y^+)$$
$$\Delta(\dots,x^+,0) = (\dots,x,1)$$
$$\Delta(\dots,x^+,y^+) = (\dots,x,x^+,y)$$

Since A is total, there is some time t for any a and b such that

$$\Delta^t(a,b) = (c)$$

Clearly this condition is primitive recursive in (a,b,c,t).

$A(3,4)$



The computation takes 10307 steps, the plot shows the lengths of the list.

Evaluation

Very rapidly growing functions such as the Ackermann function are one reason primitive recursion is not strong enough to capture computability. Here is another obstruction: we really need to deal with partial functions.

Recall the evaluation operator for our PR terms:

$$\text{eval}(\tau, \boldsymbol{x}) = \text{ value of } \tau^\star \text{ on input } \boldsymbol{x}$$

It is clear that eval is intuitively computable (take a compilers course). In fact, it is not hard to implement in eval in any modern programming language.

Question: Could eval be primitive recursive?

A useless answer would be to say **no**, the types don't match.

Indices

The first argument of eval is a term τ in our PR language, so our first step will be to replace τ by an index $\widehat{\tau} \in \mathbb{N}$.

The index $\widehat{\tau}$ will be constructed in a way that makes sure that all the operations we need on indices are clearly primitive recursive.

The argument vector $\boldsymbol{x} \in \mathbb{N}^n$ will also be replaced by its sequence number $\langle x_1, \ldots, x_n \rangle$. Hence we will be able to interpret eval as a function of type

$$\mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

and this function might potentially be primitive recursive.

Coding PR

Here is one natural way of coding primitive recursive terms as naturals:

term	code
0	$\langle 0, 0 \rangle$
P^n_i	$\langle 1, n, i \rangle$
S	$\langle 2, 1 \rangle$
$\text{Prec}[h, g]$	$\langle 3, n, \widehat{h}, \widehat{g} \rangle$
$\text{Comp}[h, g_1, \ldots, g_n]$	$\langle 4, m, \widehat{h}, \widehat{g_1}, \ldots, \widehat{g_n} \rangle$

Thus for any index e, the first component $(e)_0$ indicates the type of function, and $(e)_1$ indicates the arity.

There is nothing sacred about this particular way of coding PR terms, there are many other, equally natural ways.

Diagonalization

Now suppose eval is p.r., and define the following function

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

This may look weird, but certainly f is also p.r. and must have an index e. But then

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

and we have a contradiction.

So eval is another example of an intuitively computable function that fails to be primitive recursive.

This example may be less sexy than the Ackermann function, but it appears in similar form in other contexts.

Partial Functions

How do we avoid the problem with eval?

The only plausible solution appears to be to admit partial functions, functions that, like eval, are computable but may fail to be defined on some points in their domain. In this case, $\text{eval}(e, e)$ is undefined.

For a CS person, this is a fairly uncontroversial idea: everyone who has ever written a sufficiently sophisticated program will have encountered divergence: on some inputs, the program simply fails to terminate.

What may first seem like a mere programming error, is actually a fundamental feature of computable functions.

General Computability

We presented the last argument in the context of primitive recursive functions, but note that the same reasoning also works for any clone of computable functions—as long as

- successor and eval both belong to the clone, and
- each function in the clone is represented by an index.

But then eval must already be partial, no matter what the details of our clone are.

General computability requires partial functions, basta.

Notation Warning

We write

$$f\colon A \nrightarrow B$$

for a partial function from A to B.

Terminology:

domain	dom $f = A$
codomain	cod $f = B$
support	spt $f = \{\, a \in A \mid \exists\, b\, f(a) = b \,\}$

Warning: Some misguided authors use “domain of definition” instead of support, and then forget the “of definition” part.

Faking It

Suppose we have a partial function $f\colon \mathbb{N} \nrightarrow \mathbb{N}$. We could try to turn f into a total function $F\colon \mathbb{N} \nrightarrow \mathbb{N}$ by setting

$$F(x) = \begin{cases} f(x)+1 & \text{if } f(x)\downarrow \\ 0 & \text{otherwise.} \end{cases}$$

F clearly is total, and we can easily recover f from it.

But this it is not very useful for us: as we will see shortly, there are computable f such that F fails to be computable—though, of course, in set theory la-la land there is no problem at all.

Kleene’s Notation

Given a clone of computable functions, such as the primitive recursive ones, we write

$$\{e\}$$

for the eth function in the collection, $e \geq 0$. Here the index e is a sequence number, but it is helpful to think of it as a program (in some suitable language).

Since these functions are partial in general we have to be a bit careful and write

$$\{e\}(x) \simeq y$$

to indicate that $\{e\}$ with input x returns output y. This notation is a bit sloppy, arguably we should also indicate the arity of the function–but for us that’s overkill.

More Convergence

To express convergence we also write

$$\{e\}(x)\downarrow$$

if $\{e\}$ on input x terminates and produces some output, and

$$\{e\}(x)\uparrow$$

when the computation fails to terminate.

For example, Kleene equality $\{e\}(x) \simeq \{e'\}(x)$ should be interpreted as:

- either $\{e\}(x)\downarrow$ and $\{e'\}(x)\downarrow$ and the output is the same; or
- $\{e\}(x)\uparrow$ and $\{e'\}(x)\uparrow$.

Insane Growth

Informally, the Ackermann function cannot be primitive recursive because it grows far too fast. On the other hand, it does not really have a particular purpose other than that.

We will give another example of mind-numbing growth based on an actual counting problem. To this end, it is easier to use a slight variant of the Ackermann function.

$$B_1(x) = 2x$$
$$B_{k^+}(x) = B_k^x(1)$$

$B_k^x(1)$ means: iterate B_k x-times on 1. So B_1 is doubling, B_2 exponentiation, B_3 super-exponentiation and so on.

In general, B_k is closely related to A_{k+1}.

Subsequence Order

Recall the subsequence ordering on words where $u = u_1 \dots u_n$ precedes $v = v_1 v_2 \dots v_m$ if there exists a strictly increasing sequence $1 \leq i_1 < i_2 < \dots i_n \leq m$ of positions such that $u = v_{i_1} v_{i_2} \dots v_{i_n}$.

In symbols: $u \sqsubseteq v$.

In other words, we can erase some letters in v to get u. Note that it is easy to check for subsequences in linear time.

Subsequence order is never total unless the alphabet has size 1.

Subsequence order is independent of any underlying order of the alphabet (unlike, say, lexicographic or length-lex order).

Warmup: Antichains

An antichain in a partial order is a sequence $x_0, x_1, \dots, x_n, \dots$ of elements such that x_i and x_j are incomparable for $i < j$.

Example

Consider the powerset of $[n] = \{1, 2, \dots, n\}$ with the standard subset ordering. How does one construct a long antichain?

For example, $x_0 = \{1\}$ is a bad idea, and $x_0 = [n]$ is even worse.

What is the right way to get a long antichain?

Higman's Lemma

Theorem (Higman's 1952)

Every antichain in the subsequence order is finite.

Sketch of proof. Here is the Nash-Williams proof (1963): assume there is an infinite antichain.

For each n, let x_n be the length-lex minimal word such that $x_0, x_1, \dots, x_n$ starts such an antichain, producing a sequence $x = (x_n)$.

Construct a new sequence $y = (y_i)$ by choosing a letter a that appears infinitely often as the first letter in (x_n) and copying the words up to the first occurrence of one of these a-words. Follow by all the a-words, but with the first letter removed.

Proof, contd.

One can check that the new sequence (y_i) is also an infinite antichain.

But it violates the minimality constraint on (x_i), contradiction.

$\square$

Note that this proof is highly non-constructive. A lot of work has gone into developing more constructive versions of the theorem, but things get a bit complicated.

See Seisenberger.

Friedman's Self-Avoiding Words

We are using 1-indexing. For a finite or infinite word x write

$$x[i] = x_i, x_{i+1}, \dots, x_{2i}$$

Note this makes sense only for $i \leq |x|/2$ when x is finite.

Bizarre Definition: A word is self-avoiding if for all $1 \leq i < j \leq |x|/2$ the block $x[i]$ is not a subsequence of $x[j]$.

The following is an easy consequence of Higman's theorem.

Theorem

Every self-avoiding word is finite.

How Long?

Write Σ_k for an alphabet of size k.

By the last theorem and König's lemma, the set S_k of all finite self-avoiding words over Σ_k must itself be finite.

But then we can define the following function:

$$\alpha(k) = \max\big(|x| \mid x \in S_k \big)$$

So $\alpha(k)$ is the length of the longest self-avoiding word over Σ_k.

So α is strictly increasing. The question is how quickly α grows.

The Algorithm

Here is the obvious brute-force algorithm.

- At round 0, define the list of words $L = \{\varepsilon\}$.
- In each round, extend all words in L by all letters in Σ_k. Remove non-self-avoiding words from L.
- Stop at round $n+1$ when L becomes empty. Then $\alpha(k) = n$.

Each step is easily primitive recursive.

Termination is guaranteed by the theorem: we are essentially growing a tree (actually: a trie). If the algorithm did not terminate the tree would be infinite and thus have an infinite branch, which branch would be a infinite self-avoiding word.

How Big?

- Trivially, $\alpha(1) = 3$.
- A little work shows that $\alpha(2) = 11$, as witnessed by $abbbaaaaaaa$.
- But

  $$\alpha(3) > B_{7198}(158386),$$

  an incomprehensibly large number.

Smelling salts, anyone?

It is truly surprising that a function with as simple a definition as α should exhibit this kind of growth.

It's a Feature

At this point one might wonder whether our whole approach to computability is perhaps a bit off—we certainly did not intend to deal with monsters like α.

Alas, as it turns out this is a feature, not a bug: all reasonable definitions of computability admit things like α, and worse.

It is a fundamental property of computable functions that some of them have absurd growth rates.

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.

Why Not Turing Machines?

The class of functions defined by register machines is the same as the class of functions defined by Turing machines, so in a sense the choice does not matter.

However, Turing machines are exceedingly tedious to construct, even simple tasks like testing primality are ridiculously complicated. Building a universal machine (see below) is a mess. Register machines are much better behaved in this case.

Full disclosure: for complexity theory, Turing machines are the gold-standard. But for us, that does not matter so much.

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.

RM Programs

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 machines 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.

RM-Computability

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 of R_0, producing the result $b = f(\boldsymbol{a})$.
- 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:

- $I_p =$ `inc r k`:
  $q = k$ and $\boldsymbol{y} = \boldsymbol{x}[x_r \mapsto x_r + 1]$
- $I_p =$ `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: $(p, \boldsymbol{x}) \vdash_P^1 (q, \boldsymbol{y})$.

Note that if $(p, \boldsymbol{x})$ is halting (i.e. $I_p =$ `halt`) there is no next configuration. Ditto for $p \geq n$.

Whole Computation

Define

$$(p, \boldsymbol{x}) \vdash_P^0 (q, \boldsymbol{y}) :\Leftrightarrow (p, \boldsymbol{x}) = (q, \boldsymbol{y})$$

$$(p, \boldsymbol{x}) \vdash_P^t (q, \boldsymbol{y}) :\Leftrightarrow \exists\, q', \boldsymbol{y}' \;\; (p, \boldsymbol{x}) \vdash_P^{t-1} (q', \boldsymbol{y}') \vdash_P^1 (q, \boldsymbol{y})$$

$$(p, \boldsymbol{x}) \vdash_P (q, \boldsymbol{y}) :\Leftrightarrow \exists\, t \;\; (p, \boldsymbol{x}) \vdash_P^t (q, \boldsymbol{y})$$

A computation (or a run) of P is a sequence of configurations C_0, C_1, C_2, ... where $C_i \vdash_P^1 C_{i+1}$. 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.

Cleaning Up

Note that 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.

Wlog, $I_{\ell-1}$ is a halt instruction, and there are no others.

It follows that these clean RMs cannot get stuck, every computation either ends in halting, or is infinite. From now on, we will always assume that our programs are syntactically correct in this sense.

Exercise

Write a program transformer that converts an arbitrary RMP into an "equivalent" one that has these extra properties.

Termination

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

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

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 $E_{\boldsymbol{a}} = (0, (0, \boldsymbol{a}, \mathbf{0}))$.

Definition

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

- If $\boldsymbol{a}$ is in the support of f, then the computation of P on $C_0 = E_{\boldsymbol{a}}$ terminates in configuration $C_n = (\ell-1, b, \boldsymbol{y})$ where $f(\boldsymbol{a}) \simeq b$.
- If $\boldsymbol{a}$ is not in the support of f, then the computation of P on $E_{\boldsymbol{a}}$ fails to terminate.

A Subtlety

Recall that according to our convention, it is not admissible that an RM program could get stuck (because a goto uses a non-existing label). What if we allowed arbitrary RM programs instead of only clean ones?

The class of computable functions would not change one bit, our definitions are quite robust under (reasonable) modifications. This is a good sign, fragile definitions are usually of little interest.

Exercise

Modify the definition so "getting stuck" is allowed and show that we obtain exactly the same class of partial functions this way. Invent RMs without a halt instruction.

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.

Aside: Time Complexity

The number of steps in a finite computation provides a measure of complexity, in this case time complexity.

Given a RM P and some input $\boldsymbol{x}$ let $(C_i)_{i<N}$, where $N \leq \omega$, be the computation of P on $\boldsymbol{x}$.

We write the time complexity of P as

$$T_P(\boldsymbol{x}) = \begin{cases} N & \text{if } N < \omega, \\ \omega & \text{otherwise.} \end{cases}$$

If you are worried about ω just read it as ∞. Alternatively, we could use $N-1$ as our step-count.

This may sound trivial, but it's one of the most important ideas in all of computer science.

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.

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

Digression: Notation

There is actually something very important going on here: we are trying to produce notation that works well with the human cognitive system.

Humans are exceedingly bad at dealing with fully formalized systems; in fact, we really cannot read formal mathematics except in the most trivial (and useless) cases. Try reading Russell-Whitehead's Principia Mathematica if you don't believe me.

The current notation system in mathematics evolved over centuries and is very carefully fine-tuned to work for humans.

Computers need an entirely different presentation and it is very difficult to move between the two worlds.

Example: Multiplication

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

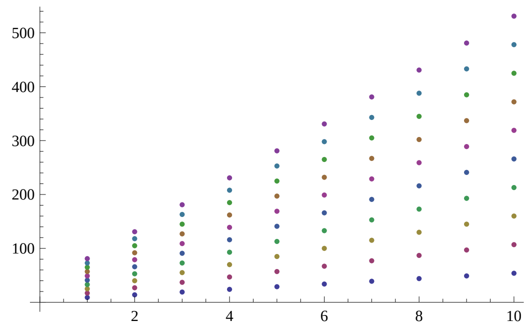
```
// multiplication   X Y --> Z
  0:   dec X  1  6
  1:   dec Y  2  4
  2:   inc Z  3
  3:   inc U  1
  4:   dec U  5  0
  5:   inc Y  4
  6:   halt
```

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
```

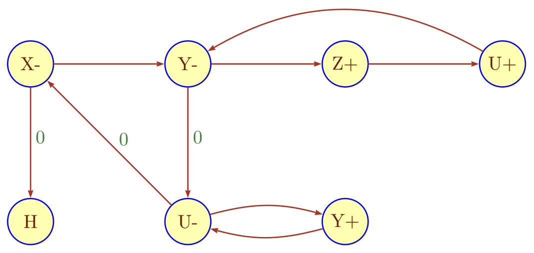
Time Complexity?



Exercise

Determine the time complexity of the multiplication RM.

Flowgraph for Multiplication

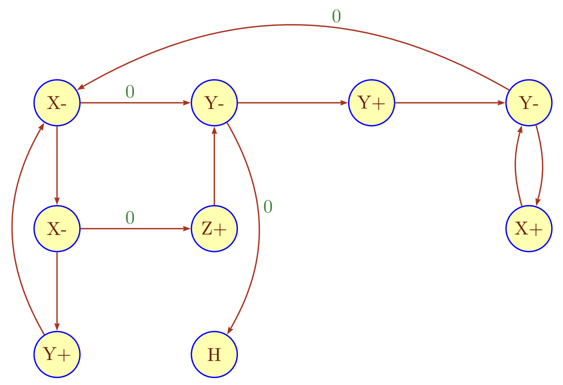


Example: Binary Digit Sum

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

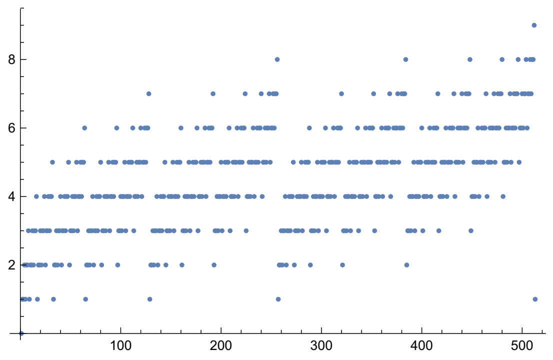
```
// binary digitsum of X --> Z
  0:   dec X  1  4
  1:   dec X  2  3
  2:   inc Y  0
  3:   inc Z  4
  4:   dec Y  5  8
  5:   inc Y  6
  6:   dec Y  7  0
  7:   inc X  6
  8:   halt
```

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. Implement a p.r. to RM compiler.

Exercise

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

Recall: Coding

Recall the three coding functions from last time:

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

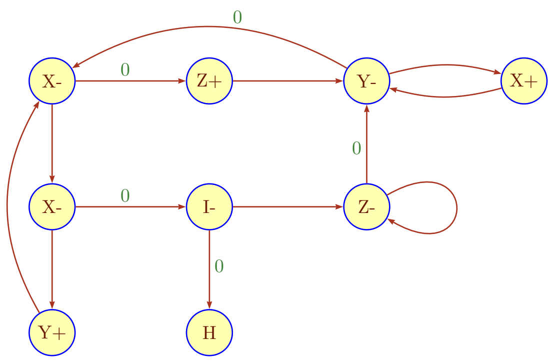
$$\mathsf{dec} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

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

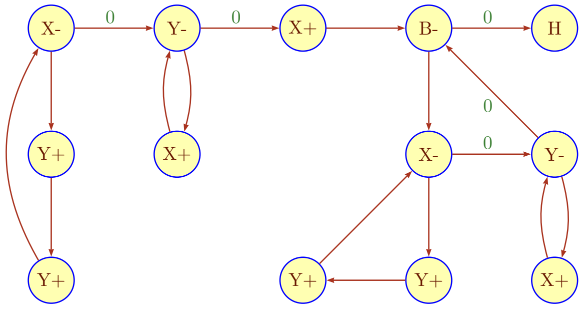
Note that len is just the binary digit-sum.

One can check that dec can be computed by a fairly small register machine. As usual, for $\langle . \rangle$ we would have to fix the number of arguments.

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: his famous incompleteness theorem takes a wrecking ball to first-order logic.

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

$$\langle \langle 0,1,2 \rangle, \langle 1,0 \rangle, \langle 0 \rangle \rangle = 88098369175552.$$

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

Turing and Universality

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)

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

WTF?

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.

Simulating Random Access Memory

How does one construct a universal computer? According to the last section, we can code a RMP $P = I_0, I_1, \dots, I_{\ell-1}$ 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.

Macros

- `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--
12:   write  R  r  x  2           // R[r] = x; goto 2
13:   halt
```

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.

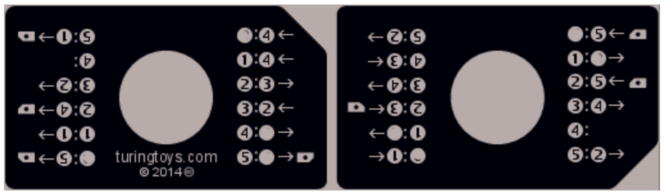
Exercise

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.

A Universal Turing Machine



Exercise

Figure out what this picture means.

Exercise (Very Hard)

Prove that this is really a universal Turing machine.