Universality of Wolfram's 2, 3 Turing Machine

Alex Smith

Submitted for the Wolfram 2, 3 Turing Machine Research Prize http://www.wolframprize.org

Table of Contents

Proof

Introduction

The main problem is to determine whether the following Turing machine is universal:

```
-0- -0- -1- -1- -2- -2-
 A   B   A   B   A   B
-1- -2- -2- -2- -1- -0-
  B A   A     B A     A
```

(This is known as 'system 0' in the proof below.) The proof I intend to give demonstrates that this Turing machine can emulate any two-colour cyclic tag system for an infinite number of steps; in order to do this, I first show that this Turing machine can emulate any two-colour cyclic tag system for an arbitrary finite number of steps (with the number of steps encoded in the initial condition), and then use this result as a basis for proving the more general result that this Turing machine can carry on such an emulation indefinitely

One main problem is to ensure that it is the system itself that's doing the calculation, and not the case that the universal part of the calculation is complete before the initial condition is even constructed. The construction I will give to show that system 0 can emulate any two-colour cyclic tag system for an arbitrary number of steps is quite complex, and it is not immediately obvious that it does not itself do the calculation rather than leaving system 0 to do the calculation. However, the proofs below leave many options open in the initial conditions (there is more than one initial condition that emulates a cyclic tag system), and after the rest of the proof I show that at least one of these initial conditions can be constructed by a process that is clearly not universal itself.

The main part of the proof proceeds by showing that the initial conjecture (conjecture 0, that system 0 can emulate any cyclic tag system for an arbitrary number of steps (and a few extra conditions)) is either equivalent to or implied by a sequence of other conjectures, each of which concerns a different system more complex than the conjecture preceding it, by showing that each system is either equivalent to the preceding system, or can be emulated by it at least for an arbitrary number of steps. Eventually, it is proved that system 5 can emulate any two-colour cyclic tag system for an arbitrary number of steps, thus implying that any of the preceding systems, and in particular system 0, can do the same. After this, it is shown how conjecture 0 implies that emulation for an infinite number of steps is possible.

Notation

Turing machines and Turing-machine-like systems will be written with 4 rows.

The top two rows are the state before each step, and the bottom two rows are the state after each step; the top row of each pair shows a portion of the tape (with a number representing a particular colour, or a dash representing “doesn't matter” in the top pair of rows or “no change to this tape element” in the bottom pair of rows), and the bottom row of each pair shows the position of the active element and which state the Turing machine is in (states are represented by letters).

All Turing machines can be represented in this format; it allows, however, for generalised systems in which more than one tape element is changed at a step, or elements other than the active element are taken into account at a step. (For instance, mobile automata can also be represented in this format using a single state.)

When showing the history of execution of such a system, more than two pairs of rows may be given; each pair shows the tape, state and active element after one more step of execution. (Dashes mean “doesn't matter” in the first pair of rows or “same as this element was in the last pair of rows” in such a case.)

Conjecture 0

The Turing machine

```
-0- -0- -1- -1- -2- -2-
 A   B   A   B   A   B
-1- -2- -2- -2- -1- -0-
  B A   A     B A     A
```

('system 0') can emulate any two-colour cyclic tag system for an arbitrary number of steps, using a finite-length initial condition in which the leftmost cell is a 0 and starts active in state A and in which the first cell to become active after the emulation has finished is the cell to the right of the initial condition, and if that cell is a 0 it becomes active in state A. (To be precise, 'finite-length initial condition', given in this and future conjectures, refers to an initial condition in which only finitely many of the cells are relevant, and no other cells are visited during the evolution of that initial condition.)

Conjecture 1

The system

```
-0- -0- -1- -1- -2- -20 -21 -22
 A   B   A   B   A   B   B   B
-1- -2- -2- -2- -1- -00 -12 -11
  B A   A     B A     A   B   B
```

('system 1') can emulate any two-colour cyclic tag system for an arbitrary number of steps, using a finite-length initial condition in which the leftmost cell is a 0 and starts active in state A and in which the first cell to become active after the emulation has finished is the cell to the right of the initial condition, and if that cell is a 0 it becomes active in state A.

Conjectures 0 and 1 are equivalent

To prove that conjectures 0 and 1 are equivalent, it only needs to be shown that the systems described in them are equivalent, as the conjectures are otherwise identical.

To see why the machines are identical, consider that the only state/active element combination in which they differ is when the active element is colour 2 and the system is in state B. There are three possibilities for the element to its right, and some of the evolution of each for the system in conjecture 0 is shown below.

```
-20 -21 -22
 B   B   B
-00 -01 -02
  A   A   A
    -02 -01
     A   A
    -12 -11
      B   B
```

The result is identical to that produced by system 1 in each case, so the systems must be equivalent.

Conjecture 2

The system

```
-0- -0- -0- -1- -1- -10 -11 -12 -2- -20 -21 -22 -2-
 A   B   C   A   B   C   C   C   A   B   B   B   C
-1- -2- -2- -2- -2- -0- -1- -1- -1- -0- -1- -1- -2-
  B A   A   A     B   A   C   C A     A   C   C   B
```

('system 2') can emulate any two-colour cyclic tag system for an arbitrary number of steps, using a finite-length initial condition in which the leftmost cell is a 0 and starts active in state A and in which the first cell to become active after the emulation has finished is the cell to the right of the initial condition, and that cell becomes active in state B.

Conjectures 1 and 2 are equivalent

To prove that conjectures 1 and 2 are equivalent, it only needs to be shown that the systems described in them are equivalent, as the conjectures are otherwise identical.

To see that the systems are identical, consider a system identical to system 2, except that if a state/tape combination would change into state C, it instead changes into state B with the new active element changed to 2 if it was 1 or 1 if it was 2 (with 0 left unchanged). This is identical to system 1. The behaviour of system 2 when the system is in state C is identical to the behaviour in state B except that the states for 1 in the active cell and 2 in the active cell have been switched; therefore, system 1 can be

emulated by system 2 by starting with the same tape and in the same state, and system 2 can be emulated by system 1 by starting with the same tape and in the same state if the system starts in state A or B, or with the active element subtracted from 3 (mod 3) and in state B if the system in conjecture 2 starts in state C.

Conjecture 3

The system

```
-0- -0- -0- -1- -1- -10 -11 -12 -2- -20 -21 -22 -2-
 A   B   C   A   B   C   C   C   A   B   B   B   C
-2- -2- -2- -1- -1- -0- -2- -2- -2- -0- -2- -2- -1-
  B A   A   A     B   A   C   C A     A   C   C   B
```

('system 3') can emulate any two-colour cyclic tag system for an arbitrary number of steps, using a finite-length initial condition in which the leftmost cell is a 0 and starts active in state A and in which the first cell to become active after the emulation has finished is the cell to the right of the initial condition, and if that cell is a 0 it becomes active in state A.

Conjectures 2 and 3 are equivalent

To prove that conjectures 2 and 3 are equivalent, it only needs to be shown that the systems described in them are equivalent, as the conjectures are otherwise identical.

To transform an initial condition from one system to the other, subtract all elements to the left of the active element from 3 (mod 3), and also the active element itself if the system starts in state A. Calculating every possible case shows the equivalence (elements that must be subtracted from 3 (mod 3) for the equivalence to hold are shown in **bold**):

```
System 2
-0- -0- -0- -1- -1- -10 -11 -12 -2- -20 -21 -22 -2-
 A   B   C   A   B   C   C   C   A   B   B   B   C
-1- -2- -2- -2- -2- -00 -1- -1- -1- -00 -1- -1- -2-
  B A   A   A     B   A   C   C A     A   C   C   B
System 3
-0- -0- -0- -2- -1- -10 -11 -12 -1- -20 -21 -22 -2-
 A   B   C   A   B   C   C   C   A   B   B   B   C
-2- -2- -2- -2- -1- -00 -2- -2- -1- -00 -2- -2- -1-
  B A   A   A     B   A   C   C A     A   C   C   B
```

Because these are identical (the states have been re-ordered so corresponding states align vertically) except that the bold elements are subtracted from 3 mod 3, the systems are equivalent.

Lemma 0

(From now on until further notice, 'the system' refers to the system in conjecture 3.)

For any set of adjacent elements all of which are 1s and 2s, in an initial condition in which the system starts in state A, and in which at least one element of the set is active in state B or C at some point during the execution of the system:

1. the leftmost element of the set will be the first one to be active in either state B or C;
2. once the leftmost element of the set is active in either state B or C, each element of the set will become active in either state B or C in order;
3. if an element of the set is active in state B, it will either stay the same or change from 2 to 0 at least until after the next time the system enters state A, and only the rightmost element can change to a 0, and only if the element to its right is a 0 (it will change to a 0 under these conditions);
4. if an element of the set is active in state C, it will change from a 2 to a 1 or a 1 to a 2 or 0 on the next step, and stay as that colour at least until after next time the system enters state A, and only the rightmost element can change to a 0, and only if the element to its right is a 0 (it will change to a 0 under these conditions);
5. if an element of the set is a 1 active in state B or C, the element to its right will either be active in the same state or possibly in state A if it started in state C on the next step, and the active state can

only be A if the element that ends up active is outside the set and it's a 0 at that time;

6. if an element of the set is a 2 active in state B or C, the element to its right will be active in the other state out of {B, C} (or possibly A if it started in state B) on the next step, and the active state can only be A if the element that ends up active is outside the set and it's a 0 at that time;
7. if the set contains an even number of 2s, the first time the element to the right of the set is active in state B or C, it will be active in the same state as the leftmost element of the set was active the first time it was active in state B or C (or possibly in state A if it started in state C and the element to the right of the set is a 0 by that time);
8. if the set contains an odd number of 2s, the first time the element to the right of the set is active in state B or C, it will be active in the other state out of {B,C} as the leftmost element of the set was active the first time it was active in state B or C (or possibly in state A if it started in state B and the element to the right of the set is a 0 by that time);
9. if the rightmost element of the set becomes a 0, it will become a 2 the next time it becomes active, which will be before the leftmost element of the set becomes active in state B or C, and it will stay as a 2 until after the leftmost element becomes active in state B or C. Likewise, if it becomes active but doesn't change to a 0, it will stay with that value until after the leftmost element becomes active in state B or C.

To prove sublemma 1, observe that if the active element in the initial condition is to the right of the set or within it, then the first time an element in the state is active, it will be in state A (either because it was in the initial condition, where state A is given, or because the active element moved to the left to enter the set, and if the active element moves to the left it must end up in state A). If any element in the set is active in state A, the active element will move left as long as it is within the set (because the set contains only 1s and 2s at that point). Therefore, the first time an element within the set becomes active in state B or C, it must be because the active element just moved to the right from outside the set, proving sublemma 1.

Sublemma 2 is proven by induction on the number of elements in the set (it's implied by sublemma 1 for a 1-element set, showing the base case, and by sublemmas 3 and 4 for the induction step, whose proofs don't depend on it).

Sublemmas 3, 4, 5, and 6 can all be proven simply by enumerating all possible cases for the element and the element to its right in each case. (“The next time the system enters state A” must happen before the next time active cell moves left, because the active cell can only move left in state A, i.e. before that particular cell becomes active again, and in this system only active cells can change.)

Sublemmas 7 and 8, when taken together, can be proven by induction, using sublemmas 5 and 6 for both the base case (a 1-element set) and the induction step (by considering the rightmost element in the set).

To prove sublemma 9, observe that the only way that the element can become a 0 causes the element to its right to become active on the next step, so that it must become active before the leftmost element becomes active in state B or C (if it *is* the leftmost element, the next time it becomes active will be in state A, and due to the rule for a 0 in state A it will become a 2 and the active element will move to the right, so it must become active at least a second time (actually at least a third time, but this isn't relevant to the proof) before it's active in state B or C). It must become a 2 the next time it becomes active, because the rule for the system always changes an active 0 to a 2; and it must stay that way (and stay that way even if it was a 1 or 2) until the leftmost element becomes active, because state A never changes a 1 or 2, and sublemma 1 prevents any element becoming active before the leftmost (sublemma 1 applies, because sublemmas 3 and 4 show that none of the other elements of the set can become a 0, and the system must reach state A at some point before it becomes active again, if it becomes active again, because only in state A can the active element move to the left.)

Lemma 1

For any set of 2^w adjacent elements, each of which is either a 1 or a 2, and an initial condition that starts in state A:

Define the nth *scan* of the set to be the nth time the leftmost element of the set is active in either state B or C, and the *parity* of the set to be the number of 0s and 2s in the set combined (mod 2); let p(n) denote the parity at the nth scan. *Even* parity is parity 0 and *odd* parity is parity 1. The *state parity* at the nth scan is 0

if the system was in state B at that scan, or 1 if the system was in state C at that scan; denote this by s(n). For all $n > 2^w$:

1. $p(n)=p(n-2^w)+s(n-2^w)$ (mod 2).
2. If at scan n p(n)+s(n)=0 (mod 2), then the next time the element to the right of the set becomes active it will be active in state B; otherwise, that element will be active in state A if it's a 0 or state C if it isn't.
3. At each scan, all elements of the set will be either a 1 or a 2.

Proof is by induction on w.

Base case

The base case is with w=0. Sublemma 1 in that case simply makes some statements about the set's only element which are all implied by lemma 0 (0.3 and 0.4 between them show how it changes according to the state the step after any scan, and 0.9 states that it stays with the same parity until the subsequent scan).

With w=0, sublemma 2 can simply be broken down into the twelve possible cases:

```
-10 -10 -11 -11 -12 -12 -20 -20 -21 -21 -22 -22
 B   C   B   C   B   C   B   C   B   C   B   C
-10 -00 -11 -21 -12 -22 -00 -10 -21 -11 -22 -12
  B   A   B   C   B   C   A   B   C   B   C   B
```

Sublemma 2 is here shown to hold in every possible case for the base case, so it must be true in the base case.

As for sublemma 3, it's a special case of lemma 0.9 in the base case.

Induction step

Suppose that Lemma 1 is true for some value of w. To show that it's true for w+1 as well, consider the two subsets of adjacent elements, one of which consists of the leftmost 2^w elements of the set, the other of which consists of the rightmost 2^w elements of the set.

First observe that a scan for the left subset will always happen exactly 2^w steps before a scan for the right subset (this is lemma 0.2), and that scans for the subsets must alternate, with a scan for the left subset happening first (by lemma 0.1 and the fact that lemma 0.2 indicates that it there must be more than 2^w steps between consecutive scans for the left subset); therefore the nth scan for the left subset always happens exactly 2^w steps before the nth scan for the right subset. Let sl(n) and pl(n) denote s(n) and p(n) for the left subset, and likewise sr(n) and pr(n) denote s(n) and p(n) for the right subset.

By applying sublemma 1 several times (sublemma 3 indicates that the conditions for sublemmas 1 and 2 are met at each scan for the left subset, and lemmas 0.3 and 0.4 indicate that if they're met at each scan for the left subset, they're met at each scan for the right subset too): (all equations mod 2)

$pl(n-2^w)=pl(n-2\_2^w)+sl(n-2\_2^w)$

$pl(n)=pl(n-2^w)+sl(n-2^w)=pl(n-2\_2^w)+sl(n-2\_2^w)+sl(n-2^w)$

$pr(n-2^w)=pr(n-2\_2^w)+sr(n-2\_2^w)$

$pr(n)=pr(n-2^w)+sr(n-2^w)=pr(n-2\_2^w)+sr(n-2\_2^w)+sr(n-2^w)$

Sublemma 2 states that sr(x)=pl(x)+sl(x) (mod 2), so: (all equations mod 2)

$pl(n)+pr(n) =pl(n-2\_2^w)+sl(n-2\_2^w)+sl(n-2^w)+pr(n-2\_2^w)+sr(n-2\_2^w)+sr(n-2^w)$

$=pl(n-2\_2^w)+sl(n-2\_2^w)+sl(n-2^w)+pr(n-2\_2^w)+pl(n-2\_2^w)+pl(n-2^w)+sl(n-2\_2^w)+sl(n-2^w)$

$=pr(n-2\_2^w)+pl(n-2^w)$

$=pr(n-2\_2^w)+pl(n-2\_2^w)+sl(n-2\_2^w)$

Observing that the parity for the right subset on its nth scan is the same as it was on the left subset's nth scan (because in all the steps in between an element of the left subset was active, and so the elements of the right subset were inactive and didn't change), that the parity for the entire set on its nth scan (i.e. the

left subset's nth scan) is the sum of the parities for the left and right subsets, and that s(x)=sl(x) for all x, this becomes

$p(n)=p(n-2^{w+1})+s(n-2^{w+1})$

which is sublemma 1, which is therefore true in the induction step.

To demonstrate that sublemma 2 is true in the induction step, observe that p(n)+s(n)=pl(n)+pr(n)+sl(n)=sr(n)+pr(n) (by sublemma 2), and by sublemma 2 for the right subset, the next time the element to the right of the set becomes active it will be active in state B if p(n)+s(n) (i.e. pr(n)+sr(n)) is 0, and otherwise that element will be active in state A if it's a 0 or state C if it isn't. This demonstrates sublemma 2 in the induction step.

Sublemma 3 is still a special case of lemma 0.9, and doesn't even need the induction assumption to prove.

As all the sublemmas are true in both the base case and induction step, lemma 1 must be true in general.

Corollary 0

Given any set of adjacent elements all of whose elements are either 1 or 2, with 2^w elements for some integer w, the parity of that set on each of its first 2^w scans is independent of the state parity on those scans. (Therefore, if the system is only allowed to run for less than 2^w scans, the parity on every scan is independent of the state parity on any scan.)

Corollary 1

Given a wanted value for the parity of a set of adjacent elements on each of its first n scans, there is a set of adjacent elements, all of whose elements are either 1 or 2, which has the wanted parity on each of those scans, regardless of whether the system was in state B or state C in each of those scans. I give two methods to construct such a set; both will be referenced later.

Method 1

Such a set can be constructed by starting with any set with $2^w>n$ elements, placing it in a tape with a 1 or 2 after it and nothing else but 0s, and changing the state at each scan to either state B or state C according to whether or not respectively the set's parity on that scan matches the wanted parity on that scan; the resulting set has the correct value on each of its *next* 2^w scans, so taking what the set has become after 2^w scans will produce a set that has the wanted value on each of its *first* 2^w scans. Therefore, if the system is only allowed to run for less than 2^w scans, it's possible to construct a set of adjacent elements whose parity has a wanted value on every scan.

Method 2

The tricky bit about using the construction given in method 1 is that it, while valid, is somewhat cumbersome, and more seriously it might be considered to be too complicated for the proof to demonstrate universality. (This is because it uses system 3 itself to do the calculations.) A simpler construction, however, is possible (but although the construction is simpler the proof that it is correct is considerably more complicated). Throughout the below demonstrations and proof, let the *parity set* of a string of 1s and 2s of length 2^w be the set of scans (out of the first w scans) on which it has odd parity. As a demonstration, look at the following strings:

```
2111111111111111
2211111111111111
2121111111111111
2222111111111111
2111211111111111
2211221111111111
2121212111111111
2222222211111111
2111111121111111
2211111122111111
2121111121211111
2222111122221111
2111211121112111
```

```
2211221122112211
2121212121212121
2222222222222222
```

These are the strings with w=4 corresponding to the parity sets {0}, {1}, {2}, and so on through to {15}. Due to the additive nature of such sets and/or strings, it's actually possible to generate a more complicated set by XORing together each of the strings corresponding to the elements of the set (treating 2 as the true logic value, and 1 as false); for instance, {6,8} would be 1121212121111111. It's also simple to generate such a table of strings in a mechanical manner; comparing the table above to some basic cellular automata would suggest that the following cellular automaton generates it (the | refers to the left-hand edge of the set):

```
|1 |2 11 12 21 22
 1  2  1  2  2  1
```

(The first of the cases given cannot be induced from the table, but is what is required for the proof below to work.) Of course, none of this has been proved yet; I will give the proof now. Proof is by induction on the row in the table. (The proof below assumes that w>=1; w will of course always be higher, usually substantially higher, than 1 in order to be able to simulate the system to the required number of steps.)

Base case

A string consisting of a 2 followed by 2^w-1 1s corresponds to the set {0}. To show this, consider the string consisting of 2^w 1s; if a scan of that string starts in state B, the string remains unchanged (due to lemma 0), so it must correspond to the empty set. A scan of that set starting in state C changes that string to the string consisting of 2^w 2s; likewise, if the string consisting of a 2 followed by 2^w-1 1s is encountered in state B, it changes to the string consisting of 2^w 2s. Therefore, the next 2^w-1 scans of each string must have an even parity of the string each time (they do in one case, so they must in the other), and the only element of the corresponding sets in which they can differ is the first scan, or the 0 element. Because the parity of an even-length string containing exactly one 2 must be odd, the element 0 must be in the set, so {0} is the only set it can correspond to.

Induction step

Lemma 0 implies that if a string of 1s and 2s has a scan in state B, the resulting string at the next scan will contain 1s up to but not including where the first was originally 2, 2s up to but not including where the next 2 was originally, 1s up to but not including where the next 2 was originally, and so on. This means that in order to produce a given string on scan x+1 when an unknown string has a scan in state B on scan x, the unknown string must contain a 2 at every position which is to the right of a boundary between consecutive 1s in the given string, a 1 at all other locations except possibly the first element, and start with the same element as the given string. This is exactly what the cellular automaton does, so if any row of the table has a scan, then on the next scan it will look like the row above. A string having a scan in system 3 is equivalent to decrementing every element in a parity set and discarding those scans that fall below 1 (because a scan has happened, the number of scans between now and each future scan on which it has odd parity reduces by 1, effectively decrementing every element in the set); therefore, each row, when decremented, must produce the row above. Also, each row other than the first must have even parity; this is because each element is set to 2 if either the element above it or the element above it and to its right is a 2, but not both, and induction on the number of 2s in the previous row (base case: no 2s produces a row of 1s which has even parity, induction step: changing a 1 to a 2 changes the parity of two, i.e. an even number of elements in the row below) shows that any row with a previous row (i.e. any but the first) must have even parity. As it has already been determined that the first row of the table corresponds to {0}, this means that the second row must correspond to {1}, the third row to {2}, and so on, proving that the cellular automaton does indeed work.

Making the method still simpler

Running a cellular automaton, whilst simpler than running system 3, may still appear to be an excessively complicated method of obtaining the system 3 string with a given parity set. Luckily, the cellular automaton in question (two-colour nearest-neighbour rule 60 with different names for the 'colours' of its cells) is one that has already been analysed and has simple nested behaviour, and in fact its behaviour can be predicted in advance without actually running the automaton. (The rule for the automaton is identical

for that for Pascal's Triangle modulo 2; it is a well established result that any given cell in Pascal's Triangle can be calculated using the formula for binomial coefficients, and that formula can therefore be used to predict the behaviour of rule 60 and provide an even simpler way to create the relevant string.)

Conjecture 4

The following system ('the system' in this section, and 'system 4' elsewhere) can emulate any two-colour cyclic tag system for an arbitrary number of steps (it's sort of an infinite-colour Turing machine), using a finite-length initial condition in which the first element of the tape starts active in state A, and in which the first time the active element leaves the defined portion of the initial condition, the system is in state C:

There is a right-infinite tape, each of whose elements is either a finite (and possibly empty) set of non-negative integers or a star. There cannot be two consecutive stars on the tape, and the tape cannot start with a star, but any number of consecutive sets are allowed. There is one active element at a time, and the system can be in one of three states, A, B, or C. (For example (and as an example of the notation I use):

```
{0,6,8}{}*{3,4}*{8}{11,20}...
          B
```

Each step of the system obeys the following rules:

1. If a set is active and the system is in state A, there is no change to the set and the element to its left becomes active (the system remains in state A). If there is no element to the left (i.e. the set is the first element), instead the same set remains active and the system changes to state B.
2. If a star is active and the system is in state A, the star is deleted from the tape (causing the elements to each side of it to become adjacent), the element that was to its right becomes active, and the system changes to state B.
3. If a set is active and the system is in state B or C, every element in the set is decremented. If this would decrement a 0, the 0 is removed from the set and the system changes to the other state out of {B,C}; whether a 0 is decremented or not, the element to the right of the active element becomes active.
4. If a star is active and the system is in state B, the star is deleted from the tape (causing the elements to each side of it to become adjacent), the element that was to its left becomes active, and the system changes to state A.
5. If a star is active and the system is in state C, the element to its right becomes active and the system remains in state C. A 1 is added to that set, or if there is a 1 in the set already, instead a 1 is removed from that set.

Conjecture 4 implies conjecture 3 (and therefore conjecture 0)

To prove that conjecture 4 implies conjecture 3, it is enough to show that with the right initial condition system 3 can emulate a finite system 4 initial condition obeying the constraints set out there for an arbitrary number of steps, with the system 3 initial condition and 'final' condition obeying the constraints set out in conjecture 3. This is done by stating what the initial condition is, and then proving that it emulates system 4. (The description below also explains what the condition of the system can be like during execution, because the proof works by showing that the system 3 program changes from one condition to another the same way the system 4 program does.)

Initial condition / Condition during execution

- The start of system 4's tape can translate to the string “0222...222” repeated 2^{w+1}-4 times (where there are 2^{w+1}-1 2s in the string, and the ... is made up entirely of 2s), followed by a 0, 2^{w+1}-2 2s, and a 1 (but only if the leftmost system 4 element is an active set, and then only if it's active in state A; this form is provided solely to meet the condition that the initial condition that 'fulfils' conjecture 3 starts with a 0 active in state A). Another possible translation is the string “221212...21210” (where the ... is made up of 1s and 2s alternating, and the string has a length of 2^{w+1}) repeated a sufficient number of times (at least 2^{w+1}-4 times minus (at later times in the evolution of the system) one repetition for each time that region of the tape has been encountered

during the evolution of the system 3 system so far), except without the 0 at the end of the last repetition, and followed by any block of 1s and 2s that has even parity and would have even parity on each of its next x scans if it such a scan happened in state B each time, where x is the number of 0s in this region of the tape. (A zero-length sequence of 1s and 2s is trivially sufficient to fulfil the 'even parity on its next x scans' condition; so is any string consisting only of 1s, because a scan starting in state B on such a string leaves it unchanged, and a string consisting only of 1s always has even parity, and so are any number of strings with that property concatenated with each other, because a scan on a string with even parity starting in state B leads to the element to the right of that string becoming active in state B.)

- A set of numbers translates to a set of consecutive 1s and 2s that has odd parity on the (n+1)th scan if and only if n is in the set (for instance, it has odd parity on the first scan if it contains a 0, and even parity otherwise) during all the time that the emulation is run (i.e. for all the arbitrary number of steps of system 5 required), and that starts with a 2 and ends with a 2. (At this point in the proof, I simply assume there is a simple procedure for finding such a set; an explicit construction will be given later, completing this part of the proof.)
- A star to the left of the active element, or at the active element if system 4 is in state A or C, translates to a 0 that replaces the rightmost 2 of the set to its left.
- A star to the right of the active element, or at the active element if system 4 is in state B or C, translates to a 0 that replaces the leftmost 2 of the set to its right. (At the active element in state C, the star translates to two 0s, between them replacing the right end of the set to their left and the left end of the set to their right.)
- After the end of the system 3 initial condition corresponding to the system 4 initial condition given is a single 1.
- The states in the two systems correspond to each other; the left-hand end of the string of 1s and 2s that represents a set is active when the set is active, or the 0 that represents a star is active when the star is active. In the case that the 0 that represents a star is active in system 4's state C, the rightmost of the two 0s representing the star in system 3 is active, but in state A (the only time in constructing the initial conditions that the systems are in different states). Note that the left-hand-end of a set can never be a 0 while the set is active, because any star to the set's left would be replacing a 0 in the set to its left. In the special case when the start of the tape is made out of repetitions of “0222...222” rather than the more complicated alternative representation of the start of the tape, the start of the tape being active in state A is instead represented by the initial 0 being active in state A. Note also that this means that the condition on the system 4 initial condition leaving the right-hand end of the program means that once this happens, the lone 1 will become active in state C; if there is a 0 to its right, that 0 will then become active in state A (due to the rules of system 3), which is just what conjecture 3 requires.

Corollary 1 proves that it's possible to find a set of consecutive 1s and 2s in system 3 with any desired parity at any finite number of scans, and gives a construction, but it needs to be proved that it's possible to construct such a set that starts and ends with a 2. To show this, two more lemmas are needed: (the rest of this section refers to system 3)

- In system 3, changing every element of a set of adjacent 1s and 2s with 2^w elements from a 1 to a 2 or vice versa doesn't affect its parity on any of the first 2^w-1 scans.
- In system 3, changing the first, third, fifth, and in general (2n+1)th elements of a set of adjacent 1s and 2s with 2^w elements from a 1 to a 2 or vice versa doesn't affect its parity on any of the first 2^w-2 scans or on the 2^wth scan.

The truth of these lemmas is sufficient to create a set of adjacent 1s and 2s of width 2^w which has a desired parity at each of the first 2^w-2 scans (and it's always possible to pick w high enough that it's correct on any given finite number of scans) and starts and ends with a 2; first create a set of adjacent 1s and 2s, not necessarily starting and ending with a 2 which has the desired parity on the first 2^w-2 scans (using the second method given in the proof of Corollary 1). Then, if it starts and ends with a 2, it's a correct set; if it starts and ends with a 2, change every element from a 1 to a 2 or vice versa; if it starts but doesn't end with a 2, change every odd-indexed element (with the leftmost element being indexed '1') from a 1 to a 2 or vice versa; and if it ends but doesn't start with a 2, change every even-indexed element

from a 1 to a 2 or vice versa.

To prove these lemmas, consider the first method given in Corollary 1, and imagine changing what happens in the last 2 scans before scan 2^w+1 (i.e. the first scan for which the desired parity is fixed) would be reached:

- A change in the state parity at the last scan will cause each element to change from a 1 to a 2 or vice versa at scan 2^w+1 (proof is by induction on the number of elements (not the $\log_2$ of the number of elements!); given lemma 1.3, the base case is lemmas 0.3 and 0.4, and the induction step is given by lemmas 0.5 and 0.6 to determine what state the rightmost element will be in and lemmas 0.3 and 0.4 to determine the effect it has on that element).
- A change in the state parity at the penultimate scan will cause each element to change from a 1 to a 2 or vice versa at scan 2^w, for the same reasons as in the previous bullet point. After the next scan, the first element will be changed from a 1 to a 2 or vice versa from what it would have been if the state parity hadn't been changed at the penultimate scan (changing the parity of elements is *additive*, i.e. changing the parity of an element causes the state the element to its right is in when it becomes active compared to what it would have been earlier to change regardless of which state the system was in when it was active a step earlier), which will cause the element to its right to be active in the opposite state to what it would have been active in. That element has however already had its parity changed, and the two changes cancel out (a consequence of additivity) and leave it with the value it would have had anyway. The third element will be active in the state it would have been active in anyway, and so end up with the opposite parity to if the state parity hadn't changed last step (because its own parity ended up opposite to what it would have been anyway last step), and so on (that this pattern continues can be proven using the same argument so far and inducting on half the number of elements in the set).

Now because there are 2^w possible choices for desired parity when using the Corollary 1 method, there are 2^w different sets that might be the outcome of the method, and each choice for desired parity must produce a different set as its outcome (because if two sets were the same, their pattern of parities would be the same), so the mapping from parity at each of the first 2^n steps to each of the possible sets is a bijection. Therefore, because changing the state parity at the last scan always subtracts all the set's elements from 3 (mod 3), subtracting all the set's elements from 3 (mod 3) must change the state parity at the last scan, and nothing else; likewise for changing the parity at every second element. So the lemmas in this section are true, and the initial condition always exists.

Why the initial condition works

To show that the initial condition works, it's enough to show that each of the possible steps for system 4 lead to the same result (taking the correspondence between the systems demonstrated in the initial condition section, except that ends of a set not currently replaced by a star need not be 2s) as running section 3 does. Taking each of the 5 rules for system 4 in turn:

1. When system 3 is in state A and a 1 or 2 is active (corresponding to state A and a set active in system 4), the active element keeps moving left until a 0 is active. So either there's another set to the left of the active set (in which case it will become active and the system will remain in state A), or there's a star to the left of the active set (in which case it will become active and the system will remain in state A), or there's the left end of system 4's tape to the left of the active set. There are two cases here:
    1. One case is that the string representing the left-hand end of the system is made out of repetitions of “0222...222”; in this case, the initial 0 changes to a 2, then alternate 2s in each repetition change to 2 and 1 (starting with a 2) as the system changes backwards and forwards between states B and C. Because the number of 2s in each batch is 2^w-1 and therefore odd (because w must be at least 1), the last 2 in each repetition is active in state B, and so changes to a 0 (by the rules of system 3, as it is followed by a 0 itself), making the 0 at the start of the next repetition active in state A. This process repeats through all the repetitions, changing them all to the “221212...21210” form. On the last repetition, the final element is a 1 not a 2, and it isn't followed by a 0; so when it becomes active in state B it stays as a 1 and the next element becomes active in state B. This therefore changes the initial sequence of the string to the other form described in the section on the initial condition above (the repetitions of the

"0222...222" form have all changed to the "221212...21210" form, and a single 1 meets the condition about having even parity for the next x steps if encountered in state B each time). The only difference it makes in terms of the system 4 equivalence of the condition, in fact, is that the system's state has changed from state A to state B, which is exactly what the rules of system 4 require, even though a lot has changed in terms of internal representation.

2. The other case is that the string representing the left-hand end of the system is made out of repetitions of "221212...21210"; in this case, the system 3 active element will move left until a 0 is active without any change to the tape (because that's how system 3 behaves in state A). Once the 0 is active, it will change into a 2, and the element to its right (a 2) will become active in state B; that 2 will remain a 2, and the element to its right (the second 2 of the repetition that's missing its final 0) will become active in state C; the element to its right (the first 1) will become active in state B, and then the system will follow a cycle where 1212 (with the first 1 active in state B) will change to 1221 (with the element to its right active in state B) as long as the alternating 1s and 2s continue. (This assumes that w is at least 2, so that the string missing its final 0 is made of 22, followed by a whole number of repetitions of 1212, followed by a 1). When the final 1 is active, this will be in state B, so it will remain a 1 and whatever was to its right will become active in state B. The number of 0s in the area to the left has reduced by 1, so the number of steps that the remainder of the system 3 string representing the left-hand end of the system 4 tape has to remain with an even parity for reduces by 1, so that portion of the string will retain the property that was required in the initial condition, and because it's been defined to have even parity, the element to *its* right (which corresponds to the start of the first element of the system 4 tape) will become active in state B. This has achieved the desired result, of changing from state A to state B whilst leaving the active element in the same place, so all that has to be shown is that the cells that became active during this have the required property of having an even parity for the next x scans if encountered in state B each time. It's been established above that the "022121.....21" which is the final 0 of the repetition beforehand, followed by the repetition missing its final 0, changes into "2211" repeated 2^{w-1} times; because everything to the right of this region has the required property, and the concatenation of two regions has the required property, all that it's left to show is that that region has the required property. To see this, observe that the parity set corresponding to this region is $\{2^{w+1}-3\}$ (because two steps of recoloured rule 60 evolution from this pattern lead to a pattern consisting of 2^{w+1} 2s, which has the parity set $\{2^{w+1}-1\}$), and therefore it must have an even parity for the next x zeros as x cannot be more than $2^{w+1}-4$.

   Showing this graphically may be easier to visualise than a text description (the bold shows where the repetition missing its final 0 was before this happened; this example uses a single 1 as the string to the right of the repetition missing its final 0):

```
022121211
 A
022121211
A
222121211
 B
222121211
  C
221121211
   B
221121211
    B
221121211
     C
221122211
      C
221122111
       B
221122111
        B
```

   The argument above requires that there is always at least 1 0 in the system 3 region representing the left-hand end of the system 4 tape; this, however, is guaranteed, because at most one zero net is removed from that region whenever the leftmost element of the system 4 tape becomes active in state A, and immediately afterwards the string representing the leftmost element of the system 4 tape becomes active in state B, and therefore has a scan, and

w has already been assumed to be sufficiently high that no string has more than 2^w scans (and 2^{w+1}-4 is more than 2^w for w at least 2). This implies that the active element will never go off the left-hand end of the initial condition, and therefore when the active element does leave the initial condition it must be to go off the right-hand end, fulfilling that condition in conjecture 3.

2. A star active in state A must be a 0 replacing the 2 at the rightmost end of a set. So this happens:

```
{ set  *{ set  }
-------0--------
       A
-------2--------
        B
```

So as required, the star has been deleted, the set previously to its right has become active, and the 2 that was replaced in the element to the left is back as a 2 again. (In this case, one step in system 4 corresponds to one step in system 3.)

3. Although this looks complicated, this is actually just a description of what happens after a scan. The condition described in the system 4 rules corresponds to a scan in system 3; a 0 in system 4's set corresponds to odd parity in system 3 (which by lemmas 0.7 and 0.8 cause the system to change to the other state out of B and C, as desired, but the "0 to the right" possibility could apply here and will be discussed later), and the fact that the scan has happened mean that the length of time until the next, next-but-one, next-but-two, and in fact all remaining instances of odd parity at a scan has been reduced by one scan (which emulates the decrementing of the set). So if there isn't a 0 to the right of the set (i.e. there is another set to the right, not a star), that set will correctly become active, and the emulation works correctly. If there *is* a 0 to the right of the set, then there are two possibilities. One possibility is that all the 1s and 2s are replaced by other 1s and 2s, and the system ends up with the 0 to the right of the set active, and in state B; this is consistent with the definition of an active star in state B, meaning that the element to the right has been activated as desired. The other possibility is that the scan would have ended up in state C, were it not for the 0 to the right, but instead ended up in state A, with the rightmost element of the set replaced with a 0 and the rightmost of the two adjacent 0s thus produced being active; this is the definition of an active star in system 4's state C. The remaining thing to check is that it must have been a 2 that was replaced by a 0, but lemmas 0.3 and 0.4 show that only an element that would otherwise have changed to 2 can change to 0 this way, so the set that was previously active has had its equivalent altered appropriately. That covers all the cases, so this rule in system 4 is accurately simulated in system 3.

4. A star active in state B must be a 0 replacing the 2 at the leftmost end of a set. So this happens:

```
{ set  }* set  }
--------0-------
        B
--------2-------
       A
```

So as required, the star has been deleted, the set previously to its left has become active (or to be precise will become active when the active element reaches its leftmost element due to the system being in state A and the set to its left consisting only of 1s and 2s), and the 2 that was replaced in the element to the right is back as a 2 again.

5. This is the most complicated of the equivalences between the rules. On the left is what happens; compare it with the example on the right.

```
{ set  ** set  }    { set  *{ set  }
-------00-------    -------02-------
        A                   B
-------02-------    -------02-------
          B                   C
```

After this rule has happened in system 4, rule 3 is always the next rule to apply (because it leaves the active element as a set (there must be a set to the right of a star) and in state C). The top two rows in the example on the right shows the state that would result according to the initial conditions if the rule 5 changed the active state to state B and didn't add/remove the 1; the two rows below show the step after it. The final results are very similar, except that in the example on the left, the system has finished in state B but not state C. So the result is the same as in the

example on the right, except that all but the first element of the set to the right of the star are negated, and (due to additivity) the state that will be active when the element to the right of the set to the right of the star becomes active will be B instead of A or C, or A or C instead of B; in other words, apart from the set to the right of the star, the result is the same as if system 4's rule 5 had been “If a star is active and the system is in state C, the element to its right becomes active and the system remains in state C”. As for the changes to the set itself, the state after the active element is outside this set after this pseudo-scan will be the same as the example on the right (with a proper scan) except that all the elements but the first will be changed from a 1 to a 2 or vice versa, so it's left to show that doing this in system 3 is equivalent to adding/removing a 1 from the set to the right before rule 3 is applied (equivalent to adding/removing a 0 after rule 3 is applied, or to changing the parity ready for the next scan but leaving the original parity in subsequent scans in the system 3 equivalent). To show that this is the case, consider what happens when just the first element of a set of adjacent 1s and 2s of width 2^w is changed from a 1 to a 2 or vice versa in system 3. It will change the parity of the set at the next scan (because exactly one element has changed), and *all* of the elements of the set will end up with the opposite parity to what they would have had if the first element hadn't been changed (the first element because it had its parity changed, and subsequent elements because the change in the first element's parity causes them to change to the opposite parity to the parity they would have had otherwise). But as has already been established above, the change in every element's parity has no effect for the next 2^w-1 steps, and so with a sufficiently high value of w will never have any effect at all. Therefore, system 3 is capable of simulating a rule 5/rule 3 pair in system 4, and so can simulate every rule in system 4.

Because system 3 can simulate all details of system 4 for an arbitrary number of steps (with the right initial condition), it follows that if system 4 can emulate any two-colour cyclic tag system for an arbitrary number of steps, so can system 3 and therefore system 0.

Conjecture 5

The following system ('system 5') can emulate any two-colour cyclic tag system for an arbitrary number of steps, using a finite-size initial condition and finite-length list of rules in which at some point, the program attempts to add the rule after the finite number of rules given:

A program consists of an initial condition (a 'bag' of non-negative integers; that is, a set that allows multiple copies of an element to be contained), and a list of rules (a list of sets of positive integers). The initial condition can't contain any 0s to start with. The following steps are followed in order:

1. If there are any duplicates in the bag of integers, they are removed in pairs until either 1 or 0 of each integer remains.
2. Each integer in the bag of integers is decremented, and every integer in each of the rules is incremented.
3. If there is a 0 in the bag of integers, it is removed and replaced by the entire contents of the first rule, which is then deleted. (Subsequent replacements use the rules that were originally second, third, fourth, and so on.)
4. Go back to step 1 and repeat forever.

An example, to demonstrate the notation I use:

```
3,4,6 1,2 5,8 "" 9,10,14
```

It's a list of space-separated comma-separated lists; the first list is the initial bag, and the subsequent lists are the rules that the system uses. "" represents a rule with no integers.

Conjecture 5 implies conjecture 4 (and therefore conjecture 0)

Again, this implication is shown by constructing an initial condition that allows system 4 to emulate system 5 for an arbitrary number of steps, in such a way that the condition on the system 5 initial condition in conjecture 5 turns out to imply the condition on the system 4 initial condition in conjecture 4.

Initial condition

The initial condition consists of the following sets in order. All sets are separated by stars in this particular initial condition (although this is not a requirement of system 4, and many of the stars will be deleted during execution). The number f is a large positive integer; the larger the value of f, the more steps that can be simulated and the higher the numbers that can be involved in the calculation (to simulate for an arbitrary number of steps, it's a matter of selecting a sufficiently large value of f).

At the start:

- A set containing all integers that are twice a member of the initial bag after duplicates are removed in pairs, minus 2. (This set is the active element in the initial condition, in state A, fulfilling the condition in conjecture 4 that states just that.)
- f empty sets.

Repeated for each rule R:

- A set containing all integers from 0 to f_3, except those that are equal to f+3 plus twice an element of R, after duplicates have been removed from R in pairs, and possibly containing other integers higher than f_3 (it doesn't matter whether it does or not).
- f_2 empty sets.
- A set containing all integers from 0 to f_3, and possibly containing other integers higher than f_3 (it doesn't matter whether it does or not).
- f_2-2 empty sets.

Why the initial condition works

First, consider how consecutive sets in system 4 behave when not separated by a star (starting in state A in the initial condition). When the rightmost set is active in state A, all the sets going leftwards in turn will become active in state A until either a star or the left end of the tape is reached. The leftmost set must be the first to become active in state B or C (following the same reasoning as in the proof for Lemma 0), and then all the consecutive sets will become active in state B or C, going to the right. The state will change between state B and C a number of times equal to the number of 0s combined in the sets, so it only matters whether an odd or even number of 0s exist in the sets combined; and because all the sets are decremented simultaneously during this process, and nothing else can cause the sets to decrement, it only matters whether an odd or even number of any other integer exist in the sets combined either. In fact, the sets can be treated as one big set that contains all elements that the consecutive sets have an odd number of times between them; for instance,

```
*{1,2,3,4,5}{1,3,5}{3,4,6}*
```

is equivalent to

```
*{2,3,6}*
```

The conglomeration of sets formed at the start of the tape using this method in system 4 represents the bag of system 5, and it removes duplicates automatically for the reasons explained above, emulating step 1 of the process; to add a rule to the bag, all that is needed is to remove all stars between the bag and the rule. (For reasons explained later, the elements of this conglomeration of sets in system 4 are each twice the relevant element in system 5.)

To see how steps 2 and 3 work, consider what happens starting from the initial condition until the first star is reached. The leftmost conglomeration of sets (the bag) will be active in state A and then in state B; all its elements will be decremented (so that they become odd numbers), and then the first star will become active. There are two possibilities from here.

The first is that the bag didn't contain a 0 (containing a 0 = 2_1-2 is equivalent to containing a 1 before step 2, i.e. a 0 after step 2); in this case, the star will be reached in state B. The star will be removed, merging an empty set with the bag (which has no effect; f is assumed to be sufficiently large that it is an empty set that is reached and not a set containing elements) and the bag will become active in state A again. It will then become active in state B, all its elements will be decremented, and because it didn't contain a 0 (because all its elements were odd) the next star will also be removed, merging another empty

set with the bag, and the bag will become active again in state A. 2 has been subtracted from every element of the leftmost conglomeration of sets, which corresponds to decrementing every element of the bag in system 5. Note also that the number of empty sets to the left of the first non-empty set that's to the right of a star has been decreased by 2; define t to be f minus this value (so t starts at 0 and increases by 2 every time this possibility happens). The elements in the rules have not been increased yet, as step 2 would suggest; the value of t 'remembers' that the elements must be increased some time in the future. I show below that when a rule is added to the bag, the elements added to the bag are t/2 higher than the corresponding elements were in that rule in the initial condition, so that this increase in t really does have the effect of incrementing each rule. There wasn't a 0 in the bag, so step 3 should (and does) do nothing in this case; therefore, steps 2, 3, and 4 (because it returns to an initial-condition-like state from which the same behaviour happens) in system 5 are correctly emulated by system 4 in this case.

The other possibility is that there is a 0 in the bag. (During this description, the value of t remains fixed until the end of the emulation of steps 2, 3, and 4.) In this case, the star will be reached in state C. Therefore, the set to its right (an empty set) becomes active in state C and a 1 is added to it; then the 1 will be decremented to a 0, and the star to its right will become active in state C. This is repeated until the first non-empty set to the right of a star (i.e. the set representing the next rule to be added) is reached, at which point all f-t previously empty sets to its left will have been replaced with {0} (and all f-t+1 stars to its left will still exist). This non-empty set contains a 0 (in fact, it has been defined to contain all integers strictly less than f, and several others too), so as it becomes active in state C (after a 1 has been removed from it, as it contains a 1 for f>1), it will be decremented and the star to its right will become active in state B. That star is removed (merging an empty set with the non-empty set), the non-empty set and the star to its left become active, the star to its left is removed, and the non-empty set becomes active in state B. It now doesn't contain a 0, as its 1 was removed on the previous step, so it will be decremented and the next star to its right will become active in state B, and will be removed, merging another empty set to the non-empty set. (The merge of a {0} to the left of the non-empty set will remove its 0 for the next step too, so at this point the non-empty set has been decremented twice, and what would be its 0 'removed' (in fact, cancelled out); at this point, f-t stars and f-t-1 {0}s exist to its left, and f_2-2 empty sets exist to its right before the next non-empty set.)

Now, the following happens f-t-1 times: the system changes to state A and moves left until it encounters a star, it removes that star (causing a {0} to merge with the set the next time the system is in state A again), the non-empty set is decremented (it didn't contain a 0, because it didn't at the start of this loop and doesn't after each iteration, as a {0} will merge with it at the end of the loop, removing its own 0), a star becomes active to its right in state B, the star is removed (merging an empty set with the non-empty set) and the system changes back to state A. At the end of this loop, there is one star and no {0}s to the left of the non-empty set, and f_2-2-(f-t-1)=f+t-1 empty sets to its right; the non-empty set itself has been decremented f-t+1 times.

After the loop, the last star is going to be removed to its left, causing the non-empty set to merge with the bag the next time the system is in state A, it will be decremented (it didn't contain a 0 before this because it didn't after the last iteration of the loop) and one more star will be removed to its right and empty set merged, and the system changes to state A. This leaves the bag containing every integer from 0 to f_3-(f-t+2)=f_2+t-2 (which is assumed to be sufficiently large that the bag now contains every integer that ever becomes relevant during the simulation), except elements that were in the bag when the star was encountered in state C (these are all odd numbers from system 4's point of view and not a legal initial condition from system 5's), and elements that weren't in the non-empty set after it had been decremented f-t+2 times total (each such element was of the form f+3+2x for x in the rule that the non-empty set represents in the initial condition, and so is now of the form t+2x+1); and f+t-2 empty sets to the right of the bag before the next non-empty set.

The bag now becomes active in state A, so it immediately becomes active in state B (being the leftmost set); it contains a 0 (all the integers it doesn't contain that ever become relevant during the simulation are odd), so it is decremented and the star to its right becomes active in state C. This causes all f+t-2 empty sets to its right, up to the next non-empty set (which can be assumed to contain *every* integer that becomes relevant, as a sufficiently large value for f can be chosen), to become {0}, and the non-empty set itself to become active in state C, with its 1 removed (just as when the 0 in the bag was encountered originally). This non-empty set contains a 0, so as it becomes active in state C, it will be decremented and the star to its right will become active in state B. That star is removed (merging an empty set with the non-empty

set), the non-empty set and the star to its left become active, the star to its left is removed, and the non-empty set becomes active in state B. It now doesn't contain a 0, as its 1 was removed on the previous step, so it will be decremented and the next star to its right will become active in state B, and will be removed, merging another empty set to the non-empty set. (The merge of a {0} to the left of the non-empty set will remove its 0 for the next step too, so at this point the non-empty set has been decremented twice, and what would be its 0 'removed' (in fact, cancelled out); at this point, f+t-2 stars and f+t-3 {0}s exist to its left, and f_2-4 empty sets exist to its right before the next non-empty set.)

There will now be a loop, identical to the previous one except with f+t-3 iterations, which does the same thing for the same reasons. At the end of this loop, there is one star and no {0}s to the left of the non-empty set, and f_2-4-(f+t-3)=f-t-1 empty sets to its right before the next non-empty set (that represents the next rule).

After the loop, the last star is going to be removed to the left of the non-empty set, causing it to merge with the bag the next time the system is in state A. The non-empty set doesn't contain a 0 but contains every other integer that becomes relevant during the course of the simulation, so after it's decremented in state B, it contains every relevant integer and the star to its right becomes active in state B, causing it to be removed and the system to change to state A (merging an empty set with the bag and non-empty set that contains an integer), and the bag to become active in state A. The bag now contains a decremented version of every integer it didn't contain last time it was active; that is, 2 less than all the integers it contained the last-but-one time it was active, and all integers t+2x where x was an element of the rule to be added; this is equivalent to the combined behaviour of steps 2 and 3 in system 5. (An integer 2x-2 in the bag in system 4 corresponds to an integer x in the bag in system 5; so t+2x in system 4 corresponds to x+t/2+1 in system 5, i.e. an element of the rule, incremented once on each previous step and once more on this step. To see why the rules other than the rule being added have been incremented, simply observe that the new t is f-(f-(old t)-2), i.e. (old t)-2.)

Therefore, the initial condition given for system 4 accurately emulates all steps of the corresponding system 5 initial condition, and so system 4 can emulate system 5. In order to show that conjecture 5 implies conjecture 4 (which is already known to imply conjecture 0), it only remains to show that when the evolution of the system 4 initial condition leaves the defined portion, it is in state C. This is implied by the condition that at some point, the system 5 system must try to add a nonexistent rule, because doing this will follow all the steps in the description above up to the point where the set corresponding to that rule would be (but isn't), at which point the system 4 system is in state C.

Proof of conjecture 5 (and therefore of conjecture 0)

First, observe that any cyclic tag system can be emulated by a cyclic tag system in which elements of the strings to be added always occur in pairs, simply by doubling each element of each string to be added, doubling each element of the initial condition, and introducing a blank string to be added between each of the original strings to be added, including between the last and first (thus causing every second element to always be ignored).

Here is the definition and notation I will use for a two-colour cyclic tag system:

A two-colour cyclic tag system consists of an initial condition (a finite string of 1s and 0s, containing at least one element) (the initial working string), and a positive number of finite strings of 1s and 0s to be added, which have a set order. Repeatedly, the first element from the working string is removed, and the current first string to be added is moved to the end of the list; if a 1 was removed, a copy of that string is also concatenated to the end of the working string.

The notation shows the initial working string, followed by each string to be added in order (with "" substituting for an empty string to be added):

```
110 11 0 01 ""
```

As before, the proof is shown by stating an initial condition for system 5 that emulates any given cyclic tag system in which the elements of the strings to be added always occur in pairs, and proving that that initial condition works, and obeys the constraint set out in conjecture 5.

Initial condition

Each element in the working string corresponds to two integers in the bag; for a 0 in the working string, there is a difference of 1 between the integers, and for a 1 in the working string, there is a difference of 2 between the integers. Each integer representing a bit nearer the start of the working string must be lower than each integer representing a bit nearer the end of the working string (i.e. the integers that represent each bit in the working string going from the start to the end must increase in order). For instance, with the working string of 110 given as an example above, the elements of the initial bag corresponding to elements in the working string could be 1,3,4,6,7,8. The bag also contains some very large integers (it doesn't matter what they are, so long as they are sufficiently large and distinct, and a construction for the exact size they need to be is given in "How many steps?" below); one for each integer anywhere in the generated system 5 system, and one for each rule in the generated system 5 initial condition, plus one extra; how many of these integers are needed (and how high they have to be) depends on other details of the generated system 5 initial condition, but the other details of that initial condition don't depend on those integers, so they can simply be calculated and added in at the end.

As the cyclic tag system need be emulated only for a finite number of steps to prove conjecture 5, repeat the strings to be added sufficiently many times that the end of the repetition is never reached during the number of steps the system is to be emulated for; then, there is no need to emulate wrapping back to the first string. Each string to be added is represented by two rules. Each pair of identical elements of the string to be added corresponds to two integers in the second rule of each pair; for a 0 in the string to be added, there is a difference of 1 between the integers, and for a 1 in the string to be added, there is a difference of 3 between the integers. Each pair of integers in the second rule of each pair must have its lower integer higher by at least 3 than any integer in the initial bag, or in any previous rule (except the first rule of the pair), or that is part of the representation of any bit in that string to be added that comes before the bit in the string to be added that this pair of integers represents. The first rule in each pair consists of the second rule with 2 added to each integer. So for instance, the example cyclic tag system above

```
110 11 0 01 ""
```

is equivalent to the cyclic tag system

```
111100 1111 "" 00 "" 0011 "" "" ""
```

and using the construction above, one way to emulate this in system 5 for 8 steps (not counting the very large integers in the initial bag) (more steps could be obtained by repeating the strings to be added) is:

```
1,3,4,6,7,9,10,12,13,14,15,16 21,24,27,30 19,22,25,28 "" "" 35,36 33,34 "" "" 41,42,45,48
39,40,43,46 "" "" "" "" "" ""
```

Why the initial condition works

To prove that the initial condition works, it is only necessary to show that starting in an initial condition corresponding to one state of the cyclic tag system and running system 5 for some length of time, the system reaches an initial condition corresponding to the next state of the cyclic tag system. It also needs to be noted that the system will finally end by trying to add a rule that wasn't in the initial condition; this is because the very large integers in the initial bag cannot be removed due to being equal to each other as they're decremented at the same rate, so each must either remove another integer (as it happens, this case can't happen, but it's easier to prove that it works even if it could happen than to prove that it can't happen), or remove a rule, and even then there'll be at least one left over, so at some point a nonexistent rule must be added. There are two cases.

If the first element of the working string is a 0 (i.e. the two lowest integers in the bag differ by 1), then the next two occasions on which a rule is added to the bag will be one iteration of system 5's steps apart. Between the additions, the elements added from the first rule of the pair are decremented by 1, and those in the second rule of the pair are incremented by 1. As a result, when the second rule of the pair is added, each of its elements will correspond to an element added by the first rule. This will form a duplicate for each element, and as a result all the elements thus added will be removed. (The addition of the first rule of the pair couldn't itself have caused any elements to be removed, as each element in it is higher than any other element in the bag at the time, having started higher than any element that could have been added previously in the program and only been incremented since.) The system ends up with the first pair of rules (i.e. the first string to be added) having been removed (moving it to the end of the list of strings to be

added isn't necessary as the end of that list is never reached), and the 0 at the start of the string removed; as all the other rules have incremented at the same rate, and every integer in the bag has decremented by the same amount, the resulting string is an acceptable 'initial condition' for the situation after the 0 and rule have been removed, and this case is accurately emulated.

If the first element of the working string is a 1 (i.e. the two lowest integers in the bag differ by 2), then the next two occasions on which a rule is added to the bag will be two iterations of system 5's steps apart. Between the additions, the elements added from the first rule of the pair are decremented by 2, and those in the second rule of the pair are incremented by 2. As a result, when the second rule of the pair is added, each integer added will be 2 higher than the corresponding integer added by the first rule of the pair. All such integers added will be higher than any other integer in the bag at that time, as each element in the pair of rules is higher than any other element in the bag at the time, having started higher than any element that could have been added previously in the program and only been incremented since. So each pair of elements in the second rule of the pair, which corresponds to a pair of identical elements of the string to be added, converts into four integers in the bag; a 00 in the string to be added will cause integers x,x+1,x+2,x+3 to be added to the bag, and a 11 in the string to be added will cause integers x,x+2,x+3,x+5 to be added to the bag. Each of these patterns corresponds to a 00 or 11 respectively in the bag (two pairs of integers differing by 1 and by 2 respectively), and because a gap of 3 is left between pairs of integers within the same rule, they will be added so that their ascending order corresponds to adding the integers in the right order at the end of the bag. So in other words, the 1 at the start of the working string has been removed, and the contents of the first string to be added have been added at the end of the rule, and that string has been removed (moving it to the end of the list of strings to be added isn't necessary, because the end of that list is never reached).

As both cases have been covered, it has been shown that system 5 can emulate any two-colour cyclic tag system for an arbitrary number of steps; it has already been shown that for an arbitrary number of steps, system 4 can emulate system 5, system 3 can emulate system 4, and system 3 is equivalent to system 2, which is equivalent to system 1, which is equivalent to system 0. Therefore, system 0 can emulate any two-colour cyclic tag system for an arbitrary number of steps, and the extra condition that conjecture 0 requests also follows through each of the conjectures, and so conjecture 0 is true.

How many steps?

A concern left over from the introduction is calculating the values of w and f that are needed to emulate a cyclic tag system to the desired number of steps. Here are algorithms to calculate the required value, and proofs that the value works.

Calculating f

f affects two things, the spacing between non-empty sets in the resulting system 4 system (and therefore the maximum possible value of t) and how high the numbers in those sets go. First, it is easy to show that it is the maximum possible value of t that is the limiting factor; when a set becomes 'merged' to the leftmost set (i.e. all stars between it and the leftmost set are removed), it has been decremented f-t+2 times (i.e. at most f+2 times), and from then on it is decremented once every time t is increased, i.e. the set can be decremented at most f+2+t* times, where t is the maximum value of t required; assuming that f is at least 3 (as we're choosing the value of f, we can simply choose it to be at least 3), the set will be decremented at most f_2+2 times, so the elements in the set above f_3 will never become relevant. Therefore, it's f as a determiner of t* that's more important; so far it's been determined that any value of f such that $f \geq 3$ and $f \geq t^*$ will do, so that all that's needed is to determine t* (the maximum possible value of t that might be involved).

Every time t increases by 2, the elements in the leftmost conglomeration of sets in system 4 decrease by 2, corresponding to the elements in the bag in system 5 being decremented, that is one step of system 5. So t* is simply twice the maximum possible number of system 5 steps that need to be run before the system 'finishes' (either by running out of rules or by running out of elements in the bag). To determine this value, consider a system 5 system in which the maximum integer in the bag or any rule in the initial condition is M-1, and induct on the number of rules in the system.

The base case: For a system with 1 rule, either the bag must be empty (no steps needed), or its minimum

element must be less than or equal to M-1; so the rule will be added and the system will be out of rules after at most M steps, and 2t*=M=M_3^0. (Likewise, if the system has more than 1 rule, the first rule must be added after at least M steps.)

Induction step: Suppose a system with maximum value in the initial condition M-1 is run until a rule is added, which happens after at most s steps. Once the rule has been added, the maximum value in the initial condition is M+s-1 (because the highest integer possible in the bag just before the rule is added is M-s-1 and in a rule just before the rule is added is M+s-1). The reasoning in the base case shows that s=M; so assuming for induction that a system with n rules finishes in at most $3^{(n-1)}$_M steps, a system with n+1 rules must finish in $3^{(n-1)}$(M_2) steps (for it to finish after the first rule is added) plus $3^{(n-1)}$_M steps (for it to reach the point where the first rule is added), that is 3^n_M steps.

Therefore, by induction, a system 5 system in which the maximum integer in the bag or any rule in the initial condition is M-1, and with n rules, must finish executing after at most $3^{(n-1)}$_M steps, and therefore t* and therefore f is twice this value, 2_$3^{(n-1)}$_M (or 3 if this value is less than 3).

The extra added large integers in the system 5 initial condition

All that's needed with these integers is that they are so large they don't interfere while the cyclic tag system is still running (they come into play later, once the required number of steps have been run and/or the working string ends up empty). Therefore, they simply have to be higher than the maximum number of steps that the evolution of the rest of the system 5 system could take; this value has already been calculated in the section above (' a system 5 system in which the maximum integer in the bag or any rule in the initial condition is M-1, and with n rules, must finish executing after at most $3^{(n-1)}$_M steps'), and adding 1 to this number gives the minimum possible value of the extra added large integers.

Calculating w

Given that a system 4 program has been compiled from a system 5 program, choosing a value of w so that the system 3 program can emulate the system 4 program is actually quite easy; simply choose it so that 2^w is at least f_3. To see why this works, consider what the value of w actually does; its only effect on the program is that if the section of the system 3 program that corresponds to a set in system 4 is decremented more than 2^w times, it starts getting 'corrupted' and having the wrong parity. But it's already been determined in the previous section that sets will be decremented fewer than f_3 times (when determining that this was not the limiting factor), so as long as 2^w is at least f_3 it won't be the limiting factor either, f will be (if compiled from system 5) or the number of steps chosen to emulate when compiling from a cyclic tag system will be.

These calculations for f and w are clearly sufficiently simple not to be universal themselves, so the problem brought up in the introduction has now been solved.

From arbitrary to infinite

For convenience, conjecture 0 (which has now been proved) is restated below:

The Turing machine

```
-0- -0- -1- -1- -2- -2-
 A   B   A   B   A   B
-1- -2- -2- -2- -1- -0-
  B A   A     B A     A
```

('system 0') can emulate any two-colour cyclic tag system for an arbitrary number of steps, using a finite-length initial condition in which the leftmost cell starts active in state A and in which the first cell to become active after the emulation has finished is the cell to the right of the initial condition, and that cell becomes active in state A. (To be precise, 'finite-length initial condition', given in this and future conjectures, refers to an initial condition in which only finitely many of the cells are relevant, and no other cells are visited during the evolution of that initial condition.)

Given that this is true, it is possible to construct an initial condition for system 0 that emulates a two-colour cyclic tag system for an infinite number of steps. A preliminary result needs to be proved; that a finite region of system 0 cannot get into a loop (i.e. given any finite region of a system 0 tape, if the active

element starts in that region it must leave it eventually). This is more easily seen in the equivalent system 1; counting from the left-hand end of a region of the tape, it's possible to add together the positions of all 0s in that region of the tape, and by considering every rule in system 1 it can be shown if this value ever increases, it decreases to a lower value than the value it increased from on the previous step; in order for the system to change from state A to state B or vice versa, this value must decrease; and the system cannot get into a loop without changing from state A to state B or vice versa, because in all rules in which the system stays in state A the active element must move left and in all rules in which the system stays in state B the active element must move right. If a loop therefore could exist inside a finite region of system 1, charting the progress of the value obtained by adding together the positions of all 0s in that region of the tape whenever that value decreased would lead to an infinite strictly decreasing sequence of nonnegative integers, which is mathematically impossible.

Therefore, it is now known that any cyclic tag system can be emulated for a finite number of steps, in a finite region of a system 0 tape (starting with the leftmost element, a 0, active in state A), and some time after the emulation finishes (which it must do), eventually if the element to its right is a 0 it will become active in state A. To emulate a two-colour cyclic tag system for an *infinite* number of steps, all that is needed is to concatenate the system 0 initial condition to emulate it for 1 step with the system 0 initial condition to emulate it for 2 steps, for 3 steps, for 4 steps, and so on up to infinity, and start the entire system with the leftmost 0 of the first initial condition active in state A; in fact, any increasing sequence of integer numbers of steps will work. (What is in the initial tape to the left of that doesn't matter, as it never becomes active.) Each of these emulations will run one after the other, in sequence; for any given number of steps, the infinite concatenated initial condition will therefore emulate that program for that number of steps, and the number chosen does not have to be input beforehand or encoded in any way in the program itself; in other words, system 0 is now emulating the cyclic tag system for an infinite number of steps.

An initial condition obtainable in a non-universal way

The above proof leaves many options open in the initial condition that is used; in this section, I aim to show that at least one initial condition exists which is sufficiently simple to calculate that the calculation can be done by an obviously non-universal algorithm.

The way in which this is achieved is to create an initial condition that represents an emulation of the cyclic tag system for one cycle (in a way that can be shown to always terminate, and therefore definitely not be universal), which can be transformed using a simple and obviously non-universal rule into an initial condition that represents an emulation for two cycles, four cycles, and in general 2^n cycles. (Here a *cycle* refers to every rule in the cyclic tag system being applied exactly once, so that the rules end up back in their original order.) Note that the translation given from system 3 to system 0 cannot possibly be universal (one simple way to prove this is by observing that each element of the system 0 tape depends only on the corresponding element on the system 3 tape, whether that element is active, which side of the active element that element is if it isn't active, and what state the system starts in, which is a finite amount of data for each element), so it is sufficient to show that an initial condition for system 3 that emulates the cyclic tag system exists that can be calculated in a non-universal manner.

Before this is done, though, I will prove a lemma, which proves the validity of some of the constructions I will use below.

Lemma 2

If a string S of 1s and 2s of length 2^w in system 3 has a 1 in all positions but the leftmost 2^v (and possibly has 1s within those 2^v as well), and v is less than w, then

1. its parity set contains no integers equal to or greater than 2^v, and
2. a string of 1s and 2s of length 2^w in system 3 consisting of 2^v 1s, then the first 2^v elements of S, and then 1s for the rest of the string has a parity set consisting of the union of the parity set of S and the same set with 2^v added to each element, and a string consisting of the first 2^v elements of S, then the same elements again, and padded to width 2^w with 1s has a parity set consisting of the parity set of S with 2^v added to each element, and
3. a string of 1s and 2s of length 2^y (for integer y>v) in system 3 which has its first 2^v elements the same as S, and 1s everywhere else, has the same parity set as S.

Proof of lemma 2

(In the description below, XORing two strings together considers 2 to be the 'true' value and 1 to be the 'false' value, therefore obeying the truth table

	1	**2**
1	1	2
2	2	1

).

To see why part 1 of lemma 2 is correct, consider the cellular automaton given in method 2 of corollary 1. On the step which gives the string corresponding to parity set {0} (call it step 0), the rightmost 2 is the leftmost element (call it element 0), and if on a step the rightmost 2 is element n, and that element is not the rightmost element, then on the next step the rightmost element will be element n+1 (because if an element was a 1 on the previous step and had nothing but 1s to its right then (i.e. it was to the right of element n), then it will become a 2 if and only if it had a 2 to its immediate left (i.e. it was element n+1)); by induction, the rightmost 2 on step n is element n. Therefore, if the highest integer h in S is greater than or equal to 2^v, then on step h of the evolution of the cellular automaton the rightmost 2 will be element h (and therefore element h, outside the leftmost 2^v, will be a 2 on that step); also, no other steps of that cellular automaton that are XORed together to give S had element h as a 2 (because h was the highest element in the parity set, all previous steps had the rightmost 2 somewhere to the left of element h), contradicting the condition of the lemma, so part 1 of lemma 2 must be correct.

To see why part 2 of lemma 2 is correct, consider that the string given is equivalent to string S XORed with a string R that consists of the leftmost 2^v elements of S written twice and then filled to the same width as S with 1s, so all that is needed to show is that R's parity set is equivalent to S's with 2^v added to each element; this is equivalent to saying that running the method 2 cellular automaton for 2^v steps on S will give R. There are several ways to prove this (one is to use the binomial formula for Pascal's Triangle; if an expression E which is the sum of x^n for all n where element n of s is a 2 is constructed, then a step of the cellular automaton corresponds to multiplying E by (x+1) (mod 2), and 2^v steps corresponds to multiplying E by $(x+1)^{2^{\wedge}v}$, which is $(x^{2^{\wedge}v}+1)$ (mod 2) because all intermediate terms are even (because the exponentiation corresponds to squaring v times, and squaring (a+b) (mod 2) gives $(a^2+2ab+b^2)$ which is just (a^2+b^2)), which duplicates the terms of the first 2^v powers of x, corresponding to a duplication of the first 2^v elements of S and therefore to R).

To see why part 3 of lemma 2 is correct, simply observe that only the first 2^v steps of the cellular automaton are relevant in calculating S, and so elements to the right of element 2^v-1 never change from a 1, and so regardless of the value of w the system 3 string with a given parity set will be the same, except for 1s padding the right-hand-side.

The one-cycle initial condition

These are the choices made when calculating the one-cycle initial condition. The algorithms given for calculating the initial condition in the main proof above always terminate, therefore this step cannot be universal (or it couldn't have been proven to always terminate).

Choices made in the system 5 initial condition

The choices to be made here are which integers are used in the rules and initial bag, and which extra high integers are added to the initial bag (the number of cyclic tag rules being emulated has already been chosen, as one cycle). As it happens, it is irrelevant to this argument what choices are made for any of these but the extra high integers, but for the sake of definiteness take the minimum possible integer for each of these values. The algorithm for choosing the extra high integers is somewhat unusual, however; it consists of choosing an integer e, and causing the extra high integers to be the other integers in the bag with 2^e added (if this isn't enough, as it may well not be, the other extra high integers used are the original integers in the bag and the duplicates just added with 2^{e+1} added, then if that isn't enough all the integers in the bag (the originals and all the duplicates added so far) with 2^{e+2} added, and so on until there are enough); e is simply chosen so that 2^e is sufficiently high (and what is sufficiently high has been defined above).

Choices made in the system 4 initial condition

The choices made here are the value of f, and which elements above 3f are in the 'large sets' which have the elements from 0 to 3f specified. In this case, f is chosen to be any power of 2 that's sufficiently large (again, what is sufficiently large for this has been defined above), and the large sets contain, apart from the elements that are fixed by the definition of the system 4 initial condition, all integers from 3f+1 to 4f-1 inclusive (but no other elements).

Choices made in the system 3 initial condition

The only choices here are the value of w (it takes its minimum allowable value, $2+\log_2 f$, which is an integer because f has been defined to be a power of 2), and the representation used for the left end of the system 4 tape (obviously the '02222...' form, because this is required for infinite emulation of a cyclic tag system).

When constructing the system 3 initial condition that corresponds to the cyclic tag system for one cycle, though, one extra piece of information has to be remembered; what the system 3 string corresponding to the system 5 initial bag would have been if no extra high integers have been added (this becomes relevant later).

Transforming the initial condition for n cycles into a condition for 2n cycles

This is a matter of working out what changes are needed to convert the n-cycle initial condition into a 2n-cycle initial condition; the original change (in the cyclic tag system) is simply to double the number of cycles of the cyclic tag system that are emulated.

Changes in the system 5 initial condition

The two things that have to change here are the list of rules and the extra high integers added to the bag.

- The list of rules needs to double in length, with the new rules having their lowest elements 3 higher than the highest elements of the old rules, and the old rules untouched. The specific method to do this that I'll use here is to add 2^w to each element of the old rules, where the value of w from the previous initial condition is used; 2^w is at least 3f, which is at least 6M, where M is the highest element in the old rules, so this is guaranteed to be high enough.
- More extra high integers are needed, and they need to be higher. Doubling their number will ensure that there are enough, because the number elements in rules has doubled and the number of elements in the initial bag has remained the same. Instead of transforming the extra high integers, the old extra high integers are discarded, and new ones created using the same method as in the initial condition; that is, the initial bag is duplicated with 2^e added to each element, with 2^{e+1} added to each element in the new bag, and so on, until there are enough; the number of duplicates needed cannot be more than twice the number of duplicates as was needed in the one-cycle initial condition for each time the number of cycles has doubled (because twice the number of duplicates would double the number even if the duplicates added when 2^e was added were not duplicated again when the duplicates for 2^{e+1} were added). Calculating the value of e is non-trivial; 2^e must be at least $3^{(r-1)}\_M$ for the new value of M, where r is the new number of rules. The highest element in the new rules is the highest element in the old rules, plus 2 to the power of the old w; this means that with the old w and new r, it's sufficient to make 2^e at least $3^{r-}\_2^w$ (because this is higher than $3^{(r-1)}\_2^{w+1}$, and the old 2^w has to be higher than the old M); one value that accomplishes this is $4^r\_2^w$, which gives a value of 2r+w for e (with the new r and old w).

Changes in the system 4 initial condition

To change the system 4 initial condition to accommodate more cycles, the value of f needs to increase, which affects the number of duplications of empty sets and the value up to which the sets go; also the number of groups of sets that represent system 5 sets needs to increase to allow for the new rules added to the system 5 initial condition, and the first set needs to change due to the changes in the system 5 bag.

- The value of f needs to be at least $2\_3^{(r-1)}\_M$, where r is the new number of rules in the system 5

initial condition and M is its new maximum value. The extra high integers are by definition the highest integers anywhere in the system 5 initial condition. Suppose d is the number of duplications used to add the extra high integers in the system 5 initial condition; M will therefore be less than 2^{e+d}, because 2^e is obviously higher than any elements in the initial bag, and they will have had at most 2^{e+d-1} added to them to create the extra high integers. So $\log_2$ M is at most 2r+w+d. 2_$3^{(r-1)}$ is less than 4^r, so a valid value for $\log_2$ f is 4r+w+d (with the new r and d, and old w).

- As in the one-cycle initial condition, each set which the rules for constructing a system 4 initial condition specify integers up to 3f also contain integers from 3f+1 to 4f-1 inclusive, and no other integers. The difference between the old and new values of f has to be added to each 'gap' where those sets don't contain a particular element.
- The padding of null sets and stars between sets containing lots of integers needs to change according to the new value of f; it's easiest just to recalculate it once the new value of f is known.
- The new sets and stars to be added are the same as the old ones (not counting the first set and its associated padding), just with 2^{w+1} (using the old w) added to the 'gaps' in each of the non-null sets.
- The first set (corresponding to the system 5 bag) will be the same as before, except that the extra copies of it that are added will use the new values of e and d.

Changes in the system 3 initial condition

The new value of w is easy to calculate; it's just 2+$\log_2$ f (taking the new value of f). This has a knock-on effect on the translation of every set and the region representing the left end of the system 4 tape, and of course the changes in the system 4 initial condition also have to be mirrored by changes to the system 3 initial condition.

- The number of repetitions of '0222...2222' increases by the ratio between the old and new 2^w, as does the length of each repeated element (totalling the 0 and all the 2s).
- The 'gaps' in parity sets in the system 3 strings corresponding to non-null system 4 sets need to be increased by the difference in the old and new values of f, and their maximum value needs to increase. An obviously non-universal algorithm to do this (Lemma 2 explains why this set of steps has the desired effect):
  - Change the 0 in the set that represents the star at the end to a 2, if there is one;
  - Undo the operation used to cause the set to start and end with a 2 (all the possibilities for this are self-inverse, so they're easy to undo, and examining the last 2 elements of the string will reveal which one was used, as they come out different depending on which method was used, and must have been “12” beforehand because 2^w-1 and 2^w-2 with the old w were both in its parity set, due to the definition of the old w, and no other elements of the parity set can effect what those two elements have to be);
  - Replace the 2 at the end of each such set with a 1, which complements its parity set;
  - Remove the duplication of the sequence where the repeated section starts at the old element f (which is always possible because that's how this set was constructed in the first place), which has the effect of subtracting the old f from each element of the new parity set (i.e. each gap in the original parity set);
  - Increase the length of the sequence to the new 2^w by padding with 1s;
  - Add a new duplication of the left end of the sequence at the new element f (so now the elements in the parity set have increased by the right amount net);
  - Replace the 1 at the end of the subset with a 2, turning the elements in the parity set back into gaps, and also causing the set to go up to 4f-1 with the new f rather than 4f-1 with the old f;
  - Cause the set to start and end with 2, using the method detailed in conjecture 4's proof;
  - Change the relevant 2 in the set back into a 0, if a star was removed to start with.

- The strings corresponding to what in system 4 are the null sets and stars that correspond to padding need to be widened for the new value of w (the null set is, however, just 2^w 1s by corollary 1's method 2, or that many 2s when changed to start and end with 2, and so widening it is easy), and more repetitions are needed (the number of repetitions needed is multiplied by the ratio of the old and new values of f).
- The strings corresponding to the new sets that are added in system 4 are obtainable by shifting the gaps in the existing parity sets. The amount by which the gaps have to be shifted is a power of 2 (2^{w+1} using the old w, in fact); the gaps can be shifted using basically the same algorithm as detailed in the combined gap-shift-and-lengthen algorithm shown two bullet points above (the only difference is that the fourth and fifth steps in the given sequence are left out, and the old 2^{w+1} rather than the new f are used in the sixth step).
- The string corresponding to the system 5 bag (that is, the first system 4 set) needs be widened, and to have its parity set changed to change what the extra large integers are. To do this, the without-extra-large-integers string that was remembered when constructing the one-cycle initial condition is used; it's widened to the new 2^w by right-padding it with 1s (lemma 2 says that this doesn't change its parity set), and then its parity set is duplicate with 2^{e+1}, with 2^{e+2}, and so on added (the new d times in total) by using the algorithm given in lemma 2 (i.e. move its elements 2^{e+1}, then 2^{e+2}, etc. to the left).

d, e, r, w, and f

The only place now that universality could exist in the algorithm used to find the initial condition is in d, e, r, w, and f, as they have all been defined in terms of each other's previous values, and as they're unbounded integers, they could in some problem potentially store enough information that they could between them in principle form a universal system. However, that is not the case in this system; the easiest way to demonstrate this is by deducing a formula for each of them. In what's written below, d_n refers to the value of d in the 2^n-cycle initial condition, and likewise for the other variables.

d doubles with each doubling of the number of cycles, so $d_n = 2^n d_0$.

Likewise, r doubles with each doubling of the number of cycles; so $r_n = 2^n r_0$.

$\log_2 f_n$ is $4r_n+w_{n-1}+d_n$, and as w_{n-1} is $2+\log_2 f_{n-1}$, $\log_2 f_n$ is $2^{n+2}r_0+(2+\log_2 f_{n-1})+2^n d_0$. This is $\log_2 f_{n-1}+2+2^n(4r_0+d_0)$, so
$\log_2 f_n$ is $\log_2 f_0$ plus the sum with i from 1 to n of $2+2^i(4r_0+d_0)$, or $(\log_2 f_0)+2n+(2^{n+1}-1)(4r_0+d_0)$, and f_n is 2 to the power of that expression.

w_n is therefore $(\log_2 f_0)+2n+(2^{n+1}-1)(4r_0+d_0)+2$.

Finally, e_n is $2^{n+1}r_0+(\log_2 f_0)+2n+(2^n-1)(4r_0+d_0)$ (which can be obtained simply by substituting the other formulae obtained into its own formula).

Therefore, this algorithm for determining an initial condition, whilst somewhat complicated, is definitely not itself universal, and so the universality discovered above is a property of system 0, and not of the algorithm used to find its initial condition.

Programs and examples

These programs and examples were from an older version of the proof; I haven't changed the definitions of any of the systems mentioned above since they were written, so the interpreters will still work without problems, but the programs for carrying out the constructions of an initial condition in one system based on the initial condition in another system fail to take the changes needed to allow the resulting programs to be concatenated (and therefore to allow infinite emulation of a cyclic tag system) into account (for instance, the program showing the system 4 to system 3 conversion just places a lot of 0s and 221 at the start as a method of changing state B to state A at the left-hand end of the system 4 tape; this works, but doesn't allow the program to be chained after another program by concatenation, and the program hasn't been updated to allow for the more complicated representation of the left-hand end of the tape needed to allow the program to be run automatically after the previous program ends). I have removed a section that was moved to the main proof.

I have written several Perl programs, to demonstrate the constructions given in the proof and to interpret the systems given in various conjectures. They all take their input directly on the command line (so cytag.pl 11011 101 01 0 "" 010 runs the cyclic tag system 11011 101 01 0 "" 010); to give them input from a file, write F and a space before the filename as the last thing on the command line (for instance, cytag.pl F test1.cy). They also all optionally take a letter as their first argument, to change the way in which the output is displayed (different methods of displaying the output can be clearer for different purposes); for instance, cytag.pl Y 11011 101 01 0 "" 010 will double each character of the output (output format Y), thus displaying the output that would be produced by the equivalent system which doubles each element of the working string and strings to be added, and adds a null string between each string to be added (and thus becoming closer to the output of the equivalent system 5 program; in fact, if test1.cy and test1.s5 are equivalent, cytag.pl Y F test1.cy produces exactly the same output as system5.pl Y F test1.s5). In examples of output shown, lines starting with a $ are lines that I typed to show how I invoked the relevant programs.

Two-colour cyclic tag system

The following program ('cytag.pl') emulates a cyclic tag system.

```
#!/bin/perl -w

use strict;

my $working;
my @rules;
my $rulecount=0;
my $temp;
my $temp2;
my $doubling=0;

# If a Y is given as the first argument, double each element of the output to
# show the similarity to system 5
$ARGV[0]eq'Y' and do{$doubling=1; shift @ARGV;};

# Read from file
$ARGV[0]eq'F' and do{
    shift @ARGV;
    my $file = shift @ARGV;
    local $/ = undef;
    open INFILE, $file;
    @ARGV = split(' ',<INFILE>);
    close INFILE;
};

$|=1;

# Load the working string
$working=shift @ARGV;
chomp $working;

# Load the rules
for my $e (@ARGV)
{
    chomp $e;
    $e eq '""' and $e = '';
    $rules[$rulecount++]=$e;
}
while($working ne '')
{
    # Remove and print the first element of the working string
    $temp=substr $working,0,1,'';
    print $temp;
    $doubling and print $temp;
    push @rules, ($temp2 = shift @rules);
    $temp eq '1' and $working .= $temp2;
}
print "\n";
```

The default output format is to print a 0 or 1 whenever a 0 or 1 respectively is removed from the start of the working string, so that the output gives a history of the working string over time. If a Y is given as the first argument, each such 0 or 1 is doubled, producing the same output as the doubled-element equivalent

of the system would without the Y. Here is an example two-element cyclic tag program:

```
11011 101 01 0 "" 010
```

When this program ('test1.cy') is run (with and without the Y option for formatting), the output is as follows:

```
$ cytag.pl F test1.cy
1101110101010101001001101010010010100010
$ cytag.pl Y F test1.cy
1111001111110011001100110011001100001100001111001100110000110000110011000000001100
```

In this case, the system terminates after 40 steps. (If I'd chosen a non-terminating system, there would have been an infinite amount of output.) The interpreter stops once the working string is of zero length (the most obvious 'termination state' for a cyclic tag system).

Emulating a two-colour cyclic tag system with system 5

I gave a construction for a system 5 initial condition that emulates a two-colour cyclic tag system above. Here's a program ('cy2s5.pl') that does the construction (doubling the cyclic tag system and converting it to system 5):

```
#!/bin/perl -w

use strict;

my $temp;
my $temp2;
my $i=1;
my $n=shift @ARGV;
my @ARGVtemp;

# Read from file
$ARGV[0]eq'F' and do{
    shift @ARGV;
    my $file = shift @ARGV;
    local $/ = undef;
    open INFILE, $file;
    @ARGV = split(' ',<INFILE>);
    close INFILE;
};

$|=1;

$temp=shift @ARGV;
chomp $temp;
$temp =~ s/ //g;

while($temp ne '')
{
    $i==1 or print ",";
    # Doubled elements
    substr($temp,0,1)eq'0' ?
        (print($i,",",$i+1,","),($i+=2)) :
            (print($i,",",$i+2,","),($i+=3));
    substr($temp,0,1,'')eq'0' ?
        (print($i,",",$i+1),($i+=2)) :
            (print($i,",",$i+2),($i+=3));
}

$i+=2;

@ARGVtemp=@ARGV;

while($n>0)
{
    for my $e (@ARGV)
    {
        chomp $e;
        $e =~ s/ //g;
        $e eq '""' and $e = '';
        print " ";
        $temp=$temp2="";
        while($e ne '')
```

```
        {
            # This automatically 'doubles' the elements
            if((substr $e,0,1,'')eq'0')
            {
                $temp .= ($i+2) . "," . ($i+3);
                $temp2 .= $i . "," . ($i+1);
                $i+=4;
            }
            else
            {
                $temp .= ($i+2) . "," . ($i+5);
                $temp2 .= $i . "," . ($i+3);
                $i+=6;
            }
            $e ne '' and $temp.="," and $temp2.=",";
        }
        $temp eq '' and $temp = '""';
        $temp2 eq '' and $temp2 = '""';
        print $temp," ",$temp2,' "" ""'; # and add a blank rule afterwards
        $n--;
        $n==0 and last;
    }
    @ARGV=@ARGVtemp;
}
```

The first argument gives the number of steps that the emulation has to run for (any arbitrary positive integer can be chosen, because the emulation works for an arbitrary but not an infinite number of steps); the remaining arguments give the cyclic tag program to emulate. In the case of the example cyclic tag system shown above, we know it terminates after 40 steps, so asking for a 50-step emulation should correctly emulate it through to the end. This is what happens:

```
$ cy2s5.pl 50 F test1.cy > test1.s5
$ cat test1.s5
1,3,4,6,7,9,10,12,13,14,15,16,17,19,20,22,23,25,26,28  33,36,39,40,43,46 31,34,37,38,41,44
"" "" 49,50,53,56 47,48,51,54 "" "" 59,60 57,58 "" "" "" "" "" "" 63,64,67,70,73,74
61,62,65,68,71,72 "" "" 77,80,83,84,87,90 75,78,81,82,85,88 "" "" 93,94,97,100 91,92,95,98
"" "" 103,104 101,102 "" "" "" "" "" "" 107,108,111,114,117,118 105,106,109,112,115,116 ""
"" 121,124,127,128,131,134 119,122,125,126,129,132 "" "" 137,138,141,144 135,136,139,142 ""
"" 147,148 145,146 "" "" "" "" "" "" 151,152,155,158,161,162 149,150,153,156,159,160 "" ""
165,168,171,172,175,178 163,166,169,170,173,176 "" "" 181,182,185,188 179,180,183,186 "" ""
191,192 189,190 "" "" "" "" "" "" 195,196,199,202,205,206 193,194,197,200,203,204 "" ""
209,212,215,216,219,222 207,210,213,214,217,220 "" "" 225,226,229,232 223,224,227,230 "" ""
235,236 233,234 "" "" "" "" "" "" 239,240,243,246,249,250 237,238,241,244,247,248 "" ""
253,256,259,260,263,266 251,254,257,258,261,264 "" "" 269,270,273,276 267,268,271,274 "" ""
279,280 277,278 "" "" "" "" "" "" 283,284,287,290,293,294 281,282,285,288,291,292 "" ""
297,300,303,304,307,310 295,298,301,302,305,308 "" "" 313,314,317,320 311,312,315,318 "" ""
323,324 321,322 "" "" "" "" "" "" 327,328,331,334,337,338 325,326,329,332,335,336 "" ""
341,344,347,348,351,354 339,342,345,346,349,352 "" "" 357,358,361,364 355,356,359,362 "" ""
367,368 365,366 "" "" "" "" "" "" 371,372,375,378,381,382 369,370,373,376,379,380 "" ""
385,388,391,392,395,398 383,386,389,390,393,396 "" "" 401,402,405,408 399,400,403,406 "" ""
411,412 409,410 "" "" "" "" "" "" 415,416,419,422,425,426 413,414,417,420,423,424 "" ""
429,432,435,436,439,442 427,430,433,434,437,440 "" "" 445,446,449,452 443,444,447,450 "" ""
455,456 453,454 "" "" "" "" "" "" 459,460,463,466,469,470 457,458,461,464,467,468 "" ""
```

(I captured the output of that in a file, because I'll need to use it later; the cat command on the second line shows what the output was.) Just looking at the start of the output (the initial bag and first rule) demonstrates the doubling of the initial working string and replacement with pairs of integers to form a bag, and likewise in a rule:

```
1,3,4,6,7,9,10,12,13,14,15,16,17,19,20,22,23,25,26,28    (initial bag)
 1   1   1    1     0     0     1     1     1     1     (the corresponding working string
is 11011)
33,36,39,40,43,46                                        (first rule of the pair)
31,34,37,38,41,44                                        (second rule of the pair)
  11    00    11                                         (the corresponding string to be
added is 101)
```

A shorter example will also be useful later to demonstrate some of the more long-winded output formats (and also, of course, a system 5 program is much longer than a cyclic tag program it emulates, and likewise for system 4 emulating a system 5 program and system 3 emulating a system 4 program), so here it is:

```
$ cy2s5.pl 3 01 1 10
1,2,3,4,5,7,8,10 15,18 13,16 "" "" 21,24,27,28 19,22,25,26 "" "" 31,34 29,32 "" ""
```

System 5

Here is the interpreter ('system5.pl') I've been using for system 5:

```
#!/bin/perl -w

use strict;

my %bag;
my %temp;
my @rules;
my $i=0;
my $cpr=0;

# Compressed output that shows the similarity to system 0
$ARGV[0]eq'C' and do{$cpr=1; shift @ARGV;};

# Compressed output that shows the similarity to a cyclic-tag system
$ARGV[0]eq'Y' and do{$cpr=3; shift @ARGV;};

# Load up the bag
chomp $ARGV[0];
for my $e (split(',',(shift @ARGV)))
{
    defined($bag{$e}) or $bag{$e}=0;
    $bag{$e}++;
}
# Load up the rules
for my $e (@ARGV)
{
    my %v;
    chomp $e;
    $e eq '""' and $e = '';
    for my $k (split(',',$e))
    {
        defined($v{$k}) or $v{$k}=0;
        $v{$k}++;
    }
    $rules[$i++]=\%v;
}
while(1)
{
    # Remove duplicates
    for my $e (keys %bag)
    {
        $bag{$e} % 2 or delete $bag{$e};
    }
    # Decrement the entire bag
    %temp=();
    for my $e (keys %bag) {$temp{$e-1}=$bag{$e}};
    %bag=%temp;
    # Print the bag
    if(!$cpr) {print +(join ',',sort {$a <=> $b} keys %bag);}
    # Increment each rule
    for my $e (@rules)
    {
        %temp=();
        for my $k (keys %$e)
        {
            $temp{$k+1}=$$e{$k};
        }
        %$e=%temp;
    }
    # Print each rule
    if(!$cpr)
    {
        for my $e (@rules)
        {
            print " ",(join ',',sort {$a <=> $b} keys %$e);
            keys %$e or print '""';
        }
        print "\n";
    }
    # If there's a 0 in the bag...
    $cpr==2 and $cpr=1;
```

```
    $cpr>=4 and $cpr++;
    if(defined($bag{0})&&$bag{0})
    {
        delete $bag{0};
        %temp=%{shift @rules};
        for my $e (keys %temp)
        {
            defined($bag{$e}) or $bag{$e}=0;
            $bag{$e}+=$temp{$e};
        }
        $cpr==1 and $cpr=2;
        if($cpr>4)
        {
            # two consecutive removals prints 0, two removals separated by one
            # step prints 1, and so on. When this is emulating a cyclic tag
            # system, there should be no higher numbers output; otherwise, put
            # the number in parentheses, both to show it's an anomaly and to
            # distinguish 1 then 0 from 10.
            $cpr<7 ? print $cpr-5 : print "(",$cpr-5,")";
            $cpr=3;
        }
        else {$cpr==3 and $cpr=4;}
    }
    $cpr==2 and print '00';
    $cpr==1 and print '11';
    if(!(keys %bag)||!defined($rules[0])) {last;}
}
$cpr==1||$cpr==2 and print "111...\n";
$cpr>2 and print "\n";
```

The default output format shows everything that's going on (the backquotes here are used to directly substitute the output of cy2s5.pl into the input of system5.pl; this is functionality of the shell I was using, not of the program). The notes in *italic* are mine to explain what's going on, and not part of the output; the bolded part of the output shows what the note refers to.

```
$ system5.pl `cy2s5.pl 3 01 1 10`
0,1,2,3,4,6,7,9 16,19 14,17 "" "" 22,25,28,29 20,23,26,27 "" "" 32,35 30,33 "" ""
The 0 at the start of the initial working string translates to two integers that differ by
1. There's a 0 here, which means that the first rule is about to be added. (The line is
printed immediately after step 2 of system 5.)
0,1,2,3,5,6,8,15,18 15,18 "" "" 23,26,29,30 21,24,27,28 "" "" 33,36 31,34 "" ""
This is what a 0 in the working string does; the 16,19 has been decremented, the 14,17 has
been incremented, and they both now say 15,18 and cancel each other out. So the 0 manages
to remove a rule.
0,1,2,4,5,7 "" "" 24,27,30,31 22,25,28,29 "" "" 34,37 32,35 "" ""
The 0 in the initial working string was doubled, so we're just about to remove a blank
rule.
0,1,3,4,6 "" 25,28,31,32 23,26,29,30 "" "" 35,38 33,36 "" ""
0,2,3,5 26,29,32,33 24,27,30,31 "" "" 36,39 34,37 "" ""
Now, there's a 1 at the start of the working string, so something will be added to the end
of the string.
1,2,4,25,28,31,32 25,28,31,32 "" "" 37,40 35,38 "" ""
0,1,3,24,27,30,31 26,29,32,33 "" "" 38,41 36,39 "" ""
Two steps later, the next rule is added. This time, it isn't identical to the other rule in
the pair because of the extra delay.
0,2,23,25,26,28,29,30,31,32 "" "" 39,42 37,40 "" ""
The added rules produce the equivalent of 1100, corresponding to the 10 in the cyclic tag
system.
1,22,24,25,27,28,29,30,31 "" 40,43 38,41 "" ""
0,21,23,24,26,27,28,29,30 "" 41,44 39,42 "" ""
Another null rule is removed. The rest of the evolution of the system isn't particularly
interesting.
20,22,23,25,26,27,28,29 42,45 40,43 "" ""
19,21,22,24,25,26,27,28 43,46 41,44 "" ""
18,20,21,23,24,25,26,27 44,47 42,45 "" ""
17,19,20,22,23,24,25,26 45,48 43,46 "" ""
16,18,19,21,22,23,24,25 46,49 44,47 "" ""
15,17,18,20,21,22,23,24 47,50 45,48 "" ""
14,16,17,19,20,21,22,23 48,51 46,49 "" ""
13,15,16,18,19,20,21,22 49,52 47,50 "" ""
12,14,15,17,18,19,20,21 50,53 48,51 "" ""
11,13,14,16,17,18,19,20 51,54 49,52 "" ""
10,12,13,15,16,17,18,19 52,55 50,53 "" ""
9,11,12,14,15,16,17,18 53,56 51,54 "" ""
8,10,11,13,14,15,16,17 54,57 52,55 "" ""
```

```
7,9,10,12,13,14,15,16 55,58 53,56 "" ""
6,8,9,11,12,13,14,15 56,59 54,57 "" ""
5,7,8,10,11,12,13,14 57,60 55,58 "" ""
4,6,7,9,10,11,12,13 58,61 56,59 "" ""
3,5,6,8,9,10,11,12 59,62 57,60 "" ""
2,4,5,7,8,9,10,11 60,63 58,61 "" ""
1,3,4,6,7,8,9,10 61,64 59,62 "" ""
0,2,3,5,6,7,8,9 62,65 60,63 "" ""
1,2,4,5,6,7,8,61,64 61,64 "" ""
0,1,3,4,5,6,7,60,63 62,65 "" ""
0,2,3,4,5,6,59,61,62,64 "" ""
1,2,3,4,5,58,60,61,63 ""
0,1,2,3,4,57,59,60,62 ""
```

This format, while useful to see how the emulation of a cyclic tag system with system 5 works, is far too lengthy for practical use, so two much shorter formats are available:

```
$ system5.pl C `cy2s5.pl 3 01 1 10`
00000000001100001100111111111111111111111111111111111111111111001100001100111...
$ system5.pl Y F test1.s5
11110011111100110011001100110011000011000011110011001100001100001100110000001100
```

The C format prints 00 whenever there's a 0 in the bag after step 2, and 11 whenever there isn't. (This is for comparison with the output of systems 0 through 4.) The Y format counts the number of steps *between* (i.e not counting either end) the first rule and the second rule of a pair of rules being added; therefore, a 0 represents a 0 being removed from the left-end of the working string in an emulated cyclic tag system, and likewise a 1 represents a 1 being removed from the left-end of the working string. (Any numbers of steps between pairs of rules above 1 are printed in decimal in parentheses, to show where the edge of multiple-digit numbers are.) As a comparison:

```
$ cytag.pl Y F test1.cy
11110011111100110011001100110011000011000011110011001100001100001100110000001100
```

This shows that at least in this case, the emulation of test1.cy proceeds the same way as test1.cy itself does (the proof above is, of course, needed to show this in general.)

Emulating system 5 with system 4

The notation used in the proof for system 4 isn't all that appropriate for entry into a computer (because it consists of multiple lines), so the following notation will be used:

```
{0,6,8}{}*{3,4}*{8}{11,20}...    (notation used in the proof)
          B
0,6,8 "" _ B 3,4 _ 8 11,20       (notation used by the program)
```

The active element is shown by writing the state letter to its left; the braces have been replaced by spaces, and an underscore is used instead of an asterisk to represent a star (because * has a special meaning in many shells). System 4 programs are supposed to be right-infinite, but obviously this cannot be achieved in practice; instead, the interpreter simply exits when the active element would be past the right-hand end of the program. (Because the number of stars in the program is non-increasing, and a star is always removed when active in state A (the only state that moves to the left), the active element always ends up past the right-hand end of the program eventually.)

Here is the program I use that implements the construction given in the proof above to emulate system 5 with system 4 ('s52s4.pl'):

```
#!/bin/perl -w

use strict;

my %temp;
my $f;
my $i;

# Read the value of f
$f = shift @ARGV;
chomp $f;

# Read from file
$ARGV[0]eq'F' and do{
    shift @ARGV;
    my $file = shift @ARGV;
```

```
    local $/ = undef;
    open INFILE, $file;
    @ARGV = split(' ',<INFILE>);
    close INFILE;
};

$|=1;

# Initial bag
chomp $ARGV[0];
for my $e (split(',',(shift @ARGV)))
{
    defined($temp{$e*2-2}) or $temp{$e*2-2}=0;
    $temp{$e*2-2}++;
    $temp{$e*2-2}==2 and delete $temp{$e*2-2};
}
print "A ";
print +(join ',',sort {$a <=> $b} keys %temp);

# f empty sets
$i=$f;
print ' _ ""' while $i--;

# Each rule
for my $e (@ARGV)
{
    chomp $e;
    $e eq '""' and $e = '';
    undef %temp;
    $i=$f*3+1;
    $temp{$i}=1 while $i--; # contains all integers from 0 to f*3
    for my $k (split(',',$e))
    {
        defined($temp{$k*2+$f+3}) or $temp{$k*2+$f+3}=0;
        $temp{$k*2+$f+3}++; # except those in the rule an odd number of times
        $temp{$k*2+$f+3}==2 and delete $temp{$k*2+$f+3};
    }
    print ' _ ';
    print +(join ',',sort {$a <=> $b} keys %temp); # must contain at least a 0
    # f*2 empty sets
    $i=$f*2;
    print ' _ ""' while $i--;
    # A set containing all integers from 0 to 3*f
    print ' _ ';
    $i=-1;
    print "$i," while ++$i<3*$f;
    print 3*$f;
    # f*2-2 empty sets
    $i=$f*2-2;
    print ' _ ""' while $i--;
}
```

The first argument given is the value of f to use in the emulation. (This has to be quite high to emulate a cyclic tag system for a reasonable number of steps, because the system 5 equivalent of the cyclic tag system that has to be translated is normally quite long.). This is followed by the system 5 system to translate, or an F and the file in which it's stored. For instance:

```
$ s52s4.pl 16 2 1,4 1,6 "" ""
A 2  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,22,23,24,25,26,28,29,30,31,32,33,34,35
,36,37,38,39,40,41,42,43,44,45,46,47,48  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _
"" _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _
"" _ "" _ "" _ "" _
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33
,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""
_ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _
"" _ ""  _ ""
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,22,23,24,25,26,27,28,29,30,32,33,34,35
,36,37,38,39,40,41,42,43,44,45,46,47,48  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _
"" _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _
"" _ "" _ "" _ "" _
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33
,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""
_ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _ ""  _
"" _ ""  _ ""
```

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33
,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48  _"" _"" _"" _"" _"" _"" _"" _"" _"" _"" _""
 "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "
"  _"  _"  _"  _"  _"  _  _"  _  _"  _  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _
"" _ _ _ _ _ _ _
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33
,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48  _"" _"" _"" _"" _"" _"" _"" _"" _"" _"" _""
 "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "
_  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _
"" _ "" _ ""
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33
,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48  _"" _"" _"" _"" _"" _"" _"" _"" _"" _"" _""
 "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "
_  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _"  _
"" _ "" _ ""  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33
,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48  _"" _"" _"" _"" _"" _"" _"" _"" _"" _"" _""
 "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "
_  _"  _"  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _
"" _ "" _ ""
```

This is quite a lot of output, and yet with a low value of f (only 16), it isn't actually enough to simulate even this simple system 5 program through to the end, as will be seen in the next section.

System 4

Here is the interpreter ('system4.pl') I've been using for system 4:

```
#!/bin/perl -w

use strict;

my @elems;
my $active=0;  # if no active element is given to start with, start at left end
my $state='A'; # and in state A
my $nelems=0;
my $cpr=0;
my $c1b0=0;
my $i;

# Compressed output that shows which state we're in when hitting the leftmost
# star in state B or C
$ARGV[0]eq'C' and do{$cpr=1; shift @ARGV;};

# Merge together sets between stars in the output
$ARGV[0]eq'M' and do{$cpr=3; shift @ARGV;};

# Merge together sets between stars in the output, and run-length-encode it
$ARGV[0]eq'R' and do{$cpr=4; shift @ARGV;};

# Count 11s between 00s in the C-format output
$ARGV[0]eq'Y' and do{$cpr=5; shift @ARGV;};

# Read from file
$ARGV[0]eq'F' and do{
    shift @ARGV;
    my $file = shift @ARGV;
    local $/ = undef;
    open INFILE, $file;
    @ARGV = split(' ',<INFILE>);
    close INFILE;
};

$|=1;

for my $e (@ARGV)
{
    chomp $e;
    $e eq '"' and $e='';
    $e =~ s/ //g;
    # Three possibilities. It could be a state letter:
    if($e eq 'A' || $e eq 'B' || $e eq 'C')
    {
        $state=$e;
        $active=$nelems;
    }
    # or a star:
    elsif($e eq '_')
    {
```

```
        # Stars are unusual, so represent them with undef.
        $elems[$nelems++]=undef;
    }
    # but if it isn't, it's a set.
    else
    {
        # Sets are represented by a reference to a hash, like the system 5
        # representation.
        my %v;
        for my $k (split(',',$e))
        {
            defined($v{$k}) or $v{$k}=0;
            $v{$k}++;
        }
        $elems[$nelems++]=\%v;
    }
}
# While we aren't off the right end:
while($active<$nelems)
{
    # Default output.
    if(!$cpr)
    {
        $i=0;
        while($i<$nelems)
        {
            $i==$active and print $state, " ";
            defined($elems[$i])
                ? print +(join ',',sort {$a <=> $b} keys %{$elems[$i]})
                : print "_";
            defined($elems[$i]) and (%{$elems[$i]} or print '""');
            print " ";
            $i++;
        }
        print "\n";
    }
    # Merged output
    if($cpr==3&&!defined($elems[$active]))
    {
        my %a;
        $i=0;
        while($i<=$nelems)
        {
            if($i==$nelems||!defined($elems[$i]))
            {
                my @temp=sort {$a <=> $b} keys %a;
                while(@temp)
                {
                    my $l=shift @temp;
                    my $h=(shift @temp)-1;
                    $l==$h ? print "$l" : print "$l-$h";
                    @temp and print ",";
                };
                undef %a;
            }
            $i==$nelems and last;
            $i==$active and print $state;
            if(!defined($elems[$i]))
            {
                print "_";
            }
            else
            {
                for my $k (keys %{$elems[$i]})
                {
                    # Toggle the state of both k and k+1;
                    # so $a shows the boundaries between runs of set and unset,
                    # rather than just the set numbers.
                    defined($a{$k}) or $a{$k}=0;
                    $a{$k}++;
                    $a{$k}==2 and delete $a{$k};
                    defined($a{$k+1}) or $a{$k+1}=0;
                    $a{$k+1}++;
                    $a{$k+1}==2 and delete $a{$k+1};
                }
```

```
            }
            $i++;
        }
        print "\n";
    }
    # Run-length-encoded output
    if($cpr==4&&!defined($elems[$active]))
    {
        my (%a, $r_0, $r_);
        $i=0; $r_0=$r_=0; undef %a;
        while($i<=$nelems)
        {
            if($i==$nelems||!defined($elems[$i]))
            {
                if(defined($a{0}) && defined($a{1}) && scalar(keys(%a))==2)
                {
                    $r_>1 and ($r_>=5 ? print "(_x$r_)" : print "_" x $r_);
                    $r_==1 ? $r_0++ : print "0";
                    $r_=0;
                    undef %a;
                }
                elsif(scalar(keys(%a)))
                {
                    $r_0 and ($r_0>=3 ? print "(_0x$r_0)" : print "_0" x $r_0);
                    $r_ and ($r_>=5 ? print "(_x$r_)" : print "_" x $r_);
                    $r_0=$r_=0;
                    my @temp=sort {$a <=> $b} keys %a;
                    while(@temp)
                    {
                        my $l=shift @temp;
                        my $h=(shift @temp)-1;
                        $l==$h ? print "$l" : print "$l-$h";
                        @temp and print ",";
                    };
                    undef %a;
                }
            }
            if($i==$active||$i==$nelems)
            {
                $r_0 and ($r_0>=3 ? print "(_0x$r_0)" : print "_0" x $r_0);
                $r_ and ($r_>=5 ? print "(_x$r_)" : print "_" x $r_);
                $r_0=$r_=0;
                $i==$nelems and last;
                print $state;
            }
            if(!defined($elems[$i]))
            {
                $r_0 and $r_ and do{
                    $r_0>=3 ? print "(_0x$r_0)" : print "_0" x $r_0;
                    $r_0=0;
                };
                $r_++;
            }
            else
            {
                for my $k (keys %{$elems[$i]})
                {
                    # Toggle the state of both k and k+1;
                    # so $a shows the boundaries between runs of set and unset,
                    # rather than just the set numbers.
                    defined($a{$k}) or $a{$k}=0;
                    $a{$k}++;
                    $a{$k}==2 and delete $a{$k};
                    defined($a{$k+1}) or $a{$k+1}=0;
                    $a{$k+1}++;
                    $a{$k+1}==2 and delete $a{$k+1};
                }
            }
            $i++;
        }
        print "\n";
    }
    if($state eq 'A') # State A
    {
        # Star active (rule 2).
```

```
        if(!defined($elems[$active]))
        {
            # Remove the star, and activate the element to its right.
            splice @elems,$active,1;
            $nelems--;
            # Change to state B.
            $state='B';
        }
        else
        {
            # Go left if we can, or change to state B (rule 1).
            # The eval on the next line indicates that the assignment is
            # deliberate.
            $active ? $active-- : do{
                $state="B";
                $cpr==1 and $cpr=2;
                $cpr==5 and $cpr=6;
            }
        }
    }
    elsif(defined($elems[$active])) # rule 3: set active but not in state A
    {
        my %temp;
        # Decrement the entire rule
        %temp=();
        for my $e (keys %{$elems[$active]})
        {
            $e ? $temp{$e-1}=${$elems[$active]}{$e} # not a 0
               : ($state=($state eq 'B'?'C':'B')); # a 0 changes state
        };
        %{$elems[$active]}=%temp;
        # Move right
        $active++;
    }
    elsif($state eq 'B') # rule 4 (star in state B)
    {
        # Delete the star and move left
        splice @elems,$active--,1;
        $nelems--;
        # Change to state A
        $state='A';
        $cpr==2 and $cpr=1 and print "1";
        $cpr==6 and $c1b0>=2 and $c1b0++;
        $cpr==6 and $cpr=5;
    }
    else # rule 5 (star in state C)
    {
        # Move right
        $active++;
        # Toggle a 1 in this set
        !defined(${$elems[$active]}{1})
            ? (${$elems[$active]}{1}=1) # add a 1
                : delete ${$elems[$active]}{1}; # remove the 1
        $cpr==2 and $cpr=1 and print "0";
        if($cpr==6)
        {
            if($c1b0>=2)
            {
                $c1b0==2 or $c1b0==4 or print '(';
                print +($c1b0-2)/2;
                $c1b0==2 or $c1b0==4 or print ')';
                $c1b0=-1;
            }
            else
            {
                $c1b0==1 and $c1b0=2;
                $c1b0==0 and $c1b0=1;
                $c1b0==-1 and $c1b0=0;
            }
            $cpr=5;
        }
    }
}
print "\n";
```

This is a pretty long program, mostly because several output formats have to be supported to deal with the

incredible length of useful translated system 4 programs. The default format shows what happens at every step of execution:

```
$ system4.pl 0,6,8 "" _ B 3,4 _ 8 11,20
0,6,8 "" _ B 3,4 _ 8 11,20
0,6,8 "" _ 2,3 B _ 8 11,20
0,6,8 "" _ A 2,3 8 11,20
0,6,8 "" A _ 2,3 8 11,20
0,6,8 "" B 2,3 8 11,20
0,6,8 "" 1,2 B 8 11,20
0,6,8 "" 1,2 7 B 11,20
```

In most cases, though, this output is far too verbose (it's mostly only useful for seeing how system 4 behaves). There are four shorter formats:

```
$ system4 M `s52s4.pl 8 2 1,4 1,6`
1B_________0-12,14-18,20-24_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
0B________0-12,14-18,20-24_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
C_______0-12,14-18,20-24_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0C______0-12,14-18,20-24_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0_0C_____0-12,14-18,20-24_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0_0_0C____0-12,14-18,20-24_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0_0_0_0C___0-12,14-18,20-24_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0_0_0_0_0C__0-12,14-18,20-24_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0_0_0_0_0_0C_0-12,14-18,20-24_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0_0_0_0_0_0_1-11,13-17,19-23B_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0_0_0_0_0_0A_1-11,13-17,19-23_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0_0_0_0_0_1-10,12-16,18-22B_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0_0_0_0_0A_1-10,12-16,18-22_________________0-24_______________0-12,14-
22,24_____________________0-24_______________
_0_0_0_0_1-9,11-15,17-21B_________________0-24_______________0-12,14-22,24_____________________0-
24_________________
_0_0_0_0A_1-9,11-15,17-21_________________0-24_______________0-12,14-22,24_____________________0-
24_________________
_0_0_0_1-8,10-14,16-20B_________________0-24_______________0-12,14-22,24_____________________0-
24_________________
_0_0_0A_1-8,10-14,16-20_________________0-24_______________0-12,14-22,24_____________________0-
24_________________
_0_0_1-7,9-13,15-19B_________________0-24_______________0-12,14-22,24_____________________0-
24_________________
_0_0A_1-7,9-13,15-19_________________0-24_______________0-12,14-22,24_____________________0-
24_________________
_0_1-6,8-12,14-18B_________________0-24_______________0-12,14-22,24_____________________0-
24_________________
_0A_1-6,8-12,14-18_________________0-24_______________0-12,14-22,24_____________________0-
24_________________
_1-5,7-11,13-17B_________________0-24_______________0-12,14-22,24_____________________0-
24_________________
A_1-5,7-11,13-17_________0-24_______________0-12,14-22,24_____________________0-
24_________________
0-4,6-10,12-16B_________0-24_______________0-12,14-22,24_____________________0-
24_________________
0-3,5-9,11-15C_________0-24_______________0-12,14-22,24_____________________0-24_______________
0-3,5-9,11-15_0C_______0-24_______________0-12,14-22,24_____________________0-24_______________
0-3,5-9,11-15_0_0C_____0-24_______________0-12,14-22,24_____________________0-
24_________________
0-3,5-9,11-15_0_0_0C_____0-24_______________0-12,14-22,24_____________________0-
24_________________
0-3,5-9,11-15_0_0_0_0C_____0-24_______________0-12,14-22,24_____________________0-
24_________________
0-3,5-9,11-15_0_0_0_0_0C____0-24_______________0-12,14-22,24_____________________0-
24_________________
0-3,5-9,11-15_0_0_0_0_0_0C___0-24_______________0-12,14-22,24_____________________0-
24_________________
```

```
0-3,5-9,11-15_0_0_0_0_0_0_0C__0-24_____________0-12,14-22,24_____________0-
24
0-3,5-9,11-15_0_0_0_0_0_0_0_0C_0-24_____________0-12,14-22,24_____________0-
24_____________
0-3,5-9,11-15_0_0_0_0_0_0_0_0_1-23B_____________0-12,14-22,24_____________0-
24_____________
0-3,5-9,11-15_0_0_0_0_0_0_0_0A_1-23_____________0-12,14-22,24_____________0-
24_____________
0-3,5-9,11-15_0_0_0_0_0_0_0_1-22B_____________0-12,14-22,24_____________0-
24_____________
0-3,5-9,11-15_0_0_0_0_0_0_0A_1-22_____________0-12,14-22,24_____________0-
24_____________
0-3,5-9,11-15_0_0_0_0_0_0_1-21B_____________0-12,14-22,24_____________0-
24_____________
0-3,5-9,11-15_0_0_0_0_0_0A_1-21_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_0_0_0_0_0_1-20B_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_0_0_0_0_0A_1-20_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_0_0_0_0_1-19B_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_0_0_0_0A_1-19_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_0_0_0_1-18B_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_0_0_0A_1-18_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_0_0_1-17B_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_0_0A_1-17_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_0_1-16B_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_0A_1-16_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15_1-15B_____________0-12,14-22,24_____________0-24_____________
0-3,5-9,11-15A_1-15_____________0-12,14-22,24_____________0-24_____________
4,10,15B_____0-12,14-22,24_____________0-24_____________
3,9,14B_____0-12,14-22,24_____________0-24_____________
2,8,13B____0-12,14-22,24_____________0-24_____________
1,7,12B___0-12,14-22,24_____________0-24_____________
0,6,11B__0-12,14-22,24_____________0-24_____________
5,10C_0-12,14-22,24_____________0-24_____________
5,10_1-11,13-21,23B_____________0-24_____________
5,10A_1-11,13-21,23_____________0-24_____________
0-4,6-9,12-20,22B_____________0-24_____________
0-3,5-8,11-19,21C_____________0-24_____________
0-3,5-8,11-19,21_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_0_0C_____________0-24_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_0_0_0_1-23B_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_0_0_0_0A_1-23_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_0_0_0_1-22B_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_0_0_0A_1-22_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_0_0_1-21B_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_0_0A_1-21_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_0_1-20B_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_0A_1-20_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0_1-19B_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_0A_1-19_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0_1-18B_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_0A_1-18_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0_1-17B_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_0A_1-17_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0_1-16B_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_0A_1-16_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0_1-15B_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_0A_1-15_____________
0-3,5-8,11-19,21_0_0_0_0_0_0_1-14B_____________
0-3,5-8,11-19,21_0_0_0_0_0_0A_1-14_____________
0-3,5-8,11-19,21_0_0_0_0_0_1-13B_____________
0-3,5-8,11-19,21_0_0_0_0_0A_1-13_____________
0-3,5-8,11-19,21_0_0_0_0_1-12B_____________
0-3,5-8,11-19,21_0_0_0_0A_1-12_____________
```

```
0-3,5-8,11-19,21 0 0 1-11B
0-3,5-8,11-19,21_0_0A_1-11_
0-3,5-8,11-19,21_0_1-10B_
0-3,5-8,11-19,21_0A_1-10
```

As was explained in the proof above, consecutive sets act as one big merged set. The M format merges consecutive sets together using this method, and also saves horizontal space by removing all spaces from the output (they're unnecessary as after merging sets sets and stars must alternate), leaving off the "" for empty sets, and combining sequences of adjacent integers in sets into ranges. (It's also a good demonstration of the proof that the system 4 emulation of system 5 works.) It only shows the steps at which a star is active. The R format is a refinement of this, developed for displaying the behaviour of system 4 initial conditions that were generated to correspond with a system 5 initial condition:

```
$ system4 R `s52s4.pl 8 2 1,4 1,6`
1B(_x9)0-12,14-18,20-24(_x17)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0B(_x8)0-12,14-18,20-24(_x17)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
C(_x7)0-12,14-18,20-24(_x17)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
_0C(_x6)0-12,14-18,20-24(_x17)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
_0_0C(_x5)0-12,14-18,20-24(_x17)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x3)C____0-12,14-18,20-24(_x17)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x4)C___0-12,14-18,20-24(_x17)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x5)C__0-12,14-18,20-24(_x17)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x6)C_0-12,14-18,20-24(_x17)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x6)_1-11,13-17,19-23B(_x17)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x6)A_1-11,13-17,19-23(_x16)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x5)_1-10,12-16,18-22B(_x16)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x5)A_1-10,12-16,18-22(_x15)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x4)_1-9,11-15,17-21B(_x15)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x4)A_1-9,11-15,17-21(_x14)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x3)_1-8,10-14,16-20B(_x14)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
(_0x3)A_1-8,10-14,16-20(_x13)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
_0_0_1-7,9-13,15-19B(_x13)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
_0_0A_1-7,9-13,15-19(_x12)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
_0_1-6,8-12,14-18B(_x12)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
_0A_1-6,8-12,14-18(_x11)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
_1-5,7-11,13-17B(_x11)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
A_1-5,7-11,13-17(_x10)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-4,6-10,12-16B(_x10)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15C(_x9)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15_0C(_x8)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15_0_0C(_x7)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x3)C(_x6)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x4)C(_x5)0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x5)C____0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x6)C___0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x7)C__0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x8)C_0-24(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x8)_1-23B(_x15)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x8)A_1-23(_x14)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x7)_1-22B(_x14)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x7)A_1-22(_x13)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x6)_1-21B(_x13)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x6)A_1-21(_x12)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x5)_1-20B(_x12)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x5)A_1-20(_x11)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x4)_1-19B(_x11)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x4)A_1-19(_x10)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x3)_1-18B(_x10)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15(_0x3)A_1-18(_x9)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15_0_0_1-17B(_x9)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15_0_0A_1-17(_x8)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15_0_1-16B(_x8)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15_0A_1-16(_x7)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15_1-15B(_x7)0-12,14-22,24(_x17)0-24(_x14)
0-3,5-9,11-15A_1-15(_x6)0-12,14-22,24(_x17)0-24(_x14)
4,10,15B(_x6)0-12,14-22,24(_x17)0-24(_x14)
3,9,14B(_x5)0-12,14-22,24(_x17)0-24(_x14)
2,8,13B____0-12,14-22,24(_x17)0-24(_x14)
1,7,12B___0-12,14-22,24(_x17)0-24(_x14)
0,6,11B__0-12,14-22,24(_x17)0-24(_x14)
5,10C_0-12,14-22,24(_x17)0-24(_x14)
5,10_1-11,13-21,23B(_x17)0-24(_x14)
5,10A_1-11,13-21,23(_x16)0-24(_x14)
0-4,6-9,12-20,22B(_x16)0-24(_x14)
```

```
0-3,5-8,11-19,21C( x15)0-24( x14)
0-3,5-8,11-19,21_0C(_x14)0-24(_x14)
0-3,5-8,11-19,21_0_0C(_x13)0-24(_x14)
0-3,5-8,11-19,21(_0x3)C(_x12)0-24(_x14)
0-3,5-8,11-19,21(_0x4)C(_x11)0-24(_x14)
0-3,5-8,11-19,21(_0x5)C(_x10)0-24(_x14)
0-3,5-8,11-19,21(_0x6)C(_x9)0-24(_x14)
0-3,5-8,11-19,21(_0x7)C(_x8)0-24(_x14)
0-3,5-8,11-19,21(_0x8)C(_x7)0-24(_x14)
0-3,5-8,11-19,21(_0x9)C(_x6)0-24(_x14)
0-3,5-8,11-19,21(_0x10)C(_x5)0-24(_x14)
0-3,5-8,11-19,21(_0x11)C____0-24(_x14)
0-3,5-8,11-19,21(_0x12)C___0-24(_x14)
0-3,5-8,11-19,21(_0x13)C__0-24(_x14)
0-3,5-8,11-19,21(_0x14)C_0-24(_x14)
0-3,5-8,11-19,21(_0x14)_1-23B(_x14)
0-3,5-8,11-19,21(_0x14)A_1-23(_x13)
0-3,5-8,11-19,21(_0x13)_1-22B(_x13)
0-3,5-8,11-19,21(_0x13)A_1-22(_x12)
0-3,5-8,11-19,21(_0x12)_1-21B(_x12)
0-3,5-8,11-19,21(_0x12)A_1-21(_x11)
0-3,5-8,11-19,21(_0x11)_1-20B(_x11)
0-3,5-8,11-19,21(_0x11)A_1-20(_x10)
0-3,5-8,11-19,21(_0x10)_1-19B(_x10)
0-3,5-8,11-19,21(_0x10)A_1-19(_x9)
0-3,5-8,11-19,21(_0x9)_1-18B(_x9)
0-3,5-8,11-19,21(_0x9)A_1-18(_x8)
0-3,5-8,11-19,21(_0x8)_1-17B(_x8)
0-3,5-8,11-19,21(_0x8)A_1-17(_x7)
0-3,5-8,11-19,21(_0x7)_1-16B(_x7)
0-3,5-8,11-19,21(_0x7)A_1-16(_x6)
0-3,5-8,11-19,21(_0x6)_1-15B(_x6)
0-3,5-8,11-19,21(_0x6)A_1-15(_x5)
0-3,5-8,11-19,21(_0x5)_1-14B(_x5)
0-3,5-8,11-19,21(_0x5)A_1-14____
0-3,5-8,11-19,21(_0x4)_1-13B____
0-3,5-8,11-19,21(_0x4)A_1-13___
0-3,5-8,11-19,21(_0x3)_1-12B___
0-3,5-8,11-19,21(_0x3)A_1-12__
0-3,5-8,11-19,21_0_0_1-11B__
0-3,5-8,11-19,21_0_0A_1-11_
0-3,5-8,11-19,21_0_1-10B_
0-3,5-8,11-19,21_0A_1-10
```

The output is in the same format, but long repetitive sequences of _ "" _ "" _ "" ... and _ 0 _ 0 _ 0 ... (which tend to be produced as system 4 programs compiled from system 4 programs progress) have been shortened by using run-length encoding.

The other formats are designed for much larger programs, and to show the similarity to other systems executing the 'same' program.

```
$ cy2s5.pl 3 0,1 1 1,0 > 01110.s5
$ cat 01110.s5
1,2,3,4,5,7,8,10,11,13,14,16 21,24 19,22 "" "" 27,30,33,36,39,40 25,28,31,34,37,38 "" ""
43,46 41,44 "" ""
$ cy2s5.pl 3 0,1 1 1,0 > 01110.s5
$ s52s4.pl 100 F 01110.s5 > 01110.s4
$ system5.pl C F 01110.s5
00000000001100001100001100001100111...
$ system4.pl C F 01110.s4
000000000011000011000011000011001111111111111111111111111111111111111111111110
$ system5.pl C 2 1,4 1,6 "" ""
11001111001111001111001111...
$ system4.pl C `s52s4.pl 16 2 1,4 1,6 "" ""`
110011110011110010
```

The C output format prints a 0 whenever the leftmost star becomes active in state C, and a 1 whenever that star becomes active in state B. As the example above illustrates, this produces the same output as a system 5 system it's emulating. (However, the system 5 output reflects the fact that there are an infinite number of 1s in the output, but even using 100 as the value of f the system 4 output shows that after a while (determined by the value of f) the system 4 output becomes incorrect.) 01110.s4 is 50636 bytes long at this point, showing just how long even simple translated system 4 systems can be. The second example shows why a large value of f is needed (continuing on from the s52s4.pl example in the previous section).

Likewise, there's a Y format designed to reflect the output of the Y format in system5.pl:

```
$ cytag.pl Y F test1.cy
1111001111110011001100110011001100001100001111001100110000110000110011000000001100
$ cy2s5.pl 50 F test1.cy > test1.s5
$ system5.pl Y F test1.s5
1111001111110011001100110011001100001100001111001100110000110000110011000000001100
$ s52s4.pl 300 F test1.s5 > test1.s4
$ system4.pl Y F test1.s4
11110011111100110011001(0.5)Exiting due to signal SIGINT
$ s52s4.pl 1000 F test1.s5 > test1.s4
$ system4.pl Y F test1.s4
1111001111110011001100110011001100001100001111001100Exiting due to signal SIGINT
```

It counts the number of 11s between consecutive 00s in what the C format output would produce. Any number other than 0 and 1 is put in parentheses. (The (0.5) at the end shows that there was one 1, i.e. half a 11, between consecutive 00s; the text after that shows that I stopped the program at that point.) Even the high value of 300 for f wasn't enough to emulate the system 5 system that emulated this cyclic tag system through to the end, but it shows how the first few steps worked. It's not clear to me whether the last example would have been correct right to the end; I interrupted it after it had been running for 12_ hours overnight. (In the first 5 hours, it managed the first 40 characters of output approximately; in the next 7_, it only managed about 14, so it was unlikely to finish any time soon. test1.s4 was 9560058 byes long.)

Emulating system 4 with system 3, 2, 1, or 0

The four Turing-machine-like systems are sufficiently similar that it makes sense to have one program that compiles from system 4 into any of the four systems, and likewise one interpreter that can interpret any of the four. Because two-line programs can be somewhat tricky to input, the letter representing the state is written just before the active cell in the notation the programs use, rather than below it.

Here's the program (s42s0-3.pl):

```
#!/bin/perl -w

use strict;

my $system;
my $w;
my $two2thew;
my $i;

# Determine which system we're compiling into
chomp $ARGV[0];
$system=shift @ARGV;

# Read w and 2**w
chomp $ARGV[0];
$w=shift @ARGV;
$two2thew=2**$w;

# Read from file
$ARGV[0]eq'F' and do{
    shift @ARGV;
    my $file = shift @ARGV;
    local $/ = undef;
    open INFILE, $file;
    @ARGV = split(' ',<INFILE>);
    close INFILE;
};

$|=1;

# 1s and 2s are swapped to the left of the active cell in systems 0-2.
# To implement this, store them in variables and swap them once the active
# cell is reached.
my $two='1';
my $one='2';

$system==3 and ($two='2'),($one='1');

# An 'infinite number' (actually 2**w) of 0s, two 2s and any number of 1s
print '0' x $two2thew, "$two$two$one";
```

```
# Pre-generate the table of strings corresponding to 1-element sets.
# Bitwise string arithmetic is used here, with \0 corresponding to 1 and
# \1 corresponding to 2.
my @setstrings;
$#setstrings=$two2thew; #preextend for efficiency
$setstrings[0]="\1" . ("\0" x ($two2thew-1)); # first string
$i=0;
# and loop for the other strings
$setstrings[$i]=substr("\0" . $setstrings[$i-1], 0, -1) ^ $setstrings[$i-1]
    while $i++<$two2thew;

# And finally, it's actually possible to translate the program.
my $leftofactive=2;
my $justhadastar=0;
while(1)
{
    my $e = shift @ARGV;
    !defined($e) and last;
    chomp $e;
    $e eq '""' and $e='';
    $e =~ s/ //g;
    # Three possibilities. It could be a state letter:
    if($e eq 'A' || $e eq 'B' || $e eq 'C')
    {
        # If we're in state B or C, or A with a set active, the next character
        # to be printed will in fact be the active element. However, if it's a
        # star and we're in state C, we need to print an A not a C.
        chomp $ARGV[0];
        my %ABA=(A=>'A',B=>'B',C=>'A');
        my %ABC=(A=>'A',B=>'B',C=>'C');
        $ARGV[0]eq'_'? (print $ABA{$e}) :
            $system<2 ? (print $ABC{$e}) : print $e;
        # We're now at the active element; set leftofactive to 1 if we're in
        # state C in system 0 or 1 (because such systems have no state C,
        # and it needs to be simulated by state B), and to 0 if we're in state
        # B or system 3, and to 1 if we're in state A in system 0, 1, or 2.
        $leftofactive--;
        (($e eq 'C' and $system<2) or ($e eq 'A' and $system<3))
            or $leftofactive--;
        # A star active in state A replaces the rightmost end of the set to
        # its left; prevent it replacing the leftmost end of the set to its
        # right too by setting $justhadastar so that it's cleared by the
        # section below.
        $e eq 'A' and $ARGV[0] eq '_' and $justhadastar=1;
    }
    # or a star:
    elsif($e eq '_')
    {
        # Dealing with stars here is the easy bit. It's the special-casing for
        # them in the next section that's hard.
        print '0';
        $justhadastar=1-$justhadastar;
        $leftofactive==1 and $leftofactive=0;
    }
    # but if it isn't, it's a set.
    else
    {
        # Figure out the string
        my $v="\0" x $two2thew;
        for my $k (split(',',$e))
        {
            $k >= $two2thew and die 'w is set too low';
            $v ^= $setstrings[$k];
        }
        # We need a 2 at the end. We're always going to have a 1 at this point
        # assuming the final setstring hasn't been used (it shouldn't have
        # been if w is sufficiently high, so XOR in that.
        $v ^= $setstrings[$two2thew-1];
        # Fix the first element of the resulting string from a 1 to a 2 if
        # necessary.
        substr($v,0,1)eq"\0" and $v ^= $setstrings[$two2thew-2];
        # If $leftofactive is 1, negate all but the first element, for the
        # benefit of systems 0 and 1.
        $leftofactive == 1 and $v ^= "\0" . ("\1" x ($two2thew-1));
```

```
        # Transform from \0 and \1 into 1s and 2s.
        eval "\$v =~ y/\0\1/$one$two/";
        # If we just had a star and are right of active, leave off the first
        # element of the string as that's what it replaces. Likewise, if the
        # star was active in state B or C. (If it was active in state A,
        # $justhadastar has already been set to 0.)
        $justhadastar and $leftofactive!=2 and substr $v,0,1,'';
        # Likewise, if we're just about to have a star and are left of active,
        # leave off the last element, and the star will replace that.
        if(defined($ARGV[0]))
        {
            chomp $ARGV[0];
            $ARGV[0]eq'_' and $leftofactive==2 and substr $v,-1,1,'';
            # If we're about to have an /active/ star in state A or C, also
            # leave off the last element, and add a 0 for state C.
            defined($ARGV[1]) and chomp $ARGV[1];
            $ARGV[0]eq'A' and $ARGV[1]eq'_' and substr $v,-1,1,'';
            $ARGV[0]eq'C' and $ARGV[1]eq'_' and substr $v,-1,1,'0';
        }
        # And output the new run of elements.
        print $v;
        $justhadastar=0;
        $leftofactive==1 and $leftofactive=0;
    }
    # Once we're past the active element, everything's as normal for every
    # system.
    $leftofactive==0 and ($one='1',$two='2');
}
```

The first argument given is the system number (0, 1, 2, or 3) to translate the system 4 initial condition into; the second argument gives the desired value of w. (Don't put the value of 2^w here; I made that mistake a few times!) The remaining arguments are either the system 4 system to emulate, or an F and the filename of a file containing the system. For instance:

```
$ s42s0-3.pl 0 4 1,2 3 _ "" A _ ""
000000000000000011212211111111111111212212121212120111111111111111A02222222222222222
$ s42s0-3.pl 3 4 1,2 3 _ "" A _ ""
000000000000000022121122222222222221211212121212102222222222222222A02222222222222222
```

(The system 3 and system 0 output is identical, except that at and to the left of the active element 1s and 2s have been swapped; this relationship holds in general between the systems in state A, as explained in the proof above.)

Systems 3, 2, 1, and 0

One interpreter ('sys0-3.pl') serves to interpret each of the four Turing-machine-like systems. (The rules are given in a human-readable format in the 'Rules for the systems' code, in case experimenting with other systems is desired.)

```
#!/bin/perl -w

use strict;

my $cpr=0;
my $program;
my $state;
my $active;
my $system;
my %rules;
my @statehist=('A','A');
my $lastinitial0;

# Input system number
chomp $ARGV[0];
$system=shift @ARGV;

# Compressed output that shows which state we're in when hitting a 0 after
# hitting the rightmost 0 to the left of the program
$ARGV[0]eq'C' and do{$cpr=1; shift @ARGV;};

# Save vertical space by not indicating the active cell and state
$ARGV[0]eq'N' and do{$cpr=2; shift @ARGV;};
```

```
# Only show steps on which a 0 is active (in system 0, only show steps on which
# a 0 is active and the states on the previous 3 steps haven't been B then A
# then A)
$ARGV[0]eq'Z' and do{$cpr=3; shift @ARGV;};

# Output position and state when a 0 is active, preceding by a newline if it's
# the leftmost 0
$ARGV[0]eq'P' and do{$cpr=5; shift @ARGV;};

# Read from file
$ARGV[0]eq'F' and do{
    shift @ARGV;
    my $file = shift @ARGV;
    local $/ = undef;
    open INFILE, $file;
    @ARGV = split(' ',<INFILE>);
    close INFILE;
};

$|=1;

# Read the program (it's all one argument)
$program = shift @ARGV;

# Find the initially active cell
do {$active = index $program, "A"; $state="A"};
do {$active = index $program, "B"; $state="B"} if $active==-1;
do {$active = index $program, "C"; $state="C"} if $active==-1;

$active==-1 and die "Couldn't find the active cell in the initial condition.";

# Remove the state letter from the program
substr $program, $active, 1, '';

# Rules for the systems
$system==0 and %rules=('A0'=>'B1>','B0'=>'A2<',
                       'A1'=>'A2<','B1'=>'B2>',
                       'A2'=>'A1<','B2'=>'A0>');
$system==1 and %rules=('A0'=>'B1>','B0'=>'A2<',
                       'A1'=>'A2<','B1'=>'B2>',
                       'A2'=>'A1<','B20'=>'A00>','B21'=>'B12>','B22'=>'B11>');
$system==2 and %rules=('A0'=>'B1>','B0'=>'A2<','C0'=>'A2<',
                       'A1'=>'A2<','B1'=>'B2>',
                       'C10'=>'A00>','C11'=>'C11>','C12'=>'C12>',
                       'A2'=>'A1<','C2'=>'B2>',
                       'B20'=>'A00>','B21'=>'C11>','B22'=>'C12>');
$system==3 and %rules=('A0'=>'B2>','B0'=>'A2<','C0'=>'A2<',
                       'A1'=>'A1<','B1'=>'B1>',
                       'C10'=>'A00>','C11'=>'C21>','C12'=>'C22>',
                       'A2'=>'A2<','C2'=>'B1>',
                       'B20'=>'A00>','B21'=>'C21>','B22'=>'C22>');

# Find the last initial 0
$lastinitial0=0;
$lastinitial0++ while substr($program,$lastinitial0,1)eq'0';
$lastinitial0--;

while(1) # will break out when we go off the edge of the tape
{
    my ($temp, $i, $d);
    $cpr or print "$program\n" . (" " x $active) . $state . "\n";
    ($cpr==1 or $cpr==4) and $active<=$lastinitial0 and
        (($lastinitial0=$active-1), ($cpr=4));
    $cpr==2 and print "$program\n";
    $cpr==3 and substr($program, $active, 1)eq'0' and
        ($system||$state ne'A'||$statehist[0]ne'A'||$statehist[1]ne'B') and
            print "$program\n" . (" " x $active) . $state . "\n";
    $cpr==4 and substr($program, $active, 1)eq'0' and
        $active>$lastinitial0+1 and
            ($system||$state ne'A'||$statehist[0]ne'A'||$statehist[1]ne'B') and
                (($cpr=1),(print ($state eq 'A' ? 0 : 1)));
    $cpr==5 and substr($program, $active, 1)eq'0' and
        ($system||$state ne'A'||$statehist[0]ne'A'||$statehist[1]ne'B') and do{
            $active<=$lastinitial0 and (($lastinitial0=$active-1),print "\n");
            print "($active,$state)";
```

```
        };
    $i=0; $temp=undef;
    $temp=$rules{$state . substr($program, $active, $i)}
    while (!defined($temp)&&$i++<2);
    defined($temp) or last;
    $statehist[1]=$statehist[0]; $statehist[0]=$state;
    $state = substr $temp, 0, 1, '';
    $d = ('<' eq substr($temp, -1, 1, '') ? -1 : 1);
    substr $program, $active, $i, $temp;
    $active += $d;
}
print "\n";
```

By default, this shows what happens at every stage of the evolution of the system, until the active element goes off the end of the input portion of the tape:

```
$ sys0-3.pl 0 00A00000
0000000
  A
0010000
   B
0012000
  A
0022000
 A
0122000
  B
0102000
   A
0101000
  A
0111000
   B
0112000
    B
0112200
   A
0111200
  A
0121200
 A
0221200
A
1221200
 B
1021200
  A
1011200
 A
1111200
  B
1121200
   B
1122200
    B
1122000
     A
1122010
      B
1122012
     A
1122022
    A
1122122
     B
1122102
      A
1122101
     A
1122111
      B
1122112
       B
```

This format (the one used in the proof) can be quite long-winded, so there are various methods for shortening it. The N format simply doesn't print the lines representing the state and position of the active

element:

```
$ sys0-3.pl 0 N 00A00000
0000000
0010000
0012000
0022000
0122000
0102000
0101000
0111000
0112000
0112200
0111200
0121200
0221200
1221200
1021200
1011200
1111200
1121200
1122200
1122000
1122010
1122012
1122022
1122122
1122102
1122101
1122111
1122112
```

The other output formats have the concept of 'relevant zeros'; the system is at a relevant zero whenever a 0 is active in system 1, 2, or 3, or when a 0 is active in system 0 and it isn't the case that the system is in state A, was in state A on the previous step and state B on the step before that (this condition for system 0 corresponds to the simpler condition given for systems 1, 2, and 3). The Z format outputs the system at each relevant zero:

```
$ sys0-3.pl 0 Z `s42s0-3.pl 0 4 1,2 B 3 _ "" _ ""` > sys0-3.out
000000000000000011212211111111111111222112211221120222222222222222202222222222222222
                                                          B
000000000000000022121122222222222222211122112211221222222222222222202222222222222222
                 A
000000000000000112211121212121212121222212221222122121212121212121202222222222222222
                                                                      B
000000000000000221122212121212121212111121112111211212121212121212122222222222222222
                 A
```

The P output format simply gives the position of the active element (as a number giving the difference from the left end of the input tape) and the active state. It places a newline before these pairs if the active zero has nothing but zeros to its left. Here's an example translated all the way from system 5. (I've cut off long lines at 100 characters; such lines can get *very* long; the 'cat' line at the start is to show what the initial program was.)

```
$ cat 124524.s5
1,2 4,5 2,4 "" "" ""
$ s52s4.pl 32 F 124524.s5 > 124524.s4
$ s42s0-3.pl 3 7 F 124524.s4 > 124524.s3
$ sys0-3.pl 3 P F 124524.s3 > sys0-3.out

(127,A)(259,A)(387,A)(515,A)(643,A)(771,A)(899,A)(1027,A)(1155,A)(1283,A)(1411,A)(1539,A)(1
667,A)(17...
(126,A)(8835,A)(8963,A)(9091,A)(9219,A)(9347,A)(9475,A)(9603,A)(9731,A)(9859,A)(9987,A)(101
15,A)(102...
(125,A)(16899,A)(17027,A)(17155,A)(17283,A)(17411,A)(17539,A)(17667,A)(17795,A)(17923,A)(18
051,A)(18...
(124,A)(24963,A)(25091,A)(25219,A)(25347,A)(25475,A)(25603,A)(25731,A)(25859,A)(25987,A)(26
115,A)(26...
(123,A)(33539,B)
(122,A)(33667,B)
(121,A)(33795,B)
(120,A)(33923,B)
(119,A)(34051,B)
(118,A)(34179,B)
```

```
(117,A)(34307,B)
(116,A)(34435,B)
(115,A)(34563,A)(34691,A)(34819,A)(34947,A)(35075,A)(35203,A)(35331,A)(35459,A)(35587,A)(35
715,A)(35...
(114,A)(40067,A)(40195,A)(40323,A)(40451,A)(40579,A)(40707,A)(40835,A)(40963,A)(41091,A)(41
219,A)(41...
(113,A)(51203,A)(51331,A)(51459,A)(51587,A)(51715,A)(51843,A)(51971,A)(52099,A)(52227,A)(52
355,A)(52...
(112,A)(56195,A)(56323,A)(56451,A)(56579,A)(56707,A)(56835,A)(56963,A)(57091,A)(57219,A)(57
347,A)(57...
(111,A)(67843,B)
(110,A)(67971,B)
(109,A)(68099,B)
(108,A)(68227,B)
(107,A)(68355,B)
(106,A)(68483,B)
(105,A)(68611,B)
(104,A)(68739,B)
(103,A)(68867,B)
(102,A)(68995,B)
(101,A)(69123,B)
(100,A)(69251,B)
(99,A)(69379,B)
(98,A)(69507,B)
(97,A)(69635,B)
(96,A)(69763,B)
(95,A)(69891,B)
(94,A)(70019,A)(70147,A)(70275,A)(70403,A)(70531,A)(70659,A)(70787,A)(70915,A)(71043,A)(711
71,A)(712...
$ system4.pl C F 124524.s4
00001111111100001111111111111111110
```

Compare this output to the last line shown (for the corresponding system 4 system); the 0s correspond exactly to 'long' lines (which have several entries and in which the second relevant zero was in state A), and the 1s correspond exactly to 'short' lines (with two relevant zeros, in state A then state B). The C output format makes this explicit:

```
$ cat 124524.s5
1,2 4,5 2,4 "" "" ""
$ s52s4.pl 32 F 124524.s5 > 124524.s4
$ s42s0-3.pl 3 7 F 124524.s4 > 124524.s3
$ s42s0-3.pl 2 7 F 124524.s4 > 124524.s2
$ s42s0-3.pl 1 7 F 124524.s4 > 124524.s1
$ s42s0-3.pl 0 7 F 124524.s4 > 124524.s0
$ system5.pl C F 124524.s5
0000111111110000111...
$ system4.pl C F 124524.s4
00001111111100001111111111111111110
$ sys0-3.pl 3 C F 124524.s3
00001111111100001111111111111111110
$ sys0-3.pl 2 C F 124524.s2
00001111111100001111111111111111110
$ sys0-3.pl 1 C F 124524.s1
00001111111100001111111111111111110
$ sys0-3.pl 0 C F 124524.s0
00001111111100001111111111111111110
```

This is using the same system 5 example as the previous example. The ... at the end of the output of system5.pl shows that the 1s continue indefinitely in that system. The system 4 version emulates this correctly for a while (eventually producing a 0 by mistake; increasing the value of f from 32 would increase the number of 1s at the end before a final malfunction). The system 3, 2, 1, and 0 versions all correctly emulate the system 4 version. (The C output format looks at each relevant zero after the active element goes further left than ever before; if it's in state A, it prints a 0, and if it isn't, it prints a 1; this algorithm is the same for all four systems.)

Appendix: Mathematica Code

```
MyTimeConstraint = 10000;

(*Code for evaluating cyclic tag systems, from the notes of A New Kind of Science. (A cyclic tag system is represented as a list of strings to be added (themselves represented as lists), followed by the initial value of the working string, bundled together in a list.)*)

CTStep[{{r_, s___}, {0, a___}}] := {{s, r}, {a}}

CTStep[{{r_, s___}, {1, a___}}] := {{s, r}, Join[{a}, r]}

CTStep[{u_, {}}] := {u, {}}

CTEvolveList[rules_, init_, t_] := Last /@ NestList[CTStep, {rules, init}, t]

ConcatBits[list_] := StringJoin[(ToString[#1] &) /@ list]

(*Output to match that of the Perl cyclic-tag program.*)

CTCompress[hist_] :=
 ConcatBits[(#1[[1]] &) /@ Select[hist, MatchQ[#1, {__}] &]]

CTCompressY[hist_] :=
 ConcatBits[
  Flatten[(#1 /.x_ -> {x, x} &) /@ (#1[[1]] &) /@
    Select[hist, MatchQ[#1, {__}] &]]]

(*Converting a cyclic-tag system to system 5. This is quite simple, but long, because the constraints of what values the integers corresponding to various parts of the cyclic tag system can take as compared to other integers generated have to be enforced; this is done using i, which is stored as a parameter to start with and later using iKnown[] and iPlaceholder[], and holds the lowest legal value a newly-added integer can take. (A system 5 initial condition is represented as a list containing the bag followed by each rule in order; the bag and rules themselves are represented as lists.)*)

CTToSystem5[rules_, init_, steps_] :=
 CTToSystem5Main[CTInitToSystem5Init[{}, init, 1], rules, steps]

CTInitToSystem5Init[sofar_, {0, reminit___}, i_] :=
 CTInitToSystem5Init[Flatten[{sofar, i, i + 1, i + 2, i + 3}], {reminit},
  i + 4]

CTInitToSystem5Init[sofar_, {1, reminit___}, i_] :=
 CTInitToSystem5Init[Flatten[{sofar, i, i + 2, i + 3, i + 5}], {reminit},
  i + 6]

CTToSystem5Main[CTInitToSystem5Init[sofar_, {}, i_], rules_, steps_] :=
 Flatten[{{sofar}, CTRulesToSystem5Rules[{}, rules, steps, iKnown[i + 2]]}, 1]

CTRulesToSystem5Rules[sofar_, _, 0, iKnown[_Integer]] := sofar

CTRulesToSystem5Rules[sofar_, {firstrule_, otherrules___}, steps_,
  iKnown[i_Integer]] :=
 CTRulesToSystem5Rules[
  Flatten[{sofar, CTRuleToSystem5Rule[{}, firstrule, i]}, 1], {otherrules,
   firstrule}, steps - 1, iPlaceholder[]]

CTRulesToSystem5Rules[{sofar___, iPlaceholder[i_]}, rules_, steps_,
  iPlaceholder[]] := CTRulesToSystem5Rules[{sofar}, rules, steps, iKnown[i]]

CTRuleToSystem5Rule[sofar_, {0, remrule___}, i_] :=
 CTRuleToSystem5Rule[Flatten[{sofar, i, i + 1}], {remrule}, i + 4]

CTRuleToSystem5Rule[sofar_, {1, remrule___}, i_] :=
 CTRuleToSystem5Rule[Flatten[{sofar, i, i + 3}], {remrule}, i + 6]

CTRuleToSystem5Rule[sofar_, {}, i_] := {(#1 + 2 &) /@ sofar, sofar, {}, {},   iPlaceholder[i]}

(*Emulating system 5 is a lot easier than constructing a system 5 program:*)

RemoveEvenDuplicates[l_List] :=
 First /@ Select[Tally[l], MatchQ[#1, {_, _?OddQ}] &]

System5Step[initial_List] :=
```

```
 Join[{Sort[(#1 - 1 &) /@
     RemoveEvenDuplicates[initial[[1]]]]}, ((#1 + 1 &) /@ (#1 &)) /@
   Drop[initial, 1]] /.{{0, x___}, {y___}, z___} -> {{x, y}, z}

(*MemoryEfficientCompress[step_,compress_,init_,steps_] and MemoryEfficientCompressNoJoin[step_,compress_,init_,steps_]
are both capable of being equivalent to compress[NestList[step,init,steps]] in some cases used below (which, if either, is correct
depends on the details of the compress function used, and sometimes neither can be used). They both also show
ProgressIndicators when doing long computations.*)

MemoryEfficientCompress[step_, compress_, init_, steps_] :=
 Monitor[{sc = 0,
   StringJoin[
    Reap[Nest[{step[#1], Sow[compress[{#1}]], sc = sc + 1}[[1]] &, init,
      steps]][[2]]]}[[2]], {ProgressIndicator[sc, {1, steps}], sc}]

SowIfNotNull[{}] := 0

SowIfNotNull[{l__}] := Sow[{l}]

MemoryEfficientCompressNoJoin[step_, compress_, init_,
  steps_] := {sc = 0,
   Monitor[Flatten[
     Reap[Nest[{step[#1], sc = sc + 1, SowIfNotNull[compress[{#1}]]}[[1]] &,
       init, steps]][[2]], 2], {ProgressIndicator[sc, {1, steps}], sc}][[2]]

(*Matching the output of the Perl cyclic-tag program.*)

System5CompressC[hist_List] :=
 ConcatBits[
  Flatten[(If[MatchQ[#1, {{1, ___}, __}], {0, 0}, {1, 1}] &) /@ hist]]

System5CompressY[hist_List] :=
 ConcatBits[(#1[[2]] &) /@
   Partition[(1/2 (Length[#1] - 1) &) /@
     Split[Flatten[(If[MatchQ[#1, {{1, ___}, __}], {0, 1}, {1, 1}] &) /@  hist]], 4]]

(*Compiling system 5 to system 4. (A system 4 initial condition is represented as a list; each element of that list is either another list
(a set in system 4), S (a star) or ActiveState[A], ActiveState[B], or ActiveState[C] (which indicate that the next element is active in
that state).)*)

System5ToSystem4[initial_List, f_Integer] :=
 Flatten[{ActiveState[
    A], {{{{RemoveEvenDuplicates[(2 #1 - 2 &) /@   initial[[1]]]}}}},
{{Table[{S, {}}, {f}]}}, ({S, {{RemoveEvenDuplicates[Join[Range[0, f 3], (2 #1 + f + 3 &) /@ #1]]}},
      Table[{S, {}}, {2 f}], S, {{RemoveEvenDuplicates[Range[0, 3 f]]}},
      Table[{S, {}}, {2 f - 2}]} &) /@ Drop[initial, 1]}, 4]

(*Emulating system 4. System 4 has several rules that don't follow an obvious pattern; each of these rules is therefore given a
separate definition below (or sometimes more than one definition).*)

ContainsA0[l_List] := MatchQ[l, {___, 0, ___}]

ListDecrement[l_List] := (#1 - 1 &) /@ Select[l, #1 > 0 &]

(*Rule 1*)

System4Step[{before___, elem_, ActiveState[A], {list___},
  after___}] := {before, ActiveState[A], elem, {list}, after}

System4Step[{ActiveState[A], {list___}, after___}] := {ActiveState[B], {list},
  after}

(*Rule 2*)

System4Step[{before___, ActiveState[A], S, after___}] := {before,
 ActiveState[B], after}

(*Rule 3*)

System4Step[{before___, ActiveState[B], {list___}, after___}] :=
 If[ContainsA0[{list}], {before, ListDecrement[{list}], ActiveState[C],
  after}, {before, ListDecrement[{list}], ActiveState[B], after}]

System4Step[{before___, ActiveState[C], {list___}, after___}] :=
 If[ContainsA0[{list}], {before, ListDecrement[{list}], ActiveState[B],
```

```
  after}, {before, ListDecrement[{list}], ActiveState[C], after}]

(*Rule 4*)

System4Step[{before___, elem_, ActiveState[B], S, after___}] := {before,
  ActiveState[A], elem, after}

(*Rule 5*)

System4Step[{before___, ActiveState[C], S, {list___}, after___}] := {before,
  S, ActiveState[C], RemoveEvenDuplicates[Join[{list}, {1}]], after}

(*The length of useful system 4 programs means that far too much output (either to read or to fit into memory) is produced when
trying to run a useful system 4 program using NestList. Here are the definitions used to reproduce or approximate the M, R, C, and
Y output formats of the Perl program. (Most of the compression functions take a history from NestList as their argument, although
the Y format takes the C format's output as its argument; as this would be too large to fit into my computer's memory,
MemoryEfficientCompressList or the NoJoin version have to be used in practice when compressing the output of a large system 4
program.)*)

ListAsRangesHelper[c_, i_] := If[c > 0, i - 2, If[c < 0, i - 1, ""]]

ListAsRangesHelper2[{x_, y_}] :=
 If[x != y, ToString[x] <> "-" <> ToString[y], ToString[x]]

ListAsRanges[l_List] :=
 StringJoin[
  Riffle[ListAsRangesHelper2 /@
    Partition[
     Select[Table[
       ListAsRangesHelper[
        ListConvolve[{-1, 1}, (If[MatchQ[l, {___, #1, ___}], 1, 0] &) /@
           Range[-1, Max[l] + 1]][[i]], i], {i, Max[l] + 2}], !
        MatchQ[#1, ""] &], 2], ","]] /; MatchQ[l, {__}]

ListAsRanges[{}] := ""

System4Combine[{before___, {list1___}, {list2___}, after___}] :=
 System4Combine[{before, RemoveEvenDuplicates[Join[{list1}, {list2}]], after}]

System4Combine[l_List] := l /; ! MatchQ[l, {___, {___}, {___}, ___}]

System4CombineMore[{before___, S, {}, S, {}, after___}] :=
 System4CombineMore[{before, RunLength[{S, {}}, 2], after}]

System4CombineMore[{before___, S, {0}, S, {0}, after___}] :=
 System4CombineMore[{before, RunLength[{S, {0}}, 2], after}]

System4CombineMore[{before___, S, x_, RunLength[{S, x_}, c_], after___}] :=
 System4CombineMore[{before, RunLength[{S, x}, c + 1], after}]

System4CombineMore[{before___, RunLength[{S, x_}, c_], S, x_, after___}] :=
 System4CombineMore[{before, RunLength[{S, x}, c + 1], after}]

System4CombineMore[{before___, RunLength[x_, c1_], RunLength[x_, c2_],
  after___}] := System4CombineMore[{before, RunLength[x, c1 + c2], after}]

System4CombineMore[l_List] :=
 l /; ! MatchQ[ l, {___, S, {}, S, {}, ___} | {___, S, {0}, S, {0}, ___} | {___, S, x_,
     RunLength[{S, x_}, _], ___} | {___, RunLength[{S, x_}, _], S,
     x_, ___} | {___, RunLength[x_, _], RunLength[x_, _], ___}]

System4CompressMHelper[S] := "_"

System4CompressMHelper[{l___}] := ListAsRanges[{l}]

System4CompressMHelper[ActiveState[s_]] := ToString[s]

System4CompressMHelper[RunLength[{S, {}}, c_]] := "(_x" <> ToString[c] <> ")"

System4CompressMHelper[RunLength[{S, {0}}, c_]] :=
 "(_0x" <> ToString[c] <> ")"

System4CompressM[
  hist_] := (StringJoin[System4CompressMHelper /@ System4Combine[#1]] &) /@
  Select[hist, MatchQ[#1, {___, ActiveState[_], S, ___}] &]
```

```
System4CompressR[
  hist_] := (StringJoin[
     System4CompressMHelper /@ System4CombineMore[System4Combine[#1]]] &) /@
  Select[hist, MatchQ[#1, {___, ActiveState[_], S, ___}] &]

System4CompressC[hist_] :=
 ConcatBits[(System4Combine[#1] /.{{_?Positive ...} ..,
         ActiveState[B], ___} -> 1 /.{___, ActiveState[C], ___} ->
       0 /.{___} -> "" &) /@    Select[hist, MatchQ[#1, {{___} .., ActiveState[B | C], S, ___}] &]]

System4CompressY[compressC_] :=
 StringJoin[(StringLength[#1]/2 &) /@ StringSplit[compressC, "00"]]

(*The cellular automaton that simplifies the construction of a system 3 initial condition from a system 4 initial condition, written
here as a recursive function using ListConvolve.*)

System3Set[0, length_] := Join[{2}, Table[1, {length - 1}]]

System3Set[row_?Positive, length_] := (Mod[#1, 2] + 1 &) /@  ListConvolve[{1, 1}, Join[{1}, System3Set[row - 1, length]]]

(*Converting system 4 to systems 3, 2, and 0. (The system 1 program corresponding to a system 4 program is the same as the
system 0 program corresponding to that system 4 program.) The system 3, 2, 1, and 0 initial conditions are written as lists, with
ActiveState[A] (or B or C) immediately before the active element. This is implemented by marking all the elements in the system 4
program as LeftOfActive or RightOfActive, translating all the system 4 sets into lists of system 3 elements and all the system 4 stars
into placeholders including ReplaceLeft or ReplaceRight (indicating that the neighbouring element should be replaced by a 0, or
possibly by an active 0), doing those replacements, and then converting the resulting system 3 program into a system 2 or system 0
form if desired.*)

System4ToSystem3Helper[{before___, ActiveState[s_], elem_, after___}] :=
 Flatten[{(LeftOfActive[#1] &) /@ {before},
   ActiveState[s, elem], (RightOfActive[#1] &) /@ {after}}]

StartsAndEndsWith2[{2, l___, 2}] := {2, l, 2}

StartsAndEndsWith2[{1, l___}] := StartsAndEndsWith2[(3 - #1 &) /@ {1, l}]

StartsAndEndsWith2[{2, l___, 1}] :=
 StartsAndEndsWith2[ Flatten[(#1 /.{a_, b_} -> {a, 3 - b} &) /@ Partition[{2, l, 1}, 2]]]

System4ElemToSystem3[(LeftOfActive | RightOfActive)[{set___}], w_] :=
 StartsAndEndsWith2[(Mod[Total[(#1 - 1 &) /@ #1], 2] + 1 &) /@
   Transpose[Join[{Table[1, {2^w}]}, (System3Set[#1, 2^w] &) /@ {set}]]]

System4ElemToSystem3[LeftOfActive[S], _] := ReplaceLeft[0]

System4ElemToSystem3[RightOfActive[S], _] := ReplaceRight[0]

System4ElemToSystem3[ActiveState[s_, {set___}], w_] := {ActiveState[s], System4ElemToSystem3[LeftOfActive[{set}], w]}

System4ElemToSystem3[ActiveState[A, S], _] := ReplaceLeft[{ActiveState[A], 0}]

System4ElemToSystem3[ActiveState[B, S], _] := {ActiveState[B],   ReplaceRight[0]}

System4ElemToSystem3[ActiveState[C, S], _] := {ReplaceLeft[0], ActiveState[A],   ReplaceRight[0]}

DoSystem4Replacements[{before___, elem_, ReplaceLeft[i_], after___}] :=
 DoSystem4Replacements[{before, i, after}]

DoSystem4Replacements[{before___, ReplaceRight[i_], elem_, after___}] :=
 DoSystem4Replacements[{before, i, after}]

DoSystem4Replacements[l_List] :=
 l /; ! MatchQ[ l, {___, _, ReplaceLeft[_], ___} | {___, ReplaceRight[_], _, ___}]

System4ToSystem3[init_, w_] :=
 Flatten[DoSystem4Replacements[
   Flatten[{Table[0, {2^w}], 2, 2, 1, (System4ElemToSystem3[#1, w] &) /@ System4ToSystem3Helper[init]}]]]

System3ToSystem2[{before___, ActiveState[A], elem_, after___}] :=
 Flatten[{(Mod[3 - #1, 3] &) /@ {before}, ActiveState[A], Mod[3 - elem, 3],    after}]

System3ToSystem2[{before___, ActiveState[B], after___}] :=
 Flatten[{(Mod[3 - #1, 3] &) /@ {before}, ActiveState[B], after}]
```

```
System2ToSystem0[{before___, ActiveState[C], 2, after___}] := {before,   ActiveState[B], 1, after}

System2ToSystem0[{before___, ActiveState[C], 1, after___}] := {before,   ActiveState[B], 2, after}

System2ToSystem0[{before___, ActiveState[C], 0, after___}] := {before,  ActiveState[B], 0, after}

System2ToSystem0[{before___, ActiveState[B], after___}] := {before,   ActiveState[B], after}

System2ToSystem0[{before___, ActiveState[A], after___}] := {before,   ActiveState[A], after}

System4ToSystem2[l_, w_] := System3ToSystem2[System4ToSystem3[l, w]]

System4ToSystem0[l_, w_] := System2ToSystem0[System4ToSystem2[l, w]]

(*Executing a system 3, 2, 1, or 0 program. A general SystemStep is used, which is given a function as input that specifies how to
change the active cell and the cells before and after.*)

SystemStep[rules_, {before___, elem1_, ActiveState[s_], elem2_, elem3_, after___}] :=
 Flatten[{before, rules[elem1, s, elem2, elem3], after}]

System0Rules[a_, A, 0, b_] := {a, 1, ActiveState[B], b}

System0Rules[a_, B, 0, b_] := {ActiveState[A], a, 2, b}

System0Rules[a_, A, 1, b_] := {ActiveState[A], a, 2, b}

System0Rules[a_, B, 1, b_] := {a, 2, ActiveState[B], b}

System0Rules[a_, A, 2, b_] := {ActiveState[A], a, 1, b}

System0Rules[a_, B, 2, b_] := {a, 0, ActiveState[A], b}

System1Rules[a_, A, 0, b_] := {a, 1, ActiveState[B], b}

System1Rules[a_, B, 0, b_] := {ActiveState[A], a, 2, b}

System1Rules[a_, A, 1, b_] := {ActiveState[A], a, 2, b}

System1Rules[a_, B, 1, b_] := {a, 2, ActiveState[B], b}

System1Rules[a_, A, 2, b_] := {ActiveState[A], a, 1, b}

System1Rules[a_, B, 2, 0] := {a, 0, ActiveState[A], 0}

System1Rules[a_, B, 2, 1] := {a, 1, ActiveState[B], 2}

System1Rules[a_, B, 2, 2] := {a, 1, ActiveState[B], 1}

System2Rules[a_, A, 0, b_] := {a, 1, ActiveState[B], b}

System2Rules[a_, B, 0, b_] := {ActiveState[A], a, 2, b}

System2Rules[a_, C, 0, b_] := {ActiveState[A], a, 2, b}

System2Rules[a_, A, 1, b_] := {ActiveState[A], a, 2, b}

System2Rules[a_, B, 1, b_] := {a, 2, ActiveState[B], b}

System2Rules[a_, C, 1, 0] := {a, 0, ActiveState[A], 0}

System2Rules[a_, C, 1, 1] := {a, 1, ActiveState[C], 1}

System2Rules[a_, C, 1, 2] := {a, 1, ActiveState[C], 2}

System2Rules[a_, A, 2, b_] := {ActiveState[A], a, 1, b}

System2Rules[a_, B, 2, 0] := {a, 0, ActiveState[A], 0}

System2Rules[a_, B, 2, 1] := {a, 1, ActiveState[C], 1}

System2Rules[a_, B, 2, 2] := {a, 1, ActiveState[C], 2}

System2Rules[a_, C, 2, b_] := {a, 2, ActiveState[B], b}

System3Rules[a_, A, 0, b_] := {a, 2, ActiveState[B], b}
```

```
System3Rules[a_, B, 0, b_] := {ActiveState[A], a, 2, b}

System3Rules[a_, C, 0, b_] := {ActiveState[A], a, 2, b}

System3Rules[a_, A, 1, b_] := {ActiveState[A], a, 1, b}

System3Rules[a_, B, 1, b_] := {a, 1, ActiveState[B], b}

System3Rules[a_, C, 1, 0] := {a, 0, ActiveState[A], 0}

System3Rules[a_, C, 1, 1] := {a, 2, ActiveState[C], 1}

System3Rules[a_, C, 1, 2] := {a, 2, ActiveState[C], 2}

System3Rules[a_, A, 2, b_] := {ActiveState[A], a, 2, b}

System3Rules[a_, B, 2, 0] := {a, 0, ActiveState[A], 0}

System3Rules[a_, B, 2, 1] := {a, 2, ActiveState[C], 1}

System3Rules[a_, B, 2, 2] := {a, 2, ActiveState[C], 2}

System3Rules[a_, C, 2, b_] := {a, 1, ActiveState[B], b}

(*Emulating or approximating the output formats of the Perl programs. (The active state is shown before the active element, rather
than on the next line; the P output format is somewhat different but still gives much the same visual appearance, due to the
embedding of literal newlines in the output list.)*)

System0To3CompressN[_, hist_] := (ConcatBits[Select[#1, IntegerQ]] &) /@ hist

SelectRelevantZeros[_?Positive, hist_] :=
 Select[hist, MatchQ[#1, {___, ActiveState[_], 0, ___}] &]

SelectRelevantZeros[0, hist_] := (#1[[3]] &) /@
  Select[Partition[hist, 3, 1],
   MatchQ[#1[[3]], {___, ActiveState[_], 0, ___}] &&
            !MatchQ[#1, {{___, ActiveState[B], ___}, {___,
       ActiveState[A], ___}, {___, ActiveState[A], ___}}] &]

CellOrStateToString[i_Integer] := ToString[i]

CellOrStateToString[ActiveState[s_]] := ToString[s]

System0To3CompressZ[s_, hist_] := (StringJoin[CellOrStateToString /@ #1] &) /@
   SelectRelevantZeros[s, hist]

System0To3CompressPHelper[{before___, a_?Positive, morebefore___,
   ActiveState[s_], ___}] := {{Length[{before, a, morebefore}], s}}

System0To3CompressPHelper[{before___,
   ActiveState[s_], ___}] := {"\n", {Length[{before}], s}} /;
  MatchQ[{before}, {(0) ...}]

System0To3CompressP[s_, hist_] :=
 Flatten[System0To3CompressPHelper /@ SelectRelevantZeros[s, hist], 1]

System0To3CompressC[pcompressed_] :=
 ConcatBits[(If[Length[#1] == 3, 1, 0] &) /@ Split[pcompressed, #2 =!= "\n" &]]
```

Epilogue

How I constructed this proof

Upon seeing the problem, the first step was to figure out what the Turing machine was actually doing. Checking the typical behaviour of the Turing machine led me to come up with system 2 as describing its behaviour more clearly than the system 0 description; I worked with system 2 as the base Turing-machine-like system throughout most of the construction of the proof. (I discovered system 3 later when I started writing the proof up rigorously; it makes the proof considerably simpler than trying to prove things directly from system 2 because the tape is not modified when 'going left', as it were. I thought of system 1 just before writing up the proof rigorously, as an intermediate step to showing the equivalence of systems 0 and 2.) I then explored possible assumptions that would make the proof possible if true. The first assumption was that a string of 1s and 2s could be constructed to simulate any finite state machine whose input was the state in which it was encountered and whose output was its parity; this assumption makes the proof almost trivial, but as lemma 1 shows is not true. The second assumption I explored was what is given above as Corollary 1; this did turn out to be true, and suggested trying to prove the emulation for an arbitrary rather than infinite number of steps. Around this time, I had a vague idea of what system 4 was like floating around my head, but nothing formal in this regard. Something similar to system 5 (although the exact details changed) is what I decided would be 'the system to aim for', after thinking about how to exploit the behaviour of a 0 in system 2 to do what in the terminology I use above I'd describe as merging system 4 sets into the leftmost conglomeration at the right time; I quickly decided that system 5 was universal, and in fact conjecture 5 was the first part of the above I proved, which spurred on my attempts to prove or disprove Corollary 1.

After that, it was a case of trying to link the parts of the proof together. I developed a not-quite-rigorous proof of Lemma 1 (and therefore its corollary), then expressed in terms of system 2, and made an attempt at emulating system 5 directly with system 2, but this last step was too complicated to do all in one go. (I had a system 5 to system 0 compiler at that point, some of which was constructed by tinkering with the values until they worked, so I was pretty sure the emulation was possible; I just had to prove my program worked. I didn't have any explicit cyclic tag to system 5 compiler until after I'd written the proof, though, just thought experiments.) At that point, I started writing up this document, discovering systems 1 and 3, and started running experiments and thinking of proofs to work out the exact details of system 4. After the proof had been written (although I made a few corrections and tweaks to it later), I worked out the programs in this document to be more-or-less direct translations of the constructions and definitions given in the proof itself, both to find errors in the proof and to give demonstrations of its ideas. I used the programs and proofs as examples to test each other (so it's possible that bugs remain in the programs, but I suspect if there are bugs they'll turn out not to happen in the cases that can come from constructions in the proof).