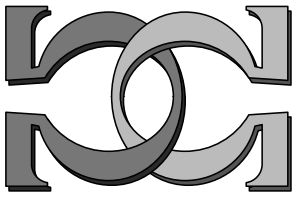

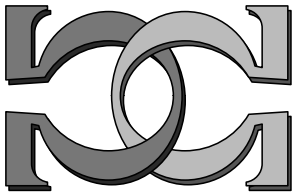

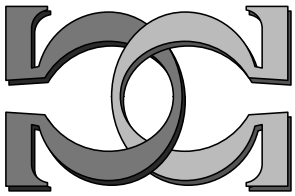

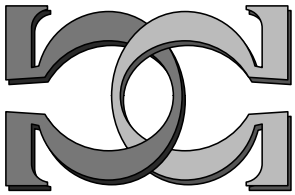

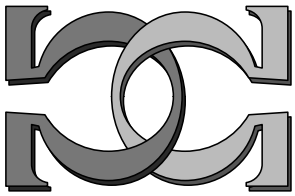

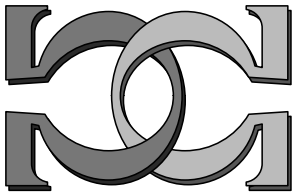

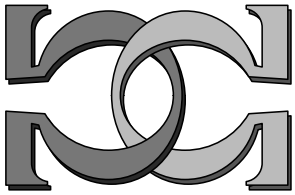

CDMTCS Research Report Series

Inductive Complexity of Goodstein's Theorem

Joachim Hertel
H-Star, Inc., USA

CDMTCS-418
7 April 2012

Centre for Discrete Mathematics and
Theoretical Computer Science

Inductive Complexity of Goodstein's Theorem

Joachim Hertel[i]

Abstract. We use the recently introduced [1,2] inductive complexity measure to evaluate the inductive complexity of Goodstein's Theorem, a statement that is independent from Peano Arithmetic.

Introduction

In mathematical logic, Goodstein's Theorem [3] is a statement about the natural numbers which states that every *Goodstein sequence* eventually terminates at 0. In 1982, Kirby & Paris showed [4] that Goodstein's Theorem is unprovable in Peano arithmetic (PA) but can be proven in stronger systems, such as second order arithmetic.

A Turing Machine TM that enumerates all integers and for each integer n then checks if the associated Goodstein sequence $(n)_k$ terminates at zero is considered a brute force tester for the Goodstein Theorem. However, in PA , this Turing Machine might never stop for two reasons:

Reason 1: Goodstein's Theorem is TRUE

Reason 2: Goodstein's Theorem is FALSE, i.e. there exists an integer n_0 such that the associated Goodstein sequence $(n_0)_k$ does not terminate at zero.

Only when we refer to stronger systems than PA can we rule out Reason 2 and can come up with a brute force testing Program P which semi-decides Goodstein's Theorem and the complexity of the semi-decidability of Goodstein's Theorem can be measured by applying the complexity measure C_U for Π_1 statements as introduced by Calude et al in [5,6,7]. The algorithm P semi-decides Goodstein's Theorem because the algorithm stops iff the theorem is FALSE.

However, if we restrict ourselves to the framework of PA, we cannot rule out Reason 2, because the Goodstein Theorem is independent from PA. That raises the question if we can nevertheless apply a meaningful complexity measure to evaluate the Goodstein Theorem without reasoning outside of PA. The method of super recursive algorithms [8] and the closely related concept of inductive Turing Machines [9,10] can be used to define a generalized complexity measure that can be applied to cover sentences of the kind $\forall n \exists k\ R(n,k)$ where R is a computable predicate. To be more specific, following [1,2], we apply this method to the Goodstein Theorem as follows:

Let n be any non-negative integer and let $(n)_k$, k ≥1, denote the k[th] term of the Goodstein Sequence [3,11] of n defined as:

$$(n)_k = \begin{cases} n & k = 1 \\ R_k((n)_{k-1}) & if\ k > 1\ and\ (n)_{k-1} > 0 \\ 0 & if\ k > 1\ and\ (n)_{k-1} = 0 \end{cases}$$

The change of base function $R_b(n)$ [see e.g. 11] takes a natural number n, and then syntactically replaces every b with b+1 in the complete base b representation of n.
Next, define the computable binary predicate GS(n, k) by setting

$$GS(n,k) := \begin{cases} 1, & (n)_k = 0 \\ 0, & otherwise \end{cases}$$

and the inductive Turing machine of first order

$$T_{GS}^{ind,1}(n) := \begin{cases} 1, & \exists k, s.t.\, GS(n,k) = 1 \\ 0, & otherwise \end{cases}$$

Note, that the inductive Turing machine $T_{GS}^{ind,1}$ is implemented by the register machine program GS, see chapter Application Library for details.
We proceed and define the inductive second order Turing Machine

$$M_{GS}^{ind,2}(n) := \begin{cases} 1, & \forall n\,(T_{GS}^{ind,1}(n) = 1) \\ 0, & otherwise \end{cases}$$

The following inductive program MAIN checks the Goodstein Theorem based on $M_{GS}^{ind,2}$.

Label	Instruction	Comments
MAIN	&OM,1	// set output register to 1
	&a,1	// init seed for Goodstein sequence
	&c,LM02	
LM01	=a,a,GS	// Check if GS sequence for a goes to 0
LM02	=OG,1,LM03	// branch if GS(a) has reached 0
	&OM,0	// indicate error
	=a,a,LM04	// branch to HALT
LM03	+a,1	// next integer a
	=a,a,LM01	// proceed checking Goodstein Theorem
LM04	%	// HALT

Applying the broader concept of super-recursive algorithms [8] and more specifically inductive Turing machines of second order, we can now measure the complexity of the decidability of the Goodstein Theorem inside PA without referring to higher order arithmetical systems. Following [1,2] we define

$$\boldsymbol{C}_{U,n}^{ind,2} := \{\rho: \rho = \forall n \exists k\, R(n,k), C_U^{ind,2}(\rho) \leq n\ kbit\}$$

with

$$C_U^{ind,2} := min\{|M_{GS}^{ind,2}|: \rho = \forall n \exists k\, R(n,k)\}$$

and R(n, k) a general binary computable predicate (in this work: R(n, k) = GS(n, k)).

Using MAIN as our register machine implementation of $M_{GS}^{ind,2}$ and the computable binary predicate R(n, k) := GS(n, k) we come up with a bit count of 7909 for the total application and thus conclude that the inductive complexity of Goodstein's Theorem is in the inductive complexity class $\mathbf{C}_{U,7}^{ind,2}$.
The Goodstein Theorem is thus far more complex than e.g. the Collatz Conjecture or the Twin Prime Conjecture which are both in $\mathbf{C}_{U,1}^{ind,2}$ as recently has been shown in [2].
Using inductive computing enables us to evaluate the complexity of the decidability of this statement without referring to arithmetic systems stronger than PA.

In chapter 2 we briefly explain how to encode hereditary representations using Dinneen's [12] Array Library for register machines. In chapter 3 we recall the underlying concept of a prefix free universal Turing machine as specified in [1,2,5,6,7], followed by a description of all arithmetic, array and application specific routines that we need to evaluate the complexity of Goodstein's Theorem. The last chapter summarizes our findings and points to related research of phase transitions from provability to unprovability.

Using Arrays to Handle Hereditary Representations

Checking the Goodstein Theorem involves the computation of complete hereditary representations for increasing base values for non negative integers [3,11].
In order to handle and store hereditary representations of integers and compute new integers by bumping up the base value we use arrays and their encoding as developed by Dinneen [12].
The Dinneen encoding represents an array as a single register variable. An integer element a_i within an array A is represented as a sequence of a_i bits 0; the bit 1 is used as a (leading) separator of the array elements. If there are no 1's (e.g. the register has value 0) then we have an array of size 0. That is integers are encoded in a unary way.
For this work it is important to APPEND elements at the end of an array and we decided to to use a left-to-right interpretation. For example the array A = [a1, a2, a3, a4] = [6, 1, 0, 4] is represented in Dinneen's encoding as 100000010110000 which is decimal 16560.
We proceed by explaining how to use arrays to represent complete hereditary representations of base b for any integer n.
Using the 3 operators +, x, ^, the so-called complete hereditary representation of an integer n, at level 3 and base b, can be expressed as follows using only the 3 operators +,x,^ and using as digits only 0, 1, ..., b-1:

For $0 \leq n \leq b-1$, n is represented simply by the corresponding digit. (1)

For $n > b-1$, the representation of n is found recursively, first representing n in the form (2)
b^X x Y + Z = n, where X,Y,Z are the largest integers satisfying (in turn)

1. b^X <= n
2. b^X * Y <= n
3. b^X * Y + Z =n

Any X, Y, Z value exceeding b-1 is then re-expressed in the same manner, repeating this procedure until the resulting expression contains only the digits 0, 1, ..., b-1.

We use the following numeric encoding scheme to represent the basic entities involved:

- base symbol b 1
- exponentiation symbol ^ 2
- multiplication symbol * 3 (3)
- addition Symbol + 4
- integer $k \geq 0$ 5+k

To avoid bracketing we use reverse Polish notation (RPN) to represent b^X * Y + Z as an 7- element array [b,X,^,Y,*,Z,+], where each symbol is represented by its encoded value.

A call to subroutine PHR (a,b) from the Application Library produces the depth-1 7-element array [b,X,^,Y,*,Z,+], where X, Y, Z are chosen according to (2) and fulfill b^X * Y + Z = a.
We then scan such an array for any occurrence of values larger or equal to encoded base value ~B=5+B, where global register B tracks the current value of the base symbol.
Each time we detect such an integer, we call PHR again, replacing that integer by a depth-1 7-element

array , which we then merge with the existing array to create a new array of length N+6.
We recursively repeat this procedure to create an 7+ j*6 -Element array, in which all integers are smaller than the encoded base value ~B=5+B. This array then represents the complete hereditary representation at level 3 and base b and has length 7+j*6 and all integers are encoded according to (3). As the array is organized in RPN it can easily be executed in further steps necessary to advance the calculation to the next element in the Goodstein sequence of any given seed integer n.

We illustrate this with the integer 266 and base B=2. Assume global registers A=0,N=0,B=2.
The first call PHR(266,2) generates depth-1 representation 266= 2^8 *1 +10, which is represented in RPN as an 7-element array A=[b,X,^,Y,*,Z,+]. Using the encoding schema (3) and X=8,Y=1,Z=10 and their encoded values 13,6,15 respectively, we get the 7-element array

A=[1, **13**, 2, **6**,3,**15**,4].

We have marked in bold the integers k in their encoded form 5+k. The non-bold integers mark the encoded values of the base symbol and the three operators ^,*,+.
We then invoke subroutine ELM to scan this array. We skip every element that is smaller than 5+B, otherwise we compute a new 7 element array by calling PHR for that integer.
We illustrate these steps here, beginning with our depth-1 array for integer 266 in base 2:
A=[1, **13**, 2, **6**,3,**15**,4]
replace encoded **15** by PHR(10,2)which gives [1, **8**, 2,**6**,3,**7**,4]
Using MERGE we build the new 7+6-element array A=[1, **13**, 2,**6**,3, 1, **8**, 2,**6**,3,**7**,4,4].
Proceeding this way until all bold integers are smaller that 5+B =7 we end up with

A=[1, 1, 1, **6**, 2,**6**,3,**6**,4,2,**6**,3,**5**,4, 2,**6**,3,1,1,**6**,2,**6**,3,**6**,4,2,**6**,3,1,**6**,2,**6**,3,**5**,4,4,4]

this N = 37 = 7 + 6*j, with j = 5, element array represents in RPN the complete hereditary representation of level 3 in base 2 of the integer 266.

Using Dinneen's encoding [12] this array is then represented as

$$10_1 10_1 10_1 10_6 10_2 10_6\ 10_3 10_6 10_4\ 10_2 10_6 10_3\ 10_5 10_4 10_2 10_6 10_3 10_1 10_1 10_6 10_2 10_6 10_3 10_6 10_4 10_2 10_6 10_3 10_1 10_6 10_2 10_6 10_5 10_4 10_4 10_4$$

where we use the shorthand 0_n to represent n 0s.

Hence the full level-3 hereditary representation of 266 in base 2 is stored as this decimal integer

795139176033715787608490082818214012671211515623886 4

in a register. Note that we are particularly using the feature of register machine models that any register can hold arbitrarily large integer. So no matter how gigantic the numeric values of any element in a Goodstein sequence might be and how gigantic the arrays are to hold the complete hereditary representation, we can always store the value in one register.

We then invoke ELM and EXO (EXecute Operator) to scan any given array and compute the next element of the Goodstein sequence by "bumping up" the base value from B to B+1. That is, whenever we scan the array A and read integer 1 in array A we put the value B+1 on the execution stack. That gives a new integer, from which we subtract 1, check if the result is zero and if not we continue the process for this new integer with base value B+1.

That outlines the general way we use arrays to handle the complete hereditary representation and then implementing the change of base function to advance to the next element in a Goodstein sequence.

A Universal Prefix-Free Binary Turing Machine

Following [1,2,5,6,7] we briefly describe the syntax and the semantics of a register machine language which implements a (natural) minimal universal prefix-free binary Turing machine U.

Any register program (machine) uses a finite number of registers, each of which may contain an arbitrarily large non-negative integer. By default, all registers, named with a string of lower or upper case letters, are initialized to 0. Instructions are labeled by default with 0,1,2,. . .

The register machine instructions are listed below. Note that in all cases R2 and R3 denote either a register or a non-negative integer, while R1 must be a register. When referring to R we use, depending upon the context, either the name of register R or the non-negative integer stored in R.

=R1,R2,R3

If the contents of R1 and R2 are equal, then the execution continues at the R3-th instruction of the program. If the contents of R1 and R2 are not equal, then execution continues with the next instruction in sequence. If the content of R3 is outside the scope of the program, then we have an illegal branch error.

&R1,R2

The contents of register R1 is replaced by R2.

+R1,R2

The contents of register R1 is replaced by the sum of the contents of R1 and R2.

!R1

One bit is read into the register R1, so the contents of R1 becomes either 0 or 1. Any attempt to read past the last data-bit results in a run-time error.

%

This is the last instruction for each register machine program before the input data. It halts the execution in two possible states: either successfully halts or it halts with an under-read error.

A register machine program consists of a finite list of labeled instructions from the above list, with the restriction that the halt instruction appears only once, as the last instruction of the list. The input data (a binary string) follows immediately after the halt instruction. A program not reading the whole data or attempting to read past the last data-bit results in a run-time error. Some programs (as the ones presented in this paper) have no input data; these programs cannot halt with an under-read error.

The instruction =R,R,n is used for the unconditional jump to the n-th instruction of the program. For Boolean data types, we use integers 0 = false and 1 = true.

For longer programs, it is convenient to distinguish between the main program and some sets of instructions called “routines” which perform specific tasks for another routine or the main program. The call and call-back of a routine are executed with unconditional jumps.

Selecting one or several registers as output registers, register machine programs can compute not only in the classical (recursive) mode, when the program halts to get a result, but also in the inductive mode described above. In the inductive mode, register machine programs can simulate inductive Turing machines of the first order in the same way they can simulate Turing machines when working in classical (recursive) mode.

The binary encoding rules for register machine programs can be found in [5,6,7].

Conventions and Libraries

Routines and their Register Names

There are two types of routines: 1-routines, that is routines that do not use any other routines and 2-routines, that is routines that call other routines. In general, unary 1- or 2- routines use the register a for input, b for keeping track of returning to the calling environment and c for storing the result. Binary 1- or 2-routines use a and b for input, c for the return address and d for the result.
As the registers are shared between the main program and routines, we need a convention to avoid any conflict in using register. We largely follow the convention as put forward in [13].
The main program uses capital letters for registers. 1-routines use single letter names, 2-routines use double letter names, where we use **h**[1] as a generic second letter and replace it by the routine's unique handle. This way we are no longer dependent on unique first letters of routines, where we anticipate sooner or later some conflicts (e.g. with names like **D**IVIDE and **D**ELETE).
We can reuse the following existing unary and binary 1- and 2-routines from various libraries [1,12,13].

Arithmetic Library

CMP

The binary 1-routine CMP [13, Table 3] takes as input two non-negative integers, stored in registers a and b, and produces its result in register d according to the formula:
d = 1 if a < b, d = 0 if a = b, and d = 2 if a > b. It then return to c.

SUBT

The binary 1-routine SUBT [12] takes two non-negative integers, stored in registers a and b and produces the result d = a-b. It then returns to c.

SUBT1

The binary 1-routine SUBT1 [13, Table 4] takes a non-negative integer ≥ 1 stored in registers a and produces the result d = a-1. It then returns to c.

MUL

The binary 1-routine MUL [12] takes as input two non-negative integers , stored in registers a and b, computes d = a * b and returns to the line stored in c.

DIV

The binary 2-routine DIV [12] takes as input two non-negative integers, stored in registers a and b. It produces the result d = FLOOR[a/b]. It then returns to c.

DIV2

The unary 2-routine DIV2 [13, Table 5] takes as input a single non-negative integers, stored in register a .It updates a with FLOOR[a/2] and sets register c = Mod(a,2). It returns to b.

---

[1] **h** stands for handle; we envision a public software repository for approved register machine routines, each routine having a unique handle h, which is then used to generate unique register names in the complete source file.

POW

The binary 2-routine POW ,takes as input two non-negative integers, stored in registers a and b. It produces the result $d = a^b$. It then returns to c. See Appendix for a reference implementation.

Array Library

AI

The binary 1-routine AI[2] [13] takes as input register a and a non-negative integer in register b and initializes register a to an array of size b. It returns to the line stored in c.

ELM

The binary 2-routine ELM [13, Table 7] takes as input a number stored in register I and an array in register A. It extracts the I^{th} element from the array A and stores it in register d, i.e. d = A[I] and returns to the line stored in c.

REPL

The binary 2-routine REPL [13, Table 8] takes as input a number stored in register I and an array in register A. It replaces the I^{th} element from the array A with the value stored in register b , A[I]=b, and returns to the line stored in c. There is no formal output other than the updated array A.

APPEND

The binary 2-routine APPEND [13, Table 6] takes as input a non-negative integer in register b and appends it to the array stored in a. Then returns to c. Using APPEND to append 2 to array [6,1,0,4] produces [6,1,0,4,2].

We add these general purpose routines to the library of Array Services:

DELETE

The unary 2- routine DELETE takes as input an array in register a and deletes the last element of it, updates a and returns to the line stored in register b. In register c the value of the deleted element is returned.

Example: Let a = $[a_1,a_2, ..., a_N]$. Delete(a) updates a to $[a_1,a_2, ..., a_{N-1}]$ and returns c = a_N.

Label	Instruction	Comments
DELETE	&b**h,**b	// save input
	&d,0	
L0	&b,L1	
	= a,a,DIV2	// divide by 2
L1	= c,1,L2	// if 1, element is deleted, go to exit

[2] AI means ArrayInit and refers to ArrayInit(a,b) in Dinneen's Array Library [1]

	+d,1	// count the zeroes
	= a,a,L0	
L2	&c,d	// load result to c
	&b,b**h**	
	= a,a,b	// return to caller

MERGE

MERGE is a binary 2-routine that takes as input in register a an array of length stored in register b. This array is then merged into the array stored in global register A of length stored in global register N at position I where A is updated and N is increased by (b-1). MERGE then returns to the line number stored in register c.
By example, MERGE operates as follows:

Given :
$A = [A_1,A_2, ..., A_{I-1}, A_I, A_{I+1,...} A_N]$
N = N
I = I
$a = [a_1,a_2, ..., a_b]$
b = b

MERGE then updates N to N+b-1 and updates array A to $[A_1,A_2, ..., A_{I-1}, a_1,a_2, ..., a_b, A_{I+1,...} A_N]$.

This new array is constructed by using ELM to first we read elements $A_1,A_2, ..., A_{I-1}$ from A and APPEND them to an empty array. Then we append all elements $a_1,a_2, ..., a_b$ and continue to APPEND the elements $A_{I+1,...} A_N$.

Label	Instruction	Comments
MERGE	&a**h**,a	// save input
	&b**h**,b	
	&c**h**,c	
	&i**h**,I	
	&r**h**,I	
	&j**h**, L04	//prepare jump label for MERGE
	&a,0	
L00	&I,1	
L01	&c,L02	
	=a,a,ELM	// fetch d= A[I]
L02	&b,d	
	&c,L03	
	=a,a,APPEND	// Append A[I] to array a
L03	+I,1	// point to next element
	=I,r**h**,j**h**	// jump if we have fetched enough A[I]
	=a,a, L01	// continue to fetch from array A
L04	&e**h**,A	// appended $A_1,...A_{I-1}$, save A

	&A,a**h**	// load save array a to A
	&r**h**,b**h**	// load compare length register
	+r**h**,1	// adjust
	&j**h**, L05	// load new jump label
	=a,a,L00	// continue scanning and appending
L05	&A,e**h**	// reload A
	&I,i**h**	// reload original index
	+I,1	// adjust
	&r**h**,N	// load compare length register
	+r**h**,1	// adjust
	&j**h**, L06	// load new jump label
	=a,a, L01	// continue to scan and append from A
L06	&A,a	// A = [A_1,A_2, ..., $A_{I-1,}$ a_1,a_2, ..., $a_{b,}$ $A_{I+1,...}$ A_N].
	+N,b**h**	// prepare N to be updated to N+b-1
	&b,N	
	&c,L07	
	=a,a,SUBT1	// Update N to N+b-1
L07	&N,d	// load N to reflect new length of A
	&a,b**h**	// reload register
	&b,b**h**	
	&c,c**h**	
	=a,a,c	// return to caller

Application Library

Here we list all application specific routines needed to measure the complexity of the Goodstein Theorem.

DECODE

DECODE takes an integer ≥ 1 stored in register a as input and produces the result c = B+1, if a=1 or c = k if a = 5 + k, for k ≥ 0. It then returns to b. Global register B stores the current value of the base symbol b.

Label	Instruction	Comments	Binary Representation[3]
DECODE	&c,B	// load current base value	01 00110 001100
	+c,1	// bump up base	111 00110 101
	=a,1,b	// return if a is a base symbol	00 010 101 011
	&c,0		01 00110 100
L0	&d,c		01 00111 00110
	+d,5	// load decoding constant	111 00111 1110101
	=d,a,b	// check if a = c+5, if so return c at b	00 00111 010 011
	+c,1		111 00110 101
	=a,a,L0	//continue	00 010 010 1110101

IRA

IRA is a unary 2-routine that takes an integer ≥ 1 stored in register a as input for the length of the reference array U it is about to initialize. Global register U will contain the initialized reference array of length 7, such that U= [1,0,2,0,3,0,4], where 1,2,3,4 are the encoded values of the base symbol and the operators ^,* and +, respectively, see encoding schema (3).
The reference array is used in PHR computations. No formal output is generated. IRA then returns to b.

Label	Instruction	Comments
IRA	&b**h**,b	// save return address
	&b,a	// Load 7 as length
	&c,L0	
	=a,a,AI	// init global array of length 7
L0	&A,a	// store initialized array in global A
	&b,1	// encode base symbol
	&I,1	// point to first element
L1	&c,L2	
	=a,a,REPL	// set A[I]=b
L2	+I,2	// point to over next element
	+b,1	// next encoding value
	=b,5,L3	// we are done with initializing
	=a,a,L1	// continue initializing the array
L3	&U,A	// load reference array to U

[3] In this binary representation the instruction label has been replaced with its actual line number 5; which will change depending on where GS is located in the final source code. Register B is chosen here to be the 5th in line after a, b, c, d.

	&b,b**h**	// restore return address
	=a,a,b	// return to caller

GS

GS is a unary 2-routine that takes an integer ≥ 1 stored in register a as input and sets the global output register OG to 1 if the associated Goodstein sequence GS(a) = $(a)_k$ terminates at zero for some finite k. Otherwise OG remains to be 0. It then returns to b.

Label	Instruction	Comments
GS	&a**h**,a	// save input
	&b**h**,b	
	&B,1	// init global register for base
	&g**h**,a	// load seed
	&OG,0	// set output register to 0
L01	=g**h**,0, L05	// if GS seq terminates, branch
	+B,1	// increase base
	&a,g**h**	
	&b, L02	
	=a,a,HER	// hereditary rep of a in base B
L02	&c, L03	
	=a,a,CBF	//change of base function
L03	&a,d	// load next element of GS sequence
	&c, L04	
	=a,a,SUBT1	// subtract 1
L04	&g**h**,d	//reload sequence register
	=a,a,L01	//continue
L05	&OG,1	// set output register to 1
	&a,a**h**	// reload register
	&b,b**h**	
	=a,a,b	// return to caller

HER

HER is an unary 2-routine that takes as input a non-negative integer a and computes its complete level 3 Hereditary Representation [3,11] in base B and stores it in array A and the length of the array in N. HER has no formal output and returns to the line number stored in register b.

HER uses recursive calls to depth-1 Hereditary Representation PHR to start with, then scans each integer in the 7-element representation and recursively replaces all integer by their PHR if the integer is equal or greater than the base value B.

Label	Instruction	Comments
HER	&a**h**,a	// save input
	&b**h**,b	
	&k**h**,5	// init the encoding constant
	+k**h**,B	// encode current base value B : enc(B)
	&b,B	// load base value
	&c,L00	
	=a,a,PHR	// call depth-1 hereditary representation
L00	&A,d	// save 7-element array to A
	&N,7	// save 7 as length to N
L10	&h**h**,1	// prepare for scanning A
L20	&I,h**h**	// load scanning index
	&c,L01	
	=a,a,ELM	// fetch d= A[I]
L01	&a,d	
	&b,k**h**	
	&c,L02	
	=a,a,CMP	//compare A[I] with enc(B)
L02	=d,1, L05	// branch if A[I] < enc(B)
	&b,L03	// A[I] is an encoded integer ≥ enc(B)
	=a,a,DECODE	// decode the integer
L03	&a,c	//load decoded value to a
	&b,B	// load value of base to b
	&c,L04	
	=a,a,PHR	// compute PHR for decoded A[I]
L04	&a,d	
	&b,7	
	&c,L10	// branch back to restart scanning
	=a,a,MERGE	// MERGE arrays a with A
L05	=h**h**,N,L06	// branch if done
	+h**h**,1	// point to next element to scan
	=a,a,L20	// continue scanning
L06	&a,ah	// reload register
	&b,bh	
	=a,a,b	// return to caller

PHR

PHR is a binary routine that takes as input two non-negative integers a and b , such that $b \geq 1$ and computes the depth-1 level-3 partial hereditary representation of integer a in base b, such that

$$a = b \wedge x * y + z,$$

where x is the largest integer such that $b^x \leq a$, $y = FLOOR[a/ b^x]$, and $z = a - y* b^x$.

Partial refers to the fact, that some of the x, y, z values are not necessarily smaller than the base value b.

PHR returns an integer in register d and returns to line number stored in register c.
The returned integer in register d represents a 7-element array data structure in reverse Polish notation [b, x, ^, y, *, z, +], where each symbol is represented by its encoding (3).

Example:

PHR(266,2) returns d = 1,407,394,228,731,920 (decimal)

This is seen as follows:
266 = 2^8*1+10, which is represented as the 7-element list [b,8,^,1,*,10,+] and plugging in the encoding (3) gives the array [1,13,2,6,3,15,4]. Using Dinneen's [1] method this array is encoded as

$$10_1 10_{13} 10_2 10_6 10_3 10_{15} 10_4$$

where we use the shorthand 0_n to represent n 0s.This binary is the indicated decimal integer returned in register d.

Label	Instruction	Comments
PHR	&a**h**,a	// save input
	&b**h**,b	
	&c**h**,c	
	&X,0	// reset global register for PHR
	&Y,0	
	&Z,0	
	&e**h**,0	// reset working register
	&f**h**,L100	// set jump label
	&g**h**,L200	// set jump label
L0	&c,L000	
	=a,a,CMP	// compare a and b
L000	=d,0,f**h**	// case a = b, go to f
	=d,1,g**h**	//case a < b , go to g
	&f**h**,L300	// case a > b, set new jump label
	&g**h**,L400	// set new jump label
	&h**h**,e	// save e to h
	+e**h**,1	// increase e
	&a,b**h**	
	&b,e	
	&c,L001	
	=a,a,POW	// compute b^e
L001	&b,d	
	&a,a**h**	
	=a,a,L0	// proceed and compare a and b^e
L002	&k**h**,A	// beginn building the output array
	&A,U	// load reference array to A
	&I,0	
	&b,X	// prepare to load X to A[2]
	&c,L020	
L010	+I,2	
	+b,5	// load encoding constant
	=a,a,REPL	// set A[I]=b
L020	&b,Y	// prepare to load Y to A[4]
	&c,L030	
	=a,a,L010	
L030	&b,Z	// prepare to load Z to A[6]
	&c,L500	//prepare to exit
	=a,a,L010	// continue
L100	&X,1	// case : a= b
	&Y,1	
	=a,a,L002	//build array

L200	&Z,a	// case a < b
	=a,a,L002	//build array
L300	&X,e**h**	// case : a = b^e
	&Y,1	
	=a,a,L002	//build array
L400	&X,h**h**	// case a = b^{e-1} continue computing Y, Z
	&b,h**h**	// store (e-1) to b
	&a,b**h**	// load b
	&c,L401	
	=a,a,POW	// compute b^{e-1}
L401	&b,d	// load result to b
	&a,a**h**	// load original value of a
	&c,L402	
	=a,a,DIV	// compute d= FLOOR[a/b^{e-1}]
L402	&Y,d	// load result to Y
	&a,d	//... and a
	&c,L403	
	=a,a,MUL	// compute d = y * b^{e-1}
L403	&b,d	// load result
	&a,a**h**	// reload original value of a
	&c,L404	
	=a,a,SUB	// compute d = a- y* b^{e-1}
L404	&Z,d	//load result to Z = a- y* b^{e-1}
	=a,a,L002	// build array
L500	&d,A	// d = A=[b,X,^,Y,*,Z,+]
	&A,k**h**	// restore A
	&a,a**h**	// reload register
	&b,b**h**	
	&c,c**h**	
	=a,a,c	// return to caller

CBF

Change of Base Function (CBF): Given the N-element representation of A, this 2-routine computes a new integer n by using the value (B+1) for each occurrence of a base symbol b in A.
CBF uses global registers A, B,N, I. It returns the computed result in register d. CBF returns to the line number stored in register c.

Label	Instruction	Comments
CBF	&a**h**,a	//save input registers
	&b**h**,b	
	&c**h**,c	
	&g**h**,0	// init computational stack
	&I,0	//init index to global array
L00	+I,1	//next scanning index
	&c,L01	
	=a,a,ELM	// fetch A[I] and return in d
L01	=d,2, L04	// this is the POW operator
	=d,3, L04	// this is the MUL operator
	=d,4, L04	// this is the + operator
	&b,d	// encoded integer k (5+k) or base symbol
	&a,g**h**	// load computational stack to register a
	&c, L03	
	=a,a,APPEND	// APPEND integer or base symbol to computational stack
L03	&g**h,a**	//reload computational stack
	=I,N, L05	// If I= N then EOF of array scanning
	=a,a, L00	// continue scanning the array
L04	&b,d	// load the operator code
	&a,g**h**	// load the computational stack
	&c, L03	
	=a,a,EXO	// execute the operator on the computational stack
L05	&a,a**h**	//restore input registers
	&b,b**h**	
	&c,c**h**	
	&d,g**h**	// load result
	=a,a,c	// return to caller

EXO

EXecute **O**perator takes as input in register a an array and in register b an operator code as assigned by the encoding schema (3):

- b = 2 : operator is ^
- b = 3 : operator is *
- b = 4 : operator is +

EXO is not producing any formal output. It updates the array a inline and returns directly to the line specified in register c.

Assume that a has length n, then the operator is applied to the last two (two rightmost) elements in array a as follows: a[n-1] OP a[n]

By construction, a[n-1] and/or a[n] are either an encoded base symbol (value 1) or an encoded integer k (value 5+k). Before we apply OP, we need to decode the numeric values in the proper form, as follows:

- value 1 (base symbol) is transformed into value (B+1), where global register B tracks the current value of the base b in the computation.
- value 5+k is transformed into k
- then OP is applied to calculate r = a[n-1] OP a[n] and the result is encoded again in the form R= 5 + r .
- We then use DELETE to delete a[n-1] and a[n] and APPEND R to a.

Example: assume current value B = 5, a = [a_1,a_2, ..., a_N,1,7] and b = 3 (i.e. exponentiation). The computation yields 6^2 = 36 which is encoded back to 41.Hence EXO produces the updated array a = [a_1,a_2, ..., a_N,41].

Label	Instruction	Comments
EXO	&a**h**,a	//save input registers
	&b**h**,b	
	&c**h**,c	
	&b, L01	
L00	=a,a,DELETE	
L01	&f**h**,c	// save deleted element a_n to f**h**
	&b, L02	
	=a,a,L00	
L02	&g**h**,c	// save deleted element a_{n-1} to g**h**
	&ah,a	// save updated array
	&b,L04	
	&a,f**h**	
L03	=a,a,DECODE	// decode
L04	&f**h**,c	// save decoded value for a_n
	&b**,** L05	
	&a,g**h**	// load second parameter
	=a,a, L03	// another Decode for the second value
L05	&g**h**,c	// save decoded value for a_{n-1}
	&a,g**h**	// load a_{n-1} to a
	&b, f**h**	// load a_n to b
	&c,L06	
	=b**h**,4, L05	// this is the + operator
	=b**h**,3, L04	// this is the * operator
	=a,a,POW	// compute a^b
L04	=a,a,MUL	// compute a*b
L05	+a,b	//compute a+b
	&d,a	// load to a+b
L06	+d,5	// add decoding constant
	&b,d	// prepare for append
	&a,a**h**	// prepare for append
	&c, L07	// append result to the array
	=a,a,APPEND	
L05	&a,a**h**	//restore input registers
	&b,b**h**	
	&c,c**h**	
	=a,a,c	// return to caller

Registry of Handles

In order to resolve the generic double letter register names such as a**h**, b**h**, c**h** ,..., we need to define the value for the handles **h.**
We anticipate and propose here to use a public open source repository such as http://sourceforge.net/ to publish approved 2-routines and their handles. Future publications can then refer to that repository and use the registered handles to resolve names without any ambiguity and regardless of names of routines. As this is not currently in place we provide here a local registry for all 2-routines used in this work.

Name of 2-Routine	Handle **h** to resolve register names
CBF	0
DELETE	1
ELM	2
EXO	3
GS	4
HER	5
MERGE	6
PHR	7
POW	8

Example: register names in MERGE will be resolved to a6, b6, c6 ... The uniqueness of the handle **h** in the registry ensures that we never assign conflicting register names. A further advantage is , that the process of resolving generic register names can be automated by using a pre-compiler step.

Final Comments

The main source of complexity of the Goodstein Theorem comes from the necessity to handle the hereditary representation of an integer and to implement the change of base function operating on it in order to calculate the numerical values of each actual element of the Goodstein sequence of any given seed integer n.

Analyzing the complete source file[4] we find that all the routines in the Application Library produce a total bit count of 6057. The total bit count for reused routines of the Arithmetic [1,13] and Array library [12] is 1852, giving a total bit count of 7909 for the Goodstein Theorem evaluation package.

Hence we classify Goodstein's Theorem to be in inductive complexity class $\mathbf{C}_{U,7}^{ind,2}$. We estimate that by further streamlining the register machine code , streamlining the use of pre-existing library routines and re-allocating register usage we might be able to shift the package down to the complexity class $\mathbf{C}_{U,6}^{ind,2}$, still a considerably high class. To reduce the complexity measure even further, new ideas for tackling the Goodstein Theorem at the register machine level are needed.

It would be interesting to compare the inductive complexity of the Goodstein Theorem with other "natural" statements independent from Peano arithmetic PA, such as the Paris-Harrington Principle [14], which is a consequence of the Infinite Ramsey Theorem and cannot be proved in PA, or the Kirby & Paris Hydra Game [4] which is another combinatorial statement that is true but unprovable in first order Peano arithmetic.

We also want to point to the very interesting field of phase transition from provability to unprovability of statements independent from PA. Weiermann [15] has calculated the critical point g_{crit} in the phase transition for the Goodstein Theorem. If the change of base function can be bounded by certain slow growing functions (see [15] for details) the Goodstein Theorem (GT) becomes provable in PA and we should be able to write a program Π_{GT} that never stops iff GT is TRUE. Hence Π_{GT} semi-decides GT and we can measure the complexity of $\Pi_{GT.}$ by using the first complexity measure as specified in [5,6,7].
Can we somehow parameterize the register machine model in a way that reflects the phase transition from provability to unprovability if we modify the procedure of changing the base function as it passes the critical point? How is the phase transition reflected in the inductive complexity class? Is there a critical point where we observe a sharp jump in inductive complexity when we vary the change of base function in the Goodstein sequence?

Acknowledgement

The author thanks Cristian S. Calude for an early review of a draft version of this work.

[4] Details are available from the author on request

Appendix

We show here a sample implementation of the Power function.

POW

POW is a binary 2-routine that take two non negative integers a > 0 and b > 0 stored in registers a and b as input and returns in register d = a^b . It then returns to line number stored in c.

Label	Instruction	Comments
POW	&a**h**,a	// save input
	&b**h**,b	
	&c**h**,c	
	&d,1	// set return value to 1
	=a,1,c	// and return if a =1
	&d,a	//set return value to a
	=b,1,c	// and return if b= 1
	&e,1	// init counter
	&b,a	// set b to a
L0	&c,L1	
	+e,1	// increase counter
	=a,a,MUL	//compute $a^k * a = a^{k+1}$
L1	&a,d	
	=e,b**h**,L2	// branch if done
	=a,a,L0	// continue
L2	&a,a**h**	// reload register
	&b,b**h**	
	&c,b**h**	
	=a,a,b	// return to caller

References

[1] C.S. Calude, Elena Calude. The Complexity of Mathematical Problems: An Overview of Results and Open Problems, CDMTCS Research Report 410, 2011.
[2] M. Burgin, C.S. Calude, Elena Calude. Inductive Complexity Measures for Mathematical Problems, CDMTCS Research Report 416, 2011.
[3] R. Goodstein, On the restricted ordinal theorem, Journal of Symbolic Logic 9: 33–41,1944.
[4] L. Kirby, L.; J. Paris., Accessible independence results for Peano arithmetic, Bulletin of the London Mathematical Society 14: 285–293, 1982.
[5] C. S. Calude, E. Calude, M. J. Dinneen. A new measure of the difficulty of problems, Journal for Multiple-Valued Logic and Soft Computing 12 (2006), 285–307.
[6] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 1 Complex Systems, 18-3 (2009), 267–285.
[7] C. S. Calude, E. Calude. Evaluating the complexity of mathematical problems. Part 2 Complex Systems, 18-4, (2010), 387–401.
[8] M. Burgin. Super-recursive Algorithms, Springer, Heidelberg, 2005.
[9] M. Burgin. Algorithmic complexity of computational problems, International Journal of Computing & Information Technology 2,1 (2010), 149–187.
[10] M. Burgin. Measuring Power of Algorithms, Computer Programs, and Information Automata,Nova Science Publishers, New York, 2010.
[11] A. Caicedo, Goodstein's function, *Revista Colombiana de Matemáticas* **41,** 381–391,2007.
[12] M. J. Dinneen. A program-size complexity measure for mathematical problems and conjectures, in M. J. Dinneen, B. Khoussainov, A. Nies (eds.). Computation, Physics and Beyond, LNCS 7160, Springer, Heidelberg, 2012, 81–93.
[13] C.S. Calude, Elena Calude, Melissa S.Queen. The Complexity of Euler's Integer Partition Theorem, CDMTCS Research Report 409-revised, 2012.
[14] J. Paris, and L. Harrington, *A Mathematical Incompleteness in Peano Arithmetic*. In Handbook for Mathematical Logic (Ed. J. Barwise). Amsterdam, Netherlands: North-Holland, 1977.
[15]A. Weiermann, Classifying the phase transition for hydra games and Goodstein sequences available at http://cage.ugent.be/~weierman//phase.html

[i] E-mail address : jhertel@h-star.com
Current address : H-Star, Inc 20801 Biscayne Blvd 4th Floor Aventura, FL 33180 USA