CDM

Wild Computation

Klaus Sutner

Carnegie Mellon University

Fall 2024

Ackermann's Function (1928)

Primitive recursion uses only a single variable. One might suspect that recursion over multiple variables could potentially produce more complicated functions, but one needs to be careful: who knows, maybe there is some clever way to express a multiple recursion in terms of a single one.

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 $x + 1$.

$$\begin{aligned} A(0, y) &= y^+ \\ A(x^+, 0) &= A(x, 1) \\ A(x^+, y^+) &= A(x, A(x^+, y)) \end{aligned}$$

On the surface, this looks more complicated than primitive recursion. We need to make sure that there really is no trick to rewrite this as a single recursion.

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:

$$A_0 = S \qquad A_{x^+}(0) = A_x(1)$$

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

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

The basic operation behind 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 uncomfortably large number (we'll see much worse in a moment).

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

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

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

$$A(7,y) \approx \text{speechlessness}$$

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

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: "Whereof one cannot speak, thereof one must be silent."*

---

*"Wovon man nicht sprechen kann, darüber muss man schweigen." *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 more small values of A (compared to the plain recursion), but does not go very far: the hash table becomes huge very soon.

It's a fun game to try to use any kind of dirty trick to compute as many values of the Ackermann function given the constraints of a particular piece of hardware.

More interesting is the following: recall our claim that proofs and computations are very closely related.

In the context of Ackermann, suppose someone claims that $A(a, b) = c$. We want proof that this result is really correct.

One way to establish correctness is to use memoizing during the computation and to hand over the corresponding table H.

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

To check the proof for correctness, it suffices to check the following consistency properties of the table H:

$$H[0, y] = y + 1$$

$$H[x^+, 0] = z \quad \text{implies} \quad H[x, 1] = z$$

$$H[x^+, y^+] = z \quad \text{implies} \quad H[x, z'] = z \text{ where } z' = H[x^+, y]$$

The whole check comes down to performing $O(N)$ table lookups where N is the number of entries in H.

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

Of course, N is usually so huge that this is not a feasible check.

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

$$A(a,b) = \mathsf{dec}\big(\min\big(\, z \mid R(a,b,z)\,\big)\big)$$

Here $\mathsf{dec}(s)$ is a p.r. decoding function.

Sketch of proof. Think of z as a code (Gödel number) of the result c and the hash table H.

R performs the consistency test described above and is clearly primitive recursive. dec just extracts the result c from z. □

Recursion Stack

In some cases, a recursion based computation unfolds in a very simple, predictable manner. If that is the case, then it is usually a good idea to try to figure out what the recursion stack looks like during the execution.

With luck, the pattern will be so simple that we can implement the operations directly on a list (representing the stack, without all the bureaucracy

Alternatively, one can try to find a systematic approach to solving the system of equations, essentially by repeated instantiations and substitutions.

Again, with luck, a simple pattern will emerge that provides a computational shortcut.

Unfolding Ackermann

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

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

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

Here is an algorithm that works on integer lists: initially, the list is (a, b). The algorithm terminates when the list has length 1. A single step looks like so:

$$\Delta(\ldots, 0, y) = (\ldots, y^+)$$

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

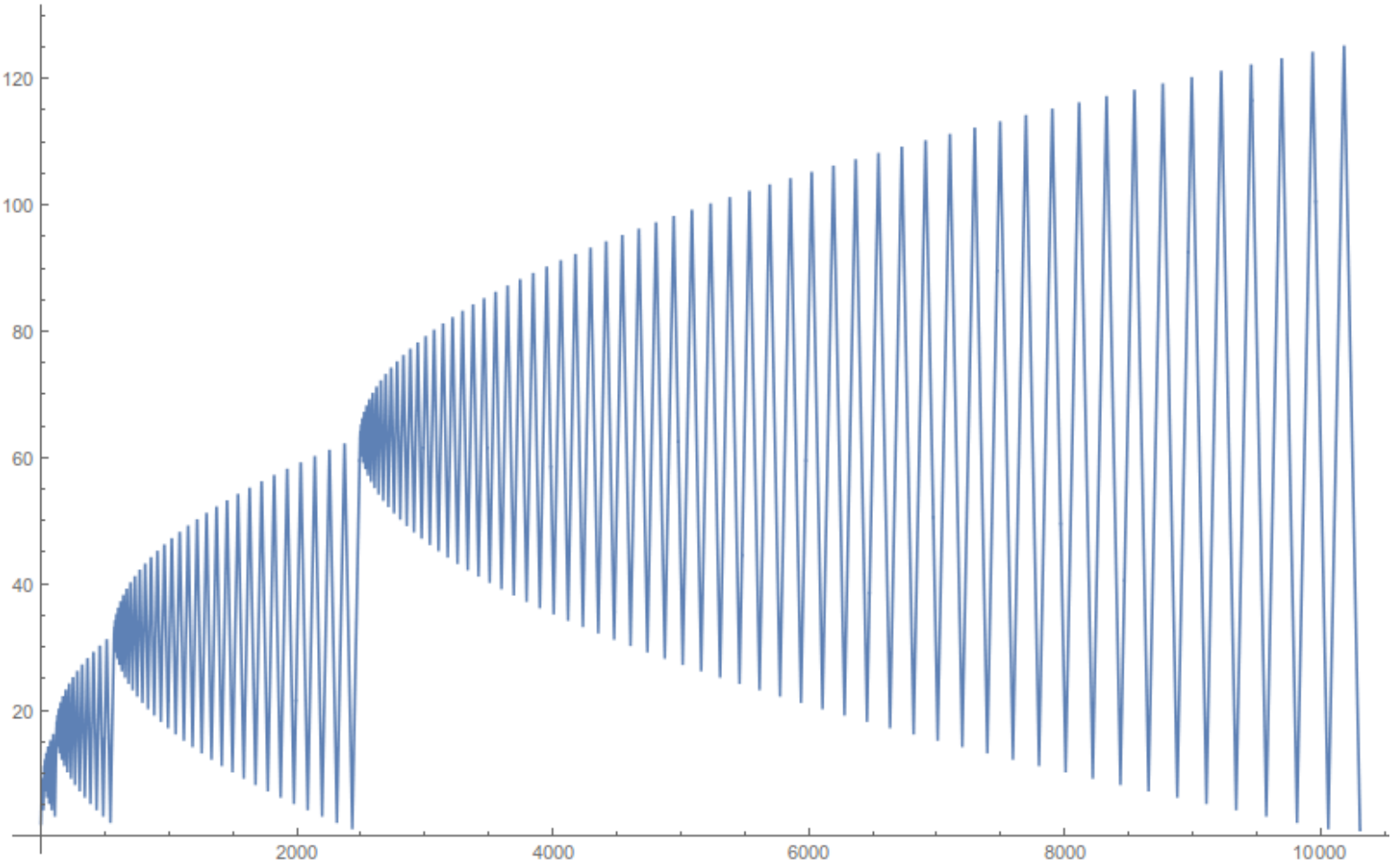
$$\Delta(\ldots, x^+, y^+) = (\ldots, x, x^+, y)$$

If we encode integer lists as integers, the single-step operation Δ is primitive recursive. Using actual data structures, Δ is just about trivial.

List Algorithm


```
int acker_list(int a, int b)
{
    L = (a,b);
    while( len(L) > 1 )
        L = Delta(L);
    return fst(L);
}
```

Everything is perfectly harmless, except that the loop runs for a long, long time (and the lists get horribly long).



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

Evaluation

We have seen that primitive recursive functions can be defined in terms of expressions τin a small programming language. Any syntactically correct term τ describes an arithmetic function $[\![\tau]\!] : \mathbb{N}^k \to \mathbb{N}$.

For example, the term

$$\tau = \mathsf{Prec}[\mathsf{Prec}[\mathsf{Prec}[\mathsf{S} \circ \mathsf{P}^3_2, \mathsf{P}^1_1] \circ (\mathsf{P}^3_2, \mathsf{P}^3_3), \mathsf{C}^{(1)}_0] \circ (\mathsf{S} \circ \mathsf{P}^2_1, \mathsf{P}^2_2), \mathsf{C}^{(0)}_1]$$

describes the factorial function, $[\![\tau]\!]$ is the factorial function.

It is intuitively clear that we could write an interpreter, a function eval that takes as input the string τ and a suitable input vector $\boldsymbol{x} = x_1, \ldots, x_k \in \mathbb{N}^k$ and returns the result of evaluating $[\![\tau]\!]$ on $\boldsymbol{x}$:

$$\mathsf{eval}(\tau, \boldsymbol{x}) = \text{ value of } [\![\tau]\!] \text{ on arguments } \boldsymbol{x}$$

Arithmetizing

The type of eval is

$$\text{eval} : \mathit{expressions} \times \mathbb{N}^\star \longrightarrow \mathbb{N}$$

But Gödel has shown how to express any finitary object in terms of natural numbers, we can translate the string τ into a Gödel number For computable functions, the corresponding Gödel number is usually called an index, written $e = \langle \tau \rangle$. Similarly we can replace $\boldsymbol{x}$ by $\langle \boldsymbol{x} \rangle$.

So its safe to think of evaluation as a map

$$\text{eval} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$$

If e is not an index, we may assume $\text{eval}(e, x) = 0$.

Indices

There are many ways to organize the coding details of $\langle \tau \rangle$, for the time being think about replacing the letters in τ by positive natural numbers (something like ASCII) and then use the coding function

$$\langle a_1, a_2, \ldots, a_k \rangle = p_1^{a_1} p_2^{a_2} \ldots p_k^{a_k}$$

where (p_i) is the usual enumeration of prime numbers.

This function is multiadic, and hence cannot be primitive recursive, but if we fix k it is indeed p.r., and we can recover the elements in a p.r. manner. Ditto for the second argument of eval.

Details next lecture.

Coding PR

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

term	code
$\mathsf{C}_0^{(0)}$	$\langle 0, 0 \rangle$
S	$\langle 1, 1 \rangle$
P_i^n	$\langle 2, n, i \rangle$
$\mathsf{Prec}[h, g]$	$\langle 3, n, \widehat{h}, \widehat{g} \rangle$
$\mathsf{Comp}[h, g_1, \ldots, g_n]$	$\langle 4, m, \widehat{h}, \widehat{g_1}, \ldots, \widehat{g_n} \rangle$

Thus for any index e, the first component $\mathsf{fst}(e)$ indicates the type of function, and $\mathsf{snd}(e)$ indicates the arity.

This type of coding makes it really easy to write an interpreter.

Question: Could eval be primitive recursive?

Let's assume eval primitive recursive. The we can define the following function

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

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

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

and we have a contradiction, $0 = 1$.

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, eval(e, e) is undefined.

Anyone 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.

Incidentally, in the early days of recursion theory, partial functions were universally avoided.

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.

A similar argument shows that an interpreter for, say, polynomial time computable functions cannot itself by polynomial time.

Fiat Halting

Since any general model of computation must deal with partial functions, it is entirely natural to ask whether a given function f is defined on some particular input x.

Another reasonable question would be to ask whether f is total; or even whether f is nowhere defined.

So we automatically run into the Halting Problem, the first example of a perfectly well-defined question that turns out to be undecidable.

Notation Warning

We write

$$f : A \nrightarrow B$$

for a partial function from A to B. Terminology:

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

It is also convenient to write $f(x)\downarrow$ for $x \in \operatorname{spt} f$, and $f(x)\uparrow$ for $x \notin \operatorname{spt} f$ (converges/diverges).

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 $f : \mathbb{N} \rightharpoonup \mathbb{N}$. We could try to turn f into a total function $f_\perp : \mathbb{N} \rightharpoonup \mathbb{N}$ by setting

$$f_\perp(x) = \begin{cases} f(x) + 1 & \text{if } x \in \text{spt } f \\ 0 & \text{otherwise.} \end{cases}$$

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

In set theory la-la land there is no problem at all. But this construction is not very useful for us: there are computable f such that $f_\perp$ fails to be computable.

Kleene's Notation

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

Given expressions α, β involving partial functions, we use Kleene equality rather than plain 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).

More Kleene

Given a clone of computable functions, such as the primitive recursive ones, and an index e for one of these functions, we write

$$\{e\}$$

for the eth function in the collection. Hence, $(\{e\})_{e\geq 0}$ is an enumeration of all the functions in the clone.

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.

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?

Conventions

- The tape alphabet is $\mathbf{2}$, 0 is the blank symbol; initially, the tape contains only 0s.
- The tape head must move left or right at each step.
- We ignore the halting state, so n means "n ordinary states plus one halting state."

These details do not matter when it comes to defining computability in general, but they make a difference here.

Hierarchy

We write $\mathrm{BB}_{\mathrm{T}}(n)$ for largest number of steps of any halting n-state machine and refer to BB_{T} 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 by any halting n-state machine.

Clearly, $\mathrm{BB}_{\mathrm{T}}(n) \geq \mathrm{BB}_{\mathrm{W}}(n)$, 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 $n = 1$

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

To see this, note that any attempt to make a second move would already lead to an infinite loop. Recall that the displacement of the head is required to be ± 1.

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

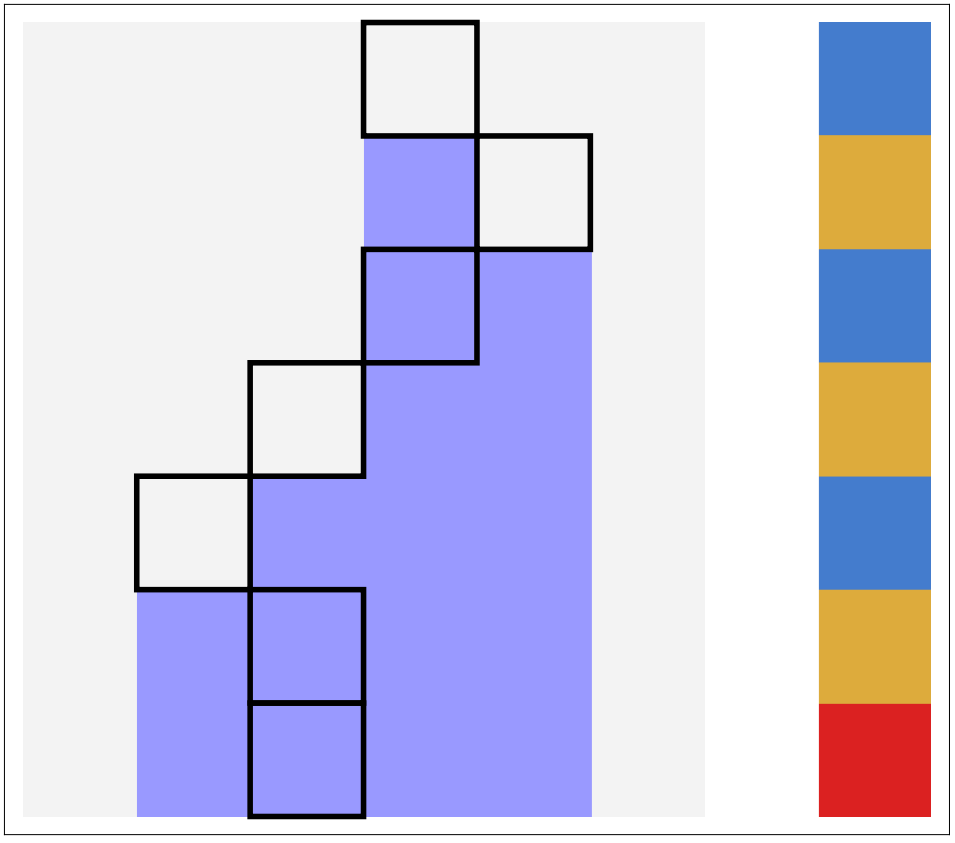
Busy Beaver $n = 2$

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



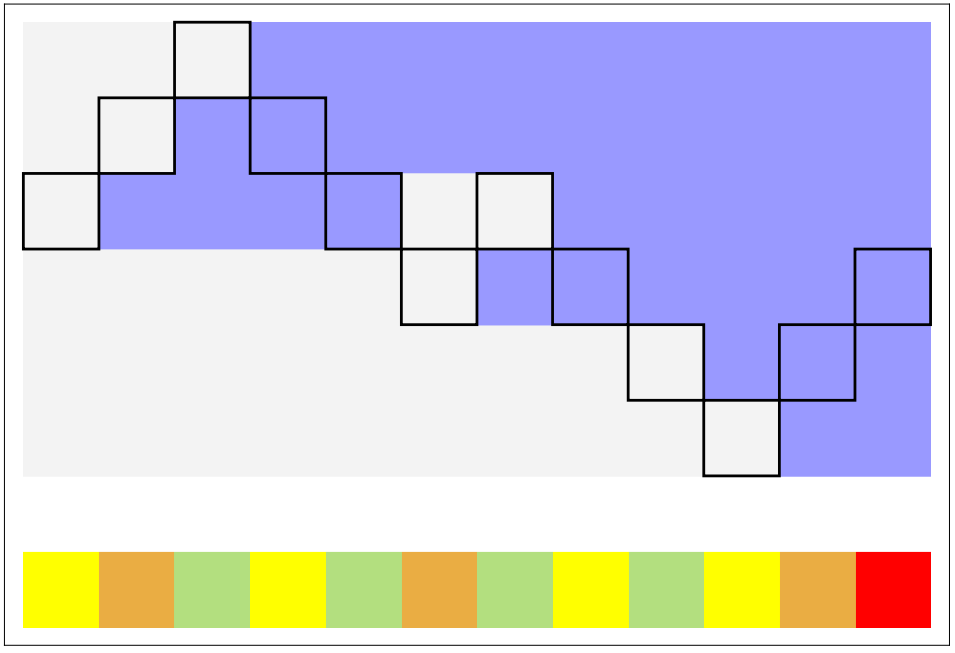
Busy Beaver $n = 3$

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

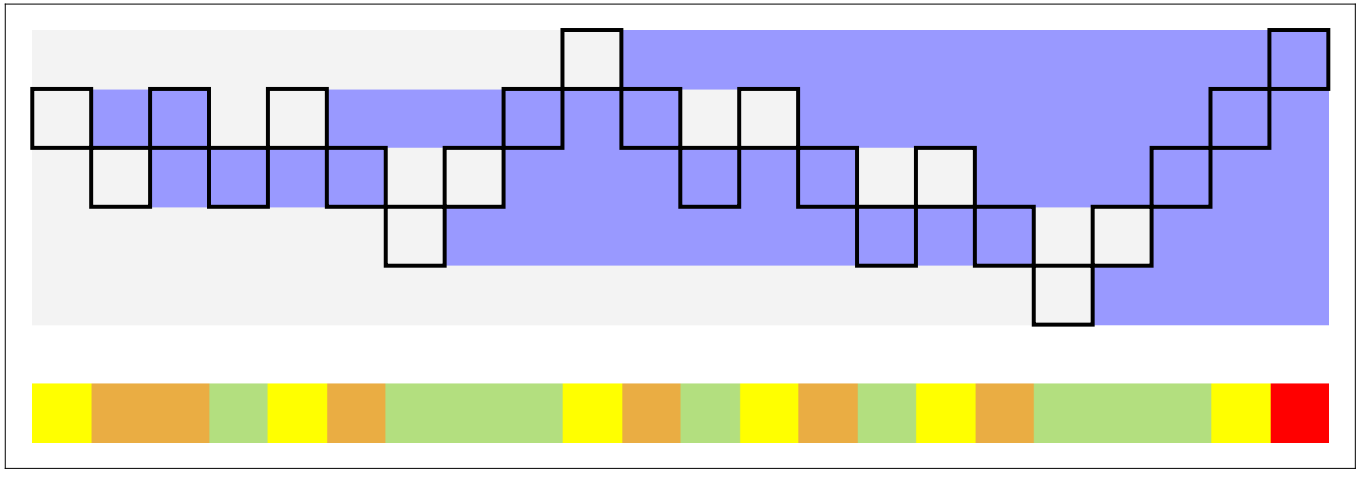
Exploiting isomorphisms, filtering out machines where all 4 states are reachable (in the diagram, not necessarily the computation on empty tape), and checking for halting we get down to $405\,072$

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.

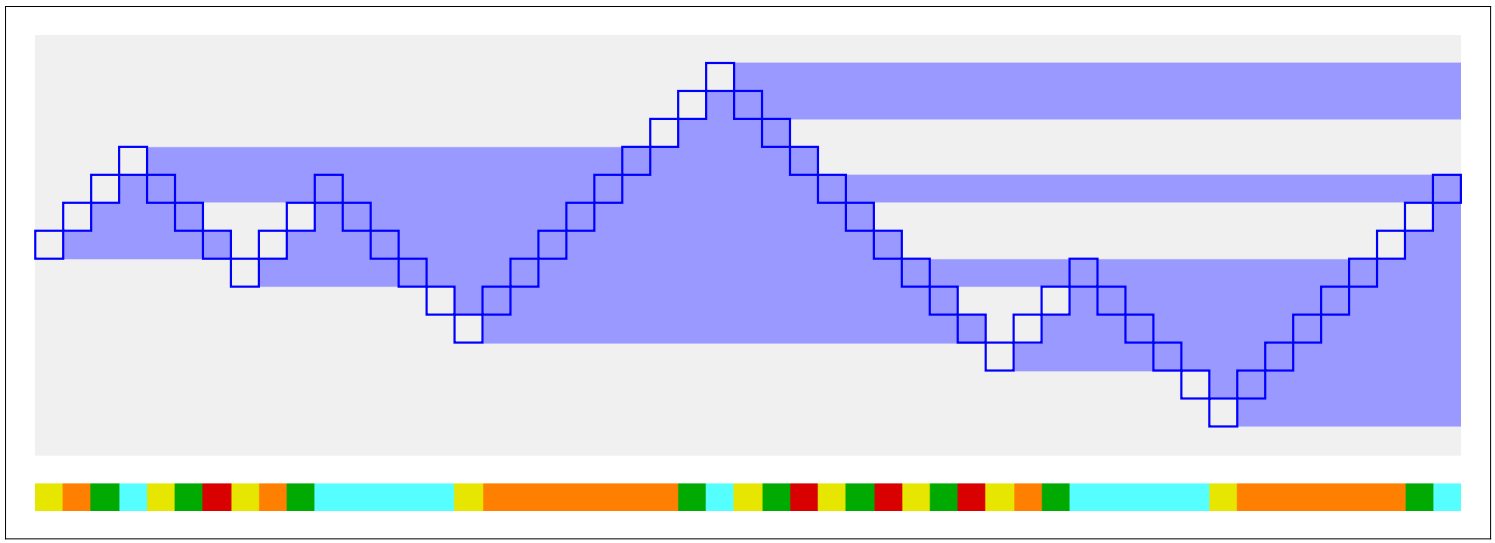
The Marxen-Buntrock Machine

The current champion machine was found by Marxen and Buntrock, and its discovery is a small miracle. Here is the table of the machine.

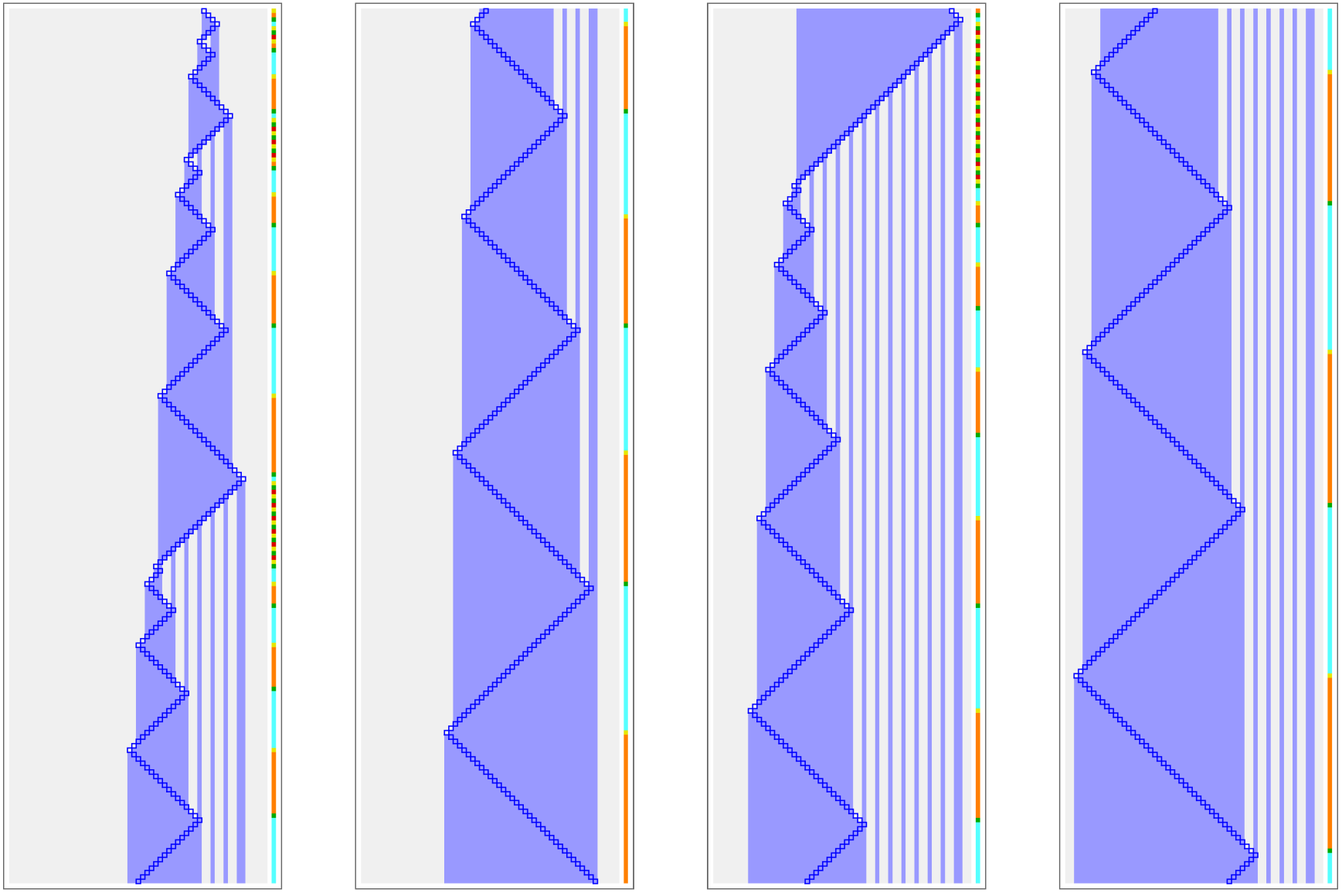
	0	1
1	(2,1,R)	(3,1,L)
2	(3,1,R)	(2,1,R)
3	(4,1,R)	(5,0,L)
4	(1,1,L)	(4,1,L)
5	halt	(1,0,L)

One can check that all 5 states are used during the the computation with initial state 1 and 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 head zig-zags back and forth, sometimes building solid blocks of 1's, sometimes a striped pattern $100100100\ldots$

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".

Still, this machine stops after 47 176 870 steps on output $10(100)^{4097}$.

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	$\mathrm{BB}_{\mathrm{T}}(n)$	$\mathrm{BB}_{\mathrm{W}}(n)$
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 available for $n \leq 4$; beyond that, we only have bounds. And these bounds soon get ridiculous:

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

Alas, these results are not as robust as one would like them to be, see Harland 16 for a critique.

For $n = 6$ all hell breaks loose.

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

Halting gets very messy here: there is no good heuristic to come to the conclusion that a machine will never halt and can thus be dismissed from the competition.

A new machine by Pavel Kropitz, takes about $10 \uparrow\uparrow 15$ steps to halt. A more precise bound is

$$10^{10^{10^{10^{10^{10^{10^{10^{10^{10^{4.023873729}}}}}}}}}}$$

You're welcome.

Busy Beaver Exercises

Exercise

Derive the transition table of the 3-state Busy Beaver machine. Give an intuitive explanation of how this machine works.

Exercise

Prove that the last machine is indeed the champion: no other halting 3-state machine writes more than 6 ones.

Exercise (Hard)

Find the Busy Beaver champion for $n = 4$.

Exercise (Extremely Hard)

Organize a search for the Busy Beaver champion for $n = 5$.

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.

One nice features of subsequence order is that is entirely 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 $x_0, x_1, \ldots, x_n, \ldots$ of elements such that x_i and x_j are incomparable for $i < j$.

Example

Consider the powerset of $[n] = \{1, 2, \ldots, n\}$ 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 1952)

Every antichain in the subsequence order is finite.

Proof. Here is the Nash-Williams proof (1963): assume there is an infinite antichain. Then there is a non-increasing sequence $\boldsymbol{x} = (x_n)$ in the sense that $i < j$ implies that $x_i \not\sqsubseteq x_j$.

By induction on n, choose the minimal such sequence in the sense that x_n is the length-lex minimal word such that $x_0, x_1, \ldots, x_n$ starts a non-increasing sequence.

There must be a letter a such that the subsequence $x_{n_j} = a\, y_j$, $j \geq 0$, of words starting with a, is infinite. Let $k = n_0$ and define a new sequence

$$\boldsymbol{z} = x_0, x_1, \ldots, x_{k-1}, y_0, y_1, \ldots$$

Proof, contd.

One can check that the new sequence z is again non-increasing.

But z violates the minimality constraint on x at position k, contradiction.

□

Note that this proof is highly non-constructive. We are essentially performing surgery on a branch in an infinite tree that exists by assumption. A lot of work has gone into developing more constructive versions of the theorem, but things get a bit complicated.

See Seisenberger.

(Weak) Kőnig Infinity Lemma

Theorem

Every infinite, finitely branching tree contains an infinite branch.

In the weak version, the tree is required to be binary.

Weak implies full: replace higher branching nodes by binary trees.

Friedman's Self-Avoiding Words

We are using 1-indexing. For a finite or infinite word x define the ith block of x to be the factor (of length $i+1$) of x:

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

Note this makes sense only for $i \leq |x|/2$ when x is finite. We will always tacitly assume that this bound holds.

Bizarre Definition: A word is self-avoiding if, for all $i < j$, the block $x[i]$ fails to be a subsequence of block $x[j]$.

For example,

$abbbaaaa$	is self-avoiding
$abbbaaab$	is not self-avoiding

Only Finite

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

Theorem

Every self-avoiding word is finite.

If there were an infinite self-avoiding word $x \in \Sigma^\omega$, the collection $\{ x[i] \mid i \geq 1 \}$ of all its blocks would form an infinite antichain.

How Long?

Write Σ_k for an alphabet of size k.

By the last theorem and Koenig'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 max-length 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.

Clearly, α is total and it is strictly increasing.

Moreover, α is easily computable, there is a very straightforward algorithm to determine the value of $\alpha(k)$.

The Algorithm

Note that any prefix of a self-avoiding word must also be self-avoiding. This produces a simple, brute-force algorithm to compute α.

- At round 0, define $S = \{\varepsilon\}$.
- In each round, extend all words in $x \in S$ by all letters $a \in \Sigma_k$. If xa is still self-avoiding, keep it; otherwise toss it.
- When S becomes empty at round $n+1$, return $\alpha(k) = n$.

Each step is easily primitive recursive, really just some wordprocessing.

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, corresponding to an infinite self-avoiding word; contradiction.

Some Examples

Here is the number of self-avoiding words of length up to 12, for $k \leq 4$.

1	2	3	4	5	6	7	8	9	10	11	12
1	1	1	0	0	0	0	0	0	0	0	0
2	4	8	8	16	12	24	4	8	2	4	0
3	9	27	60	180	348	1044	1518	4554	5334	16002	16674
4	16	64	216	864	2688	10752	29376	117504	285108	1140432	2569248

So $\alpha(1) = 3$: the first time a word contains 2 blocks, it is not longer self-avoiding.

A little fumbling (or writing a program) shows that $\alpha(2) = 11$, as witnessed by $abbbaaaaaaa$ and $abbbaaaaaab$ and their duals.

Challenge for Hackers

Write a program in your favorite fast language that extends the table, ideally by a column or a row.

I suspect the former is feasible, the latter may be tricky.

And $k = 3$?

Alas, $\alpha(3)$ is a bit harder to describe. We will use a slight variant of the Ackermann function for this purpose.

$$B_1(x) = 2x$$

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

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

Just like the Ackermann function, B_5 essentially makes no sense to mere mortals, its growth rate is stupendous.

Drum Roll

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

This is an incomprehensibly, mind-numbingly large number.

Never mind the 158386, it's the 7198 that kills any chance of understanding, at least roughly, what this means.

Smelling salts, anyone?

It is truly surprising that a function like α with a really simple algorithm should exhibit this kind of growth.

It's a Feature

And, of course, there is $\alpha(\alpha(3))$, $\alpha(\alpha(\alpha(3)))$, and so on. Or how about

$$\alpha^{\alpha(3)}(3)$$

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. Far worse.

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

Theorem

The Busy Beaver functions fail to be computable.

Proof.

We will only deal with BB_{T} and leave the other one for homework. So suppose BB_{T} is computable. Given a Turing machine M and input x we can an "equivalent" machine M' that uses tape alphabet $\mathbf{2}$: M' on the empty tape simulates M on x.

Say, M' has $n+1$ states, including a single halting state. Compute $\mathrm{BB}_{\mathrm{T}}(n)$ to bound the computation of M', and we have solved the Halting Problem. Contradiction.

□

And Reasoning?

So $\mathrm{BB}_{\mathrm{T}}(n)$ is not computable, but how far could we get with reasoning? Say, we adopt a fairly strong (and presumably consistent) base theory $\mathcal{T}$ in first-order logic; something like Zermelo-Fraenkel, and throw in Choice for good measure, or even $V = L$ if you like.

Question: Can $\mathcal{T}$ prove that $\mathrm{BB}_{\mathrm{T}}(n) = m$ for many n and m?

One might think that ZF should be able to handle this just fine. But there is a problem: $\mathcal{T}$ is an axiomatic system: the axioms are decidable and the rules of inference are easily computable. It follows that the collection of all proofs in $\mathcal{T}$ is decidable: given a sequence of formulae, we can check whether it forms a correct proof.

Disaster Strikes

But then we can enumerate all proofs: just run through all possible sequences of formulae, and check which ones are proofs. $\mathcal{T}$ is inconsist iff one of these proofs ends in $0 = 1$.

Buth then it is straightforward (in principle, not practice) to construct a Turing machine on, say, s states that searches for an inconsistency in $\mathcal{T}$: it just enumerates all possible proofs, and halts if it finds a bad one that ends in $0 = 1$.

If $\mathcal{T}$ could prove that $\mathrm{BB}_{\mathrm{T}}(s) = m$, then it could prove its own consistency, contradicting Gödel's incompleteness theorem.

Theorem

Any consistent theory can only prove finitely many values of BB_{T}.

By carefully constructing the ZF-consistency checker from scratch, one can show that even $n = 1000$ is out of reach. This is hard.

The last method works since consistency is a Π_1 statement, it requires only one universal quantifier und looks like

$$\forall\, x\, \Phi(x).$$

This is meant to be an arithmetic formula, interpreted over the natural numbers. The matrix $\Phi(x)$ contains only bounded quantifiers of the form $\forall\, u < v$ and $\exists\, u < v$, propositional logic, and ordinary arithmetic.

More Π_1 Games

The actual meaning of $\forall\, x\, \Phi(x)$ does not really matter, the same approach also works for other Π_1 statements. It so happens that there are some very interesting ones.

Riemann Hypothesis Far from obvious, but true. It turns out that one can build a machine of less than 1000 states that halts iff the Riemann Hypothesis is false.

Goldbach Conjecture It seems that there is a machine on 27 states that halts iff Goldbach is false. This number is rather uncomfortably small.