How to Run Algorithmic Information Theory on a Computer

Studying the limits of mathematical reasoning

G. J. CHAITIN

The most important application of algorithmic information theory is to show the limits of mathematical reasoning. And in particular what I've constructed and exhibited are mathematical facts which are true for no reason. These are mathematical facts which are true by accident. And since they're true for no reason you can never actually prove logically whether they're true or not. They're sort of accidental mathematical facts. They're mathematical facts which are analogous to the outcome of a coin toss, because an independent toss of a fair coin has got to come out heads or tails but there's no reason why it should come out one or the other. And I've found mathematical facts that mirror this very precisely.

Algorithmic information theory is recursive function theory plus one new idea which is program-size complexity. So the equation you want to have in mind is algorithmic information theory equals recursive function theory plus one more notion which is to look at the size of programs. That's the new element:

$$\text{AIT} = \text{recursive function theory} + \text{program size}.$$

It's roughly at the level of the kind of thing that Turing and Gödel were doing, but we throw in a complexity measure which is program-size complexity.

EVOLUTION OF AIT

There are three main stages in the development of this theory. The first is AIT_1, from the mid 1960s. You begin with an n-bit string and you ask what is the size of the smallest program for it, and it's usually about n bits. Most n-bit strings require programs of about the same size as themselves. So an n-bit string usually has program-size complexity n. That's the first version:

$$\text{AIT}_1\text{: } n\text{-bit string, complexity } n.$$

Then there's another algorithmic information theory from ten years later, the mid 1970s. I like to call that the self-delimiting version of the theory. There the basic idea is that you should be able to concatenate subroutines. Information should be additive. This sounds like a technical detail but the whole theory is transformed when you make this change.

Now, n-bit strings don't have complexity n. Instead most n-bit strings have what complexity? Well, it's not just that you need to know what the n bits are, you also need to know how many bits you're getting. So it's usually n plus the base-two logarithm of n, roughly speaking:

$$\text{AIT}_2\text{: } n\text{-bit string, complexity } n + \log_2 n.$$

It's really n plus the program-size complexity of n, to give a more precise statement. So this is the idea that a program should be self-delimiting, should indicate its own size within itself. One of the problems with this theory is how to explain this.

The new algorithmic information theory, which I've just done in the past year, is AIT_3. It's formally identical to AIT_2; from an abstract mathematical viewpoint there's no difference. But there's a world of difference!

Gregory J. Chaitin is at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. He has worked on program-size complexity, on randomness, and on the limits of mathematical reasoning, and has published a monograph on algorithmic information theory (Cambridge University Press), and several collections of his papers (World Scientific Publishing). One of his most recent articles is included in the volume "Nature's Imagination" published by Oxford University Press in 1995. He has also published expository papers in "Scientific American," in the "New Scientist," and in "La Recherche." His Web site is http://www.research.ibm.com/people/c/chaitin/, e-mail: chaitin@watson.ibm.com

CCC 1076-2787/96/010015-07

The difference is basically two-fold. Let's go back to recursive function theory in the equation

AIT = **recursive function theory** + program size.

There are lots of books on recursive functions, like Hartley Rogers [1]. This book is about computer programs. All that Hartley Rogers cares about and all my theory cares about is whether the program *eventually* halts. It doesn't matter how long it takes. What I add is that I do care about the size in bits of the program:

AIT = recursive function theory + **program size**.

But the interesting thing about these books from the point of view of a computer programmer is that they're lousy because there isn't one program that you can actually run. Maybe that was okay then, but I like actually using computers! So I'd like to tell you how to take algorithmic information theory—which, roughly speaking, is how to take recursive function theory—and actually have it run on the computer. Basically my position is that the right way to do recursive function theory is to use pure LISP as the programming language:

AIT = recursive function theory (**pure lisp**) + program size.

But that's not good enough, changes must be made to pure LISP so that it works for actually writing out programs in algorithmic information theory.

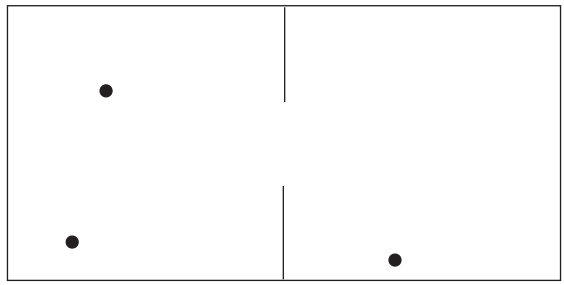
Let me say more forcefully why this is important. AIT says the complexity measure is the size of the smallest program for a universal Turing machine, and in AIT_2 it's got to be self-delimiting. Now there are two problems. First of all, which universal Turing machine do you pick? They're all pretty much equivalent but they give you slightly different complexity measures. Every universal Turing machine can simulate every other, but having to tell it which machine to simulate adds a certain number of bits to the complexity.

The other problem with the universal Turing machines used in AIT_1 and AIT_2 is that while you construct them mathematically, they're not machines that you can actually use to write programs and run them. I think that they should be.

So I have two things I want to do. On the one hand, I want to make algorithmic information theory, the theory of program-size complexity, be the size of programs in an actual powerful, usable programming language instead of an abstract, theoretical programming language that you can't use. The other thing I want is to actually pick one universal Turing machine, and let me emphasize why it's important to do that.

...I want to make algorithmic information theory, the theory of program-size complexity, be the size of programs in an actual powerful, usable programming language...

During an earlier visit here to the Santa Fe Institute I had a conversation with Carlton Caves. Carl, a physicist, was interested in applying program-size complexity to gedanken experiments in thermodynamics and statistical mechanics like Maxwell's demon. A typical thing that he would think about is that you have a chamber divided in half, and you have Avogadro's number of gas molecules, about 10^{23}. He was interested in whether each gas molecule is on the left or on the right in this chamber divided in two.



So you have about 10^{23} gas molecules and you get a bit string which is 10^{23} bits long where a zero means that a particular molecule is on the left and a one means it's on the right. Then Carl would use as his entropy measure the size of the smallest program that calculates this 10^{23}-bit string. But there's a problem. Since this theory, AIT, depends on the choice of universal Turing machine, you might pick a tremendously complicated universal Turing machine which would overwhelm the 10^{23} bits. You want to know that the choice of the machine is not going to make a difference that will be that big, otherwise AIT cannot be applied in this particular case. So it'd be nice to pick a particular machine and know what the complexity measure will be. So I've picked a particular machine, and that gives a much more concrete version of the theory.

WRITING AN AIT PROGRAM

But now let me tell you what we have to do to LISP so that you'll be able to write programs in algorithmic information theory, and how I pick a particular universal Turing machine to use to measure the size of programs. It's not that difficult, it's a few simple ideas. Here's the first step.

Since the main application of algorithmic information theory is to study the limits of mathematical reasoning, I need some way to talk about the complexity of a formal axiomatic system. The abstract view is that a formal axiomatic system is a recursively enumerable set of assertions. In other words, you have a proof-checking algorithm as Hilbert emphasized. You run through all possible proofs in size order, and you check which proofs are

correct. That means that given a set of axioms and a set of rules of inference both of which are specified very, very precisely, you can in principle just sit down and run through all possible proofs in size order and check which ones are correct—it's like a million monkeys typing away. This way, in principle you can print out all the theorems.

A formal axiomatic system is an unending computation that prints out an infinite set of strings which are the theorems, and my AIT gives you results limiting the power of a formal axiomatic system in terms of its complexity,...

A formal axiomatic system is an unending computation that prints out an infinite set of strings which are the theorems, and my AIT gives you results limiting the power of a formal axiomatic system in terms of its complexity, which is the size in bits of the smallest program that will print out all the theorems. So I need this complexity measure.

Now to pure LISP. Roughly speaking, pure LISP is like set theory; pure LISP is to computational mathematics as set theory is to theoretical, non-computational mathematics.

Instead of having the set

$$\{1, 2, 3\},$$

which is unordered, what pure LISP has is a list

$$(1\ 2\ 3)$$

where you use parentheses and you don't use commas to separate things. The main difference is that

$$(1\ 2\ 3)$$

is an ordered set (list!). In other words, there's a first, a second, and a third element. And you can nest things arbitrarily deep:

$$((1)\ (2\ 3)).$$

In pure LISP this is the data and these things are also the programs; everything is constructed out of this. It's the substance out of which you build your universe.

Also LISP is a functional language, rather than an imperative language. In other words, what you have are mathematical expressions for breaking apart and putting together lists. You don't *do* anything; there's no notion of time in pure LISP. Instead you define functions and apply the functions to arguments to see the values you get; no GO TO's, no assignment statements. Instead it's very much like mathematics in that you have expressions and you evaluate them. So it's a functional language, and a pure LISP program is actually a large expression that you evaluate giving you a value.

There's only one problem with this beautiful functional notion, with this arithmetic of lists, not numbers, with this expression-based language that's so clean and mathematical: you can't compute infinite sets. But I want to print out one by one all the theorems of a formal axiomatic system.

To add this capability to LISP, first of all, you put into LISP a primitive function called DISPLAY:

(display ...).

This is an identity function; the value of this

(display ...)

is the same as the value of the argument.... Oh, by the way, in LISP

$$f(x,y)$$

is written like this

$$(f\ x\ y).$$

That's the LISP notation for everything. So this

(display ...)

is just the identity function. But it does have a side-effect, which is to display the value of its argument. This is actually used in normal LISP for debugging. It's a way to get more than the final value, it's a way to look at intermediate results. But you don't care about the final value, if any. What you care about are the intermediate values, which are the theorems which you output using DISPLAY. So it's an evaluation which starts and will go on forever (but it might halt if there are only a finite number of theorems), and each theorem is put out as an intermediate result like this:

(display theorem).

This already exists in normal LISP. But it's not enough.

If somebody gives you a formal axiomatic system—that's a LISP expression whose evaluation will never complete and that will put out one by one these intermediate results—you need some way to get those intermediate results. The next step is to add a primitive function called TRY that's very, very important. TRY has two arguments. One is a time limit and the other is some LISP expression, which will often be a formal axiomatic system:

(try time-limit formal-axiomatic-system).

TRY starts the formal axiomatic system going, and it runs it for a limited amount of time. TRY captures the intermediate output, it captures the theorems. TRY is like EVAL. There's a thing called EVAL in LISP. You can put together an expression and then you can run it, and what EVAL does is tell you the value that you get by running the expression. TRY is like a time-limited EVAL, plus it gives you the intermediate results too. You can't EVAL a formal axiomatic system; that's an infinite com-

putation and you never get anything back. But you can TRY a formal axiomatic system, and then you get back three different things. If the evaluation of the LISP expression finishes, TRY will say that it has been completed and will give you the value. If not, TRY will let you know that it ran out of time. And in either case TRY will let you know all the intermediate results, all the theorems that were displayed. So the value that you get back from TRY is a pair:

(value/out-of-time captured-displays).

If the expression being tried is a formal axiomatic system with an infinite number of theorems, then the first element of the pair coming back from a TRY will always say that it ran out of time, and the second element of the pair will be all the theorems that it managed to prove before the time limit ran out.

TRY is really at the heart of my whole new theory AIT_3! If you understand this you understand everything that I've added to LISP to get algorithmic information theory to work.

This gives you a way to deal with infinite sets in LISP, but that's not enough. We also have to add binary data to LISP.

The obvious way to get a program-size complexity measure using LISP is to use as your measure the size in characters of LISP expressions, which are actually called "S-expressions." This is a nice concrete program-size measure, but it doesn't provide the correct complexity measure of AIT_2. We need a way to give LISP raw binary data, because LISP expressions aren't a good way to package algorithmic information. This is because LISP syntax means that there's redundancy and you're not using all the bits efficiently enough. So what we really want is a LISP expression plus a way to give it raw bits on the side. For a LISP expression to have access to binary data "on the side" means that the environment in which a LISP expression is evaluated doesn't just include the current variable bindings, it also includes a list of bits, a list of zeros and ones. To give a LISP expression access to this binary data we provide two new primitive functions: one to read the next bit, and one to read a complete LISP expression from the binary data. These are functions with no explicit arguments: you just write

(read-next-bit)

or

(read-next-S-expression).

In the first case, READ-NEXT-BIT, what you get when you evaluate it is either a zero or a one. It's the next bit of the binary data *if there is a next bit!* It's very important that if you've used up all the binary data and there is no next bit to read, then READ-NEXT-BIT explodes. In fact, this is the key step in getting AIT_2 out of LISP, that if you try to read a bit that isn't there you explode. What about READ-NEXT-S-EXPRESSION? What it does is read, say, seven bits at a time and interpret them as an ASCII character in a LISP expression. It keeps reading until parentheses balance and it has a complete LISP expression, or until it runs out of bits and fails.

The next question is, how do you give binary data to a LISP expression that wants some raw bits on the side? Well, you do it with TRY. I'm making TRY work overtime! TRY actually has three arguments. There's a time limit, there's an expression to be evaluated, and finally there's the binary data. So TRY ends up having three arguments:

(try time-limit expression binary-data).

TRY is really at the heart of my whole new theory AIT_3! If you understand this you understand everything that I've added to LISP to get algorithmic information theory to work. We've already seen the time limit and the expression that you try to evaluate for that amount of time. What's new is the third argument, the binary data, which could in fact be the empty list, in which case there is no binary data. And what's also new, and this is very, very important, is that now a TRY can fail in two ways. It can fail because you run out of time. Or it can fail because you run out of binary data, because you tried to read bits that weren't there. So the value that TRY returns now looks like this:

(value/out-of-time/out-of-data captured-displays).

So there are now two parts to the program, the expression and the data. I take the LISP expression and I measure its size in characters. Then I multiply by eight or seven bits per character. Or perhaps it's sixteen bits per character if you're using extended characters for Japanese. This gives me the size of the expression measured in bits instead of characters. And finally I just add the number of bits in the binary data. This is how I measure the size of a LISP expression with binary data on the side, and this includes the possibility that there's actually no binary data. So LISP expressions can now use two new primitive functions with no arguments to read a single bit or a LISP expression from the binary data. And if they do this they are charged one bit for each bit of the binary data that they read. So that's how I now measure the size of a program. And the program-size complexity of an object, of a LISP expression, is defined to be the size of the smallest program, of the smallest expression/binary data pair, that produces it, that yields it as its value.

In this way of looking at it, I'm not really using LISP as my programming language. Instead this LISP is sort of a high-level assembler to produce binary programs that I feed to a universal Turing machine.

So this is how we give raw bits to LISP expressions. And it is very important to note that you fail if you run out of binary data. You do not get a graceful end-of-file indication! If you did, we would get AIT_1 out of LISP, not AIT_2 with self-delimiting programs. And why are our programs self-delimiting? The LISP expression part is self-delimiting because parentheses have to balance. And the binary data that the LISP expression reads is self-delimiting because we are not allowed to run off the end of the data. It follows that our program-size complexity measure is additive. This means that the program-size complexity $H(x,y)$ of a pair of LISP expressions is bounded by the sum of the individual complexities $H(x)$ and $H(y)$ plus a constant:

$$H(x,y) \le H(x) + H(y) + c.$$

This only works with self-delimiting programs. It does not work in the original algorithmic information theory from the 1960s.

Let me explain this complexity measure in another way. Here is a reformulation of what I've just explained using TRY, using a universal Turing machine with binary programs. In this way of looking at it, I'm not really using LISP as my programming language. Instead this LISP is sort of a high-level assembler to produce binary programs that I feed to a universal Turing machine. This universal Turing machine reads its program from the binary data, bit by bit. The first thing it does is to read a complete LISP expression from the beginning of the binary data, which just means that it goes on until the parentheses balance. Then the Turing machine starts to run this prefix, to evaluate it, running it against the remainder of the binary data (if any's left). So there is a prefix, which is read eight or seven or sixteen bits at a time, and then the prefix starts to run and it can read in additional bits if it wants to by using READ-NEXT-BIT or READ-NEXT-S-EXPRESSION. And the prefix has to decide by itself how many bits to read, because it's not allowed to discover that no bits are left. If the prefix asks for a bit that isn't there, then the whole thing fails, and this wasn't a valid program for our universal Turing machine.

That turns out to be the whole story! That's how to get algorithmic information theory, and the right version of it, AIT_2, running on a computer. However, this only works because computers are so powerful now. If I had had this idea years ago, I wouldn't have been able to run any interesting examples, because the machines were too small and too slow.

So now I've picked out a particular universal Turing machine and my program-size complexity measure is very concrete, and I can actually write out the programs in LISP. Now let me tell you some of the sharp results that I get in this new more concrete theory, AIT_3.

Using this new approach, algorithmic information theory now becomes very concrete. Every time you have a theorem about program-size complexity, you can now actually write down the program that proves the theorem, and the size of this program gives you a precise numerical value for what was previously an undetermined constant in the statement of the theorem. Here is an important example, the inequality that

Every time you have a theorem about program-size complexity, you can now actually write down the program that proves the theorem, and the size of this program gives you a precise numerical value for what was previously an undetermined constant in the statement of the theorem.

$$H(x,y) \le H(x) + H(y) + c.$$

Let's go back to thinking about programs in the form of expression/data pairs:

(try time-limit expression binary-data).

So we have an expression/data pair that calculates a LISP expression x, and another expression/data pair that calculates a LISP expression y. The above inequality states that you can combine them to get an expression/data pair that calculates the list (xy), and this combined expression/data pair is exactly c bits bigger than the sum of the sizes of the given expression/data pairs. The LISP programming required to show this is trivial—although the programming details would require some explanation. It finally turns out that c is twenty characters which at seven bits per character is exactly 140 bits. So that's the value of the constant c:

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

Now this may not sound terribly exciting, but for me it was a tremendous revelation! Why? Because I've been proving theorems about program-size complexity all my life. But I never actually had a program in front of me and I never actually measured its size, and I never knew what the constant was in this inequality! For all I knew, this constant could have been 10^{99}! Now I know that it's only 140, thank goodness! This is important for Carlton Caves, because if this constant were large compared to Avogadro's number, then Carl couldn't use this basic inequality in his program-size complexity analysis of Maxwell's demon.

Now I'll discuss my main incompleteness theorem and how it looks in this new more concrete formulation of my theory. My main incompleteness result has to do with a number I call Ω, which is the halting probability of our universal Turing machine. One of the reasons that you want binary programs to be self-delimiting, to indicate within themselves where they end, is so that you can define this halting

My main result about Ω, and about this particular Ω too, is that Ω shows that you have randomness in pure mathematics.

probability. Here's how it works. Each time our universal Turing machine asks for a bit, feed it the result of an independent toss of a fair coin. This works because the machine decides by itself how many bits to read. That's why we can define the probability that a program of any size will halt. If programs weren't self-delimiting, then there wouldn't be a natural probability measure to put on the space of all programs. If programs weren't self-delimiting, then all the n-bit programs, if you give each of them probability 2^{-n}, would add up to probability unity, and how do you throw in programs of different sizes? So to be able to have a halting probability defined over programs of *any size*, these programs have to be self-delimiting. So it's very important for the universal Turing machine to decide by itself how many bits to read. Since it does, we get this halting probability Ω which is a real number between zero and one. It's the halting probability of the specific universal Turing machine that AIT_3 is based on. I explained before how this Turing machine works. Since this is a specific Turing machine, its halting probability Ω is now a specific real number. Before, Ω depended on our choice of universal Turing machine. The same theorems applied to each of these Ω's, but now it's a specific Ω that we're thinking about.

My main result about Ω, and about this particular Ω too, is that Ω shows that you have randomness in pure mathematics. Why? Let's say that you're trying to use formal reasoning, you're trying to use a formal axiomatic system to prove what the bits of this halting probability are. But you can't because these bits are accidental, there's no reason why they should be what they are, they're irreducible mathematical information. Essentially the only way to prove what an individual bit in a particular place in the binary expansion of Ω is, whether it's a zero or a one, is to add that fact as a new axiom. In other words, each bit of Ω has got to come out zero or one, but it's so delicately balanced whether it should come out one way or the other, that we're never going to know.

That's my old result from AIT_2. The new AIT_3 concrete incompleteness result is this: To determine n bits of Ω, you need a theory of complexity at least $n - 7581$. For the first 7581 bits of the halting probability, it might be that a formal axiomatic system can prove what these first 7581 bits are. But afterward, every time you want to prove what one more bit of Ω is, you have to add a bit to the complexity of the formal axiomatic system that you're using.

I can actually write down a program that computes better and better lower bounds on the halting probability. In fact, it gives the halting probability in the limit from below.

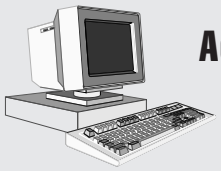
By the way, seven thousand bits is only a thousand characters which is only twenty lines of LISP code. So my proof of this incompleteness result involves only twenty lines of LISP code, if you compress out all the blanks and the comments.

In other words, after the first 7581 bits of Ω, every additional bit is going to cost you! In fact essentially the only way to be able to get a theorem out of a formal axiomatic system telling you what that bit is, is if you put the theorem in as a hypothesis, as a new axiom! That means that at that point reasoning is not really getting you anywhere any more.

In fact, the first seven bits of this particular halting probability Ω are all ones. I'm telling you that it's impossible to know the bits of the halting probability, but in fact I do know the first seven bits! This is an embarrassing fact, but now I know how bad it can be. Somewhere between the first seven bits and the first seven thousand it becomes impossible! The first seven bits are all ones, but now I know that you can go out at most a thousand times farther than that. After that, every time you want to prove what another bit of the halting probability is, you have to add a bit to your axioms, you have to add a bit to the complexity of your formal axiomatic system.

What exactly is the complexity of a formal axiomatic system? Well, the formal axiomatic system is now considered to be a LISP expression with binary data on the side. The formal axiomatic system is a program that goes on forever printing out the theorems. And you measure the complexity of the formal axiomatic system by taking the LISP expression, converting its size from characters to bits, and adding that to the number of bits in the binary data. So we're using the same complexity measure for infinite computations that we do for finite computations.

Here's another thing I can do with my augmented LISP. I have a program that's only about ten lines of LISP, which can actually compute lower bounds on the halting probability. Given n, this program looks at all n-bit programs, runs them for time n, and divides the number that halt by 2^n. That gives a lower bound on the halting probability, and the lower bounds get better and better as n gets larger. I have this program, and I've actually run it for all n up to 22. I've looked at all 22-bit programs. It's very easy to write out this program! This could never be done before. I can actually write down a program



Access Supplementary Material Relating To This Article At:

http://journals.wiley.com/complexity/

that computes better and better lower bounds on the halting probability. In fact, it gives the halting probability in the limit from below. So if I can write a program to compute it in the limit from below, it seems to me like a pretty definite number!

By the way, this most definitely doesn't mean that Ω is a computable real number like π. It isn't, because the convergence of this thing is unbelievably slow. You never know how far out to go to get a given degree of accuracy. If the halting probability were computable, then it would be very easy to prove what its bits are. It would be like π, you'd just calculate them! But at least I can write out this program that computes lower bounds, and I can even run it for small *n*.

A COURSE ON LIMITS

I put together a hands-on, computer-oriented course that I call "The limits of mathematics." There are several versions of this. And you can get all this stuff from **`chao-dyn`** at **`xyz.lanl.gov`**. What you do is you just go to Web address **`http://xyz.lanl.gov/`**. From there it's easy to find my stuff and download it.

The first version of this stuff that I sent to **`xyz.lanl.gov`** is a preliminary version of all of this in which the LISP interpreter is written in Mathematica [2]. You can also get this version of my course from MathSource via **`http://www.wri.com/`**. The LISP interpreter is a few pages of Mathematica code.

Then a friend of mine, George Markowsky, who's a professor at the University of Maine in Orono, invited me to give an intensive short course on the limits of mathematics using this hands-on approach. With his help I took this Mathematica program, and converted it into C so that it could run on small personal computers. The C version of my LISP interpreter [3] does exactly the same thing that the Mathematica version does, but it's much faster. It's also a much larger program, and the code is much more difficult to understand.

Besides the LISP interpreter in C, I also improved the LISP programs. This second version of "The limits of mathematics" starts off with a reference manual for my LISP, and the rest of the book is just software written in this LISP. Each of the LISP programs has comments. The comments in the program tell you what theorem the program proves. And there's usually a constant in the statement of the theorem, and that constant is the size of the program. This book may be the ultimate in constructive mathematics, but it is not easy to read! The problem is that the result is a reference manual and a lot of software, and it is not easy to understand. In fact, there is a more aggressive version [4] of my course with much smaller constants, that's even harder to understand!

There's also a rather technical extended abstract that states all the results in a few pages [5], and a summary of the main ideas [6] that is probably the most understandable version of all of this.

Maybe the problem is that my LISP is a bit Spartan. It only allows one-character variable names, and arithmetic has to be programmed out, it is not built-in. So perhaps I should take the trouble to flesh out my LISP and make it friendlier [7, 8].

Nature's Imagination includes an article of mine on "Randomness in arithmetic and the decline and fall of reductionism in pure mathematics" [9] which is a talk I gave at Cambridge University two years ago, and at that time I thought it summarized everything fundamental that I had to say about the limits of mathematics. Although AIT_3 is not in there, it's a pretty understandable summary of what I think all of this implies about how we should actually do mathematics. This article is also in a book [10] edited by Anders Karlqvist and John Casti.

ACKNOWLEDGMENT

This is an edited transcript of a lecture given Friday, April 7, 1995 at the Santa Fe Institute, Santa Fe, New Mexico. The lecture was videotaped.

REFERENCES

1. H. Rogers: Theory of recursive functions and effective computability. McGraw-Hill, 1967.
2. G. J. Chaitin: The limits of mathematics—course outline & software. 127 pp., December 1993.
3. G. J. Chaitin: The limits of mathematics. 270 pp., July 1994.
4. G. J. Chaitin: The limits of mathematics IV. 231 pp., July 1994.
5. G. J. Chaitin: The limits of mathematics—extended abstract. 7 pp., July 1994.
6. G. J. Chaitin: A new version of algorithmic information theory. Complexity 1, pp. 55–59, 1996.
7. G. J. Chaitin: The limits of mathematics—tutorial version. 143 pp., September 1995.
8. G. J. Chaitin: The limits of mathematics. 45 pp., September 1995.
9. G. J. Chaitin: Randomness in arithmetic and the decline and fall of reductionism in pure mathematics. In: Nature's imagination. J. Cornwell (Ed.) Oxford University Press, 1995, pp. 27 - 44.
10. J. L. Casti and A. Karlqvist: Cooperation and conflict in general evolutionary processes. Wiley, New York, 1995.

ADDITIONAL READING

- G. J. Chaitin: Algorithmic information theory. Revised third printing, Cambridge University Press, 1990.
- G. J. Chaitin: Information, randomness & incompleteness. Second edition, World Scientific, 1990.
- G. J. Chaitin: Information-theoretic incompleteness. World Scientific, 1992.
- G. J. Chaitin: The Berry paradox. Complexity 1: pp. 26 - 30, 1995.