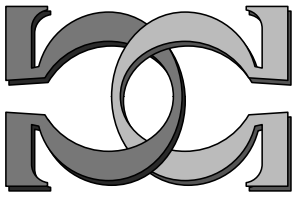
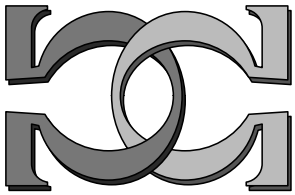
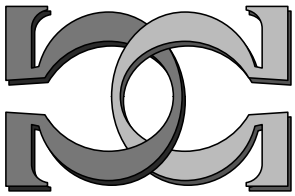
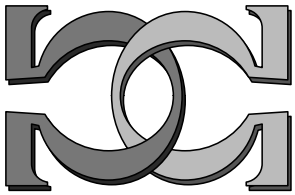


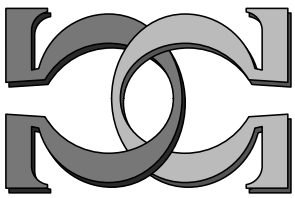




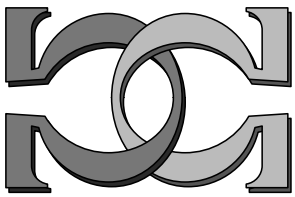
CDMTCS Research Report Series



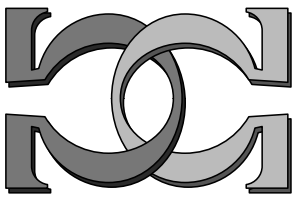
Exact Approximations of Omega Numbers



C. S. Calude and M. J. Dinneen
University of Auckland, NZ



CDMTCS-293
December 2006



Centre for Discrete Mathematics and
Theoretical Computer Science

Exact Approximations of Omega Numbers

Cristian S. Calude and Michael J. Dinneen
Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand
http://www.cs.auckland.ac.nz/~{cristian,mjd}

Abstract

A Chaitin Omega number is the halting probability of a universal prefix-free Turing machine. Every Omega number is simultaneously computably enumerable (the limit of a computable, increasing, converging sequence of rationals), and algorithmically random (its binary expansion is an algorithmic random sequence), hence uncomputable. The value of an Omega number is highly machine-dependent. In general, no more than finitely many scattered bits of the binary expansion of an Omega number can be exactly computed; but, in some cases, it is possible to prove that no bit can be computed.

In this paper we will simplify and improve both the method and its correctness proof proposed in an earlier paper, and we will compute the exact approximations of two Omega numbers of the same prefix-free Turing machine, which is universal when used with data in base 16 or base 2: we compute 43 exact bits for the base 16 machine and 40 exact bits for the base 2 machine.

1 Introduction

Chaitin Omega number is the halting probability of a universal prefix-free (self-delimiting) Turing machine. Every Omega number is simultaneously computably enumerable (the limit of a computable, increasing, converging sequence of rationals), and algorithmically random (its binary expansion is an algorithmic random sequence). Any Omega number is more than uncomputable: no more than finitely many scattered bits of the binary expansion of an Omega number can be exactly computed. The value of an Omega number is highly machine-dependent; in some cases, it is possible to prove that no bit can be computed.

This paper describes a simplification and improvement of the hybrid procedure proposed in [5] (which combines Java programming and mathematical proofs) for computing exact approximations of Omega numbers. We will apply this technique to a compact prefix-free (Turing) machine which will be proved to be universal when computing in base 16 and in base 2; we will compute exact approximations for the corresponding Omega numbers.

Computing lower bounds is not difficult: we just generate more and more halting programs and add their contributions to Omega. Computing upper bounds is much more demanding. The idea is to systematically run more and more programs, filter out all non-halting programs and try to use this information to obtain better and better upper bounds. If the first bit of the approximation happens to be 1, then sure, it is

exact. However, if the provisional bit given by an approximation is 0, then, due to possible overflows, this bit is uncertain. This situation extends to other bits as well. Only an initial run of 1's gives exact values for some bits of the Omega number. Of course, we can compute exact 0 digits, but as a result of balancing the lower and upper bounds.

The paper is structured as follows. Section 2 introduces the notation and the basic background in algorithmic information theory: computably enumerable (c.e.) reals, program-size complexity, random reals and c.e. random (Omega) reals. In Section 3 we present the simplified new prefix-free machine and in Section 4 we describe a simulator for it. In Section 5 we present and justify our procedure for obtaining approximations. In Section 6 we present the result justifying the computation of the first exact 43 bits of Omega for the case when the prefix-free machine works with data in base 16. In Section 7 we present the results of approximation for the Omega for the same prefix-free machine, this time working with data in base 2. The last section includes some final comments. The appendix contains some details about the computation process and the data used for the computation.

2 Notation and theoretical background

We will use notation that is standard in algorithmic information theory; we will assume familiarity with Turing machine computations, computable and computably enumerable (c.e.) sets (see, for example, [14, 15]) and elementary algorithmic information theory (see, for example, [1]).

By $\mathbf{N}$ we denote the set of non-negative integers (natural numbers). Let $\Sigma = \{0, 1\}$ denote the binary alphabet. Let Σ^* be the set of (finite) binary strings, and Σ^ω the set of infinite binary sequences. The length of a string x is denoted by $|x|$. A subset A of Σ^* is *prefix-free* if whenever s and t are in A and s is a prefix of t, then $s = t$.

For a sequence $\mathbf{x} = x_0x_1 \cdots x_n \cdots \in \Sigma^\omega$ and integer $n \geq 1$, $\mathbf{x}(n)$ denotes the prefix of length n of $\mathbf{x}$ and x_i denotes the ith digit of $\mathbf{x}$, i.e. $\mathbf{x}(n) = x_0x_1 \cdots x_{n-1} \in \Sigma^*$.

A *prefix-free Turing machine*, shortly, a *machine*, C is a Turing machine processing binary strings such that its program set (domain) $PROG_C = \{x \in \Sigma^* \mid C(x) \text{ halts}\}$ is a prefix-free set of strings. Clearly, $PROG_C$ is c.e.; conversely, every prefix-free c.e. set of strings is the domain of some machine. The *program-size complexity* of the string $x \in \Sigma^*$ (relatively to C) is $H_C(x) = \min\{|y| \mid y \in \Sigma^*,\ C(y) = x\}$, where $\min \emptyset = \infty$. A major result in algorithmic information theory is the following universality theorem:

Theorem 1 *We can effectively construct a machine U (called* universal*) such that for every machine C, there is a constant $c > 0$ (depending upon U and C) such that for every $x, y \in \Sigma^*$ with $C(x) = y$, there exists a string $x' \in \Sigma^*$ with $U(x') = y$ and $|x'| \leq |x| + c$.*

In complexity-theoretic terms, $H_U(x) \leq H_C(x) + c$. Note that $PROG_U$ is c.e. but not computable.

If $S \subset \Sigma^*$ is prefix-free, then $\Omega_S = \sum_{s \in S} 2^{-|s|}$. If C is a machine, then $\Omega_C = \Omega_{PROG_C}$ represents its halting probability (see more detailed probability facts in [1]). When $C = U$ is a universal prefix-free machine, then its halting probability Ω_U is called a *Chaitin Omega number*, shortly, *Omega number*, [9, 10].

There are many equivalent ways to define (algorithmic) random sequences, see [1]. For our aim we will use the following complexity-theoretic definition (see [9]): an infinite sequence $\mathbf{x}$ is *(algorithmically) random* if there is a constant c such that $H(\mathbf{x}(n)) > n - c$, for every integer $n > 0$. A real α is *(algorithmic) random* if its binary expansion $\mathbf{x}$ (i.e. $\alpha = 0.\mathbf{x}$) is random. The choice of the binary base does not play any role, cf. [6, 12, 17]: randomness is a property of reals not of names of reals.

A real α is called *computably enumerable (c.e.)* if there is a computable, increasing sequence of rationals which converges (*not necessarily computably*) to α.

The following characterization of c.e. and random reals was proved in the following series of papers [9, 7, 13] (see also [2, 1]):

Theorem 2 *Let $\alpha \in (0, 1)$. The following conditions are equivalent:*

1. *The real α is c.e. and random.*
2. *There effectively exists a universal machine U such that $\alpha = \Omega_U$.*

In contrast with π, for which, given enough computational resources, one can compute as many as we wish digits of its binary expansion, for any Omega number we cannot compute more than finitely many digits, [9]. We present this result in terms of provability capacity of ZFC (Zermelo-Fraenkel set theory with choice):

Theorem 3 *Assume that ZFC is arithmetically sound, that is, any theorem of arithmetic proved by ZFC is true. Then, for every universal machine U, ZFC can determine the value of only finitely many bits of Ω_U.*

As the value of an Omega number is highly machine-dependent, the difficulty of computing the initial bits of an Omega number varies considerably, from no bit to as many but finitely. To understand this phenomenon we need the following concept.

A machine U for which Peano Arithmetic can prove its universality and ZFC cannot determine more than the initial block of 1 bits of the binary expansion of its halting probability, Ω_U, is called *Solovay machine*.[1] In [3] the following result was proved:

Theorem 4 *Assume that ZFC is arithmetically sound. Then, every c.e. random real is the halting probability of a Solovay machine.*

For example, if $\alpha \in (3/4, 7/8)$ is c.e. and random, then in the worst case ZFC can determine its first two bits (11), but no more. For $\alpha \in (0, 1/2)$ we obtain Solovay's famous result (which also motivates the name "Solovay machine") [16]:

Theorem 5 *Assume that ZFC is arithmetically sound. Then, every c.e. random real $\alpha \in (0, 1/2)$ is the halting probability of a Solovay machine which cannot determine any single bit of α. No c.e. random real $\alpha \in (1/2, 1)$ has the above property.*

In general only the initial run of 1's (if any) can be exactly computed.

[1] Clearly, U depends on ZFC.

3 A simplified language for register machine programs

The Omega numbers studied in this paper will be defined, following [10], using an implementation based on a register machine language.

Any register machine has a finite number of registers, each of which may contain an arbitrarily large non-negative integer. The list of instructions is given below in two forms: our compact form and its corresponding Chaitin style version, [10]. The current model is "denser" than both models used in [10, 5]. The main difference between the implementations in [10, 5] and ours is in the encoding: we use 4 bits instead of 7 or 8 bits per character. The data will be given in two different formats: for the base 16, as a string of 4-bit characters (that is, the read instruction reads 4 raw bits (1 character) at a time), for the base 2 as 1-bit strings (the read instruction reads 1 raw bit at a time).

By default, all registers, labelled with a string of 'a' to 'h' characters, are initialized to 0. It is a syntax error if the first occurrence of register j appears before register i in a program, where j is lexicographic greater than i. Also, all registers lexicographic less than j must have occurred in the program.

To summarize, the alphabet of the machine instructions consists of 16 elements, $\Sigma_{16} = \{a, b, c, d, e, f, g, h, i, \text{comma}, =, \&, !, \%, 0, 1\}$. For the base 16 model the data uses the alphabet Σ_{16}; for the base 2 model the data uses the alphabet $\Sigma_2 = \{0, 1\}$. The concatenation of strings x and y is denoted by $x \cdot y$. The character length of a string x is denoted by $|x|$. If $w = x \cdot d$ is a program consisting of string of instructions x followed by data d (also a string), then the *program (bit) length* of w, denoted by $\|w\|$ is $4|w|$ and $4|x| + |d|$ in the base 16 and base 2 models, respectively.

The register machine instructions are listed below. Note that in all cases R2 denotes either a register or a binary constant of the form $1(0+1)^* + 0$, while R1 and R3 must be a register variable.

= R1, R2, R3 **(EQ R1 R2 R3)**

If the contents of R1 and R2 are equal, then the execution continues at the R3-th instruction, where R3 = 0 denotes the first instruction. If they 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 **(SET R1 R2)**

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

+ R1, R2 **(ADD R1 R2)**

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

! R1 **(READ R1)**

For the base b model, b bits are read into the register R1, so the numerical value of R1 becomes at most $b - 1$. Any attempt to read past the last data-bit results in a run-time error.

% **(HALT)**

This is the last instruction for each register machine program before the raw data. It halts the execution in two possible states: either successfully halts or it halts with an under-read error (i.e., not all data read).

A *register machine program* consists of a finite list of labelled instructions from the above list, with the restriction that the `HALT` instruction appears only once, as the last instruction of the list. The data (a base 2 or base 16 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.

A *canonical program* is a register machine program in which (1) labels appear in increasing numerical order starting with 0, (2) new register names appear in increasing lexicographical order starting from 'a', (3) there are no leading or trailing spaces, (4) operands are separated by a single space, (5) there is no space after labels or operators, (6) instructions are separated by a single space, (7) delete all labels as they implicitly appear in increasing order starting with 0, (8) delete spaces and the colon symbol with the first non-data instruction having an implicit label 0, (9) separate multiple operands by a single comma symbol.

Note that *for every register machine program there is a unique canonical program which is equivalent to it*, that is, both programs have the same domain and produce the same output on a given input. If x is a program and y is its canonical program, then $|y| \leq |x|$.

We next give an example of a canonical program that reads two 16 bit integers and computes the product by using a "subroutine" for multiplication.

To aid the presentation and development of these programs (see [4] for more realistic examples) we use a consistent style for subroutines, using the following conventions:

1. The letter 'L' followed by characters (usually $1, \ldots, 9$) and terminated by ':' is used to mark line numbers. These are local within the subroutine. References to them are replaced with the binary constant in the final program.

2. For unary subroutines, registers a = argument, b = return line, c = answer (a and b are unchanged on return). In our example, we have a subroutine READINT that takes an argument a denoting the number of binary bits to read and returns a register c that binary number.

3. For binary subroutines, registers a = argument1, b = argument2, c = return line, d = answer (a, b and c are unchanged on return). In our example, we have a subroutine MULT that takes registers a and b and multiplies them together and returns the product in register d.

4. For Boolean data types, as expected, we use integers 0 = `false` and 1 = `true`. Also, the characters '0' and '1' are assumed to be encoded numerically as 0 and 1, in the 4-bit (base 16) data model,

```
&a,Main
=b,c,a      // b=c=0 so jumps to line denoted by "Main".
```

```
&d,0        // MUL(a,b) returns a*b
=a,0,c
=b,0,c      // if a or b is zero return current product
&e,L1
&f,1
+d,a        // keep adding a to our answer d for b times.
L1: =f,b,c
+f,1
+d,a
=a,a,e

&c,0        // READINT(a) return an a-bit positive integer b
&d,0
=a,0,b      // return if a is not at least 1
L1: +c,c    // shift all c's bits left and add next bit
!e
+c,e        // read bit into e and add to c
+d,1
=a,d,b
&e,L1
=a,a,e

&a,10000    // Main: example multiplies two 16 bit integers
&b,L1
&d,READINT
=a,a,d
L1: &f,c   // save first integer
&b,L2
&d,READINT
=a,a,d
L2: &a,f   // recover argument 'a' for MULT
&c,L3
=a,a,MUL
L3: &a,d   // move the answer to register a
%          // halt
1100001110111110
```

The actual (non-commented) version of the previous example looks like this (where one reads the instructions down the columns then across the rows):

```
&a,10110      +f,1          +d,1          &b,11110
=b,c,a        +d,a          =a,d,b        &d,1100
&d,0          =a,a,e        &e,1111       =a,a,d
=a,0,c        &c,0          =a,a,e        &a,f
=b,0,c        &d,0          &a,10000      &c,100001
&e,1000       =a,0,b        &b,11010      =a,a,10
&f,1          +c,c          &d,1100       &a,d
+d,a          !e            =a,a,d        %
=f,b,c        +c,e          &f,c          1100001110111110
```

Here the line address of `Main` is 22 (10110), `MUL` is 2 (10), and `READINT` is 12 (1100). Finally newlines and spaces are not part of a register machine program so we actually have the following (wrapped) string with data over the alphabet Σ_{16}:

```
&a,10110=b,c,a&d,0=a,0,c=b,0,c&e,1000&f,1+d,a=f,b,c+f,1+d,a=a,a,e&c,0&d,
0=a,0,b+c,c!e+c,e+d,1=a,d,b&e,1111=a,a,e&a,10000&b,11010&d,1100=a,a,d&f,
c&b,11110&d1100=a,a,d&a,f&c,100001=a,a,10&a,d%1100001110111110
```

Our register machine language can be used in two forms: (a) in base 2 or (b) in base 16, depending on whether the READ instruction (!) reads the data 1 bit at a time or 4-bit nibbles at a time, respectively. In the above example, we assumed the program encodes integers in a binary text string (either model). This is convenient, but not the most space efficient way, to represent integers if we are working in base 16 (i.e., we ignore 3 bits out of every 4 data bits). The reader can easily modify the example to use all 4 bits of data by modifying the subroutine READINT(a) to shift 4 bits to the left instead of 1 (see register c on line L1).

It is easy to prove, using Chaitin's standard technique [10], that, in both cases, the simplified register machine language implements a universal machine (working in base 16 or base 2).

3.1 A simulator for register machine programs

We have implemented a Java program `Simulate(String p, int jtime)` that reads in a register machine program `p` and an integer `jtime` and simulates the running of the program for at most `jtime` executions of an EQ instruction (branch). Here `p` consists of a program (base 16) + data (either base 2 or base 16) and `jtime` is an upper bound on the number of instruction jumps that occur in the simulation.

The simulator first tests the program `p` for syntactical correctness. If the test is not passed, then it returns:

5 – syntax error

If the test was passed, then it simulates the computation of the program on the given data until one of the following conditions have been reached. It then returns:

0 – underread (halts but not all data read)
1 – overread (error as the result of trying to read past end of data)
2 – illegal branch (error in the EQ instruction)
3 – halt success
4 – loop condition (exceeded `jtime`)

4 The procedure to approximate Omega

Our simulator was used to test the halting problem for all register machine programs of a certain length (in both bases). The results have been obtained according to the following procedure (which is adapted from [5]), elaborated here for the base 16 case:

1. Start by generating all programs of 4 bits (1 character) and test which of them stops.[2]. All strings of length 4 which can be extended to programs are considered prefixes for possible halting programs of longer length; they will simply be called *prefixes*. In general, all strings of length n which can be extended to programs are *prefixes* for possible halting programs of length $n+4$ or longer. *Canonical prefixes* are prefixes of canonical programs.

2. Testing the halting problem for programs of length $n \in \{4, 8, 12, \ldots, 80\}$ was done by running all candidates (that is, programs of length n which are extensions of prefixes of length $n-4$) for up to `jtime=1000`.[3]

The most important step of our procedure is *to solve the halting problem for small-size programs.* We establish that a program does not halt by using these three methods:

1. Syntactically eliminate correct programs which contain "obvious" loops or correct programs which have a prefix which loops "onto itself" (that is, the loop does not involve other instructions than those in the prefix).

2. Simulate any non-filtered program for `jtime=1000`; if the computation does not stop, then store the program for further analysis (a combination of special syntactic tests and eventually manual screening).

3. Perform a global analysis of possible errors (mainly due to manual screening) to ensure that they do not affect the final result.

A simple example of a prefix filter is `=a,a,a`. This filter could have been discovered "automatically" by observing, via the simulator, that the program `=a,a,a%` exceeded our `jtime` limit. Thus, the filter pattern `=a,a,a<SEQ>%`, where `<SEQ>` is any sequence of instructions will also loop. In general, any program `<LSEQ>%<DATA>` (after human verification) that loops will imply that `<LSEQ><SEQ>%<DATA>` also loops. Here `<LSEQ>` is a sequence of instructions (which, in fact, will never be encountered) and `<DATA>` denotes the possible data that the program reads. Since we are mainly concerned about counting only the halting programs, we can also check for prefixes (when extended) that would always yield 'overread' case (e.g. the simulator returns 1).

The simulation of remaining programs obtained by extending the set of canonical prefixes was done with the simulator described earlier. Note that a canonical prefix is one of two types: (a) a syntactically correct sequence of instructions without the HALT instruction (`%`) and (b) a syntactically correct program that the simulator returns 1

[2] In this trivial case only the program `%` halts.

[3] Note that in the earlier work [5] we simulated each program up to an upper bound of 100 steps. In our case, if we encounter the return value of 5 from the simulator then this implies that we have executed at least `jtime` (and probably much more) steps.

('overread' data case). The enumeration process is quite efficient since we know that only very few short programs stop after a very long time, see [8]. The main bottleneck in our approach is having to store all canonical prefixes (filtered) of a given length in preparation for counting the halting programs of the next length.

Finally, the remaining programs, which have been considered "non-halting" have been screened for further signs of "looping-ness". This screening was partially automated. Although the screening process was very thorough, due to the largeness of analyzed data some mistakes could have been introduced for large programs, i.e. for programs of length greater than 17. Is our analysis fatally flawed? To prove that this is not the case we assumed that for programs of length greater than 17 the proportion of mistakes was up to 5% of the corresponding counts (which implies a huge variation in our data, easy to be discounted); in this case we prove the values of the exact calculated bits (first 43) remain unchanged.

We end this section with a result that is useful in proving upper bounds.

Theorem 6 *Let P denote the set of all canonical prefixes that are overreads of (program) length k. The contribution of halting extensions of P to Ω_{Ub}, using the base b data model, is at most $b \cdot |P| \cdot 2^{-k-\lg_2(b)} = |P| \cdot 2^{-k}$.*

Proof. Here is a short proof (suggested by [18]). The set of halting suffixes extending some prefix $x \in P, \{z \mid Ub(xz) \text{ halts}\}$, is a prefix-free set itself, and thus, by virtue of Kraft's inequality,

$$\sum_{\{z|Ub(xz) \text{ halts}\}} b^{-|z|} \leq 1.$$

Accordingly, the contribution of all halting extensions of prefixes in P to Ω_{Ub}, expressed in base 2, is at most

$$\sum_{\{z|x\in P, Ub(xz) \text{ halts}\}} 2^{-(k+|z|\lg_2(b))} = |P| \cdot 2^{-k} \cdot \sum_{\{z|Ub(xz) \text{ halts}\}} b^{-|z|} \leq |P| \cdot 2^{-k}.$$

Kraft's inequality somehow obscures the computational phenomenon described in the theorem. For a better understanding we provide more details for the case $b = 2$, where the contribution is at most $|P| \cdot 2^{-k}$.

We claim that the contribution to Ω_{U2} is maximised when for all $p \in P$ both machines $p \cdot 0$ and $p \cdot 1$ halt. Here the contribution to Ω_{U2} with these length $k+1$ halting programs is $2|P| \cdot 2^{-(k+1)} = |P| \cdot 2^{-k}$. Assume this is not the largest case. Then there is some $p \in P$ such that either $p \cdot 0$ or $p \cdot 1$ does not halt. In this case, without loss of generality, if $p \cdot 1$ does not halt (i.e. it's still an overread), we would have a non-empty set

$$X = \{p' = p \cdot 1 \cdot x \mid x \in \{0,1\}^*, |x| \geq 1 \text{ and } p' \text{ halts}\} \neq \emptyset.$$

By the prefix-free property of the domain, if we have $p_1, p_2 \in X$ and $x_1 \neq x_2$, then neither p_1 or p_2 is a prefix of the other. Thus we can view X as a set of leaves of an enumeration binary tree $\{0,1\}^*$ rooted at $r = p \cdot 1$ (possibly of infinite size).

(1) *The finite case:* We prove our result by induction on the height of the tree. If the height is 1, then the worst case is two leaves at distance 1 from r (and strings of length $k+2$). The contribution to Ω_{U2} is at most $2^{-(k+2)} + 2^{-(k+2)} = 2^{-k-1}$ which is the

same contribution as if the machine r would have halted. Now consider the case when the height is $t > 1$. Let T_1 and T_2 be the subtrees rooted at the children of r. The contribution of each subtree to Ω_{U2} is the same as $1/2$ times the contribution if each tree T_i had been rooted at r because of the distances to the leaves (i.e. program lengths are 1 bit longer). Thus, adding together both cases yields a contribution to Ω_{U2} that is at most as if we had subtrees of height at most $t - 1$.

(2) *The infinite case:* First consider the situation where there is a halting extension for every length $t > k+1$. The tree looks like an infinite path with a single pendant hanging off each internal node (to either the left or right) of this path. The amount added to Ω_{U2} is simply

$$\Sigma_{i=k+2}^{\infty} 2^{-i} = 2^{-(k+2)} \cdot \Sigma_{i=0}^{\infty} 2^{-i} = 2^{-k-1}.$$

Since this is the same contribution as if the prefix machine r would have halted, the statement of the lemma is not contradicted. To complete the proof we claim that any infinite enumeration tree rooted at r can also contribute at most this sum to Ω_{U2}. We assume that every internal node has two children, otherwise we can contract an edge and get a better tree with respect to a subtree's contribution to Ω_{U2}. The idea is to successively modify the tree without decreasing it's "halting" contribution to Ω_{U2} and end up with a tree isomorphic to the simple infinite caterpillar tree that we just investigated. Let t denote the smallest distance from r that have at least two leaves. If no such t exists, then we are done. If two of these leaves are siblings, then assume their internal parent node represents a halting program. (It has length one less so the contribution to Ω_{U2} is preserved.) Otherwise, consider the roots r_1 and r_2 of two of these leaves at distance t from r. Swap the subtree (non-leaf child) rooted at r_1 with the leaf node child rooted at r_2. All distances are still preserved in this new tree but now we can eliminate two leaf nodes (since they are now siblings). We can repeat this process until the caterpillar tree emerges. □

5 The first 43 bits of Omega in base 16

The distribution of programs of up to 80 bits for $U16$ is presented in Table 1 in Appendix B. All binary strings representing programs have their length divisible by 4. In this table we do not include counts for all programs that loop, since this property was automatically detected (via a syntactical analysis) and filtered as mentioned earlier.

As a concrete example (referring entries in Table 1 in Appendix B), we have these four extendable prefixes: `!a`, `&a`, `+a` and `=a`. The first overread (at the next level) is `!a%` and this string combined with the seven extendable strings `!a!`, `!a&`, `!a+`, `!a=`, `&a,`, `+a,` and `=a,` form a complete set of eight canonical prefixes of length 12 ($= 3 \times 4$).

At every stage we compute a lower bound and an upper bound for Ω_{U16} in base 16. The lower bound is easy: we just add the contributions of halting programs generated at that stage. Let Ω_{U16}^k be the approximation of Ω_{U16} given by the summation of all halting programs of up to k bits in length, that is,

$$\Omega_{U16}^k = \sum_{\{\|w\| \leq k,\, U16(w)\,\text{halts}\}} 16^{-|w|}. \tag{1}$$

In Table 1 in Appendix B we summarize the numbers of halting, looping, overhead, illegal branch and extendable programs up to data string length 20 for the base 16 model. Using the halting counts, given in the table, we obtain the following approximations:

$\Omega^{0}_{U16} = 0.$
$\Omega^{4}_{U16} = 0.0001$
$\Omega^{8}_{U16} = 0.00010000$
$\Omega^{12}_{U16} = 0.000100000000$
$\Omega^{16}_{U16} = 0.0001000000010000$
$\Omega^{20}_{U16} = 0.00010000000100001000$
$\Omega^{24}_{U16} = 0.000100000001000010000100$
$\Omega^{28}_{U16} = 0.00010000000100001010010010 10$
$\Omega^{32}_{U16} = 0.00010000000100001010011011110100$
$\Omega^{36}_{U16} = 0.00010000000100001010011100010010 0100$
$\Omega^{40}_{U16} = 0.00010000000100001010011101100101001110 10$
$\Omega^{44}_{U16} = 0.00010000000100001010011101101111000110111001$
$\Omega^{48}_{U16} = 0.00010000000100001010011101101111110010101101 0101$
$\Omega^{52}_{U16} = 0.00010000000100001010011101110000110100001111001100 10$
$\Omega^{56}_{U16} = 0.00010000000100001010011101110000111111001100111010001101$
$\Omega^{60}_{U16} = 0.00010000000100001010011101110001000000001111110110101100 0011$
$\Omega^{64}_{U16} = 0.00010000000100001010011101110001000001001100000100001001110 00111$
$\Omega^{68}_{U16} = 0.00010000000100001010011101110001000001011001010001011101000010111001$
$\Omega^{72}_{U16} = 0.00010000000100001010011101110001000001011010110111001000000101110010 1010$
$\Omega^{76}_{U16} = 0.00010000000100001010011101110001000001011011111001100010001110101011 0111111$
1
$\Omega^{80}_{U16} = 0.00010000000100001010011101110001000001011100001010100110010100110010 1011001$
1100

To be able to compute the exact values of the first N bits of Ω_{U16} we need to be able to *prove* upper bounds on the obtained approximations. Our strategy consists in showing that longer programs do not affect the first N bits of Ω_{U16}. Due to our specific procedure for solving the halting problem, *any halting program of length N has a prefix of length $N-4$.* This gives an upper bound for the number of possible halting programs of length N.

Note that each halting program with non-empty data has to have at least one READ instruction.

With Ω^{k}_{U16} from (1) we have:

$$\Omega_{U16} = \sum_{U16(w)\,\text{halts}} 16^{-|w|} = \Omega^{k}_{U16} + \sum_{\{\|x\|>k,\, U16(w)\,\text{halts}\}} 16^{-|w|},$$

so to obtain an exact approximation of order k for Omega we need to find an upper bound for the series:

$$\sum_{\{\|x\|>k,\, U16(w)\,\text{halts}\}} 16^{-|w|}.$$

The upper bound will be obtained by observing that each halting program of length greater than k has to extend an extendable prefix of length k. There are two types of extendable prefixes: those which contain `HALT` and those which do not contain `HALT`. The upper bound will be the sum between the number of halting programs extending these two types of extendable prefixes:

- if x contains `HALT` and we have M_k prefixes x of length k, then, in view of Theorem 6 (for $b = 16$), the worst case scenario leading to a halting program is to add just one character and halt, i.e. the contribution is $16 \cdot M_k \cdot 2^{-4k-4} = M_k \cdot 16^{-k}$;
- if y does not contain `HALT`, and we have N_k prefixes y of length k, then the worst case scenario leading to a halting program is to assume that all extensions halt with `HALT`, hence $y \cdot \%$ contributes $N_k \cdot 16^{-k-1}$.

Consequently, our approximation of the tail series is:

$$\sum_{\{\|x\|>k,\, U16(x)\,\text{halts}\}} 16^{-|x|} \leq M_k \cdot 16^{-k} + N_k \cdot 16^{-k-1} = (M_k + 16^{-1} \cdot N_k) \cdot 16^{-k}.$$

The "tail" contribution of all programs of length greater or equal to 84 ($M_{21} = 75582618484$ and $N_{21} = 1386091346$) is bounded by

$$\begin{aligned} 2^{-42} &\leq \sum_{\{\|x\|>20,\, U16(x)\,\text{halts}\}} 16^{-|x|} \qquad (2) \\ &\leq (75582618484 + 16^{-1} \cdot 1386091346) \cdot 16^{-20} < 2^{-43}, \end{aligned}$$

consequently, using Table 1 in Appendix B, the lower bound is:

0.00010000000100001010011101110001000001011100001010100110010100110010101100111100

and, in view of (2), the upper bound is:

0.00010000000100001010011101110001000001011101010001000100100100000100100001000101001

In conclusion, the first 43 exact bits of Ω_{U16} are:

00010 00000 01000 01010 01110 11100 01000 00101 110

Comments. (a) Assume that the "screening process" for halting programs described in Section 4 has an unreasonable high number of mistakes at levels 18–20, say 5% of declared non-halting programs actually halt. With this "new data" the counts of halting programs in Table 1, in Appendix B, will be changed at the last levels as follows:

Programs + data string length	**Number of halting programs**
18	426582153
19	4506223687
20	18329694665

Using the above approximation method we get a "new" lower bound for Omega whose *first 43 are exactly the bits of* Ω_{U16}.

(b) Comparing with the model in [5], for the present more compact machine we computed more bits of its Omega number: $43 * 7/4 = 75.25 > 75 = 64 * 7/4$.

6 The first bits of Omega in base 2

For the base 2 model our task to compute the initial bits of Ω_{U2} is a little harder. We have to consider all program lengths (including data) and not just those that are a multiple of 4. In Table 2, in Appendix B, we list the counts of the halting programs that we have discovered for various lengths. The bold entries (i.e. to length at most 61) in this table are *exact* while the other entries are *just lower bounds.*

Let Ω^k_{U2} be the approximation of Ω_{U2} given by the summation of all halting programs of up to k bits in length, that is,

$$\Omega^k_{U2} = \sum_{\{\|w\| \leq k,\, U2(w)\,\text{halts}\}} 2^{-\|w\|}. \tag{3}$$

Recall that in this case if $U2(w)$ halts, then w is of the form $w = x \cdot d$, where x is a string of instructions (the last is `HALT`, denoted by `%`) and d is a sequence of raw bits. The program length of w is now $\|w\| = 4|x| + |d|$, where $|x|$ is the number of characters in x and $|d|$ is the length of the binary string d.

Again using these counts, we can compute (for the base 2 model) an Omega lower bound of Ω_{U2} at each stage. The successive approximations are:

$\Omega^0_{U2} = 0.$
$\Omega^1_{U2} = 0.0$
$\Omega^2_{U2} = 0.00$
$\Omega^3_{U2} = 0.000$
$\Omega^4_{U2} = 0.0001$
$\Omega^5_{U2} = 0.00010$
$\Omega^6_{U2} = 0.000100$
$\Omega^7_{U2} = 0.0001000$
$\Omega^8_{U2} = 0.00010000$
$\Omega^9_{U2} = 0.000100000$
$\Omega^{10}_{U2} = 0.0001000000$
$\Omega^{11}_{U2} = 0.00010000000$
$\Omega^{12}_{U2} = 0.000100000000$
$\Omega^{13}_{U2} = 0.0001000000010$
$\Omega^{14}_{U2} = 0.00010000000100$
$\Omega^{15}_{U2} = 0.000100000001000$
$\Omega^{16}_{U2} = 0.0001000000010000$
$\Omega^{17}_{U2} = 0.00010000000100000$
$\Omega^{18}_{U2} = 0.000100000001000000$
$\Omega^{19}_{U2} = 0.0001000000010000000$
$\Omega^{20}_{U2} = 0.00010000000100001000$

$\Omega_{U2}^{21} = 0.000100000001000010000$
$\Omega_{U2}^{22} = 0.000100000010000101000$
$\Omega_{U2}^{23} = 0.000100000100001010000$
$\Omega_{U2}^{24} = 0.000100000001000010100100$
$\Omega_{U2}^{25} = 0.000100000001000010100100 0$
$\Omega_{U2}^{26} = 0.000100000001000010100100 00$
$\Omega_{U2}^{27} = 0.000100000001000010100100 000$
$\Omega_{U2}^{28} = 0.000100000001000010100100 1010$
$\Omega_{U2}^{29} = 0.000100000001000010100110 11100$
$\Omega_{U2}^{30} = 0.000100000001000010100110 111000$
$\Omega_{U2}^{31} = 0.000100000001000010100111 0011000$
$\Omega_{U2}^{32} = 0.000100000001000010100111 01000100$
$\Omega_{U2}^{33} = 0.000100000001000010100111 01010100 0$
$\Omega_{U2}^{34} = 0.000100000001000010100111 01010100 00$
$\Omega_{U2}^{35} = 0.000100000001000010100111 01010100 000$
$\Omega_{U2}^{36} = 0.000100000001000010100111 01100001 0100$
$\Omega_{U2}^{37} = 0.000100000001000010100111 01100100 0010$
$\Omega_{U2}^{38} = 0.000100000001000010100111 01101101 100101$
$\Omega_{U2}^{39} = 0.000100000001000010100111 01101101 1010011$
$\Omega_{U2}^{40} = 0.000100000001000010100111 01101111 101111111$
$\Omega_{U2}^{41} = 0.000100000001000010100111 01101111 110101111$
$\Omega_{U2}^{42} = 0.000100000001000010100111 01111000 0001010111$
$\Omega_{U2}^{43} = 0.000100000001000010100111 01111000 0000010110111$
$\Omega_{U2}^{44} = 0.000100000001000010100111 01111000 0001111001000$
$\Omega_{U2}^{45} = 0.000100000001000010100111 01111000 0101010010 01$
$\Omega_{U2}^{46} = 0.000100000001000010100111 01111000 0101101100 01111$
$\Omega_{U2}^{47} = 0.000100000001000010100111 01111000 0110111111 101011$
$\Omega_{U2}^{48} = 0.000100000001000010100111 01111000 0111001011 1000111$
$\Omega_{U2}^{49} = 0.000100000001000010100111 01111000 0111011110 01101011$
$\Omega_{U2}^{50} = 0.000100000001000010100111 01111000 0111100001 10101101 1$
$\Omega_{U2}^{51} = 0.000100000001000010100111 01111000 0111100011 10101011$
$\Omega_{U2}^{52} = 0.000100000001000010100111 01111000 0111101000 00110101 10$
$\Omega_{U2}^{53} = 0.000100000001000010100111 01111000 0111101010 10001111 00001$
$\Omega_{U2}^{54} = 0.000100000001000010100111 01111000 0111110000 10100111 00111$
$\Omega_{U2}^{55} = 0.000100000001000010100111 01111000 0111110001 00100011 011001$
$\Omega_{U2}^{56} = 0.000100000001000010100111 01111000 0111110011 00100000 1101101$
$\Omega_{U2}^{57} = 0.000100000001000010100111 01111000 0111110011 01111010 10111101$
$\Omega_{U2}^{58} = 0.000100000001000010100111 01111000 0111110011 11110110 01010000 11$
$\Omega_{U2}^{59} = 0.000100000001000010100111 01111000 0111110011 11111110 01010111 1111$
$\Omega_{U2}^{60} = 0.000100000001000010100111 01111000 0111110100 00011100 00001001 011$
$\Omega_{U2}^{61} = 0.000100000001000010100111 01111000 0111110100 01010111 00101000 1001$

The following lower bound approximations $\Omega_{U2}^{<j}$ incorporate the counts of those register machine programs of length j which *we know* that halt. Due to the large number of prefixes we did not extend some of the prefixes when the string length was more than 30 or when the program+data length was more than 80.

$\Omega_{U2}^{<62} = 0.000100000001000010100111 01111000 0111110100 01101001 11011101 10101$

$\Omega_{U2}^{<63} = 0.000100000001000010100111011100001111101001001001001011101100001$
$\Omega_{U2}^{<64} = 0.0001000000010000101001110111000011111010010011010011001000111010$
$\Omega_{U2}^{<65} = 0.0001000000010000101001110111000011111010010100111000110011101110 0$
$\Omega_{U2}^{<66} = 0.0001000000010000101001110111000011111010010101010011111000010100001$
$\Omega_{U2}^{<67} = 0.0001000000010000101001110111000011111010010101101100110000100100001$
$\Omega_{U2}^{<68} = 0.0001000000010000101001110111000011111010010101111001000011001110 1000$
$\Omega_{U2}^{<69} = 0.0001000000010000101001110111000011111010010110000010011111001000010011$
$\Omega_{U2}^{<70} = 0.0001000000010000101001110111000011111010010110010111010100001011110110$
$\Omega_{U2}^{<71} = 0.0001000000010000101001110111000011111010010110011011100100010111101 1101$
$\Omega_{U2}^{<72} = 0.0001000000010000101001110111000011111010010111010010110001011110101100001$
$\Omega_{U2}^{<73} = 0.0001000000010000101001110111000011111010010111010100001110011100001111 1010$
$\Omega_{U2}^{<74} = 0.0001000000010000101001110111000011111010010111010101101101010011000010 10100$
$\Omega_{U2}^{<75} = 0.0001000000010000101001110111000011111010010111010101111111100110011101 110110$
$\Omega_{U2}^{<76} = 0.0001000000010000101001110111000011111010010111010110011100010110101010000011 1$
$\Omega_{U2}^{<77} = 0.0001000000010000101001110111000011111010010111010110100010111011110101000101 11$
$\Omega_{U2}^{<78} = 0.0001000000010000101001110111000011111010010111010110100111001100001101110000 000$
$\Omega_{U2}^{<79} = 0.0001000000010000101001110111000011111010010111010110101010100100011001111111 1010$
$\Omega_{U2}^{<80} = 0.0001000000010000101001110111000011111010010111010110101100101111101011011010 10010$
$\Omega_{U2}^{<81} = 0.0001000000010000101001110111000011111010010111010110101101011100101110000100 001101$
$\Omega_{U2}^{<82} = 0.0001000000010000101001110111000011111010010111010110101101011101101110111101 1100111$
$\Omega_{U2}^{<83} = 0.0001000000010000101001110111000011111010010111010110101101011110100100101010 00011010$
$\Omega_{U2}^{<84} = 0.0001000000010000101001110111000011111010010111010110101101100001011011000011 110010000$

We now explain how we calculated an upper bound for Ω_{U2}. We will slightly change the strategy used for the base 16 model because in the base 2 model there are too many programs to generate. To make the process feasible we will apply several times Theorem 6 for various sets of canonical prefixes (grouped by program+data length sizes) which will be aggregated into a final upper bound.

As in the base 16 model there are two types of extendable prefixes: those which contain `HALT` and those which do not contain `HALT`. The upper bound will be obtained by summing the potential halting programs that can occur by extending these two types of extendable prefixes. Using a similar analysis as in base 16 model, if we have M_k canonical prefixes of length k which contain `HALT` and N_k prefixes of length k which do not contain `HALT`, then an upper bound for the tail series:

$$\sum_{\{\|x\|>k,\, U2(w)\,\text{halts}\}} 2^{-\|w\|} = \Omega_{U2} - \Omega_{U2}^{k}$$

is

$$2 \cdot M_k \cdot 2^{-k-1} + 1 \cdot N_k \cdot 2^{-k-4} = M_k \cdot 2^{-k} + N_k \cdot 2^{-k-4}. \quad (4)$$

The computation is feasible for extendable prefixes that do not contain `HALT` till $k = 80$: we have obtained $N_{80} = 550974535$, so these prefixes contribute with $N_{80} \cdot 2^{-84} = 550974535 \cdot 2^{-84}$.

Next we consider the overread prefixes, i.e. prefixes which contain `HALT`. Ideally, we would have computed M_{80}, so the upper bound would have been $M_{80} \cdot 2^{-80} + N_{80} \cdot 2^{-84}$. Unfortunately, in the base 2 case we did not reach M_{80}, hence we have a set of non-extended prefixes of different program lengths. Still, by iterating Theorem 6, we still get an upper bound considering where the remaining prefixes have lengths 61, 67, 73, 76, 79, 81 or 84. More precisely, instead of $M_{80} \cdot 2^{-80}$ we use the bound $M'_{61} \cdot 2^{-61} + M'_{67} \cdot 2^{-67} + M'_{73} \cdot 2^{-73} + M'_{76} \cdot 2^{-76} + M'_{79} \cdot 2^{-79} + M'_{81} \cdot 2^{-81} + M'_{84} \cdot 2^{-84}$. Here M'_j, represents the count of the non-extended prefixes of length j. These counts are listed below:

Program+data length (j)	M'_j
61	10
67	8745008
73	891868952
76	37749002
79	2587456080
81	921721461
84	149028029

A concrete example is M'_{61} representing just these 10 overread prefixes (program+data lengths are $4 * 9 + 25 = 61$ bits):

```
!a=a,0,a%0000000000000000000000000      !a=a,b,c%0000000000000000000000000
!a=a,0,b%0000000000000000000000000      !a=b,a,a%0000000000000000000000000
!a=a,1,b%1111111111111111111111111      !a=b,a,b%0000000000000000000000000
!a=a,b,a%0000000000000000000000000      !a=b,a,c%0000000000000000000000000
!a=a,b,b%0000000000000000000000000      !a=b,c,a%0000000000000000000000000
```

It is interesting to note that 9 of these 10 prefixes can extend to either a halt or an overread (repeatedly). These enumeration trees are isomorphic to the caterpillar trees discussed in the proof of Theorem 6. So we saved a bit of computation by not extending them (which we could have done much earlier). The other/last case always extends to a loop or an overread[4].

An upper bound for Ω_{U2} is now obtained by adding a "tail" of potential halting programs, by iterating Theorem 6, to our best lower bound:

$$\Omega_{U2} \leq \Omega_{U2}^{\leq 84} + N_{80} \cdot 2^{-84} + \sum_{i=1}^{\infty} M'_i \cdot 2^{-i}.$$

The value of M'_i is 0 for all but a finite number of cases; moreover, the set of lengths $\{61, 67, 73, 76, 79, 81, 84\}$ accounts for all possible extensions (of data) of these prefixes that halt.

[4]Thus, we could set $M'_{61} = 9$, since this particular prefix never extends to a halting program.

Indeed, we just look at the set of overread programs that we did not extend. The enumeration tree has a finite height, so from the definition of M'_i, the set of overread prefixes (that were not extended) fall into at most a finite number of $M'_i > 0$ cases.

If a program halts, then it must be in *exactly one* of these three cases:

1. It has been enumerated and counted (i.e. its size is at most 80),
2. It is an extension of one of the N_j prefixes without `HALT` (i.e. its size is at least 84), or
3. It has a prefix that is in (at most) one of the non-extended overread sets (i.e. its size is at least 62).

There are no other cases. Let O_i denote the overread set corresponding to M'_i. We also know there does not exist $x \in O_i$ and $y \in O_j, i \neq j$, such that x is a prefix of y: this follows from our lexicographic enumeration procedure over Σ_{16}. Consequently, tallying the contribution of each of these three cases we have:

$$\Omega_{U2} \leq \Omega_{U2}^{<84} + N_{80} \cdot 2^{-84} + M'_{61} \cdot 2^{-61} + M'_{67} \cdot 2^{-67} + M'_{73} \cdot 2^{-73} + M'_{76} \cdot 2^{-76} + M'_{79} \cdot 2^{-79} + M'_{81} \cdot 2^{-81} + M'_{84} \cdot 2^{-84}.$$

Thus evaluating this formula we get an upper bound. (Implicit in Theorem 6 we assume, for the worst, that all canonical prefixes extend to a halting program of length at most 85.)

$$\Omega_{U2}^{85}(ub) = 0.000100000001000010100111011100001111101010000111100100000000101100011010000000111000.$$

After comparing with our earlier lower bound we conclude that we have calculated the 40 exact bits (underlined below):

$$\Omega_{U2}^{84}(lb) = 0.\underline{0001000000010000101001110111000011111010}10110101101011011000010110110000011110010000.$$

that is, the first exact 40 bits of Ω_{U2} are

00010 00000 01000 01010 01110 11100 00111 11010

Finally, let us observe that stopping our computation at level 60 would give us $M_{60} = 486283, N_{60} = 2764738$. As expected, the resulting upper bound is less exact than the upper bound previously obtained, but using the formula (4) we still get 40 exact bits of Ω_{U2}.

7 Final comments

From the results of two previous sections we can conclude, as expected, that the two models (base 16 and base 2) of our compact register machine yield different Omegas. We also observe that the semantics of the READ instruction getting 4 bits at a time (instead of 1) allows for more short programs to halt. However, we are not advocating the use of one particular model over the other. The base 2 model is more flexible in allowing any length of data—not just a multiple of 4 bits, which is more in line with modern computer architectures having bytes being the smallest unit of storage for a program. The encoding of data is paramount (base 2 vs. base 16): one can see that $\Omega_{U16} \neq \Omega_{U2}$.

In connection with our time-bounded simulation, one would naturally want to know the shortest program that halts with more than 1000 branches. Why? For example, if this program is larger than 80 bits, then all of our looping programs (in both base models) never halt. One such short program is given below, with instruction labels given above the operators:

```
0           1   2     3   4     5     6
&a,111111111&b,1&c,110+d,1=a,d,c=a,a,b%
```

This program has length 156 bits (39 characters). We can reduce this by 12 bits if we are using the base 16 register machine (replace the instruction `&c,110` with `!c` and add one character/nibble of data that has value 6). The authors are interested in knowing if there are any shorter programs.

Finally, one method for improving our upper bound (and possibly an easy way to yield more exact bits known) is to consider more closely the tail of the N_k prefixes that do not contain `HALT`. Our analysis ignored the syntactical properties of valid register machine programs and just assumed every prefix string can be extended to a halting program by appending 4 bits. However, the halt instruction `%` may only appear after a complete (preceding) instruction has been well specified with its operator and operand arguments. In fact, by a simple probabilistic observation, it is easy to see that at most 1/4 of prefixes have their last instruction well-specified with all required operands. The authors suspect that a more detailed analysis of the syntax of valid programs might provide us one more known bit by lowering the contribution to Omega of this tail by a factor of 1/2.

Acknowledgement

We thank Ivy Jiang and Simona Mancaş who helped us write (and debug) preliminary versions of the Java simulator for the register machine language(s) described here. We are grateful to Greg Chaitin and John Tromp for valuable criticism and comments which improved this paper.

References

[1] C. S. Calude. *Information and Randomness: An Algorithmic Perspective*, 2nd Edition, Revised and Extended, Springer Verlag, Berlin, 2002.

[2] C. S. Calude. A characterization of c.e. random reals, *Theoret. Comput. Sci.* 271 (2002), 3–14.

[3] C. S. Calude. Chaitin Ω numbers, Solovay machines and incompleteness, *Theoret. Comput. Sci.* 284 (2002), 269–277.

[4] C. S. Calude, Elena Calude, M. J. Dinneen. A new measure of the difficulty of problems, *Journal for Multiple-Valued Logic and Soft Computing* 10 (2006), 1–21.

[5] C. S. Calude, M. J. Dinneen and C.-K. Shu. Computing a glimpse of randomness, *Experimental Mathematics* 11, 2 (2002), 369–378.

[6] C. Calude, H. Jürgensen. Randomness as an invariant for number representations, in H. Maurer, J. Karhumäki, G. Rozenberg (eds.). *Results and Trends in Theoretical Computer Science*, Springer-Verlag, Berlin, 1994, 44–66.

[7] C. S. Calude, P. Hertling, B. Khoussainov, Y. Wang. Recursively enumerable reals and Chaitin Ω numbers, *Theoret. Comput. Sci.* 255 (2001), 125–149.

[8] C. S. Calude, M. A. Stay. Most Programs Stop Quickly or Never Halt, *CDMTCS Research Report* 284, 2006.

[9] G. J. Chaitin. A theory of program size formally identical to information theory, *J. Assoc. Comput. Mach.* 22 (1975), 329–340.

[10] G. J. Chaitin. *Algorithmic Information Theory*, Cambridge University Press, Cambridge, 1987. (third printing 1990)

[11] G. J. Chaitin. *The Limits of Mathematics*, Springer-Verlag, Singapore, 1997.

[12] P. Hertling, K. Weihrauch. Randomness spaces, in K. G. Larsen, S. Skyum, and G. Winskel (eds.). *Automata, Languages and Programming, Proceedings of the 25th International Colloquium, ICALP'98* (Aalborg, Denmark), Springer-Verlag, Berlin, 1998, 796–807.

[13] A. Kučera, T. A. Slaman. Randomness and recursive enumerability, *SIAM J. Comput.* 31, 1 (2001), 199–211.

[14] P. Odifreddi. *Classical Recursion Theory*, North-Holland, Amsterdam, Vol.1, 1989, Vol. 2, 1999.

[15] R. I. Soare. *Recursively Enumerable Sets and Degrees*, Springer-Verlag, Berlin, 1987.

[16] R. M. Solovay. A version of Ω for which ZFC can not predict a single bit, in C.S. Calude, G. Păun (eds.). *Finite Versus Infinite. Contributions to an Eternal Dilemma*, Springer-Verlag, London, 2000, 323-334.

[17] L. Staiger. The Kolmogorov complexity of real numbers, in G. Ciobanu and Gh. Păun (eds.). *Proc. Fundamentals of Computation Theory*, Lecture Notes in Comput. Sci. No. 1684, Springer-Verlag, Berlin, 1999, 536–546.

[18] J. Tromp. Email to C. Calude, 21 December 2006.

Appendix

A Some details about the computation process

The source-code for our register machine simulator (written in Java) that we used to compute the first bits of the two Omegas is available for download at `ftp://ftp.cs.auckland.ac.nz/pub/research/CDMTCS/Omega`.

Our computational process was pretty much automated with the use of shell scripts. For example, we used the following bash script to classify register machines at each new level.

```
#!/bin/bash

i=$*
echo extending len=$i

java -cp Java_Progs -Xmx2000M EnumOverread overread.$i
rm runnable
cp overread overread.$((i+1))
mkdir -p $((i+1))/overread/*
mv halt loop extendable underread overread illegalbranch $((i+1))/overread
gzip -f -q $((i+1))/overread/*

java -cp Java_Progs -Xmx2000M Enumerate extendable.$i
rm runnable
cp extendable extendable.$((i+1))
mkdir -p $((i+1))/extendable/*
mv halt loop extendable underread overread illegalbranch $((i+1))/extendable
gzip -f -q $((i+1))/extendable/*
```

In the above script, we extend the two types of canonical prefixes, ones with halt (overread) and ones without (extendable), and store the compressed results of the simulator in directories that are named to correspond to the current level i. Both the programs `Enumerate` and `EnumOverread` call the register machine simulator on each of their generated strings. In the case of `EnumOverread` we do not need to (re)check the syntax of the generated strings since we are just adding more data.

To extract register machine programs of a given length for the base 2 model we used the following simple perl script. (For the base 16 model we just multiply the string length by 4.)

```
#!/usr/bin/perl

$goal = $ARGV[0];

while (<>) # this file is named in $ARGV[1]
{
```

```
  if (/(.*%)(.*)/)  # program has data
  {
    $prog = $1;
    $data = $2;
    $pdlen = 4*length($prog)+length($data);
    print $prog.$data,"\n" if ($pdlen == $goal);
  }
  else              # program has no data
  {
    chomp $_;
    $prog = $_;
    $pdlen = 4*length($prog);
    print $prog,"\n" if ($pdlen == $goal);
  }
}
```

Finally, we used the following program to produce an Omega lower bound from a list of halting counts.

```
import java.io.*;
import java.util.*;
import java.math.BigInteger;

class approx
{
 public static void main(String[] argv) throws IOException
 {
   BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
   PrintWriter out = new PrintWriter(System.out);

   int maxlen = Integer.parseInt( in.readLine() );

   BigInteger total = new BigInteger("1");

   for (int i=1; i<=maxlen; i++)
   {
    String line = in.readLine();
    StringTokenizer tokens = new StringTokenizer(line);

    int len = Integer.parseInt(tokens.nextToken());
    System.out.print("Omega_"+len+" = 0.");

    BigInteger num = new BigInteger(tokens.nextToken());
    total = total.shiftLeft(1);
    total = total.add(num);
    System.out.println(total.toString(2).substring(1));
   }
 }
}
```

B Data

Table 1: Distribution of programs into simulator categories (base 16 model)

Program + data length	**Number of halting**	**Number of looping**	**Number of overread**	**Number of illegal branch**	**Number of extendable**
0	0	0	0	0	0
1	1	0	0	0	0
2	0	0	0	0	4
3	0	0	1	0	7
4	16	0	0	0	20
5	8	0	2	0	56
6	4	0	32	0	158
7	522	0	41	0	358
8	596	0	96	0	1162
9	468	0	1507	0	3043
10	21498	5	2925	52	10311
11	40473	2052	7574	53	25746
12	44869	18	79507	208	92901
13	1073634	7932	219114	3070	245506
14	2874221	2601	656188	7174	929279
15	4387315	1242508	5108174	64750	2467020
16	63135127	92453	18590195	270447	9829406
17	221590601	6219247	71366861	1563562	27034182
18	426445722	2728623	711343532	5358476	113105879
19	4456593631	992601131	5887007185	83028791	315733989
20	18322322348	147446347	75582618484	187451863	1386091346

Table 2: Number of halting register machine programs of a given length (base 2 model)

Program + data length	**Number of halting**	**Program + data length**	**Number of halting**	**Program + data length**	**Number of halting**
1	**0**	29	**72**	57	**23011**
2	**0**	30	**0**	58	**58313**
3	**0**	31	**40**	59	**16441**
4	**1**	32	**20**	60	**64205**
5	**0**	33	**32**	61	**242163**
6	**0**	34	**0**	62	153251
7	**0**	35	**0**	63	663543
8	**0**	36	**212**	64	263032
9	**0**	37	**90**	65	832872
10	**0**	38	**609**	66	441577
11	**0**	39	**9**	67	819167
12	**0**	40	**473**	68	805542
13	**2**	41	**177**	69	1235011
14	**0**	42	**249**	70	5463760
15	**0**	43	**9**	71	2229745
16	**0**	44	**602**	72	10462631
17	**0**	45	**3513**	73	6092344
18	**0**	46	**765**	74	12432992
19	**0**	47	**5325**	75	4798158
20	**8**	48	**1521**	76	15074843
21	**0**	49	**4829**	77	6900489
22	**8**	50	**1669**	78	9773970
23	**0**	51	**2037**	79	12471418
24	**4**	52	**9304**	80	18254686
25	**0**	53	**9605**	81	11807337
26	**0**	54	**49957**	82	531661
27	**0**	55	**7947**	83	879692
28	**10**	56	**65211**	84	5976924