



EXCERPTED FROM

STEPHEN  
WOLFRAM  
A NEW  
KIND OF  
SCIENCE

---

CHAPTER 11

*The Notion of  
Computation*



---

# The Notion of Computation

## **Computation as a Framework**

In earlier parts of this book we saw many examples of the kinds of behavior that can be produced by cellular automata and other systems with simple underlying rules. And in this chapter and the next my goal is to develop a general framework for thinking about such behavior.

Experience from traditional science might suggest that standard mathematical analysis should provide the appropriate basis for any such framework. But as we saw in the previous chapter, such analysis tends to be useful only when the overall behavior one is studying is fairly simple.

So what can one do when the behavior is more complex?

If traditional science was our only guide, then at this point we would probably be quite stuck. But my purpose in this book is precisely to develop a new kind of science that allows progress to be made in such cases. And in many respects the single most important idea that underlies this new science is the notion of computation.

Throughout this book I have referred to systems such as cellular automata as simple computer programs. So now the point is actually to think of these systems in terms of the computations they can perform.

In a typical case, the initial conditions for a system like a cellular automaton can be viewed as corresponding to the input to a computation, while the state of the system after some number of steps corresponds to the output. And the key idea is then to think in purely

abstract terms about the computation that is performed, without necessarily looking at all the details of how it actually works.

Why is such an abstraction useful? The main reason is that it potentially allows one to discuss in a unified way systems that have completely different underlying rules. For even though the internal workings of two systems may have very little in common, the computations the systems perform may nevertheless be very similar.

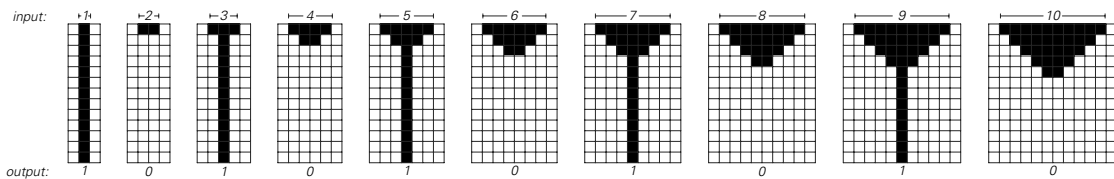
And by thinking in terms of such computations, it then becomes possible to imagine formulating principles that apply to a very wide variety of different systems—quite independent of the detailed structure of their underlying rules.

### Computations in Cellular Automata

I have said that the evolution of a system like a cellular automaton can be viewed as a computation. But what kind of computation is it, and how does it compare to computations that we typically do in practice?

The pictures below show an example of a cellular automaton whose evolution can be viewed as performing a particular simple computation.

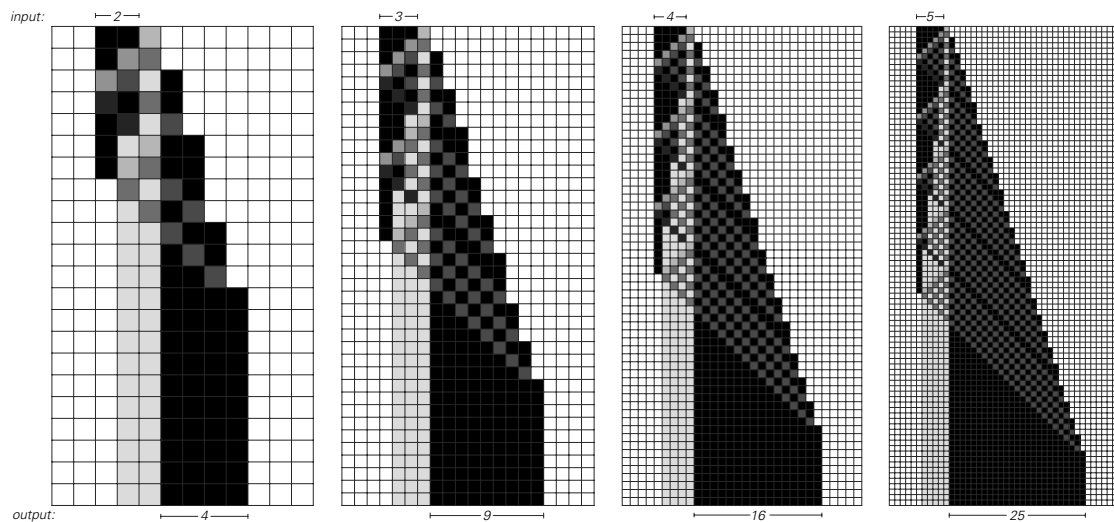
If one starts this cellular automaton with an even number of black cells, then after a few steps of evolution, no black cells are left. But if instead one starts it with an odd number of black cells, then a single black cell survives forever. So in effect this cellular automaton can be viewed as computing whether a given number is even or odd.



A simple cellular automaton whose evolution effectively computes the remainder after division of a number by 2. Starting from a row of  $n$  black cells, 0 black cells survive if  $n$  is even, and 1 black cell survives if  $n$  is odd. The cellular automaton follows elementary rule 132, as shown on the left.

One specifies the input to the computation by setting up an appropriate number of initial black cells. And then one determines the result of the computation by looking at how many black cells survive in the end.

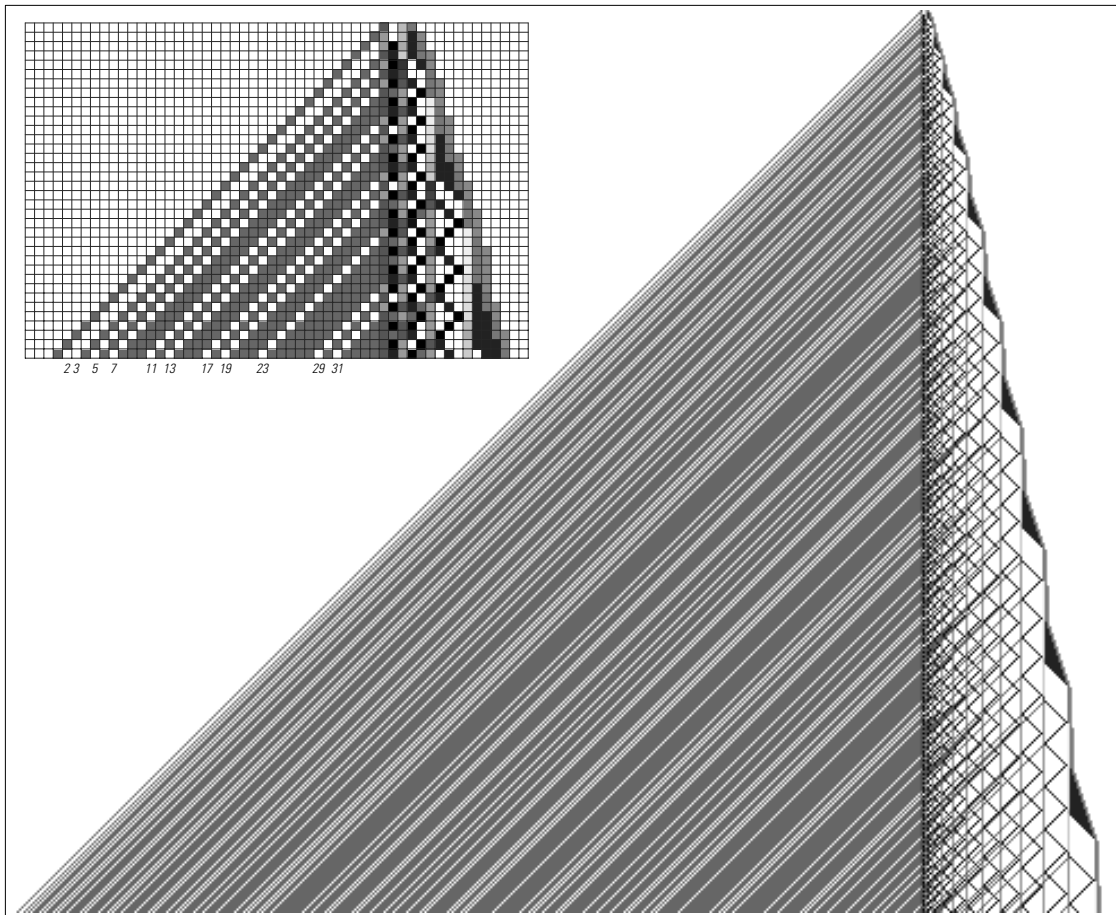
Testing whether a number is even or odd is by most measures a rather simple computation. But one can also get cellular automata to do more complicated computations. And as an example the pictures below show a cellular automaton that computes the square of any number. If one starts say with 5 black squares, then after a certain number of steps the cellular automaton will produce a block of exactly  $5 \times 5 = 25$  black squares.



A cellular automaton that computes the square of any number. The cellular automaton effectively works by adding the original number  $n$  together  $n$  times. The underlying rule used here involves eight possible colors for each cell.

At first it might seem surprising that a system with the simple underlying structure of a cellular automaton could ever be made to perform such a computation. But as we shall see later in this chapter, cellular automata can in fact perform what are in effect arbitrarily sophisticated computations. And as one example of a somewhat more sophisticated computation, the picture on the next page shows a cellular automaton that computes the successive prime numbers: 2, 3, 5, 7, 11, 13, 17, etc.

The rule for this cellular automaton is somewhat complicated—it involves a total of sixteen colors possible for each cell—but the example demonstrates the point that in principle a cellular automaton can compute the primes.



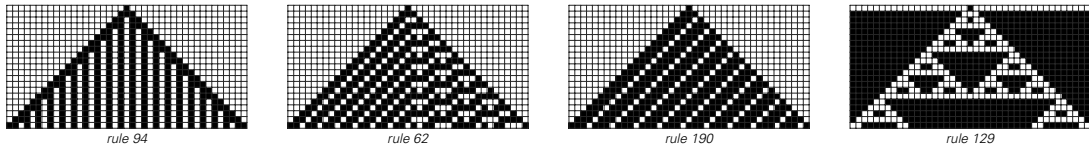
A cellular automaton constructed to compute the prime numbers. The system generates a dark gray stripe on the left at all positions that correspond to any product of numbers other than 1. White gaps then remain at positions that correspond to the prime numbers 2, 3, 5, 7, 11, 13, 17, etc. The cellular automaton effectively does its computation using the standard sieve of Eratosthenes method. The structures on the right bounce backwards and forwards with repetition periods corresponding to successive odd numbers. Once in each period they produce a gray stripe which propagates to the left, so that in the end there is a gray stripe corresponding to every multiple of every number. The rule for the cellular automaton shown here involves 16 possible colors for each cell.

So what about the cellular automata that we discussed earlier in this book? What kinds of computations can they perform?

At some level, any cellular automaton—or for that matter, any system whatsoever—can be viewed as performing a computation that determines what its future behavior will be.

But for the cellular automata that I have discussed in this section, it so happens that the computations they perform can also conveniently be described in terms of traditional mathematical notions.

And this turns out to be possible for some of the cellular automata that I discussed earlier in this book. Thus, for example, as shown below, rule 94 can effectively be described as enumerating even numbers. Similarly, rule 62 can be thought of as enumerating numbers that are multiples of 3, while rule 190 enumerates numbers that are multiples of 4. And if one looks down the center column of the pattern it produces, rule 129 can be thought of as enumerating numbers that are powers of 2.

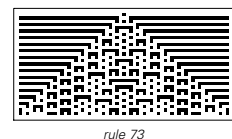
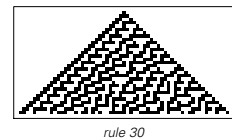


Examples of simple cellular automata whose evolution corresponds to computations that can easily be described in traditional mathematical terms. In analogy to the previous page, the positions of white cells at the bottom of the rule 94 picture correspond to even numbers, on the left in rule 62 to multiples of 3, in rule 190 to multiples of 4, and in the center column of rule 129 to powers of 2.

But what kinds of computations are cellular automata like the ones on the right performing? If we compare the patterns they produce to the patterns we have seen so far in this section, then immediately we suspect that we cannot describe these computations by anything as simple as saying, for example, that they generate primes.

So how then can we ever expect to describe these computations? Traditional mathematics is not much help, but what we will see is that there are a collection of ideas familiar from practical computing that provide at least the beginnings of the framework that is needed.

Examples of cellular automata that have simple underlying rules but whose overall behavior does not seem to correspond to computations with any kind of simple description in standard mathematical or other terms. ▶



## The Phenomenon of Universality

In the previous section we saw that it is possible to get cellular automata to perform some fairly sophisticated computations. But for each specific computation we wanted to do, we always set up a cellular automaton with a different set of underlying rules. And indeed our everyday experience with mechanical and other devices might lead us to assume that in general in order to perform different kinds of tasks we must always use systems that have different underlying constructions.

But the remarkable discovery that launched the computer revolution is that this is not in fact the case. And instead, it is possible to build universal systems whose underlying construction remains fixed, but which can be made to perform different tasks just by being programmed in different ways.

And indeed, this is exactly how practical computers work: the hardware of the computer remains fixed, but the computer can be programmed for different tasks by loading different pieces of software.

The idea of universality is also the basis for computer languages. For in each language, there are a certain set of primitive operations, which are then strung together in different ways to create programs for different tasks.

The details of a particular computer system or computer language will certainly affect how easy it is to perform a particular task. But the crucial fact that is by now a matter of common knowledge is that with appropriate programming any computer system or computer language can ultimately be made to perform exactly the same set of tasks.

One way to see that this must be true is to note that any particular computer system or computer language can always be set up by appropriate programming to emulate any other one.

Typically the way this is done is by having each individual action in the system that is to be emulated be reproduced by some sequence of actions in the other system. And indeed this is ultimately how, for example, *Mathematica* works. For when one enters a command such as `Log[15]`, what actually happens is that the program which implements the *Mathematica* language interprets this command

by executing the appropriate sequence of machine instructions on whatever computer system one is using.

And having now identified the phenomenon of universality in the context of practical computing, one can immediately see various analogs of it in other areas of common experience. Human languages provide an example. For one knows that given a single fixed underlying language, it is possible to describe an almost arbitrarily wide range of things. And given any two languages, it is for the most part always possible to translate between them.

So what about natural science? Is the phenomenon of universality also relevant there? Despite its great importance in computing and elsewhere, it turns out that universality has in the past never been considered seriously in relation to natural science.

But what I will show in this chapter and the next is that in fact universality is for example quite crucial in finding general ways to characterize and understand the complexity we see in natural systems.

The basic point is that if a system is universal, then it must effectively be capable of emulating any other system, and as a result it must be able to produce behavior that is as complex as the behavior of any other system. So knowing that a particular system is universal thus immediately implies that the system can produce behavior that is in a sense arbitrarily complex.

But now the question is what kinds of systems are in fact universal.

Most present-day mechanical devices, for example, are built only for rather specific tasks, and are not universal. And among electronic devices there are examples such as simple calculators and electronic address books that are not universal. But by now the vast majority of practical electronic devices, despite all their apparent differences, are based on computers that are universal.

At some level, however, these computers tend to be extremely similar. Indeed, essentially all of them are based on the same kinds of logic circuits, the same basic layout of data paths, and so on. And knowing this, one might conclude that any system which was universal must include direct analogs of these specific elements. But from



experience with computer languages, there is already an indication that the range of systems that are universal might be somewhat broader.

Indeed, *Mathematica* turns out to be a particularly good example, in which one can pick very different sets of operations to use, and yet still be able to implement exactly the same kinds of programs.

So what about cellular automata and other systems with simple rules? Is it possible for these kinds of systems to be universal?

At first, it seems quite implausible that they could be. For the intuition that one gets from practical computers and computer languages seems to suggest that to achieve universality there must be some fundamentally fairly sophisticated elements present.

But just as we found that the intuition which suggests that simple rules cannot lead to complex behavior is wrong, so also the intuition that simple rules cannot be universal also turns out to be wrong. And indeed, later in this chapter, I will show an example of a cellular automaton with an extremely simple underlying rule that can nevertheless in the end be seen to be universal.

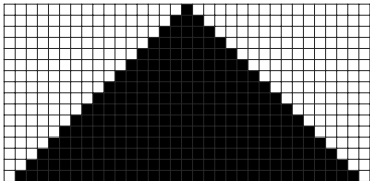
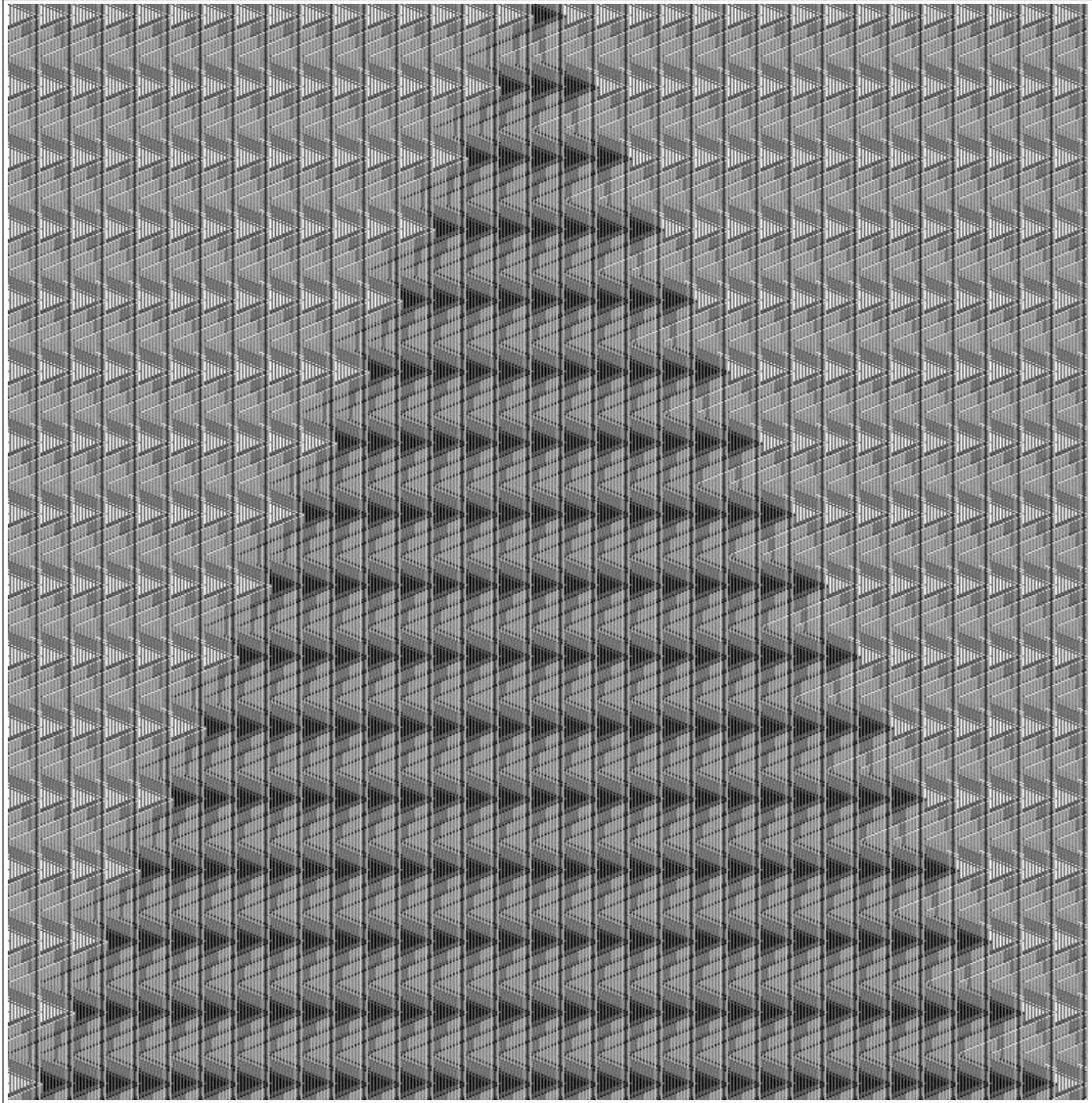
In the past it has tended to be assumed that universality is somehow a rare and special quality, usually possessed only by systems that are specifically constructed to have it. But one of the results of this chapter is that in fact universality is a much more widespread phenomenon. And in the next chapter I will argue that for example it also occurs in a wide range of important systems that we see in nature.

## **A Universal Cellular Automaton**

As our first specific example of a system that exhibits universality, I discuss in this section a particular universal cellular automaton that has been set up to make its operation as easy to follow as possible.

The rules for this cellular automaton itself are always the same. But the fact that it is universal means that if it is given appropriate initial conditions it can effectively be programmed to emulate for example any possible cellular automaton—with any set of rules.

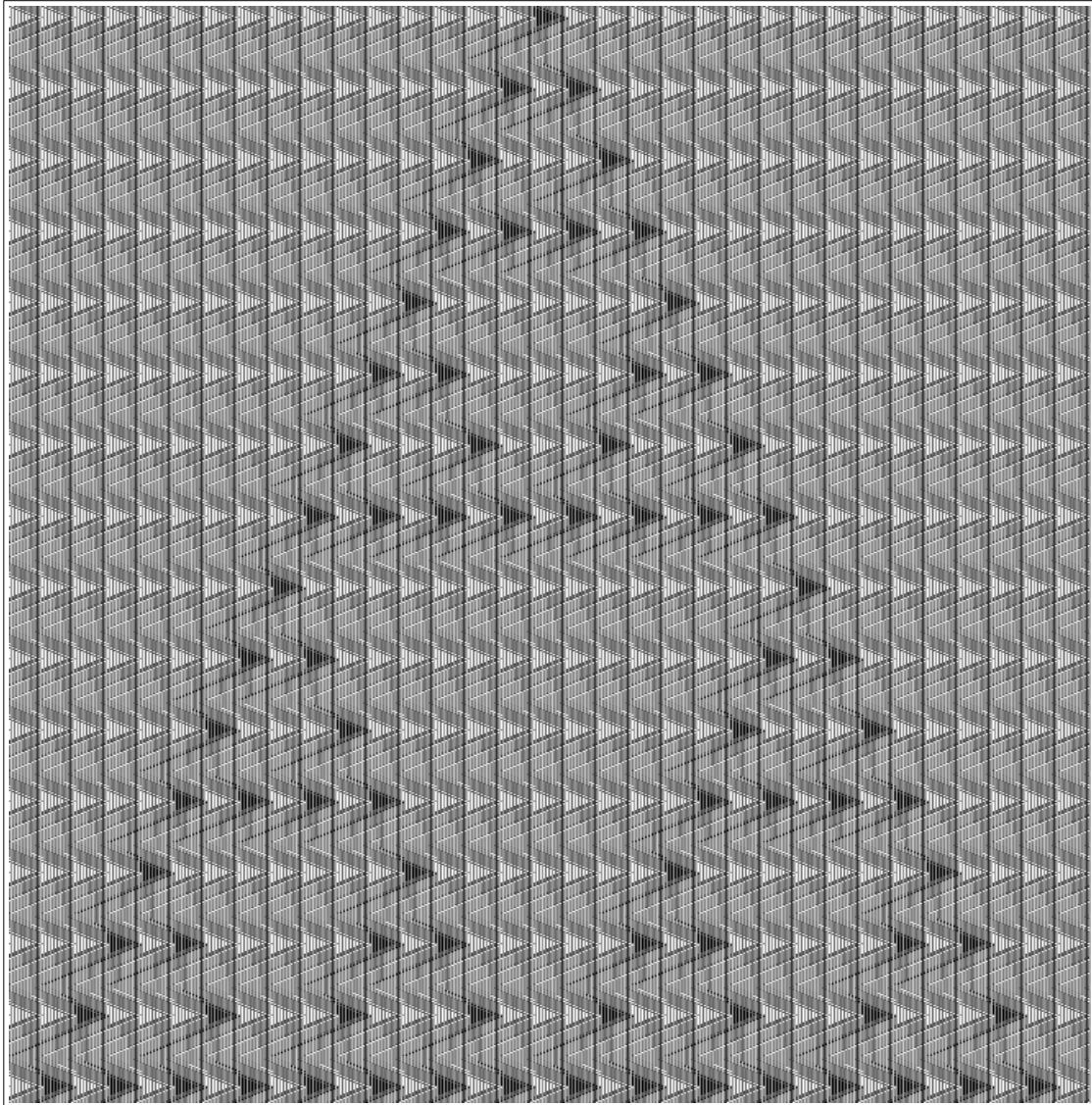
The next three pages show three examples of this.



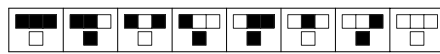
The universal cellular automaton emulating elementary rule 254. Each cell in rule 254 is represented by a block of 20 cells in the universal cellular automaton. Each of these blocks encodes both the color of the cell it represents, and the rule for updating this color.



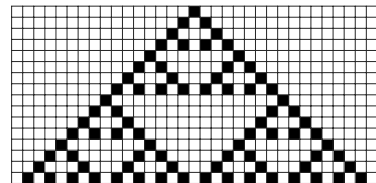
rule 254

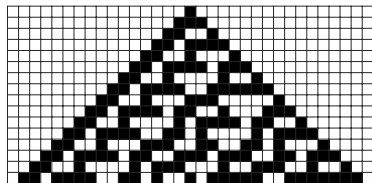
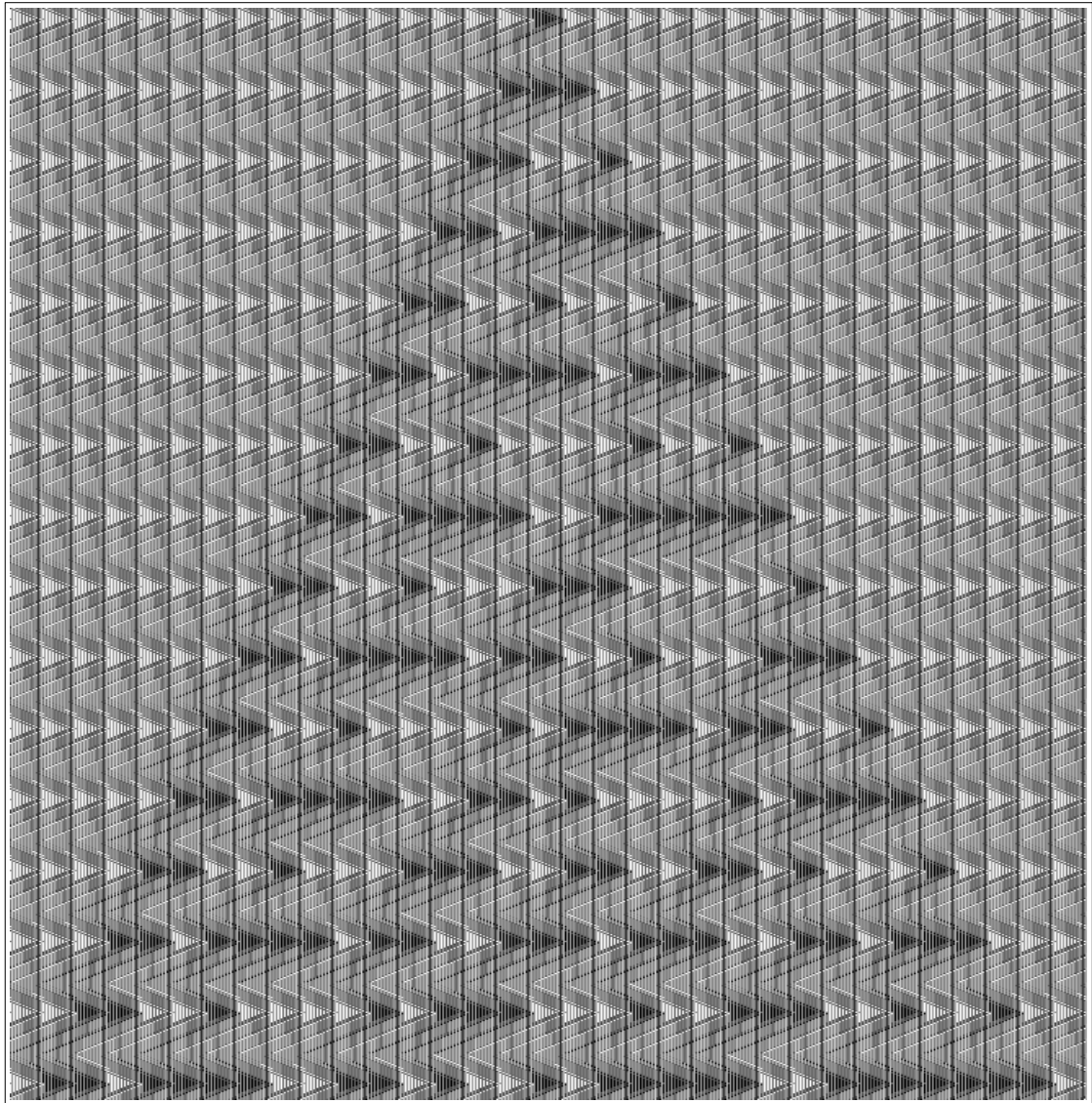


The universal cellular automaton emulating elementary rule 90. The underlying rules for the universal cellular automaton are exactly the same as on the previous page. But each block in the initial conditions now contains a representation of rule 90 rather than rule 254.

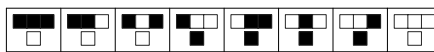


rule 90





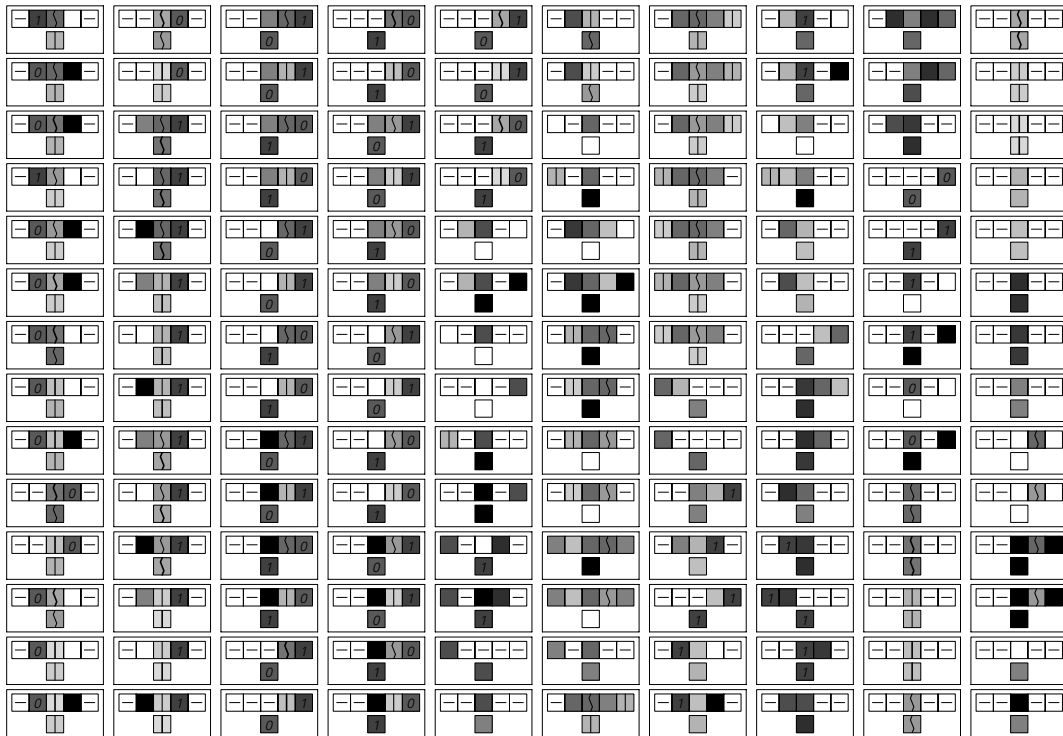
The universal cellular automaton emulating rule 30. A total of 848 steps in the evolution of the universal cellular automaton are shown, corresponding to 16 steps in the evolution of rule 30.



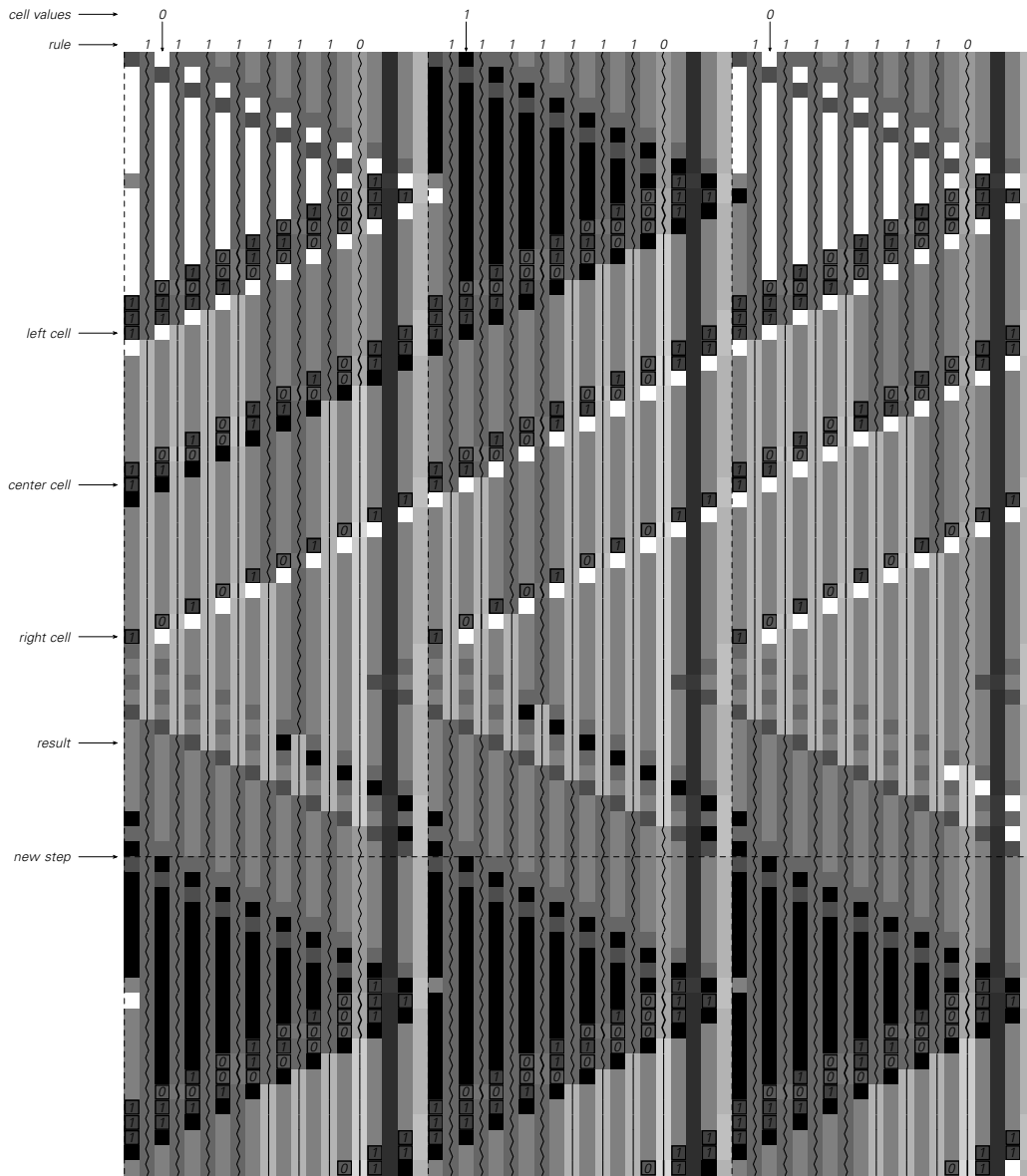
*rule 30*

On each page the underlying rules for the universal cellular automaton are exactly the same. But on the first page, the initial conditions are set up so as to make the universal cellular automaton emulate rule 254, while on the second page they are set up to make it emulate rule 90, and on the third page rule 30.

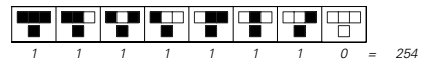
The pages that follow show how this works. The basic idea is that a block of 20 cells in the universal cellular automaton is used to represent each single cell in the cellular automaton that is being emulated. And within this block of 20 cells is encoded both a specification of the current color of the cell that is being represented, as well as the rule by which that color is to be updated.

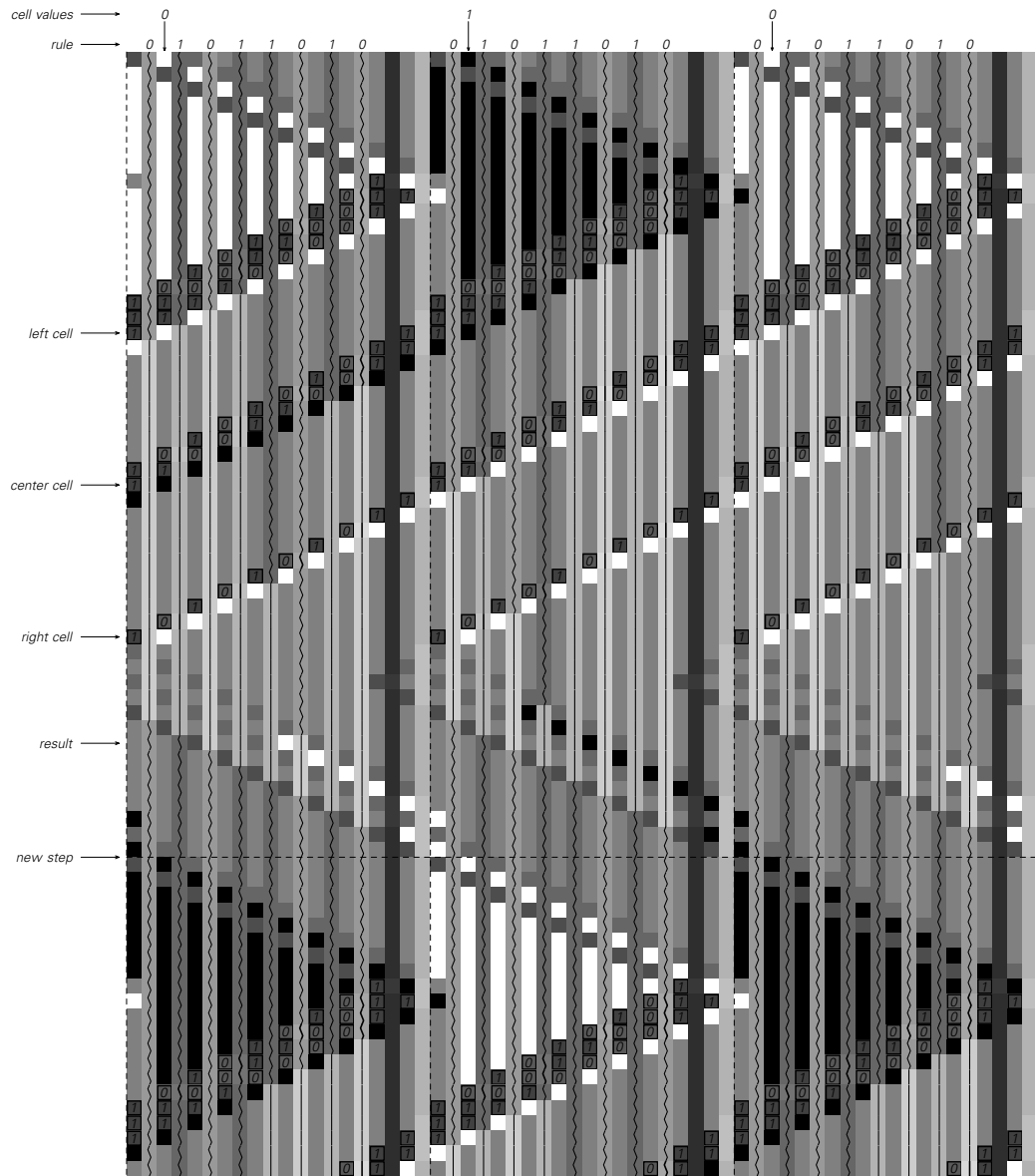


The rules for the universal cellular automaton. There are 19 possible colors for each cell, represented here by 19 different icons. Since the new color of each cell depends on the previous colors of a total of five cells, there are in principle 2,476,099 cases to cover. But by using □ to stand for a cell with any possible color, many cases are combined. Note that the cases shown are in a definite order reading down successive columns, with special cases given before more general ones. With the initial conditions used, there are some combinations of cells that can never occur, and these are not covered in the rules shown.

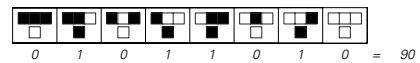


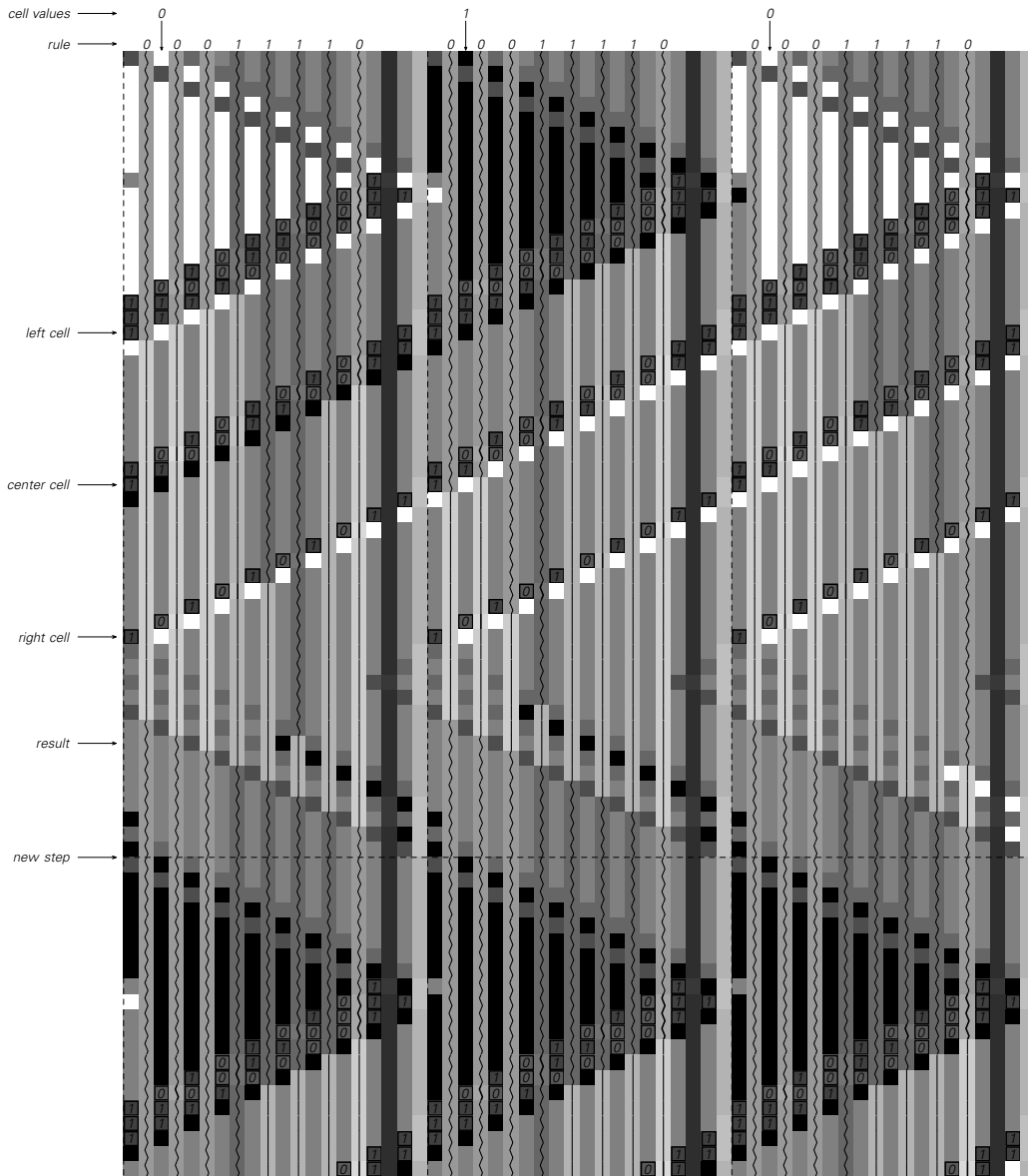
Details of how the universal cellular automaton emulates rule 254. Each of the blocks in the universal cellular automaton represents a single cell in rule 254, and encodes both the current color of the cell and the form of the rule used to update it.



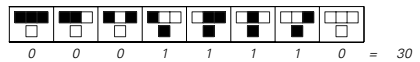


Details of how the universal cellular automaton emulates rule 90. The only difference in initial conditions from the picture on the previous page is that each block now encodes rule 90 instead of rule 254.





Details of how the universal cellular automaton emulates rule 30. Once again, the only difference in initial conditions from the facing page is that each block now encodes rule 30 instead of rule 90.





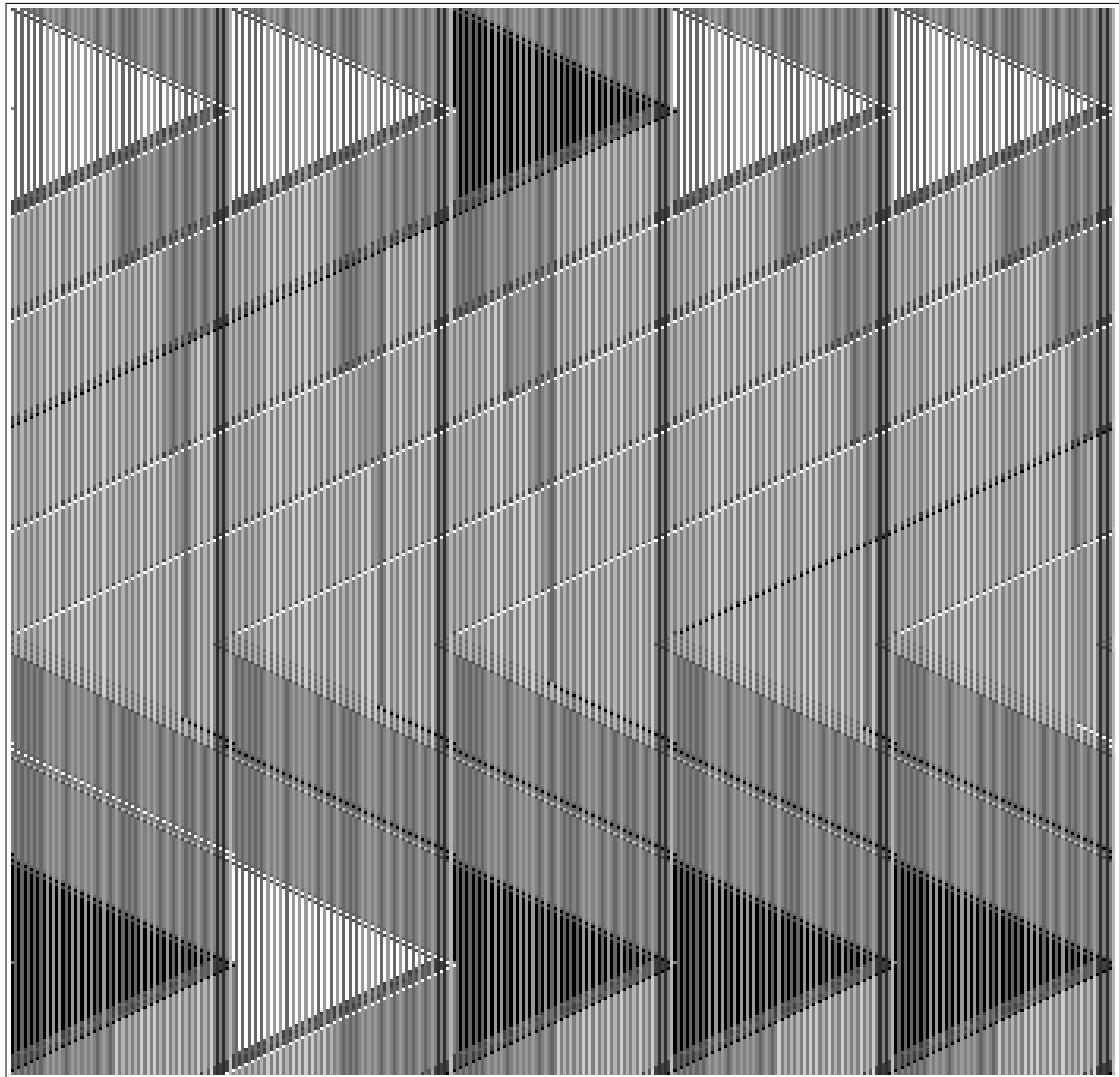
In the examples shown, the cellular automata being emulated have 8 cases in their rules, with each case giving the outcome for one of the 8 possible combinations of colors of a cell and its immediate neighbors. In every block of 20 cells in the universal cellular automaton, these rules are encoded in a very straightforward way, by listing in order the outcomes for each of the 8 possible cases.

To update the color of the cell represented by a particular block, what the universal cellular automaton must then do is to determine which of the 8 cases applies to that cell. And it does this by successively eliminating cases that do not apply, until eventually only one case remains. This process of elimination can be seen quite directly in the pictures on the previous pages. Below each large black or white triangle, there are initially 8 vertical dark lines. Each of these lines corresponds to one of the 8 cases in the rule, and the system is set up so that a particular line ends as soon as the case to which it corresponds has been eliminated.

It so happens that in the universal cellular automaton discussed here the elimination process for a given cell always occurs in the block immediately to the left of the one that represents that cell. But the process itself is not too difficult to understand, and indeed it works in much the way one might expect of a practical electronic logic circuit.

There are three basic stages, visible in the pictures as three stripes moving to the left across each block. The first stripe carries the color of the left-hand neighbor, and causes all cases in the rule where that neighbor does not have the appropriate color to be eliminated. The next two stripes then carry the color of the cell itself and of its right-hand neighbor. And after all three stripes have passed, only one of the 8 cases ever survives, and this case is then the one that gives the new color for the cell.

The pictures on the last few pages have shown how the universal cellular automaton can in effect be programmed to emulate any cellular automaton whose rules involve nearest neighbors and two possible colors for each cell. But the universal cellular automaton is in no way restricted to emulating only rules that involve nearest neighbors. And thus on the facing page, for example, it is shown emulating a rule that involves next-nearest as well as nearest neighbors.



The universal cellular automaton emulating one step in the evolution of the rule shown above, which involves next-nearest as well as nearest-neighbor cells. The rule now covers a total of 32 cases, corresponding to the possible arrangements of colors of a cell and its nearest and next-nearest neighbors. The picture shows the evolution of five cells according to the rule shown, with each cell now being represented by a block of 70 cells in the universal cellular automaton.

The blocks needed to represent each cell are now larger, since they must include all 32 cases in the rule. There are also five elimination stages rather than three. But despite these differences, the underlying rule for the universal cellular automaton remains exactly the same.

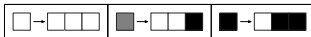
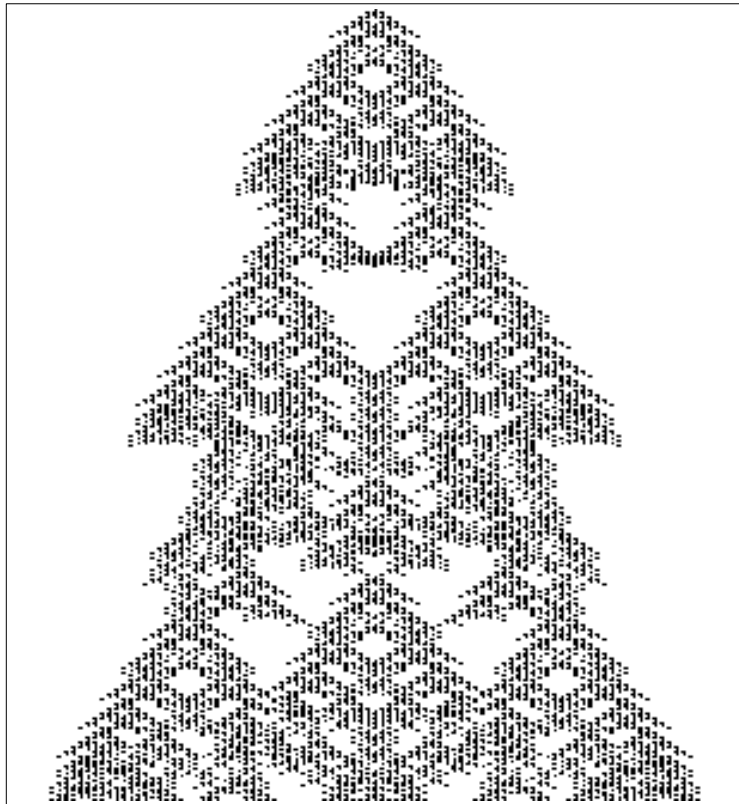
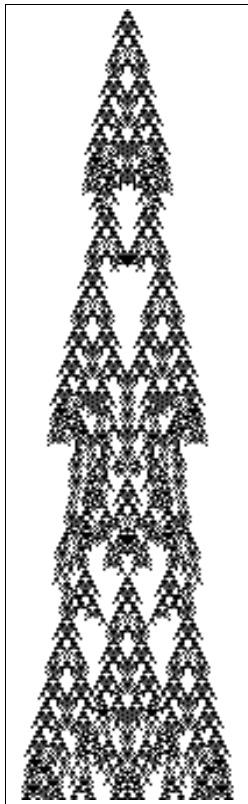
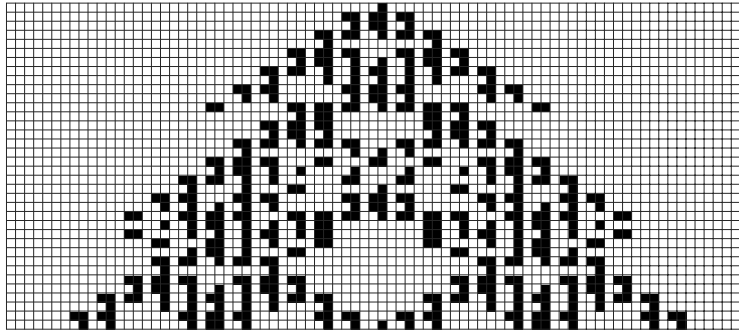
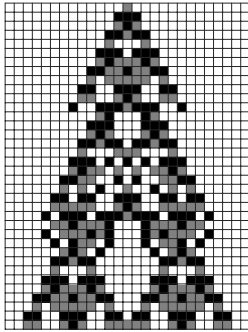
What about rules that have more than two possible colors for each cell? It turns out that there is a general way of emulating such rules by using rules that have just two colors but a larger number of neighbors. The picture on the facing page shows an example. The idea is that each cell in the three-color cellular automaton is represented by a block of three cells in the two-color cellular automaton. And by looking at neighbors out to distance five on each side, the two-color cellular automaton can update these blocks at each step in direct correspondence with the rules of the three-color cellular automaton.

The same basic scheme can be used for rules with any number of colors. And the conclusion is therefore that the universal cellular automaton can ultimately emulate a cellular automaton with absolutely any set of rules, regardless of how many neighbors and how many colors they may involve.

This is an important and at first surprising result. For among other things, it implies that the universal cellular automaton can emulate cellular automata whose rules are more complicated than its own. If one did not know about the basic phenomenon of universality, then one would most likely assume that by using more complicated rules one would always be able to produce new and different kinds of behavior.

But from studying the universal cellular automaton in this section, we now know that this is not in fact the case. For given the universal cellular automaton, it is always in effect possible to program this cellular automaton to emulate any other cellular automaton, and therefore to produce whatever behavior the other cellular automaton could produce.

In a sense, therefore, what we can now see is that nothing fundamental can ever be gained by using rules that are more complicated than those for the universal cellular automaton. For given the universal cellular automaton, more complicated rules can always be emulated just by setting up appropriate initial conditions.



An example of how a cellular automaton with three possible colors and nearest-neighbor rules can be emulated by a cellular automaton with only two possible colors but a larger number of neighbors (in this case five on each side). The basic idea is to represent each cell in the three-color rule by a block of three cells in the two-color rule, according to the correspondence given on the left. The three-color rule illustrated here is totalistic code 1599 from page 70.

Looking at the specific universal cellular automaton that we have discussed in this section, however, we would probably be led to assume that while the phenomenon of universality might be important in principle, it would rarely be relevant in practice. For the rules of the universal cellular automaton in this section are quite complicated—involving 19 possible colors for each cell, and next-nearest as well as nearest neighbors. And if such complication was indeed necessary in order to achieve universality, then one would not expect that universality would be common, for example, in the systems we see in nature.

But what we will discover later in this chapter is that such complication in underlying rules is in fact not needed. Indeed, in the end we will see that universality can actually occur in cellular automata with just two colors and nearest neighbors. The operation of such cellular automata is considerably more difficult to follow than the operation of the universal cellular automaton discussed in this section. But the existence of universal cellular automata with such simple underlying rules makes it clear that the basic results we have obtained in this section are potentially of very broad significance.

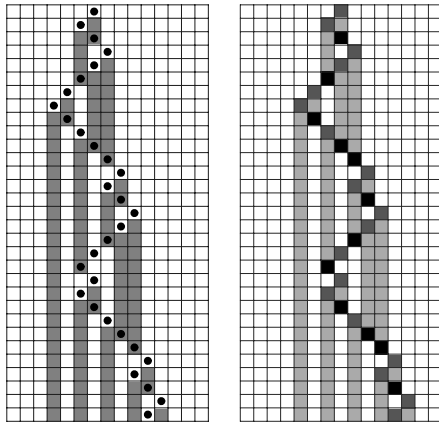
### **Emulating Other Systems with Cellular Automata**

The previous section showed that a particular universal cellular automaton could emulate any possible cellular automaton. But what about other types of systems? Can cellular automata also emulate these?

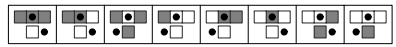
With their simple and rather specific underlying structure one might think that cellular automata would never be capable of emulating a very wide range of other systems. But what I will show in this section is that in fact this is not the case, and that in the end cellular automata can actually be made to emulate almost every single type of system that we have discussed in this book.

As a first example of this, the picture on the facing page shows how a cellular automaton can be made to emulate a mobile automaton.

The main difference between a mobile automaton and a cellular automaton is that in a mobile automaton there is a special active cell that moves around from one step to the next, while in a cellular



An example of a mobile automaton (see page 71) being emulated by a cellular automaton. In the mobile automaton shown on the left each cell has two possible colors. In the cellular automaton shown on the right, the cells have four possible colors, with two darker colors corresponding to the active cell in the mobile automaton. The rules for the mobile automaton and the cellular automaton are shown below. In the rules for the cellular automaton,  $\square$  indicates a cell of any color.

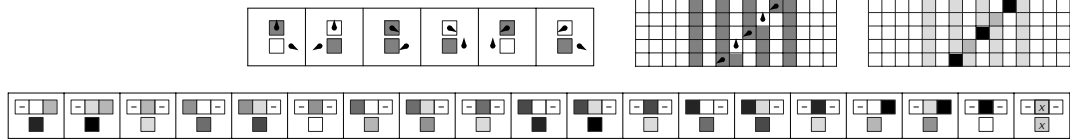


automaton all cells are always effectively treated as being exactly the same. And to emulate a mobile automaton with a cellular automaton it turns out that all one need do is to divide the possible colors of cells in the cellular automaton into two sets: lighter ones that correspond to ordinary cells in the mobile automaton, and darker ones that correspond to active cells. And then by setting up appropriate rules and choosing initial conditions that contain only one darker cell, one can produce in the cellular automaton an exact emulation of every step in the evolution of a mobile automaton—as in the picture above.

The same basic approach can be used to construct a cellular automaton that emulates a Turing machine, as illustrated on the next page. Once again, lighter colors in the cellular automaton represent ordinary cells in the Turing machine, while darker colors represent the cell under the head, with a specific darker color corresponding to each possible state of the head.

One might think that the reason that mobile automata and Turing machines can be emulated by cellular automata is that they both consist of fixed arrays of cells, just like cellular automata. So then one may wonder what happens with substitution systems, for example, where there is no fixed array of elements.

An example of a Turing machine being emulated by a cellular automaton. In the Turing machine on the left each cell has two possible colors, and the head has three possible states. In the cellular automaton, the cells have eight possible colors, with the lightest two colors being used for cells not at the position of the head. The rules for the Turing machine and the cellular automaton are shown below. In the rules for the cellular automaton,  $\square$  indicates a cell of any color.

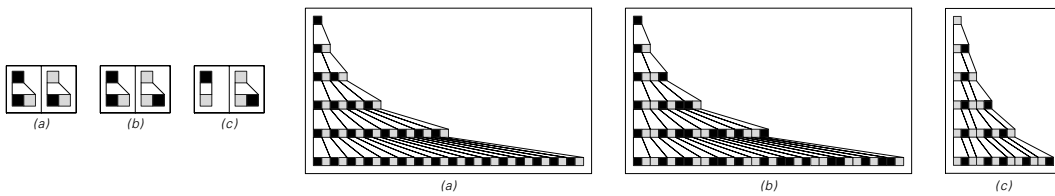
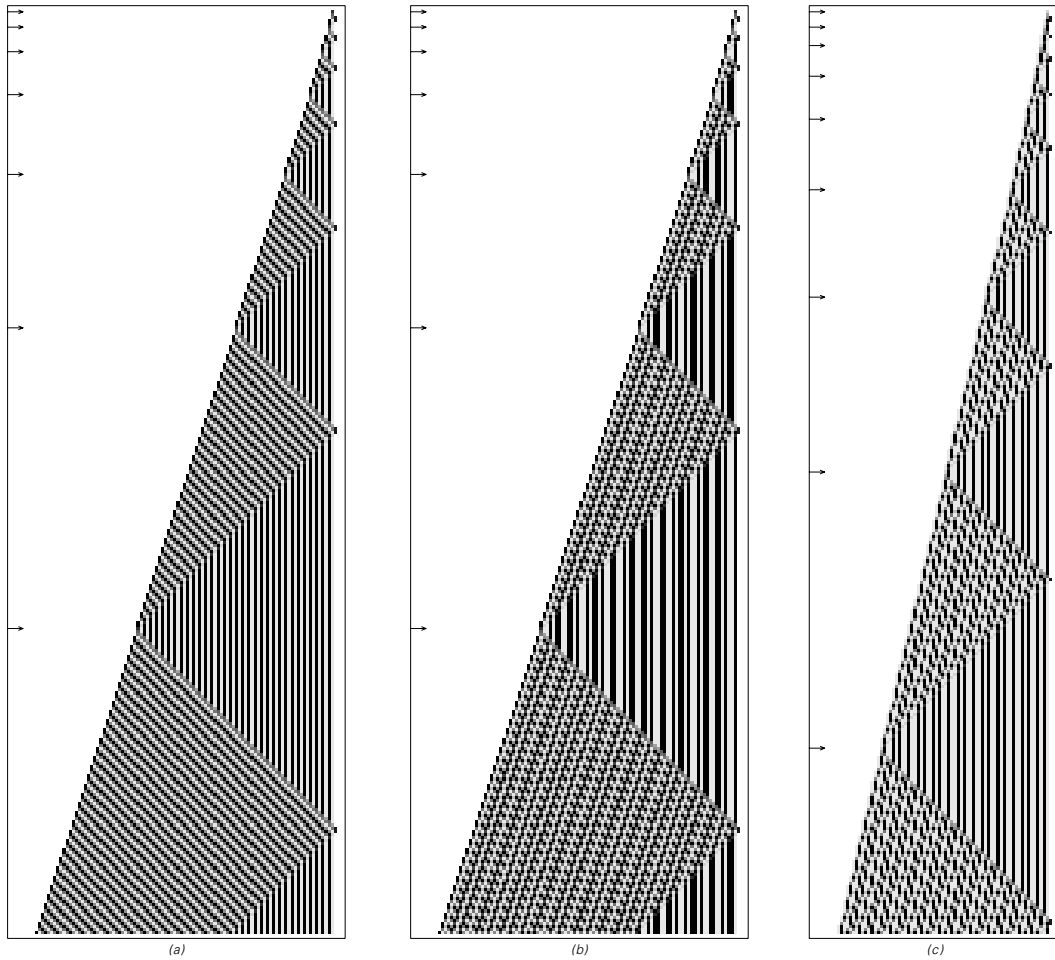


The pictures on the facing page demonstrate that in fact these can also be emulated by cellular automata. But while one can emulate each step in the evolution of a mobile automaton or a Turing machine with a single step of cellular automaton evolution, this is no longer in general true for substitution systems.

That this must ultimately be the case one can see from the fact that the total number of elements in a substitution system can be multiplied by a factor from one step to the next, while in a cellular automaton the size of a pattern can only ever increase by a fixed amount at each step. And what this means is that it can take progressively larger numbers of cellular automaton steps to reproduce each successive step in the evolution of the substitution system—as illustrated in the pictures on the facing page.

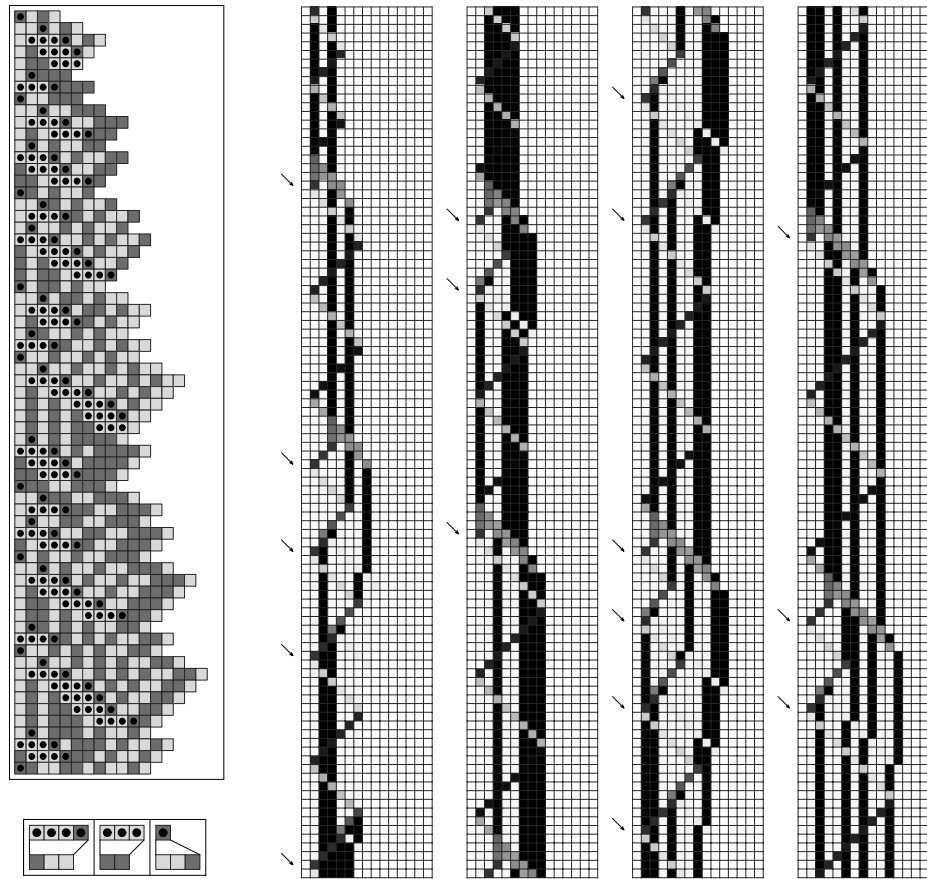
The same kind of problem occurs in sequential substitution systems—as well as in tag systems. But once again, as the pictures on page 660 demonstrate, it is still perfectly possible to emulate systems like these using cellular automata.

But just how broad is the set of systems that cellular automata can ultimately emulate? All the examples of systems that I have shown so far can at some level be thought of as involving sequences of elements that are fairly directly analogous to the cells in a cellular automaton.



Examples of cellular automata that emulate substitution systems. The successive steps in the evolution of each substitution system are obtained at the points indicated by arrows. Note that the sequences of elements generated by the cellular automata are aligned at the right, while in the pictures of the substitution systems shown they are aligned at the left. The rules for the three cellular automata involve only nearest neighbors, and allow 12 possible colors for each cell.

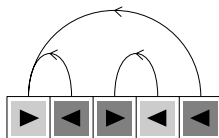
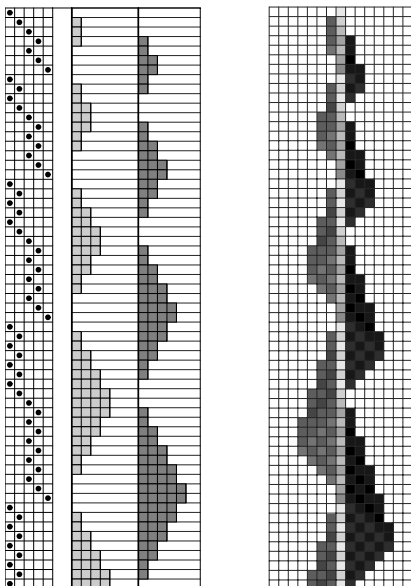




A cellular automaton set up to emulate a sequential substitution system. The cellular automaton involves 28 colors and nearest-neighbor rules. The strings produced by the sequential substitution system appear on successive diagonal stripes indicated by arrows in the evolution of the cellular automaton on the right.

But one example where there is no such direct analogy is a register machine. And at the outset one might not imagine that such a system could ever readily be emulated by a cellular automaton.

But in fact it turns out to be fairly straightforward to do so, as illustrated at the top of the facing page. The basic idea is to have the cellular automaton produce a pattern that expands and contracts on each side in a way that corresponds to the incrementing and decrementing of the sizes of numbers in the first and second registers of

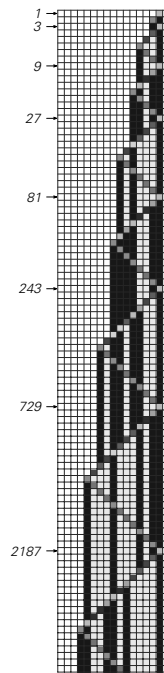


An example of a register machine being emulated by a cellular automaton. The cellular automaton has 12 possible colors for each cell. Of these, 5 are used by the center cell to represent the point that has been reached in the register machine program. The other 7 are used to implement signals that propagate out to the left and right to do the analog of incrementing and decrementing each register.

the register machine. In the center of the cellular automaton is then a cell whose possible colors correspond to possible points in the program for the register machine. And as the cell makes transitions from one color to another, it effectively emits signals that move to the left or right modifying the pattern in the cellular automaton in a way that follows each instruction in the register machine program.

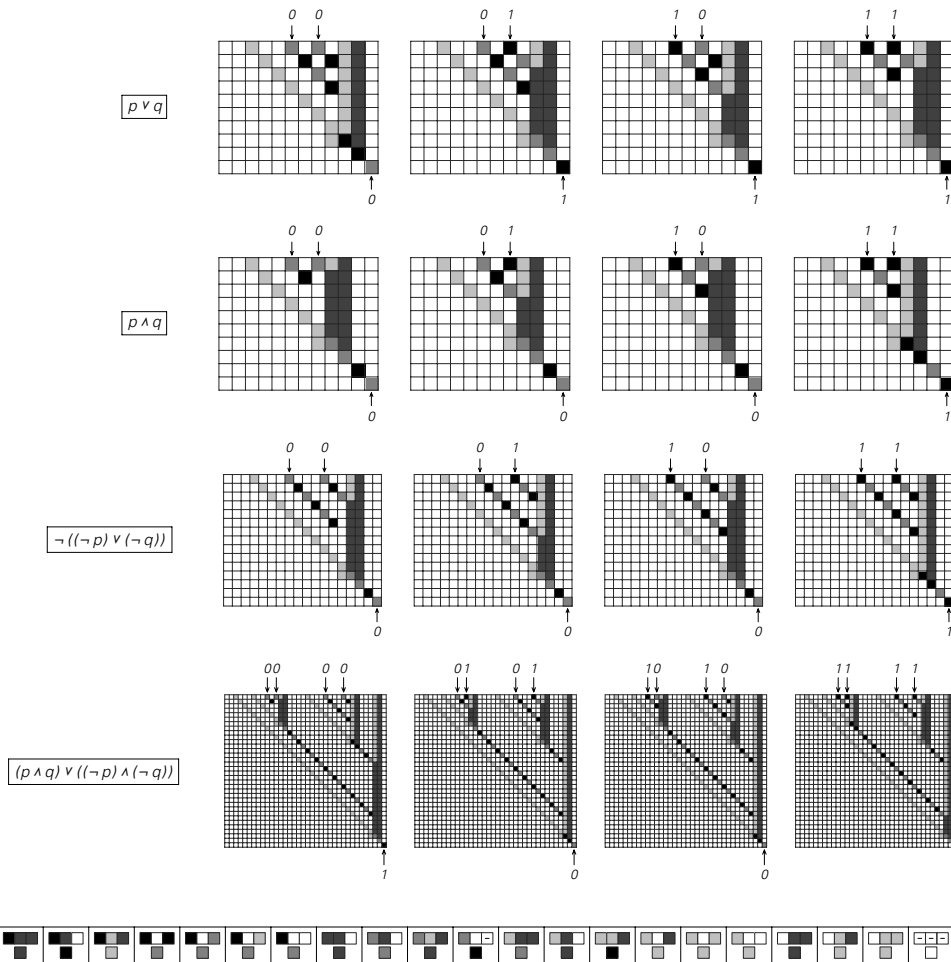
So what about systems based on numbers? Can these also be emulated by cellular automata? As one example the picture on the right shows how a cellular automaton can be set up to perform repeated multiplication by 3 of numbers in base 2. And the only real difficulty in this case is that carries generated in the process of multiplication may need to be propagated from one end of the number to the other.

So what about practical computers? Can these also be emulated by cellular automata? From the examples just discussed of register machines and systems based on numbers, we already know that cellular automata can emulate some of the low-level operations typically found in computers. And the pictures on the next two pages show how cellular automata can also be made to emulate two other important aspects of practical computers.

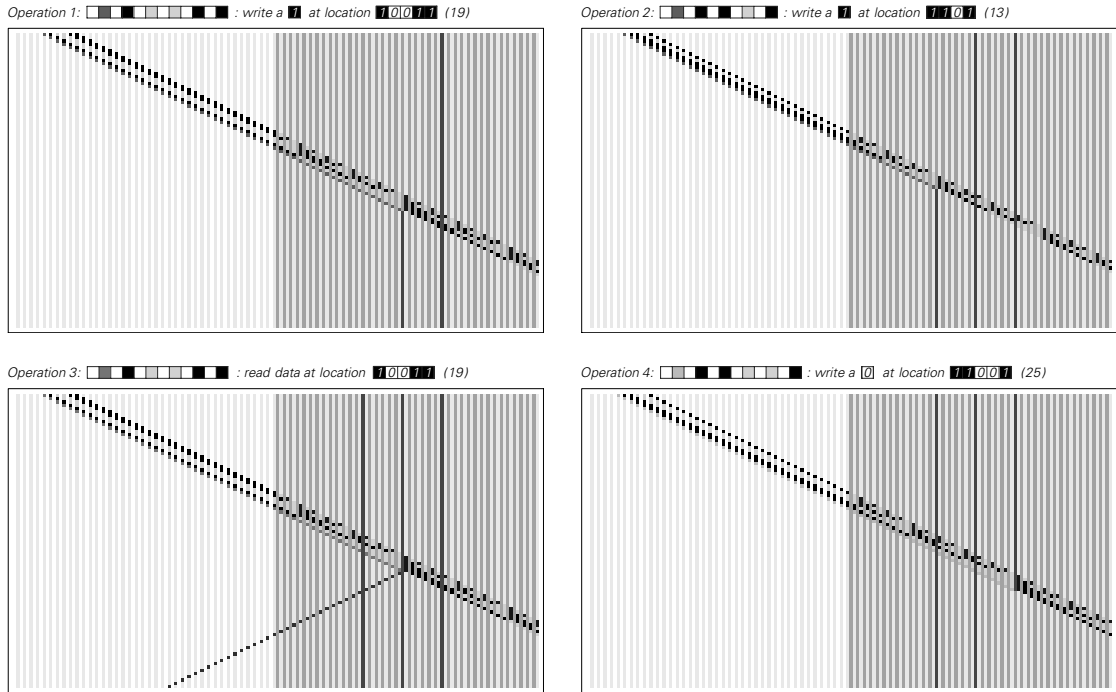


Repeated multiplication by 3 in base 2 being performed by a cellular automaton with 11 colors.

The pictures below show how a cellular automaton can evaluate any logic expression that is given in a certain form. And the picture on the facing page then shows how a cellular automaton can retrieve data from a numbered location in what is effectively a random-access memory.



A cellular automaton which emulates basic logic circuits. The underlying rules for the cellular automaton are exactly the same in each case, and involve nearest neighbors and five possible colors for each cell. But the initial condition can represent a logic expression that involves any number of variables together with the operations of AND, OR and NOT. In the examples above, two variables,  $p$  and  $q$ , are used, and in each case the behavior obtained with all four possible combinations of values for  $p$  and  $q$  are shown.



A cellular automaton set up to emulate random-access memory in a computer. The memory is on the right, and can be of any size. Instructions come in from the left, with memory locations specified by addresses consisting of binary digits.

The details for any particular case are quite complicated, but in the end it turns out that it is in principle possible to construct a cellular automaton that emulates a practical computer in its entirety.

And as a result, one can conclude that any of the very wide range of computations that can be performed by practical computers can also be done by cellular automata.

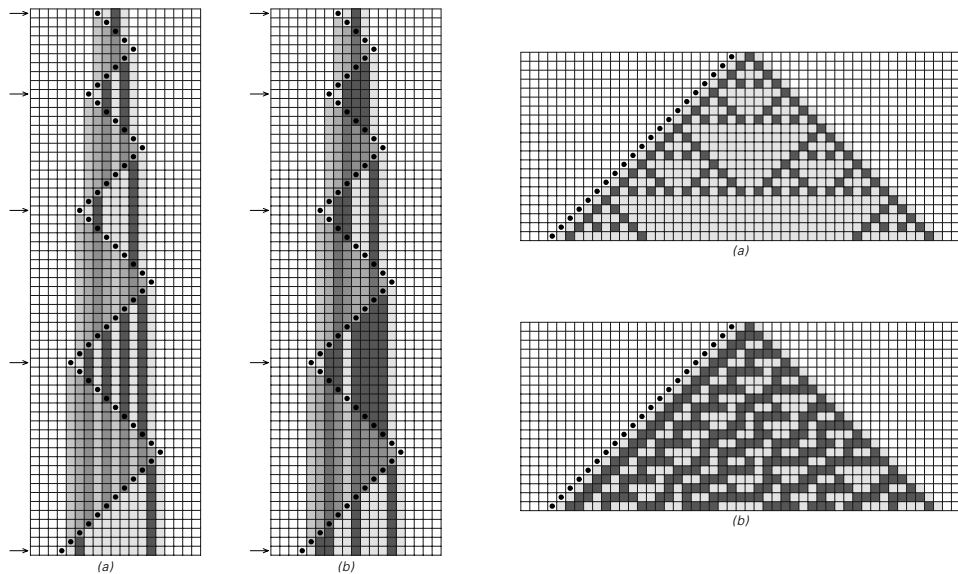
From the previous section we know that any cellular automaton can be emulated by a universal cellular automaton. But now we see that a universal cellular automaton is actually much more universal than we saw in the previous section. For not only can it emulate any cellular automaton: it can also emulate any of a wide range of other systems, including practical computers.

## Emulating Cellular Automata with Other Systems

In the previous section we discovered the rather remarkable fact that cellular automata can be set up to emulate an extremely wide range of other types of systems. But is this somehow a special feature of cellular automata, or do other systems also have similar capabilities?

In this section we will discover that in fact almost all of the systems that we considered in the previous section—and in Chapter 3—have the same capabilities. And indeed just as we showed that each of these various systems could be emulated by cellular automata, so now we will show that these systems can emulate cellular automata.

As a first example, the pictures below show how mobile automata can be set up to emulate cellular automata. The basic idea is to have the active cell in the mobile automaton sweep backwards and forwards, updating cells as it goes, in such a way that after each complete sweep it has effectively performed one step of cellular automaton evolution.



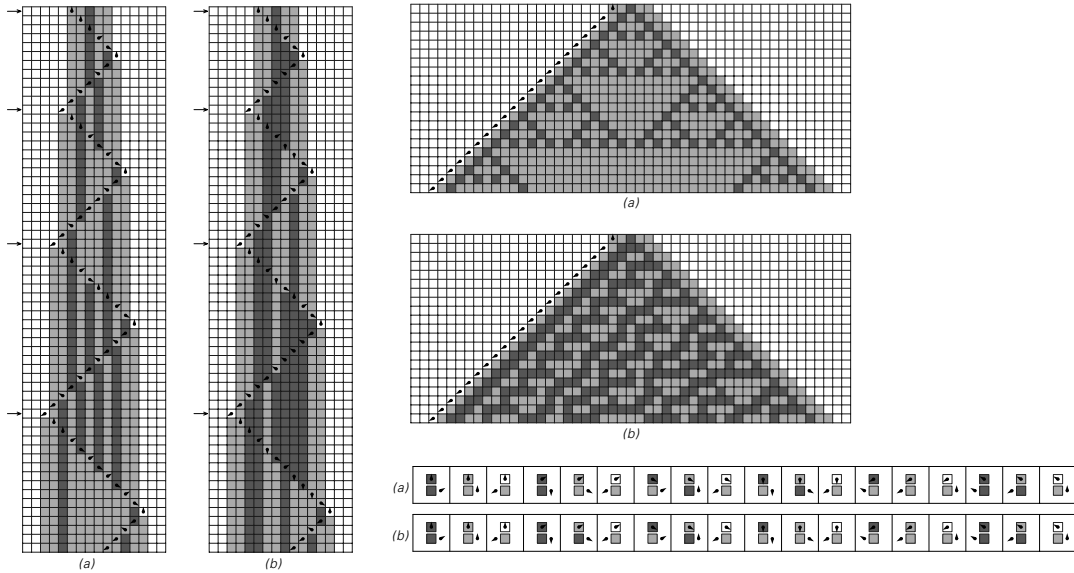
Examples of mobile automata emulating cellular automata. In case (a) the rules for the mobile automaton are set up to emulate the rule 90 elementary cellular automaton; in case (b) they are set up to emulate rule 30. The pictures on the right are obtained by keeping only the steps indicated by arrows on the left, corresponding to times when the active cell in the mobile automaton is further to the left than it has ever been before. The mobile automata used here involve 7 possible colors for each cell.

The specific pictures at the bottom of the facing page are for elementary cellular automata with two possible colors for each cell and nearest-neighbor rules. But the same basic idea can be used for cellular automata with rules of any kind. And this implies that it is possible to construct for example a mobile automaton which emulates the universal cellular automata that we discussed a couple of sections ago.

Such a mobile automaton must then itself be universal, since the universal cellular automaton that it emulates can in turn emulate a wide range of other systems, including all possible mobile automata.

A similar scheme to the one for mobile automata can also be used for Turing machines, as illustrated in the pictures below. And once again, by emulating the universal cellular automaton, it is then possible to construct a universal Turing machine.

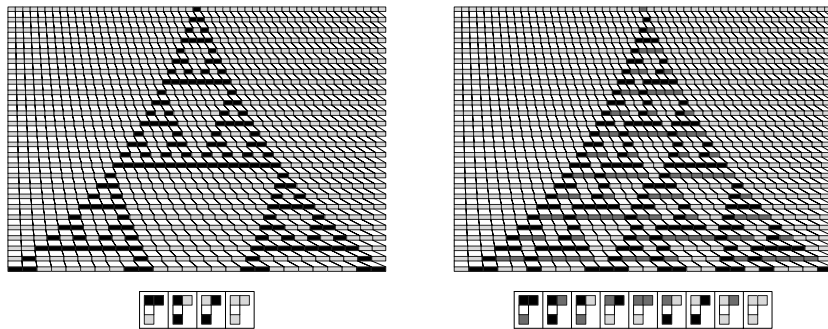
But as it turns out, a universal Turing machine was already constructed in 1936, using somewhat different methods. And in fact that universal Turing machine provided what was historically the very first clear example of universality seen in any system.



Examples of Turing machines that emulate cellular automata with rules 90 and 30. The pictures on the right are obtained by keeping only the steps indicated by arrows on the left. The Turing machines have 6 states and 3 possible colors for each cell.

Continuing with the types of systems from the previous section, we come next to substitution systems. And here, for once, we find that at least at first we cannot in general emulate cellular automata. For as we discussed on page 83, neighbor-independent substitution systems can generate only patterns that are either repetitive or nested—so they can never yield the more complicated patterns that are, for example, needed to emulate rule 30.

But if one generalizes to neighbor-dependent substitution systems then it immediately becomes very straightforward to emulate cellular automata, as in the pictures below.

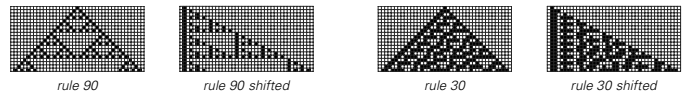
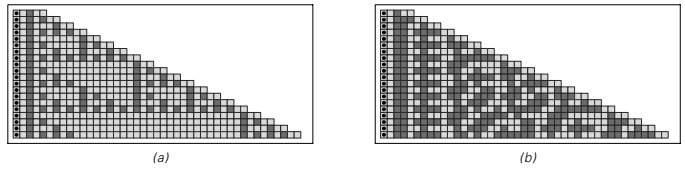
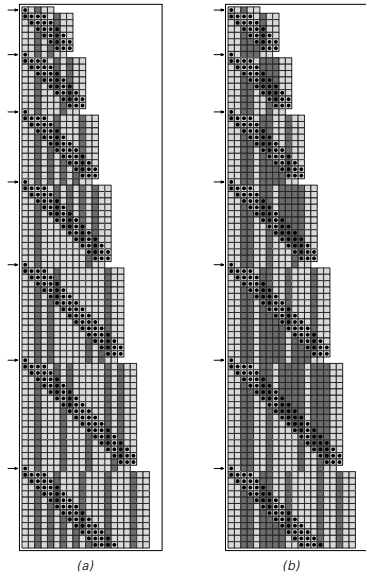


Neighbor-dependent substitution systems that emulate cellular automata with rules 90 and 30. The systems shown are simple examples of neighbor-dependent substitution systems with highly uniform rules always yielding just one cell and corresponding quite directly to cellular automata.

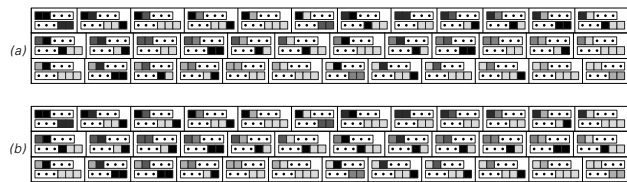
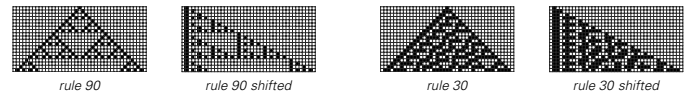
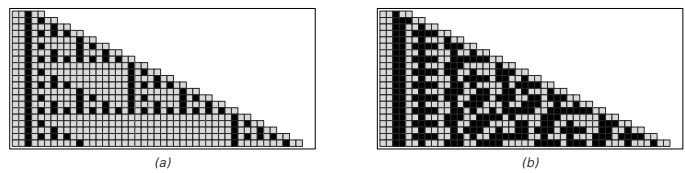
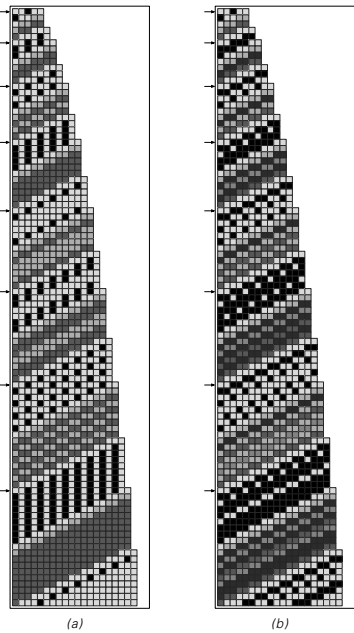
What about sequential substitution systems? Here again it turns out to be fairly easy to emulate cellular automata—as the pictures at the top of the facing page demonstrate.

Perhaps more surprisingly, the same is also true for ordinary tag systems. And even though such systems operate in an extremely simple underlying way, the pictures at the bottom of the facing page demonstrate that they can still quite easily emulate cellular automata.

What about symbolic systems? The structure of these systems is certainly vastly different from cellular automata. But once again—as the picture at the top of page 668 shows—it is quite easy to get these systems to emulate cellular automata.

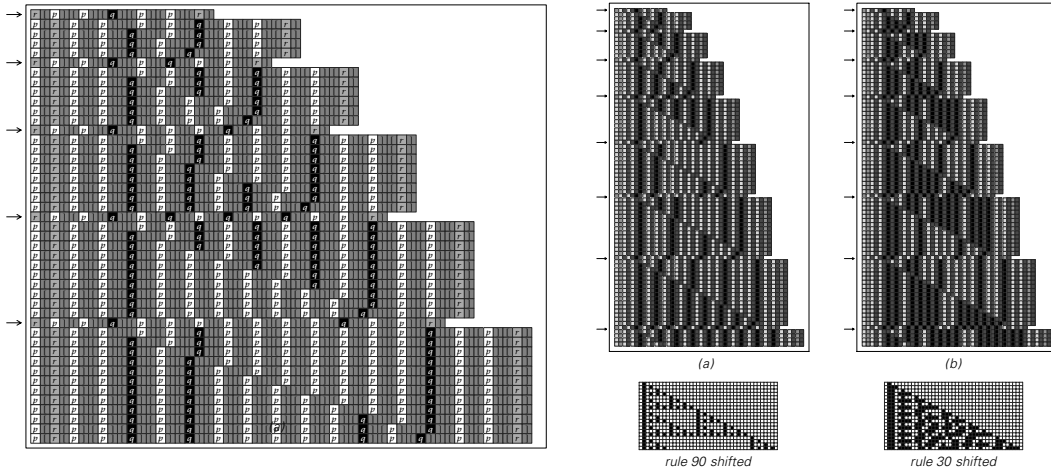


Sequential substitution systems that emulate cellular automata with rules 90 and 30. The pictures at the top above are obtained by keeping only the steps indicated by arrows on the left. The sequential substitution systems involve elements with 3 possible colors.



Tag systems that emulate the rule 90 and rule 30 cellular automata. The pictures at the top above are obtained by keeping only the steps indicated by arrows on the left. Both tag systems involve 6 colors.





(a)  $p[x\_][p][p] \rightarrow p[x(p)][p][p], p[x\_][p][p][q] \rightarrow p[x(q)][p][q], p[x\_][p][q][p] \rightarrow p[x(p)][q][p], p[x\_][p][q][q] \rightarrow p[x(q)][q][q], p[x\_][q][p][p] \rightarrow p[x(q)][p][p], p[x\_][q][p][q] \rightarrow p[x(p)][p][q], p[x\_][q][q][p] \rightarrow p[x(q)][q][p], p[x\_][q][q][q] \rightarrow p[x(p)][q][q], r[x\_] \rightarrow p[r(p)][p][x], p[x\_][p][p][r] \rightarrow x[p][p][r]$

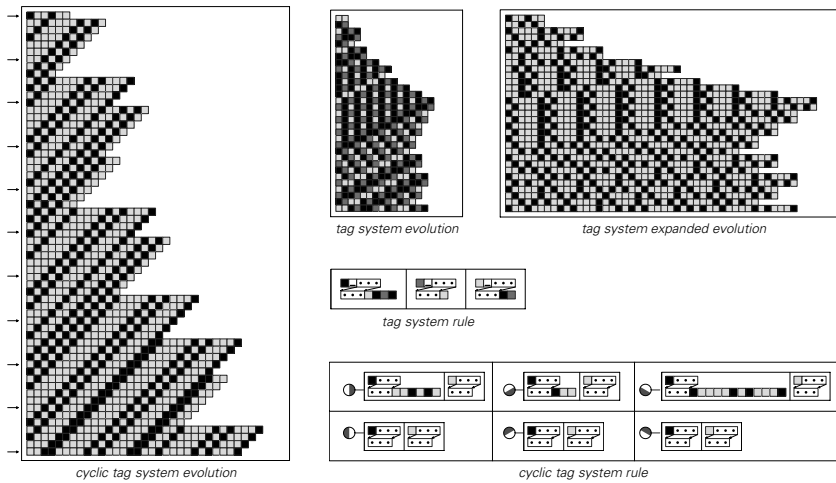
(b)  $p[x\_][p][p][p] \rightarrow p[x(p)][p][p], p[x\_][p][p][q] \rightarrow p[x(q)][p][q], p[x\_][p][q][p] \rightarrow p[x(q)][q][p], p[x\_][p][q][q] \rightarrow p[x(q)][q][q], p[x\_][q][p][p] \rightarrow p[x(q)][p][p], p[x\_][q][p][q] \rightarrow p[x(p)][p][q], p[x\_][q][q][p] \rightarrow p[x(p)][q][p], p[x\_][q][q][q] \rightarrow p[x(p)][q][q], r[x\_] \rightarrow p[r(p)][p][x], p[x\_][p][p][r] \rightarrow x[p][p][r]$

Symbolic systems set up to emulate cellular automata that have rules 90 and 30. Unlike the examples of symbolic systems in Chapter 3, which involve only one symbol, these symbolic systems involve three symbols,  $p$ ,  $q$  and  $r$ .

And as soon as one knows that any particular type of system is capable of emulating any cellular automaton, it immediately follows that there must be examples of that type of system that are universal.

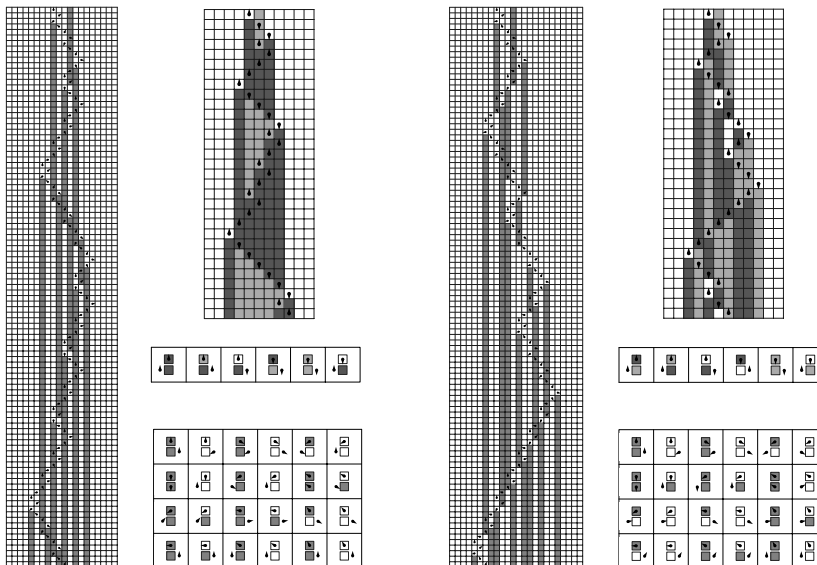
So what about the other types of systems that we considered in Chapter 3? One that we have not yet discussed here are cyclic tag systems. And as it turns out, we will end up using just such systems later in this chapter as part of establishing a dramatic example of universality.

But to demonstrate that cyclic tag systems can manage to emulate cellular automata is not quite as straightforward as to do this for the various kinds of systems we have discussed so far. And indeed we will end up doing it in several stages. The first stage, illustrated in the picture at the top of the facing page, is to get a cyclic tag system to emulate an ordinary tag system with the property that its rules depend only on the very first element that appears at each step.

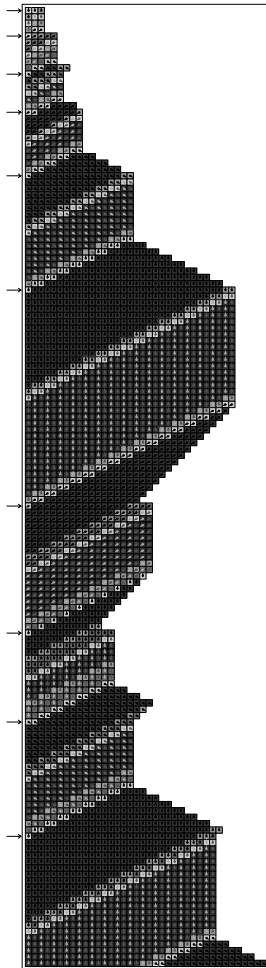


A cyclic tag system emulating a tag system that depends only on the first element at each step. In the expanded tag system evolution, successive colors of elements are encoded by having a black cell at successive positions inside a fixed block of white cells.

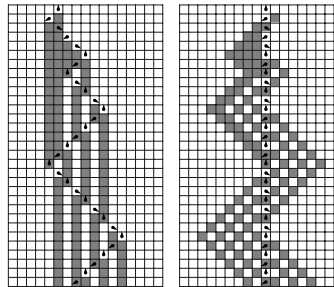
And having done this, the next stage is to get such a tag system to emulate a Turing machine. The pictures on the next page illustrate how this can be done. But at least with the particular construction shown, the resulting Turing machine can only have cells with two possible colors. The pictures below demonstrate, however, that such a Turing



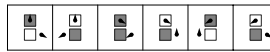
Turing machines with two colors emulating ones with more colors.



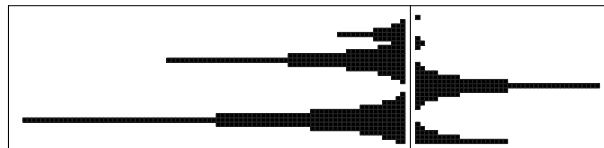
tag system evolution (150 steps)



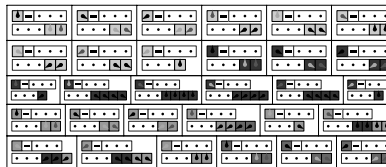
Turing machine evolution



Turing machine rule



Turing machine left and right numbers



tag system rule

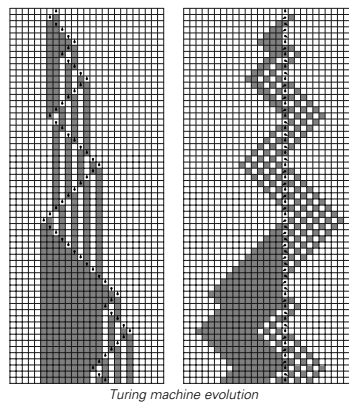
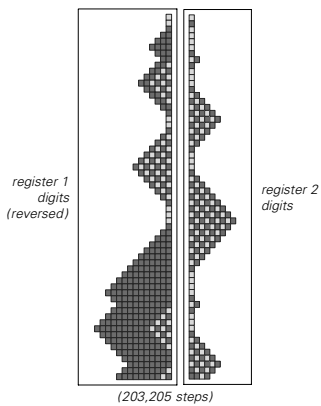


tag system compressed evolution (1500 steps)

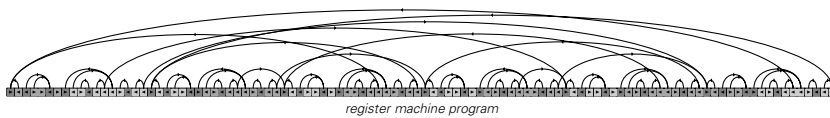
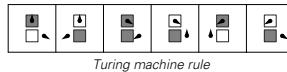
Emulating a Turing machine with a tag system that depends only on the first element at each step. The configuration of cells on each side of the head in the Turing machine is treated as a base 2 number. At the steps indicated by arrows the tag system yields sequences of dark cells with lengths that correspond to each of these numbers.

machine can readily be made to emulate a Turing machine with any number of colors. And through the construction of page 665 this then finally shows that a cyclic tag system can successfully emulate any cellular automaton—and can thus be universal.

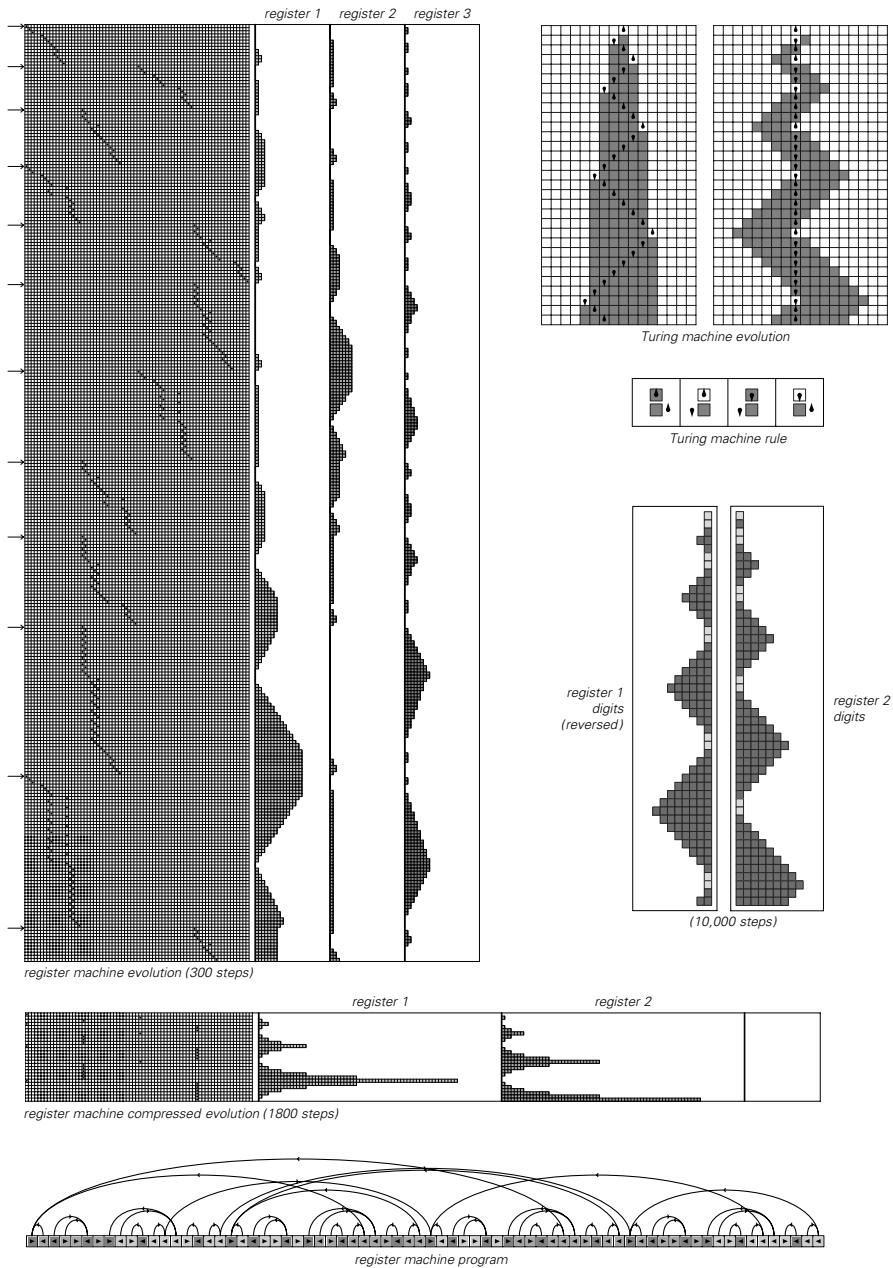
This leaves only one remaining type of system from Chapter 3: register machines. And although it is again slightly complicated, the pictures on the next page—and below—show how even these systems can be made to emulate Turing machines and thus cellular automata.



A register machine emulating a slightly more complicated Turing machine than on the next page.



So what about systems based on numbers, like those we discussed in Chapter 4? As an example, one can consider a generalization of the arithmetic systems discussed on page 122—in which one has a whole number  $n$ , and at each step one finds the remainder after dividing by a constant, and based on the value of this remainder one then applies some specified arithmetic operation to  $n$ .

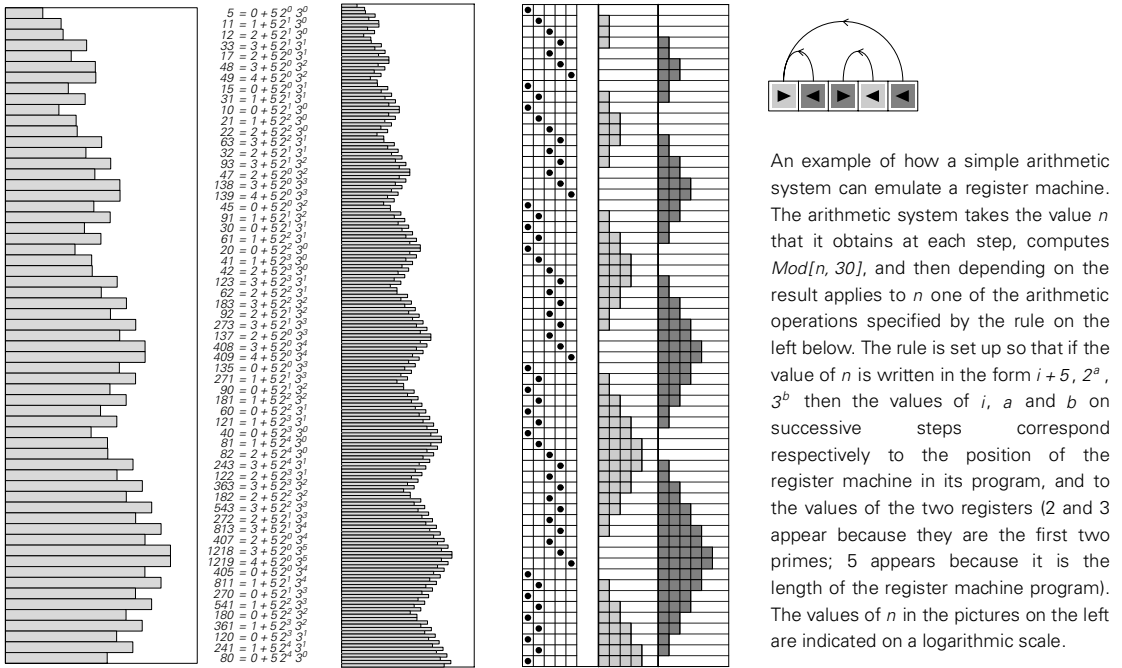


An example of a register machine set up to emulate a Turing machine. The Turing machine used here has two states for the head; the register machine program has 72 instructions and uses three registers. The register machine compressed evolution keeps only steps corresponding to every other time the third register gets incremented from zero.

The picture below shows that such a system can be set up to emulate a register machine. And from the fact that register machines are universal it follows that so too are such arithmetic systems.

And indeed the fact that it is possible to set up a universal system using essentially just the operations of ordinary arithmetic is closely related to the proof of Gödel’s Theorem discussed on page 784.

But from what we have learned in this chapter, it no longer seems surprising that arithmetic should be capable of achieving universality. Indeed, considering all the kinds of systems that we have found can exhibit universality, it would have been quite peculiar if arithmetic had somehow not been able to support it.



$2n+1$	$(n-1)/3$	$3(n-1)$	$(n+1)/2$	$(n-4)/3$	$2n+1$	$n+1$	$3(n-1)$	$n+1$	$n+1$
0	1	2	3	4	5	6	7	8	9
$2n+1$	$n+1$	$3(n-1)$	$(n+1)/2$	$n+1$	$2n+1$	$(n-1)/3$	$3(n-1)$	$n+1$	$(n-4)/3$
10	11	12	13	14	15	16	17	18	19
$2n+1$	$n+1$	$3(n-1)$	$(n+1)/2$	$n+1$	$2n+1$	$n+1$	$3(n-1)$	$n+1$	$n+1$
20	21	22	23	24	25	26	27	28	29

## Implications of Universality

When we first discussed cellular automata, Turing machines, substitution systems, register machines and so on in Chapter 3, each of these kinds of systems seemed rather different. But already in Chapter 3 we discovered that at the level of overall behavior, all of them had certain features in common. And now, finally, by thinking in terms of computation, we can begin to see why this might be the case.

The main point, as the previous two sections have demonstrated, is that essentially all of these various kinds of systems—despite their great differences in underlying structure—can ultimately be made to emulate each other.

This is a very remarkable result, and one which will turn out to be crucial to the new kind of science that I develop in this book.

In a sense its most important consequence is that it implies that from a computational point of view a very wide variety of systems, with very different underlying structures, are at some level fundamentally equivalent. For one might have thought that every different kind of system that we discussed for example in Chapter 3 would be able to perform completely different kinds of computations.

But what we have discovered here is that this is not the case. And instead it has turned out that essentially every single one of these systems is ultimately capable of exactly the same kinds of computations.

And among other things, this means that it really does make sense to discuss the notion of computation in purely abstract terms, without referring to any specific type of system. For we now know that it ultimately does not matter what kind of system we use: in the end essentially any kind of system can be programmed to perform the same computations. And so if we study computation at an abstract level, we can expect that the results we get will apply to a very wide range of actual systems.

But it should be emphasized that among systems of any particular type—say cellular automata—not all possible underlying rules are capable of supporting the same kinds of computations.

Indeed, as we saw at the beginning of this chapter, some cellular automata can perform only very simple computations, always yielding

for example purely repetitive patterns. But the crucial point is that as one looks at cellular automata with progressively greater computational capabilities, one will eventually pass the threshold of universality. And once past this threshold, the set of computations that can be performed will always be exactly the same.

One might assume that by using more and more sophisticated underlying rules, one would always be able to construct systems with ever greater computational capabilities. But the phenomenon of universality implies that this is not the case, and that as soon as one has passed the threshold of universality, nothing more can in a sense ever be gained.

In fact, once one has a system that is universal, its properties are remarkably independent of the details of its construction. For at least as far as the computations that it can perform are concerned, it does not matter how sophisticated the underlying rules for the system are, or even whether the system is a cellular automaton, a Turing machine, or something else. And as we shall see, this rather remarkable fact forms the basis for explaining many of the observations we made in Chapter 3, and indeed for developing much of the conceptual framework that is needed for the new kind of science in this book.

### **The Rule 110 Cellular Automaton**

In previous sections I have shown that a wide variety of different kinds of systems can in principle be made to exhibit the phenomenon of universality. But how complicated do the underlying rules need to be in a specific case in order actually to achieve universality?

The universal cellular automaton that I described earlier in this chapter had rather complicated underlying rules, involving 19 possible colors for each cell, depending on next-nearest as well as nearest neighbors. But this cellular automaton was specifically constructed so as to make its operation easy to understand. And by not imposing this constraint, one might expect that one would be able to find universal cellular automata that have at least somewhat simpler underlying rules.

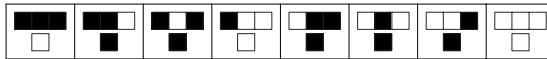
Fairly straightforward modifications to the universal cellular automaton shown earlier in this chapter allow one to reduce the number



of colors from 19 to 17. And in fact in the early 1970s, it was already known that cellular automata with 18 colors and nearest-neighbor rules could be universal. In the late 1980s—with some ingenuity—examples of universal cellular automata with 7 colors were also constructed.

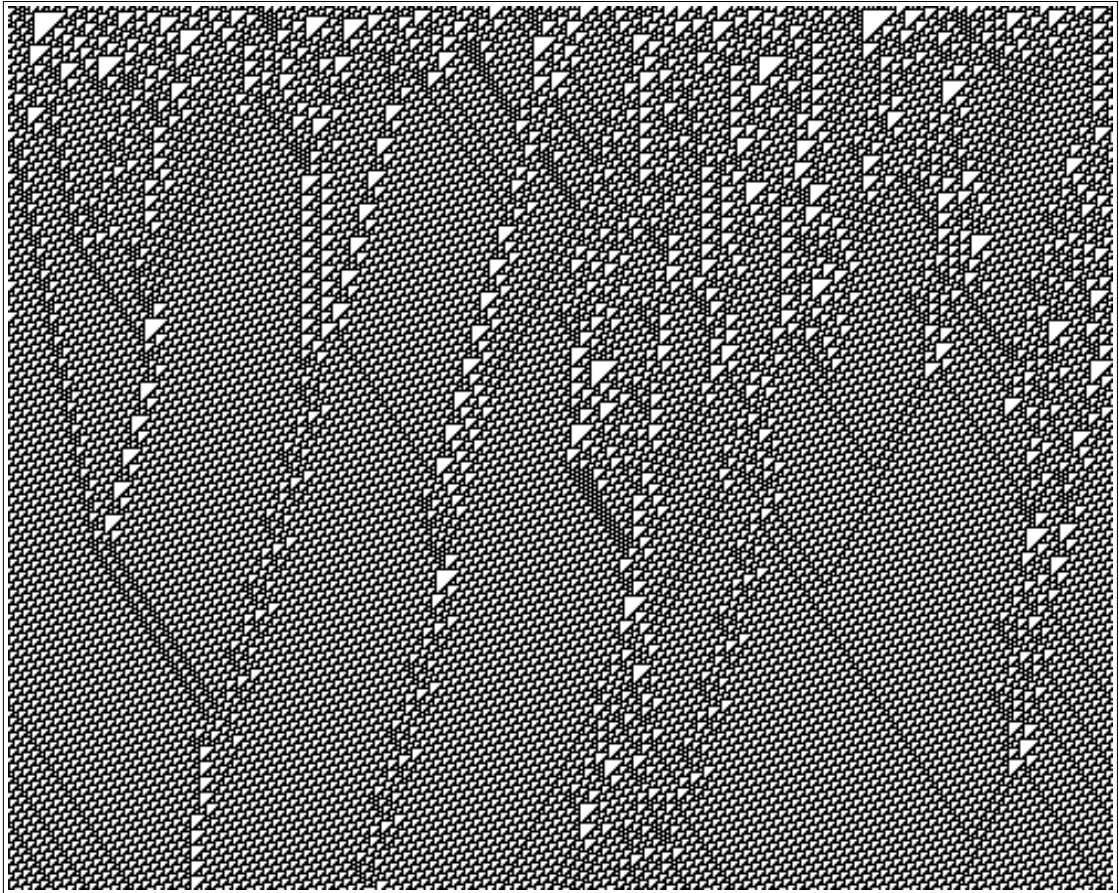
But such rules still involve 343 distinct cases and are by almost any measure very complicated. And certainly rules this complicated could not reasonably be expected to be common in the types of systems that we typically see in nature. Yet from my experiments on cellular automata in the early 1980s I became convinced that very much simpler rules should also show universality. And by the mid-1980s I began to suspect that even among the very simplest possible rules—with just two colors and nearest neighbors—there might be examples of universality.

The leading candidate was what I called rule 110—a cellular automaton that we have in fact discussed several times before in this book. Like any of the 256 so-called elementary rules, rule 110 can be specified as below by giving the outcome for each of the eight possible combinations of colors of a cell and its nearest neighbors.



The underlying rules for the rule 110 cellular automaton discussed in this section. As elsewhere in the book, each of the eight cases shows what the new color of a cell should be based on its own previous color, and on the previous colors of its neighbors. Despite the extreme simplicity of its underlying rules, what this section will demonstrate is that the rule 110 cellular automaton is in fact universal, and is thus in a sense capable of arbitrarily complex behavior. If the values of the cells in each block are labelled  $p$ ,  $q$  and  $r$ , then rule 110 can be written as  $\text{Mod}[(1+p)qr + q + r, 2]$  or  $\neg(p \wedge q \wedge r) \wedge (q \vee r)$ .

Looking just at this very simple specification, however, it seems at first quite absurd to think that rule 110 might be universal. But as soon as one looks at a picture of how rule 110 actually behaves, the idea that it could be universal starts to seem much less absurd. For despite the simplicity of its underlying rules, rule 110 supports a whole variety of localized structures—that move around and interact in many complicated ways. And from pictures like the one on the facing page, it begins to seem not unreasonable that perhaps these localized structures could be arranged so as to perform meaningful computations.



A typical example of the behavior of rule 110 with random initial conditions. From looking at pictures like these one can begin to imagine that it could be possible to arrange localized structures in rule 110 so as to be able to perform meaningful computations. Note that page 292 already showed many of the types of localized structures that can occur in rule 110.

In the universal cellular automaton that we discussed earlier in this chapter, each of the various kinds of components involved in its operation had properties that were explicitly built into the underlying rules. Indeed, in most cases each different type of component was simply represented by a different color of cell. But in rule 110 there are only two possible colors for each cell. So one may wonder how one could ever expect to represent different kinds of components.

The crucial idea is to build up components from combinations of localized structures that the rule in a sense already produces. And if this works, then it is in effect a very economical solution. For it potentially allows one to get a large number of different kinds of components without ever needing to increase the complexity of the underlying rules at all.

But the problem with this approach is that it is typically very difficult to see how the various structures that happen to occur in a particular cellular automaton can be assembled into useful components.

And indeed in the case of rule 110 it took several years of work to develop the necessary ideas and tools. But finally it has turned out to be possible to show that the rule 110 cellular automaton is in fact universal.

It is truly remarkable that a system with such simple underlying rules should be able to perform what are in effect computations of arbitrary sophistication, but that is what its universality implies.

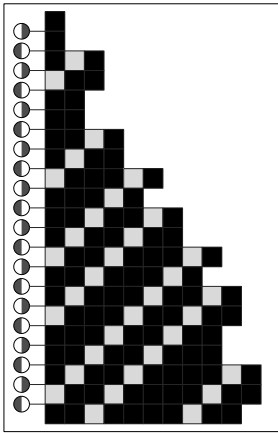
So how then does the proof of universality proceed?

The basic idea is to show that rule 110 can emulate any possible system in some class of systems where there is already known to be universality. And it turns out that a convenient such class of systems are the cyclic tag systems that we introduced on page 95.

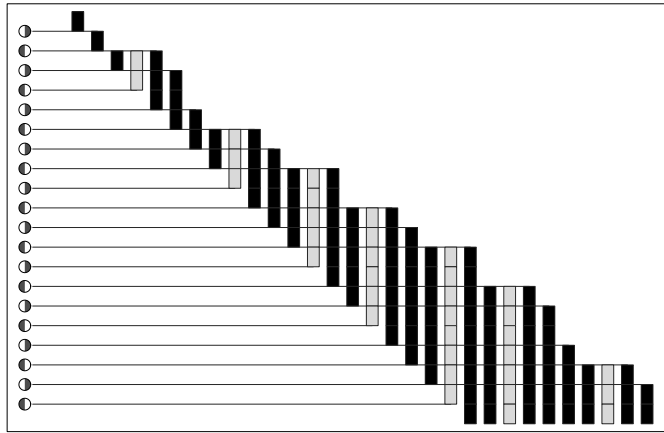
Earlier in this chapter we saw that it is possible to construct a cyclic tag system that can emulate any given Turing machine. And since we know that at least some Turing machines are universal, this fact then establishes that universal cyclic tag systems are possible.

So if we can succeed in demonstrating that rule 110 can emulate any cyclic tag system, then we will have managed to prove that rule 110 is itself universal. The sequence of pictures on the facing page shows the beginnings of what is needed. The basic idea is to start from the usual representation of a cyclic tag system, and then progressively to change this representation so as to get closer and closer to what can actually be emulated directly by rule 110.

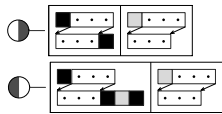
Picture (a) shows an example of the evolution of a cyclic tag system in the standard representation from pages 95 and 96. Picture (b) then shows another version of this same evolution, but now rearranged so that each element stays in the same position, rather than always shifting to the left at each step.



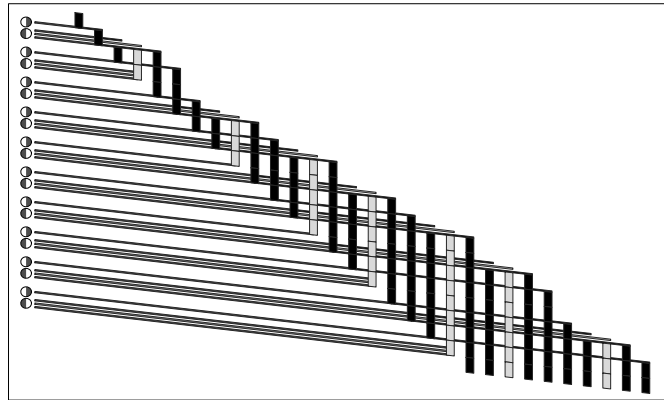
(a)



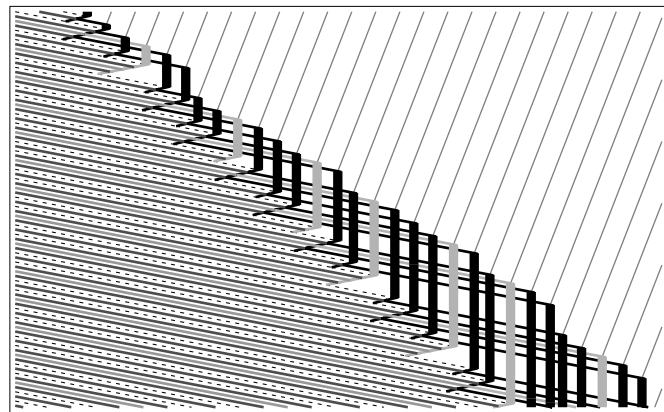
(b)



summary: 



(c)



(d)

Four views of a cyclic tag system with rules as shown above, drawn so as to be progressively closer to what can be emulated directly in rule 110. Picture (a) shows the cyclic tag system in the same form as on pages 95 and 96. Picture (b) shows the system with sequences on successive steps rearranged so that they do not shift to the left when the first element is removed. Picture (c) is a skewed version of (b) in which the way information is used from the underlying rules at each step is explicitly indicated. Picture (d) shows a more definite mechanism for the evolution of the system in which different lines effectively indicate the motions of different pieces of information.

A cyclic tag system in general operates by removing the first element from the sequence that exists at each step, and then adding a new block of elements to the end of the sequence if this element is black. A crucial feature of cyclic tag systems is that the choice of what block of elements can be added does not depend in any way on the form of the sequence. So, for example, on the previous page, there are just two possibilities, and these possibilities alternate on successive steps.

Pictures (a) and (b) on the previous page illustrate the consequences of applying the rules for a cyclic tag system, but in a sense give no indication of an explicit mechanism by which these rules might be applied. In picture (c), however, we see the beginnings of such a mechanism.

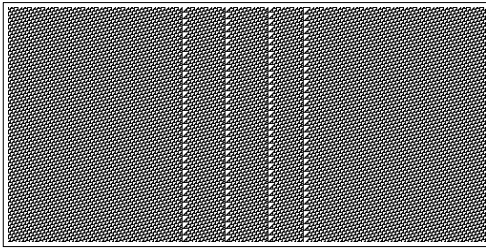
The basic idea is that at each step in the evolution of the system, there is a stripe that comes in from the left carrying information about the block that can be added at that step. Then when the stripe hits the first element in the sequence that exists at that step, it is allowed to pass only if the element is black. And once past, the stripe continues to the right, finally adding the block it represents to the end of the sequence.

But while picture (c) shows the effects of various lines carrying information around the system, it gives no indication of why the lines should behave in the way they do. Picture (d), however, shows a much more explicit mechanism. The collections of lines coming in from the left represent the blocks that can be added at successive steps. The beginning of each block is indicated by a dashed line, while the elements within the block are indicated by solid black and gray lines.

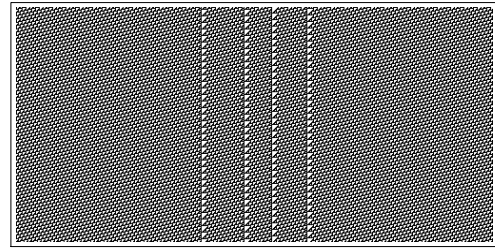
When a dashed line hits the first element in the sequence that exists at a particular step, it effectively bounces back in the form of a line propagating to the left that carries the color of the first element.

When this line is gray, it then absorbs all other lines coming from the left until the next dashed line arrives. But when the line is black, it lets lines coming from the left through. These lines then continue until they collide with gray lines coming from the right, at which point they generate a new element with the same color as their own.

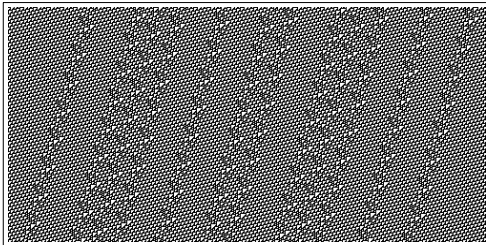
By looking at picture (d), one can begin to see how it might be possible for a cyclic tag system to be emulated by rule 110: the basic



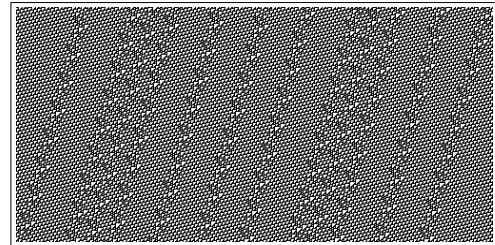
*a black element in the sequence*



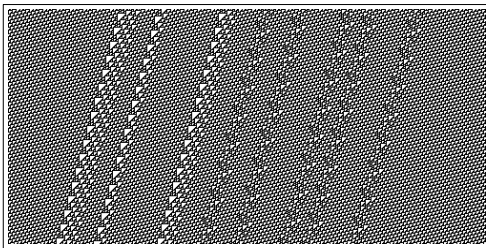
*a white element in the sequence*



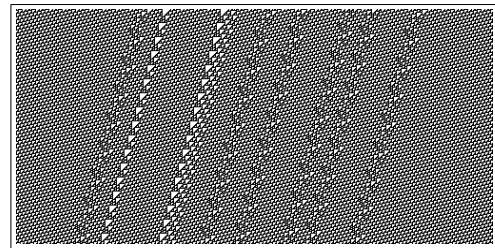
*a black element in a block*



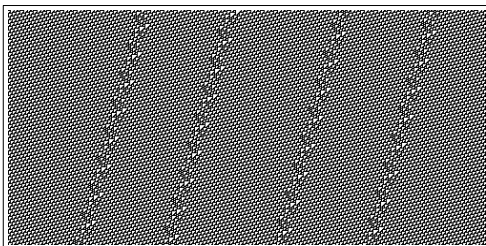
*a white element in a block*



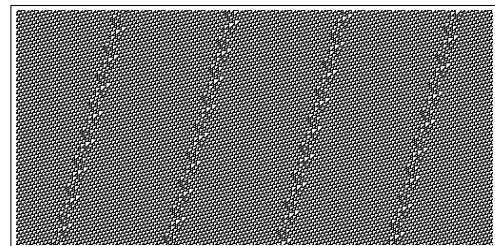
*the initial form of a separator between blocks*



*the later form of a separator between blocks*



*a black element ready to be added*



*a white element ready to be added*

Objects constructed from localized structures in rule 110, used for the emulation of cyclic tag systems. Each of the pictures shown is 500 cells wide. The objects in the top two pictures correspond to the thick vertical black and gray lines in picture (d) on page 679. The objects in the next two pictures correspond to the dark and light gray lines that come in from the left in picture (d). (Note that all the structures are left-right reversed in rule 110.) The third pair of pictures correspond to two versions of the dashed lines in picture (d). And the fourth pair of pictures correspond to right-going lines on the right-hand side of picture (d). All the localized structures involved in the pictures above were shown individually on page 292. Note that the spacings between structures are crucial in determining the objects they represent.

idea is to have each of the various kinds of lines in the picture be emulated by some collection of localized structures in rule 110.

But at the outset it is by no means clear that collections of localized structures can be found that will behave in appropriate ways.

With some effort, however, it turns out to be possible to find the necessary constructs, and indeed the previous page shows various objects formed from localized structures in rule 110 that can be used to emulate most of the types of lines in picture (d) on page 679.

The first two pictures show objects that correspond to the black and white elements indicated by thick vertical lines in picture (d). Both of these objects happen to consist of the same four localized structures, but the objects are distinguished by the spacings between these structures.

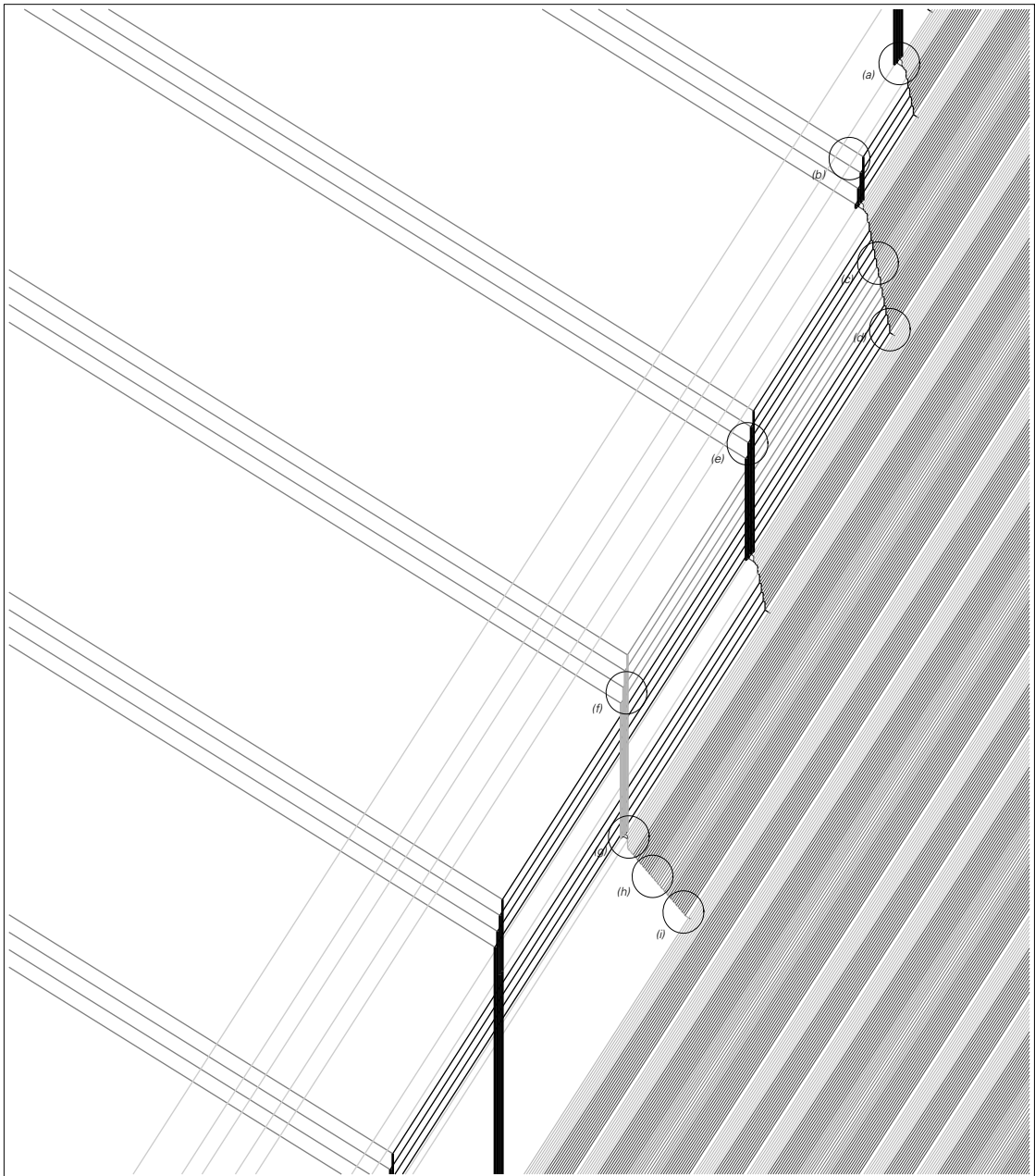
The second two pictures on the previous page use the same idea of different spacings between localized structures to represent the black and gray lines shown coming in from the left in picture (d) on page 679.

Note that because of the particular form of rule 110, the objects in the second two pictures on the previous page move to the left rather than to the right. And indeed in setting up a correspondence with rule 110, it is convenient to left-right reverse all pictures of cyclic tag systems. But using the various objects from the previous page, together with a few others, it is then possible to set up a complete emulation of a cyclic tag system using rule 110.

The diagram on the facing page shows schematically how this can be done. Every line in the diagram corresponds to a single localized structure in rule 110, and although the whole diagram cannot be drawn completely to scale, the collisions between lines correctly show all the basic interactions that occur between structures.

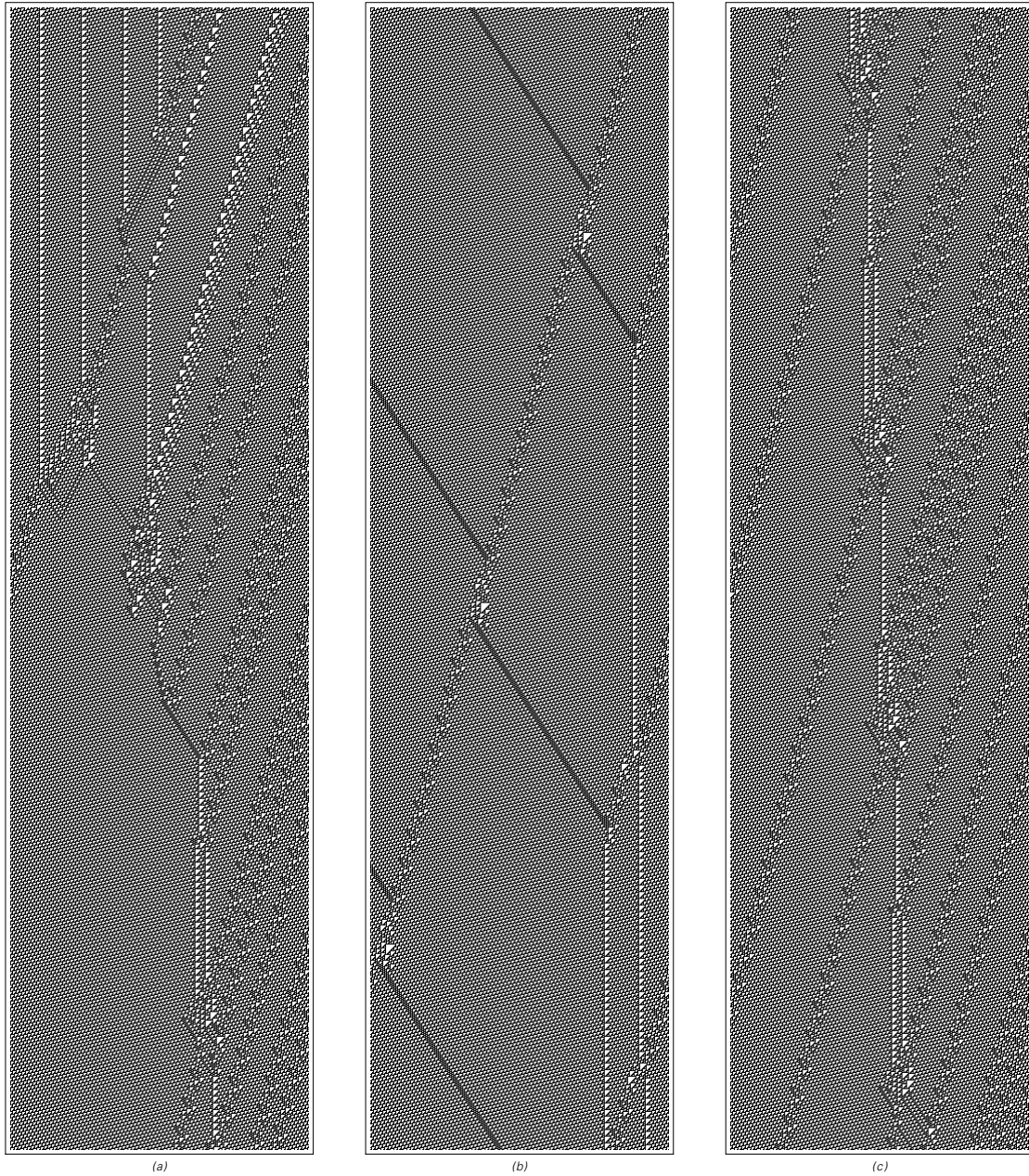
The next several pages then give details of what happens in each of the regions indicated by circles in the schematic diagram.

Region (a) shows a block separator—corresponding to a dashed line in picture (d) on page 679—hitting the single black element in the sequence that exists at the first step. Because the element hit is black, an object must be produced that allows information from the block at this step to pass through. Most of the activity in region (a) is concerned with producing such an object. But it turns out that as a side-effect two

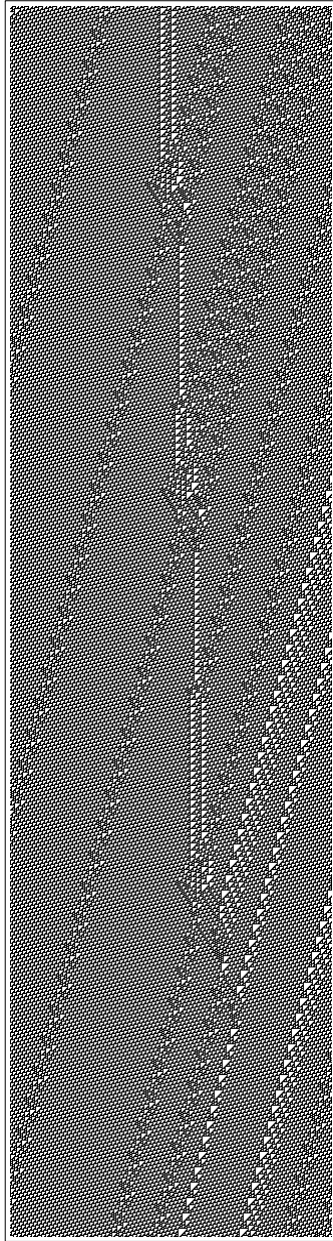


A schematic diagram of how rule 110 can be made to emulate a cyclic tag system. Each line in this diagram corresponds to one localized structure in rule 110. Note that the relative slopes of the structures are reproduced faithfully here, but their spacings are not. Note also that lines shown in different colors here often correspond to the same structure in rule 110.

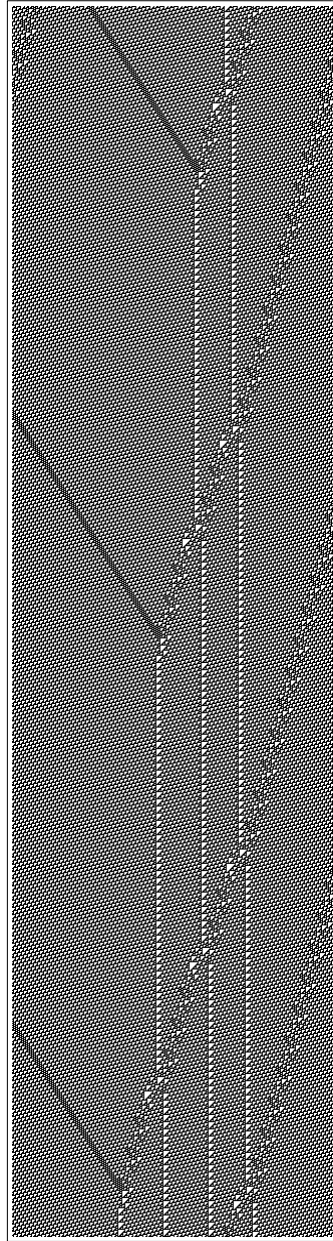




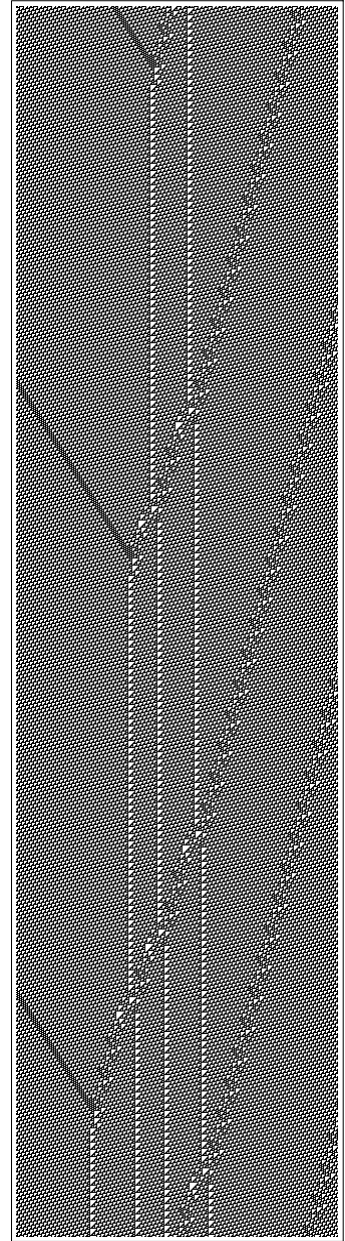
Close-ups of circled regions shown schematically on the previous page. Each picture is 320 cells wide and shows 1200 evolution steps.



(d)

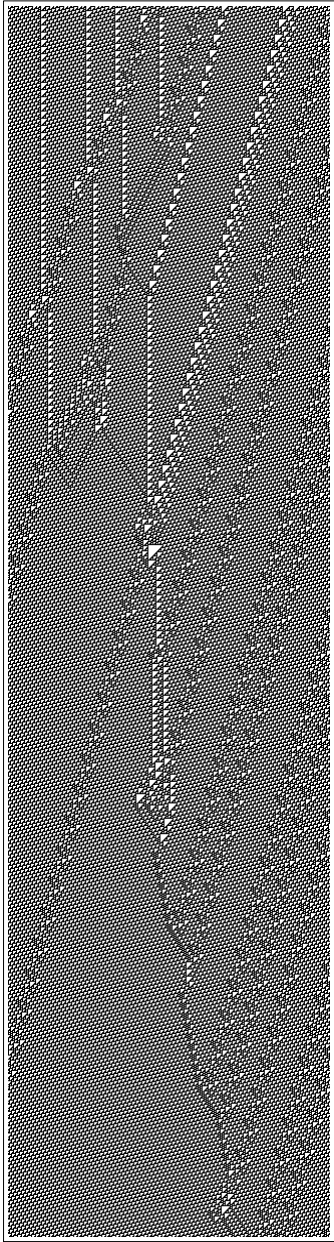


(e)

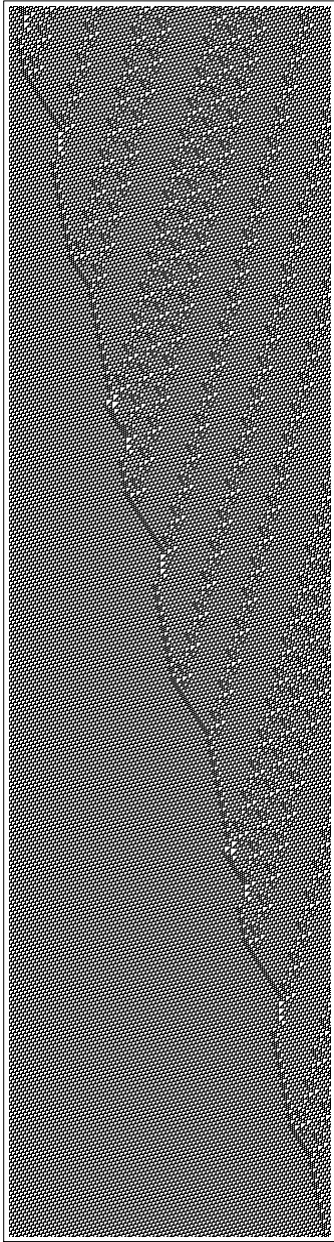


(f)

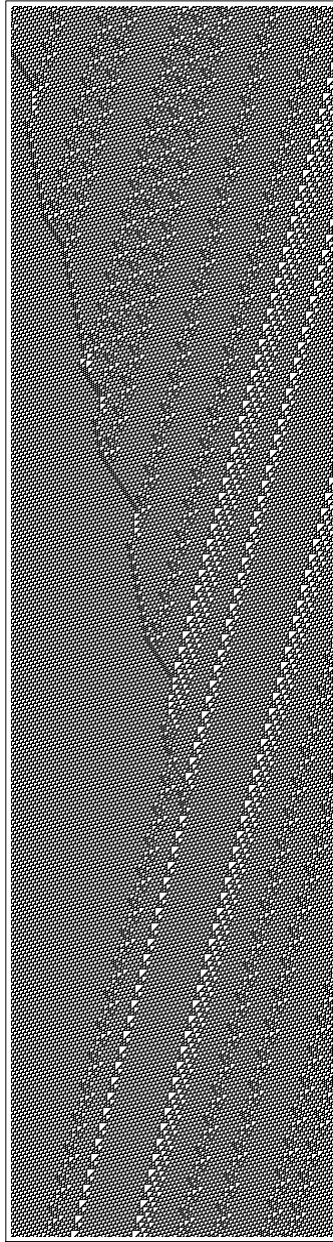
Close-ups (continued).



(g)



(h)



(i)

Close-ups (continued).

additional localized structures are produced that can be seen propagating to the left. These structures could later cause trouble, but looking at region (b) we see that in fact they just pass through other structures that they meet without any adverse effect.

Region (c) shows what happens when the information corresponding to one element in a block passes through the kind of object produced in region (a). The number of localized structures that represent the element is reduced from twelve to four, but the spacings of these structures continue to specify its color. Region (d) then shows how the object in region (c) comes to an end when the beginning of the block separator from the next step arrives.

Region (e) shows how the information corresponding to a black element in a block is actually converted to a new black element in the sequence produced by the cyclic tag system. What happens is that the four localized structures corresponding to the element in the block collide with four other localized structures travelling in the opposite direction, and the result is four stationary structures that correspond to the new element in the sequence.

Region (f) shows the same process as region (e) but for a white element. The fact that the element is white is encoded in the wider spacing of the structures coming from the right, which results in narrower spacing of the stationary structures.

Region (g) shows the analog of region (a), but now for a white element instead of a black one. The region begins much like region (a), except that the four localized structures at the top are more narrowly spaced. Starting around the middle of the region, however, the behavior becomes quite different from region (a): while region (a) yields an object that allows information to pass through, region (g) yields one that stops all information, as shown in regions (h) and (i).

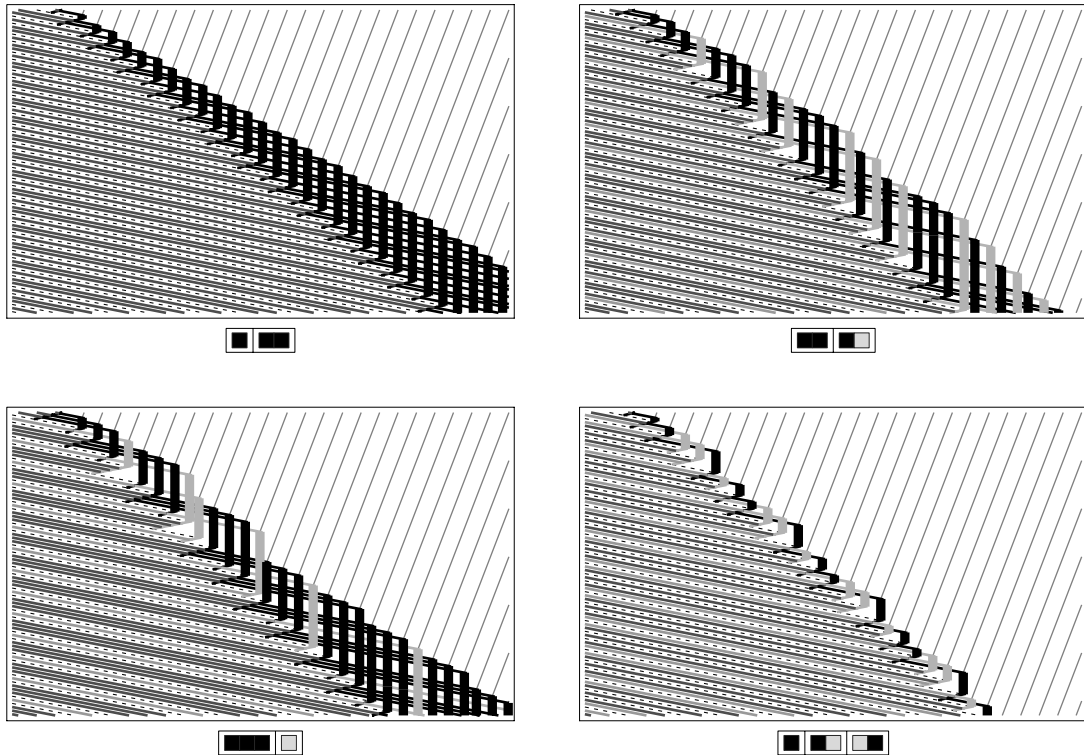
Note that even though they begin very differently, regions (d) and (i) end in the same way, reflecting the fact that in both cases the system is ready to handle a new block, whatever that block may be.

The pictures on the last few pages were all made for a cyclic tag system with a specific underlying rule. But exactly the same principles

can be used whatever the underlying rule is. And the pictures below show schematically what happens with a few other choices of rules.

The way that the lines interact in the interior of each picture is always exactly the same. But what changes when one goes from one rule to another is the arrangement of lines entering the picture.

In the way that the pictures are drawn below, the blocks that appear in each rule are encoded in the pattern of lines coming in from the left edge of the picture. But if each picture were extended sufficiently far to the left, then all these lines would eventually be seen to start from the top. And what this means is that the arrangement of lines can therefore always be viewed as an initial condition for the system.



Schematic diagrams of how cyclic tag systems with four different underlying rules can be emulated. The lines in each diagram correspond essentially to collections of localized structures in rule 110. The processes that occur in the interior of each picture are always the same; the different cyclic tag system rules are implemented by different arrangements of lines entering each picture.

This is then finally how universality is achieved in rule 110. The idea is just to set up initial conditions that correspond to the blocks that appear in the rule for whatever cyclic tag system one wants to emulate.

The necessary initial conditions consist of repetitions of blocks of cells, where each of these blocks contains a pattern of localized structures that corresponds to the block of elements that appear in the rule for the cyclic tag system. The blocks of cells are always quite complicated—for the cyclic tag system discussed in most of this section they are each more than 3000 cells wide—but the crucial point is that such blocks can be constructed for any cyclic tag system. And what this means is that with suitable initial conditions, rule 110 can in fact be made to emulate any cyclic tag system.

It should be mentioned at this point however that there are a few additional complications involved in setting up appropriate initial conditions to make rule 110 emulate many cyclic tag systems. For as the pictures earlier in this section demonstrate, the way we have made rule 110 emulate cyclic tag systems relies on many details of the interactions between localized structures in rule 110. And it turns out that to make sure that with the specific construction used the appropriate interactions continue to occur at every step, one must put some constraints on the cyclic tag systems being emulated.

In essence, these constraints end up being that the blocks that appear in the rule for the cyclic tag system must always be a multiple of six elements long, and that there must be some bound on the number of steps that can elapse between the addition of successive new elements to the cyclic tag system sequence.

Using the ideas discussed on page 669, it is not difficult, however, to make a cyclic tag system that satisfies these constraints, but that emulates any other cyclic tag system. And as a result, we may therefore conclude that rule 110 can in fact successfully emulate absolutely any cyclic tag system. And this means that rule 110 is indeed universal.

## The Significance of Universality in Rule 110

Practical computers and computer languages have traditionally been the only common examples of universality that we ever encounter. And from the fact that these kinds of systems tend to be fairly complicated in their construction, the general intuition has developed that any system that manages to be universal must somehow also be based on quite complicated underlying rules.

But the result of the previous section shows in a rather spectacular way that this is not the case. It would have been one thing if we had found an example of a cellular automaton with say four or five colors that turned out to be universal. But what in fact we have seen is that a cellular automaton with one of the very simplest possible 256 rules manages to be universal.

So what are the implications of this result? Most important is that it suggests that universality is an immensely more common phenomenon than one might otherwise have thought. For if one knew only about practical computers and about systems like the universal cellular automaton discussed early in this chapter, then one would probably assume that universality would rarely if ever be seen outside of systems that were specifically constructed to exhibit it.

But knowing that a system like rule 110 is universal, the whole picture changes, and now it seems likely that instead universality should actually be seen in a very wide range of systems, including many with rather simple rules.

A couple of sections ago we discussed the fact that as soon as one has a system that is universal, adding further complication to its rules cannot have any fundamental effect. For by virtue of its universality the system can always ultimately just emulate the behavior that would be obtained with any more complicated set of rules.

So what this means is that if one looks at a sequence of systems with progressively more complicated rules, one should expect that the overall behavior they produce will become more complex only until the threshold of universality is reached. And as soon as this threshold is passed, there should then be no further fundamental changes in what one sees.

The practical importance of this phenomenon depends greatly however on how far one has to go to get to the threshold of universality.

But knowing that a system like rule 110 is universal, one now suspects that this threshold is remarkably easy to reach. And what this means is that beyond the very simplest rules of any particular kind, the behavior that one sees should quickly become as complex as it will ever be.

Remarkably enough, it turns out that this is essentially what we already observed in Chapter 3. Indeed, not only for cellular automata but also for essentially all of the other kinds of systems that we studied, we found that highly complex behavior could be obtained even with rather simple rules, and that adding further complication to these rules did not in most cases noticeably affect the level of complexity that was produced.

So in retrospect the results of Chapter 3 should already have suggested that simple underlying rules such as rule 110 might be able to achieve universality. But what the elaborate construction in the previous section has done is to show for certain that this is the case.

#### **Class 4 Behavior and Universality**

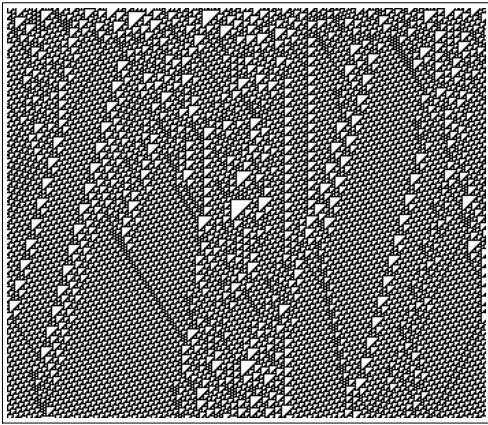
If one looks at the typical behavior of rule 110 with random initial conditions, then the most obvious feature of what one sees is that there are a large number of localized structures that move around and interact with each other in complicated ways. But as we saw in Chapter 6, such behavior is by no means unique to rule 110. Indeed, it is in fact characteristic of all cellular automata that lie in what I called class 4.

The pictures on the next page show a few examples of such class 4 systems. And while the details are different in each case, the general features of the behavior are always rather similar.

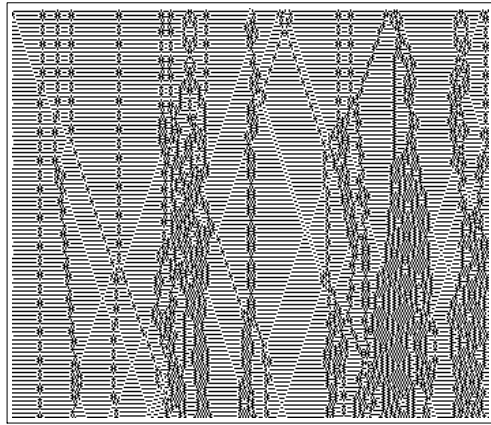
So what does this mean about the computational capabilities of such systems? I strongly suspect that it is true in general that any cellular automaton which shows overall class 4 behavior will turn out—like rule 110—to be universal.

We saw at the end of Chapter 6 that class 4 rules always seem to yield a range of progressively more complicated localized structures. And my expectation is that if one looks sufficiently hard at any

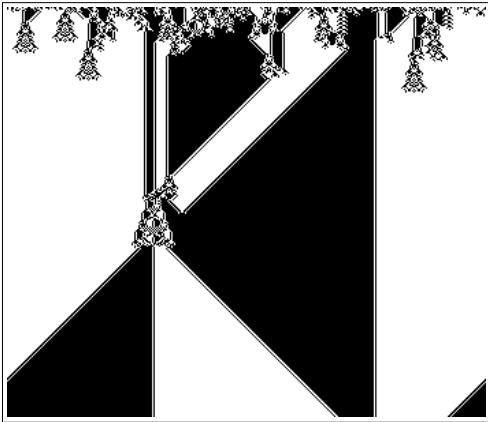




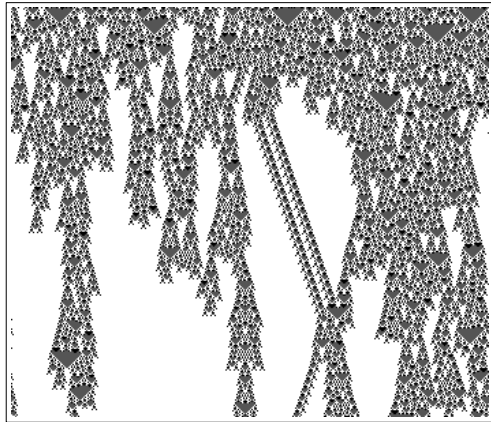
(a) rule 110



(b) second-order rule 37



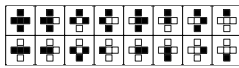
(c) totalistic 2-color next-nearest-neighbor code 52



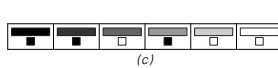
(d) totalistic 3-color code 1815



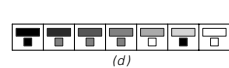
(a)



(b)



(c)



(d)

Examples of cellular automata with class 4 overall behavior, as discussed in Chapter 6. I strongly suspect that all class 4 rules, like rule 110, will turn out to be universal.

particular rule, then one will always eventually be able to find a set of localized structures that is rich enough to support universality.

The final demonstration that a given rule is universal will no doubt involve the same kind of elaborate construction as for rule 110.

But the point is that all the evidence I have so far suggests that for any class 4 rule such a construction will eventually turn out to be possible.

So what kinds of rules show class 4 behavior?

Among the 256 so-called elementary cellular automata that allow only two possible colors for each cell and depend only on nearest neighbors, the only clear immediate example is rule 110—together with rules 124, 137 and 193 obtained by trivially reversing left and right or black and white. But as soon as one allows more than two possible colors, or allows dependence on more than just nearest neighbors, one immediately finds all sorts of further examples of class 4 behavior.

In fact, as illustrated in the pictures on the facing page, it is sufficient in such cases just to use so-called totalistic rules in which the new color of a cell depends only on the average color of cells in its neighborhood, and not on their individual colors.

In two dimensions class 4 behavior can occur with rules that involve only two colors and only nearest neighbors—as shown on page 249. And indeed one example of such a rule is the so-called Game of Life that has been popular in recreational computing since the 1970s.

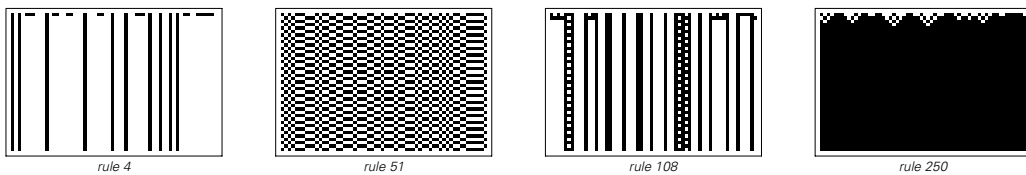
The strategy for demonstrating universality in a two-dimensional cellular automaton is in general very much the same as in one dimension. But in practice the comparative ease with which streams of localized structures can be made to cross in two dimensions can reduce some of the technical difficulties involved. And as it turns out there was already an outline of a proof given even in the 1970s that the Game of Life two-dimensional cellular automaton is universal.

Returning to one dimension, one can ask whether among the 256 elementary cellular automata there are any apart from rule 110 that show even signs of class 4 behavior. As we will see in the next section, one possibility is rule 54. And if this rule is in fact class 4 then it is my expectation that by looking at interactions between the localized structures it supports it will in the end—with enough effort—be possible to show that it too exhibits the phenomenon of universality.

## The Threshold of Universality in Cellular Automata

By showing that rule 110 is universal, we have established that universality is possible even among cellular automata with the very simplest kinds of underlying rules. But there remains the question of what is ultimately needed for a cellular automaton—or any other kind of system—to be able to achieve universality.

In general, if a system is to be universal, then this means that by setting up an appropriate choice of initial conditions it is possible to get the system to emulate any type of behavior that can occur in any other system. And as a consequence, cellular automata like the ones in the pictures below are definitely not universal, since they always produce just simple uniform or repetitive patterns of behavior, whatever initial conditions one uses.



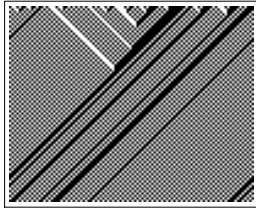
Examples of elementary cellular automata which only ever show purely uniform or purely repetitive behavior, and which therefore definitely cannot be universal. These cellular automata are necessarily all class 1 or class 2 systems.

In a sense the fundamental reason for this—as we discussed on page 252—is that such class 1 and class 2 cellular automata never allow any transmission of information except over limited distances. And the result of this is that they can only support processes that involve the correlated action of a limited number of cells.

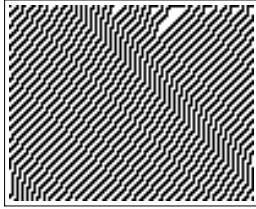
In cellular automata like the ones at the top of the facing page some information can be transmitted over larger distances. But the way this occurs is highly constrained, and in the end these systems can only produce patterns that are in essence purely nested—so that it is again not possible for universality to be achieved.

What about additive rules such as 90 and 150?

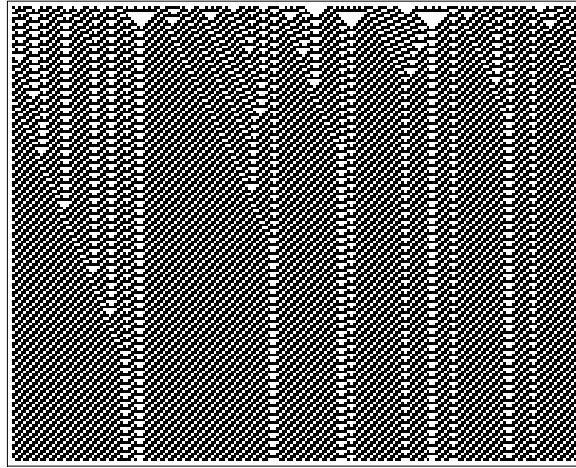
With simple initial conditions these rules always yield very regular nested patterns. But with more complicated initial conditions, they produce more complicated patterns of behavior—as the pictures at



rule 184



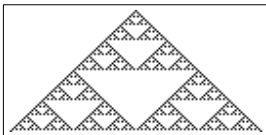
rule 14



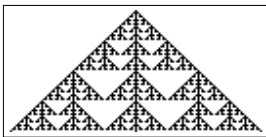
rule 62

Examples of cellular automata that do allow information to be transmitted over large distances, but only in very restricted ways. The overall patterns produced by such cellular automata are essentially nested. No cellular automata of this kind can ever be universal.

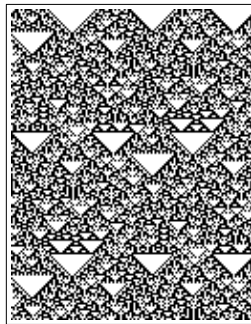
the bottom of this page illustrate. As we saw on page 264, however, these patterns never in fact really correspond to more than rather simple transformations of the initial conditions. Indeed, even after say 1,048,576 steps—or any number of steps that is a power of two—the array of cells produced always turns out to correspond just to a simple superposition of two or three shifted copies of the initial conditions.



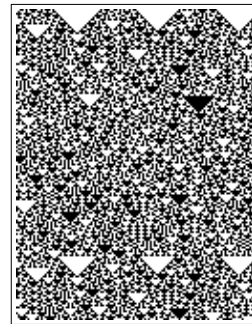
rule 90



rule 150



rule 90

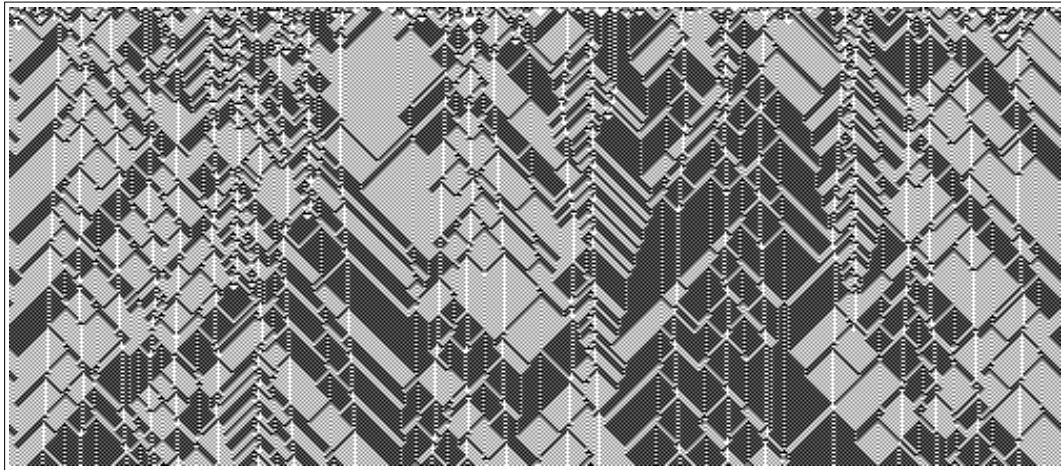
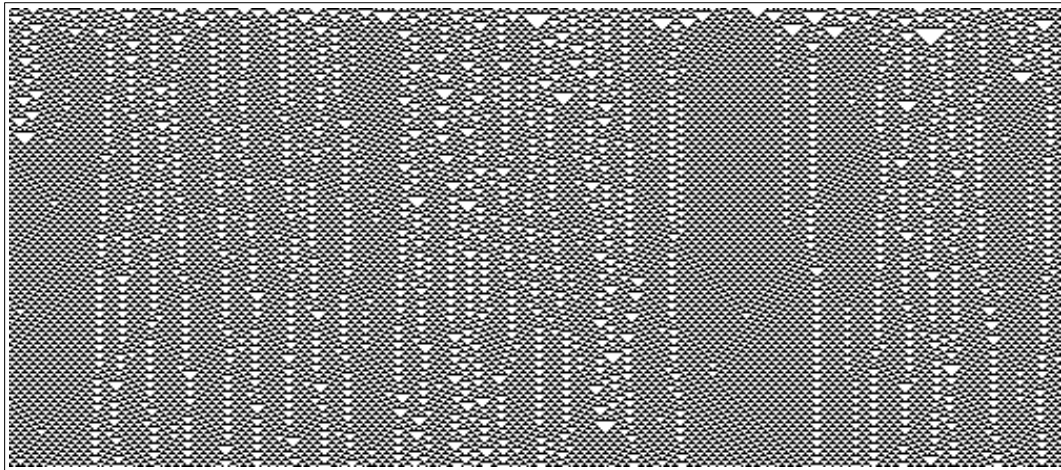


rule 150

Examples of cellular automata with additive rules. The repetitive occurrence of states that correspond to simple transformations of the initial conditions prevent such cellular automata from ever being universal.

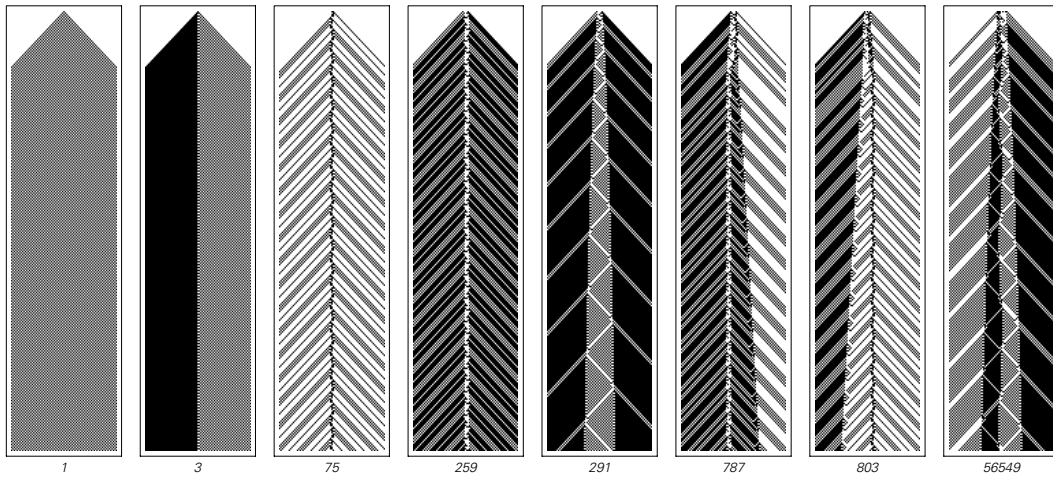
And since there are many kinds of behavior that do not return to such predictable forms after any limited number of steps, one must conclude that additive rules cannot be universal.

At the end of the last section I mentioned rule 54 as another elementary cellular automaton besides rule 110 that might be class 4. The pictures below show examples of the typical behavior of rule 54.



Two views of the evolution of rule 54 from typical random initial conditions. The top view shows the color of every cell at every step. The bottom groups together pairs of cells, and shows only every other step. There are various localized structures—and hints of class 4 behavior.

Some localized structures are definitely seen. But are they enough to support class 4 behavior and universality? The pictures below show what happens if one starts looking in turn at each of the possible initial conditions for rule 54. At first one sees only simple repetitive behavior. At initial condition 291 one sees a very simple form of nesting. And as one continues one sees various other repetitive and nested forms. But at least up to the hundred millionth initial condition one sees nothing that is fundamentally any more complicated.



Forms of behavior seen in the first 100 million initial conditions for rule 54. With initial condition 291 the  $n^{\text{th}}$  new stripe on the right is produced at step  $2n^2 + 8n - 9$ . Even in the last case shown, the arrangement of stripes eventually becomes completely regular, with the  $n^{\text{th}}$  new stripe being produced at step  $n^2 + 21n/2 - \{6, 5, -4, 3\} \llbracket \text{Mod}[n, 4] + 1 \rrbracket / 2$ . Pairs of cells are grouped together in each picture, as at the bottom of the facing page.

So can rule 54 achieve universality? I am not sure. It could be that if one went just a little further in looking at initial conditions one would see more complicated behavior. And it could be that even the structures shown above can be combined to produce all the richness that is needed for universality. But it could also be that whatever one does rule 54 will always in the end just show purely repetitive or nested behavior—which cannot on its own support universality.

What about other elementary cellular automata?

As I will discuss in the next chapter, my general expectation is that more or less any system whose behavior is not somehow fundamentally repetitive or nested will in the end turn out to be universal. But I suspect that this fact will be very much easier to establish for some systems than for others—with rule 110 being one of the easiest cases.

In general what one needs to do in order to prove universality is to find a procedure for setting up initial conditions in one system so as to make it emulate some general class of other systems. And at some level the main challenge is that our experience from programming and engineering tends to provide us with only a limited set of methods for coming up with such a procedure. Typically what we are used to doing is constructing things in stages. Usually we start by building components, and then we progressively assemble these into larger and larger structures. And the point is that at each stage, we need think directly only about the scale of structures that we are currently handling—and not for example about all the pieces that make up these structures.

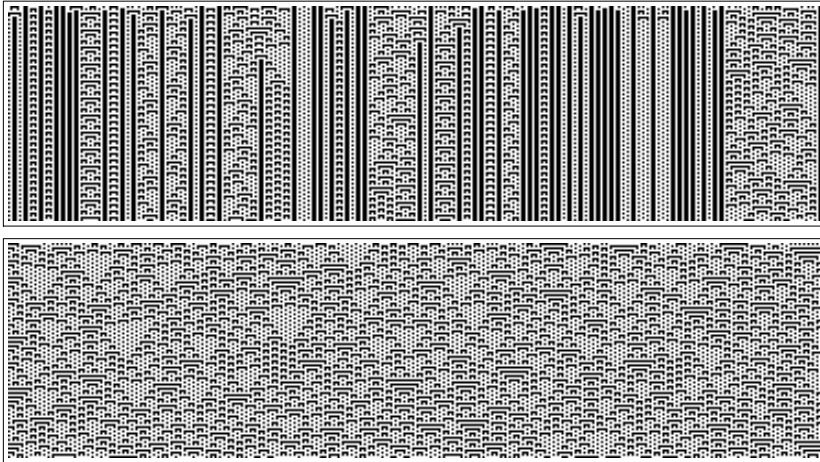
In proving the universality of rule 110, we were able to follow essentially the same basic approach. We started by identifying various localized structures, and then we used these structures as components in building up the progressively larger structures that we needed.

What was in a sense crucial to our approach was therefore that we could readily control the transmission of information in the system. For this is what allowed us to treat different localized structures as being separate and independent objects.

And indeed in any system with class 4 behavior, things will typically always work in more or less the same way. But in class 3 systems they will not. For what usually happens in such systems is that a change made even to a single cell will eventually spread to affect all other cells. And this kind of uncontrolled transmission of information makes it very difficult to identify pieces that could be used as definite components in a construction.

So what can be done in such cases? The most obvious possibility is that one might be able to find special classes of initial conditions in which transmission of information could be controlled. And an example where this can be potentially done is rule 73.

The pictures below show the typical behavior of rule 73—first with completely random initial conditions, and then with initial conditions in which no run of an even number of black squares occurs.



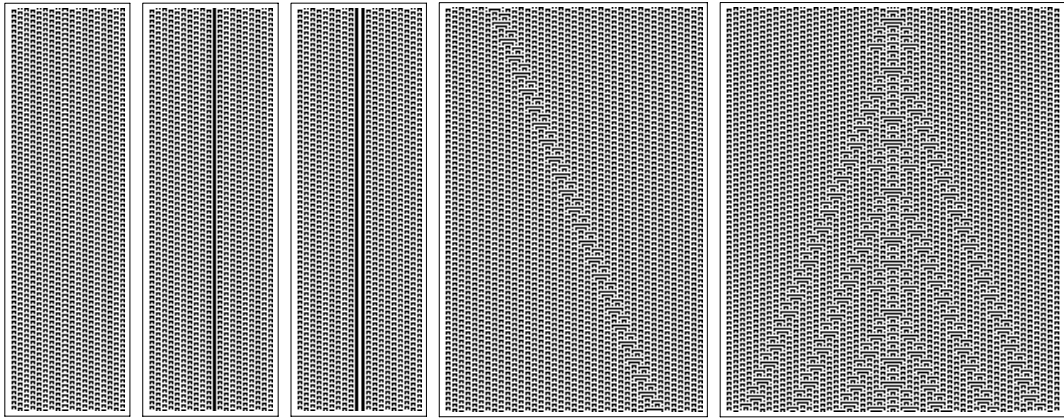
Two examples of rule 73. The top example uses completely random initial conditions; the bottom example uses initial conditions in which no run of an even number of black squares ever occurs. The bottom example is actually part of the pattern obtained from a single black cell—just to the right of the center column, starting with step 1000.

In the second case rule 73 exhibits typical class 3 behavior—with the usual uncontrolled transmission of information. In the first case, however, the black walls that are present seem to prevent any long-range transmission of information at all.

So can one then achieve something intermediate in rule 73—in which information is transmitted, but only in a controlled way?

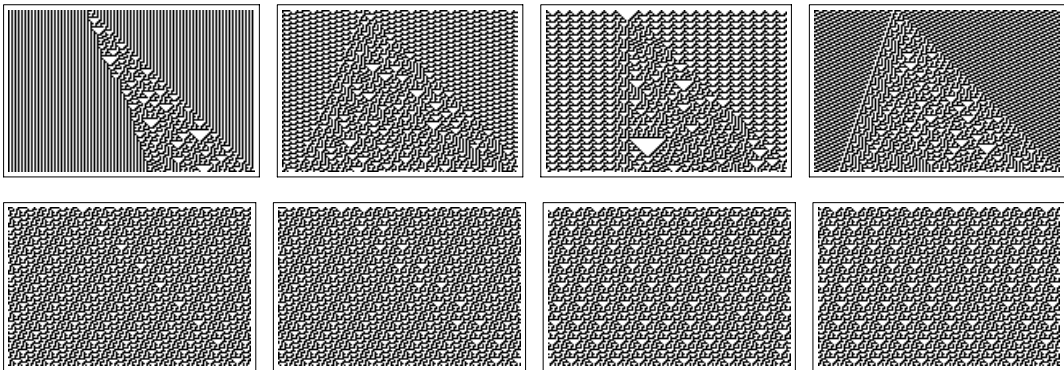
The pictures at the top of the next page give some indication of how this might be done. For they show that with an appropriate background rule 73 supports various localized structures, some of which move. And while these structures may at first seem more like those in rule 54 than rule 110, I strongly suspect that the complexity of the typical behavior of rule 73 will be reflected in more sophisticated interactions between the structures—and will eventually provide what is needed to allow universality to be demonstrated in much the same way as in rule 110.





Examples of localized structures in rule 73. Note that in the last case shown, the background patterns on either side are mirror images.

So what about a case like rule 30? With strictly repetitive initial conditions—like any cellular automaton—this must yield purely repetitive behavior. But as soon as one perturbs such initial conditions, one normally seems to get only complicated and seemingly random behavior, as in the top row of pictures below.

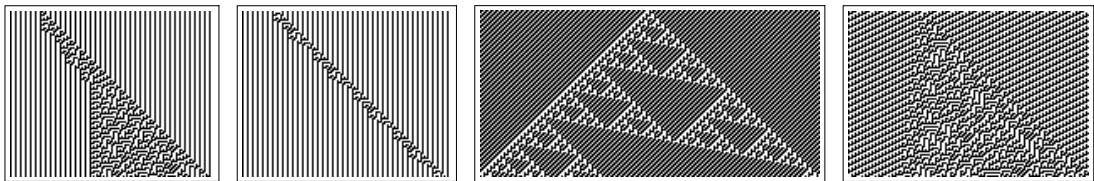


Examples of patterns produced by rule 30 with repetitive backgrounds. The top row shows the effect of inserting a single extra black cell into various backgrounds. The bottom row shows all localized structures involving up to 25 cells supported by rule 30 on repetitive backgrounds with blocks of up to 25 cells. Note that these localized structures always move one cell to the right at each step—making it impossible for them to interact in non-trivial ways.

Yet it turns out still to be possible to get localized structures—as the bottom row of pictures above demonstrate. But these structures

always seem to move at the same speed, and so can never interact. And even after searching many billions of cases, I have never succeeded in finding any useful set of localized structures in rule 30.

The picture below shows what happens in rule 45. Many possible perturbations to repetitive initial conditions again yield seemingly random behavior. But in one case a nested pattern is produced. And structures that remain localized are now fairly common—but just as in rule 30 always seem to move at the same speed.



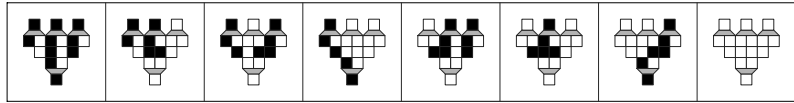
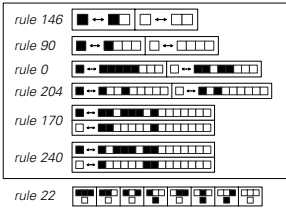
Examples of patterns produced by inserting a single extra black cell into repetitive backgrounds for rule 45. Note the appearance of a slanted version of the nested pattern from rule 90. In rule 45, localized structures turn out to be fairly common—but as in rule 30 they always seem to move at the same speed, and so presumably cannot interact to produce any kind of class 4 behavior.

So although this means that the particular type of approach we used to demonstrate the universality of rule 110 cannot immediately be used for rule 30 or rule 45, it certainly does not mean that these rules are not in the end universal. And as I will discuss in the next chapter, it is my very strong belief that in fact they will turn out to be.

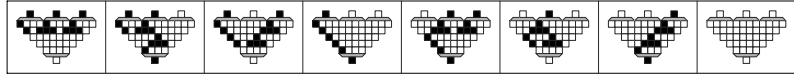
So how might we get evidence for this?

If a system is universal, then this means that with a suitable encoding of initial conditions its evolution must emulate the evolution of any other system. So this suggests that one might be able to get evidence about universality just by trying different possible encodings, and then seeing what range of other systems they allow one to emulate.

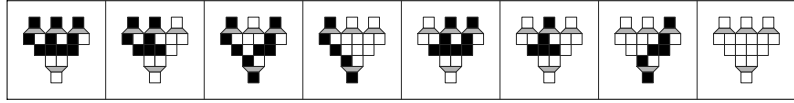
In the case of the 19-color universal cellular automaton on page 645 it turns out that encodings in which individual black and white cells are represented by particular 20-cell blocks are sufficient to allow the universal cellular automaton to emulate all 256 possible elementary cellular automata—with one step in the evolution of each of these corresponding to 53 steps in the evolution of the original system.



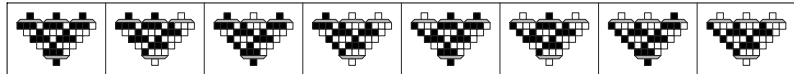
rule 146



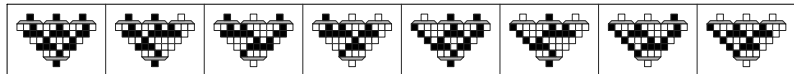
rule 90



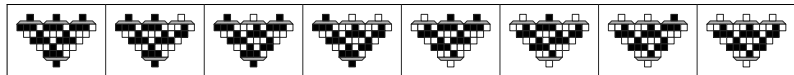
rule 50



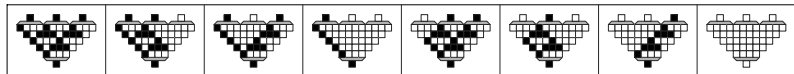
rule 170



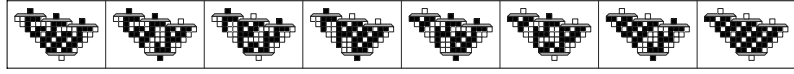
rule 204



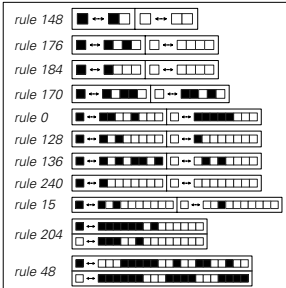
rule 240



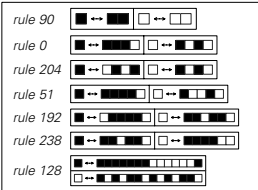
rule 254



rule 90



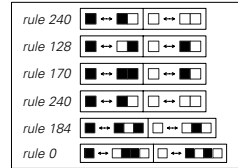
rule 41



rule 94



rule 110



rule 184

Examples of using various specific elementary cellular automata to emulate other elementary cellular automata. In each case single cells are encoded as blocks of cells, and all distinct such encodings with blocks up to length 20 are shown.

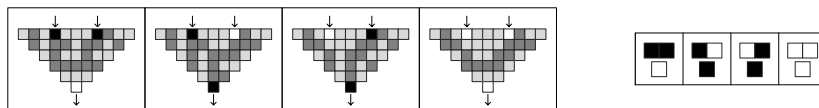
So given a particular elementary cellular automaton one can then ask what other elementary cellular automata it can emulate using blocks up to a certain length.

The pictures on the facing page show a few examples.

The results are not particularly dramatic. No single rule is able to emulate many others—and the rules that are emulated tend to be rather simple. An example of a slight surprise is that rule 45 ends up being able to emulate rule 90. But at least with blocks up to length 25, rule 30 for example is not able to emulate any non-trivial rules at all.

From the proof of universality that we gave it follows that rule 110 must be able to emulate any other elementary cellular automaton with blocks of some size—but with the actual construction we discussed this size will be quite astronomical. And certainly in the picture on the facing page rule 110 does not seem to stand out.

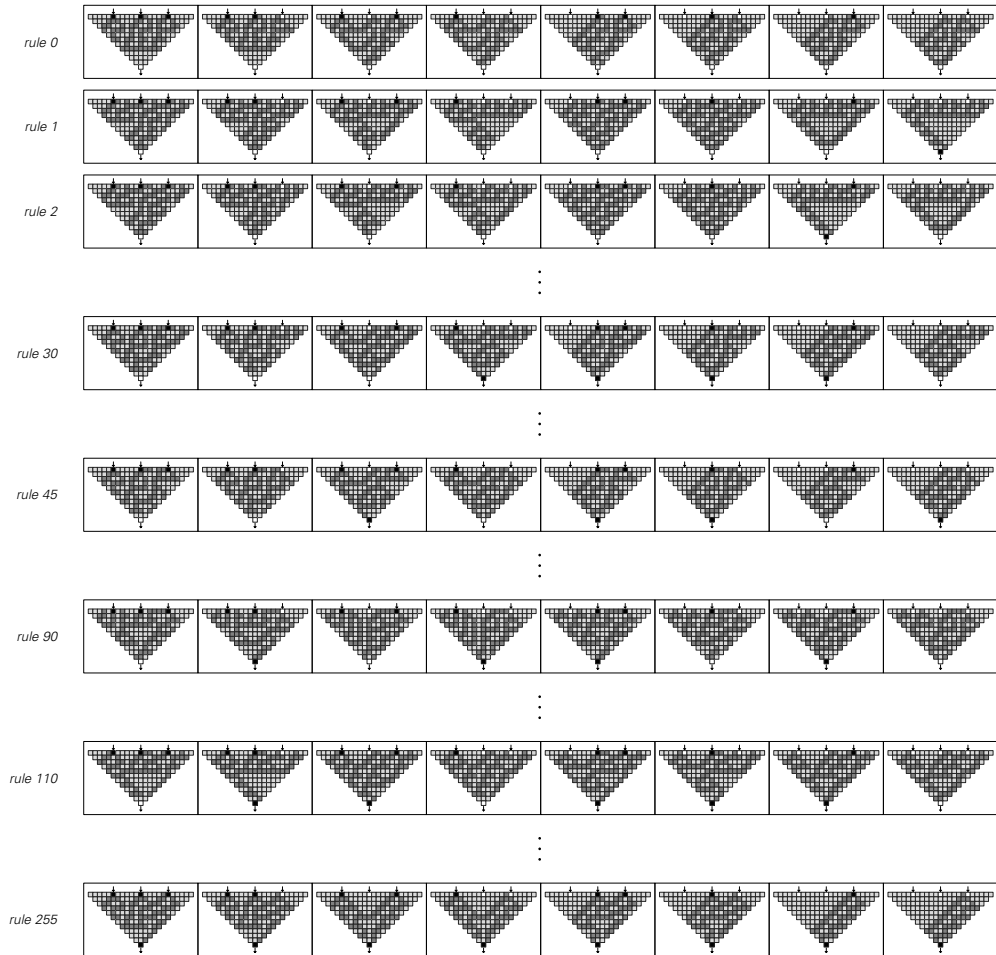
But although it seems somewhat difficult to emulate the complete evolution of one cellular automaton with another, it turns out to be much easier to emulate fragments of evolution for limited numbers of steps. And as an example the picture below shows how rule 30 can be made to emulate the basic action of one step in rule 90.



Rule 30 set up to emulate a single XOR operation—as used in a step of rule 90 evolution. The initial conditions for rule 30 are fixed except at the two positions indicated, where input can effectively be given. The picture shows that for each possible combination of inputs, the result from the rule 30 evolution corresponds exactly to the output from the XOR.

The idea is to set up a configuration in rule 30 so that if one inserts input at particular positions the output from the underlying rule 30 evolution corresponds exactly to what one would get from a single step of rule 90 evolution. And in the particular case shown, this is achieved by having blocks 3 cells wide between each input position.

But as the picture on the next page indicates, by having appropriate blocks 5 cells wide rule 30 can actually be made to emulate



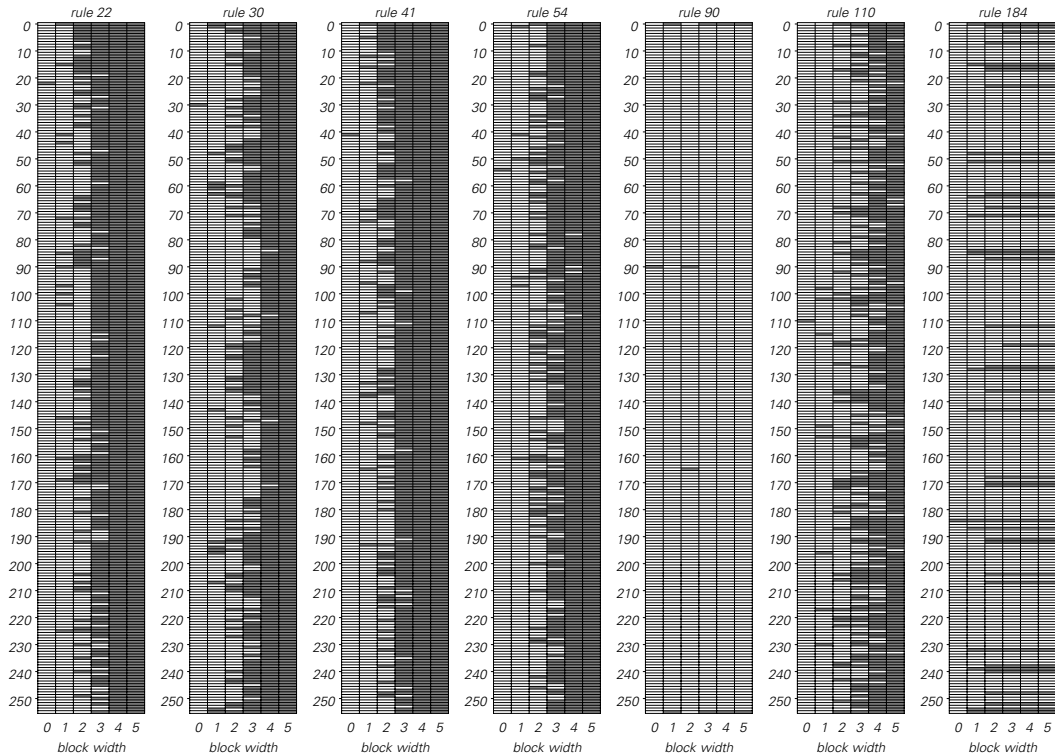
Illustrations of how rule 30 can be set up to emulate a single step in the evolution of all elementary cellular automata.

one step in the evolution of every single one of the 256 possible elementary cellular automata.

So what about other underlying rules?

The picture on the facing page shows for several different underlying rules which of the 256 possible elementary rules can successfully be emulated with successively wider blocks. In cases where the underlying rules have only rather simple behavior—as with rules 90 and 184—it turns out that it is never possible to emulate more than a

few of the 256 possible elementary rules. But for underlying rules that have more complex behavior—like rules 22, 30, or 110—it turns out that in the end it is always possible to emulate all 256 elementary rules.



Summaries of how various underlying cellular automata do in emulating a single step in the evolution of each of the 256 possible elementary cellular automata using the scheme from the facing page with blocks of successively greater widths.

The emulation here is, however, only for a single step. So the fact that it is possible does not immediately establish universality in any ordinary sense. But it does once again support the idea that almost any cellular automaton whose behavior seems to us complex can be made to do computations that are in a sense as sophisticated as one wants.

And this suggests that such cellular automata will in the end turn out to be universal—with the result that out of the 256 elementary rules one expects that perhaps as many as 27 will in fact be universal.

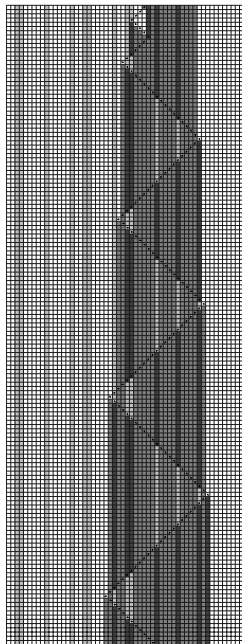
## Universality in Turing Machines and Other Systems

From the results of the previous few sections, we now have some idea where the threshold for universality lies in cellular automata. But what about other kinds of systems—like Turing machines? How complicated do the rules need to be in order to get universality?

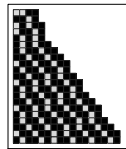
In the 1950s and early 1960s a certain amount of work was done on trying to construct small Turing machines that would be universal. The main achievement of this work was the construction of the universal machine with 7 states and 4 possible colors shown below.



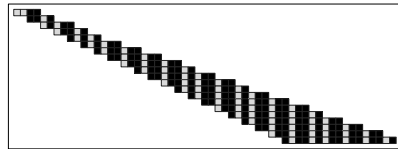
The rule for a universal Turing machine with 7 states and 4 colors constructed in 1962. Until now, this was essentially the simplest known universal Turing machine. Note that one element of the rule can be considered as specifying that the Turing machine should “halt” with the head staying in the same location and same state.



*Turing machine evolution*

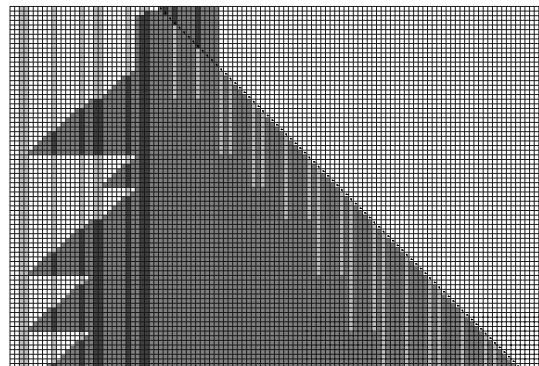
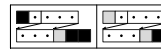


*tag system evolution*



*tag system evolution shifted*

*tag system rule:*



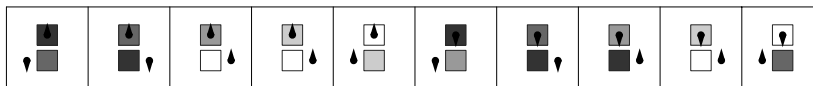
*Turing machine evolution compressed*

An example of how the Turing machine above can emulate a tag system. A black element in the tag system is set up to correspond to a block of four cells in the Turing machine, while a white element corresponds to a single cell.

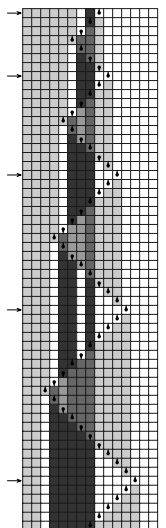
The picture at the bottom of the facing page shows how universality can be proved in this case. The basic idea is that by setting up appropriate initial conditions on the left, the Turing machine can be made to emulate any tag system of a certain kind. But it then turns out from the discussion of page 667 that there are tag systems of this kind that are universal.

It is already an achievement to find a universal Turing machine as comparatively simple as the one on the facing page. And indeed in the forty years since this example was found, no significantly simpler one has been found. So one might conclude from this that the machine on the facing page is somehow at the threshold for universality in Turing machines.

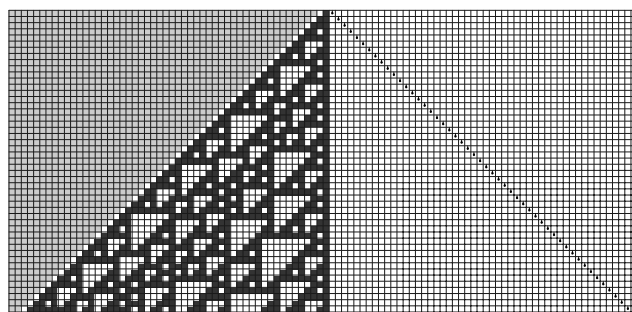
But as one might expect from the discoveries in this book, this is far from correct. And in fact, by using the universality of rule 110 it turns out to be possible to come up with the vastly simpler universal Turing machine shown below—with just 2 states and 5 possible colors.



The rule for the simplest Turing machine currently known to be universal, based on discoveries in this book. The machine has 2 states and 5 possible colors.



*Turing machine evolution*



*Turing machine evolution compressed*

An example of how the Turing machine above manages to emulate rule 110. The compressed picture is made by keeping only the steps indicated at which the head is further to the right than ever before. To get the picture shown requires running the Turing machine for a total of 5000 steps.

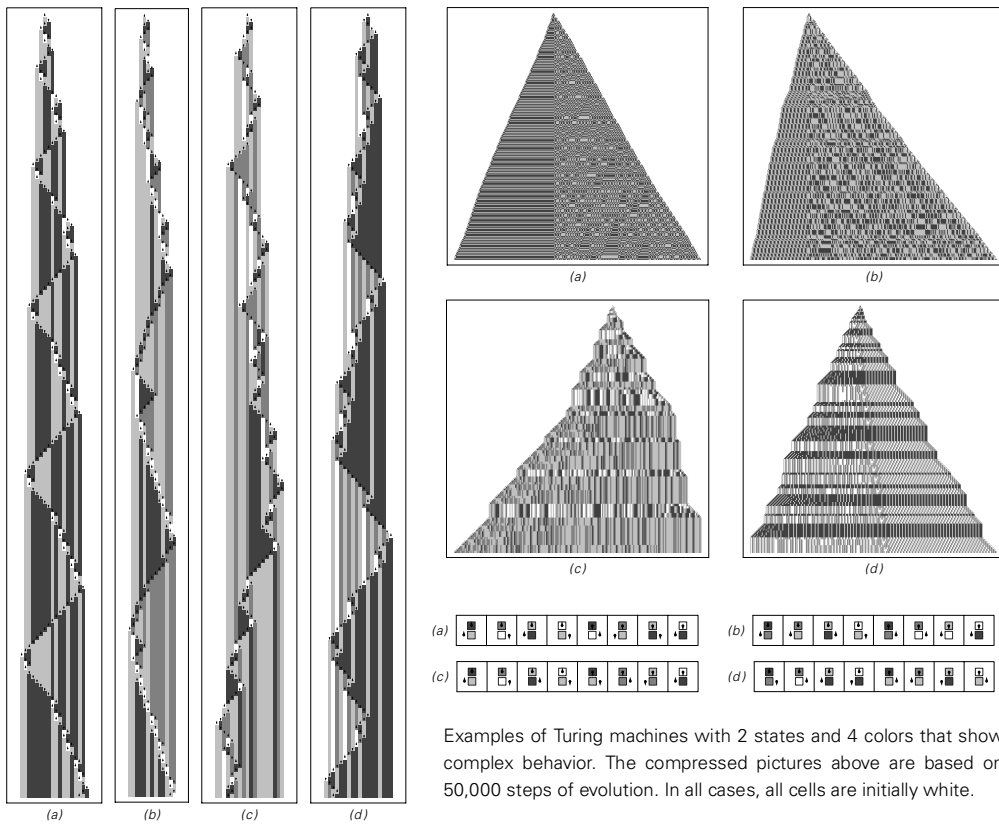


As the picture at the bottom of the previous page illustrates, this Turing machine emulates rule 110 in a quite straightforward way: its head moves systematically backwards and forwards, at each complete sweep updating all cells according to a single step of rule 110 evolution. And knowing from earlier in this chapter that rule 110 is universal, it then follows that the 2-state 5-color Turing machine must also be universal.

So is this then the simplest possible universal Turing machine?

I am quite certain that it is not. And in fact I expect that there are some significantly simpler ones. But just how simple can they actually be?

If one looks at the 4096 Turing machines with 2 states and 2 colors it is fairly easy to see that their behavior is in all cases too simple to support universality. So between 2 states and 2 colors and 2 states and 5 colors, where does the threshold for universality in Turing machines lie?



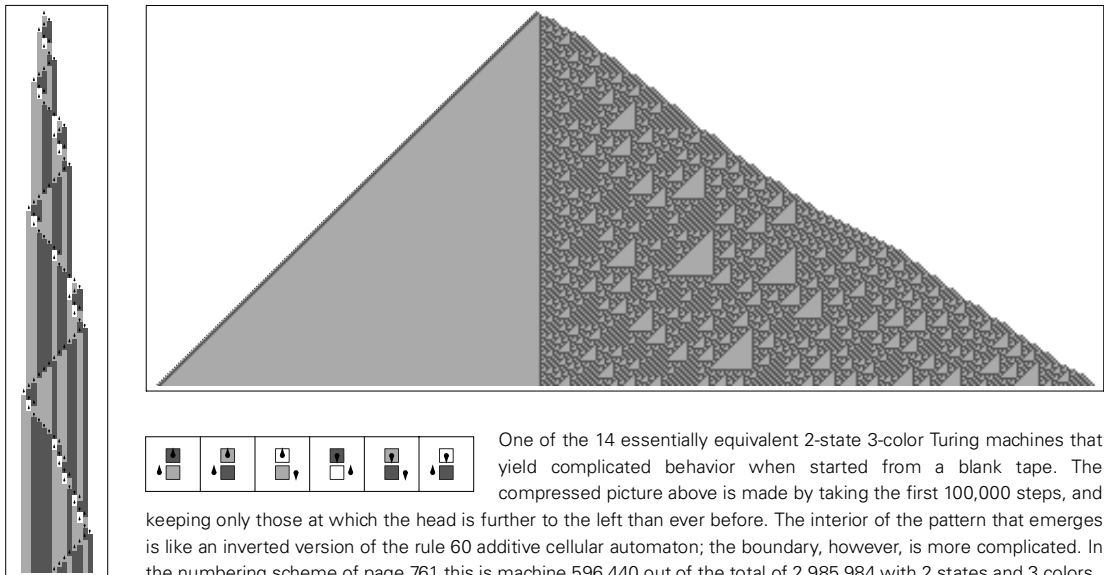
Examples of Turing machines with 2 states and 4 colors that show complex behavior. The compressed pictures above are based on 50,000 steps of evolution. In all cases, all cells are initially white.

The pictures at the bottom of the facing page give examples of some 2-state 4-color Turing machines that show complex behavior. And I have little doubt that most if not all of these are universal.

Among such 2-state 4-color Turing machines perhaps one in 50,000 shows complex behavior when started from a blank tape. Among 4-state 2-color Turing machines the same kind of complex behavior is also seen—as discussed on page 81—but now it occurs only in perhaps one out of 200,000 cases.

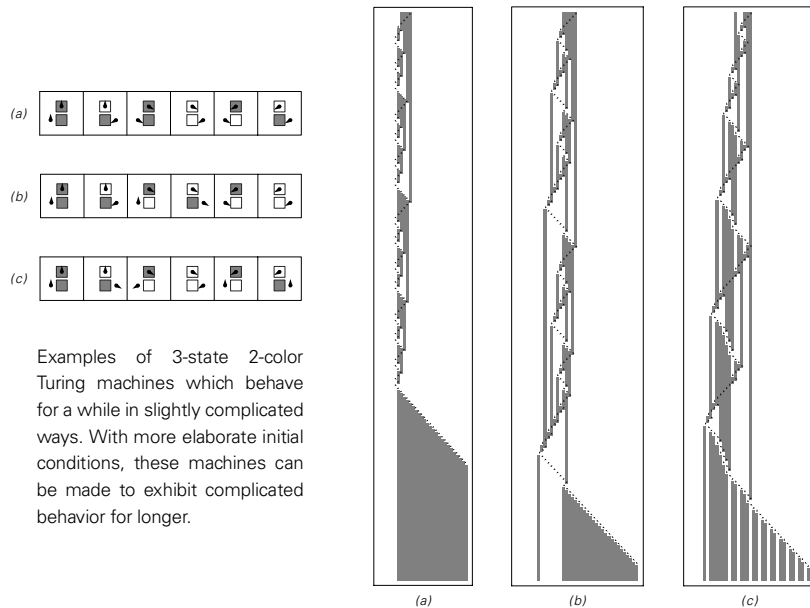
So what about Turing machines with 2 states and 3 colors? There are a total of 2,985,984 of these. And most of them yield fairly simple behavior. But it turns out that 14 of them—all essentially equivalent—produce considerable complexity, even when started from a blank tape.

The picture below shows an example.



And although it will no doubt be very difficult to prove, it seems likely that this Turing machine will in the end turn out to be universal. And if so, then presumably it will by most measures be the very simplest Turing machine that is universal.

With 3 states and 2 colors it turns out that with blank initial conditions all of the 2,985,984 possible Turing machines of this type quickly evolve to produce simple repetitive or nested behavior. With more complicated initial conditions the behavior one sees can sometimes be more complicated, at least for a while—as in the pictures below. But in the end it still always seems to resolve into a simple form.



Yet despite this, it still seems conceivable that with appropriate initial conditions significantly more complex behavior might occur—and might ultimately allow universality in 3-state 2-color Turing machines.

From the universality of rule 110 we know that if one just starts enumerating cellular automata in a particular order, then after going through at most 110 rules, one will definitely see universality. And from other results earlier in this chapter it seems likely that in fact one would tend to see universality even somewhat earlier—after going through only perhaps just ten or twenty rules.

Among Turing machines, the universal 2-state 5-color rule on page 707 can be assigned the number 8,679,752,795,626. So this means

that after going through perhaps nine trillion Turing machines one will definitely tend to find an example that is universal. But presumably one will actually find examples much earlier—since for example the 2-state 3-color machine on page 709 is only number 596,440.

And although these numbers are larger than for cellular automata, the fact remains that the simplest potentially universal Turing machines are still very simple in structure, suggesting that the threshold for universality in Turing machines—just like in cellular automata—is in many respects very low.

So what about other types of systems?

I suspect that in almost any case where we have seen complex behavior earlier in this book it will eventually be possible to show that there is universality. And indeed, as I will discuss at length in the next chapter, I believe that in general there is a close connection between universality and the appearance of complex behavior.

Previous examples of systems that are known to be universal have typically had rules that are far too complicated to see this with any clarity. But an almost unique instance where it could potentially have been seen even long ago are what are known as combinators.

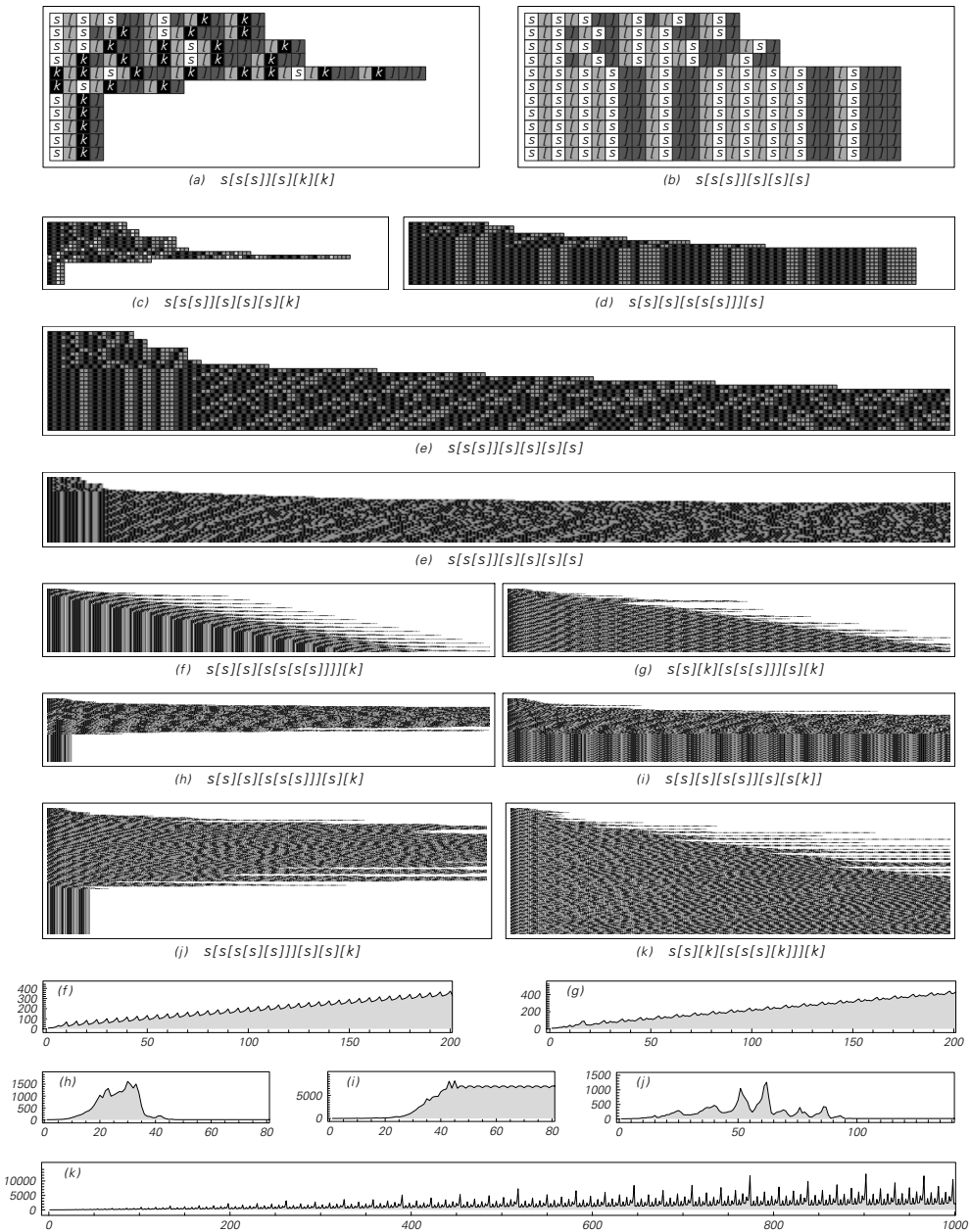
Combinators are a particular case of the symbolic systems that we discussed on page 102 of Chapter 3. Originally intended as an idealized way to represent structures of functions defined in logic, combinators were actually first introduced in 1920—sixteen years before Turing machines. But although they have been investigated somewhat over the past eighty years, they have for the most part been viewed as rather obscure and irrelevant constructs.

The basic rules for combinators are given below.

$$\begin{array}{l} s[x\_][y\_][z\_]\rightarrow x[z][y[z]] \\ k[x\_][y\_]\rightarrow x \end{array}$$

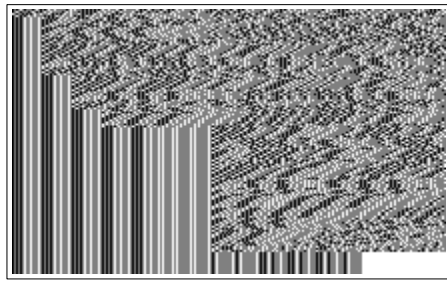
Rules for symbolic systems known as combinators, first introduced in 1920, and proved universal by the mid-1930s.

With short initial conditions, the pictures at the top of the next page demonstrate that combinators tend to evolve quickly to simple fixed points. But with initial condition (e) of length 8 the pictures show

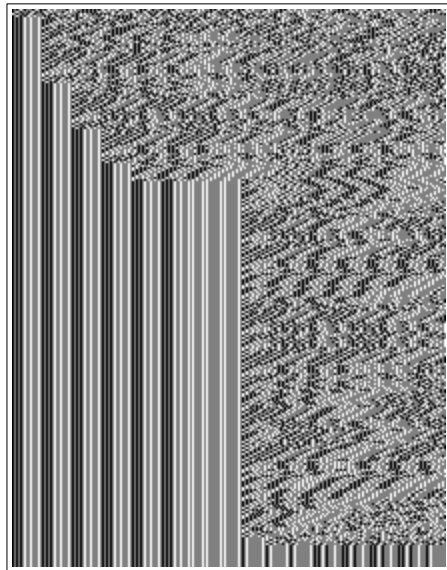
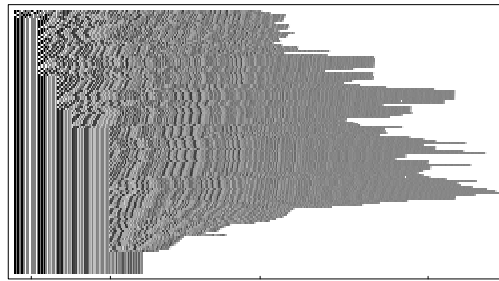


Examples of combinator evolution. The expression in case (e) is the shortest that leads to unlimited growth. The plots at the bottom show the total sizes of expressions reached on successive steps. Note that the detailed pattern of evolution—though not any final fixed point reached—can depend on the fact that the combinator rules are applied at each step in *Mathematica* / order.

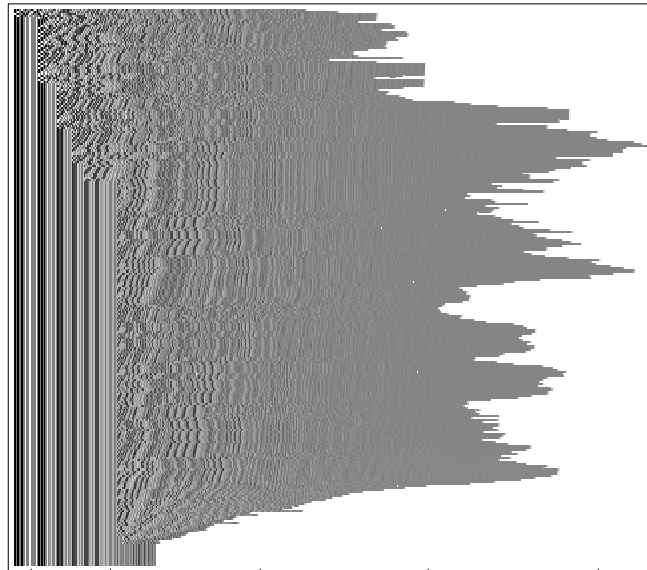




step 1



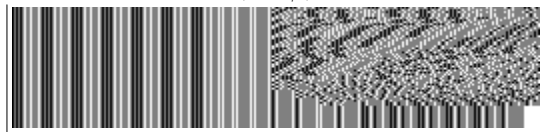
step 2



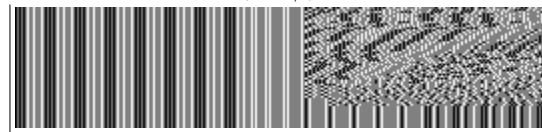
(459 steps)



(664 steps)



step 3



step 4



Emulating the rule 110 cellular automaton using combinators. The rule 110 combinator from the previous page is applied once for each step of rule 110 evolution. The initial state is taken to consist of a single black cell.

# The Notion of Computation

## Computation as a Framework

■ **History of computing.** Even in prehistoric times there were no doubt schemes for computation based for example on making specific arrangements of pebbles. Such schemes were somewhat formalized a few thousand years ago with the invention of the abacus. And by about 200 BC the development of gears had made it possible to create devices (such as the Antikythera device from perhaps around 90 BC) in which the positions of wheels would correspond to positions of astronomical objects. By about 100 AD Hero had described an odometer-like device that could be driven automatically and could effectively count in digital form. But it was not until the 1600s that mechanical devices for digital computation appear to have actually been built. Around 1621 Wilhelm Schickard probably built a machine based on gears for doing simplified multiplications involved in Johannes Kepler's calculations of the orbit of the Moon. But much more widely known were the machines built in the 1640s by Blaise Pascal for doing addition on numbers with five or so digits and in the 1670s by Gottfried Leibniz for doing multiplication, division and square roots. At first, these machines were viewed mainly as curiosities. But as the technology improved, they gradually began to find practical applications. In the mid-1800s, for example, following the ideas of Charles Babbage, so-called difference engines were used to automatically compute and print tables of values of polynomials. And from the late 1800s until about 1970 mechanical calculators were in very widespread use. (In addition, starting with Stanley Jevons in 1869, a few machines were constructed for evaluating logic expressions, though they were viewed almost entirely as curiosities.)

In parallel with the development of devices for digital computation, various so-called analog computers were also built that used continuous physical processes to in effect perform computations. In 1876 William Thomson (Kelvin)

constructed a so-called harmonic analyzer, in which an assembly of disks were used to sum trigonometric series and thus to predict tides. Kelvin mentioned that a similar device could be built to solve differential equations. This idea was independently developed by Vannevar Bush, who built the first mechanical so-called differential analyzer in the late 1920s. And in the 1930s, electrical analog computers began to be produced, and in fact they remained in widespread use for finding approximate solutions to differential equations until the late 1960s.

The types of machines discussed so far all have the feature that they have to be physically rearranged or rewired in order to perform different calculations. But the idea of a programmable machine already emerged around 1800, first with player pianos, and then with Marie Jacquard's invention of an automatic loom which used punched cards to determine its weaving patterns. And in the 1830s, Charles Babbage described what he called an analytical engine, which, if built, would have been able to perform sequences of arithmetic operations under punched card control. Starting at the end of the 1800s tabulating machines based on punched cards became widely used for commercial and government data processing. Initially, these machines were purely mechanical, but by the 1930s, most were electromechanical, and had units for carrying out basic arithmetic operations. The Harvard Mark I computer (proposed by Howard Aiken in 1937 and completed in 1944) consisted of many such units hooked together so as to perform scientific calculations. Following work by John Atanasoff around 1940, electronic machines with similar architectures started to be built. The first large-scale such system was the ENIAC, built between 1943 and 1946. The focus of the ENIAC was on numerical computation, originally for creating ballistics tables. But in the early 1940s, the British wartime cryptanalysis group (which included Alan Turing) constructed fairly large electromechanical machines that performed logical, rather than arithmetic, operations.



All the systems mentioned so far had the feature that they performed operations in what was essentially a fixed sequence. But by the late 1940s it had become clear, particularly through the writings of John von Neumann, that it would be convenient to be able to jump around instead of always having to follow a fixed sequence. And with the idea of storing programs electronically, this became fairly easy to do, so that by 1950 more than ten stored-program computers had been built in the U.S. and in England. Speed and memory capacity have increased immensely since the 1950s, particularly as a result of the development of semiconductor chip technology, but in many respects the basic hardware architecture of computers has remained very much the same.

Major changes have, however, occurred in software. In the late 1950s and early 1960s, the main innovation was the development of computer languages such as FORTRAN, COBOL and BASIC. These languages allowed programs to be specified in a somewhat abstract way, independent of the precise details of the hardware architecture of the computer. But the languages were primarily intended only for specifying numerical calculations. In the late 1960s and early 1970s, there developed the notion of operating systems—programs whose purpose was to control the resources of a computer—and with them came languages such as C. And then in the late 1970s and early 1980s, as the cost of computer memory fell, it began to be feasible to manipulate not just purely numerical data, but also data representing text and later pictures. With the advent of personal computers in the early 1980s, interactive computing became common, and as the resolution of computer displays increased, concepts such as graphical user interfaces developed. In more recent years continuing increases in speed have made it possible for more and more layers of software to be constructed, and for many operations previously done with special hardware to be implemented purely in software.

■ **Practical computers.** At the lowest level the hardware of a practical computer consists of digital electronic circuits. In these circuits, lumps of electric charge (in 2001 about half a million electrons each) flow through channels which cross to form various kinds of gates. Each gate performs a simple logic operation; for example, letting charge pass in one channel only if charge is present in the other channel. From circuits containing millions of such gates are built the two main elements of the computer: the processor which actually performs computations, and the memory which stores data. The memory consists of an array of cells, with the presence or absence of a lump of charge at gates in each cell representing a 1 or 0 value for the bit of data associated with that cell.

One of the crucial ideas of a general-purpose computer is that sequences of such bits of data in memory can represent information of absolutely any kind. Numbers for example are typically represented in base 2 by sequences of 32 or more bits. Similarly, characters of text are usually represented by sequences of 8 or more bits. (The character “a” is typically 01100001.) Images are usually represented by bitmaps containing thousands or millions of bits, with each bit specifying for example whether a pixel at a particular location should, say, be black or white. Every possible location in memory has a definite address, independent of its contents. The address is typically represented as a number which itself can be stored in memory.

What makes possible essential universality in a practical computer is that the data which is stored in memory can be a program. At the lowest level, a program consists of a sequence of instructions to be executed by the processor. Any particular kind of processor is built to support a certain fixed set of possible kinds of instructions, each represented by a specific number or opcode. There are typically a few tens of possible instructions, each executed by a certain part of the circuit in the processor. A typical one of these instructions might add two numbers together; a program would specify which numbers to add by giving their addresses in memory.

What practical computers always basically do is to repeat millions of times a second a simple cycle, in which the processor fetches an instruction from memory, then executes the instruction. The address of the instruction to be fetched at each point is specified by the current value of the program counter—a number stored in memory that is incremented by the processor, or can be modified by instruction in the program. At any given time, there are usually several programs stored in the memory of a computer, all organized by an operating system program which determines when other programs should run. Devices like keyboards, mice and microphones convert input into data that is inserted into memory at certain fixed locations. The operating system periodically checks these locations, and if necessary runs programs to respond to the input that is given.

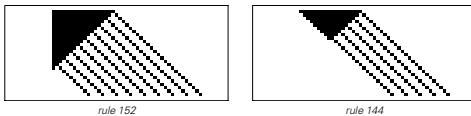
A crucial achievement in practical computing over the past several decades has been the creation of more and more sophisticated software. Often the programs that make up this software are several million instructions long. They usually contain many subprograms that perform parts of their task. Some programs are set up to perform very specific applications, say word processing. But an important class of programs are languages. A language provides a fixed set of constructs that allow one to specify computations. The set of instructions performed by the

processor in a computer constitutes a low-level “machine” language. In practice, however, programs are rarely written at such a low level. More often, languages like C, FORTRAN, Java or *Mathematica* are used. In these languages, each construct represents what is often a large number of machine instructions. There are two basic ways that languages can operate: compiled or interpreted. In a compiled language like C or FORTRAN, the source code of the program must always first be translated by a compiler program into object code that essentially consists of machine instructions. Once compiled, a program can be executed any number of times. In an interpreted language, each piece of input effectively causes a fixed subprogram to be executed to perform an operation specified by that input.

■ **Intuition from practical computing.** See page 872.

**Computations in Cellular Automata**

■ **Page 639 · Other examples.** Rule 152 and rule 144, which effectively compute  $\lceil n/2 \rceil$  and  $\lceil n/4 \rceil$ , respectively, are shown below with  $n = 18$  initial black cells.



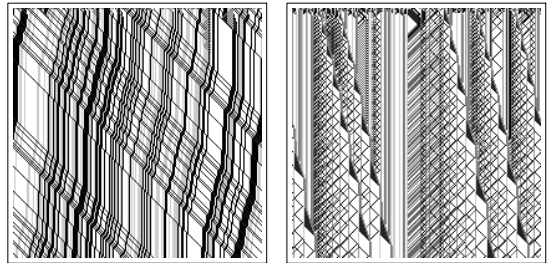
As discussed on page 989 rule 184 effectively determines whether its initial conditions correspond to a balanced sequence of open and close parentheses. (Rule 132 can be viewed as being like a syntax checker for a regular language; rule 184 for a context-free language.)

■ **Page 639 · Squaring cellular automaton.** The rules are  $\{(0, \_ , 3) \rightarrow 0, \{ \_ , 2, 3 \} \rightarrow 3, \{1, 1, 3\} \rightarrow 4, \{ \_ , 1, 4 \} \rightarrow 4, \{1|2, 3, \_ \} \rightarrow 5, \{p : (0|1), 4, \_ \} \rightarrow 7-p, \{7, 2, 6\} \rightarrow 3, \{7, \_ , \_ \} \rightarrow 7, \{ \_ , 7, p : (1|2) \} \rightarrow p, \{ \_ , p : (5|6), \_ \} \rightarrow 7-p, \{5|6, p : (1|2), \_ \} \rightarrow 7-p, \{5|6, 0, 0\} \rightarrow 1, \{ \_ , p : (1|2), \_ \} \rightarrow p, \{ \_ , \_ , \_ \} \rightarrow 0$  and the initial conditions consist of  $Append[Table[1, \{n\}], 3]$  surrounded by 0’s. The rules can be implemented using *GeneralCARule* as given on page 867. (See also page 1186.)

■ **Page 640 · Primes cellular automaton.** The rules are  $\{(13, 3, 13) \rightarrow 12, \{6, \_ , 4\} \rightarrow 15, \{10, \_ , 3|11\} \rightarrow 15, \{13, 7, \_ \} \rightarrow 8, \{13, 8, 7\} \rightarrow 13, \{15, 8, \_ \} \rightarrow 1, \{8, \_ , \_ \} \rightarrow 7, \{15, 1, \_ \} \rightarrow 2, \{ \_ , 1, \_ \} \rightarrow 1, \{1, \_ , \_ \} \rightarrow 8, \{2|4|5, \_ , \_ \} \rightarrow 13, \{15, 2, \_ \} \rightarrow 4, \{ \_ , 4, 8\} \rightarrow 4, \{ \_ , 4, \_ \} \rightarrow 5, \{ \_ , 5, \_ \} \rightarrow 3, \{15, 3, \_ \} \rightarrow 12, \{ \_ , x : (2|3|8), \_ \} \rightarrow x, \{ \_ , x : (11|12), \_ \} \rightarrow x-1, \{11, \_ , \_ \} \rightarrow 13, \{13, \_ , 1|2|3|5|6|10|11\} \rightarrow 15, \{13, 0, 8\} \rightarrow 15, \{14, \_ , 6|10\} \rightarrow 15, \{10, 0|9|13, 6|10\} \rightarrow 15, \{6, \_ , 6\} \rightarrow 0, \{ \_ , \_ , 10\} \rightarrow 9, \{6|10, 15, 9\} \rightarrow 14, \{ \_ , 6|10, 9|14|15\} \rightarrow 10, \{ \_ , 6|10, \_ \} \rightarrow 6, \{6|10, 15, \_ \} \rightarrow 13, \{13|14, \_ , 9|15\} \rightarrow 14, \{13|14, \_ , \_ \} \rightarrow 13, \{ \_ , \_ , 15\} \rightarrow 15, \{ \_ , \_ , 9|14\} \rightarrow 9, \{ \_ , \_ , \_ \} \rightarrow 0$

and the initial conditions consist of  $\{10, 0, 4, 8\}$  surrounded by 0’s. The right-hand region in the pattern grows like  $\sqrt{T}$ . (See also page 132.)

■ **Random initial conditions.** The pictures below show the squaring and primes cellular automata starting from random initial conditions. Note that for both systems the majority of cases in their rules are not used in the specific computations for which they were constructed. Changing these cases can lead to different behavior with random initial conditions.



■ **Efficiency of computations.** Present-day practical computers almost always process data in a basically sequential manner. Cellular automata, however, intrinsically operate in parallel, and can thus presumably perform at least some computations in fundamentally fewer steps. (Compare the discussion of P completeness on page 1149.)

■ **Minimal programs for sequences.** See page 1186.

**The Phenomenon of Universality**

■ **History of universality.** In Greek times it was noted as a philosophical matter that any single human language can be used to describe the same basic range of facts and processes. And with logic introduced as a way to formalize arguments (see page 1099), Gottfried Leibniz in the 1600s considered the idea of setting up a universal language based on logic that would provide a precise description analogous to a mathematical proof of any fact or process. But while Leibniz considered the possibility of checking his descriptions by machine, he apparently did not imagine setting up the analog of a computation in which something is explicitly generated from input that has been given.

The idea of having an abstract procedure that can be fed a range of different inputs had precursors in antiquity in the use of letters to denote objects in geometrical constructions, and in the 1500s in the introduction of symbolic formulas and algebraic variables. But the notion of abstract functions

in mathematics reached its modern form only near the end of the 1800s.

At the beginning of the 1800s practical devices such as the player pianos and the Jacquard loom were invented that could in effect be fed different inputs using analogs of punched cards. And in the 1830s Charles Babbage and Ada Lovelace noted that a similar approach could be used to specify the mathematical procedure to be followed by a mechanical calculating machine (see page 1107). But it was somehow assumed that the specification of the procedure must be done quite separately from the specification of the data to which the procedure was to be applied.

Starting in the 1880s attempts to build up both numbers and the operations of arithmetic from logic and set theory began to suggest that both data and procedures could potentially be described in common terms. And in the 1920s work by Moses Schönfinkel on combinators and by Emil Post on string rewriting systems provided fairly concrete examples of this.

In 1930 Kurt Gödel used the same basic idea to set up Gödel numbers to encode logical and other procedures as numbers. (Leibniz had in fact already done this for basic logic expressions in 1679.) But Gödel then took the crucial step of showing that the process of finding outputs from all such procedures could in effect be viewed as equivalent to following relations of logic and arithmetic—thus establishing that these relations are in a certain sense universal (see page 784). This fact, however, was embedded inside the rather technical proof of Gödel's Theorem, and it was at first not at all clear how specific it might be to the particular mathematical systems considered.

But in 1935 Alonzo Church constructed a system in lambda calculus that he showed could be made to emulate any other system in lambda calculus if given appropriate input, and in 1936 Alan Turing did the same thing for Turing machines. As discussed on page 1125, both Church and Turing argued that the systems they set up would be able to perform any reasonable computation. In both cases, their original motivation was to use this fact to construct an argument that the so-called decision problem (Entscheidungsproblem) of mathematical logic was undecidable (see page 1136). But Turing in particular gradually realized that his notion of universality could be applied to practical computers.

Turing's results were used in the 1940s—notably in the work of Warren McCulloch and Walter Pitts—as a basis for the assertion that electric circuit analogs of neural networks could achieve the sophistication of brains, and this appears to have influenced John von Neumann's thinking about the general programmability of electronic computers.

Nevertheless, by the late 1940s, practical computer engineering had also been led to the idea of storing programs—like data—electronically, and in the 1950s it became widely understood that general-purpose practical computers could be viewed as universal systems.

Many theoretical investigations of universality were made in the 1950s and 1960s, but gradually the emphasis shifted more towards issues of languages and algorithms.

■ **Universality in *Mathematica*.** As an example of how different primitive operations can be used to do the same computation, the following are a few ways that the factorial function can be defined in *Mathematica*:

```
f[n_] := n!
f[n_] := n f[n - 1]; f[1] = 1
f[n_] := Product[i, {i, n}]
f[n_] := Module[{t = 1}, Do[t = t i, {i, n}]; t]
f[n_] := Module[{t = 1, i}, For[i = 1, i ≤ n, i++, t *= i]; t]
f[n_] := Apply[Times, Range[n]]
f[n_] := Fold[Times, 1, Range[n]]
f[n_] := If[n == 1, 1, n f[n - 1]]
f[n_] := Fold[#2[#1] &, 1, Array[Function[t, #1 t] &, n]]
f = If[#1 == 1, 1, #1 #0[#1 - 1]] &
```

## A Universal Cellular Automaton

■ **Page 648 · Universal cellular automaton.** The rules for the universal cellular automaton are

```
{_, 3, 7, 18, _} → 12, {_, 5, 7 | 8, 0, _} → 12, {_, 3, 10, 18, _} → 16,
{_, 5, 10 | 11, 0, _} → 16, {_, 5, 8, 18, _} → 7, {_, 5, 14, 0 | 18, _} →
12, {_, _ 8, 5, _} → 7, {_, _ 14, 5, _} → 12, {_, 5, 11, 18, _} → 10,
{_, 5, 17, 0 | 18, _} → 16, {_, _ x: (11 | 17), 5, _} → x - 1,
{_, 0 | 9 | 18, x: (7 | 10 | 16), 3, _} → x + 1, {_, 0 | 9 | 18, 12, 3, _} →
14, {_, _ 0 | 9 | 18, 7 | 10 | 12 | 16, x: {3 | 5}} → 8 - x,
{_, _ 8 | 11 | 14 | 17, x: {3 | 5}} → 8 - x, {_, 13, 4, _ x: (0 | 18)} →
x, {18, _ 4, _} → 18, {_, _ 18, _ 4} → 18, {0, _ 4, _} → 0,
{_, _ 0, _ 4} → 0, {4, _ 0 | 18, 1, _} → 3, {4, _ _ _} → 4,
{_, _ 4, _ _} → 9, {4, 12, _ _} → 7, {4, 16, _ _} → 10,
{x: (0 | 18), _ 6, _} → x, {_, 2, 6, 15, x: (0 | 18)} → x, {_, 12 | 16,
6, 7, _} → 0, {_, 12 | 16, 6, 10, _} → 18, {_, 9, 10, 6, _} → 16,
{_, 9, 7, 6, _} → 12, {9, 15, 6, 7, 9} → 0, {9, 15, 6, 10, 9} → 18,
{9, _ 6, _} → 9, {_, 6, 7, 9, 12 | 16} → 12, {_, 6, 10, 9, 12 | 16} →
16, {12 | 16, 6, 7, 9, _} → 12, {12 | 16, 6, 10, 9, _} → 16,
{6, 13, _ _} → 9, {6, _ _ _} → 6, {_, _ 9, 13, 3} → 9,
{_, 9, 13, 3, _} → 15, {_, _ _ 15, 3} → 3, {_, 3, 15, 0 | 18, _} → 13,
{_, 13, 3, _ 0 | 18} → 6, {x: (0 | 18), 15, 9, _} → x,
{_, 6, 13, _} → 15, {_, 4, 15, _} → 13, {_, _ _ 15, 6} → 6,
{_, _ 2, 6, 15} → 1, {_, _ 1, 6, _} → 2, {_, 1, 6, _} → 9, {_, 3, 2,
_} → 1, {3, 2, _ _} → 3, {_, _ 3, 2, _} → 3, {_, 1, 9, 1, 6} → 6,
{_, _ 9, 1, 6} → 4, {_, 4, 2, _} → 1, {_, _ _ x: (3 | 5)} → x,
{_, _ 3 | 5, _ x: (0 | 18)} → x, {_, _ x: (1 | 2 | 7 | 8 | 9 | 10 | 11 |
12 | 13 | 14 | 15 | 16 | 17), _} → x, {_, _ 18, 7 | 10, 18} → 18,
{_, _ 0, 7 | 10, 0} → 0, {_, _ 0 | 18, _} → 9, {_, _ x, _} → x
```

where the numbers correspond to the icons shown in the main text according to



The block in the initial conditions for the universal cellular automaton corresponding to a cell with color  $a$  is given by

```
Flatten[Transpose[Join[{4, 18(1-a), 6}, Table[9,
{22r+1 - 3}], 10-3rtab]], Table[{9, 1}, {r}, 9, 13]]
```

where  $r$  is the range of the rule to be emulated ( $r = 1$  for elementary rules) and  $rtab$  is the list of outcomes for that rule (starting with the outcome for  $\{1, 1, (1) \dots\}$ ). In general, there are  $2^{2r+1}$  cases in the rule to be emulated; each block in the universal cellular automaton is  $2(2^{2r+1} + r + 1)$  cells wide, and each step in the rule to be emulated corresponds to  $(3r+2)2^{2r+1} + 3r^2 + 7r + 3$  steps in the evolution of the universal cellular automaton.

■ **Page 655 • More colors.** Given a rule that involves three colors and nearest neighbors, the following converts each case of the rule to a collection of cases for a rule with two colors:

```
CA3ToCA2[{a_, b_, c_} → d_] := Union[Flatten[Table[Thread[
Partition[Flatten[{l, a, b, c, r} /. coding], 1, 1][{2,
3, 4}]] → {d /. coding}], {l, 0, 2}, {r, 0, 2}], 2]]
coding = {0 → {0, 0, 0}, 1 → {0, 0, 1}, 2 → {0, 1, 1}}
```

The problem of encoding cells with several colors by blocks of black and white cells is related to standard problems in coding theory (see page 560). One approach is to use  $\{1, 1\}$  to indicate the boundary of each block, and then within each block to use all possible digit sequences which do not contain  $\{1, 1\}$ , as in the Fibonacci number system discussed on page 892. Note that the original rule with  $k$  colors and  $r$  neighbors involves  $\text{Log}[2, k^{2r+1}]$  bits of information; the two-color rule that emulates it involves  $\text{Log}[2, 2^{2^{2r+1}}]$  bits. As a result, the minimum possible  $s$  for  $k=3, r=1$  is about 2.2; in the specific example shown in the main text it is 5.

**Emulating Other Systems with Cellular Automata**

■ **Page 657 • Mobile automata.** Given a mobile automaton with rules in the form used on page 887, a cellular automaton which emulates it can be constructed using

```
MAToCA[rules_] :=
Append[Flatten[Map[g, rules]], {_, _, x_, _, _} → x]
g[{a_, b_, c_} → {d_, e_}] := {_, a, b+2, c, _} → d, If[e == 1,
{a, b+2, c, _} → c+2, {_, _, a, b+2, c} → a+2]
```

This specific definition assumes that the mobile automaton has two possible colors for each cell; it yields a cellular automaton with four possible colors for each cell. An initial

condition with a single 2 surrounded by 0's corresponds to all cells being white in the mobile automaton.

■ **Page 658 • Turing machines.** Given any Turing machine with rules in the form used on page 888 and  $k$  possible colors for each cell, a cellular automaton which emulates it can be constructed using

```
TMTToCA[rules_, k_ : 2] :=
Flatten[Map[g[#, k] &, rules], {_, x_, _} → x]]
g[{s_, a_} → {sp_, ap_, d_}, k_] := {If[d == 1, Identity,
Reverse][{k s + a, x_, _}] → k sp + x, {_, k s + a, _} → ap}
```

If the Turing machine has  $s$  states for its head, then the cellular automaton has  $k(s+1)$  colors for each cell. An initial condition with a single cell of color  $k$  surrounded by 0's corresponds to being in state 1 with a blank tape in the Turing machine.

■ **Page 659 • Substitution systems.** Given a substitution system with rules in the form such as  $\{1 \rightarrow \{0\}, 0 \rightarrow \{0, 1\}\}$  used on page 889, the rules for a cellular automaton which emulates it are obtained from

```
SSToCA[rules_] := {{b, b, p[x_, _]} → s[x],
{_, s[v : {0|1}], p[x_, _]} → p[v, x], {_, p[_, y_, _]} → s[y],
{_, s[v : {0|1}], _m} → m[v], {s[0|1], m[v : {0|1}], _} →
s[v], {b, m[v : {0|1}], _} → r[v], {_, r[v : {0|1}], _} →
(Replace[v, rules] /. {x_} → s[x], {x_, y_} → p[x, y])],
{_, s[v : {0|1}], _} → r[v], {_, b, _} → m[b],
{s[0|1], m[b], _} → b, {_, v_, _} → v}
```

where specific values for cells can be obtained from

```
{b → 0, s[0] → 1, m[0] → 2, p[0, 0] → 3,
r[0] → 4, p[0, 1] → 5, p[1, 0] → 6, r[1] → 7,
p[1, 1] → 8, m[1] → 9, m[b] → 10, s[1] → 11}
```

An initial condition consisting of a single element with color  $i$  in the substitution system is represented by  $m[i]$  surrounded by  $b$ 's in the cellular automaton. The specific definition given above works for neighbor-independent substitution systems whose elements have two possible colors, and in which each element is replaced at each step by at most two new elements.

■ **Page 660 • Sequential substitution systems.** Given a sequential substitution system with rules in the form used on page 893, the rules for a cellular automaton which emulates it can be obtained from

```
SSSToCA[rules_] := Flatten[{{v[_, _, _], u, _} → u, {_, v[rn_,
x_, _], u} → r[rn+1, x], {_, v[_, x_, _]} → x, MapIndexed[
With[{rn = #2[[1]], rs = #1[[1]], rr = #1[[2]]}, {If[Length[rs] ==
1, {u, r[rn, First[rs]], _} → q[0, rr], {u, r[rn, First[rs]], _} →
v[rn, First[rs], Take[rs, 1]]}, {u, r[rn, x_, _]} → v[rn, x, {}],
{v[rn, _, Drop[rs, -1]], Last[rs], _} → q[Length[rs]-1, rr],
Table[{v[rn, _, Flatten[Take[rs, i-1]]], rs[[i]], _} → v[
rn, rs[[i]], Take[rs, i], {i, Length[rs]-1, 1, -1}], {v[rn, _, _],
y_, _} → v[rn, y, {}]}] &, rules /. s → List], {_, q[0, {x_, _}],
```

```

_} → q[0, {x}], {_, q[0, {x_}], _} → r[1, x], {_, q[0, {}], x_} →
r[1, x], {_, q[_, {_, x_}], _} → x, {_, q[_, {}], x_} → x,
{_, x_, q[0, _]} → x, {_, _, q[n_, {}]} → q[n-1, {}],
{_, _, q[n_, {x_}], _} → q[n-1, {x}], {q[_, {}], _} → w,
{q[0, {_, x_}], p[y_, _], _} → p[x, y], {q[0, {_, x_}], y_, _} →
p[x, y], {p[_, x_], p[y_, _], _} → p[x, y], {p[_, x_], u, _} → x,
{p[_, x_], y_, _} → p[x, y], {_, p[x_, _], _} → x, {w, u, _} → u,
{w, x_, _} → w, {_, w, x_} → x, {_, r[m_, x_], _} → x,
{_, u, r[_, _]} → u, {_, x_, r[m_, _]} → r[m, x], {_, x_, _} → x]]

```

The initial condition is obtained by applying the rule  $s[x_, y_] \rightarrow \{r[1, x], y\}$  and then padding with  $u$ 's.

■ **Page 661 · Register machines.** Given the program for a register machine in the form used on page 896, the rules for a cellular automaton that emulates it can be obtained from

```

g[i[1], p_, m_] :=
  {{_, p, _} → p + 1, {_, 0, p} → m + 2, {_, _} → m + 3}
g[i[2], p_, m_] :=
  {{_, p, _} → p + 1, {p, 0, _} → m + 5, {p, _} → m + 6}
gd[1, q_, p_, m_] := {{m + 2 | m + 3, p, _} → q, {m + 1,
  p, _} → p, {0, p, _} → p + 1, {_, m + 2 | m + 3, p} → m + 1}
gd[2, q_, p_, m_] := {{_, p, m + 5 | m + 6} → q, {_, p,
  m + 4} → p, {_, p, 0} → p + 1, {p, m + 5 | m + 6, _} → m + 4}
RMToCA[prog_] := With[{m = Length[prog]], Flatten[
  {MapIndexed[g[#1, First[#2], m] &, prog], {{0, 0 | m + 1,
    m + 3} → m + 2, {0, m + 1, _} → 0, {0, 0, m + 1} → 0,
    {_, _, x : (m + 1 | m + 3)} → x, {_, m + 1 | m + 3, _} → m + 2,
    {m + 6, 0 | m + 4, 0} → m + 5, {_, m + 4, 0} → 0,
    {m + 4, 0, 0} → 0, {x : (m + 4 | m + 6), _} → x,
    {_, m + 4 | m + 6, _} → m + 5, {_, x_, _} → x}}]}

```

If  $m$  is the length of the register machine program, then the resulting cellular automaton has  $m + 7$  possible colors for each cell. If the initial numbers in the two registers are  $a$  and  $b$ , then the initial conditions for the cellular automaton are  $\text{Join}[\text{Table}[m + 2, \{a\}], \{1\}, \text{Table}[m + 5, \{b\}]]$  surrounded by 0's.

■ **Page 661 · Multiplication systems.** The rules for the cellular automaton shown here are

```

{_, 0, 3 | 8} → 5, {_, 0, 2 | 7} → 8, {_, 1, 4 | 9} → 9,
{_, 1, 3 | 8} → 4, {_, 1, 2 | 7} → 8, {_, 10, 4 | 9} → 3,
{_, 10, 3 | 8} → 7, {_, 10, 2 | 7} → 2, {5 | 6, 1, 0} → 9,
{5 | 6, 10, 0} → 3, {5 | 6, 1, _} → 6, {5 | 6, 10, _} → 5,
{_, 2 | 3 | 4 | 5, _} → 10, {_, 6 | 7 | 8 | 9, _} → 1, {_, x_, _} → x}

```

and the initial condition consists of a single 3 surrounded by 0's. The idea used is that multiplication by 3 can be achieved by scanning digits from right to left, adding to each digit the value of the digit on its immediate right, as well as a carry that can propagate any distance but cannot be larger than 1. Note that as discussed on page 614 multiplication by some multipliers in some bases (such as by 3 in base 6) can be achieved by a single step in the evolution of a suitable cellular automaton. After  $t$  steps, the width of the pattern shown here is about  $\text{Sqrt}[\text{Log}[2, 3]t]$ . (See also page 119.)

■ **Continuous systems.** See page 1128.

■ **Page 662 · Logic circuits.** The rules for the cellular automaton shown here are

```

{{0, 1, 1 | 3} → 1, {0, 3, 3} → 3, {1, 0, 0 | 1 | 3} → 1,
  {1, 1, 3} → 4, {1, 3, 0} → 3, {1, 3, 3} → 2, {2, 1, 3} → 3,
  {2, 3, 0} → 2, {2, 0, _} → 4, {3, 3, 0} → 3, {4, 0, 0 | 1 | 2 | 4} → 2,
  {4, 3, 3} → 3, {4, 1, 3} → 1, {4, 3, 0} → 4, {_, _} → 0}

```

The initial conditions are given by

```

Flatten[Block[{And, Or}, Map[{0, 2 (# + 1)} &, expr, {-1}] //
  {! x_ :> {0, x, 0}, And[x_] :> {0, 0, 1, 0, x, 1, 3, 0, 0},
  Or[x_] :> {0, 0, 1, 0, x, 0, 1, 3, 0}}]]

```

and in terms of these initial conditions the cellular automaton must be run for  $\text{Length}[\text{list}] / \{0, x_-\} \rightarrow \{x\} - 1$  steps in order to find the result.

■ **Page 663 · RAM.** The rules for the cellular automaton shown here are

```

{{2, 4 | 8, 2 | 11, _} → 2, {11 | 10, 4 | 8, 2 | 11, _} → 11,
  {2, 4 | 8, _} → 10, {11 | 10, 4 | 8, _} → 2,
  {2, 0, _} → 2, {11, 0, _} → 11,
  {3 | 7 | 6, _} → 1, {x : (3 | 7 | 6), _} → x,
  {_, _} → 6, 4, 10} → 5, {_, _} → 6, 8, 10} → 9, {_, 3, _} → 4,
  {_, 7, _} → 10, 10, _} → 8, {_, _} → 1, x : (5 | 9)} → x, {1, _} → 1,
  {_, _} → 1, _} → 1, {_, _} → 1} → 1, {_, _} → x : (4 | 8 | 0), _} → x}

```

The initial conditions are divided into two parts: instructions on the left and memory on the right. Given a list of 0 and 1 values for successive memory locations, the right-hand initial conditions are  $\text{Flatten}[\text{list} /. \{1 \rightarrow \{8, 1\}, 0 \rightarrow \{4, 1\}\}]$ . To access location  $n$  the left-hand initial conditions must contain  $\text{Flatten}[\{0, i, \text{IntegerDigits}[n, 2]\} /. \{1 \rightarrow \{0, 11\}, 0 \rightarrow \{0, 2\}\}]$  inserted in a repetitive  $\{0, 1\}$  background. If  $i$  is 7, a 1 will be written to location  $n$ ; if it is 3, a 0 will be written; and if it is 6, the contents of location  $n$  will be read and sent back to the left.

## Emulating Cellular Automata with Other Systems

■ **Page 664 · Mobile automata.** Given the rules for an elementary cellular automaton in the form used on page 867, the following will construct a mobile automaton which emulates it:

```

vals = {x, p[0], q[0, 0], q[0, 1], q[1, 0], q[1, 1], p[1]}
CAToMA[rules_] := Table[#, Replace[#, {{q[a_, b_], p[c_],
  p[d_]} :> {q[c, {a, c, d} /. rules], 1}, {q[a_, b_], p[c_], x} :>
  {q[c, {a, c, 0} /. rules], 1}, {q[_, _], x, x} :> {p[0], -1},
  {q[_, _], q[_, a_], p[_, _]} :> {p[a], -1}, {x, q[_, a_], p[_, _]} :>
  {p[a], -1}, {x, x, p[_, _]} :> {q[0, 0], 1}, {_, _} :>
  {x, 0}}] &][vals][IntegerDigits[i, 7, 3] + 1]], {i, 0, 7^3 - 1}]

```

The ordering in  $\text{vals}$  defines a mapping of symbolic cell values onto colors. Given a list of initial cell colors for the cellular automaton, the initial conditions for the mobile automaton are given by  $\text{Flatten}[\{p[0], \text{Map}[p, \text{list}], p[0]\}]$  surrounded by  $x$ 's, with the active cell being placed initially just before the first  $p[0]$ .

■ **Page 665 · Turing machines.** Given the rules for an elementary cellular automaton in the form used on page 867, the following will construct a Turing machine which emulates it:

```
CAToTM[rules_] :=
  {{q[a_, b_], c : {0 | 1}} → {q[b, c], {a, b, c} /. rules, 1},
  {q[_ , _], x} → {p[0], 0, -1}, {p[a_], b : {0 | 1}} →
  {p[b], a, -1}, {p[_], x} → {q[0, 0], 0, 1}}
```

Given a list of initial cell colors for the cellular automaton, the initial tape for the Turing machine consists of `Join[{0, 0}, list, {0, 0}]` surrounded by `x`'s, with the head of the Turing machine on the first `0` in state `q[0, 0]`.

For specific cellular automata it is often possible to construct smaller Turing machines, as on pages 707 and 1119. By combining identical cases in rules and writing rules as compositions of ones with smaller neighborhoods one can for example readily construct Turing machines with 4 states and 3 colors that emulate 166 of the elementary cellular automata.

■ **Page 667 · Sequential substitution systems.** Given the rules for an elementary cellular automaton in the form used on page 867, the following will construct a sequential substitution system which emulates it:

```
CAToSSS[rules_] := Join[rules /.
  {{a_, b_, c_} → d_} → {1, 2a, 2b, 2c} → {2d, 1, 2b, 2c},
  {{1, 0, 0} → {0, 0}, {0} → {1, 0, 0, 0}}]
```

The initial condition `{0, 0, 2, 0, 0}` for the sequential substitution system corresponds to a single black cell surrounded by white cells in the cellular automaton.

■ **Page 667 · Tag systems.** Given the rules for an elementary cellular automaton in the form used on page 867, the following will construct a tag system which emulates it:

```
CAToTS[rules_] := {2, {{s[x_], s[y_]} →
  {d[x, y], d[x, y]}, {d[w_, x_], d[y_, z_]} →
  {s[{w, x, y} /. rules], s[{x, y, z} /. rules]},
  {s[x_], d[y_, z_]} → {s[0], s[0]}, s[{0, y, z} /. rules]},
  {d[x_, y_], s[z_]} → {s[{x, y, 0} /. rules], s[0], s[0]}}]
```

The initial condition for the tag system that corresponds to a single black cell in the cellular automaton is `{s[0], s[0], s[1], s[0], s[0]}`. Given a list of all steps in the evolution of the tag system, `Cases[list, {_s}]` picks out successive steps in the cellular automaton evolution.

■ **Page 668 · Symbolic systems.** Given the rules for an elementary cellular automaton in the form used on page 867 (with `{0, 0, 0} → 0`), the following will construct a symbolic system which emulates it:

```
Flatten[{Array[p[x_][#1][#2][#3] →
  p[x][##] /. rules][#2][#3] &, {2, 2, 2}, 0] /. {0 → p, 1 → q},
  {r[x_] → p[r[p][p]][x], p[x_][p][p][r] → x[p][p][r]}}
```

The initial condition for the symbolic system is given by

```
Fold[#1[#2] &, r[p][p], init /. {0 → p, 1 → q}][p][p][r]
```

Step `t` in the cellular automaton corresponds to step `t + Length[init] + 3` in the symbolic system.

Note that the succession of states shown here depends on the detailed order in which rules are applied (see page 898). It is also possible to construct symbolic systems with the so-called confluence property, in which results from any fixed number of steps of cellular automaton evolution can be found by applying rules in any possible order (see page 1036).

■ **Page 669 · Cyclic tag systems.** From a tag system which depends only on its first element, with rules given as in the note below, the following constructs a cyclic tag system emulating it:

```
TS1ToCT[{n_, subs_}] := With[{k = Length[subs]},
  Join[Map[v[Last[#], k] &, subs], Table[#, {k (n - 1)}]]]
u[l_, k_] := Table[If[j == i + 1, 1, 0], {j, k}]
v[list_, k_] := Flatten[Map[u[#, k] &, list]]
```

The initial condition for the tag system can be converted using `v[list, k]`. The list representing the complete history of the resulting cyclic tag system can then be interpreted using

```
Map[Map[Position[#, 1][[1, 1]] - 1 &, Partition[#, k]] &,
  Take[history, {1, -1, nk}]]
```

This construction is relevant to the proof of the universality of rule 110 starting on page 678.

■ **Page 669 · Multicolor Turing machines.** Given rules in the form on page 888 for a Turing machine with `s` states and `k` colors the following yields an equivalent Turing machine with `With[{c = Ceiling[Log[2, k]]}, ((32c) + 2c - 7)s]` states (always less than `6.03ks`) and 2 colors:

```
TMTToTM2[rule_, s_, k_] := (# /. MapIndexed[
  #1 → First[#2] &, Union[Map[#[[1, 1]] &, #]]] &)[
  With[{b = Ceiling[Log[2, k]] - 1}, Flatten[Table[
    {Table[{Table[{{m, i, n, d}, c] → {{m, Mod[i, 2n-1], n - 1,
      d}, Quotient[i, 2n-1], 1}, {n, 2, b}, {i, 0, 2n-1 - 1}], Table[
      {{m, i, 1, d}, c} → {{m, -1, 1, d}, i, d}, {i, 0, 1}], Table[
      {{m, -1, n, d}, c} → {{m, -1, n + 1, d}, c, d}, {n, b - 1}],
      {{m, -1, b, d}, c} → {{0, 0, m}, c, d}}, {d, -1, 1, 2}],
    Table[{{i, n, m}, c] → {{i + 2c, n + 1, m}, c, -1},
    {n, 0, b - 1}, {i, 0, 2n-1 - 1}], With[{r = 2b}, Table[
      If[i + r c ≥ k, {}], Cases[rule, {{m, i + r c} → {x_, y_, z_}] →
      {{i, b, m}, c} → {{x, Mod[y, r], b, z}, Quotient[y, r,
        1}], {i, 0, r - 1}]]], {m, s}, {c, 0, 1}]]]]]
```

Some of these states are usually unnecessary, and in the main text such states have been pruned. Given an initial condition `{i, list, n}` the initial condition for the 2-color Turing machine is

```
With[{b = Ceiling[Log[2, k]]},
  {i, Flatten[IntegerDigits[list, 2, b]], b n}]
```

■ **Page 670 · One-element-dependence tag systems.** Writing the rule  $\{3, \{0, \_ \_ \} \rightarrow \{0, 0\}, \{1, \_ \_ \} \rightarrow \{1, 1, 0, 1\}\}$  from page 895 as  $\{3, \{0 \rightarrow \{0, 0\}, 1 \rightarrow \{1, 1, 0, 1\}\}$  the evolution of a tag system that depends only on its first element is obtained from

```
TS1EvolveList[rule_, init_, t_]:=
  NestList[TS1Step[rule, #] &, init, t]
TS1Step[{n_, subs_}, {}] = {}
TS1Step[{n_, subs_}, list_] :=
  Drop[Join[list, First[list]/. subs], n]
```

Given a Turing machine in the form used on page 888 the following will construct a tag system that emulates it:

```
TMTToTS1[rules_] :=
  {2, Union[Flatten[rules /. {{l_, u_} -> {j_, v_, r_} ->
    {Map[#[] -> {#[i, 1], #[i, 0]} &, {a, b, c, d}], If[r == 1,
    {a[i, u] -> {a[j], a[j]}, b[i, u] -> Table[b[j], {4}], c[i, u] ->
    Flatten[{Table[b[j], {2 v}], Table[c[j], {2 - u}]}],
    d[i, u] -> {d[j]}], {a[i, u] -> Table[a[j], {2 - u}],
    b[i, u] -> {b[j]}, c[i, u] -> Flatten[{c[j], c[j]},
    Table[d[j], {2 v}]}], d[i, u] -> Table[d[j], {4}]}]}]}
```

A Turing machine in state  $i$  with a blank tape corresponds to initial condition  $\{a[i], a[i], c[i]\}$  for the tag system. The configuration of the tape on each side of the head in the Turing machine evolution can be obtained from the tag system evolution using

```
Cases[history, x : {a[_], ___} ->
  Apply[{#, Reverse[#2]}] &, Map[
  Drop[IntegerDigits[Count[x, #], 2], -1] &, {_b, _d}]]]
```

■ **Page 672 · Register machines.** Given the rules for a Turing machine in the form used on page 888, a register machine program to emulate the Turing machine can be obtained by techniques analogous to those used in compilers for practical computer languages. Here *TMCompile* creates a program segment for each element of the Turing machine rule, and *TMTtoRM* resolves addresses and links the segments together.

```
TMTtoRM[rules_] := Module[{segs, adrs}, segs =
  Map[TMCompile, rules]; adrs = Thread[Map[First, rules] ->
  Drop[FoldList[Plus, 1, Map[Length, segs]], -1]];
  MapIndexed[#1 /. {dr[r_, n_] -> d[r, n + First[#2]],
  dm[r_, z_] -> d[r, z /. adrs]} &, Flatten[segs]]]
TMCompile[_ -> z : {_, _} 1] := f[z, {1, 2}]
TMCompile[_ -> z : {_, _} -1] := f[z, {2, 1}]
f[{s_, a_, _}, {ra_, rb_}] := Flatten[{i[3], dr[ra, -1],
  dr[3, 3], i[ra], i[ra], dr[3, -2], If[a == 1, i[ra], {}], i[3],
  dr[rb, 5], i[rb], dr[3, -1], dr[rb, 1], dm[rb, {s, 0}],
  dr[rb, -6], i[rb], dr[3, -1], dr[rb, 1], dm[rb, {s, 1}]}]}
```

A blank initial tape for the Turing machine corresponds to initial conditions  $\{1, \{0, 0, 0\}\}$  for the register machine. (Assuming that the Turing machine starts in state 1, with a 0 under its head, other initial conditions can be encoded just by taking the values of cells on the left and right to give the digits of the numbers that are initially in the first two

registers.) Given the list of successive configurations of the register machine, the steps that correspond to successive steps of Turing machine evolution can be obtained from

```
(Flatten[Partition[Complement[#, # - 1], 1, 2]] &][
  Position[list, {_, {_, _}, 0}]]]
```

The program given above works for Turing machines with any number of states, but it requires some simple extensions to handle more than two possible colors for each cell. Note that for a Turing machine with  $s$  states, the length of the register machine program generated is between  $34s$  and  $36s$ .

■ **Register machines with many registers.** It turns out that a register machine with any number of registers can always be emulated by a register machine with just two registers. The basic idea is to encode the list of values of all the registers in the multiregister machine in the single number given by

```
RMEncode[list_] :=
  Product[Prime[j]^list[[j]], {j, Length[list]}]
```

and then to have this number be the value at appropriate steps of the first register in the 2-register machine. The program in the multiregister machine can be converted to a program for the 2-register machine according to

```
RMTtoRM2[prog_] :=
  Module[{segs, adrs}, segs = MapIndexed[seg, prog];
  adrs = FoldList[Plus, 1, Map[Length, segs]];
  MapIndexed[#1 /. {ds[r_, s_] -> d[r, adrs[[s]]],
  dr[r_, j_] -> d[r, j + First[#2]]] &, Flatten[segs]]]
seg[i[r_], {a_}] := With[{p = Prime[r]},
  Flatten[{Table[i[2], {p}], dr[1, -p], i[1],
  dr[2, -1], Table[dr[1, 1], {p + 1}]}]}]
seg[d[r_, n_], {a_}] := With[{p = Prime[r]}, Flatten[{i[2], dr[
  1, 5], i[1], dr[2, -1], dr[1, 1], ds[1, n], Table[{If[m == p - 1,
  ds[1, a], dr[1, 3 p + 2 - m]}], Table[i[1], {p}], dr[2, -p],
  Table[dr[1, 1], {2 p - m - 1}], ds[1, a + 1]}, {m, p - 1}]}]}
```

The initial conditions for the 2-register machine are given by  $\{1, \{RMEncode[list], 0\}\}$  and the results corresponding to each step in the evolution of the multiregister machine appear whenever register 2 in the 2-register machine is incremented from 0.

■ **Computations with register machines.** As an example, the following program for a 3-register machine starting with initial condition  $\{n, 0, 0\}$  will compute  $\{\text{Round}[\sqrt{n}], 0, 0\}$ :

```
{d[1, 4], i[1], d[1, 15], i[2], d[1, 6], d[1, 11], i[1],
  d[2, 7], d[3, 7], d[1, 15], d[3, 4], i[3], d[2, 12], d[3, 4]}
```

■ **Page 673 · Arithmetic systems.** Given the program for a register machine with  $nr$  registers in the form on page 896, an arithmetic system which emulates it can be obtained from

```
RMtoAS[prog_, nr_] := With[{p = Length[prog], g =
  Product[Prime[j], {j, nr}]}, {pg, Sort[Flatten[MapIndexed[
  With[{n = First[#2] - 1}, #1 /. {i[r_] -> Table[n + j p ->
  (1 + n + Prime[r](-n + #) &), {j, 0, g - 1}], d[r_, k_] ->
  Table[n + j p -> If[Mod[j, Prime[r]] == 0, -1 + k + (-n +
  #)/Prime[r] &, # + 1] &, {j, 0, g - 1}]}]}]} &, prog]]]
```

The rules for the arithmetic system are represented so that the system from page 122 becomes for example  $\{2, \{0 \rightarrow (3\# / 2 \&), 1 \rightarrow (3(\# + 1) / 2 \&)\}$ . If the register machine starts at instruction  $n$  with values  $regs$  in its registers, then the corresponding arithmetic system starts with the number  $n + \text{Table}[\text{Prime}[i]^{\text{reg}[i]}, \{i, nr\}]p - 1$  where  $p = \text{Length}[\text{prog}]$ . The evolution of the arithmetic system is given by

```
ASEvolveList[{n_, rules_}, init_, t_] :=
  NestList[(Mod[# , n] /. rules)[#] &, init, t]
```

Given a value  $m$  obtained in the evolution of the arithmetic system, the state of the register machine to which it corresponds is

```
{Mod[m, p] + 1, Map[Last, FactorInteger[
  Product[Prime[i], {i, nr}] Quotient[m, p]]] - 1}
```

Note that it is possible to have each successive step involve only multiplication, with no addition, at the cost of using considerably larger numbers overall.

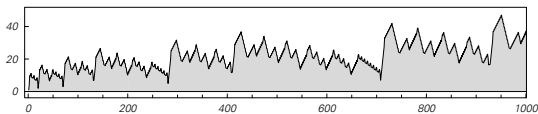
■ **History.** The correspondence between arithmetic systems and register machines was established (using a slightly different approach) by Marvin Minsky in 1962. Additional work was done by John Conway, starting around 1971. Conway considered fraction systems based on rules of the form

```
FSEvolveList[fracs_, init_, t_] :=
  NestList[First[Select[fracs #, IntegerQ, 1]]] &, init, t]
```

With the choice

```
fracs = {17/91, 78/85, 19/51, 23/38, 29/33, 77/29, 95/
  23, 77/19, 1/17, 11/13, 13/11, 15/14, 15/2, 55/1}
```

starting at 2 the result for  $\text{Log}[2, \text{list}]$  is as shown below, where  $\text{Rest}[\text{Log}[2, \text{Select}[\text{list}, \text{IntegerQ}[\text{Log}[2, \#]]] \&]]$  gives exactly the primes.



(Compare the discussion of universality in integer equations on page 786.)

■ **Multway systems.** It is straightforward to emulate a  $k$ -color multway system with a 2-color one, just by encoding successive colors by strings like "AAABBB", "AAABAB" and "AABABB" that have no overlaps. (Compare page 1033.)

### The Rule 110 Cellular Automaton

■ **History.** The fact that 1D cellular automata can be universal was discussed by Alvy Ray Smith in 1970—who set up an 18-color nearest-neighbor cellular automaton rule capable of emulating Marvin Minsky's 7-state 4-color universal Turing machine (see page 706). (Roger Banks also mentioned in 1970

a 17-color cellular automaton that he believed was universal.) But without any particular reason to think it would be interesting, almost nothing was done on finding simpler universal 1D cellular automata. In 1984 I suggested that cellular automata showing what I called class 4 behavior should be universal—and I identified some simple rules (such as  $k = 2, r = 2$  totalistic code 20) as explicit candidates. A piece published in *Scientific American* in 1985 describing my interest in finding simple 1D universal cellular automata led me to receive a large number of proofs of the fact (already well known to me) that 1D cellular automata can in principle emulate Turing machines. In 1989 Kristian Lindgren and Mats Nordahl constructed a 7-color nearest-neighbor cellular automaton that could emulate Minsky's 7,4 universal Turing machine, and showed that in general a rule with  $s + k + 2$  colors could emulate an  $s$ -state  $k$ -color Turing machine (compare page 658). Following my ideas about class 4 cellular automata I had come by 1985 to suspect that rule 110 must be universal. And when I started working on the writing of this book in 1991, I decided to try to establish this for certain. The general outline of what had to be done was fairly clear—but there were an immense number of details to be handled, and I asked a young assistant of mine named Matthew Cook to investigate them. His initial results were encouraging, but after a few months he became increasingly convinced that rule 110 would never in fact be proved universal. I insisted, however, that he keep on trying, and over the next several years he developed a systematic computer-aided design system for working with structures in rule 110. Using this he was then in 1994 successfully able to find the main elements of the proof. Many details were filled in over the next year, some mistakes were corrected in 1998, and the specific version in the note below was constructed in 2001. Like most proofs of universality, the final proof he found is conceptually quite straightforward, but is filled with many excruciatingly elaborate details. And among these details it is certainly possible that a few errors still remain. But if so, I believe that they can be overcome by the same general methods that have been used in the proof so far. Quite probably a somewhat simpler proof can be given, but as discussed on page 722 it is essentially inevitable that proofs of universality must be at least somewhat complicated. In the future it should be possible to give a proof in a form that can be checked completely by computer. (The initial conditions in the note below quite soon become too large to run explicitly on any existing computer.) And in addition, with sufficient effort, I believe one should be able to construct an automated system that will allow many universality proofs of this general kind to be found almost entirely by computer (compare page 810).



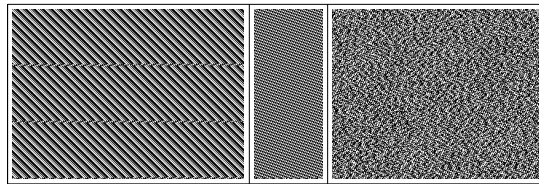
■ **Page 683 · Initial conditions.** The following takes the rules for a cyclic tag system in the form used on page 895 (with the restrictions in the note below), together with the initial conditions for the tag system, and yields a specification of initial conditions in rule 110 which will emulate it. This specification gives a list of three blocks  $\{b_1, b_2, b_3\}$  and the final initial conditions consist of an infinite repetition of  $b_1$  blocks, followed by  $b_2$ , followed by an infinite repetition of  $b_3$  blocks. The  $b_1$  blocks act like “clock pulses”,  $b_2$  encodes the initial conditions for the tag system and the  $b_3$  blocks encode the rules for the tag system.

```
CTToR110[rules_ /;
  Select[rules, Mod[Length[#], 6] # 0 &] == {}, init_] :=
Module[{g1, g2, g3, nr = 0, x1, y1, sp}, g1 = Flatten[
  Map[If[# == {}, {{2}}, {{1, 3, 5 - First[#]}}, Table[
    {4, 5 - #][[n]], {n, 2, Length[#]}]]] &, rules] /. a_Integer ->
  Map[{d[#][1], #][2]], s[#][3]]] &, Partition[c[a], 3]], 4];
g2 = g1 = MapThread[If[#1 == #2 == {d[22, 11], s3}, {d[
  20, 8], s3}, #1] &, {g1, RotateRight[g1, 6]}]; While[Mod[
  Apply[Plus, Map[#][1, 2]] &, g2]], 30] # 0, nr++; g2 = Join[
  g2, g1]; y1 = g2[[1, 1, 2]] - 1; If[y1 < 0, y1 += 30]; Cases[
  Last[g2][2]], s[d[x_, y1], _., a_] -> {x1 = x + Length[a]}];
g3 = Fold[sadd, {d[x1, y1], {}}, g2]; sp = Ceiling[5 Length[
  g3[2]]/(28 nr + 2)]; Join[Fold[sadd, {d[17, 1], {}},
  Flatten[Table[{d[sp 28 + 6, 1], s[5]}, {d[398, 1], s[5]},
  {d[342, 1], s[5]}, {d[370, 1], s[5]}], {3}, 1]]][2]], bg[
  4, 1]], Flatten[Join[Table[bgi, {sp 2 + 1 + 24 Length[init]}],
  init] /. {0 -> init0, 1 -> init1}, bg[1, 9], bg[6, 60 - g2[[1, 1, 1]] +
  g3[[1, 1]] + If[g2[[1, 1, 2]] < g3[[1, 2]], 8, 0]]]]]
```

```
s[1] = struct[{3, 0, 1, 10, 4, 8}, 2];
s[2] = struct[{3, 0, 1, 1, 619, 15}, 2];
s[3] = struct[{3, 0, 1, 10, 4956, 18}, 2];
s[4] = struct[{0, 9, 10, 4, 8}];
s[5] = struct[{5, 0, 9, 14, 1, 1}];
{c[1], c[2]} = Map[Join[{22, 11, 3, 39, 3, 1}, #] &,
  {{63, 12, 2, 48, 5, 4, 29, 26, 4, 43, 26, 4, 23, 3, 4, 47, 4, 4},
  {87, 6, 2, 32, 2, 4, 13, 23, 4, 27, 16, 4}}];
{c[3], c[4], c[5]} = Map[Join[#, {4, 17, 22, 4,
  39, 27, 4, 47, 4, 4}], {{17, 22, 4, 23, 24, 4, 31, 29},
  {17, 22, 4, 47, 18, 4, 15, 19}, {41, 16, 4, 47, 18, 4, 15, 19}}];
{init0, init1} = Map[IntegerDigits[216 (# + 432 1049), 2] &,
  {246005560154658471735510051750569922628065067661,
  1043746165489466852897089830441756550889834709645}];
bgi = IntegerDigits[9976, 2]
bg[s_, n_] := Array[bgi[[1 + Mod[# - 1, 14]]] &, n, s]
ev[s[d[x_, y_], pl_, pr_, b_]] := Module[{r, pl1, pr1}, r =
  Sign[BitAnd[2^ListConvolve[{1, 2, 4}, Join[bg[pl - 2, 2], b,
  bg[pr, 2]]], 110]]; pl1 = (Position[r - bg[pl + 3, Length[r]],
  1] - 1) /. {} -> {{Length[r]}}][1, 1]; pr1 = Max[pl1,
  (Position[r - bg[pr + 5 - Length[r], Length[r]], 1] - 1) /. {} ->
  {{1}}][1, 1]; s[d[x + pl1 - 2, y + 1], pl1 + Mod[pl + 2, 14],
  1 + Mod[pr + 4, 14] + pr1 - Length[r], Take[r, {pl1, pr1}]]]
```

```
struct[{x_, y_, pl_, pr_, b_, bl_}, p_Integer : 1] := Module[
  {gr = s[d[x, y], pl, pr, IntegerDigits[b, 2, bl]], p2 = p + 1},
  Drop[NestWhile[Append[#, ev[Last[#]]] &, {gr},
  If[Rest[Last[#]] == Rest[gr], p2 - 1; p2 > 0 &], -1]]
  sadd[{d[x_, y_], b_}, {d[dx_, dy_], st_}] :=
  Module[{x1 = dx - x, y1 = dy - y, b2, x2, y2}, While[y1 > 0,
  {x1, y1} += If[Length[st] == 30, {8, -30}, {-2, -3}]];
  b2 = First[Cases[st, s[d[x3_, -y1], pl_, _., sb_] ->
  Join[bg[pl - x1 - x3, x1 + x3], x2 = x3 + Length[sb];
  y2 = -y1; sb]]]; {d[x2, y2], Join[b, b2]}]
```

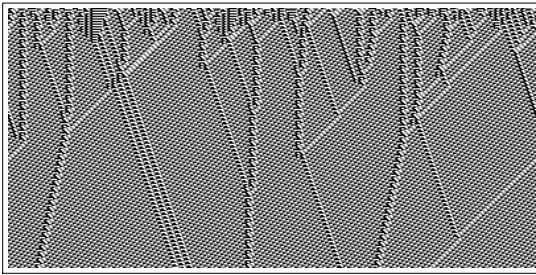
CTToR110[{}], {1}] yields blocks of lengths {7204, 1873, 7088}. But even CTToR110[{{0, 0, 0, 0, 0}, {}, {1, 1, 1, 1, 1, 1}, {}], {1}] already yields blocks of lengths {105736, 34717, 95404}. The picture below shows what happens if one chops these blocks into rows and arranges these in 2D arrays. In the first two blocks, much of what one sees is just padding to prevent clock pulses on the left from hitting data in the middle too early on any given step. The part of the middle block that actually encodes an initial condition grows like  $180 \text{Length}[init]$ . The core of the right-hand block grows approximately like  $500 (\text{Length}[\text{Flatten}[\text{rules}]] + \text{Length}[\text{rules}])$ , but to make a block that can just be repeated without shifts, between 1 and 30 repeats of this core can be needed.



■ **Page 689 · Tag systems.** The discussion in the main text and the construction above require a cyclic tag system with blocks that are a multiple of 6 long, and in which at least one block is added at some point in each complete cycle. By inserting  $k = 6 \text{Ceiling}[\text{Length}[\text{subs}]/6]$  in the definition of TS1ToCT from page 1113 one can construct a cyclic tag system of this kind to emulate any one-element-dependence tag system.

#### Class 4 Behavior and Universality

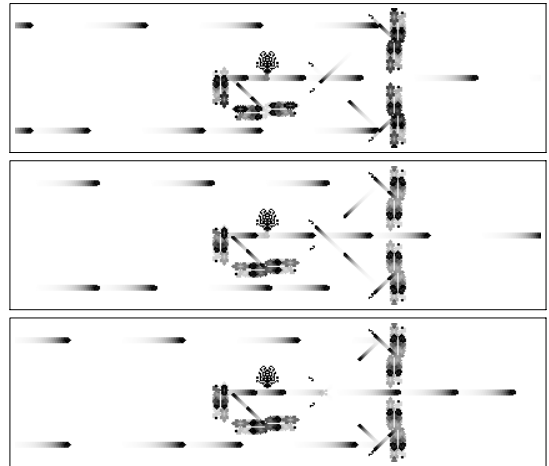
■ **2-neighbor rules.** Among 3-color 2-neighbor rules class 4 behavior seems to be comparatively rare; the picture at the top of the facing page shows an example with rule number 2144.



■ **Totalistic rules.** It is straightforward to show that totalistic cellular automata can be universal. Explicit simple candidates include  $k=2, r=2$  rules with codes 20 and 52, as well as the various  $k=3, r=1$  class 4 rules shown in Chapter 3.

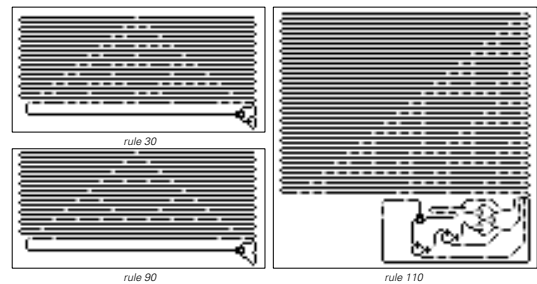
■ **Page 693 · 2D cellular automata.** Universality was essentially built in explicitly to the underlying rules for the 2D cellular automaton constructed by John von Neumann in 1952 as a model for self-reproduction. For among the 29 possible states allowed for each cell were ones set up to behave quite directly like components for practical electronic computers like the EDVAC—as well as to grow new memory areas and so on. In the mid-1960s Edgar Codd showed that a system similar to von Neumann’s could be constructed with only 8 possible states for each cell. Then in 1970 Roger Banks managed to show that the 2-state 5-neighbor symmetric 2D rule 4005091440 was able to reproduce all the same logical elements. (This system, like rule 110, requires an infinite repetitive background in order to support universality.) Following the invention of the Game of Life, considerable work was done in the early 1970s to identify structures that could be used to make the analog of logic circuits. John Conway worked on an explicit proof of universality based on emulating register machines, but this was apparently never completed. Yet by the 1980s it had come to be generally believed that the Game of Life had in fact been proved universal. No particularly rigorous treatments of the system were given, and the mere existence of configurations that can act for example like logic gates was often assumed immediately to imply universality. From the discoveries I have made, I have no doubt at all that the Game of Life is in the end universal, and indeed I believe that the kind of elaborate behavior needed to support various components is in fact good evidence for this. But the fact remains that a complete and rigorous proof of universality has apparently still never been given for the Game of Life. Particularly in recent years elaborate constructions have been made of for example Turing machines. But so far they have always had

only a fixed number of elements on their tape, which is not sufficient for universality. Extending constructions is often very tricky; much as in rule 110 it is easy for there to be subtle bugs associated with rare mismatches in the placement of structures and timing of interactions. The pictures below nevertheless show a rather simple implementation of a NAND gate in Life. The input comes from the left encoded as the presence or absence of spaceships 92 cells apart. The spaceships are converted to gliders. When only one glider is present, a new spaceship emerges on the right as the output. But when two gliders are present, their collision forms a wall, which prevents output of the spaceship.




If one considers rules with more than two colors, it becomes straightforward to emulate standard logic circuits. The pictures below show how 1D cellular automata can be implemented in the 4-color WireWorld cellular automaton of Brian Silverman from 1987, whose rules find the new value of a cell from its old value  $a$  and the number  $u$  of its 8 neighbors that are 1’s according to


$$a / \{0 \rightarrow 0, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow \text{If}[0 < u < 3, 1, 3]\}$$






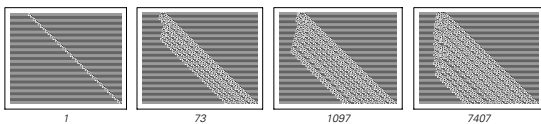
**The Threshold of Universality in Cellular Automata**

■ **Claims of non-universality.** Over the years, there have been a few erroneous claims of proofs that universality is impossible in particular kinds of simple cellular automata. The basic mistake is usually to make the implicit assumption that computation must be done in some rather specific way—that does not happen to be consistent with the way we have for example seen that it can be done in rule 110.

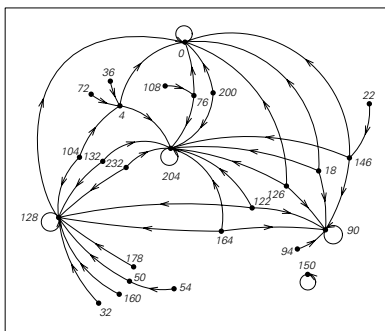
■ **Page 700 · Rule 73.**  on a white background yields a pattern that contains the last structure shown here.

■ **Page 700 · Rule 30.** For the first background shown, no initial region up to size 25 yields a truly localized structure, though for example  starts off growing quite slowly.

■ **Rule 41.** Various rules like rule 41 below can perhaps be viewed as having localized structures—though ones that apparently always travel in the same direction at the same speed. None of the first million initial conditions for rule 41 yield unbounded growth, though some can still generate fairly wide patterns, as in the pictures below. (The initial condition consisting of  repeated, followed by , followed by  repeated nevertheless yields a region that grows forever.)



■ **Page 702 · Rule emulations.** The network below shows which quiescent symmetric elementary rules can emulate which with blocks of length 8 or less. (Compare page 269.)



In all cases things are set up so that several steps in one rule emulate a single step in another. The examples shown in detail in the main text all have the feature that the block size  $b$  and number of steps  $t$  are matched, so that  $rt = b$  (where

the range  $r = 1$  for elementary rules). It is also possible to set up emulations where this equality does not hold—and indeed some of the cases listed in the main text and shown in the picture above are of this type. In those where  $rt < b$  there are more cells that are in principle determined by a given set of initial blocks—but the outermost of these cells are ignored when the outcome for a particular cell is deduced. In cases where  $rt > b$  there are more initial cells whose values are specified—but the outermost of these turn out to be irrelevant in determining the outcome for a particular cell. This lack of dependence makes it somewhat inevitable that the only rules that end up being emulated in this way are ones with very simple behavior.

In any 1D cellular automaton the color of a particular cell can always be determined from the colors  $t$  steps back of a block of  $2rt + 1$  cells (compare pages 605 and 960). But such a block corresponds in a sense to a horizontal slice through the cone of previous cell colors. And it turns out also to be possible to determine the color of a particular cell from slices at essentially any rational angle corresponding to a propagation speed less than  $r$ . So this means that one can consider encodings based on blocks that have a kind of staircase shape—as in the rule 45 example shown.

■ **Encodings.** Generalizing the setup in the main text one can say that a cellular automaton  $i$  can emulate  $j$  if there is some encoding function  $\phi$  that encodes the initial conditions  $a_j$  for  $j$  as initial conditions for  $i$ , and which has an inverse that decodes the outcome for  $i$  to give the outcome for  $j$ . With evolution functions  $f_i$  and  $f_j$  the requirement for the emulation to work is  $f_j[a_j] = \text{InverseFunction}[\phi][f_i[\phi[a_j]]]$

In the main text the encoding function is taken to have the form  $\text{Flatten}[a /. \text{rules}]$ —where  $\text{rules}$  are say  $\{1 \rightarrow \{1, 1\}, 0 \rightarrow \{0, 0\}\}$ —with the result that the decoding function for emulations that work is  $\text{Partition}[\bar{a}, b] /. \text{Map}[\text{Reverse}, \text{rules}]$ .

An immediate generalization is to allow  $\text{rules}$  to have a form like  $\{1 \rightarrow \{1, 1\}, 1 \rightarrow \{1, 0\}, 0 \rightarrow \{0, 0\}\}$  in which several blocks are in effect allowed to serve as possible encodings for a single cell value. Another generalization is to allow blocks at a variety of angles (see above). In most cases, however, introducing these kinds of slightly more complicated encodings does not fundamentally seem to expand the set of rules that a given rule can emulate. But often it does allow the emulations to work with smaller blocks. And so, for example, with the setup shown in the main text, rule 54 can emulate rule 0 only with blocks of length  $b = 6$ . But if either multiple blocks or  $\delta = 1$  are allowed,  $b$  can be reduced to 4, with  $\text{rules}$  being  $\{1 \rightarrow \{1, 1, 1, 1\}, 0 \rightarrow \{0, 0, 0, 0\}, 0 \rightarrow \{0, 1, 1, 1\}\}$  and  $\{0 \rightarrow \{0, 1, 0, 0\}, 1 \rightarrow \{0, 0, 1, 0\}\}$  in the two cases.

Various questions about encoding functions  $\phi$  have been studied over the past several decades in coding theory. The block-based encodings discussed so far here correspond to block codes. Convolutional codes (related to sequential cellular automata) are the other major class of codes studied in coding theory, but in their usual form these do not seem especially useful for our present purposes.

In the most general case the encoding function can involve an arbitrary terminating computation (see page 1126). But types of encoding functions that are at least somewhat powerful yet can realistically be sampled systematically may perhaps include those based on neighbor-dependent substitution systems, and on formal languages (finite automata and generalizations).

■ **Logic operations and universality.** Knowing that the circuits in practical computers use only a small set of basic logic operations—often just *Nand*—it is sometimes assumed that if a particular system could be shown to emulate logic operations like *Nand*, then this would immediately establish its universality. But at least on the face of it, this is not correct. For somehow there also has to be a way to store arbitrarily large amounts of data—and to apply suitable combinations of *Nand* operations to it. Yet while practical computers have elaborate circuits containing huge numbers of *Nand* operations, we now know that for example simple cellular automata that can be implemented with just a few *Nand* operations (see page 619) are enough. And from what I have discovered in this book, it may well be that in fact most systems capable of supporting even a single *Nand* operation will actually turn out to be universal. But the point is that in any particular case this will not normally be an easy matter to demonstrate. (Compare page 807.)

**Universality in Turing Machines and Other Systems**

■ **Page 706 · Minsky’s Turing machine.** The universal Turing machine shown was constructed by Marvin Minsky in 1962. If the rules for a one-element-dependence tag system are given in the form  $\{2, \{(0, 1), \{0, 1, 1\}\}$  (compare page 1114), the initial conditions for the Turing machine are

```
TagToMTM[{2, rule_}, init_] :=
  With[{b = FoldList[Plus, 1, Map[Length, rule] + 1]},
    Drop[Flatten[Reverse[Flatten[{1, Map[{Map[
      {1, 0, Table[0, {b[[# + 1]]}]} &, #, 1] &, rule], 1}]],
      0, 0, Map[{Table[2, {b[[# + 1]]}, 3] &, init}], -1]]
```

surrounded by 0’s, with the head on the leftmost 2, in state 1. An element -1 in the tag system corresponds to halting of the Turing machine. The different cases in the rules for the tag system are laid out on the left in the Turing machine. Each step of tag system evolution is implemented by having the

head of the Turing machine scan as far to the left as it needs to get to the case of the tag system rule that applies—then copy the appropriate elements to the end of the sequence on the right. Note that although the Turing machine can emulate any number of colors in the tag system, it can only emulate directly rules that delete exactly 2 elements at each step. But since we know that at least with sufficiently many colors such tag systems are universal, it follows that the Turing machine is also universal.

■ **History.** Alan Turing gave the first construction for a universal Turing machine in 1936. His construction was complicated and had several bugs. Claude Shannon showed in 1956 that 2 colors were sufficient so long as enough states were used. (See page 669; conversion of Minsky’s machine using this method yields a  $\{43, 2\}$  machine.) After Minsky’s 1962 result, comparatively little more was published about small universal Turing machines. In the 1980s and 1990s, however, Yuri Rogozhin found examples of universal Turing machines for which the number of states and number of colors were:  $\{24, 2\}$ ,  $\{10, 3\}$ ,  $\{7, 4\}$ ,  $\{5, 5\}$ ,  $\{4, 6\}$ ,  $\{3, 10\}$ , and  $\{2, 18\}$ . The smallest product of these numbers is 24 (compare note below), and the rule he gave in this case is:



Note that these results concern Turing machines which can halt (see page 1137); the Turing machines that I consider do not typically have this feature.

■ **Page 707 · Rule 110 Turing machines.** Given an initial condition for rule 110, the initial condition for the Turing machine shown here is obtained as *Prepend*[4list, 0] with 1’s on the left and 0’s on the right. The Turing machine

```
{1, 2} → {2, 2, -1}, {1, 1} → {1, 1, -1}, {1, 0} → {3, 1, 1},
{2, 2} → {4, 0, -1}, {2, 1} → {1, 2, -1}, {2, 0} → {2, 1, -1},
{3, 2} → {3, 2, 1}, {3, 1} → {3, 1, 1}, {3, 0} → {1, 0, -1},
{4, 2} → {2, 2, 1}, {4, 1} → {4, 1, 1}, {4, 0} → {2, 2, -1}}
```

with  $s = 4$  states and  $k = 3$  possible colors also emulates rule 110 when started from *Prepend*[list + 1, 1] surrounded by 0’s. The  $s = 3, k = 4$  Turing machine

```
{1, 0} → {1, 2, 1}, {1, 1} → {2, 3, 1},
{1, 2} → {1, 0, -1}, {1, 3} → {1, 1, -1}, {2, 0} → {1, 3, 1},
{2, 1} → {3, 3, 1}, {3, 0} → {1, 3, 1}, {3, 1} → {3, 2, 1}}
```

started from *Append*[list, 0] with 0’s on the left and 2’s on the right generates a shifted version of rule 110. Note that this Turing machine requires only 8 out of the 12 possible cases in its rules to be specified.

■ **Rule 60 Turing machines.** One can emulate rule 60 using the 8-case  $s = 3, k = 3$  Turing machine (with initial condition *Append*[list + 1, 1] surrounded by 0’s)

```
{1, 2} → {2, 2, 1}, {1, 1} → {1, 1, 1},
{1, 0} → {3, 1, -1}, {2, 2} → {2, 1, 1}, {2, 1} → {1, 2, 1},
{3, 2} → {3, 2, -1}, {3, 1} → {3, 1, -1}, {3, 0} → {1, 0, 1}}
```

or by using the 6-case  $s=2$ ,  $k=4$  Turing machine (with initial condition `Append[3 list, 0]` with 0's on the left and 1's on the right)

```
{1, 3} → {2, 2, 1}, {1, 2} → {1, 3, -1}, {1, 1} → {1, 0, -1},
{1, 0} → {1, 1, 1}, {2, 3} → {2, 1, 1}, {2, 0} → {1, 2, 1}}
```

This second Turing machine is directly analogous to the one for rule 110 on page 707. Random searches suggest that among  $s=3$ ,  $k=3$  Turing machines roughly one in 25 million reproduce rule 60 in the same way as the machines discussed here. (See also page 665.)

■ **Turing machine enumeration.** Of the 4096  $s=2$ ,  $k=2$  Turing machines (see page 888) 560 are distinct after taking account of obvious symmetries and equivalences. Ignoring machines which cannot escape from one of their possible states or which yield motion in only one direction or cells of only one color leaves a total of 237 cases. If one now ignores machines that do not allow the head to move more than one step in one of the two directions, that always yield the same color when moving in a particular direction, or that always leave the tape unchanged, one is finally left with just 25 distinct cases.

Of the 2,985,984  $s=3$ ,  $k=2$  machines, 125,294 survive after taking account of obvious symmetries and equivalences, while imposing analogs of the other conditions above yields in the end 16,400 distinct cases. For  $s=2$ ,  $k=3$  machines, the first two numbers are the same, but the final number of distinct cases is 48,505.

■ **States versus colors.** The total number of possible Turing machines depends on the product  $sk$ . The number of distinct machines that need to be considered increases as  $k$  increases for given  $sk$  (see note above).  $s=1$  or  $k=1$  always yield trivial behavior. The fraction of machines that show non-repetitive behavior seems to increase roughly like  $(s-1)(k-1)$  (see page 888). More complex behavior—and presumably also universality—seems however to occur slightly more often with larger  $k$  than with larger  $s$ .

■  **$s=2$ ,  $k=2$  Turing machines.** As illustrated on page 761, even extremely simple Turing machines can have behavior that depends in a somewhat complicated way on initial conditions. Thus, for example, with the rule

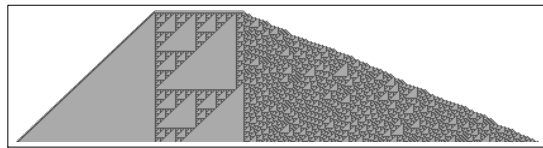
```
{1, 0} → {1, 1, -1}, {1, 1} → {2, 1, 1},
{2, 0} → {1, 0, -1}, {2, 1} → {1, 0, 1}}
```

the head moves to the right whenever the initial condition consists of odd-length blocks of 1's separated by single 0's; otherwise it stays in a fixed region.

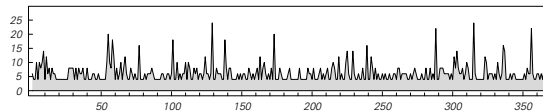
■ **Page 709 • Machine 596440.** For any list of initial colors *init*, it turns out that successive rows in the first *t* steps of the compressed evolution pattern turn out to be given by

```
NestList[Join[{0}, Mod[1 + Rest[FoldList[Plus, 0, #]], 2],
  {{0}, {1, 1, 0}}][Mod[Apply[Plus, #], 2] + 1]] &, init, t]
```

Inside the right-hand part of this pattern the cell values can then be obtained from an upside-down version of the rule 60 additive cellular automaton, and starting from a sequence of 1's the picture below shows that a typical rule 60 nested pattern can be produced, at least in a limited region.



The presence of glitches on the right-hand edge of the whole pattern means, however, that overall there is nothing as simple as nested behavior—making it conceivable that (possibly with analogies to tag systems) behavior complex enough to support universality can occur. The plot below shows the distances between successive outward glitches on the right-hand side; considerable complexity is evident.



■ **Page 710 •  $s=3$ ,  $k=2$  Turing machines.** Compare page 763 and particularly the discussion of machine 600720 on page 1145.

■ **Tag systems.** Marvin Minsky showed in 1961 that one-element-dependence tag systems (see page 670) can be universal. Hao Wang in 1963 constructed an example that deletes just 2 elements at each step and adds at most 3 elements—but has a large number of colors. I suspect that universal examples with blocks of the same size exist with just 3 colors.

■ **Encoding sequences by integers.** In many constructions it is useful to be able to encode a list of integers of any length by a single integer. (See e.g. page 1127.) One way to do this is by using the Gödel number `Product[Prime[i]^list[[i]], {i, Length[list]}]`. An alternative is to use the Chinese Remainder Theorem. Given  $p = \text{Array}[\text{Prime}, \text{Length}[\text{list}], \text{PrimePi}[\text{Max}[\text{list}] + 1]]$  or any list of integers that are all relatively prime and above  $\text{Max}[\text{list}]$  (the integers in *list* are assumed positive)

```
CRT[list_, p_] :=
  With[{m = Apply[Times, p]}, Mod[Apply[Plus,
    MapThread[#1 (m/#2)^EulerPhi[#2] &, {list, p}]], m]]
```

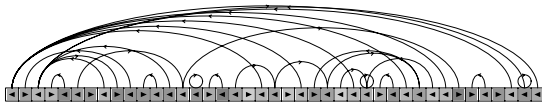
yields a number *x* such that `Mod[x, p] == list`. Based on this

```
LE[list_] := Module[{n = Length[list], i = Max[MapIndexed[
  #1 - #2 &, PrimePi[list]]] + 1}, CRT[PadRight[
  list, n + i], Join[Array[Prime[i + #] &, n], Array[Prime, i]]]]
```

will yield a number  $x$  that can be decoded into a list of length  $n$  using essentially the so-called Gödel  $\beta$  function

```
Mod[x, Prime[Rest[NestList[NestWhile[# + 1 &,
    # + 1, Mod[x, Prime[#]] == 0 &], 0, n]]]
```

■ **Register machines.** The results of page 100 suggest that with 2 registers and up to 8 instructions no universal register machines (URMs) exist. Using the method of page 672 one can construct a URM with 3 registers and 175 instructions (or 2 registers and 4694 instructions) that emulates the universal Turing machine on page 706. Using work by Ivan Korec from the 1980s and 1990s one can also construct URMs which directly emulate other register machines. An example with 8 registers and 41 instructions is:



or

```
{d[4, 40], i[5], d[3, 9], i[3], d[7, 4], d[5, 14], i[6],
d[3, 3], i[7], d[6, 2], i[6], d[5, 11], d[6, 3], d[4, 35],
d[6, 15], i[4], d[8, 16], d[5, 21], i[1], d[3, 1], d[5, 25],
i[2], d[3, 1], i[6], d[5, 32], d[1, 28], d[3, 1], d[4, 28],
i[4], d[6, 29], d[3, 1], d[5, 24], d[2, 28], d[3, 1],
i[8], i[6], d[5, 36], i[6], d[3, 3], d[6, 40], d[4, 3]}
```

Given any register machine, one first applies the function *RMToRM2* from page 1114, then takes the resulting program and initial condition and finds an initial condition for the URM using

```
R2ToURM[prog_, init_] := Join[init, With[
    {n = Length[prog]}, {1 + LE[Reverse[prog] /. {i[x_] -> x,
    d[x_, y_] -> 4 + 2n + x - 2y}], n + 1, 0, 0, 0}]]
```

For the first example on page 98 this gives  $\{0, 0, 211680, 3, 0, 0, 0, 0\}$ . The process of emulation is quite slow, with each emulated step in this example taking about 20 million URM steps.

■ **Recursive functions.** The general recursive functions from page 907 provided an early example of universality (see page 907). That such functions are universal can be demonstrated by showing for example that they can emulate any tag system. With the state of a 2-color tag system encoded as an integer according to *FromDigits[Reverse[list] + 1, 3]* the following takes the rule for any such tag system (in the first form from page 894) and yields a primitive recursive function that emulates a single step in its evolution:

```
TSToPR[{n_, rule_}] := Fold[Apply[c, Flatten[{#1, Array[p, #
2], c[r[z, c[r[p[1], s], c[r[z, p[2]], c[r[z, r[c[s, z], c[r[c[s,
c[s, z]], z], p[2]]]], p[2]]], p[1]]], p[#2]]]] &, c[c[r[p[1],
s], p[1], c[r[p[1], r[z, c[s, c[s, s]]]], c[c[r[z, c[r[p[1], s],
c[r[z, c[s, z]], c[r[p[1], r[z, c[r[p[1], s], c[r[z, p[2]], c[
r[z, r[c[s, z], c[r[c[s, c[s, z]], z], p[2]]]], p[2]]], p[1]]]],
p[2], p[3]]], p[1]]], p[1], p[1]], p[1]], p[2]]], p[n + 1],
```

```
MapIndexed[c[r[z, c[r[p[1], p[4]], p[2], p[3], p[4]]], c[r[z,
r[c[s, z], c[r[c[s, c[s, z]], z], p[2]]]], p[Length[#2] + 1]], #
1[[1]], #1[[2]]] &, Nest[Partition[#, 2] &, Table[Nest[c[s, #] &
z, FromDigits[Reverse[IntegerDigits[i, 2, n] /. rule] + 1, 3]],
{i, 0, 2^n - 1}], n - 1, {0, n - 1}], Range[n, 1, -1]]
```

(For tag system (a) from page 94 this yields a primitive recursive function of size 325.) The result of  $t$  steps of evolution is in general given in terms of this function  $f$  by *Nest[f, init, t]*, or equivalently  $r[p[1], f][t, init]$ . Any fixed number of steps of evolution can thus be emulated by applying a primitive recursive function. But if one wants to find out what happens after an arbitrarily large number of steps, one needs to use the  $\mu$  operator, yielding a general recursive function. (So for example  $\mu[r[p[1], f]][init]$  returns the smallest  $t$  for which the tag system reaches state  $\{\}$ —and never returns if the tag system does not halt.) Note that the same basic approach can be used to emulate Turing machines with recursive functions; the Turing machine configuration  $\{s, list, n\}$  can be encoded by an integer such as

```
2 ^ FromDigits[Reverse[Take[list, n - 1]]]
3 ^ FromDigits[Take[list, {n + 1, -1}]] 5 ^ list[[n]] 7 ^ s
```

■ **Lambda calculus.** Formulations of recursive function theory from the 1920s and before tended to be based on making explicit definitions like those in the note above. But in the so-called lambda calculus of Alonzo Church from around 1930 what were instead used were pure functions such as  $s = \text{Function}[x, x + 1]$  and  $\text{plus} = \text{Function}[\{x, y\}, \text{If}[x == 0, y, s[\text{plus}[x - 1, y]]]]$ —of just the kind now familiar from *Mathematica*. Note that the explicit names of (“bound”) variables in such pure functions are never significant—which is why in *Mathematica* one can for example use  $s = \# + 1 \&$ . (See page 907.)

The definitions in the note above involve both symbolic functions and literal integers. In the so-called pure lambda calculus integers are represented by symbolic expressions. The typical way this is done is to say that a function  $f_n$  corresponds to an integer  $n$  if  $f_n[a][b]$  yields *Nest[a, b, n]* (see note below).

■ **Page 711 · Combinators.** After it became widely known in the 1910s that *Nand* could be used to build up any expression in basic logic (see page 1173) Moses Schönfinkel introduced combinators in 1920 with the idea of providing an analogous way to build up functions—and to remove any mention of variables—particularly in predicate logic (see page 898). Given the combinator rules

```
crules = {s[x_][y_][z_] -> x[z][y[z]], k[x_][y_] -> x}
```

the setup was that any function  $f$  would be written as some combination of  $s$  and  $k$ —which Schönfinkel referred to respectively as “fusion” and “constancy”—and then the result of applying the function to an argument  $x$  would be

given by  $f[x]//crules$ . (Multiple arguments were handled for example as  $f[x][y][z]$  in what became known as “currying”.) A very simple example of a combinator is  $id = s[k][k]$ , which corresponds to the identity function, since  $id[x]//crules$  yields  $x$  for any  $x$ . (In general any combinator of the form  $s[k][_]$  will also work.) Another example of a combinator is  $b = s[k[s]][k]$ , for which  $b[x][y][z]//crules$  yields  $x[y[z]]$ .

With the development of lambda calculus in the early 1930s it became clear that given any expression  $expr$  such as  $x[y[x][z]]$  with a list of variables  $vars$  such as  $\{x, y, z\}$  one can always find a combinator equivalent to a lambda function such as  $Function[x, Function[y, Function[z, x[y[x][z]]]]]$ , and it turns out that this can be done simply using

```
ToC[expr_, vars_] := Fold[rm, expr, Reverse[vars]]
rm[v_, v_] = id
rm[f_[v_], v_] := FreeQ[f, v] = f
rm[h_, v_] := FreeQ[h, v] = k[h]
rm[f_[g_], v_] := s[rm[f, v]][rm[g, v]]
```

So this shows that any lambda function can in effect be written in terms of combinators, without anything analogous to variables ever explicitly having to be introduced. And based on the result that lambda functions can represent recursive functions, which can in turn represent Turing machines (see note above), it has been known since the mid-1930s that combinators are universal. The rule 110 combinator on page 713 provides however a much more direct proof of this.

The usual approach to working with combinators involves building up arithmetic constructs from them. This typically begins by using so-called Church numerals (based on work by Alonzo Church on lambda calculus), and defining a combinator  $e_n$  to correspond to an integer  $n$  if  $e_n[a][b]//crules$  yields  $Nest[a, b, n]$ . (The  $e$  on page 103 can thus be considered a Church numeral for 2 since  $e[a][b]$  is  $a[a[b]]$ .) This can be achieved by taking  $e_n$  to be  $Nest[inc, zero, n]$  where

```
zero = s[k]
inc = s[s[k[s]][k]]
```

With this setup one then finds

```
plus = s[k[s]][s[k[s[k[s]]]][s[k[k]]]]
times = s[k[s]][k]
power = s[k[s[s[k][k]]]][k]
```

(Note that  $power[x][y]//crules$  is  $y[x]$ , and that by analogy  $x[x[y]]$  corresponds to  $y^{x^2}$ ,  $x[y[x]]$  to  $x^{xy}$ ,  $x[y][x]$  to  $x^{y^x}$ , and so on.)

Another approach involves representing integers directly as combinator expressions. As an example, one can take  $n$  to be

represented just by  $Nest[s, k, n]$ . And one can then convert any Church numeral  $x$  to this representation by applying  $s[s[s[k][k]][k[s]][k[k]]]$ . To go the other way, one uses the result that for all Church numerals  $x$  and  $y$ ,  $Nest[s, k, n][x][y]$  is also a Church numeral—as can be seen recursively by noting its equality to  $Nest[s, k, n-1][y][x[y]]$ , where as above  $x[y]$  is  $power[y][x]$ . And from this it follows that  $Nest[s, k, n]$  can be converted to the Church numeral for  $n$  by applying

```
s[s[s[s[s[k][k]][k[s[s[k[s]][k]][s[k][k]]]]]]
k[s[s[k[s]][k]][s[s[k[s]][k]][s[k][k]]]]][s[s[k[s]][
s[s[k[s]][s[k[s[s[s[s[s[s[k][k]][k[s]]][k[k]]][k[s[s[
k[s]][k]][s[k][k]]]]][k[s[s[k[s]][k]][s[s[k[s]][k]][s[k][
k]]]]][k[s[s[s[s[k][k]][k[s[s[k[s]][s[k[s[s[k][k]]]][s[
k[k]][s[k[s[s[k[s]][k]]][s[s[k][k]][k[k]]]]]]][s[k[k]][s[
s[k][k]][k[k]]]]][k[s[s[s[k][k]][k[s[k]]]][k[s[k]]]]][
k[s[k]]]]]]][s[k[k]][s[s[s[k][k]][k[s[s[k[s]][k]][s[k][
k]]]]][k[s[s[k[s]][k]][s[s[k[s]][k]][s[k][k]]]]]]][
k[s[k][k]][s[s[k[s]][k]]]]][k[s[k][k]]][k[s[k]]]
```

Using this one can find from the corresponding results for Church numerals combinator expressions for  $plus$ ,  $times$  and  $power$ —with sizes 377, 378 and 382 respectively. It seems certain that vastly simpler combinator expressions will also work, but searches indicate that if  $inc$  has size less than 4,  $plus$  must have size at least 8. (Searches based on other representations for integers have also not yielded much. With  $n$  represented by  $Nest[k, s[k][k], n]$ , however,  $s[s[s[s]][k]][k]$  serves as a decrement function, and with  $n$  represented by  $Nest[s[s], s[k], n]$ ,  $s[s[s][k]][k[k[s[s]]]$  serves as a doubling function.

■ **Page 712 • Combinator properties.** The size of a combinator expression is conveniently measured by its *LeafCount*. If the evolution of a combinator expression reaches a fixed point, then the expression generated is always the same (Church-Rosser property). But the behavior in the course of the evolution can depend on how the combinator rules are applied; here  $expr//crules$  is used at each step, as in the symbolic systems of page 896. The total number of combinator expressions of successively greater sizes is  $\{2, 4, 16, 80, 448, 2688, 16896, 109824, \dots\}$  (or in general  $2^n \text{ Binomial}[2n-2, n-1]/n$ ; see page 897). Of these,  $\{2, 4, 12, 40, 144, 544, 2128, 8544, \dots\}$  are themselves fixed points. Of combinator expressions up to size 6 all evolve to fixed points, in at most  $\{1, 1, 2, 3, 4, 7\}$  steps respectively (compare case (a)); the largest fixed points have sizes  $\{1, 2, 3, 4, 6, 10\}$  (compare case (b)). At size 7, all but 2 of the 16,896 possible combinator expressions evolve to fixed points, in at most 12 steps (case (c)). The largest fixed point has size 41 (case (d)).  $s[s[s]][s][s][s]$  (case (e)) and  $s[s][s][s[s]][s][s]$  lead to expressions that grow like  $2^{t/2}$ . The maximum number of levels in these expressions (see

page 897) grows roughly linearly, although  $Depth[expr]$  reaches 14 after 26 and 25 steps, then stays there. At size 8, out of all 109,824 combinator expressions it appears that 49 show exponential growth, and many more show roughly linear growth.  $s[s][k][s[s[s]]][s]$  goes to a fixed point of size 80.  $s[s[s]][s][s][s][k]$  (case (i)) increases rapidly to size 7050 but then repeats with period 3.  $s[s[s[s]][s]][s][s][k]$  (case (j)) grows to a maximum size of 1263, but then after 98 steps evolves to a fixed point of size 17. For  $s[s][k][s[s[s][k]]][k]$  (case (k)) the size at step  $t-7$  is given by

$$h[1] = h[2] = h[3] = 12$$

$$h[t_] := If[Mod[t, 4] == 2, 2, 1] (h[Ceiling[t/2] - 1] + t) +$$

$$\{3, 5, -7, -1\} [Mod[t, 4] + 1]$$

Examples with similar behavior are  $s[s[s][k]][s][s[s][k]]$ ,  $s[s[s]][s][s[s][k]][k]$  and  $s[s[s][s]][s][s[s][k]]$ . Among those with roughly exponential growth but seemingly random fluctuations are  $s[s[s[s]][s][s][s][k]$ ,  $s[s[s]][s][s[s][s]][k]$  and  $s[s[s[s]][s][s][k]][s]$ .

■ **Single combinators.** As already noted by Moses Schönfinkel in 1920, it is possible to set up combinator systems with just a single combinator. In such cases, combinator expressions can be viewed as binary trees without labels, equivalent to balanced strings of parentheses (see page 989) or sequences of 0's and 1's. One example of a single combinator system can be found using  $\{s \rightarrow j[j], k \rightarrow j[j[j]]\}$ , and has combinator rules (whose order matters):

$$\{j[j][x\_][y\_][z\_]\rightarrow x[z][y[z]], j[j][j][x\_][y\_]\rightarrow x\}$$

The smallest initial conditions in this case that lead to unbounded growth are of size 14; two are versions of those for  $s, k$  combinators above, while the third is  $j[j][j[j]][j[j]][j[j]][j[j]][j[j]][j[j]][j[j]]$ .

The forms  $j[j]$  and  $j[j[j]]$  appear to be the simplest that can be used for  $s$  and  $k$ ;  $j$  and  $j[j]$ , for example, do not work.

■ **Page 714 • Cellular automaton combinators.** With  $k$  and  $s[k]$  representing respectively cell values 0 and 1, a combinator  $f$  for which  $f[a\_][a_0][a_1]$  gives the new value of a single cell in an elementary cellular automaton with rule number  $m$  can be constructed as

$$Apply[p[p[p[\#1][\#2]][p[\#3][\#4]]][p[p[\#5][\#6]][p[\#7][\#8]]] /. \{0 \rightarrow k, 1 \rightarrow s[k]\} \&, IntegerDigits[m, 2, 8]] // crules$$

where

$$p = ToC[z[y][x], \{x, y, z\}] // crules$$

The resulting combinator has size 61, but for specific rules somewhat smaller combinators can be found—an example for rule 90 is  $s[k[k]][s[s][k[s[s[k][k]][k[s[k]]][k[k]]]]$  with size 16.

To emulate cellular automaton evolution one starts by encoding a list of cell values by the single combinator

$$p[num[Length[list]]][$$

$$Fold[p[Nest[s, k, \#2]][\#1] \&, p[k][k], list]] // crules$$

where

$$num[n_] := Nest[inc, s[k], n]$$

$$inc = s[s[k][k]]$$

One can recover the original list by using

$$Extract[expr, Map[Reverse[IntegerDigits[\#, 2]] \&, 3 + 59/15 (16^Range[Depth[expr[s[k]][s][k]] // crules] - 1, 1, -1) - 1]]] /. \{k \rightarrow 0, s[k] \rightarrow 1\}$$

In terms of the combinator  $f$  a single complete step of cellular automaton evolution can be represented by

$$w = cr[p[inc[inc[x[s[k]]]]][$$

$$inc[x[s[k]]][cr[p[y[s[k]][s[k]][y[k]]],$$

$$\{y\}][p[x[s[k]][cr[p[p[f[y[k][k][k][s[k]]][$$

$$y[k][k][s[k]][y[k][s[k]]][y[s[k]][y[k][k]], \{y\}][$$

$$p[p[k][k]][p[k][x[k]]][s[k]][p[k][p[k][k]][k]], \{x\}]$$

$$cr[expr_, vars_] := ToC[expr // crules, vars]$$

where there is padding with 0 on either side. With this setup  $t$  steps of evolution are given simply by  $Nest[w, init, t]$ . With an initial condition of  $n$  cells, this then takes roughly  $(100 + 35n)t + 33t^2$  steps of combinator evolution.

■ **Testing universality.** One can tell that a symbolic system is universal if one can find expressions that act like the  $s$  and  $k$  combinators, so that, for example, for some expression  $e$ ,  $e[x][y][z]$  evolves to  $x[z][y[z]$ .

■ **Criteria for universality.** See page 1126.

■ **Classes of systems.** This chapter has shown that various individual systems with fixed rules exhibit universality when suitable initial conditions are chosen. One can also consider whole classes of systems in which rules as well as initial conditions can be chosen. And then one can say for example that as a class of systems cellular automata are universal, but neighbor-independent substitution systems are not.