EXCERPTED FROM

STEPHEN WOLFRAM

A NEW KIND OF SCIENCE

NOTES FOR CHAPTER 11:

The Notion of Computation

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.

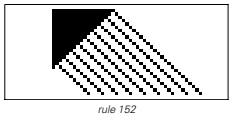
A crucial achievement in practical computing over the past several decades has been the creation of more and more sophisticated software. Often the programs that make up this software are several million instructions long. They usually contain many subprograms that perform parts of their task. Some programs 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

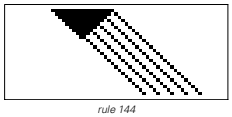
processor in a computer constitutes a low-level "machine" language. In practice, however, programs are rarely written at such a low level. More often, languages like C, FORTRAN, Java or *Mathematica* are used. In these languages, each construct represents what is often a large number of machine instructions. 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 *Ceiling[n/2]* and *Ceiling[n/4]*, respectively, are shown below with *n = 18* initial black cells.


rule 152


rule 144

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}→0, {_, 2, 3}→3, {1, 1, 3}→4, {_, 1, 4}→4, {1|2, 3,
    _}→5, {p:(0|1), 4, _}→7-p, {7, 2, 6}→3, {7, _, _}→7,
  {_, 7, p:(1|2)}→p, {_, p:(5|6), _}→7-p, {5|6, p:(1|2), _}→
    7-p, {5|6, 0, 0}→1, {_, p:(1|2), _}→p, {_, _, _}→0}
```

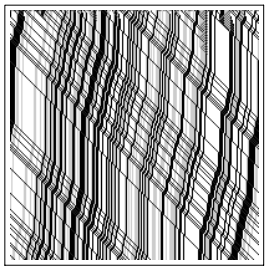
and the initial conditions consist of *Append[Table[1, {n}], 3]* surrounded by *0*'s. The rules can be implemented using *GeneralCARule* as given on page 867. (See also page 1186.)

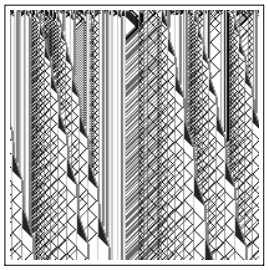
▪ **Page 640 · Primes cellular automaton.** The rules are

```
{{13, 3, 13}→12, {6, _, 4}→15, {10, _, 3|11}→15, {13, 7, _}→
    8, {13, 8, 7}→13, {15, 8, _}→1, {8, _, _}→7, {15, 1, _}→2,
  {_, 1, _}→1, {1, _, _}→8, {2|4|5, _, _}→13, {15, 2, _}→4,
  {_, 4, 8}→4, {_, 4, _}→5, {_, 5, _}→3, {15, 3, _}→12,
  {_, x:(2|3|8), _}→x, {_, x:(11|12), _}→x-1, {11, _, _}→13,
  {13, _, 1|2|3|5|6|10|11}→15, {13, 0, 8}→15,
  {14, _, 6|10}→15, {10, 0|9|13, 6|10}→15, {6, _, 6}→0,
  {_, _, 10}→9, {6|10, 15, 9}→14, {_, 6|10, 9|14|15}→10,
  {_, 6|10, _}→6, {6|10, 15, _}→13, {13|14, _, 9|15}→14,
  {13|14, _, _}→13, {_, _, 15}→15, {_, _, 9|14}→9, {_, _, _}→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,
    {2^(2 r+1) - 3}]], 10 - 3 rtab}], 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) ...}*). 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], 11, 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 $Log[2, k^{2r+1}]$ bits of information; the two-color rule that emulates it involves $Log[2, 2^{2^{2s+1}}]$ 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

```
TMToCA[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 → {0}, 0 → {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]}),
  {_r, s[v : (0 | 1)], _} → r[v], {_r, 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[rn_, x_], _} → x,
 {_, u, r[_, _]} → u, {_, x_, r[rn_, _]} → r[rn, x], {_, x_, _} → x}]
```

The initial condition is obtained by applying the rule *s[x_, y__] → {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, {_, _, p} → m + 3}
g[i[2], p_, m_] :=
  {{_, p, _} → p + 1, {p, 0, _} → m + 5, {p, _, _} → m + 6}
g[d[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}
g[d[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 *Join[Table[m + 2, {a}], {1}, 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 *Sqrt[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 *Length[list //. {0, x__} → {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, _, 10, _, _} → 1, {x : (3 | 7 | 6), _, _, _, _} → x,
  {_, _, 6, 4, 10} → 5, {_, _, 6, 8, 10} → 9, {_, 3, _, 10, _} → 4,
  {_, 7, _, 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 *Flatten[list /. {1 → {8, 1}, 0 → {4, 1}}]*. To access location *n* the left-hand initial conditions must contain *Flatten[{0, i, IntegerDigits[n, 2] /. {1 → {0, 11}, 0 → {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 *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 *Flatten[{p[0], Map[p, 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, 2 a, 2 b, 2 c} → {2 d, 1, 2 b, 2 c},
   {{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 (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[i_, 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, n k}]]
```

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]]}, ((3 2^c) + 2 c - 7) s]* states (always less than *6.03 k s*) and 2 colors:

```
TMToTM2[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, 2^(n-1)], n - 1,
         d}, Quotient[i, 2^(n-1)], 1}, {n, 2, b}, {i, 0, 2^n - 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 + 2^n c, n + 1, m}, c, -1},
      {n, 0, b - 1}, {i, 0, 2^n - 1}], With[{r = 2^b}, 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, _, _} → {0, 0}, {1, _, _} → {1, 1, 0, 1}}}* from page 895 as *{3, {0 → {0, 0}, 1 → {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:

```
TMToTS1[rules_] :=
 {2, Union[Flatten[rules /. ({i_, u_} → {j_, v_, r_}) :>
    {Map[#[i] → {#[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[{#1, 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 *TMToRM* resolves addresses and links the segments together.

```
TMToRM[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 *34 s* and *36 s*.

■ **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

```
RMToRM2[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 *{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}]}, {p g, 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 :→ (3 #/2 &), 1 :→ (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 + Table[Prime[i] ^ reg[[i]], {i, nr}] p - 1* where *p = Length[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.

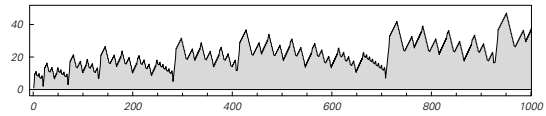
▪ **History.** The correspondence between arithmetic systems and register machines was established (using a slightly different approach) by Marvin Minsky in 1962. Additional work was done by John Conway, starting around 1971. Conway considered fraction systems based on rules of the form

```
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 *Log[2, list]* is as shown below, where *Rest[Log[2, Select[list, IntegerQ[Log[2, #]] &]]]* gives exactly the primes.



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

▪ **Multiway systems.** It is straightforward to emulate a *k*-color multiway 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.)

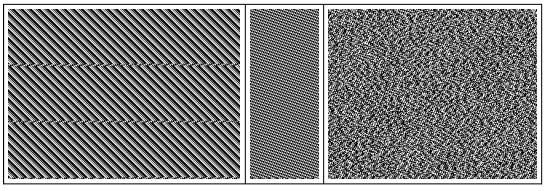
The Rule 110 Cellular Automaton

▪ **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]] - 11; 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, 11]], 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]]]], g3[[2]]}]

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, 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 10^49), 2] &,
{24600556015465847173551005175056992262806506766 1,
 10437461654894668528970898304417565508898347096 45}]

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--]; 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]}]
```

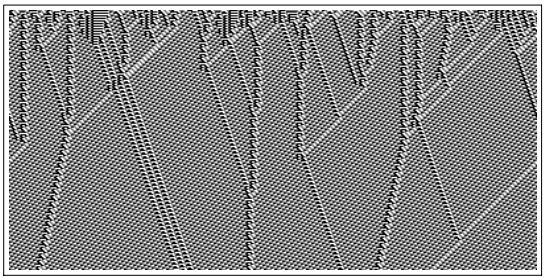
`CTToR110[{{}}, {1}]` yields blocks of lengths `{7204, 1873, 7088}`. But even `CTToR110[{{0, 0, 0, 0, 0, 0}, {}, {1, 1, 1, 1, 1, 1}, {}}, {1}]` already yields blocks of lengths `{105736, 34717, 95404}`. The picture below shows what happens if one chops these blocks into rows and arranges these in 2D 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 Length[init]`. The core of the right-hand block grows approximately like `500 (Length[Flatten[rules]] + Length[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 Ceiling[Length[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.

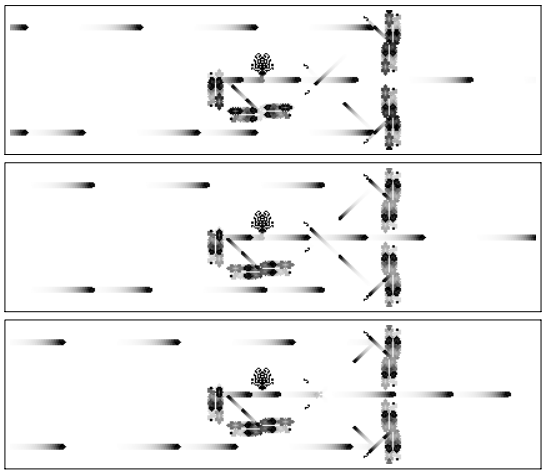
Class 4 Behavior and Universality

■ **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.



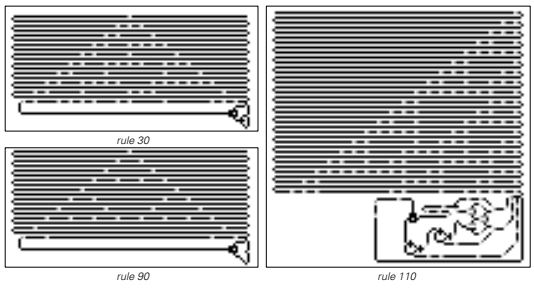
■ **Totalistic rules.** It is straightforward to show that totalistic cellular automata can be universal. Explicit simple candidates include $k = 2$, $r = 2$ rules with codes 20 and 52, as well as the various $k = 3$, $r = 1$ class 4 rules shown in Chapter 3.

■ **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

$a /. \{0 \to 0,\ 1 \to 2,\ 2 \to 3,\ 3 :\to If[0 < u < 3,\ 1,\ 3]\}$



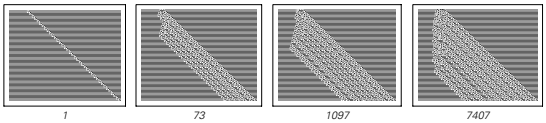
The Threshold of Universality in Cellular Automata

■ **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.

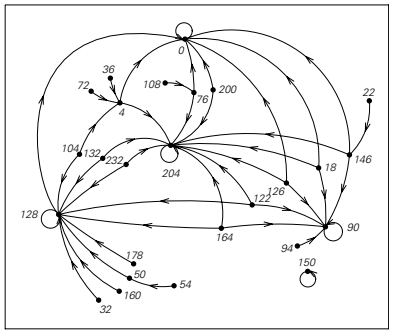
■ **Page 700 · Rule 73.** ■■■ on a white background yields a pattern that contains the last structure shown here.

■ **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 □□■■■□□■■■■ 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 $r\,t = b$ (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 $r\,t < b$ 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 $r\,t > b$ 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 $2\,r\,t + 1$ 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 ϕ 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 *Flatten[a /. rules]*—where *rules* are say *{1 → {1, 1}, 0 → {0, 0}}*—with the result that the decoding function for emulations that work is *Partition[ã, b] /. Map[Reverse, rules]*.

An immediate generalization is to allow *rules* to have a form like *{1 → {1, 1}, 1 → {1, 0}, 0 → {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 *rules* being *{1 → {1, 1, 1, 1}, 0 → {0, 0, 0, 0}, 0 → {0, 1, 1, 1}}* and *{0 → {0, 1, 0, 0}, 1 → {0, 0, 1, 0}}* in the two cases.

Various questions about encoding functions ϕ 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.)

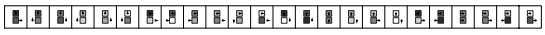
Universality in Turing Machines and Other Systems

▪ **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, Yurii 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[4 list, 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 $s\,k$. The number of distinct machines that need to be considered increases as k increases for given $s\,k$ (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

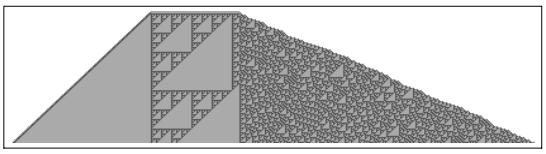
```
{{1, 0} → {1, 1, -1}, {1, 1} → {2, 1, 1},
 {2, 0} → {1, 0, -1}, {2, 1} → {1, 0, 1}}
```

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.

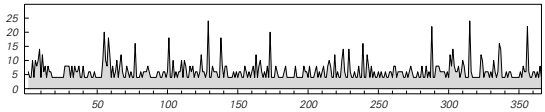
■ **Page 709 · Machine 596440.** For any list of initial colors *init*, it turns out that successive rows in the first t steps of the compressed evolution pattern turn out to be given by

```
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 = Array[Prime, Length[list], PrimePi[Max[list]] + 1]* or any list of integers that are all relatively prime and above *Max[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]]
```

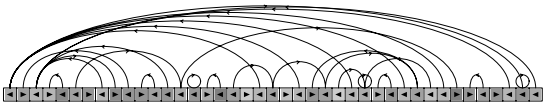
yields a number x such that *Mod[x, p] == list*. Based on this

```
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 β 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 + 2 n + x - 2 y}], n + 1, 0, 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 μ operator, yielding a general recursive function. (So for example *μ[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 a 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 = Function[x, x + 1]* and *plus = Function[{x, y}, If[x == 0, y, s[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]][k]][s[k][
 k]]]]]][k[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[k[s[k[s]][k]]][s[k][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, ...}` (or in general 2^n `Binomial[2 n - 2, n - 1]/n`; see page 897). Of these, `{2, 4, 12, 40, 144, 544, 2128, 8544, ...}` 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][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]][s]* goes to a fixed point of size 80. *s[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 → j[j], k → j[j[j]]}*, and has combinator rules (whose order matters):

```
{j[j][x_][y_][z_] → x[z][y[z]], j[j[j]][x_][y_] → 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]]*.

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 $f[a_{-1}][a_0][a_1]$ 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 → k, 1 → 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[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[s]][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 → 0, s[k] → 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]][k]][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 $(100 + 35\,n)\,t + 33\,t^2$ 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.