The Largest Number Representable in 64 Bits

John Tromp *

Posted Date: 24 March 2026

doi: 10.20944/preprints202603.1897.v1

Keywords: Busy Beaver; Turing machine; lambda calculus

 doi:10.20944/preprints202603.1897.v1

Article

The Largest Number Representable in 64 Bits

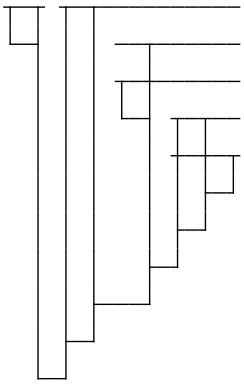
John Tromp

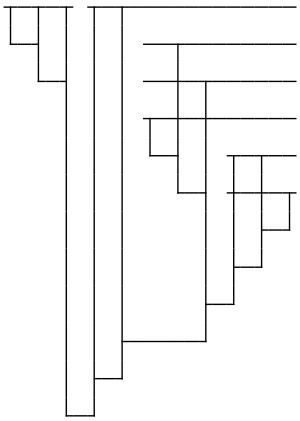
Independent Researcher, , Amsterdam, The Netherlands; Homepage: https://tromp.github.io/; john.tromp@gmail.com

Abstract

We investigate how large an output can be computed by programs fitting inside a single register.

Keywords: Busy Beaver; Turing machine; lambda calculus





1. Introduction

Most people believe $2^{64} - 1 = 18446744073709551615$, or FFFFFFFFFFFFFFFF in hexadecimal, to be the largest number representable in 64 bits. In English it's quite the mouthful: eighteen quintillion four hundred forty-six quadrillion seven hundred forty-four trillion seventy-three billion seven hundred nine million five hundred fifty-one thousand six hundred fifteen.

That is indeed the maximum possible value of 64 bit-unsigned integers, available as data type `uint64_t` in C or `u64` in Rust. Floating point data types can represent much larger values, courtesy of their base 2 exponent. The 64-bit floating point standard known as IEEE-754, available as data type `double` in C or `f64` in Rust, has a largest (finite) representable value of $2^{1024}(1 - 2^{-53}) \approx 1.8 \times 10^{308}$.

But we needn't limit ourselves to plain data types. What if we allow richer types of representations? Since we want representations to remain computable, the most general kind of representation would be a program in some programming language. But the program must be small enough to fit in 64 bits. Our quest then will be for the largest number programmable in 64 bits.

2. Small Programs

Some programming languages are not suitable for this task, due to some required scaffolding. A popular languages like C, for instance, while sporting the famous IOCCC (International Obfuscated C Code Contest) featuring small programs doing unbelievable things, requires every valid program to declare the main function such as `main(){}`, consisting of 8 ASCII characters. While technically, ASCII is a 7-bit character encoding standard representing 128 unique characters, all modern computers use 8-bit bytes to store either plain ASCII or UTF-8, a Unicode character encoding that's backward compatible with ASCII. So `main(){}` is essentially the only valid 64-bit C program.

Plenty other languages require no such scaffolding. For instance, Linux features the arbitrary precision calculator bc, which happily evaluates programs like `9^999999` to the 954242 digit number

 doi:10.20944/preprints202603.1897.v1

35908462...48888889, making it programmable in 8 bytes. So is the much larger `9^9^9^99` $= 9^{9^{99}}$ with over $10^{10^{953}}$ digits, which bc is less happy to compute. If bc supported the symbol ! for computing factorials, then `9!!!!!!!` would represent a much larger number still.

3. To Cheat or Not to Cheat

Allowing such primitives feels a bit like cheating though. Would we allow a language that has the Ackermann function predefined, letting the 8 byte expression `ack(9,9)` represent a truly huge number?

Many people will be quick to dismiss our quest entirely, arguing that whatever number N may be the largest representable in 64 bits, they can propose a new language in which the 1-bit program `0` is defined to "evaluate" to N+1. That is a clear example of cheating, i.e. designing a language to get the results you want. Those people may then also ask exactly what qualifies as non-cheating, but this is something that cannot be formalized, only recognized after the fact.

Our quest is for the largest number programmable in 64 bits without cheating.

4. Ackermann Considered Unhelpful

As it turns out, the question of whether Ackermann is allowed, is moot. One can blow way past `ack(9,9)` in under 64 bits in a language with no built in primitive whatsoever. A language with no basic arithmetic; not even numbers themselves. A language in which all those must be defined from scratch.

But let's first look at another language lacking the usual programming primitives, one that has been particularly well studied for producing largest possible outputs. That is the language of Turing machines.

5. The Busy Beaver

The famous Busy Beaver function, introduced by Tibor Radó in 1962 [1], which we'll denote BB(n), is defined as the maximal number of steps taken by an n-state Turing Machine (TM) with a binary tape alphabet, starting from an all 0 tape, before halting.

Here we have a discrepancy between how the size of a TM is measured, in states, versus how program size is measured, in bits. Fortunately there is a straightforward binary encoding of n-state TMs, which is entirely determined by its transition function. For each of the n states that the machine's finite control can be in, and each of its 2 tape symbols that could be scanned by its tape head, the transition function specifies what new symbol to write in the scanned tape cell (1 bit), whether to move the tape head left or right (1 bit), and what new state (or special halt state) to transition to ($\lceil \log_2(n+1) \rceil$ bits). This encoding takes $6 \times 2 \times (2+3) = 60$ bits for a a 6-state TM, and $7 \times 2 \times (2+3) = 70$ bits for a a 7-state TM.

We're also stretching the meaning of "representable" a bit, since BB considers the runtime of the machine instead of its output. Besides the above BB (that Radó called S), Radó also defined another function called Σ that considers the output of the machine as a number in unary, i.e. the number of 1s in the final tape contents. But BB has received more attention as it allows one to determine from BB(n) all halting n-state machines. For 6-states and up though, there is no discernable difference in magnitude between the two functions so we could have just as easily used Σ.

So the largest number TM programmable in 64 bits is BB(6).

6. How Large Is BB(6)?

Unfortunately, we may never know. While BB(n) has been determined (and even formally proven) for all $n \leq 5$, there are some 6-state TMs whose halting behaviour are closely related to very hard mathematical problems. Most of these so-called cryptids are likely not to halt, while some, like one called "Lucy's Moonlight", are likely to halt but unlikely to beat the current champion.

 doi:10.20944/preprints202603.1897.v1

The current 6-state champion shows that $BB(6) > 2 \uparrow\uparrow 2 \uparrow\uparrow 2 \uparrow\uparrow 10$. Here, $m \uparrow\uparrow n$ is Knuth's up-arrow notation for an exponential tower of n m's, so that for example $2 \uparrow\uparrow 3 = 2^{2^2}$. Large as this number is, it's still very small compared to $\text{ack}(9,9) = 2 \uparrow^7 12 - 3 = 2 \uparrow\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow 12 - 3$.

It is known however that $BB(7) > 2 \uparrow^{11} 2 \uparrow^{11} 3 > \text{ack}(9,9)$. Several leading BB researchers believe that $BB(7)$ is even larger than the famous Graham's Number, which iterates the function $(n \mapsto 3 \uparrow^n 3)$ 64 times starting from $n = 4$. This is a rather bold belief, considering that the smallest known Graham exceeding TM has twice as many states (14). Bold enough for me to offer a $1000 bet that a proof of BB(7) > Graham's Number won't be found within 10 years, which leading BB researcher Shawn Ligocki was happy to accept.

Meanwhile, Graham's Number is easily surpassed within 64 bits, by moving beyond Turing machines into the language of

7. Lambda Calculus

Alonzo Church conceived the λ-calculus circa 1928 as a formal logic system for expressing computation based on function abstraction and application using variable binding and substitution.

The Graham beating lambda term originates in a Code Golf [4] challenge asking for the "Shortest terminating program whose output size exceeds Graham's number", answered by user Patcail and further optimized by user 2014MELO03. The following 49 bit program

$$01\ 00\ 01\ 10\ 10\ 00\ 01\ 10\ 01\ 10\ 00\ 00\ 01\ 01\ 10\ 110\ 00\ 00\ 01\ 110\ 01\ 110\ 10$$

is the Binary Lambda Calculus [3] encoding of the term

$$(\lambda 11)(\lambda 1(1(\lambda\lambda 12(\lambda\lambda 2(21)))))$$

where λ (lambda) denotes an anonymous function, and number i is the variable bound by the i-th nested λ. This is known as De Bruijn notation, a way to avoid naming variables. A more conventional notation using variable names would be

$$(\lambda J.J\ J)(\lambda y.y\ (y\ (\lambda g\lambda m.m\ g\ (\lambda f\lambda x.f\ (f\ x)))))$$

The top left illustration depicts it as a so-called lambda diagram. The last 16 bits of the program, making up almost a third of its size, encodes the term $\lambda f\lambda x.f\ (f\ x)$, which takes arguments f and x in turn, and iterates f twice on x. In its generalized form, the function $\lambda f\lambda x.f^n x$, called Church numeral n, is the most common way of representing natural numbers in the λ-calculus. The encoding of Church numeral n is $0000(01110)^n 10$, of size $5n + 6$ bits. The program, which we'll name after its discoverer, can be expressed more legibly as

$$\text{Melo} = \text{let}\ \{\ 2 = \lambda f\lambda x.f\ (f\ x);\ H = \lambda g\lambda m.m\ g\ 2;\ J = \lambda y.y\ (y\ H)\}\ \text{in}\ J\ J$$

Melo evaluates to a Church numeral, "Melo's Number", that comfortably exceeds Graham's Number, as we'll prove after 3 supporting Lemmas.

Lemma 1. $J\ J = 2 \uparrow\uparrow 6\ (HH)\ 2$.

 doi:10.20944/preprints202603.1897.v1

Proof.

$$\begin{aligned}
J\,J &= J\,(J\,H) = J\,(H\,(HH)) \\
&= H\,(HH)\,(H\,(HH)\,H) \\
&= H\,(HH)\,H && (HH)\,2 \\
&= H\,(HH)\,2 && (HH)\,2 \\
&= 2\,(HH)\,2 && (HH)\,2 \\
&= (HH)\,(HH\,2) && (HH)\,2 \\
&= HH\,2 && H\,2\,(HH)\,2 \\
&= 2\,H\,2 && H\,2\,(HH)\,2 \\
&= H\,(H\,2)\,H && 2\,(HH)\,2 \\
&= H\,(H\,2)\,2 && 2\,(HH)\,2 \\
&= 2\,(H\,2)\,2 && 2\,(HH)\,2 \\
&= H\,2\,(H\,2\,2) && 2\,(HH)\,2 \\
&= H\,2\,2 && 2\,2\,2\,(HH)\,2 \\
&= 2\,2\,2 && 2\,2\,2\,(HH)\,2 \\
&= 2 \uparrow\uparrow 6\,(HH)\,2
\end{aligned}$$

□

Lemma 2. *For $k, n \geq 2$, $k\ H\ 2\ n > 3 \uparrow^k (1+n)$.*

Proof. By induction on k.

Base: Since $H\ 2\ (2 \uparrow\uparrow i) = (2 \uparrow\uparrow i)\ 2\ 2 = 2^{2^{2\uparrow\uparrow i}} = 2 \uparrow\uparrow (i+2)$,

$2\ H\ 2\ n = H\ (H2)\ n = n\ (H2)\ 2 = 2 \uparrow\uparrow (1+2n) > 3 \uparrow^2 (1+n)$ already at $n = 2$, since $2 \uparrow\uparrow 5 = 2^{2^{16}} > 3^{27} = 3 \uparrow\uparrow 3$.

Step: $(k+1)\ H\ 2\ n = H\ (k\ H\ 2)\ n = n\ (k\ H\ 2)\ 2 \overset{Lm\ 2}{>} 3 \uparrow^k (1+\ 3 \uparrow^k (1+\ldots 3 \uparrow^k (1+2)\ldots)) > 3 \uparrow^k 3 \uparrow^k \ldots 3 \uparrow^k 3 = 3 \uparrow^{k+1} (1+n)$. □

Lemma 3. *For $n \geq 2$, $HH(HH\ n) > 3 \uparrow^n 3$.*

Proof. Lemma 1's proof shows $HH(HH\ 2) = 2 \uparrow\uparrow 6 > 3 \uparrow^2 3$. Now suppose $m = n-1 \geq 2$. Then $HH\ (HH\ n) = HH\ n\ H\ 2 = (1+m)\ H\ 2\ H\ 2 = H\ (m\ H\ 2)\ H\ 2 = H\ (m\ H\ 2)\ 2\ 2 = 2\ (m\ H\ 2)\ 2\ 2 = m\ H\ 2\ (m\ H\ 2\ 2)\ 2 \overset{Lm\ 2}{>} 3 \uparrow^m (1+3 \uparrow^m (1+2))\ 2 > (3 \uparrow^m 3 \uparrow^m 3)\ 2 = 2^{3\uparrow^n 3} > 3 \uparrow^n 3$. □

Theorem 1. *Melo > Graham's Number $G(64)$, where $G(n) = n(n \mapsto 3 \uparrow^n 3)4$.*

Proof. Melo $= J\ J \overset{Lm\ 1}{=} 2 \uparrow\uparrow 6\ (HH)\ 2 \overset{Lm\ 3}{>} (2 \uparrow\uparrow 6/2-1)\ (n \mapsto 3 \uparrow^n 3)\ (3 \uparrow^2 3) > (2 \uparrow\uparrow 6/2-1)\ (n \mapsto 3 \uparrow^n 3)\ 4 = G(2 \uparrow\uparrow 6/2-1) > G(64)$. □

8. Leaving Melo's Number in the Dust

With $64 - 49 = 15$ bits to spare, opportunities for vastly boosting Melo abound. Discord users 50_ft_lock and Sam found the following term that extends Melo's H with an extra argument:

$$w218 = \text{let } \{\ 2 = \lambda f \lambda x.f\ (f\ x);\ A = \lambda a \lambda b \lambda c.c\ a\ b\ 2;\ T = \lambda y.y\ (y\ A)\ \} \text{ in } T\ T\ T$$

which desugars to lambda term

$$(\lambda T.T\ T\ T)(\lambda y.y\ (y\ (\lambda a \lambda b \lambda c.c\ a\ b\ (\lambda f \lambda x.f\ (f\ x)))))$$

in conventional notation, or

$$(\lambda\ 1\ 1\ 1)\ (\lambda\ 1\ (1\ (\lambda\lambda\lambda\ 1\ 3\ 2\ (\lambda\lambda\ 2\ (2\ 1)))))$$

 doi:10.20944/preprints202603.1897.v1

in de Bruijn notation, with 61-bit encoding

01 00 01 01 10 10 10 00 01 10 01 10 00 00 00 01 01 01 10 1110 110 00 00 01 110 01 110 10

This term is depicted on the top right. Let's first establish an analogue of Lemma 1:

Lemma 4. $T\ T\ T = 2 \uparrow\uparrow 18\ A\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2.$

Proof.

$T\ T$	T	
$=$	$T\ (T\ A)$	T
$=$	$T\ (A\ (AA))$	T
$=$	$A\ (AA)\ (A\ (AA)\ A)\ T$	
$=$	$T\ (AA)$	$(A(AA)A)2$
$=$	$AA\ (AA\ A)\ (A\ (AA)\ A)$	2
$=$	$A\ (AA)\ A\ A$	$(AAA)22$
$=$	$A\ (AA)\ A\ 2$	$(AAA)22$
$=$	$2\ (AA)\ A$	$2(AAA)22$
$=$	$AA\ (AA\ A)\ 2$	$(AAA)22$
$=$	$2A\ (AA\ A)$	$2(AAA)22$
$=$	$A\ (A\ (AA\ A))\ 2\ (AA\ A)$	22
$=$	$AA\ A\ (A\ (AA\ A))$	2222
$=$	$A\ (AA\ A)\ A\ A$	22222
$=$	$A\ (AA\ A)\ A\ 2$	22222
$=$	$2\ (AA\ A)\ A$	222222
$=$	$AA\ A\ (AA\ A\ A)$	222222
$=$	$AA\ A\ A$	$AA2222222$
$=$	$A\ A\ A\ 2$	$AA2222222$
$=$	$2\ A\ A$	$2AA2222222$
$=$	$A\ (AA)\ 2\ A$	$A2222222$
$=$	$A\ (AA)\ 2\ 2$	$A2222222$
$=$	$2\ (AA)\ 2$	$2A2222222$
$=$	$AA\ (AA\ 2)\ 2$	$A2222222$
$=$	$2\ A\ (AA\ 2)$	$2A2222222$
$=$	$A\ (A\ (AA\ 2))\ 2\ A$	2222222
$=$	$A\ (A\ (AA\ 2))\ 2\ 2$	2222222
$=$	$2\ (A\ (AA\ 2))\ 2$	22222222

 doi:10.20944/preprints202603.1897.v1

=	$A\ (AA\ 2)\ (A\ (AA\ 2)\ 2)\ 2$	2222222
=	$2\ (AA\ 2)\ (A\ (AA\ 2)\ 2)$	22222222
=	$AA\ 2\ (AA\ 2\ (A\ (AA\ 2)\ 2))$	22222222
=	$AA\ 2\ (A\ (AA\ 2)\ 2)$	$A2222222222$
=	$A\ (AA\ 2)\ 2\ A$	$22A2222222222$
=	$A\ (AA\ 2)\ 2\ 2$	$22A2222222222$
=	$2\ (AA\ 2)\ 2$	$222A2222222222$
=	$AA\ 2\ (AA\ 2\ 2)$	$222A2222222222$
=	$AA\ 2\ 2$	$A22222A2222222222$
=	$2\ A\ 2$	$2A22222A2222222222$
=	$A\ (A\ 2)\ 2\ A$	$22222A2222222222$
=	$A\ (A\ 2)\ 2\ 2$	$22222A2222222222$
=	$2\ (A\ 2)\ 2$	$222222A2222222222$
=	$A\ 2\ (A\ 2\ 2)\ 2$	$22222A2222222222$
=	$2\ 2\ (A\ 2\ 2)$	$222222A2222222222$
=	$4\ (A\ 2\ 2)\ 2$	$22222A2222222222$
=	$A\ 2\ 2\ (A\ 2\ 2\ (A\ 2\ 2\ (A\ 2\ 2\ 2)))$	$22222A2222222222$
=	$A\ 2\ 2\ (A\ 2\ 2\ (A\ 2\ 2\ 2))$	$22222222A2222222222$
=	$A\ 2\ 2\ (A\ 2\ 2\ 2)$	$2222222222A2222222222$
=	$A\ 2\ 2\ 2$	$22222222222222A2222222222$
=	$2\ 2\ 2\ 2$	$22222222222222A2222222222$
=	$2 \uparrow\uparrow 18\ A\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2$	

□

These $2 \uparrow\uparrow 18$ iterations of A let us relate its magnitude to the Fast Growing Hierarchy, a family of ordinal indexed functions $[\alpha]$ from natural numbers to natural numbers.

Definition 1. *The (slightly accelerated) Fast Growing Hierarchy (FGH) below ω^ω, mapping Church numerals to Church numerals, is defined as*

- $[0]\ n = 2\ n = n^2$
- $[\alpha + 1]\ n = n\ 2\ [\alpha]\ 2 = A\ 2\ [\alpha]\ n$
- $[\omega^{i+1}(\alpha + 1)]\ n = [\omega^{i+1}\alpha + \omega^i n]\ 2$

We deviated from the standard FGH's $[0]n = n + 1$ and $[\alpha + 1]\ n = n\ [\alpha]\ n$ to make the following Lemma exact rather than a mere lower bound.

Lemma 5. *For $k \geq 0, n \geq 2 : (k + 1)\ A\ 2\ [\omega^k \alpha]\ n = [\omega^k(\alpha + 1)]\ n$*

Proof. Base $k = 0$: $(0 + 1)\ A\ 2\ [\omega^0\alpha]\ n = A\ 2\ [\alpha]\ n = n\ 2\ [\alpha]\ 2 = [\alpha + 1]\ n$.

Step $k > 0$: $(k + 1)\ A\ 2\ [\omega^k\alpha]\ n = A\ (k\ A\ 2)\ [\omega^k\alpha]\ n = n\ (k\ A\ 2)\ [\omega^k\alpha]\ 2 = [\omega^k\alpha + \omega^{k-1}n]\ 2 = [\omega^k(\alpha + 1)]\ n$. □

Lemmas 4,5 give $w218 = 2 \uparrow\uparrow 18\ A\ 2\ [0]\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2 = 2^{2^{2^{2^{2^{([\omega^{2\uparrow\uparrow 18-1}]\ 2)}}}}}$. In comparison, Graham's and Melo's Numbers are known to be much smaller at around $[\omega + 1]\ 64$ and $[\omega + 1](2 \uparrow\uparrow 6)$ respectively.

9. A Functional Busy Beaver

The λ-calculus analogue to BB is:

Definition 2. *$BB\lambda(n)$ = the maximum beta normal form size of any closed lambda term of size n*

 doi:10.20944/preprints202603.1897.v1

which appears in the Online Encyclopedia of Integer Sequences (OEIS) as sequence A333479. Besides being simpler than BB, it has the advantage of using the standard unit of information theory, bits rather than states.

The much more fine-grained use of bits allows the first 36 values of BBλ to be currently known, versus only 5 values of BB.

As a Church numeral, term Melo implies that BB$\lambda(49) \geq 5(\text{Melo's Number}) + 6$, and w218 implies a similar lower bound on BB$\lambda(61)$.

10. BB Compared Bit-by-Bit to BBλ

The growth rates of the two BB functions may be compared by how quickly they are known to exceed certain large number milestones, that correspond to well known ordinals in the Fast Growing Hierarchy. Lambda terms for the sizes given below can be found at the bbchallenge webpage [5].

For Graham's Number, at ordinal $\omega + 1$, we saw earlier that Melo's 49 bits compares with 14 states, which take $14 \times 2 \times (2 + 4) = 168$ bits to encode. If I lose my bet, then the comparison becomes rather closer at 49 vs 70 bits.

For Goodstein's function, at ordinal ϵ_0, 111 bits [6] compares with 51 states [7] taking $51 \times 2 \times (2 + 6) = 816$ bits.

The ϵ_0 growth term is actually obsoleted by a 100-bit [8] term recently discovered by Patcail that grows at the unfathomably larger Bucholz' Ordinal, the catching point between the SGH and the FGH. As such, that term easily exceeds another famously large number, TREE(3).

For the limit of Bashicu Matrix System (BMS), at (presumed) ordinal PTO(Z_2), 331 bits [9] compares with 150 states [7] taking $150 \times 2 \times (2 + 8) = 3000$ bits.

Finally, for Loader's Number, at (presumed) ordinal PTO(Z_ω), 1850 bits [10] compares with 1015 states [11] taking $1015 \times 2 \times (2 + 10) = 24360$ bits.

One reason for TMs taking many more bits to achieve comparable growth, especially at the larger milestones, is the extremely poor programmability of TMs. The λ-calculus, despite its similar bare bones nature, doesn't share this drawback. Modern high level pure functional languages like Haskell are essentially just syntactically sugared λ-calculus, with programmer friendly features like Algebraic Data Types translating directly through Scott encodings. The bruijn programming language is an even thinner layer of syntactic sugar for the pure untyped lambda calculus, whose extensive standard library contains many datatypes and functions. It is this excellent programmability of the λ-calculus that facilitated the construction of highly optimized programs for BMS and Loader.

Because programming a Turing machine is so impossibly tedious, people have resorted to implementing higher level languages like Not-Quite-Laconic for writing nontrivial programs, such as the TM that halts only upon finding an inconstency in ZFC. The above 1015 state TM for exceeding Loader's Number even includes a λ-calculus interpreter written in NQL.

11. But Why Turing Machines?

In his paper [2] "The Busy Beaver Frontier", Scott Aaronson poses and tries to answer this question:

> For all their historic importance, haven't Turing machines been completely superseded by better alternatives—whether stylized assembly languages or various codegolf languages or Lisp? As we'll see, there is a reason why Turing machines were a slightly unfortunate choice for the Busy Beaver game: namely, the loss incurred when we encode a state transition table by a string of bits or vice versa.

He continues to point out two supposedly massive advantages that TMs have over alternatives, which we can assess for the case of the λ-calculus.

11.1. Interesting Behaviour at Small Sizes

Aaronson continues:

 doi:10.20944/preprints202603.1897.v1

> because Turing machines have no "syntax" to speak of, but only graph structure, we immediately start seeing interesting behavior even with machines of only 3, 4, or 5 states, which are feasible to enumerate.

The number of uniquely behaving TMs with 5 states is $4^1 0 \times 632700 = 663434035200$ (see OEIS sequence A107668), which is more than the number of closed lambda terms of size at most 52 bits (513217604750). The latter exhibits just as much interesting behaviour, so TMs hold little advantage here.

11.2. Ancient and Fixed Computational Model

Aaronson ends his answer with::

> because the Turing machine model is so ancient and fixed, whatever emergent behavior we find in the Busy Beaver game, there can be no suspicion that we cheated by changing the model until we got the results we wanted.

The λ-calculus is slightly more ancient and arguably more fixed, without any of the TM's choices: tape alphabet size, whether the tape head moves in every transition, halting and output convention, number of tapes or tape heads. Both TMs and the λ-calculus are maximally non-cheating, so again TMs hold little advantage here.

The only remaining advantage of BB over BBλ is the many decades of research behind, and publications about it.

12. An Optimal Busy Beaver

Is BBλ then an ideal Busy Beaver function? Not quite. It's still lacking one desirable property, namely optimality.

This property mirrors a notion of optimality for shortest description lengths, where it's known as the Invariance theorem: an optimal language L_{opt} is at least as efficient as any other description language L, with at most constant additive overhead in shortest description length.

In the realm of beavers, an optimal Busy Beaver BB_{opt} should surpass any Busy Beaver function bb (based on self-delimiting programs) with at most constant lag: for some constant c depending on bb, and for all $n : BB_{opt}(n + c) \geq bb(n)$.

Busy beavers are closely related to shortest description lengths. Since the largest output of an program of at most n bits is also the largest object whose description length is at most n bits, they are like an inverse of the Kolmogorov Complexity.

While BBλ is not optimal, the closely related

Definition 3. *$BB\lambda 2)(n)$ = the maximum output size of self-delimiting BLC programs of size n*

(appearing as sequence A361211 in the OEIS) achieves optimality by giving λ-calculus terms access to pure binary data. BLC programs consist of an encoded lambda term, followed by arbitrary binary data, that the term is applied to.

Since $(\lambda\_.t)$ applied to any (standard lambda representation of) binary data equals t, BBλ champions provide lower bounds for BBλ2: for all n, $\text{BB}\lambda 2(2 + n) \geq \text{BB}\lambda(n)$.

In BBλ2 and BB$\lambda(n)$, we thus have a moderately simple optimal Busy Beaver function and a really simple one that close approximates it for all the smaller values of n that we can hope to analyze.

13. Conclusion

The largest number (currently known to be) representable in 64 bits without cheating is w218, which lower bounds both $\text{BB}\lambda(61)$ and $\text{BB}\lambda 2(63)$.

References

1. T. Radó, On non-computable functions, *Bell System Technical Journal* **1962** *3*, 877-–884.

 doi:10.20944/preprints202603.1897.v1

2. Scott Aaronson, The Busy Beaver Frontier. *SIGACT News* **2020**, *3*, 32—54.
3. John Tromp, Binary Lambda Calculus and Combinatory Logic. In *Randomness And Complexity, from Leibniz To Chaitin*; Ed. Cristian S. Calude; World Scientific Publishing Company, 2008; pp. 237–260.
4. Code Golf. Available online: https://codegolf.stackexchange.com/ (accessed on 14 March 2026).
5. Busy Beaver for lambda calculus. Available online: https://wiki.bbchallenge.org/wiki/Busy_Beaver_for_lambda_calculus (accessed on 14 March 2026).
6. BBE0.lam. Available online: https://github.com/tromp/AIT/blob/master/fast_growing_and_conjectures/BBE0.lam (accessed on 14 March 2026).
7. Champions Available online: https://wiki.bbchallenge.org/wiki/Champions (accessed on 14 March 2026).
8. BO.lam. Available online: https://github.com/tromp/AIT/blob/master/fast_growing_and_conjectures/BO.lam (accessed on 14 March 2026).
9. bms.lam. Available online: https://github.com/tromp/AIT/blob/master/fast_growing_and_conjectures/bms.lam (accessed on 14 March 2026).
10. Binary Lambda Calculus: 1850 bits Available online: https://codegolf.stackexchange.com/questions/176966/golf-a-number-bigger-than-loaders-number/274634#274634 (accessed on 14 March 2026).
11. metamath-turing-machines. Available online: https://github.com/CatsAreFluffy/metamath-turing-machines/tree/master (accessed on 14 March 2026).