RC 20092 (6/7/95)
Computer Science/Mathematics

IBM Research Report

A New Version of Algorithmic Information Theory

G. J. Chaitin

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

A NEW VERSION OF ALGORITHMIC INFORMATION THEORY[1]

G. J. Chaitin, IBM Research Division, P. O. Box 704, Yorktown Heights, NY 10598, USA, chaitin @ watson.ibm.com

1. Introduction

Algorithmic information theory may be viewed as the result of adding the idea of program-size complexity to recursive function theory. The main application of algorithmic information theory is its information-theoretic incompleteness theorems. This theory is concerned with the size of programs, but up to now these have never been programs that one could actually program out and run on interesting examples.

I have now figured out how to actually program the algorithms in the proofs of the all the key information-theoretic incompleteness theorems in algorithmic information theory. I have published this material electronically in a series of detailed reports [1,2,3,4]

[1] *This material was presented in a series of lectures at the Santa Fe Institute, the Los Alamos National Laboratory, and the University of New Mexico, during a one-month visit to the Santa Fe Institute, April 1995.*

that include a lot of software. These reports are available on the Web at `http://xyz.lanl.gov` or by sending e-mail to `chao-dyn @ xyz.lanl.gov` with `Subject: get yymmnnn`, where `yy` is the year, `mm` is the month, and `nnn` is the number of each report.

Here I shall give an overview of this material. I'll present the main ideas, which are a mixture of abstract mathematics and practical computer software technology.

In order to program out the algorithms in the proofs of the main information-theoretic incompleteness theorems, I use a stripped-down version of pure LISP designed especially for this purpose. The interpreter for this LISP was originally written in *Mathematica* [1]. A faster version of this interpreter was then written in *C* [2]. [3] uses a slightly different LISP with an interpreter that is also written in *C*. In addition, [1] gives the *Mathematica* code for producing a monster diophantine equation that exhibits randomness in arithmetic. This equation is basically just an interpreter for the LISP used in [1] and [2], dressed up as a diophantine equation. I haven't taken the trouble yet to produce a diophantine equation for the LISP used in [3].

The main idea is this:

In algorithmic information theory, given a self-delimiting universal Turing machine, one defines a program-size complexity measure from it. If one picks a different universal Turing machine, one gets a slightly different complexity measure.

Now I pick a specific machine to base my whole theory on.

My universal Turing machine U produces output in the form of LISP S-expressions, and U's input is a program in the form of a binary string. The machine U starts by reading a LISP S-expression π from the start of its program tape. U reads this S-expression π from the tape a character at a time in 7-bit ASCII chunks, until parentheses balance and U knows that it has read the complete S-expression π. Then U starts to evaluate the prefix π that it has read, with the rest of the program tape β available as binary data to the S-expression π in a highly controlled manner. The S-expression π can use a READ-NEXT-BIT pseudo-function to read individual bits from U's program tape. π cannot look at β, the rest of U's program tape, directly. The READ-NEXT-BIT function always returns the value 0 or 1, but it cannot return an end-of-data indication. If the S-expression π attempts to read beyond the end of the binary data β, the computation aborts. This forces U's program $\pi\beta$ to be self-delimiting. (There is no punishment for failing to read all of β.)

The main difference between the pure LISP that I use and ordinary pure LISP is that there is a mechanism for giving binary data to LISP S-expressions. There is also a mechanism that makes it possible for an S-expression whose evaluation never ends to output an infinite set of S-expressions (which is all that a formal axiomatic system is). The heart of an interpreter for ordinary LISP is the recursive expression evaluator EVAL. The heart of my interpreter is not EVAL, it is a mechanism for doing a time-limited evaluation called TRY.

2. LISP

The above hints are intended for people who already know LISP. Now let me present my LISP from the ground up, using lots of examples. Then I'll use it to get some incompleteness theorems.

In set theory one constructs the entire universe from the empty set using the set forming operation. In LISP, one constructs the entire universe by making lists starting with atoms. Lists are ordered, sets are not.

In this LISP the atoms are the 128 7-bit ASCII characters. The empty list, which is also an atom, is `()`; `(abcd)` is a list of four atoms; and `((aa)(bb)(cc))` is a nested list with three elements, each of which is in turn a list with a repeated atom. The union of the set of atoms and the set of lists yields the set of S-expressions. S-expressions are the universal LISP substance: both programs and data are S-expressions.

True and false are `1` and `0`.

My LISP is so simple that numbers are not provided. But it is easy to program unary or base-two arithmetic using lists of 0's and 1's.

Normally, the result of applying the function f to the arguments x, y and z is written $f(x, y, z)$. In LISP this is written `(fxyz)`.

Pure LISP is not an imperative language with side-effects. Rather one defines functions and evaluates expressions. One starts with a number of primitive functions.

In this LISP all the primitive functions have a fixed number of arguments. This makes it possible for there to be a parenthesis-free meta-notation called M-expressions in addition to the "official" S-expression notation.

Now I present the primitive functions using examples.

The primitive function QUOTE yields its unevaluated argument, which is how one distinguishes data from programs. For example, the M-expression `'(abc)` denotes the S-expression `('(abc))` whose value is the S-expression `(abc)`. Without the QUOTE, this would indicate that the function `a` has to be applied to the arguments `b` and `c`.

The function HEAD gives the first element of a list. The M-expression `+'(abc)` yields `a`.

The function TAIL gives the rest of a list. The M-expression `-'(abc)` yields `(bc)`.

JOIN is the inverse of HEAD and TAIL: `*'a'(bc)` yields `(abc)`.

The predicate ATOM tells whether its argument is an atom or not: `.'a` yields `1` and `.'(abc)` yields `0`.

The predicate EQUAL compares two S-expressions: `='a'b` yields `0` and `='(ab)'(ab)` yields `1`.

The pseudo-function IF-THEN-ELSE has three arguments, and evaluates its second or third argument depending on its first argument. `/1'x'y` yields `x` and `/0'x'y` yields `y`.

A function definition is a triple consisting of an ampersand followed by the list of parameters followed by the function body. For example, the M-expression `('&(xy)y 'a 'b)` denotes the S-expression `(('(&(xy)y)) ('a) ('b))` whose value is `b`. In other words, this is an unnamed function of two parameters whose value is the second parameter.

Here are two additional pieces of notation, which are extremely convenient.

The M-expression `:xv e` is an abbreviation for the M-expression `('&(x)e v)`. In other words, let `x` have value `v` in expression `e`.

The M-expression `:(fxyz)b e` is an abbreviation for the M-expression `('&(f)e '&(xyz)b)`. In other words, let the function `(fxyz)` be defined to be `b` in expression `e`.

We can now present a complete LISP program. Let's define list concatenation recursively and use it to concatenate two lists.

`:(Cxy)/.xy*+x(C-xy) (C'(abc)'(def))` yields `(abcdef)`. Let's say this in words. Here is how we define the concatenation of two lists. If the first list is empty, then the concatenation is the second list. Otherwise take the first element of the first list and join it to the result of concatenating the rest of the first list to the second list.

Actually, for efficiency's sake concatenation is provided as a primitive function `^`. Also, in another concession to practicality, there are mechanisms for making function definitions permanent, so that it is not really necessary to give all function definitions locally.

There is a mechanism for displaying intermediate results. This is the pseudo-function DISPLAY which considered as a pure function is just the identity function, but which has the side-effect of displaying its argument. DISPLAY is written as a comma. Let's change the definition of concatenation to see how this works. `:(Cxy)/.,xy*+x(C-xy) (C'(abc)'(def))` finally yields `(abcdef)` just as before, but first it displays the intermediate results `(abc)`, `(bc)`, `(c)`, and `()`.

The primitive function EVAL allows one to construct an S-expression and then evaluate it. EVAL is written as an exclamation mark. For example, `!'+'(abc)` yields `a`.

The major difference between this LISP and traditional pure LISP is the primitive function TRY. TRY, which is written as a question mark, has three arguments, α, β and γ. The first, α, is a time bound, given in unary notation, that is, as a list of 1's. The second argument β is the expression to be evaluated. And the third argument γ is a list of 0's and 1's which are made available in a highly controlled manner to the expression β that is being evaluated. To access the binary data γ, the S-expression β must use two primitive functions READ-NEXT-BIT (`@`) and READ-NEXT-S-EXPRESSION (`%`), both of which are pseudo-functions with no argument.

The value of a TRY is a list whose first element is an error indication if the TRY failed, or the value of the expression β wrapped in parentheses if the TRY succeeded. The first element is `?` if the TRY failed because the time α ran out. The first element is `!` if the TRY failed because the binary data γ ran out. And the rest of the value of a TRY consists of captured intermediate results that were produced by the expression β using DISPLAY.

My LISP has permissive semantics, so that a TRY can only fail by running out of time or binary data, not by giving bad arguments to a primitive function.

Note that one is punished if one runs out of binary data, but not if one fails to read all the binary data.

EVAL and TRY always start with a fresh environment in which each atom is bound to itself. EVAL has no time limit, TRY does. EVAL inherits the binary data, if any, in its context, while TRY switches to new binary data. EVAL is dangerous; it may run forever or may abort its user. TRY is safe; it can only run for a fixed amount of time and it cannot abort its user. Using EVAL can eat up one's binary data; using TRY doesn't touch one's binary data. DISPLAY's from within EVAL take place normally; DISPLAY's within TRY are suppressed.

What if one TRY's the concatenation example given above, with the first arguments `()` `(1)` `(11)` `(111)` `(1111)` and `(11111)` and the third argument `()`? In other words, let's TRY running the concatenation example with longer and longer time bounds and no binary data. Here are the values of TRY that one gets: `(?)` `(?)` `(?(abc))` `(?(bc)(abc))` `(?(c)(bc)(abc))` and `(((abcdef))()(c)(bc)(abc))`.

The extra pair of parentheses around the result of the concatenation makes it possible to distinguish the errors `?` and `!` from the valid values `(?)` and `(!)`.

If the first argument of TRY is not a list, then there is no time limit. Then one immediately gets `(((abcdef))()(c)(bc)(abc))` as the value of the TRY.

We actually use a machine-independent time limit α that is a limit on the maximum interpreter stack depth. The body of a function definition and expressions that are EVAL'ed or TRY'ed are evaluated using a time limit $\alpha' = \alpha - 1$ that is one unit less than the time limit α for the containing expression.

The most delicate thing in the interpreter is getting nested TRY's to work properly. First of all, within nested TRY's the most constraining time limit must apply. Secondly, if time runs out one must unwind the interpreter stack back up to the correct TRY, which is the one that imposed the strongest constraint.

A final primitive function CONVERT-TO-BITS (`#`) converts its argument from an S-expression into the list of the consecutive bits in the ASCII for its character string representation. CONVERT-TO-BITS and READ-NEXT-S-EXPRESSION are in effect inverse functions.

3. Complexity

That concludes the presentation of my LISP. That's all there is to it! It's very simple! It has to be, if we are going to make it into a diophantine equation. But this simple programming language is powerful enough to code all the proofs of my information-theoretic incompleteness theorems. This LISP is powerful enough to program an interpreter for my universal Turing machine U in a single line of LISP code. This LISP can also be used as a very high-level assembler to produce complicated binary programs for U.

In fact, here is how we program U. `(Up)`, the result of running the universal machine U on the binary program p, is defined to be a TRY with no time limit of EVAL of READ-NEXT-S-EXPRESSION using p as the associated binary data. More precisely, `(Up)` is defined to be the M-expression `++?0'!%p` which is just the S-expression `(+(+(?0('(!(%)))p)))` which says, read a complete S-expression from the program tape, then TRY to evaluate it with no time limit using the rest of the program tape as binary data.

Now that we have U, let's use it to measure program-size complexity. Let's define the complexity $H(x)$ of an arbitrary S-expression x to be the size in bits of the smallest program for U to compute x. In other words, $H(x)$ is the size in bits of the smallest p such that $U(p) = x$.

What properties does this complexity measure H have?

Let's start by considering two arbitrary S-expressions x and y. Consider the M-expression `*!% *!% ()`, which is the S-expression `(*(!(%)) (*(!(%)) ()))`, which is 20 characters and $7 \times 20 = 140$ bits. This expression says to read two S-expressions from the binary data, and then evaluate them and put the results together into a list. If we concatenate this 140-bit prefix with a minimum-size program for U to calculate x and a minimum-size program for U to calculate y, we get a program for U to calculate the pair (xy) that is precisely $140 + H(x) + H(y)$ bits long. Thus we see in a very concrete manner that the joint complexity $H(x, y)$ is bounded by the sum of the individual complexities $H(x)$ and $H(y)$ plus the constant $c = 140$:

$H(x,y) \leq H(x) + H(y) + 140$.

Now consider a binary string x that is $|x|$ bits long. It's easy to see that $H(x) \leq 2|x| + 471$ and $H(x) \leq |x| + H(|x|) + 1148$.

Why?

To prove that $H(x) \leq 2|x| + 471$ one programs a 469-bit prefix π so that $U(\pi\, 00\, 11\, 00\, 11\, 01) = 0101$. In other words, the prefix π makes U read two bits at a time from its program tape as long as they are equal. U stops as soon as two unequal bits are found.

And to prove that $H(x) \leq |x| + H(|x|) + 1148$ one programs a 1148-bit prefix π so that $U(\pi\beta x) = x$ if β is a minimum-size program for U to compute the base-two notation for the size of the bit string x. Here is how this works: first the prefix π makes U run the program that follows π on the program tape to determine how many bits there are, then π makes U read that many bits from the program tape and stop.

For more details, see the program `univ.lisp` in [2].

Next we show how to compute the halting probability Ω of U in the limit from below. TRY running U for time N on each N-bit program. This gives a monotone increasing sequence W_N of lower bounds on Ω: W_N is (the number of N-bit programs that halt on U within time N) divided by 2^N.

Here is a simple way to calculate W_N. Given N, first generate the list of all N-bit strings. Then use TRY to select the subset of this list of programs that halt within time N when run on U. Then count the size of the list of programs that halt, and develop this count in base-two notation. Finally pad the base-two count on the left with enough 0's to make an N-bit string, and prepend "0.". By using a slightly more sophisticated counting technique, one can avoid having to generate these two enormous lists of programs. That's a good idea, because my fast C interpreter does not have garbage collection. In this manner I was able to compute W_{22} in about an hour on a 512-megabyte IBM RS/6000 workstation.

For more details, see the program `omega.lisp` in [1] and [2].

Now we show that the bits of the halting probability Ω are irreducibly complex. More precisely, let Ω_N be the first N bits of the base-two representation of the real number Ω. We shall show that $H(\Omega_N) > N - 4431$.

Why?

The reason is that knowing Ω_N enables one to solve the halting problem for all N-bit programs. More precisely, there is a prefix π that is 4431 bits long that when concatenated to a minimum-size program ω for U to compute Ω_N does the following. First π runs ω to compute Ω_N. Then π calculates W_T for larger and larger values of T until the first N bits of W_T are okay, that is, agree with Ω_N. At this point π knows that any N-bit program for U that halts must halt within time T. So π can give as its final result the list B_N of the values of all N-bit programs for U that halt. B_N includes every S-expression with program-size complexity $\leq N$. Hence this big list B_N must itself have program-size complexity greater than N. Thus $N < H(B_N) \leq |\pi\omega| = 4431 + H(\Omega_N)$, which was to be proved.

For more details, see the program `omega2.lisp` in [2].

Now let's get an incompleteness result. We show that a formal axiomatic system of complexity N cannot enable us to determine more than $N + 4431 + 3150$ bits of the binary representation of the halting probability Ω.

Why? It's just a version of the Berry paradox discussed in [5], which deals with "the first positive integer that cannot be named in less than a billion words."

This time we employ a prefix π that is 3150 bits long and contains within itself the binary constant for the sum of 3150 and 4431. We concatenate this prefix π to the formal axiomatic system α, which is an unending minimum-size program for U to DISPLAY S-expressions of the form `(110X0XXX10)` standing for partial determinations of the bits of Ω. At any given time the prefix π has read ρ bits of α as it TRY's to DISPLAY more and more theorems of the formal axiomatic system α. This continues until π is able to determine more than $4431 + 3150 + \rho$ bits of Ω using ρ bits of α. At that point the missing `X` bits of Ω are read by π from the program tape at a cost of exactly one bit added for each missing bit. The result is a program for U that is exactly $3150 + \rho +$ (the number of missing bits) bits long that calculates a complete initial segment of the binary expansion of Ω that is more than 4431 bits longer than it is. But this contradicts the previously established fact that $H(\Omega_N) > N - 4431$.

For more details, see the program `godel3.lisp` in [2].

There is also a more sophisticated M-expression version of our universal Turing machine U. It is used in [3]. This new and improved version of U reads an M-expression prefix from the beginning of the program tape, not an S-expression prefix. Then, as before, U runs this M-expression with the rest of the program tape as binary data. To change U to work this way only requires that two primitive functions, READ-NEXT-S-EXPRESSION and CONVERT-TO-BITS, be changed to work with M-expressions instead of S-expressions. And in [3] we also use a more aggressive kind of M-expression, in which defined functions as well as primitive functions have their parentheses omitted. Thus in [3] our concatenation example simplifies to `:(Cxy)/.xy*+xC-xy C'(abc)'(def)`. The constants in our previous results, which come from [2], shrink substantially in [3]. Now we get $H(x,y) \leq H(x) + H(y) + 56$, $H(x) \leq 2|x| + 142$, $H(x) \leq |x| + H(|x|) + 441$, $H(\Omega_N) > N - 1883$, and a formal axiomatic system of complexity N cannot enable us to determine more than $N + 2933$ bits of the binary representation of the halting probability Ω.

4. Arithmetic

The halting probability Ω shows that some mathematical questions are irreducible, that is, have the property that essentially the only way to prove them is to add them as new axioms. The bits of the Ω are irreducible mathematical facts. But can we find irreducible mathematical facts in elementary number theory? The answer is yes, and I can even explicitly exhibit arithmetical versions of the bits of Ω. I do this by using *Mathematica* to convert an interpreter for the LISP used in [1] and [2] into an enormous diophantine equation. (I haven't done this work yet with the LISP used in [3].)

Here is an outline of how to get a diophantine equation that can be used as a LISP interpreter. (For the programming details and the *Mathematica* software, see [1]. For a detailed explanation, see Chapter 2 of [6].)

The first step is to write abstract register machine code for a LISP interpreter that works one character at a time on the reversed character strings for S-expressions. It's hard work to debug such low-level code! In order to speed up the job, I use a *Mathematica* program to compile the register machine code into *C*. Then I use another *Mathematica* program to compile the register machine code into an exponential diophantine equation. ("Exponential" just means that unknowns can occur in exponents.) This equation $L(k, x_1, x_2, \ldots) = R(k, x_1, x_2, \ldots)$ is several hundred pages long and has tens of thousands of unknowns. Let the positive integer k be the ASCII representation of the reversal of the character string representation of the S-expression K. Then $L(k, x_1, x_2, \ldots) = R(k, x_1, x_2, \ldots)$ is constructed so that it has precisely one solution in positive integers $x_1, x_2, \ldots$ if evaluation of the LISP S-expression K halts, and this equation is constructed so that it has no solution in positive integers $x_1, x_2, \ldots$ if evaluation of the LISP S-expression K fails to halt.

Plug into this monster equation an S-expression K that loops forever if the kth bit of W_n is a 0 and that halts if the kth bit of W_n is a 1. Then $L(k, n, x_1, x_2, \ldots) = R(k, n, x_1, x_2, \ldots)$ has finitely many solutions in positive integers $n, x_1, x_2, \ldots$ if the kth bit of Ω is a 0, and this equation has infinitely many solutions in positive integers $n, x_1, x_2, \ldots$ if the kth bit of Ω is a 1.

Thus whether $L(k, \ldots) = R(k, \ldots)$ has finitely or infinitely many positive integer solutions is so delicately balanced that it is completely accidental whether it goes one way or the other! In other words, this is a completely accidental mathematical fact, it is a mathematical fact that is true for no reason, and therefore escapes the power of mathematical reasoning.

What are we to make of this incompleteness result? I have discussed this at length in my reductionism paper [7]. Briefly, it makes me ask, is mathematics quasi-empirical? In other words, should one perhaps add new axioms to elementary number theory based on the results of computer experiments?!

To conclude, I think that the incompleteness results of algorithmic information theory seem much more real and concrete now that the programming details have been worked out and all this software is available.

References

[1] G. J. Chaitin, "The limits of mathematics—Course outline & software," chao-dyn/9312006, *IBM Research Report RC-19324,* 127 pp., December 1993.

[2] G. J. Chaitin, "The limits of mathematics," chao-dyn/9407003, *IBM Research Report RC-19646,* July 1994, 270 pp.

[3] G. J. Chaitin, "The limits of mathematics IV," chao-dyn/9407009, *IBM Research Report RC-19671,* July 1994, 231 pp.

[4] G. J. Chaitin, "The limits of mathematics (Extended abstract)," chao-dyn/9407010, *IBM Research Report RC-19672,* July 1994, 7 pp.

[5] G. J. Chaitin, "The Berry paradox," *Complexity* 1 (1995), pp. ???–???.

[6] G. J. Chaitin, *Algorithmic Information Theory,* Cambridge University Press, 1987.

[7] G. J. Chaitin, "Randomness in arithmetic and the decline and fall of reductionism in pure mathematics," in J. Cornwell, *Nature's Imagination,* Oxford University Press, 1995, pp. 27–44.