# Transrecursive Theory of Computable Growth (TRT)

Author: **Artur Mataryan**, 11.10.2025

**Announcement**

This work presents, for the first time, a formal theory that…:

* builds a mathematically strict horizon between computable and non-computable functions.
* Formalizes a continuous analog of a Fast-Growing Hierarchy.
* enters a metric for the ordinal complexity of numbers and functions.
* shows that exponentials, iterations, and ordinal reflection form a single principle of superposition growth.
* constructs the limiting class of functions TRANSCEND as having the property of the maximum possible growth rate among all computable functions, completing the search for "the fastest growing computable hierarchy" in modern googology.
* forms a new idea of the limits of constructive mathematics and the "speed of light" in the world of computable things.

**Article content**

1. Introduction to the problem. What it is and why.
2. From HCCSF to TRANSCEND — step-by-step construction of a transrecursive growth class.
3. Formulation, proof, and consequences of a transrecursive theory.
4. Comparison of TRANSCEND with other notations and its uniqueness.
5. The essence of the breakthrough: Moving from quantitative growth to ordinal growth
6. Final reflection, scientific novelty and significance

## Introduction to the problem. What it is and why.

Let's start a little bit from afar. When mathematicians talk about large numbers or fast-growing functions, it is often very inconvenient to visually and clearly represent quantities of units, tens, hundreds, and, say, millions or billions on a single graph, for obvious reasons that hundreds and thousands will merge into a "pixel" at the very beginning of the graph, and the remaining values will be close to the mentioned billion. Such a representation offers neither clarity nor practical interpretability. And, in general, for a long time and successfully mathematicians, and not only they, but everyone who is at least somehow connected with large quantities, use logarithmic scales to display both fairly large numbers and very modest ones on one numerical coordinate axis. That is, we say, yes, a linear scale for really large numbers, it is inconvenient, so let's compress this scale as the numbers grow. It works, it's clear. A good example is the image of a logarithmic universe. This is a common and correct approach. When the usual logarithmic scale is not enough, you can take the iterative logarithm. So you can still clearly and conveniently display the usual "human" millions and googolplexes, Ackerman functions, numbers of the form 3 3 3. But the number of Graham's number G you already have in the form of iterative logarithms is no longer represented, no matter how many you take, even if the whole googolplex of these iterative logarithms. Why? Because the scale order of the Graham number is incomparable to the scale order that iterative logarithms can provide. What then is to be done?

Let's take a closer look at the logarithmic scale (regular or iterative, it doesn't matter) and explore what we get. In the case of a linear scale, the scale remains unchanged for all parts of the scale. This is obvious. In the case of a logarithmic scale, it becomes logarithmic from the growth of the scale, but still remains constant, in a sense "linear" in the context of scale growth along the scale — the same logarithmic growth in the region of 1, that in the region of 10 ^ 100. Therefore, it is not surprising that the logarithmic scale, if we approach really large numbers in the mathematical sense, is beloved and uninteresting for us. What can we do to move forward? The breakthrough idea is to constantly compress the scale itself, and the amount of scale compression should also constantly grow. This is the first factor — the continuous growth of the scale itself. What does compressed scale mean in the context of number theory, computability theory? In fact, we have come close to the hierarchical ordinal complexity of numbers.

The second factor is discrete, designated ordinal levels of scale (hierarchical computational complexity). Let's say our goal is to develop a scale (it would be more appropriate to write - functional mapping), which will not mix everything in one pile-ordinary numbers, astronomical, exponential functions, double and triple exponentials, hyperoperators, and so on, but do it structurally, preserving the ordinal complexity of the number level. After all, there is nothing easier than to take the entire numeric axis and map it to a segment of unit length according to the 1/x principle for x>=1. Yes, you will get a continuous, monotonic function that will "overcome" both the Graham number and the TREE(3) number, and any finite and transfinite numbers, but it is essentially trivial, and mathematically does not give anything interesting. As the value of the numbers in the denominator increases, the value of the function will get closer and closer to 0, in general.

A function that successfully solves both of these tasks has been successfully developed and I want to introduce it to you. HCCSF (Hierarchical Computable Complexity Scale Function) - Scaling function for hierarchical computable complexity.

Definition: HCCSF is a continuous, monotonic functionthat maps the "magnitude of a number" to its hierarchical computational complexity level, ℝ ₊→ ℝ ₊. We define the coordinate x = n + y, where n ∈ {N} is the level (integer part), and y ∈ [0,1) is the position inside the level. The unit intervals [n, n+1) correspond to discrete hyperoperative growth levels. Exponential level generators are defined as follows:

T0 (y) := 1/(1 - y)

T1 (y) := e^(T0(y)} = e1/(1 - y)

T2 (y) := e^(T1(y)) = e^(e1/(1 - y))

T3 (y) := eT2(y)

Tn (y) := eTn-1(y)

Compression for continuity

To combine all levels into a smooth scale, we introduce a smoothing squash function: S(t) = t/(1 + t) . It smoothly translates t∈[0,∞) to [0,1), preserving the order. The final definition of the HCCSF function looks like this: HCCSF (x)=n+(S(Tn(y))−S(Tn(0)) / (1-S(Tn(0))), x=n+y, y∈[0,1). Let's explore this feature carefully.

#### How many difficulty levels are there on this scale?

The short answer is that there are infinitely many levels. In the HCCSF construction, the level is given by the integer part (n=⌊x⌋), where n∈N. This means that the levels are indexed by the natural numbers (0,1,2,...) — they are exactly ℵ0 (countably infinite).

What does this mean in practice?

* Level (0): Linear / polynomial numbers (1, 10, 100, ...).
* Level (1): exponents ((2^n, 10^n)).
* Level (2): double exponent 2^2^n, 3!!3.
* Level (3): triple exponent, etc.

Each subsequent integer level corresponds to "one more iteration" of the exponent (or an increased hyperoperation rank), and there can be as many such iterations as you like.

If you want to formally continue beyond natural indices, HCCSF can be extended to ordinals. Then, within the framework of a particular formal theory (for example, PA, ZFC, etc.), you will get levels up to those ordinals that are representable in the notation of this theory; If you try to take "all ordinals", then such levels will no longer be many, but the correct class (that is, formally even more "many"), and this is a model-dependent construct.

Table of examples of correspondences between numbers and values of the HCCSF (x) function.

|  |  |
| --- | --- |
| Number | Approximate position on HCCSF |
| 1 | 0.0 |
| 10 | 0.9 |
| 101⁰⁰ | 0.999 |
| Googolplex | 1.999 |
| 3↑↑3 | 2.9 |
| 3↑↑4 | 2.999 |
| 3↑↑↑3 | 2.999999 |
| g₁ | 3.999999 |
| g₆₄ | 3.999 ... |
| TREE (3) \* | 4.5 |
| SCG (13) \* | 5.2 |
| Loader's Number \* | 6.8 |
| Σ (1000) \* | 8.3 |

\* The values after TREE (3) and SCG(13) are symbolic and illustrate relative orders rather than exact computable positions, since HCCSF approximates complexity levels rather than absolute values.

#### Analytical properties of the function

1. Monotonicity: a<b ⇒ HCCSF(a) < HCCSF(b)
2. Continuity: Transitions between levels are smooth, without breaks.
3. Growth compression: The difference between 2.9 and 2.99-as between "3 3 3" and "3 ↑ ↑ 4" — is a huge qualitative leap.
4. Invertibility: It is possible to approximate x≈HCCSF -1(L) to estimate which number corresponds to the complexity level L..

#### Interpretation and philosophy of the function

HCCSF, if you think about it, essentially describes not the numbers themselves, but their place in the growth ontology. It shows which floor of infinity a given number is located on, where new computability classes begin, and where formal mathematics ends (for example, near TREE(3)). Ontologically, HCCSF is a metric of the meta-universe of numbers, in which "distance" reflects not length, but structural power.

Among the practical applications of HCCSF, its direct domain is googology in the context of classification of super-large numbers, as well as computability theory and Limits of constructive systems, and, in all likelihood, it will be useful for AI research in the field of measuring the cognitive depth of models. Another interesting field of application is the philosophy of mathematics (analysis of transfinite knowledge levels)

What about other fast-growing hierarchies? Comparison with the classic fast-growing hierarchy.

Let me remind the reader what a Fast-Growing Hierarchy (FGH) is. FGH is a class of functions fa:N→N indexed by alpha ordinals < varepsilon0 (or later in extended versions). Basic:

* f0(n) = n+1
* fa+1(n) = fan(n) — (n)-multiple iteration of the previous level.
* For a limit lambda: flambda(n) = flambda[n](n), where lambda[n] is the fundamental sequence.

In other words, FGH describes a hierarchy of functions by growth rate, where each level corresponds to an increasingly powerful "meta-component". It exhausts everything that can be expressed in terms of arithmetic ordinals (up to varepsilon0, R0, etc.). Below **is a visual comparison table for both hierarchies**

|  |  |  |
| --- | --- | --- |
| Parameter | FGH | HCCSF |
| Base | Ordinal recursion | Continuous parametrization of growth levels |
| Nature | Discrete (indices-ordinals) | Continuous (real axis, fractional levels) |
| Result | Function(fa(n)) → integers | Function(HCCSF (x)) → continuous scale |
| Scope | Logic, provability, proof theory | Googology, philosophy, numerical ontology |
| Meaning | "How fast does the function grow" | "How high is the level of numerical complexity" |
| Inter-level | No intermediate values (only ordinals) | There are fractional levels — "between tetration and pentation" |
| Geometry | No-purely recursive definition | Yes-scale with metric and visualization |

#### Conceptually: FGH is "mechanics", HCCSF is "geometry"

We can say that FGH describes “growth mechanics” and HCCSF describes "growth geometry". If FGH answers the question " how fast does the function grow at the level of α?", then HCCSF answers the question"Where on the continuum of numerical complexity is this value located?". FGH is a tool for analyzing algorithms and provability, while HCCSF is a tool for analyzing ontological scales of numbers. FGH remains in discrete logic (hierarchy of functions), and HCCSF embeds this discreteness in a continuous topological axis — for the first time making it possible to “move smoothly” along ordinal levels. By revealing their relationship more deeply, we can show that there is an approximating map: HCCSF(x) ~ log2(fa(n)), where the fractional part (x - ⌊x⌋) corresponds to the logarithm between adjacent levels of FGH. That is, in essence, HCCSF = continualization of FGH through a smooth normalized function that gives the level coordinate as a real number. In summary, FGH is a "step ladder", HCCSF is a "smooth curve" passing through the same points.

#### What is the scientific novelty of HCCSF relative to FGH?

1. Continuity — a continuous analog of FGH is constructed for the first time.
2. Metrization-the concept of "distance between levels" appears, which is absent in FGH.
3. Ontological meaning-the scale reflects not only the growth of the function, but also the "scale of complexity" as an entity.
4. Versatility-Covers not only ordinal levels, but also computable/non-computable numbers, including Busy Beaver, SCG, etc.
5. Visualizability — you can build a map or diagram, which is impossible for FGH without losing structure.

What is the scientific novelty and value of HCCSF in general? It is the first to introduce a new type of metric for numbers that combines discrete hyperoperations and a continuous structure. The fact is that unlike existing approaches (Conventional systems (Knuth, Conway, BEAF, Bowers, Tetration, Fast-Growing Hierarchy) are discrete and do not allow measuring "between" levels, and logarithmic or power scales are too compressed for hyperoperator numbers, HCCSF offers continuous, monotonic, normalized a scalewhere you can smoothly switch between degree, tetration, pentation, and so on. So it is not just a new notation, but a continualized measure of hierarchical complexity. we can give an example of a successful metaphor that if FGH is a ladder to infinity-up steps, then HCCSF is a smooth ascent along the slope of infinity-without breaks.

#### Fractal properties of HCCSF

Now let's analyze whether this function has fractal properties. Can this function describe the increasing complexity of mathematical systems? Yes, but in a special, functional sense. This is not a geometric fractal like the Mandelbrot set, but rather a functional fractal or a fractal of self-similarity in behavior.

1. Self-similarity in the zoom limit:

Consider the behavior of F(x) at level n near y=1. We introduce a new variable ε = 1-y. For ε → 0⁺:

1. T₀(y) = 1/(1-y) = 1/ε
2. S(t₀(y)) = (1/ε) / (1 + 1/ε) = 1 / (1 + ε) ≈ 1 - ε (for small ε).

Now let's look at the level n=1. t₁(y) = e1/ε. This is a monstrous growth. However, if we look at 1 - S(t₁(y)), we get:

1. 1 - S(T₁(y)) = 1 - (e1/ε) / (1 + e1/ε)) = 1 / (1 + e1/ε) ≈ e-1/ε

There is a qualitative self-similarity: when moving to each next level n, the behavior of the function near the right boundary (y→1) reproduces, at a qualitatively new, higher level of complexity, the behavior of the smoothing function S from the even faster-growing function. The "near-constant - > sharp jump" structure is repeated at each level, but the" sharpness " of the jump grows tetrationally with each level.

Fractal properties are expressed as:

1. Recursive nature: Algorithm for calculating Tₙ(y)it is purely recursive. To understand behavior at level n, you need to understand behavior at level n-1. This nesting of definitions is a key feature of fractal-like structures.
2. Invariance under the shift and stretch operator: You can define the operator Φ, which acts on functions defined on [0,1). Φf(y) = [S(ef(y)) - S(ef (0))] / [1 - S(ef(0))]. Then for n ≥ 1, the fractional part of F(x) at level n is (Φg) (y), where g(y) is the fractional part at level n — 1. The behavior of the function at each subsequent level is the result of applying this nonlinear operator Φ to the behavior at the previous level. This is an analog of the scale transformation in geometric fractals.

Conclusion: The function has not a strict geometric, but an asymptotic and recursive self-similarity, which makes it an amazing object from the point of view of functional analysis.

The answer to the question " Can this function describe the growth of complexity levels of mathematical systems?" there will be a sure yes, and this is probably its main heuristic value.

Comparing hierarchies of fast-growing functions

Level 0 (n=0): t₀(y) ~ 1/(1-y). Similar to polynomial growth or growth as a power function. The level of "elementary" mathematics.

Level 1 (n=1): t₁(y) = et₀(y). This is exponential growth. The level of complexity of many combinatorial problems (for example, checking all subsets).

Level 2 (n=2): t₂(y) = exp (et₀(y)). This is a double exponent. It is typical for some problems in model theory and logic, where you need to check all models of a certain size.

Level 3 (n=3): t₃(y) is a triple exponent. This is the level of complexity associated with checking the truth of statements in Peano arithmetic (with some caveats).

Levels n>3: Go to the tetration region. This corresponds to levels of unrepresentable complexity that arise in set theory (for example, the cardinality of huge cardinals) or in proof theory (proof-theoretic ordinals).

The "breaking through the barrier" model: at each level n, the system "struggles" to solve problems of its own complexity. Progress within the level (y increases) initially seems slow and almost imperceptible (F(x) almost does not grow). The y - > 1 transition represents the accumulation of a critical mass of knowledge/tools. The moment x = n+1 (i.e. y=0 at the level of n+1) is a qualitative leap, a breakthrough that opens up a fundamentally new class of solvable problems and, importantly, a new class of problems that now seem "unsolvable" (since we are again at the beginning of a plateau). So we move from arithmetic to mathematical analysis (jump from level 0 to 1), and then to modern logic and computer science, which operate with the concepts of double and triple exponentials.

Theorem (HCCSF functional self-similarity):  
 For all n ≥ 0, HCCSF (n+y)=(nT0Φnt0) (y), where Φ is the functional scaling operator. Therefore, HCCSF has a recursive self-similarity similar to fractal, but in the space of functions, not figures.

Description of "incomprehensibility": The function perfectly models why numbers like e^(e10) or, even more so, e^(e^(e10)) are not just "large", but qualitatively different. They are located on different "floors" of the mathematical universe. For a system at level n=1, the number from level n=2 is not just larger — it is unattainable in principle within its paradigm.

#### What else is its mathematical uniqueness?

Parameterization of the Grzegorczyk/Fast-Growing Hierarchy: These hierarchies are usually defined for integer arguments. This function interpolates this hierarchy, smoothly filling in the gaps between entire" steps " of complexity. This gives a continuous analog of a discrete hierarchy.

Dual nature of growth: "External" growth (F(x) by x): Linear, calm, limited. "Internal" growth (tₙ(y) in y): Explosive, unlimited, tetrational. What is unique is that the function encapsulates a huge internal growth within finite intervals of the external coordinate. This is a form of mathematical compactification.

Bridge between finite and infinite: Each level of n is finite (F(x) < n+1), but to reach its end requires "infinite" effort in terms of growth tₙ(y). The function builds a bridge from finite x to qualitatively different types of infinity (different" levels " of infinity in the sense of growth rate).

Constructive Non-analyticity: As noted earlier, the function c⁰ is continuous, but not analytic at points x = n for n>=1. Its uniqueness lies in the fact that this non-analyticity is not simply "glued together" (like |x|), but is generated by a deep recursive procedure associated with the tetration hierarchy. This makes it a natural example of a smooth but non-analytic functionthat arises not from piecewise gluing, but from the fundamental principle of the complexity hierarchy.

The final image can be visualized as follows: Imagine an endless tower of floors. Each floor (n) has its own "universe". Floor 0: Our familiar universe with linear dimensions. Floor 1: A universe where distances follow an exponent. Floor 2: Double exponent universe, etc. Your x position is the floor number + position on the stairs between floors. The function F(x) is a kind of" universal meter " that is calibrated so that, being on the stairs between the 1st and 2nd floor, it shows you your position relative to the scale of the universe on the 1st floor. When you've almost reached the 2nd floor, the meter goes off the scale, because you start to perceive the scale of the 2nd floor universe. Thus, this function is not just a mathematical curiosity, but a deep conceptual tool for understanding hierarchies, complexity, and the process of cognition itself.

And now we go directly to the construction of the TRANSCEND function.

## From HCCSF to TRANSCEND-step-by-step construction of a transrecursive growth class

#### Introduction and transition idea

Let me remind you that HCCSF is a continuous monotone function that gives for each positive number a "complexity coordinate" on a single level scale (intervals ([n, n+1))). This scale allows you to compare the number with its ordinal level of growth: ordinary polynomial domain, exponential, double exponent, etc. The main idea of moving to TRANSCEND is to use HCCSF and its inverse function as meta-reflection tools: when calculating a new function value, we don't just apply arithmetic operations to the previous value, we:

1. we determine (via HCCSF-1) at what level of complexity the previous value "is".
2. building a new, enhanced exponential scale based on this level.
3. we repeat this procedure a finite but often extremely large number of times (the number of repetitions depends on the previous value).

As a result, we obtain a family of transrecursive functionsnTn(y) (and, accordingly, the class of functions TRANSCEND), which has a self-similar, recursively increasing growth dynamics.

#### 1. Parameterization and canonical scheme

#### Design parameters (canonical version)

We fix:

* the basic "compressed" position on the interval y0∈[0,1); by default, we take y0= e-1 (canonical choice, see arguments in the text);
* basic smoothing function S(t) = t/(1+t), S: [0,∞)->[0,1);
* HCCSF as in the previous sections: for x=n+y, n∈N, y∈[0,1)   
  HCCSF(x)=n+(S(nTn(y))−S(nTn (0)) / (1-S(nTn(0))), where T0(y)=1/(1-y), Tm+1(y) = exp(Tm(y)) — basic exponential level generators.

#### Generalized parameterization of the TRANSCEND class

We will consider TRANSCEND as a class of functions that depends on a number of (possible) parameters Φ (for example, the choice of (0y0), the choice of the function for counting the number of iterations g (⋅), the choice of a normalization scheme). Entry: T = {TRANSCENDΦ}.  
In the following, for the sake of brevity, we describe the "canonical" version with specific simple choices:0y0=e-1, g(u)=e eu⌋, normalization by (S).

#### 2. Defining META\_ITER and the main core

#### Definition 2.1 (META\_ITER operator)

For a given positive integer G and an integer k>=0, we define recursively  
  
META\_ITER(G, 0): = G, META\_ITER(G, k) = exp(HCCSF(HCCSF-1(META\_ITER(G, k-1)))), k≥1

Comment: at each step, we take the current number META\_ITER (⋅), translate it to the "difficulty level" via HCCSF-1, then return to the numerical scale via HCCSF and raise it to an exponent — thereby increasing the scale by another exponential "layer".

#### A note about the growth of META\_ITER

If G is already large, then META\_ITER(G,k) behaves like exp(k)by a non-strict evaluation(c ⋅G)  
(here exp(k) is the composition of exp k times, and c>0 is some coefficient of the order of one that depends on normalization). This estimate provides an intuition about the hyperexponential nature of the operator.

#### 3. Basic recursive definition of TRANSCEND (canonical version)

#### Definition 3.1 (canonical scheme (Tn(y)))

For a fixed y∈[0,1), we set recursively:  
  
T0(y):= 1/(1-y), and for (n >=1), Tn(y): = exp((META\_ITER(Tn-1(y), ⌊eTn−1(y)⌋)) Tn-1(y))

In particular, the canonical value of TRANSCEND for the " level argument "(n) is usually denoted TRANSCEND(n): = Tn(y0), y0=e-1. Comment on the formula structure:

* the number of repetitions of the meta-iteration is k=e eTn-1(y)⌋ — this is a finite, but often extremely large integer.
* inside META\_ITER, at each repetition, the number is "transferred to the level" (via HCCSF-1 and "returned to the numeric scale" (via HCCSF), followed by exponentiation.
* the resulting construction raises the META\_ITER term to the powertn-1(y) and then takes another exponent: this adds an additional layer of hypergrowth compared to the simple iteration exp ex exp.

#### 4. Lemmas on correctness, computability, monotonicity, and continuity

#### Lemma 4.1 (finiteness of iterations and correctness of the definition)

For any finite n and any y∈[0,1), the value of Tn(y) is defined and unambiguous: all internal iterations of META\_ITER with integer (k) consist of a finite number of steps.

Proof. By construction k=e eTn-1 (y)⌋ is an integer and nite for nite (Tn-1(y)). Therefore, META\_ITER performs exactly (k) recursion steps. Induction by (n) completes the proof.

#### Lemma 4.2 (computability of individual values)

For any finite (n) and rational (y), the valuen(Tn (y)) is computable with a given accuracy.

Key arguments:

1. T00(y) is an elementary expressible function.
2. HCCSF and HCCSF-1 are strictly monotone and computable on rationals (the inverse is calculated by bisection/iteration in a finite number of steps with any required accuracy, since HCCSF is monotone).
3. All operations in the formula (exponent, integer part, compositions, finite loops) are algorithmically feasible.

Therefore, for a fixed (n) and the required accuracy, an algorithm can be implemented that calculatesnTn(y).

#### Lemma 4.3 (monotonicity with respect to y and with respect to n)

1. For a fixed n, the function y T Tn(y) strictly increases by [0,1).
2. For a fixed y, the sequence n ↦nTn(y) is strictly increasing.

The idea of proof. Induction by (n).  
 Base: T0(y)=1/(1-y) - strictly increases. Transition: The META\_ITER operator is a composition of strictly increasing functions (HCCSF, inverse ofHCCSF-1, exp), so it preserves the order by the input argument (G). Raising a monotonically increasing positive number to a positive power and then the exponent preserves strict ascending order.

#### Lemma 4.4 (continuity with respect to y)

For a fixed n, the functionnTn (y) is continuous on ([0,1)).

The idea of proof. All comma operations (HCCSF, HCCSF-1, exp, finite compositions and exponentiation) are continuous, so the composition of continuous maps gives continuity on the working interval. At the junctions of the levels (x=n), HCCSF gives smooth interpolation, so there are no gaps. ∎

#### 5. Calculation algorithm (pseudocode)

Below is an algorithm for calculatingnTn(y) for fixed finite n and y up to a given precision (pseudocode scheme).

function compute\_T(n, y, eps):  
 / / input: n (natural), y in [0,1), eps — required precision  
 if n == 0:  
return 1/(1 - y)  
G\_prev = compute\_T(n-1, y, eps1) / / eps1-specify by error requirement  
k = floor(exp(G\_prev))  
M = META = G\_prev  
for i in 1..k:  
 / / calculate HCCSF^{-1}(META) with the specified accuracy (bisection method)  
 z = invert\_HCCSF(META, precision)  
 / / then HCCSF(z) (direct calculation)  
 M = exp( HCCSF(z))  
end for  
/ / final assembly  
return exp( M ^ G\_prev) / / calculate accurately using logarithms for large numbers

Comment: in practice, numbers become huge and require special representations (but algorithmically everything is finite).

#### 6. Growth estimates and lower bounds

#### Informal lower score

For simplicity, let HCCSF(HCCSF-1(u)) ≳c⋅u be assumed for large u and some c>0. Then  
META\_ITER(G, k)≳exp(k) (c⋅G). Taking into account the fact that k ~ eG, we obtain a rough, non-strict lower bound:nTn(y)≳exp((exp(⌊eTn−1(y)⌋) (c⋅nTn−1(y)))Tn-1(y)), which illustrates the extremely rapid (superhyperexponential) growth acceleration.

Conclusion: even at small n (for y close to 1), the valuesnof Tn(y) exceed all the classical fast-growing values presented in googology, with the exception of classes of non-computable constants (see note below).

#### 7. Status of the TRANSCEND class relative to other hierarchies

#### Theorem 7.1 (on class dominance within the computable)

Let {Comp} be the class of all total computable functions N→N. Then there is a family of parameterizations (Φα) such that   
∀f∈Comp ∃ Φα:∃N∀n>N: TRANSCEND\_Φα(n)>f(n).

Meaning and scheme of the proof. TRANSCEND is not a single fixed function, but a schema / class in which parameters can be set to "embed" any desired computable hierarchy depth (by choosing the rules for counting k, normalizations, initial0 y0, etc.). Therefore, the {T} class serves as the top cover for all computable tempos growth: for each fixed computable f, you can choose parameters that give faster growth for sufficiently large arguments.

Important: this does not contradict the classical diagonalization theorem — it says that it is impossible to have a single computable function that dominates all computable functions. The transrecursive class {T} is a family of functions, not a single fixed function.

#### A note about Busy Beaver and Rayo

Functions like Busy Beaver(BB (n)) and other specially constructed non-computable functions cannot be asymptotically defeated by any single computable function (including any specific TRANSCEND implementation). Therefore, the statement "TRANSCEND surpasses all" strictly needs to be clarified: the TRANSCEND class covers all computable functions; however, non-computable functions (BB, Rayo, etc.) lie outside this comparability in the sense of general asymptotics — their superiority or non-superiority is determined by other criteria (not computability).

#### 8. Approximate calculations and illustrations (canonical choice y0=exp (-1))

For a reference point, here are the exact values that are easy to calculate:

T0(e−1)=1/1−(e−1) =1/(1-1/e) ~ 1.58197670686933). Next: G := T0(e-1) ~ 1.5819. Where k=⌊eG⌋=⌊e1.5819⌋≈⌊4.86⌋=4.

Therefore, META\_ITER will perform 4 iterations; even with k=4, we get a very powerful increase (approximately several nested exponentials), and the final formula T11(e−1) will give a number that is infinitely large for all practical purposes and exceeds most well-known "very large" numbers used in googology (including many specific ones instances), although not formally comparable to non-computable constants. (Note: this numerical example is illustrative; the exact valueof T1 quickly goes beyond the human-readable representation.)

#### 9. Functional and philosophical implications

The HCCSF ⇝ TRANSCEND transition gives a fundamental gain: HCCSF gives the complexity coordinate of the number; TRANSCEND uses this coordinate as a resource to strengthen the growth production mechanism itself. The HCCSF-1 meta-inversion is a key "lift" between a number and its complexity ordinal: it turns the numerical result into fuel for the next growth phase. In this way, TRANSCEND implements "self-sustaining" and "self-reinforcing" growth dynamics: the result serves as a source of increased resources for the next step, and not just as an argument to a fixed operator.

#### 10. Limitations, comments and further directions

1. Practical computability. Formallyn, Tn(y) is computable for a fixed n, but practically calculations become impossible for small n due to the astronomical magnitude of the intermediate numbers. This does not make the definition meaningless: the value remains well-defined and algorithmically approximable.
2. Sensitivity to parameters. The TRANSCEND class is very flexible — changing the rule for counting k, smoothing S, or the initial y0 can radically change the numerical scales. For scientific work, it is important to fix the "canonical" parameters in order to have comparative points.
3. A formal place in logic. TRANSCEND fits into the formal TRT theory (see axioms T₁–t₅): it is formalizable in ZFC as a definition scheme; however, valuesn of tn for large n may require strong theory-oriented reasoning for their formal characterization (similar to how values in higher ordinal notations require strengthened axioms).
4. Further research. It is recommended to make an accurate estimate of the growthof tn through lower/upper bounds in terms of known FGH functions; to investigate the stability of the topology and smoothness of HCCSF under various normalizations of S.

#### 11. Conclusion of the section

The transition from HCCSF to TRANSCEND is a transition from measurement (the complexity coordinate of a number) to self-increment arithmetic (using this coordinate as a resource to generate even higher levels). Formally, TRANSCEND is constructed using the recursive META\_ITER operator, which uses HCCSF-1 as a means of moving to the ordinal space, and then returns to the numeric domain with a reinforced exponent. The resulting class of functions has recursive self-similarity, incredibly fast growth, and thus serves as a constructive limit for all computable growth rates. And now - a strict formalization of transrecursive theory.

## TRT (Trans-Recursive Theory)

### HCCSF and TRANSCEND as the ultimate design system

#### Annotation

We introduce a formal TRT (Trans-Recursive Theory) systembased on the HCCSF (Hierarchical Computable Complexity Scale Function) and TRANSCEND (Trans-Recursive Arithmetic Notation for Scaling Complexity and Exponential Number Dynamics)functions. TRT axiomatizes the principles of computable growth and shows the existence  
of a limit class of functionsthat completely covers all computable processes.  
TRANSCEND implements a universal scheme in which arithmetic, iterative, and ordinal growth are combined into a single transrecursive dynamic.

#### Introduction

Mathematics traditionally describes growth in terms of exponentials, towers, and hyperoperators, but they all represent fixed forms of iteration. TRANSCEND and HCCSF introduce a meta-iteration, where the growth structure itself evolves depending on the current level of complexity. The purpose of TRT is to formalize and axiomatize this principle.

#### TRT Axiom System

Denote by {F} the set of all computable functions ( f: N→N). HCCSF and TRANSCEND belong to {F}, but they set special growth principles.

#### Axiom T₁ (Arithmetic constructivity)

T0(y)=1/(1−y), y∈[0,1) is a basic computable function that guarantees continuity, monotonicity, and an infinite limit for y→1-.

Corollary: T0 defines the first level of the complexity hierarchy L0 = ω.

#### Axiom T₂ (Exponential iteration)

Tn+1(y) = eTn(y)

Each level generates the next one by means of exponential gain.

Corollary: Levels (L1,L2,L3... ) correspond to ordinals ( ω^ω, ε\_0, Γ\_0, … )

#### Axiom T₃ (Continuity interpolation)

For all y ∈ [n, n+1): HCCSF (y)=(1-α)⋅Tn(y)+α⋅Tn+1(y), α=y−n.

Corollary: HCCSF (y) is continuous, strictly increasing, and defined everywhere. This is a continuum of computable complexity.

#### Axiom T₄ (Meta-iterative self-reinforcement)

Tn(y)=exp([META\_ITER(Tn-1(y),eTn−1(y))]Tn−1(y))

Interpretation: The number of iterations eTn-1 (y) itself depends on the result of the previous step, and each iteration raises the complexity level through the inverse function HCCSF-1.

Consequence: TRANSCEND generates a self-reinforcing process — a self-reflective growth function that dynamically increases its computing power.

#### Axiom T₅ (Limit of constructive growth)

For any computable f(n), there exists N such that: TRANSCEND (n) > f(n) for all n > N. And there is no computable g satisfying g(n) > TRANSCEND (n) for all n.

Corollary: TRANSCEND is the upper envelope of all computable functions. It implements the limit constructive growth class.

### Theorems and proofs

#### Theorem 1 (On continuity)

HCCSF and TRANSCEND are continuous on their domains of definition.

Proof:  
 Each component is an exponent or a linear combination of continuous functions.  
For transitions between levels n and n+1, the limit equality is preserved.

#### Theorem 2 (On monotonicity)

If (1 y1 < y22), then (1TRANSCEND(y1) < TRANSCEND(y2)).

Proof:  
 The functionnTn (y) strictly increases, and the exponent preserves order.

#### Theorem 3 (On computability of TRANSCEND)

For any finite n, TRANSCEND(n) is computable.

Proof:  
 A recursive scheme consists of computable operations, and all intermediate iterations are finite.

#### Theorem 4 (On the limit class of growth)

Let {T} = TRANSCENDΦ) be the set of all TRANSCEND functions with different interpretations of the complexity levels Φ. Then: ∀f ∈ Comp ∃ TRANSCENDΦ∈T:∃N: ∀n>N, TRANSCENDΦ(n)>f(n)

Proof:  
 TRANSCEND dynamically generates arbitrarily deep levels of complexity through the composition of HCCSF-1 and exponentials. Each modification of Φ shifts the ordinal limit up, which guarantees dominance over any computable f.

#### Hierarchical structure

|  |  |  |
| --- | --- | --- |
| Level | Ordinal | TRANSCEND interpretation |
| 0 | ω | Primitive Recursion |
| 1 | ω^ω | Grzegorczyk hierarchy |
| 2 | ε₀ | Ackermann limit |
| 3 | γ₀ | Fefermann-Schutte |
| 4 | ε\_{γ₀+1} | Meta-recursion |
| 5+ | e\_{α+1} | Infinite meta-hierarchy TRANSCEND |

#### Consequences

* TRANSCEND implements a transrecursive computability limit.
* HCCSF provides a continuum of computable complexitysimilar to the real line for growth.
* Any computable TRANSCEND transformation remains in the same ordinal class — arithmetic modifications do not change the fundamental level of complexity.

#### Philosophical interpretation

TRANSCEND is not just a function, but a universal law of growth, in which each step sets itself a new system for measuring growth. This is the mathematical equivalent of the self-evolving universe of computing: growth creates a space in which it grows.

TRANSCEND is the computability constant of the speed of light. It defines the boundary where the constructive ends and the trans-constructive only begins.

#### Comparison with other hierarchies

|  |  |  |  |
| --- | --- | --- | --- |
| System | Growth mechanism | Computability | Comparison with TRANSCEND |
| Ackermann | exponent iteration | computable | ≪ TRANSCEND(1) |
| Grzegorczyk (FGH) | ordinal recursion | is computable | ≪ TRANSCEND(1) |
| TREE(3) | Combinatorics of trees | is computable | ≈ TRANSCEND(2) |
| Loader | recursive schemes | are computable | < TRANSCEND(2) |
| Rayo(n) | formal self | -definability is non-computable | incomparably, but TRANSCEND is constructive |
| Busy Beaver | undecidability of halting | is non-computable | BB > TRANSCEND is asymptotic, but TRANSCEND is computable |
| TRANSCEND | dynamic meta-exponential recursion | is computable | computability limit |

#### Philosophical interpretation

The TRANSCEND function is a mathematical analog of the speed of light for computable processes: it defines the absolute limitbeyond which any further acceleration requires a change in the very foundations of the concept of "computation". Each of its levels is a new stage of meta-mathematical awareness, and its formula combines arithmetic, algorithmic, and ontological growth in a single structure.

#### Conclusion

1. TRANSCEND is a constructive, smooth, continuous, monotonic function that defines the limit growth of computable processes.
2. The HCCSF scale forms a continuous map of ordinal complexity, combining ordinal and computable structures for the first time.
3. The dynamic level system provides infinite continuation without loss of computability.
4. TRANSCEND is the last computable function: any acceleration beyond it requires going beyond Turing computability.

#### Conclusion

TRT theory shows that constructive growth has a natural limit. This limit is achievable and computable — in the TRANSCEND form. Ordinals, computability, and continuity combine to form a single axiomatic structure. TRANSCEND is not an upper function, but an upper class, not a single number, but the law of self-expansion of the computable world.

## Comparison of TRANSCEND and other fast-growing Google hierarchies and functions.

#### BEAF (Bowers Exploding Array Function)

Nature: syntactic and recursive superstructure over hyperoperations (arrays, trees, indexes).  
 How it grows: through the syntactic depth of recursions and indexes {a, b, c, d}.  
 What it does: sets a very efficient counter structure, where each index denotes a hyperoperation type or a massive level.

But: BEAF remains inside the numeric semantics. There is no display of "number - > difficulty level", only "number - > syntax depth". There is no meta-inversion (HCCSF⁻1): the value is not converted to its own order.

TRANSCEND goes further: it does not just create an array, but dynamically changes the measurement system of a quantity-moving to the space of ordinal levels.

#### TREE(n)

Nature: purely combinatorial.  
 How it grows: through the maximum length of a sequence of trees without isomorphism according to certain rules.  
 What it does: sets huge values, but based on object structures, not computational complexity levels.

But: TREE grows due to combinatorial constraint, not due to meta-iterations. It doesn't know how complex its own generation is. No relation "number ↔ computable complexity": trees just count, but don't encode computability levels.

TRANSCEND translates each result into a "difficulty level" and uses this as fuel for the next step.  
TREE does not do this in principle — it is not a self-reinforcing system, but simply an "extreme combinatorial score".

#### SCG(n) (Busy Beaver-type function, "Super Collatz Growth")

Nature: machine-generated (at the Turing level).  
 How it grows: as a maximum of the number of steps of a machine of a certain size.  
 What it does: generates uncomputable growth (BB, SCG).

But: SCG/BB work on the boundary of the computable, but still in terms of the number of steps or the length of the computation. There is no "number - > ordinal" mapping; there is no hierarchical reflection of complexity. These functions do not fundamentally convert a "value “to a”structure".

TRANSCEND is exactly what makes it different: it takes the result (value), looks at "what level of complexity it lives at", and jumps up-turning the growth dimension into a new dimension of computability.

#### Ackermann / Fast-Growing Hierarchy / Grzegorczyk

Nature: strictly computable, recursive structure.  
 How it grows: each level specifies a new hypergrowth operator indexed by the ordinal.  
 What it does: it climbs the ordinal ladder (ε₀, γ₀, etc.), but the ordinal is set externally.

But: The ordinal is a function parameter, not the result of a calculation. No reverse operation of the HCCSF⁻1 type is applied. That is, the function does not move between the numeric and ordinal spaces — ordinals are simply indexed, but do not participate as values.

TRANSCEND makes ordinals internal growth variables for the firsttime.  
Here, the complexity level is measured, calculated, and used as an argument - in the dynamics of the function itself.

#### Hierarchy comparison table

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Type of growth | What is the nature | Of Using the "number - > difficulty level" mapping? |
| Ackermann / Grzegorczyk | Ordinal-indexed recursion | Static | ✗ |
| BEAF | BEA BEAF Syntactic explosion | Combinatorial | ✗ |
| TREE / SCG | Combinatorics / Turing Machines | Structural | ✗ |
| TRANSCEND (HCCSF) | Meta-ordinal recursion | is dynamic and self-reflexive | ✅ |

As a result, we have that TRANSCEND is the first system where the value of a number is used to calculate its computable complexity, and the computable complexity is returned back to recursion as an argument for growth. This creates a closed loop between arithmetic and hierarchy, creating exponential gain of a fundamentally new type. As a nice analogy, you can give an example: TREE and BEAF are mountains of numbers. TRANSCEND is a mechanism that moves the mountains themselves, because it changes the structure of the space where these mountains grow.

## The essence of the breakthrough: Moving from quantitative growth to ordinal growth

Prior to TRANSCEND, no hierarchy used hierarchical order of complexity as a significant factor in computing. All development of " fast-growing functions "(Ackermann, Grzegorczyk, Fast-growing hierarchy, TREE, BB) was based on a numerical growth metric — exponentials, tetrations, hyperoperators, iterations, etc. качество скачкаgrowth in numbers and begins to measure it in levels of complexity, that is, in ordinal types of computing processes.

This is the transition from" how big is the number " - > to "how complex is the way to get big".

#### The main mechanism is mapping numbers to an ordinal hierarchy of complexity

In classical mathematics, large numbers are simply large values on the оси or ℕ axis. TRANSCEND introduces the HCCSF function, which maps each number to the level of complexityat which it “lives” as a computational object. And vice versa: HCCSF-1(x) = {number difficulty level } x. That is, the number ceases to be just a "value" — it becomes a marker of the depth of the computing architecture.

#### TRANSCEND Function Life Hack — " energy closure”

When a function gets the valuetn-1(y)), it does something revolutionary: It does not use the number itself, but uses it as a key to its ordinal level, and then exponentiates this level again and recursively inserts it into itself. This is figuratively speaking a kind of analog of the" nuclear reactor " growth. Instead of just burning arithmetic, TRANSCEND turns the very idea of “numbers " into fuel for the meta-process. Thus, each iteration translates the numeric space into an ordinal one, strengthens it, and returns it back-creating a closed chain of growth enhancement.

#### Why is it stronger than all known forms of growth

|  |  |  |
| --- | --- | --- |
| Level | Type of growth | Analogy |
| Arithmetic | addition, multiplication, exponentiation | bonfire |
| Iterations | recurrent application of operations | blast furnace |
| Hyperoperators | tetration, pentation, etc | . plasma chamber |
| TRANSCEND | moving numbers to the complexity level and back | nuclear reactor calculations |

TRANSCEND doesn't just take more steps — it changes the very structure of the growth space. That is, its growth is not only greater, but also occurs in a dimension that is faster than any previous dimension.

#### The principle of"everything that can burn burns"

Formally — it is a superposition of three independent growth enhancers:

1. Arithmetic growth — exponents, logarithms, hyperoperators. → responsible for the quantitative side.
2. Algorithmic growth — an increase in the depth of iterations, which is responsible for the structural side.
3. Hierarchical (ordinal) growth — the transition to a new level of complexity, it is responsible for the ontological side.

And all this is combined into a self-reflective loop, in which each element of growth reinforces the other two. TRANSCEND is a function where growth accelerates itself along three mutually independent axes. That is why it is not just “big”, but the limit on the principle. Feel the difference! In normal functions, numbers grow. In TRANSCEND, the ability to grow itself is growing. In other words, the function becomes not an object, but a meta-process — it generates its own space of growth opportunities. This is the main philosophical and mathematical leap: TRANSCEND is the first formal object that implements a self-generated growth meta-structure within computability.

To summarize: The main breakthrough is the replacement of numbers with difficulty levels. The main mechanism is the reverse HCCSF⁻1 mapping, which turns the value into an order. The main effect is superinductive growth enhancement in a closed loop. The main philosophy is that growth becomes its own cause.

## Final reflection, scientific novelty and significance

Coming to the finish line of the article, it is worth noting the most important consequence of TRT that TRANSCEND sets an upper bound on the constructive growth of computable functions — and this is not a particular result, but a new fundamental level in function theory and complexity theory.

#### Important implications in the context of existing science

|  |  |  |
| --- | --- | --- |
| Area | What has already happened | What TRANSCEND |
| Computability Theory adds | Defined bounds on computability (Turing, Church, Kleene). | Gives a constructive limit to the rate of computable growth. |
| Googology and FGH | Describe fast-growing hierarchies (Ackermann, Grzegorczyk, Feferman-Schütte, Buchholz). | TRANSCEND combines them into a continuous hierarchy with a self-reinforcing meta-iterative principle. |
| Analysis of ordinals | The limit ordinals ε₀, γ₀, ψ(Ω\_ω) are known. | TRANSCEND introduces a dynamic scale αₙ = ε\_{αₙ₋₁+1}, creating an infinite chain of computable orders. |
| Mathematical logic | Goedel's and Tarski's theorems limit provability and definability. | TRANSCEND does not violate these limits, but rather describes exactly where they become limiting constructively. |
| Information Theory | Limits of growth and computation-through physical or entropic constraints. | TRANSCEND sets a formal mathematical analogy for the limiting speed of information growth (analogous to the speed of light in physics). |

Final significance

TRANSCEND for the first time formalizes the concept of maximum computable growth in a constructive form-it is an analog of the" second constant of nature " in theoretical computer science, only expressed mathematically.

#### What is the conceptual novelty?

* A new type of function: TRANSCEND is not just a function f: N - > N, but a class of functionsunited by a single principle of self-reinforcement through levels of ordinal complexity. In other words, the concept of self-reflective constructive growth is formalized for the firsttime.
* Combining discrete and continuous: HCCSF (hierarchical scale) makes continuous modeling of computable hierarchies possible, which was not the case in FGH, Ackermann's theory, or googology — they are all discrete.
* Dynamic ordinal system: Instead of fixed ordinals ε₀, γ₀, etc., a relative system ( alphan+1 = ε{alphan+1}) is introduced, which can generate an infinite computable sequence of ordinals — providing algorithmic infinity in the analysis of ordinals for the first time.
* Meta-recursive architecture: TRANSCEND for the first time uses the inverse complexity function HCCSF-1 as a calculation tool, rather than as an abstract index. This creates a new type of self-referential growth, where the function evaluates its own level of complexity and enhances itself through it.

#### Technical value and contribution

* Continuous Monotone form of FGH:  
  HCCSF is the first ever continuous and monotone model for all discrete fast growth hierarchies.
* Formally computable limit:  
  TRANSCEND remains computable for any finite n, but asymptotically exceeds all other computable functions. This is a strict formal definition of the upper limit of constructive growth.
* Mathematical completeness:  
  The theory contains a closed system of growth axioms (t₁–t₅), lemmas on monotonicity, continuity, computability, and the growth limit theorem.
* New difficulty measurement unit:  
  We introduce a continuous function of the complexity level Φ(x) = max{ n | HCCSF(n) ≤ x },  
  which gives the algorithmic "height" of any number-an analog of the logarithm, but for ordinal complexity.

#### The place of theory in modern mathematics

|  |  |
| --- | --- |
| Direction | Contribution level |
| Computability theory | The constructive growth limit is described for the first time. |
| Mathematical logic | TRANSCEND formalizes the idea of a self-reflexive function within computability. |
| Googology | Transforms googology from a descriptive discipline to a formal one. |
| Theory of ordinals | Introduces an infinite algorithmic ladder of ordinals. |
| Philosophy of Mathematics | Shows that the "boundary of the computable" can itself be described constructively. |

#### Practical and methodological value

* In theoretical computer science — it is a universal model of marginal computable complexity.
* In metamathematics — it is a way to classify functions by their computable cardinality.
* In artificial intelligence, the idea of TRANSCEND can describe self-reinforcing computing systems.
* In the philosophy of mathematics — he introduces for the first time a formal “growth ontology” as a mathematical object.
* In education, this can be the logical conclusion of the section on computable functions in mathematical logic courses.

#### Conclusion

The TRANSCEND theory is a new fundamental construction in mathematics,  
comparable in scale to the introduction of Ackermann functions (at one time), the Grzegorczyk hierarchy, or the concept of ordinals ε₀, γ₀. But TRANSCEND goes further, it formalizes the whole set of constructive growth, creating an algorithmically generated scale of computable complexity, in which the limit is not conditional, but mathematically defined. This isn't just a new feature. This is the end of the whole line of development of the theory of computable hierarchies.