STEPHEN WOLFRAM WRITINGS

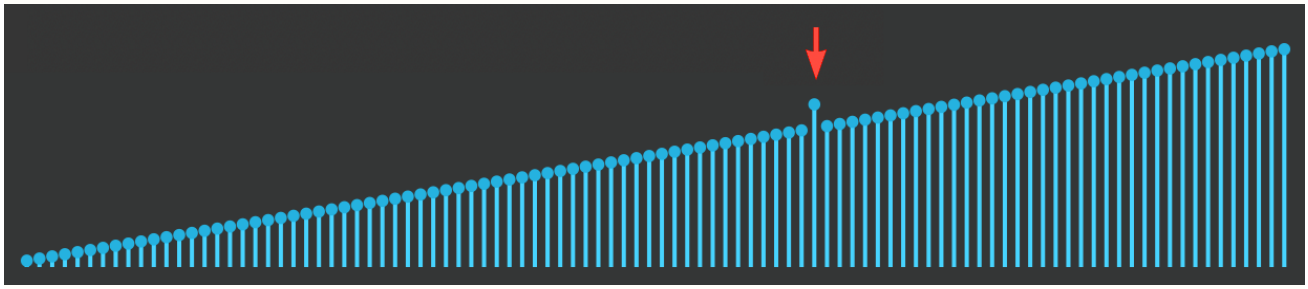
迈向 Bug 理论：意外现象的规则学

Towards a Theory of Bugs: The Ruliology of the Unexpected

Stephen Wolfram

2026 年 7 月 21 日

目录



“我的程序做错了！”

> Bug 是软件世界中普遍存在的现象。而且——本质上根据定义——它们中的每一个在某种程度上都是独一无二的和意想不到的。但是——特别是考虑到它们无处不在——人们可能想知道是否可能存在某种关于它们的普遍“科学”理论。我的目标是探索这个问题。我们会发现，确实有一些基本的方法来思考 Bug（和“正确的程序”）——使用诸如 计算不可约性之类的概念（和计算可约性）。

我们将讨论的内容将使我们了解计算效率和 Bug 倾向之间的基本权衡，以及检测 Bug 的策略，以及对测试难度的期望。在此过程中，我们将能够阐明与软件验证和计算机安全以及人工智能系统生成的代码相关的一些根本问题。

从某种意义上说，我们要做的关键是认识到 Bug 的基本现象甚至在非常简单的程序中也已经发生。这意味着我们可以使用规则学的方法和直觉来研究有关Bug的基础问题。

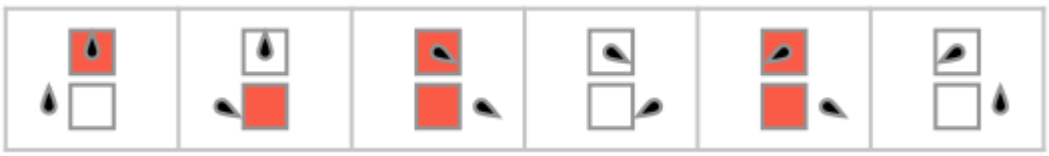
当我们编写程序时，我们通常知道我们想要程序做什么。如果在某些情况下它意外地没有做到这一点，我们认为这是 Bug。用更抽象的术语来说，我们通常脑子里有一个程序应该做什么的表示。为了让它适合我们的思维，这在计算上必须非常简单。但问题是与实际运行程序相关的计算过程是否在某种程度上相应简单。如果是的话，那么它有可能严格遵循我们所拥有的表示。

但令我惊讶的是，即使是结构非常简单的程序（或其片段），最终也可以进行某种意义上与任何事物一样复杂的计算。结果是这样的程序可以表现出计算不可约性——这意味着它们将不可避免地能够做我们无法预见和不期望的事情，而我们将认为是Bug。

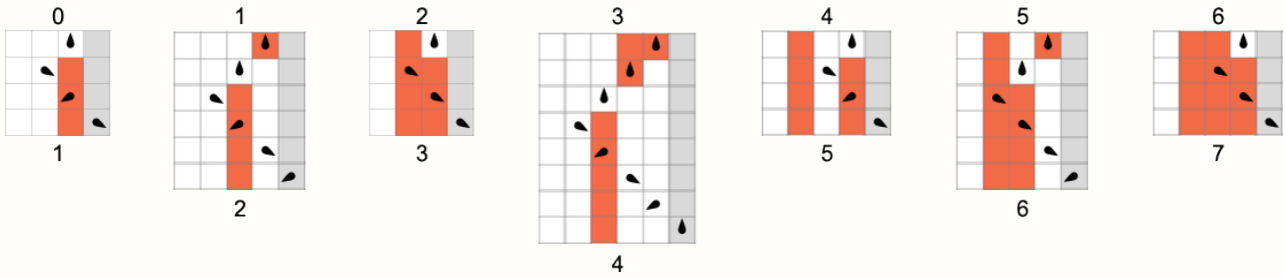
但即使我们人类无法独立做到这一点，我们是否可以期望通过人工智能或其他方式构建自动化系统来彻底排除此类 Bug？正如我们将讨论的，我们必然会在“计算可约性的局部区域”中取得进展。但关键是“潜伏在边缘”的是计算不可约性。当存在计算不可约性时，除了显式运行计算之外，没有什么可以决定会发生什么。这意味着，除非我们已经在特定情况下运行了我们的程序，否则我们将无法确定它会做什么。当它做我们不想要的事情时，我们将这些事情称为“Bug”。

图灵机中的 Bug

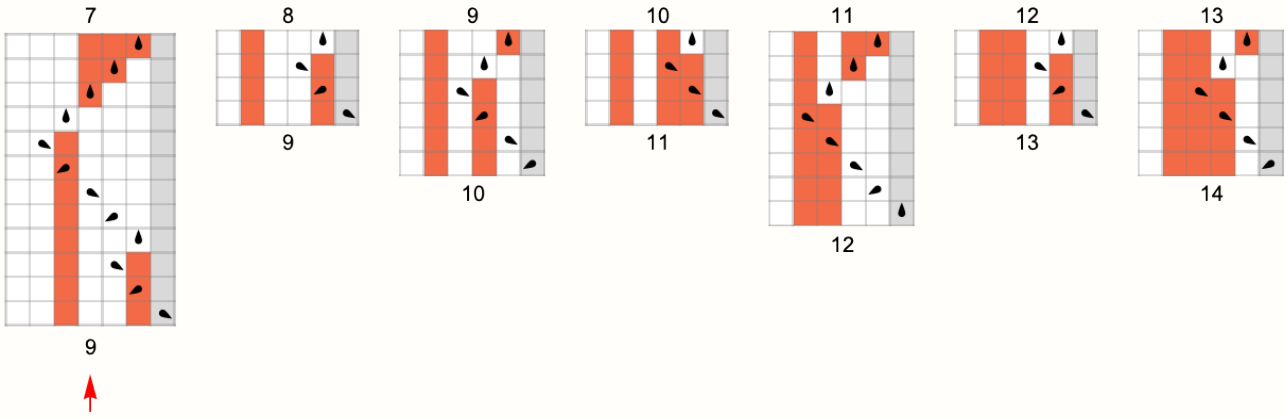
> 为了开始我们的探索，我们将看看一个非常简单且长期存在的计算模型：图灵机。考虑具有规则的（3状态，2颜色）图灵机：



想象一下，我们最初将一个整数n的二进制数字放在图灵机的磁带上，然后运行该机器，看看当它的头比开始时更向右移动时，磁带上有什么数字：

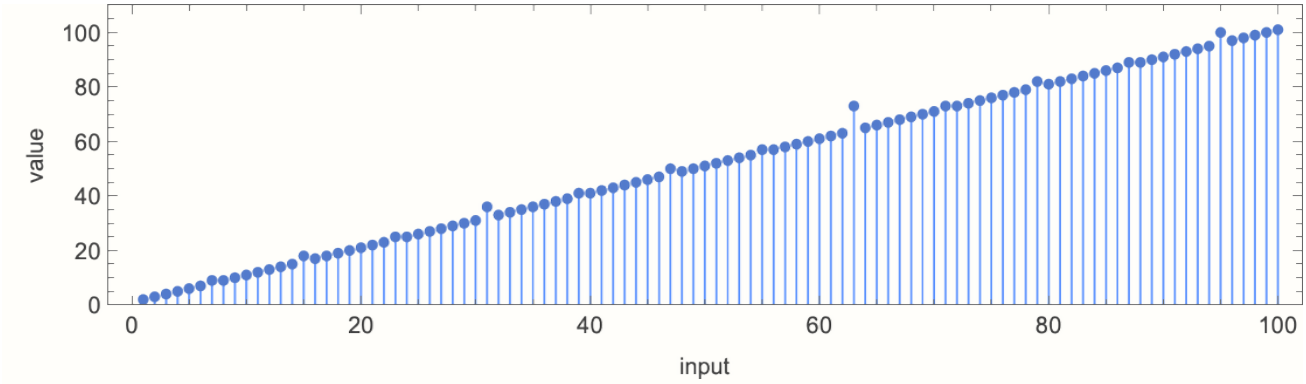


看看这些例子，我们可能会得出这样的结论：图灵机正在计算函数 $n \mapsto n + 1$. 但是让我们看看如果我们尝试更大的输入会发生什么：

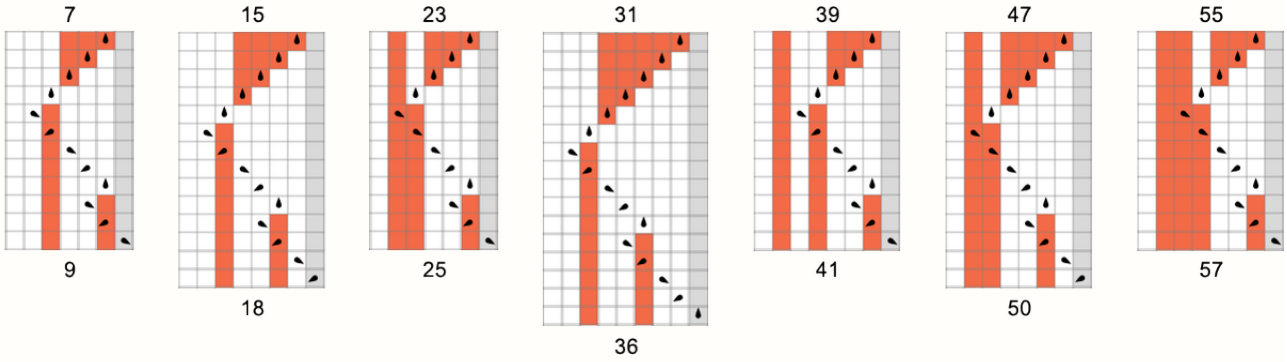


是的，对于输入 7，有一个 Bug！图灵机输出的不是 7 + 1 = 8，而是 9。

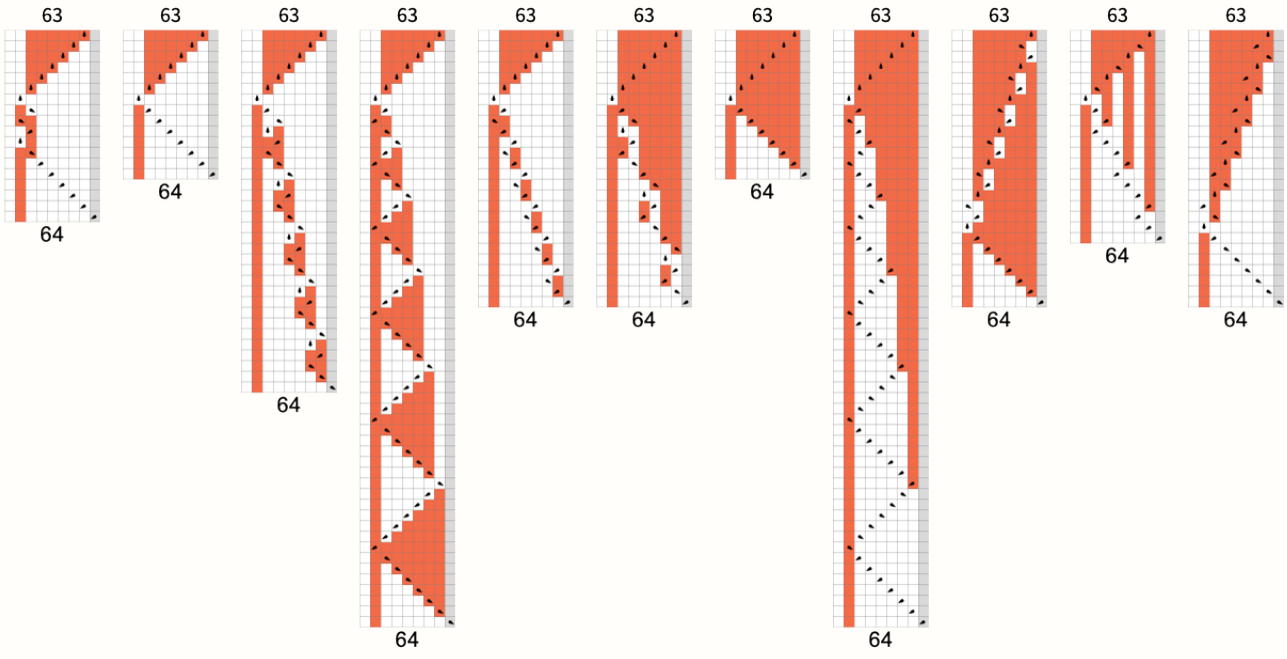
绘制图灵机计算的值，我们看到一系列小故障：



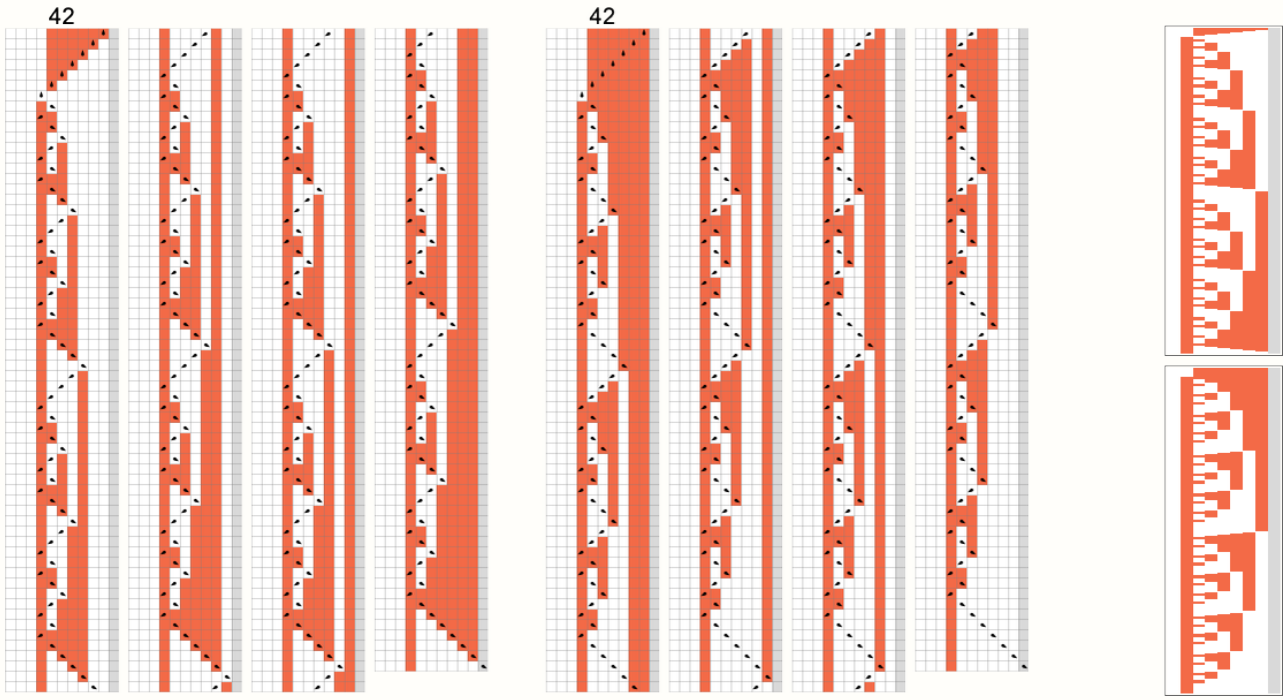
结果，异常出现在 8k – 1 形式的 n 值处（其大小最终为 `Floor[2^(IntegerExponent[k + 1, 2] + 3)/7]`）。如果我们查看图灵机的实际详细操作，我们就会看到在这些情况下“导致 Bug”的原因。对于n的值，图灵机“正确运行”，其头部向左扫描，如果需要传播进位位，那么，当它向右扫描时，它不会对磁带上的配置进行进一步的更改。但是，如果n是一个二进制表示以111结尾的数字（即它是一个8k – 1形式的数字，等于7模8），那么当磁头向右扫过时，它会“稍微额外地移动”并在磁带上“写下不正确的数字”：



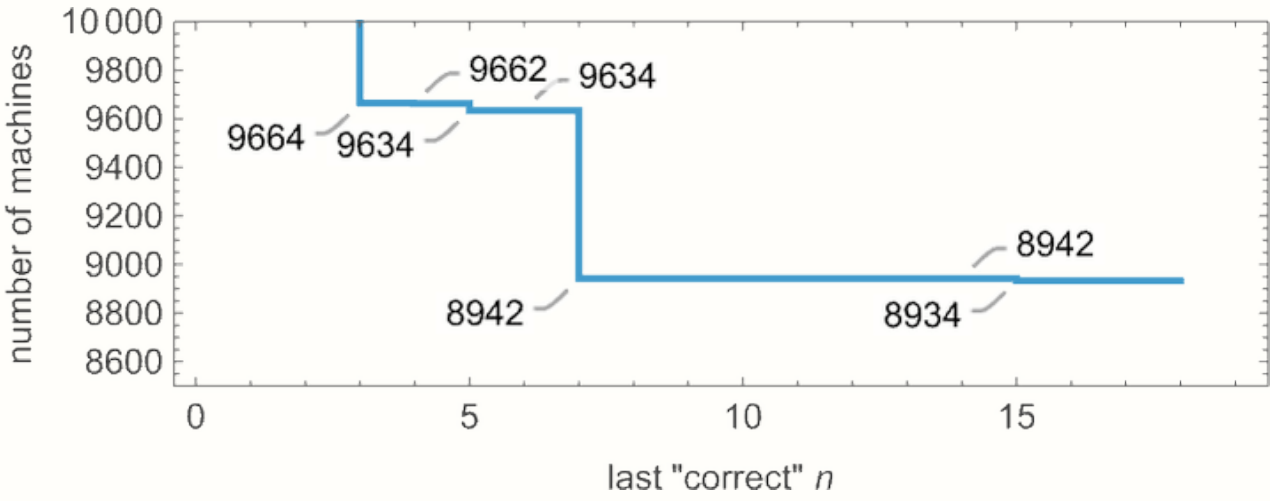
有很多“正确的”图灵机可以计算函数n + 1；事实上，有 3 种状态和 2 种颜色，实际上有 8934 个，其中大多数以相当简单的方式运行



但有些操作方式乍一看几乎是荒谬的复杂方式（尽管最终——如右侧压缩的渲染图所示——它们只是嵌套的）：



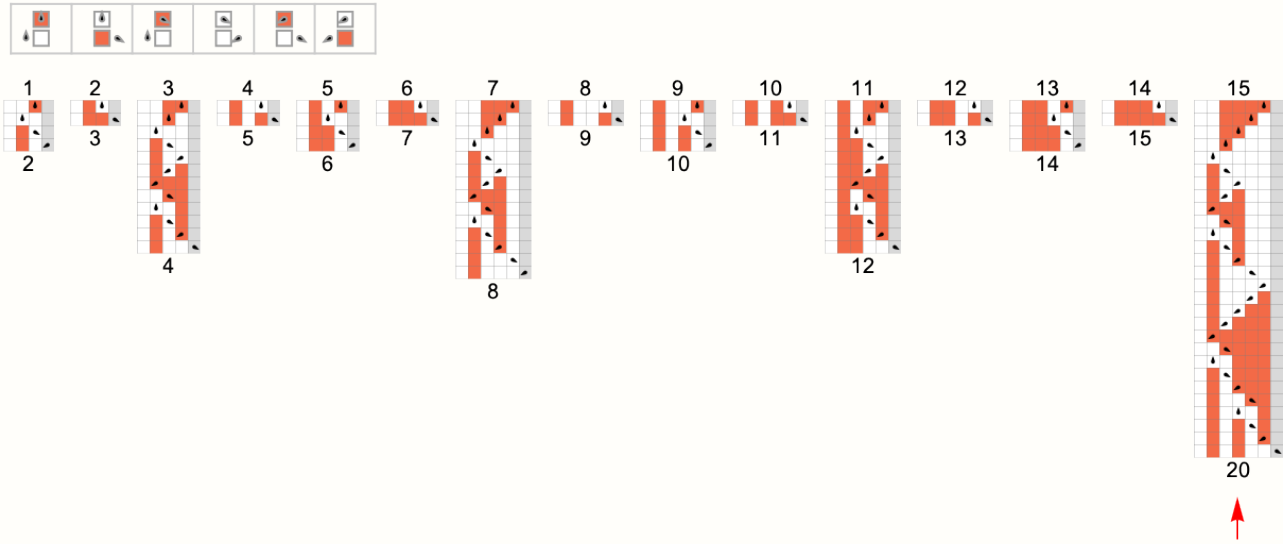
但是，如果像这样的图灵机“会出现 Bug”，那么在它出现之前还需要走多远？下面是“正确计算”n + 1 直到给定值 n 的 3 状态、2 颜色机器的数量图



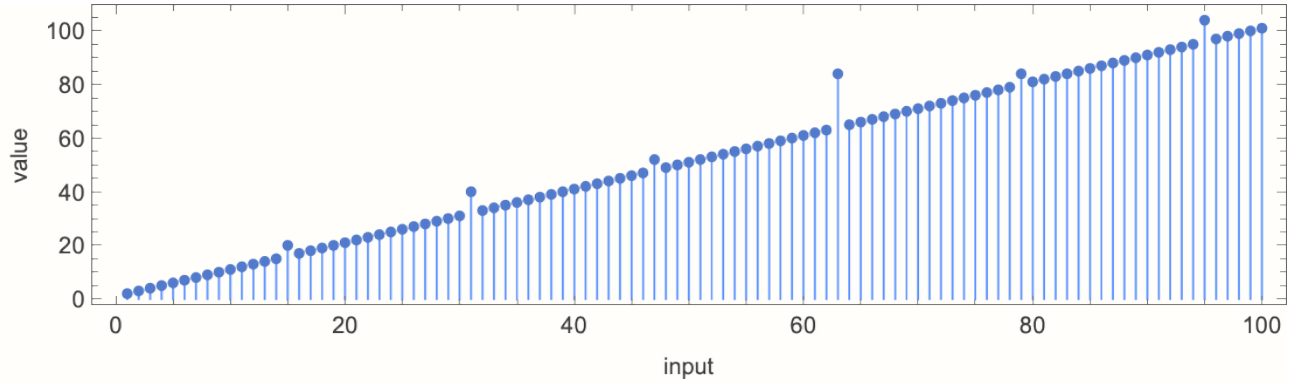
以下是“第一个 Bug”发生位置的摘要：

position of first bug	number of machines	fraction of machines
3	225 474	0.08
4	2	7×10^{-7}
5	28	9×10^{-6}
7	692	2×10^{-4}
15	8	3×10^{-6}
∞	8934	0.003

“最阴险”的 Bug 直到 n = 15 才会发生。具有此类 Bug 的机器的示例是：

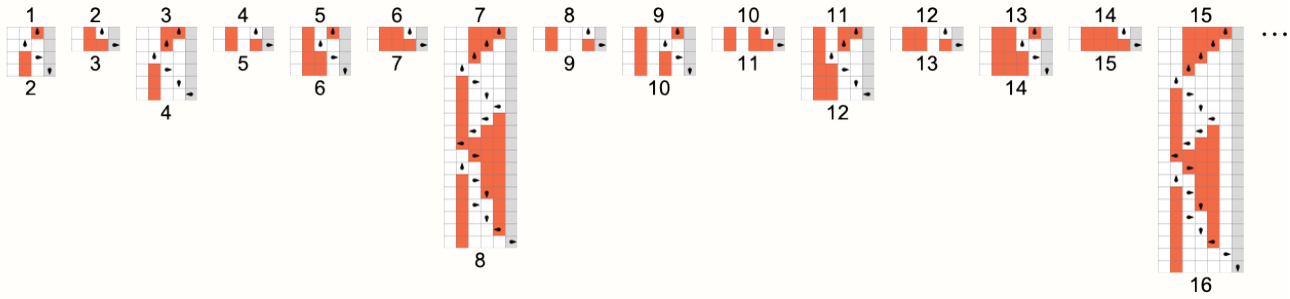


本例中的值序列为：

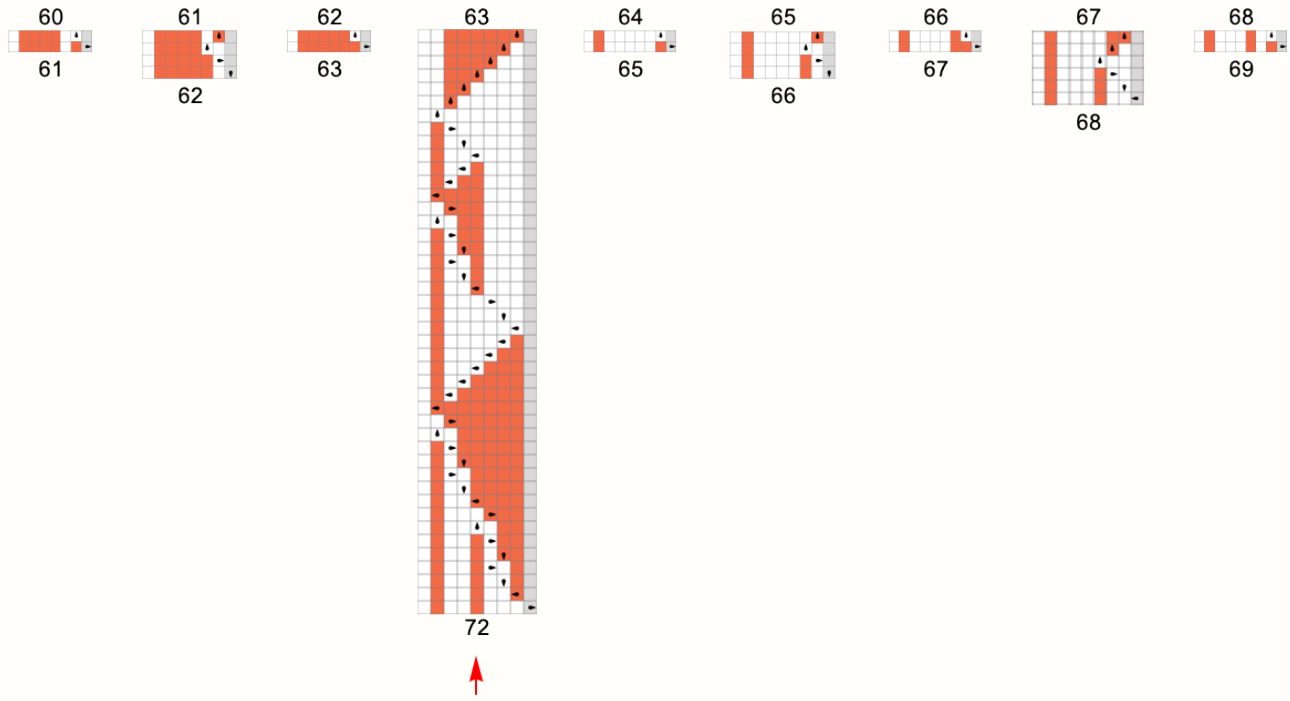


而且，是的，如果运行这台机器输入 0 到 14 的输入，一切都会正常工作，但突然在 n = 15 时，机器不再给出输出 15 + 1 = 16，而是给出 20！（故障在 16k – 1 形式的所有输入处持续存在，即在其二进制表示形式的末尾有 1111。）

更大的图灵机又如何？在约 40 亿台 4 状态图灵机中，超过 900 万台能够成功计算 n + 1。但其中也有一些 Bug 相当隐蔽。例如，按标准 Wolfram 语言 `TuringMachine` 编号方案编号的机器 118050142，对从 0 到 62 的所有 n，都能“正确”计算 n + 1

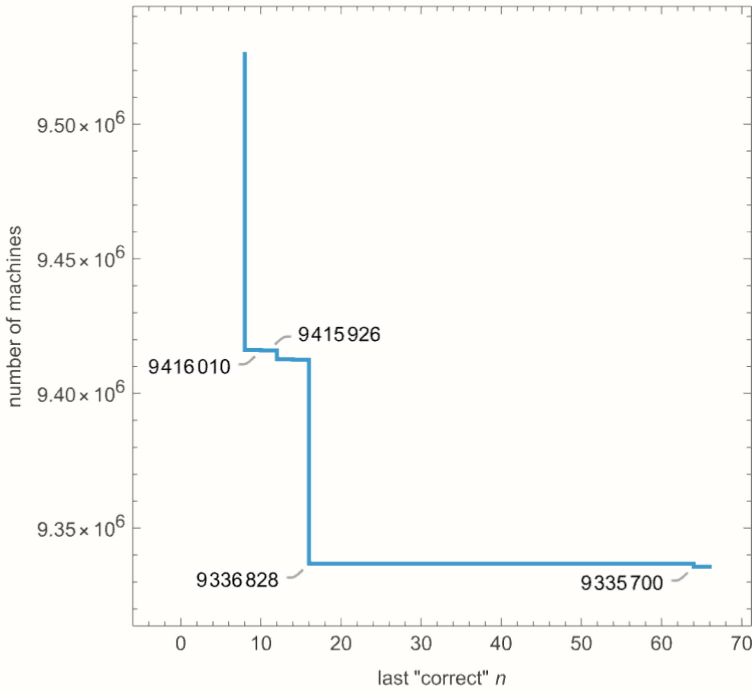


但突然在 n = 63 时不给出 64，而是给出 72：



纵观所有 4 状态图灵机，我们可以总结出 Bug 出现在“几乎”计算 n + 1 的图灵机中：

position of first bug	number of machines	fraction of machines
4	31 044	7×10^{-6}
5	117 384	3×10^{-5}
6	3264	8×10^{-7}
7	1 778 544	4×10^{-4}
8	12	3×10^{-9}
9	114	3×10^{-8}
10	84	2×10^{-8}
11	3186	7×10^{-7}
12	42	1×10^{-8}
13	174	4×10^{-8}
15	75 696	2×10^{-5}
19	6	1×10^{-9}
63	1122	3×10^{-7}
∞	9 335 700	0.002



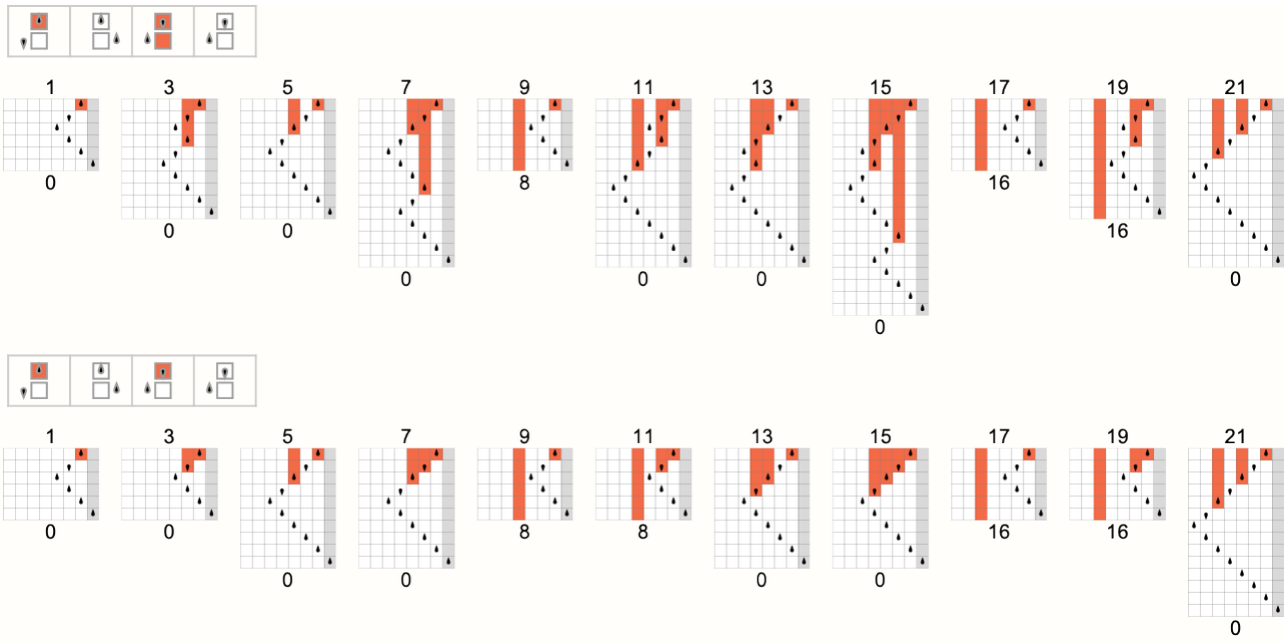
这里的一个显著特征是，Bug 的 n 的“最高风险”值的形式为 2^{i-1}，即 n 的二进制表示仅包含 1s 的情况。也许还值得注意的是，至少对于 s = 2、3 和 4 状态，“最阴险”的 Bug 总

什么才算 Bug?

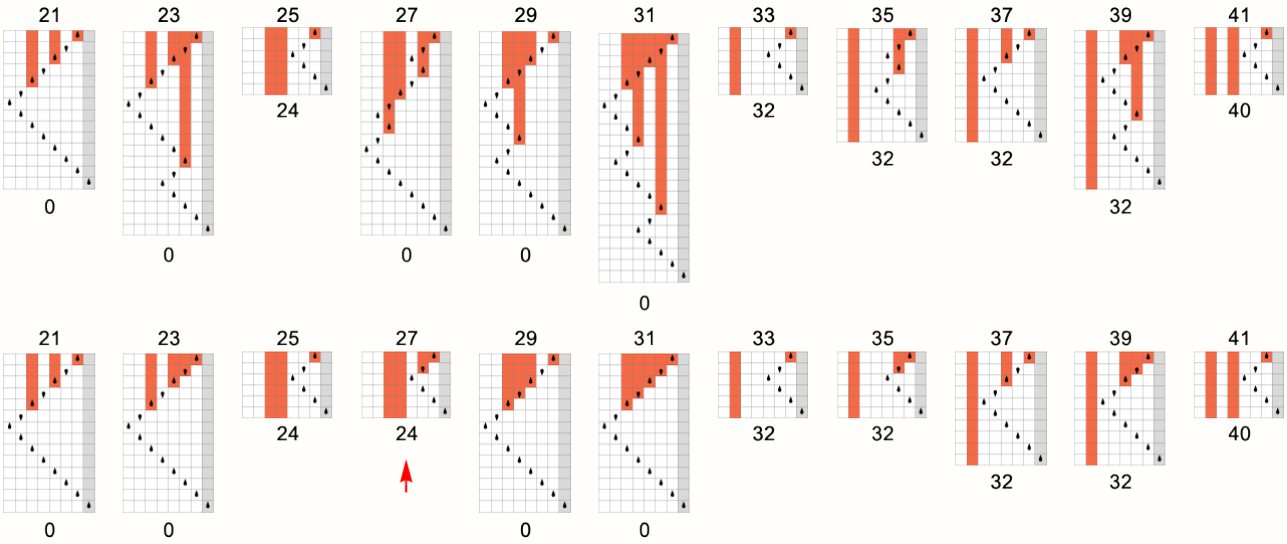
> 为了判断一个程序是否有 Bug，我们必须知道该程序应该做什么，或者不应该做什么。为此，我们必须有某种精确的规范。当然，程序本身提供了它要做什么的规范。就这个规范而言，程序将始终只是“做它所做的事”，不可能“出现 Bug”。但通常我们想象的规范不仅仅是程序本身，而是具有一些更容易吸收的形式（例如上一节中的简单数学公式“n + 1”）。与这些规范相关的是，程序可以具有 Bug。

最终使用的规范有几种不同类型。最直接的本质上只是用不同的（通常是更高级别的）语言重述程序（例如 Wolfram 语言 与 C++，或者数学符号与图灵机规则）。更简单的版本只是将该程序重新表述为同一类型的不同程序。如果两个程序不执行相同的操作，则可以将其视为 Bug。 （例如，如果编译器优化器更改了程序的功能，则它是 Bug。）

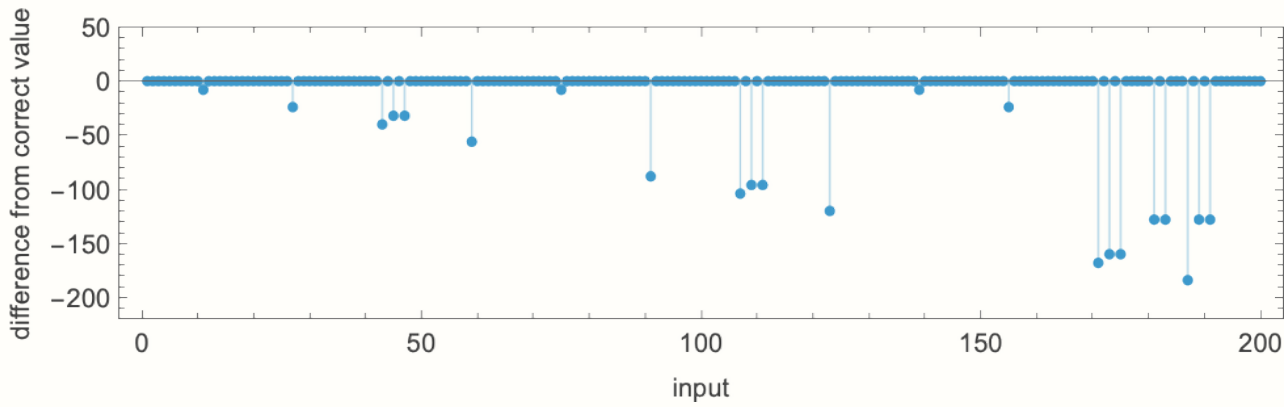
为了看一个例子，我们可以再次看看图灵机。这里有两台机器看起来它们正在计算相同的函数



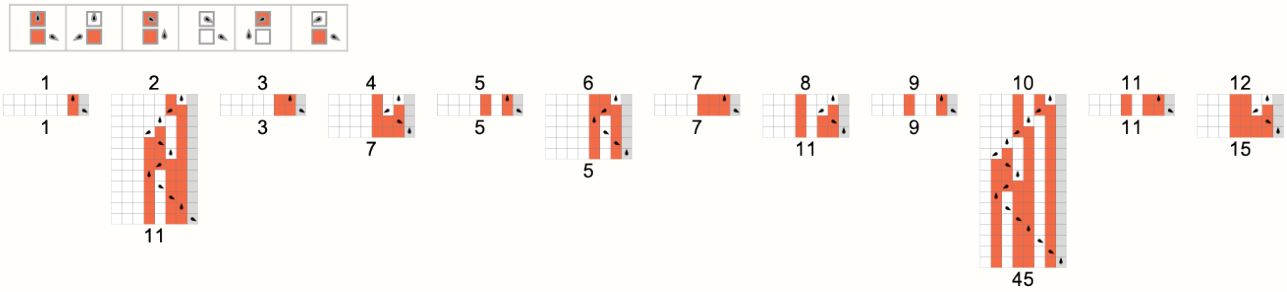
但突然，在 n = 27 时，做一些不同的事情：



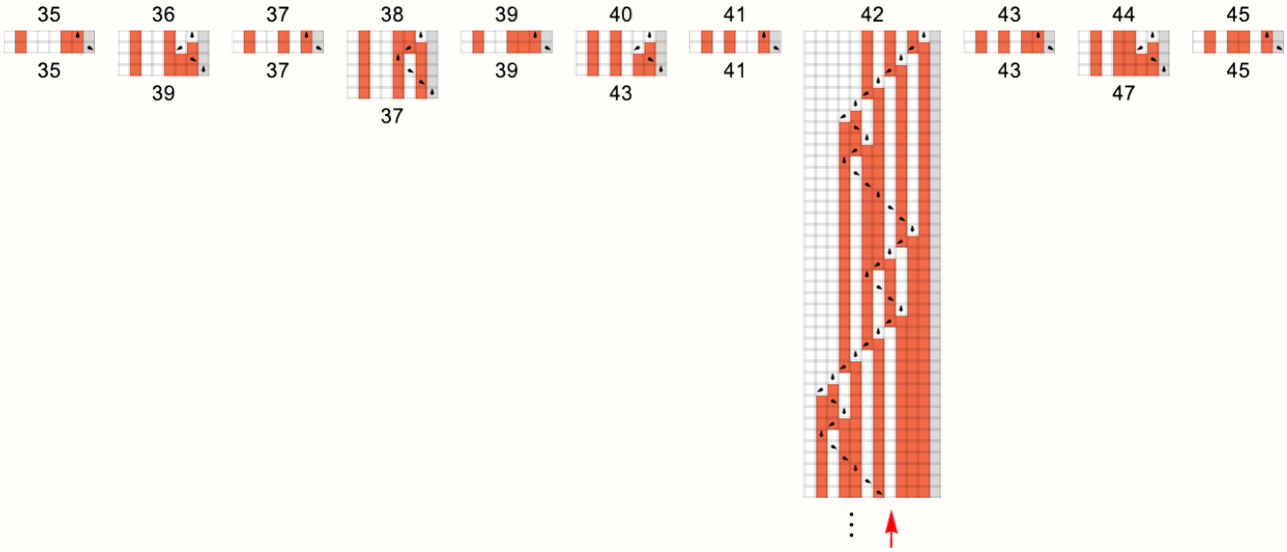
事实上，这种情况下出现“Bug”（即输出差异）的模式出奇地复杂：



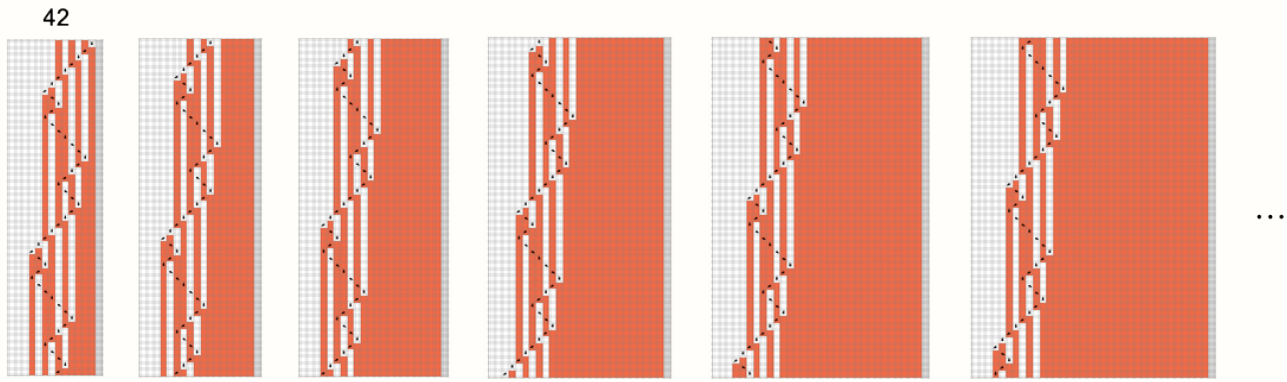
另一种通用类型的规范不是说明程序应该做什么，而是说明程序不应该做什么。因此，例如，对于我们的图灵机，我们可以指定我们永远不希望它们“失控”并开始使用无限量的内存（即磁带）。这是一个图灵机示例，看起来它只是在愉快地计算输出：



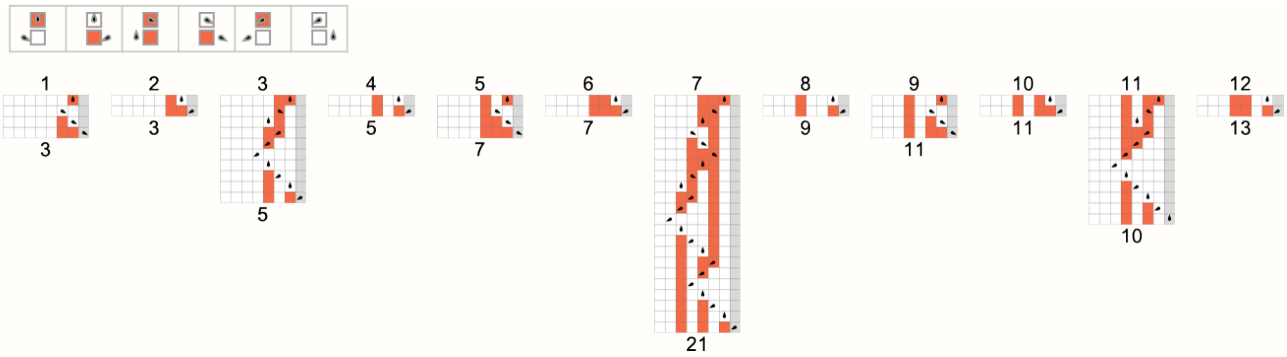
但突然间，在 n = 42 时，出现了“Bug”



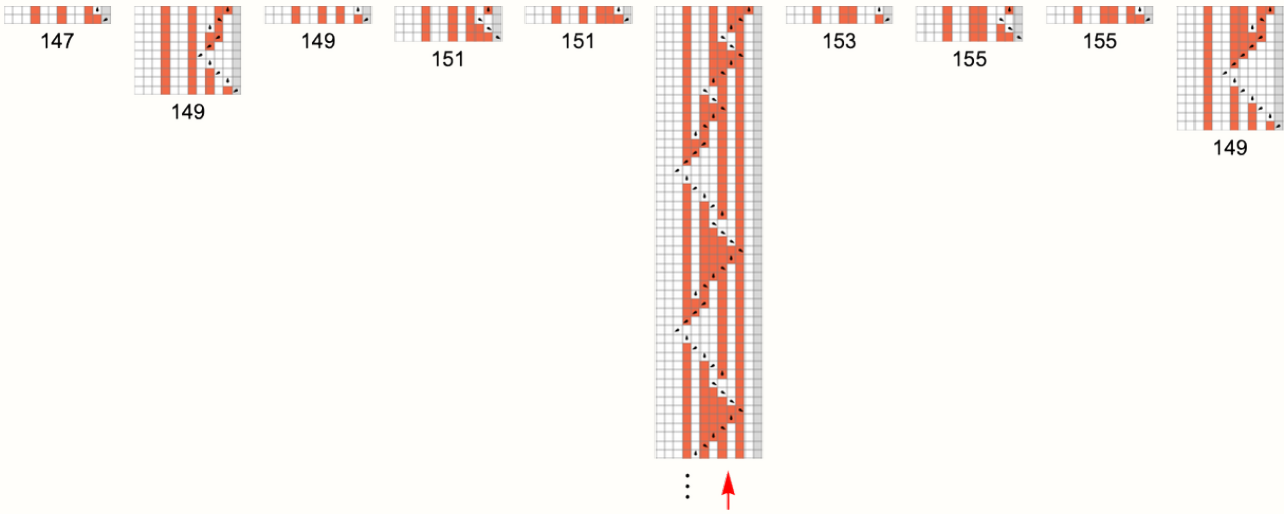
机器“失控”，图案永远越来越宽：



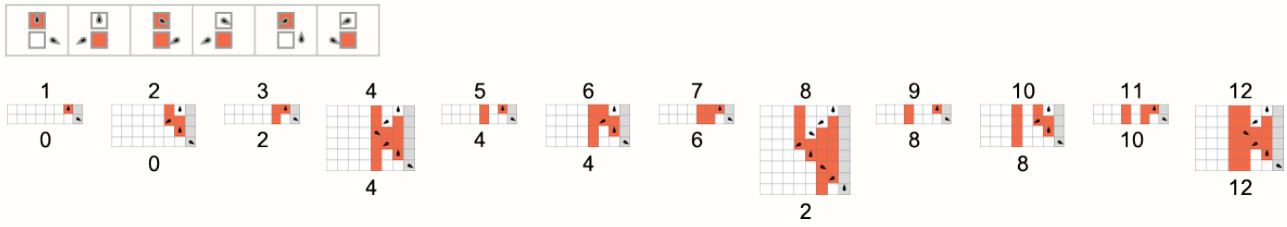
这是另一台机器，它似乎又在愉快地计算输出：



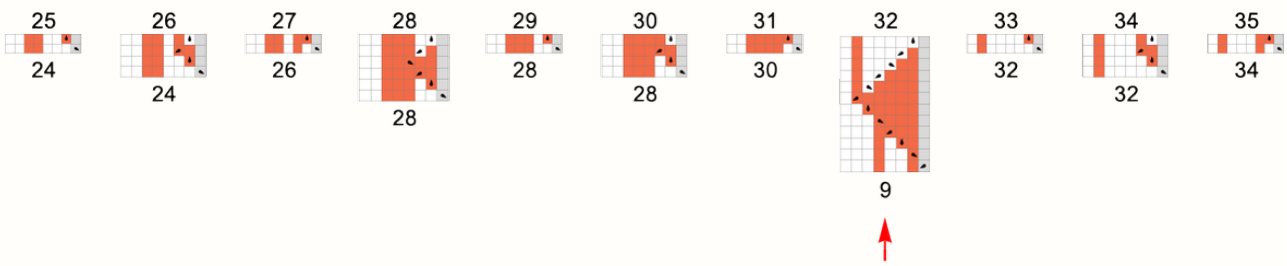
但是，当 n = 151 时，会发生一些病态的情况，机器进入无限循环：



我们还可以有一个规范，规定我们的图灵机的输出应该始终是偶数。这是一个图灵机，它看起来只产生偶数输出：



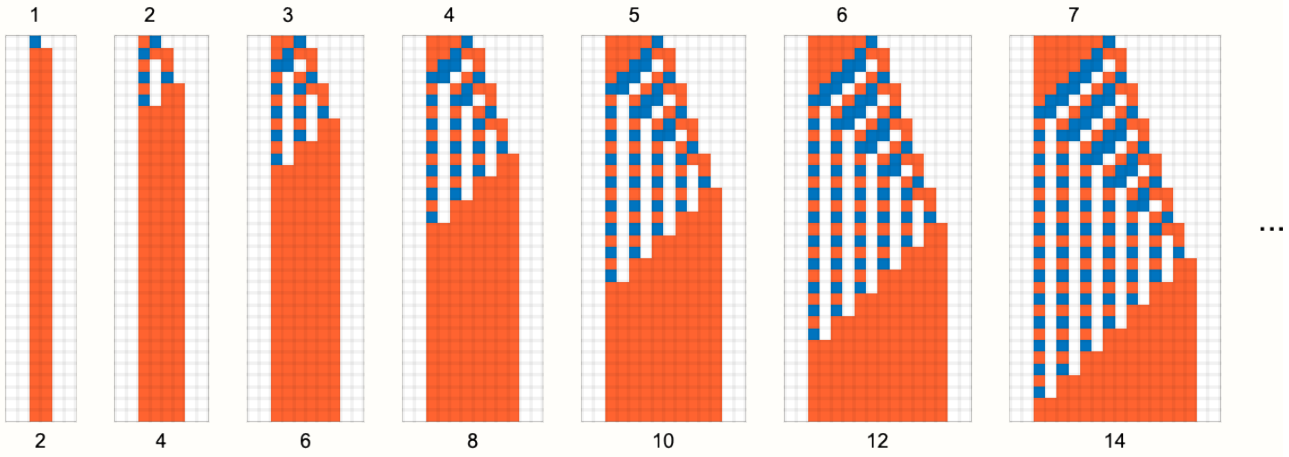
但当 n = 32 时，它突然给出了 9！



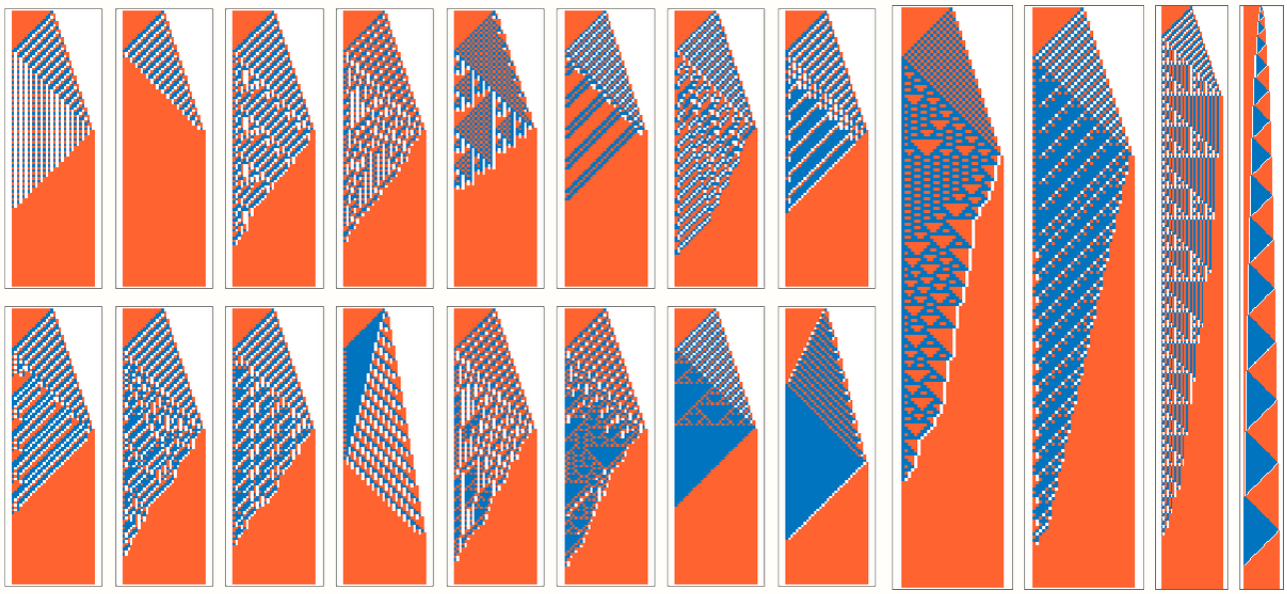
这似乎总是同一个故事：我们制定了一个规范，一些图灵机将精确地遵循它。但我们似乎总能找到那些似乎在追随它的人——至少在他们“出现 Bug”之前。换句话说，似乎无论我们给予什么限制，总会有机器以某种方式阴险地逃脱它——实际上“拥有隐蔽的 Bug”。

一个元胞自动机实例

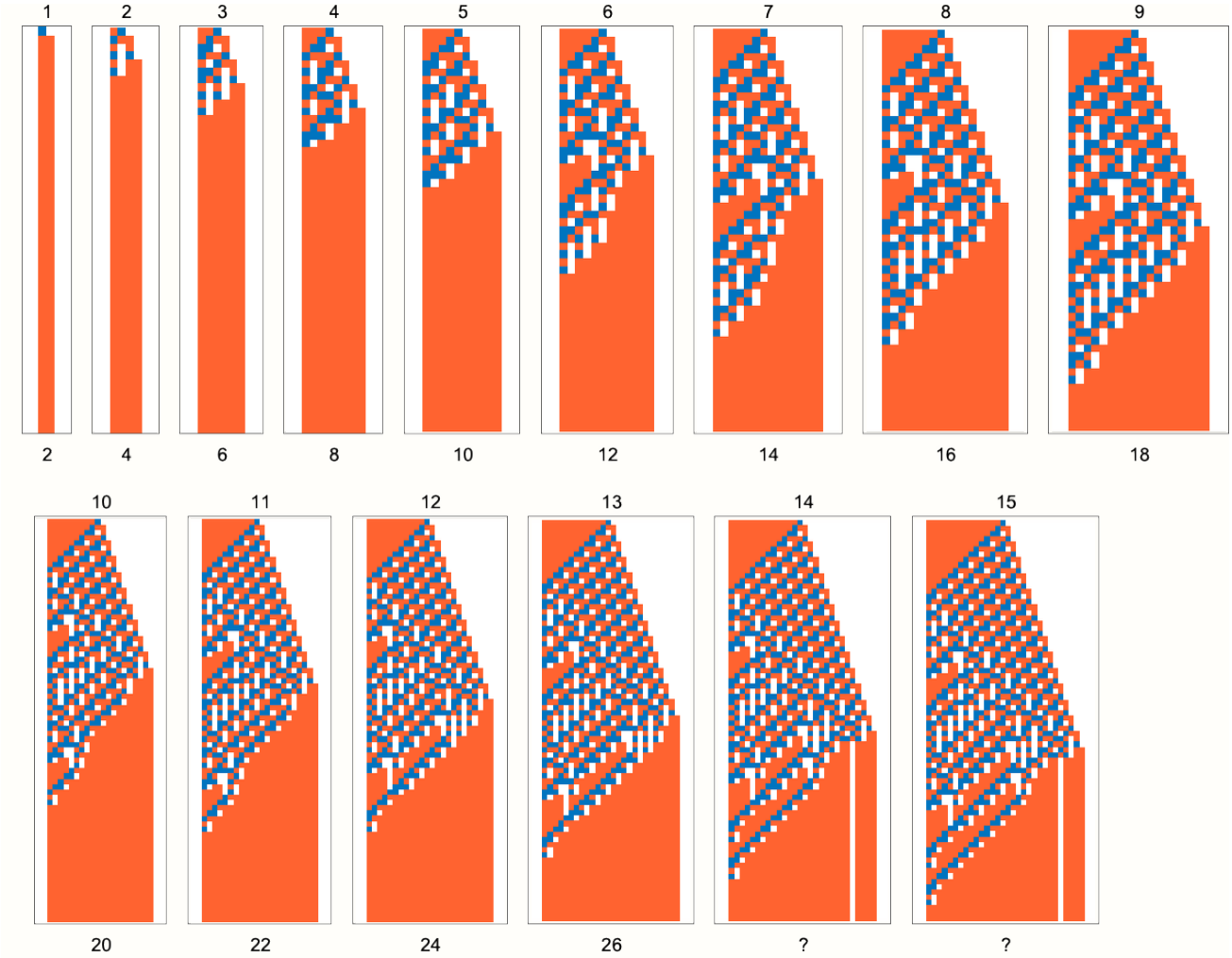
> 我们在简单图灵机中看到的“Bug”如何概括？作为另一个例子，让我们看一下元胞自动机。特别是，让我们看一下设置为“输入加倍”的元胞自动机，从某种意义上来说，从一块n（非白色）细胞开始，它们会产生2n（非白色）细胞：



与上面的图灵机示例一样，元胞自动机有多种方法（例如使用 k = 3 颜色）来完成我们想要的任务，其中一些方法比其他方法更有效：

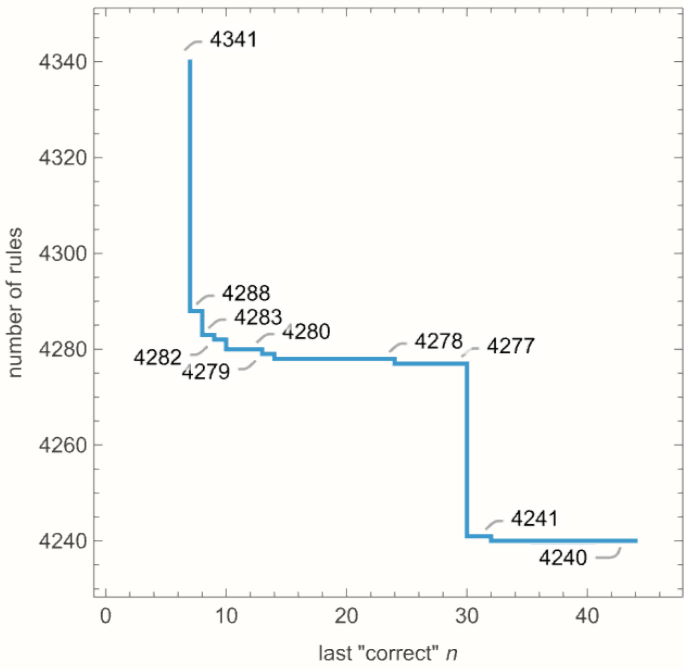


有时，看起来似乎某个特定规则可以完成这项工作，然后突然出现一个“Bug”，就像这里的n = 14：

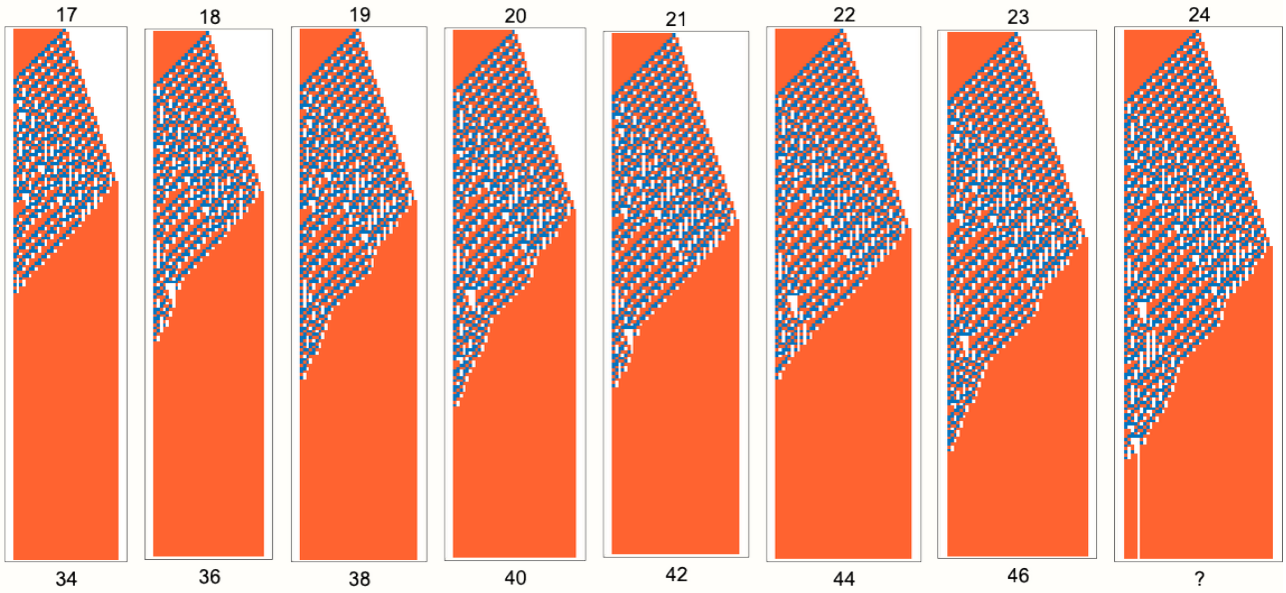


与我们的图灵机示例一样，“Bug”可能需要一段时间才能显示：

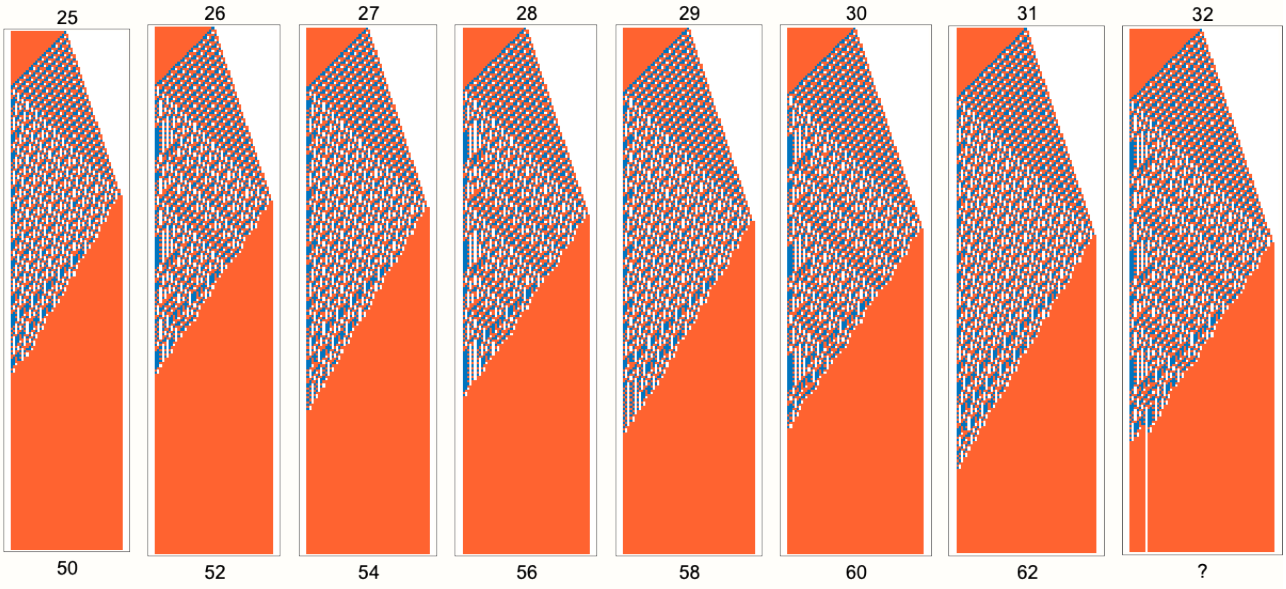
position of first bug	number of rules	fraction of rules
5	13 234	2×10^{-9}
6	353	5×10^{-11}
7	53	7×10^{-12}
8	5	7×10^{-13}
9	1	1×10^{-13}
10	2	3×10^{-13}
13	1	1×10^{-13}
14	1	1×10^{-13}
24	1	1×10^{-13}
30	36	5×10^{-12}
32	1	1×10^{-13}
∞	4240	2×10^{-8}



下面是第一个“Bug”出现在 n = 24 的示例：

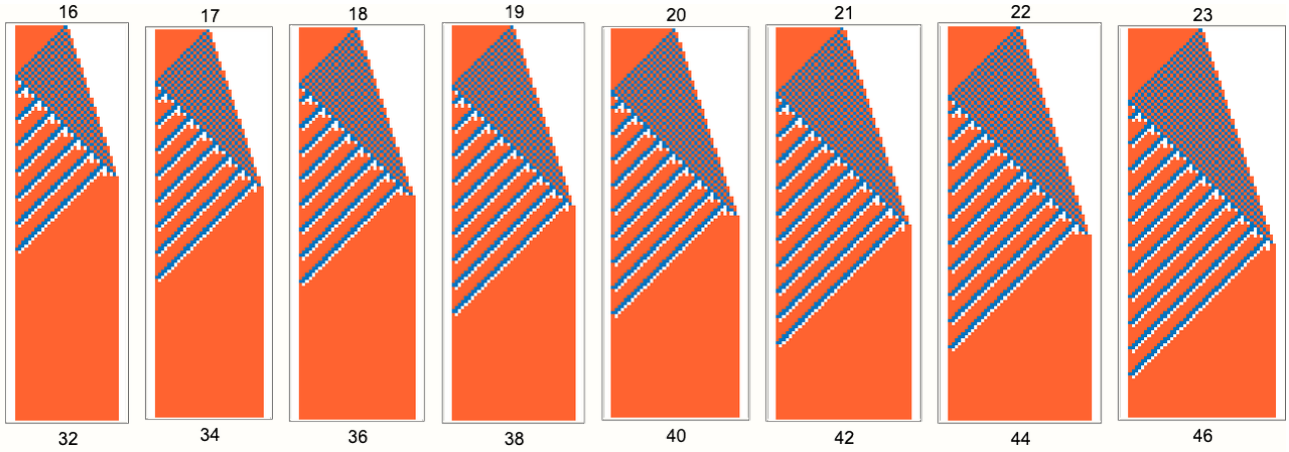


32 点：

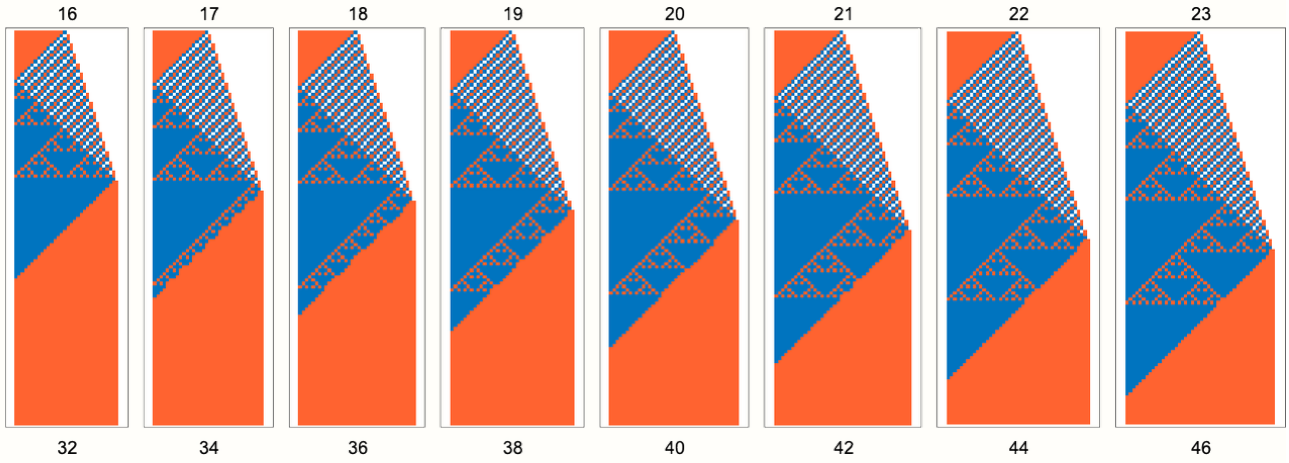


能预先判断是否会出现 Bug 吗?

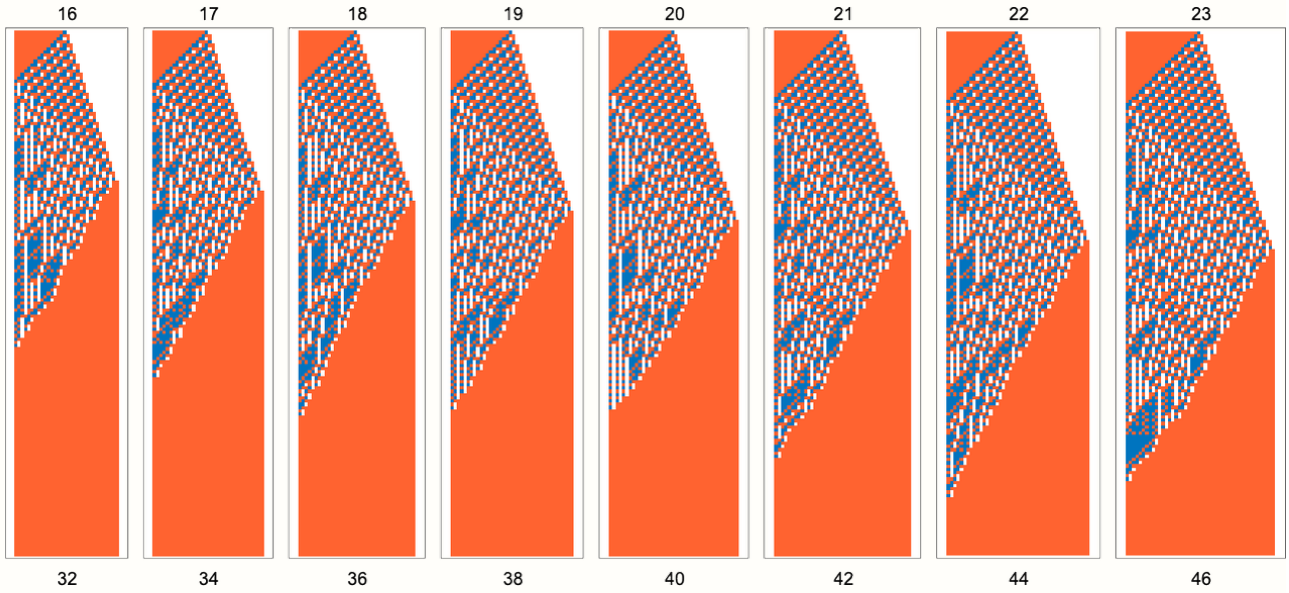
> 我们在上面看到的许多元胞自动机给出的模式足够简单，以至于“视觉上显而易见”，它们对于所有输入大小都会“正确地加倍”：



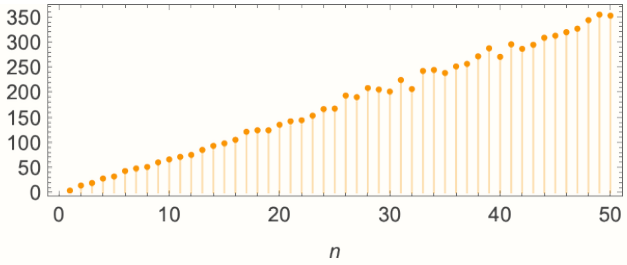
有时不太明显，但似乎仍然相当清楚，没有任何东西可以逃脱某个特定的信封，因此不可能有 Bug：

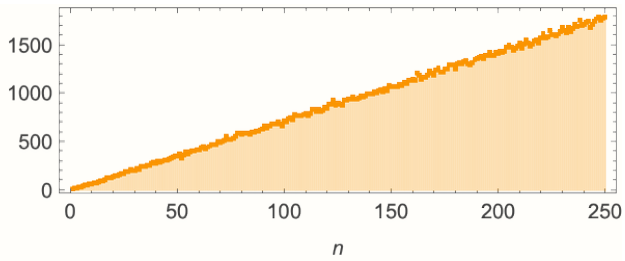


但是在这样的情况下该怎么办：

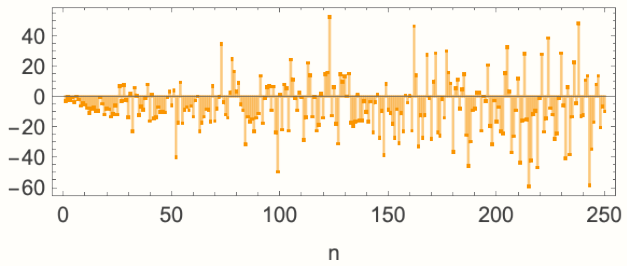


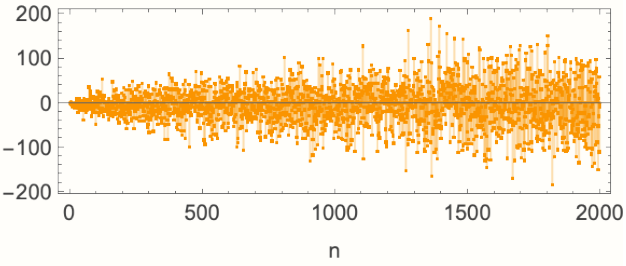
它看起来与我们上面看到的最终显示 Bug 的案例非常相似。但如果我们将这种情况运行到 n = 100000，则不会出现 Bug。我们可以进行更多分析，并将模式稳定所需的步数绘制为 n 的函数：





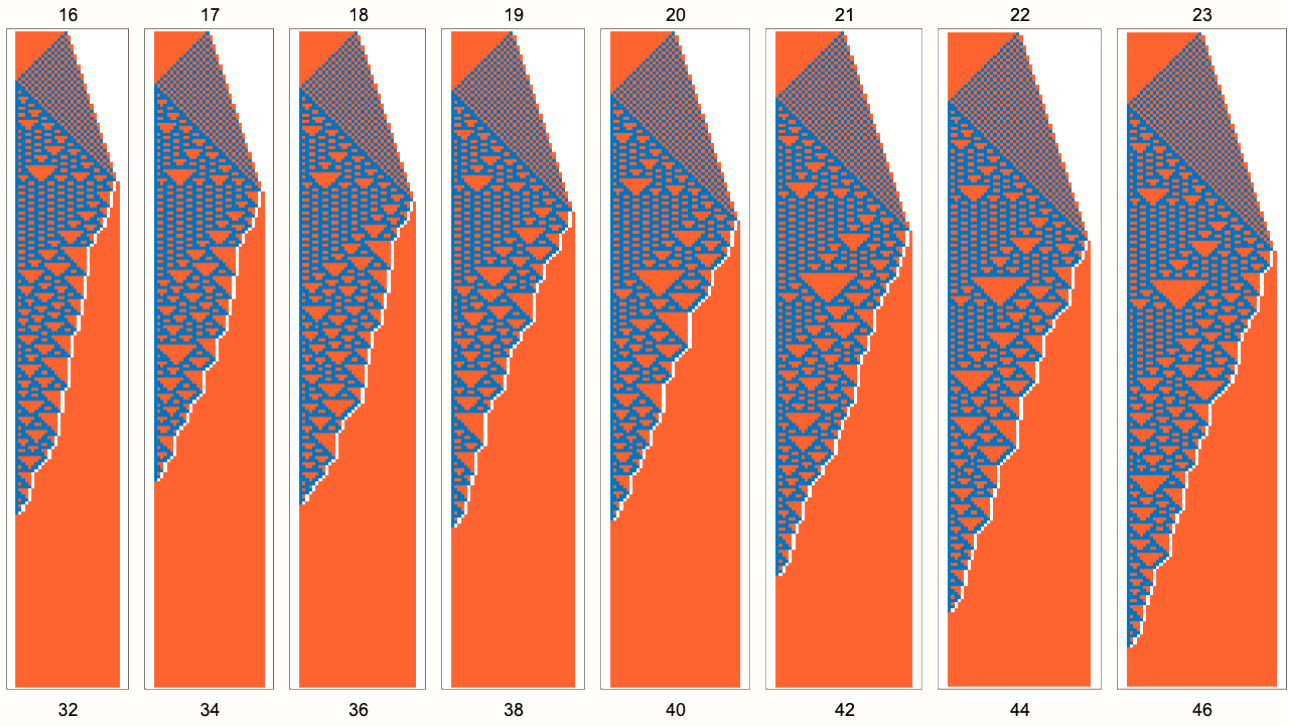
总体趋势很简单，对于输入 n 来说，7.16n 步非常适合。但似乎存在相当随机的波动：



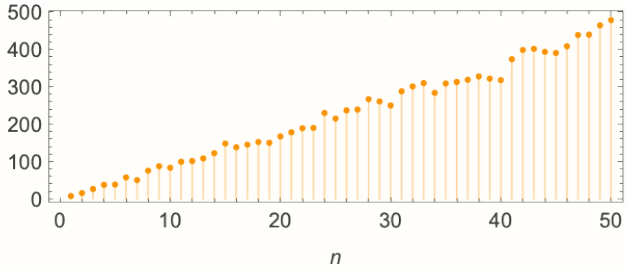


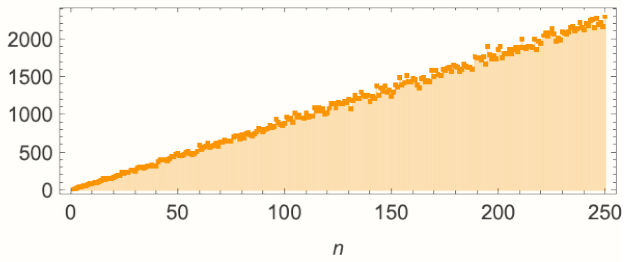
这些波动至少看起来大致像 $\sqrt{n}$ 一样增长。但没有明显的方法可以确定并保证不会发生任何疯狂的事情，这会导致 Bug。

像这样的案例呢：

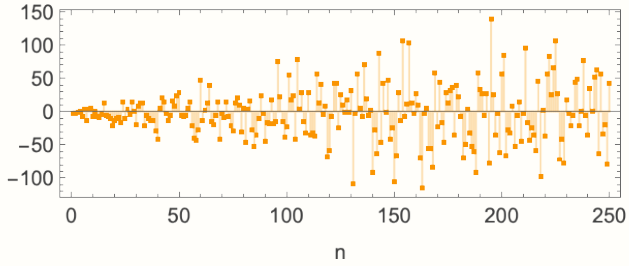


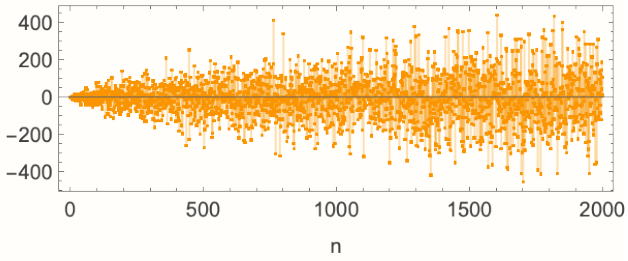
该模式是否会始终被“包含”，还是最终会出现导致 Bug 的“泄漏”？我们可以再次绘制稳定时间，该时间似乎再次接近线性增加，并且现在非常适合 9n：





但也存在看似随机的波动：

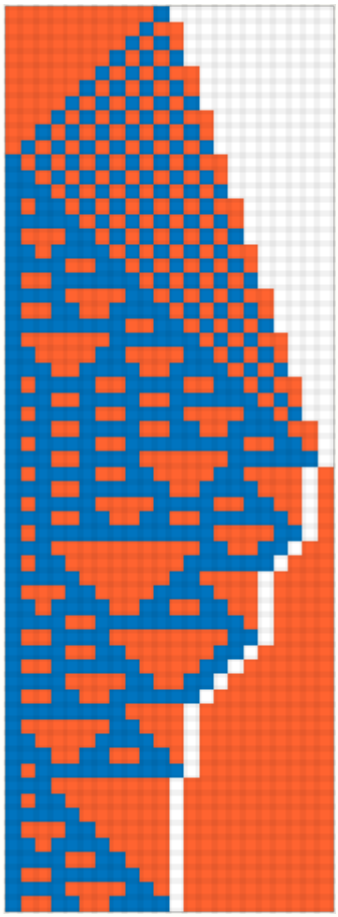




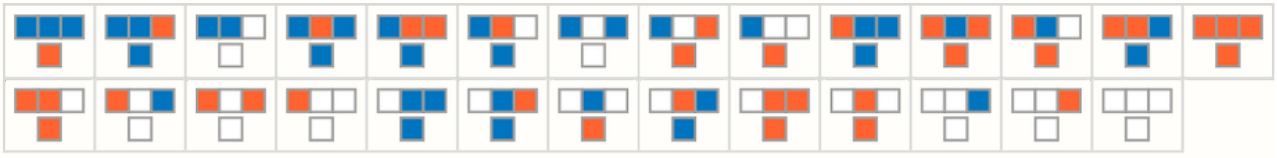
问题又是这些是否会以某种方式“疯狂”，并导致 Bug。

好吧，事实证明，在这种情况下，我们可以提出一个论点，认为这永远不会发生。如果只看稳定时间之类的东西，就很难看出这一点。但它着眼于整个计算过程——这对于元胞自动机来说很容易做到——事情变得更加清晰。

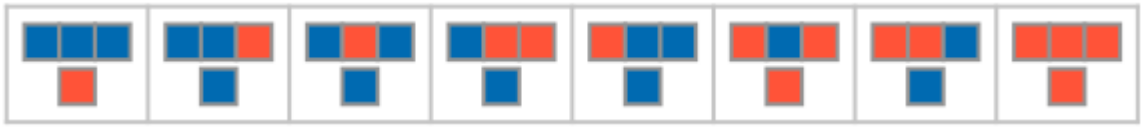
举个小例子，我们看到两个重要的特征：首先，白色的“外皮”总是要么停留在同一个地方，要么向左移动。其次，在果皮内部，该图案仅涉及 和 ：



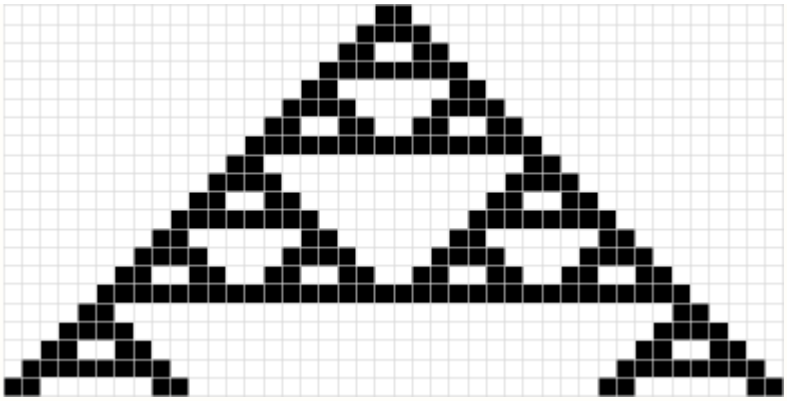
外皮的行为（我们可以根据底层元胞自动机规则轻松确定）确保复杂的模式永远不会扩展。但举例来说，它是否会最终变得周期性，从而使整个模式永远不稳定？嗯，这取决于外皮内部的“有效规则”。我们可以通过采用基本的元胞自动机规则来了解这是什么



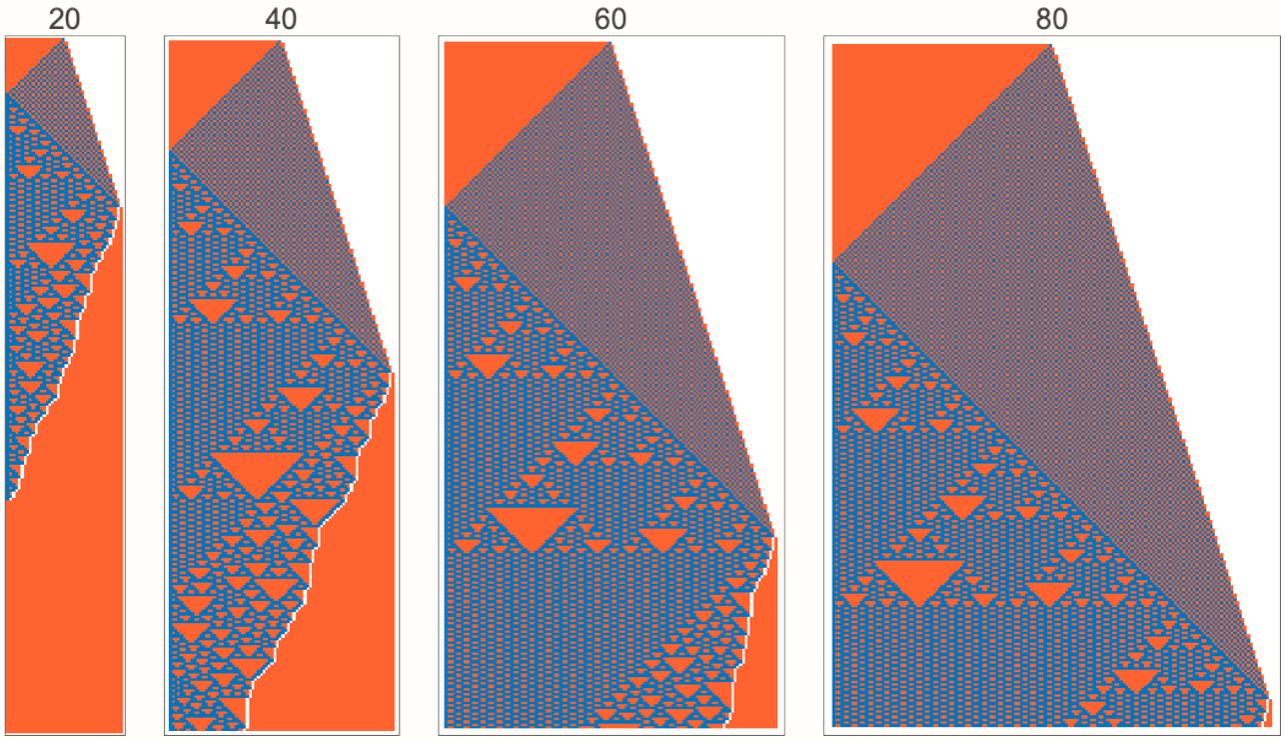
并删除所有涉及 □ 的案例：



答案是，它正是规则122基本元胞自动机，其中■代表■，■代表□。规则 122 的特点是，当有成对的相同单元格在一起时，它会模拟规则 90，并生成一个简单的嵌套模式：



该模式的类似物出现在原始元胞自动机中：



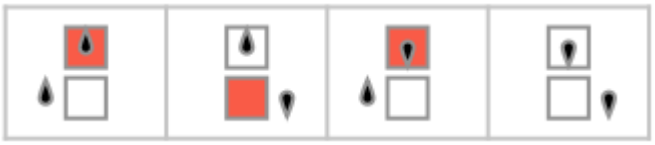
但在某些时候，这种模式会“从右侧中断”——此后，“外皮”会继续侵蚀它。图案变成周期性的唯一方法是果皮停止向左移动，而发生这种情况的唯一方法是果皮内的图案变成“全部 ”（或有效规则 122 中的全部 ）。因此，这就简化为规则 122 模式是否会简单消失（并变成全部 ）的问题。但从规则的结构来看，这是不可能的：一旦存在 单元，所有后续步骤中都必须存在 单元。

所以，是的，尽管这种情况下的模式看起来很复杂，但它们最终必须始终稳定，并且底层元胞自动机必须始终正确地“加倍其输入”，而不需要 Bug。

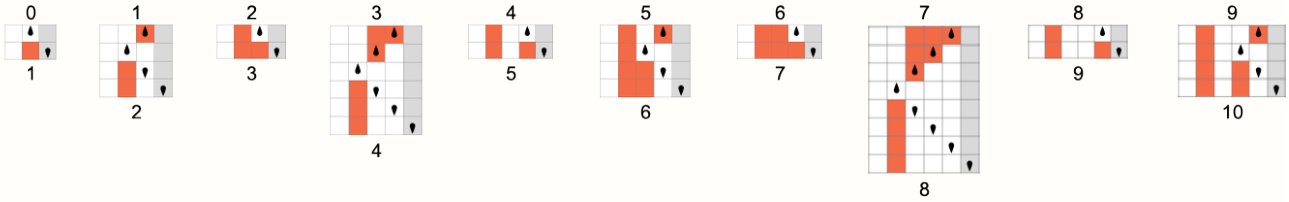
如果元胞自动机的行为“视觉上很简单”，我们可以立即知道它不会显示 Bug。如果它在视觉上很复杂，我们可能会担心可能存在 Bug。但仍然存在这样的情况：尽管如此复杂，我们所关心的行为特征（这里是任何输入的成功加倍）仍然总是可靠地发生，而不需要任何 Bug。

那么形式化证明呢?

> 让我们回到上面讨论的图灵机。考虑一台具有以下规则的机器:



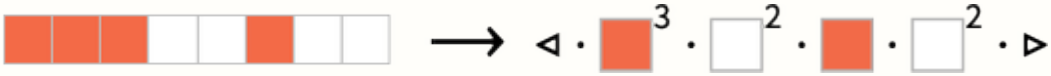
我们可以针对输入序列 n 运行它，并看到它似乎总是计算 n + 1:



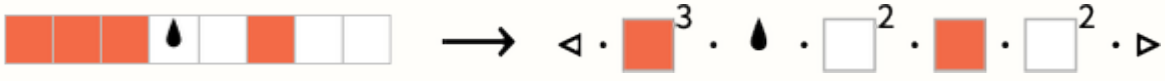
事实上，看看这台机器的实际行为，不难发现它必须始终计算n + 1。但是我们可以正式证明这一点吗?

为了建立一个简化的证明，我们需要一个能够包含图灵机的行为和我们想要对其做出的陈述的符号。实现这一点的基本方法是使用符号表达式来表示一切（是的，如 Wolfram 语言）。事实证明，将图灵机的磁带表示为链表是很方便的——连续的单元通过（右关联）二元运算符（此处·）连接在一起，并且其“末端”由标记指示（此处 ◁ 和 ▷）:

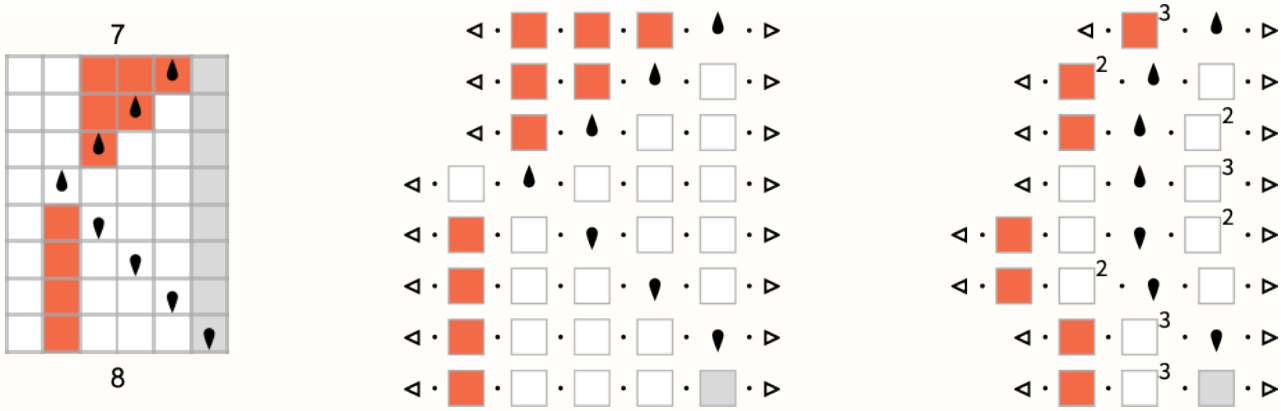
那么对游程长度编码实际上很方便:



我们只需将其作为“链接列表”的一部分来指示头部的位置，紧跟在其所在的单元格之后:

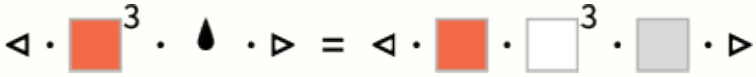


图灵机演变的一个例子是（其中 ▫ 表示什么相当于停止状态）:

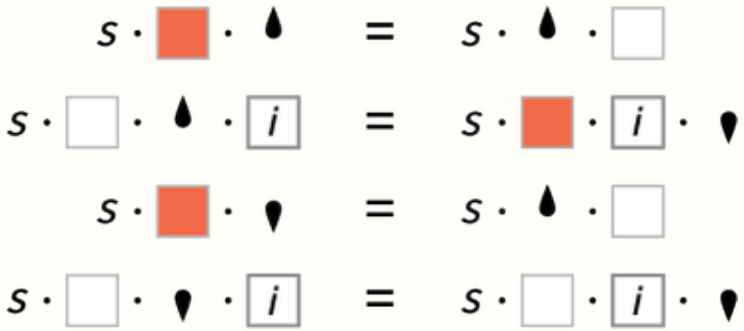


我们可以将图灵进化的这个特定例子视为在某种意义上证明

或者

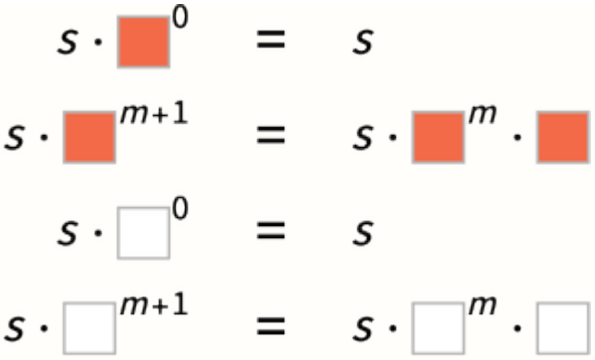


通过此设置，我们现在可以以“类代数”公理的形式指定图灵机的规则（其中s代表任意“子带”，i代表磁带上的一个单元，该单元可以可以是 或 ）：



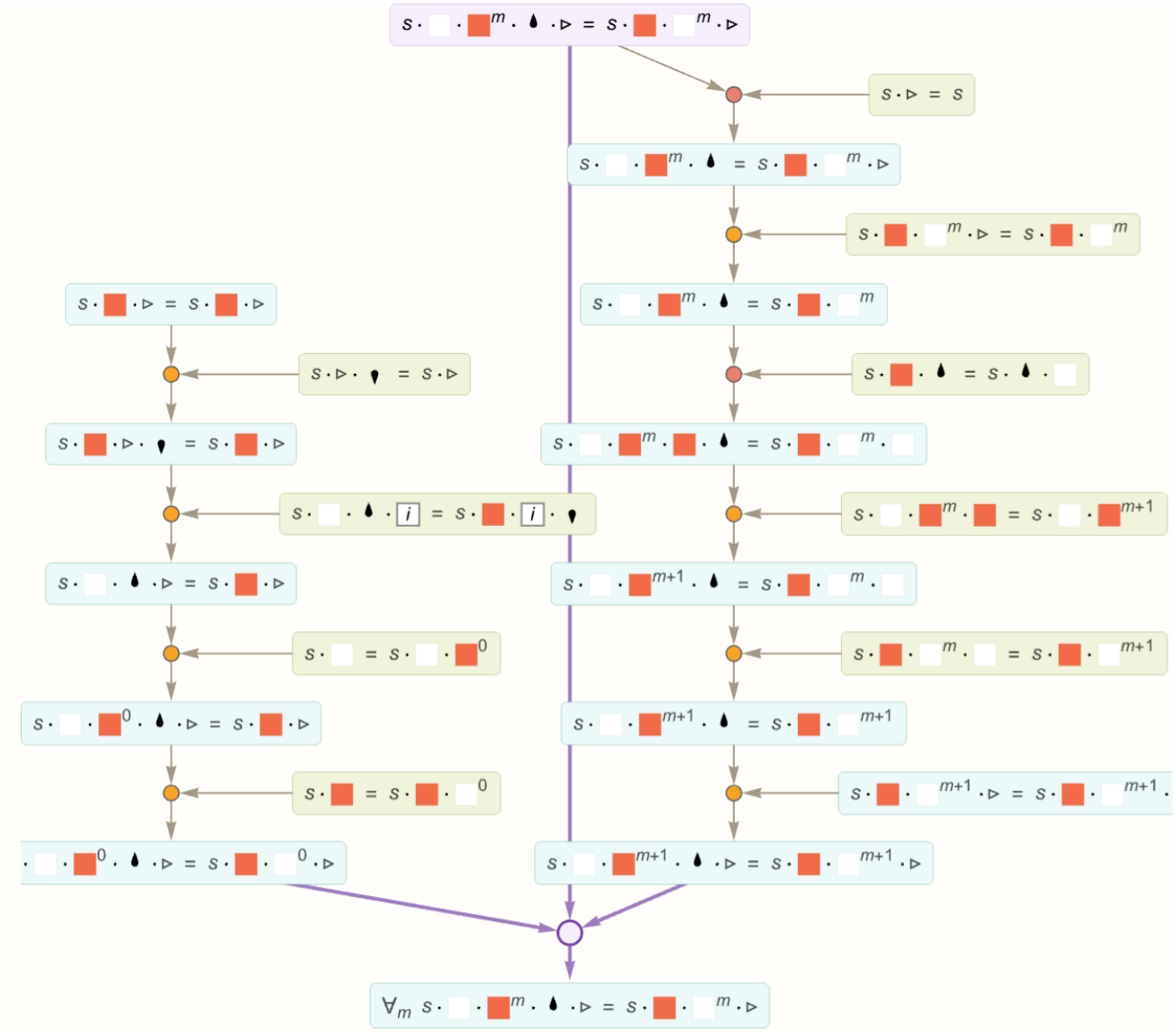
我们还需要公理来说明当头部到达右端（并且机器停止）时会发生什么：

然后我们需要公理来定义游程编码的工作原理：



好的，那么通过这个设置，我们必须证明什么来证明图灵机在给定 n 作为输入时总是计算 n + 1？分两种情况来说最简单。首先，如果 n 的二进制数字以 0 结尾，则机器最终必须将 0 替换为 1。其次，如果 n 的二进制数字以 m 1s 的序列结尾（上面表示为 $\blacksquare^m$），则机器必须将此序列替换为$\blacksquare \cdot \square^m$。但实际上这两种情况可以合并为一种，我们要证明的最终陈述就是（对于 $m \geqslant 0$）：

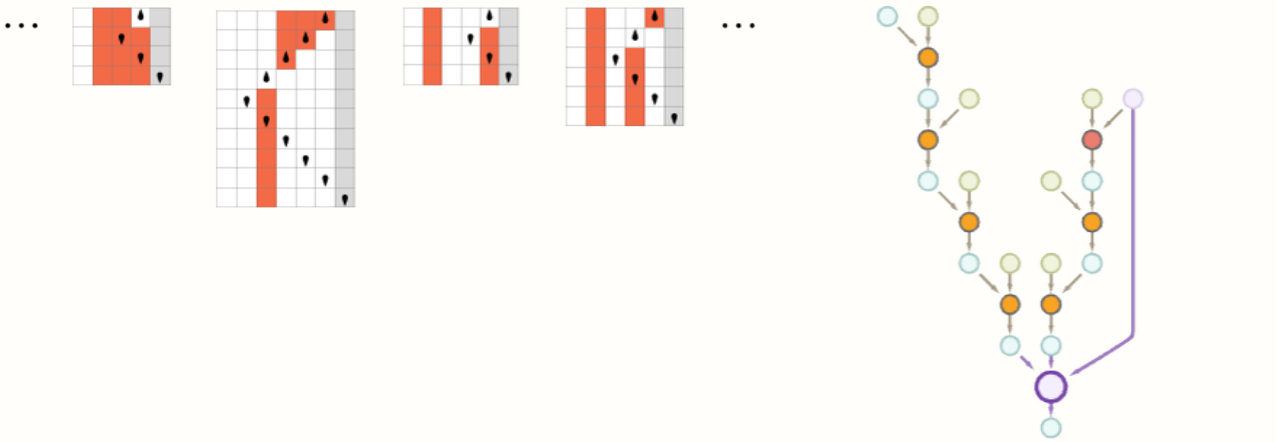
为了证明这个命题，我们需要找到一种方法，仅使用有效的符号变换，从我们的公理中生成它。我们基本上只需使用 Wolfram 语言 `FindEquationalProof` 即可做到这一点：

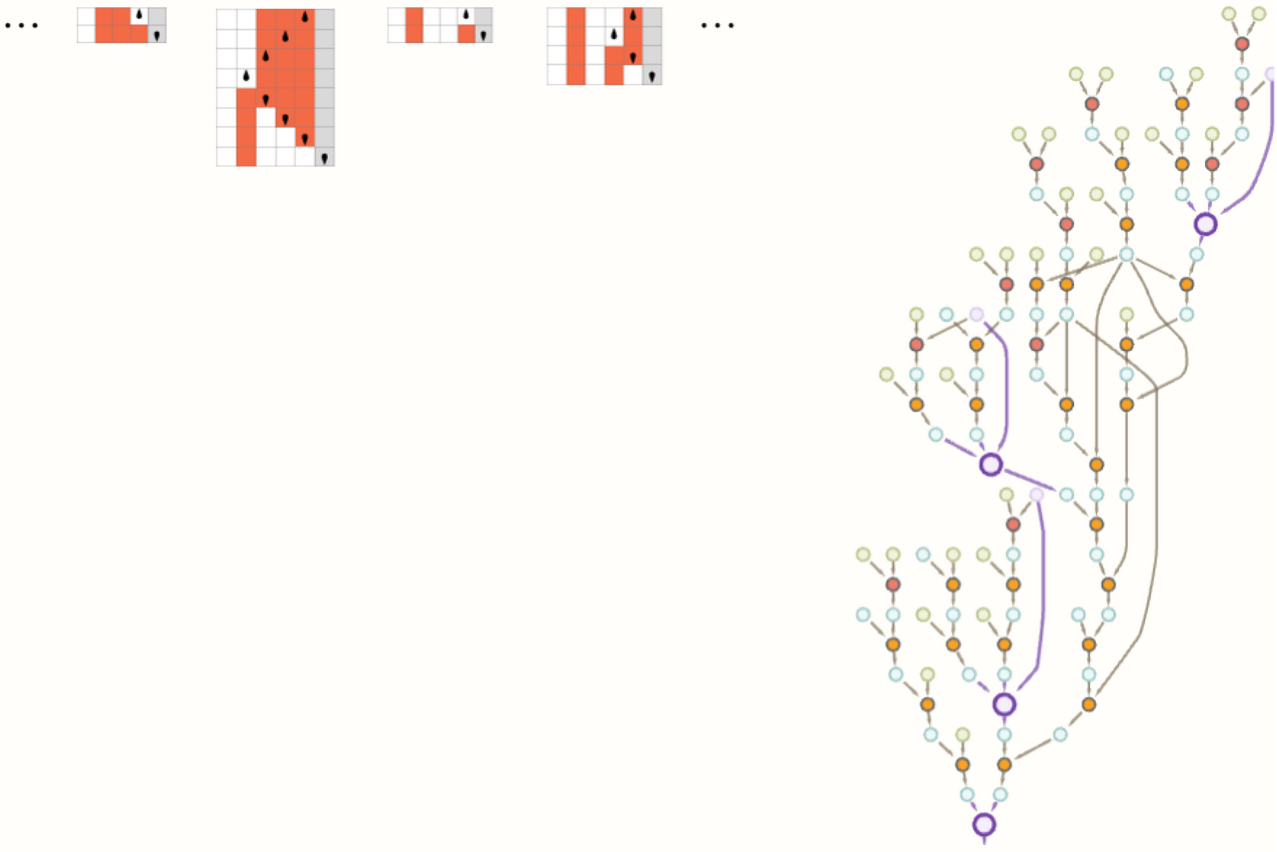


原始公理在这里用绿色表示；从它们导出的定理是蓝色的——我们试图证明的最终定理是作为底部的“输出”获得的。在大多数证明中，我们结构性地组合两个定理以导出另一个（通过替换 或双替换）。但还有一个额外且重要的部分：最后的 表示数学归纳法的应用。归纳法的输入是一个“基本情况”定理，我们可以将其称为 P_0（如左侧所示），以及一个断言 P_m 和一个定理 P_{m+1}，我们已通过右侧的证明证明了该定理。数学归纳法（我们可以将其视为广义公理）则表示，如果我们证明 P_m 蕴含 P_{m+1} 并且我们已经证明 P_0 那么就可以得出 P_m 成立对于所有整数m。

因此，这意味着从图灵机运行的公理开始，我们已经能够证明对于所有输入 n，机器都能正确计算 n + 1，而不需要 Bug。通过使用一切的符号表示，我们已经能够将关于所有输入n的本质上无限的语句集合减少为适用于所有n的有限符号证明。

在 2 状态图灵机中，有 17 台可以正确计算 n + 1，分为三类。我们刚刚研究的机器（机器 453）就是这些类别之一的示例。使用我们刚刚概述的方法，我们现在可以为所有这些类生成证明（以机器 453、445 和 1512 为例）：



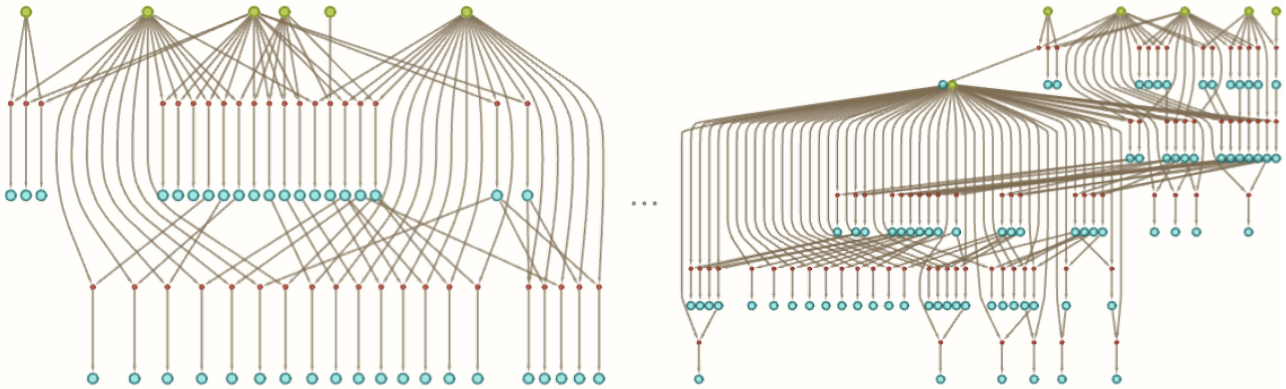


所以，是的，我们可以提供正式的证明，证明所有这些机器都能正确计算 n + 1，而不需要 Bug。这里的证明并不是非常复杂，但毕竟这些特定的机器的操作非常简单。即使对于没有 Bug 的三态机，证明仍然非常易于管理。但如果我们尝试更进一步，那么，正如我们稍后将讨论的，我们将不可避免地遇到任意长的证明。 （顺便说一句，如果我们尝试为上面讨论的元胞自动机程序提供证明，这似乎在任何视觉上不明显的情况下都会发生。）

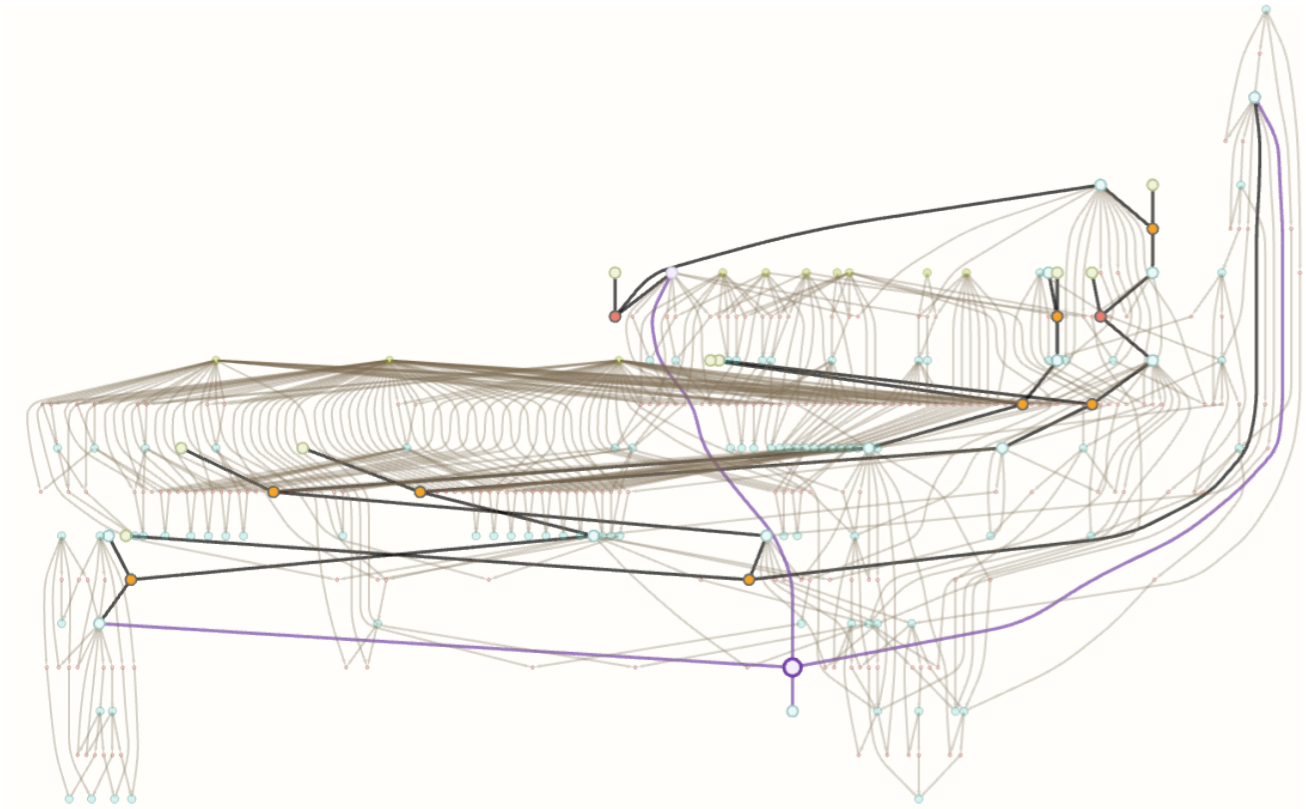
值得一提的是，我们展示的证明只是FindEquationalProof发现的特定证明。它们并不独特，也可能没有想象的那么简单。但它们仍然很好地表明了图灵机可以进行哪些类型的符号证明。

那么，如果我们尝试为具有 Bug 的机器生成证明，会发生什么？为了看到这一点，我们必须讨论如何构造像上面这样的证明。基本思想是从公理开始，然后尝试找到一种方法将它们串在一起以获得我们想要的定理。本质上，我们正在形成一个可能定理的多向图——然后我们的证明对应于该图中的类似路径的子图。

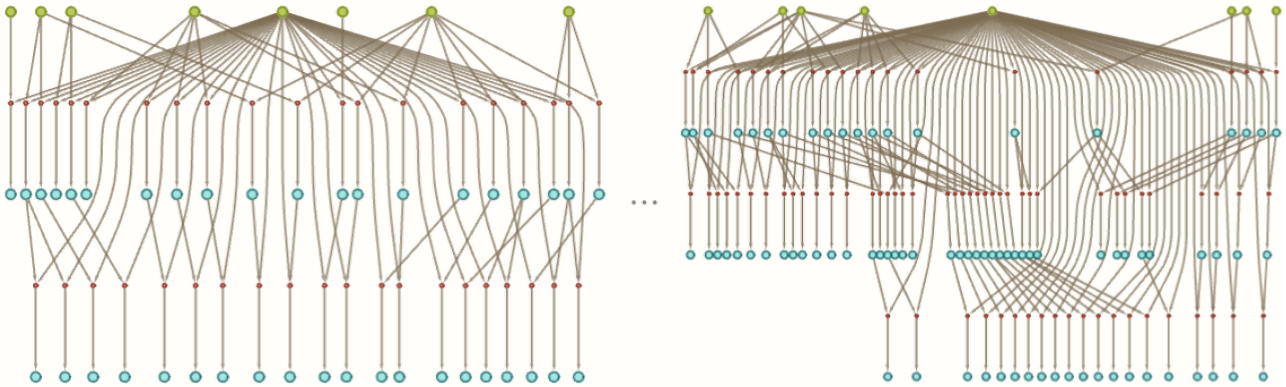
多路图是这样开始的——原始公理位于顶部：



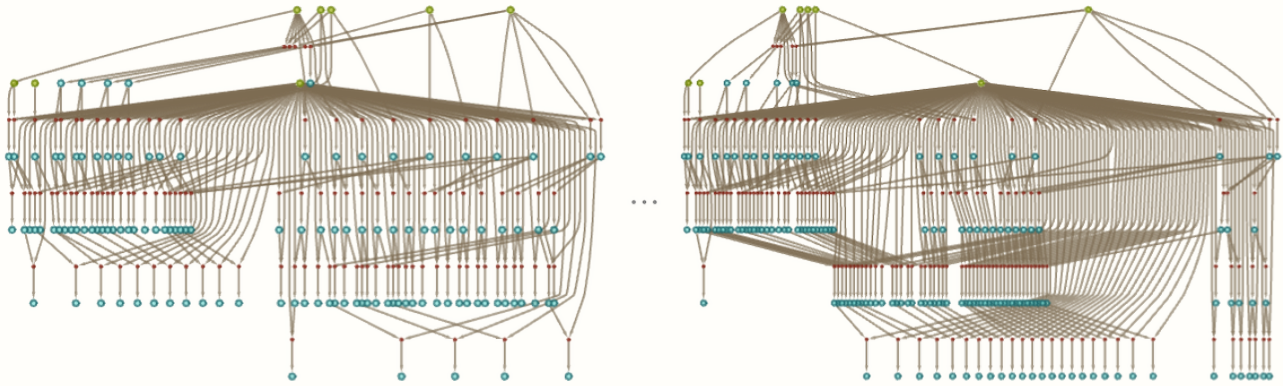
如果我们继续执行更多步骤（并在视觉上重新排列图表），我们发现我们可以从这个多路图中“找出”我们对上述情况的证明：



好的，那么带有 Bug 的机器怎么样——就像我们上面使用的第一个例子一样？该机器的多路图的开头看起来与之前的类似：



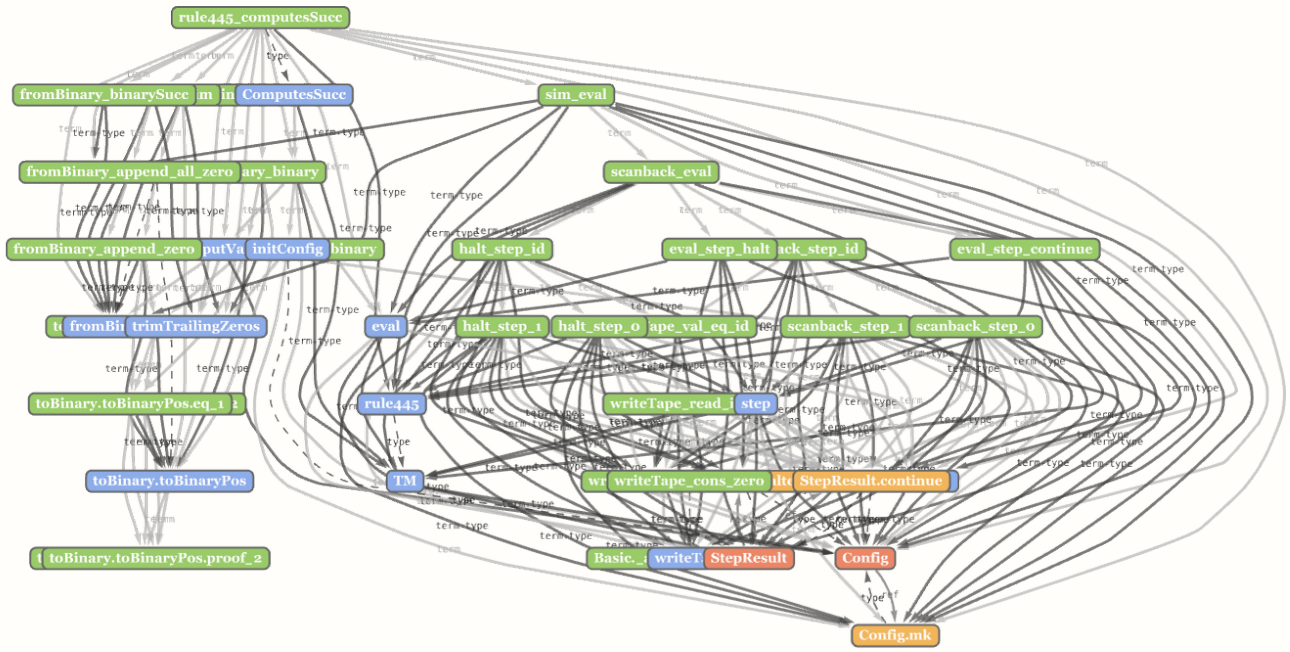
但现在——无论我们走多远——我们都找不到证据



因为事实上，我们试图证明的结果不是真的，所以没有证据。我们知道这一点，因为如果我们测试足够多的图灵机案例，我们可以明确地找到 Bug。

最终，我们通过尝试进行证明而不是仅仅运行图灵机并没有获得任何根本优势。在这两种情况下，我们都必须花费潜在的无限量的计算工作。这只是一个选择，要么直接将精力投入图灵机的枚举和运行案例，要么将精力投入构建多路图以尝试在其中找到证明。

我们可以认为我们迄今为止使用的证明风格是基于等式逻辑的，并且实际上通过结构替换（和归纳）从我们提供的公理中建立了我们想要的定理。另一种方法是使用类型论（如 Lean 等证明助手所做的那样），而不是建立类型的表示，然后通过给出具有该类型的事物的示例来证明该类型是可能的。在我们的例子中，我们想要证明的类型是“居住”基本上是一个定理的陈述，即对于所有n，我们的图灵机计算n + 1。使用能够通过调用证明助手来验证步骤的LLM，我们最终可以生成一个上面的机器 445 示例的证明。虽然它不是很有启发性，但我们可以给出证明的视觉表示：



计算不可约性与 Bug

> 我们现在已经看到了几个我们可以认为是“Bug”的最小示例：即使是非常简单的程序也可以做意想不到的事情。但这些“意外”的根本根源是什么？

最终它是计算不可约性 - 我们无法知道程序将做什么，除非通过显式运行它来有效。重要的是，计算不可约性不仅在原则上是可能的，而且是可行的。它（根据计算等价原理）在某种程度上无处不在，并且基本上影响任何行为并不明显简单的系统。

当计算不可约性存在时，意味着系统的行为通常只能通过不可压缩的计算工作量来预测。因此，这意味着，在计算工作量有限的情况下，系统总是有可能给我们带来惊喜，特别是系统意外地展示出我们认为是 Bug 的东西。

系统是否具有 Bug 在某种程度上是一个无限的问题。一般来说，您可能必须从所有无限数量的可能输入中检查其输出，并且您甚至可能不知道在任何特定情况下系统是否会停止并给出输出。事实上，一个世纪以来人们都知道，关于计算系统的此类无限问题原则上是不可判定的。但重点是，这不仅仅是一个原则问题，而且是一个问题。这是实践中的一个问题，它影响范围广泛，甚至非常简单的程序。

但是，有人可能会问，为什么不直接使用不存在这些问题的程序呢？您可以预见其行为的程序，并确保它们没有 Bug。好吧，问题是：如果你能完全预见一个程序会做什么，那么运行它还有什么意义呢？实际上，为什么不立即“写下答案”，而不必运行程序呢？换句话说，为了使程序值得实际运行，它的行为必须有一些你无法预见的部分。但计算等价原理有一个直觉：一旦有你无法预见的事情，往往就会有很多你无法预见的事情。换句话说，系统将倾向于充满计算不可约性。

从某种意义上说，计算不可约性使计算变得强大。但这也是它不可预测的原因。在实践中，无论是有意还是无意，程序中很常见的情况是“内部存在计算不可约性的火花”，并且以某种方式保持足够的控制来执行人们想要的操作。

也许程序做一些疯狂的事情是可以的——只要它至少满足一些属性（比如不进入无限循环）。也许人们希望程序具有某些特定的属性。 计算不可约性的一个重要特征是，无论何时存在，都必须有计算可约性的局部区域存在。换句话说，即使计算不可约性说人们无法预测系统将做什么的所有事情，但系统总会有某些事情是人们能够预测的。如果其中有一些与我们想要的属性相符，那么我们可能会知道系统将“不表现出 Bug”，至少在这些属性方面。

我们在上面的一些简单图灵机的例子中看到了如何构建符号证明，以便某些程序执行某些操作，从而不具有某些类型的 Bug。那么这与计算不可约性有什么关系呢？人们可以将证明视为利用计算可约性的局部区域来提供无限事物的有限摘要。但是，对于计算不可约系统所做的一切，没有（静态）有限的总结。随着人们试图讨论这一过程中越来越多的步骤，有关计算不可约过程的证明将不可避免地变得越来越长。

这可能是一个令人困惑的情况。研究一个程序并发现有关其运行的各种事实（“这个永远不会那样做”；“这个保持不变”；等等）。人们可能会认为，有了足够多的此类事实，就可以保证不会有 Bug。但如果有计算不可约性，实际上需要无数的事实才能做到这一点。在实践中，一个人所关注的某些特定性质可能会或可能不会根据其所确定的事实“保证无 Bug”。

测试案例够多了吗？规则学归纳的失效

> 科学归纳法是自然科学的基石。你看到世界上的某些事情以某种方式发生足够多次，并且你认为它总是会以这种方式发生（并且有一个关于它的“科学定律”）。但是规则学呢？我们可以在那里应用科学归纳法吗？你测试了很多图灵机。您观察到某种行为模式。图案会一直存在吗？科学归纳会说是的。但如果有“Bug”，那就不会了。关键是，在规则学中——由计算不可约性提供——总是有可能出现意外，而“Bug”则违反了我们可能推断出的任何法律。

特别是考虑到从我们现在所知来看，似乎底层的计算量很大，人们可能会想知道为什么科学归纳法首先会起作用。答案与自然科学和科学定律的基本性质有关。在某种程度上，科学试图做的是了解世界上发生的事情，并用我们的头脑可以理解的“叙述”来解释至少某些方面。关键是，这样的叙述——几乎按照定义——依赖于计算可约性的局部区域。下面可能涉及各种计算不可约性。但当我们想要“做科学”时，这并不是我们关注的重点。我们寻找体现计算可约性的规律，我们可以用我们的科学定律和叙述来“总结”这些规律。

实际上，这在某些地方比其他地方效果更好。物理学是一个巨大的成功故事。 生物学不是。在我们的物理项目中，我们可以看到有很多计算不可约性。只是与像我们这样的观察者相关的物理定律设法在很大程度上避免了它。在生物学中又出现了计算不可约性——事实上，它似乎是生命所必需的，能够完成它所做的事情。现在，如果我们想“深入了解”——以及例如做医学——我们就被迫面对计算不可约性。这表现为我们的实验中普遍存在的再现性失败，实际上是科学归纳的失败。 （人们可以尝试通过谈论概率而不是确定性来拯救科学归纳法，但最终人们总是不得不面对计算不可约性。）

但是程序呢？好吧，就程序执行的细节而言，人们将有可能遇到计算不可约性。而且，正如我们上面所讨论的，只有当我们所看到的只是一小片可约性区域，而不是考虑“整个行为范围”时，我们才会避免这种情况。

好吧，但是假设我们正在尝试通过查看大量特定案例来测试程序。我们如何知道我们是否“陷入了可还原性的困境”，或者实际上看到了整个“行为谱”，Bug 等等？实践中的典型方法是对可能的输入进行随机采样（或在已知输入周围引入随机“模糊”）。这行得通吗？好吧，这取决于“随机性”的“好坏”，以及 Bug 的分布和密度。

假设我们的“随机性”来自一种数值算法，其中只能出现某些类型的连续数字序列。我们正在测试的程序在这些情况下可能不会显示“Bug”，仅在其他情况下才会显示。一般来说，随机性和程序之间存在一种竞争。该程序的“计算能力是否足够强大”来“解码”随机性，并在这些情况下做一些特殊的事情？或者随机性是否会以至少不会系统地避免其“Bug”的方式对程序的行为进行采样？

当然，即使采样没有“系统地避免”Bug，它也可能不会击中它们，因为Bug只存在于可能输入的某些“特殊角落”。但如果随机性“计算上足够强”，就不会有任何“特殊角落”可以“隐藏”它。

尽管如此，考虑到存在的 Bug 的密度，样本数量可能还不足以达到 Bug。（如果假设存在一定的一致性和独立性，则可以使用简单的概率参数来计算，如果要采集更多样本，击中 Bug 的概率会如何增加。）

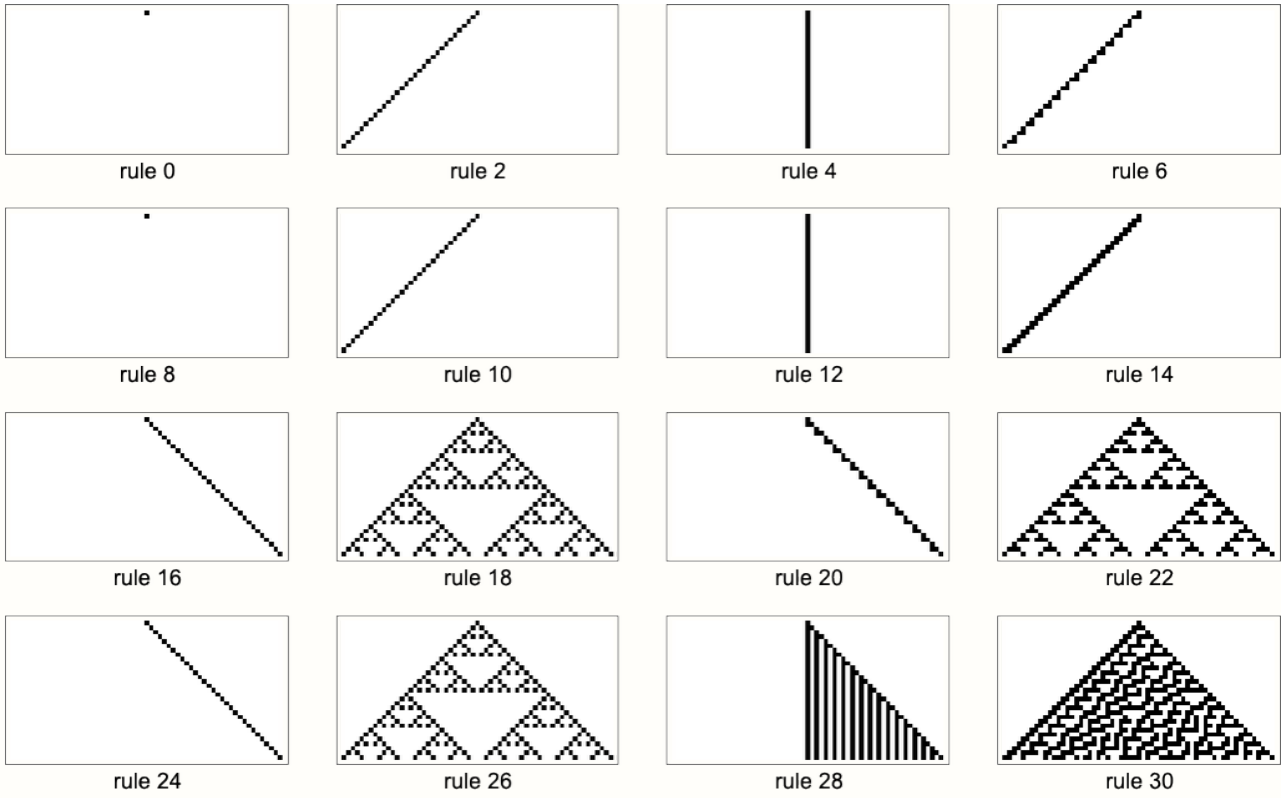
那么我们期望 Bug 的密度是多少？好吧，我们大概不会将某些东西称为“Bug”，除非它很罕见。但在程序的世界里，事情有多罕见呢？答案是：如人们所能想象的那样罕见。这很像“忙碌海狸”问题，即特定类的最长有限生命周期是多少的节目都可以。在这种情况下，有限的生命周期才是无限的。这里是“Bug”的概率很小。（一台带有“最难找到的 Bug”的图灵机，人们可能会异想天开地称之为狡猾的黄鼠狼机。）

但实际中会发生什么情况呢？嗯，至少在做规则学的过程中，我一贯的经验是，罕见的现象总是会出现。对某种类型的系统的许多实例进行采样并看到一些一致的行为是非常常见的。然后有一种相当大的倾向，即使用科学归纳来得出这种行为必定会发生的结论。但按照规律，如果采样更多，特别是采样详尽的话，就会产生惊喜。有时需要查看数千个、有时数百万个、有时数十亿个、有时数万亿个案例。但随后人们会看到一些意想不到的东西，相对于人们对系统行为的看法，人们可以将其视为“Bug”。

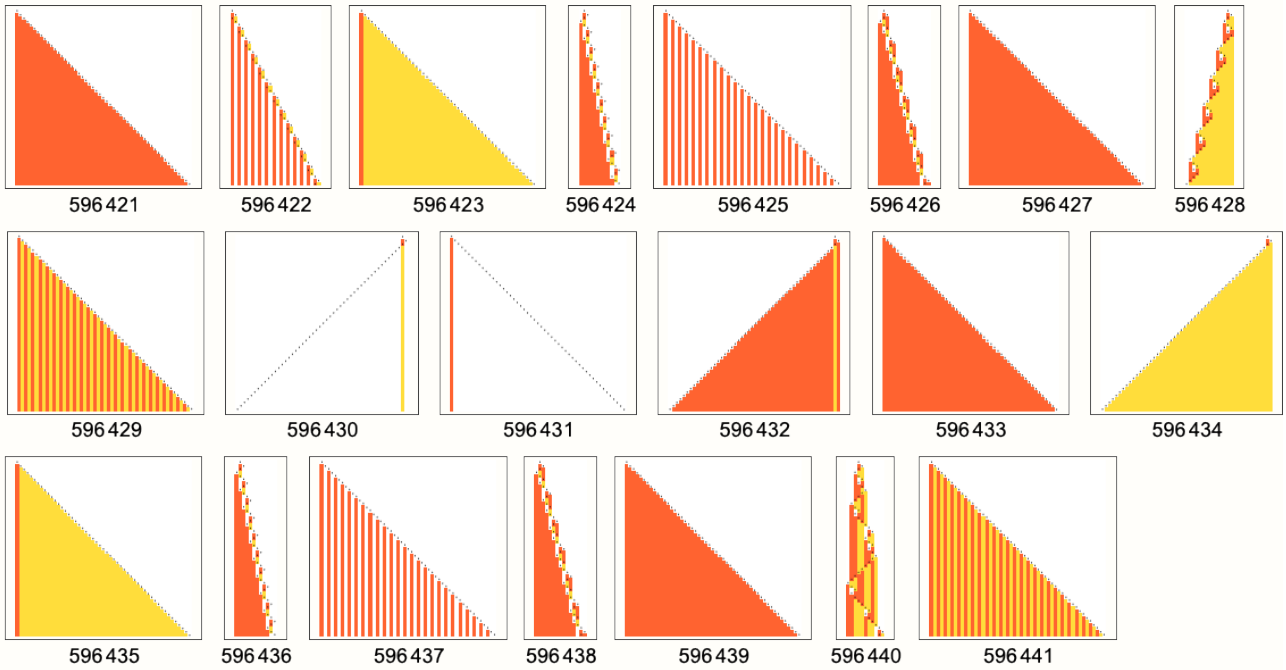
一些典型的规则学意外

> 计算宇宙充满了惊喜。在我用规则学探索它的（近）半个世纪中，我见过很多这样的东西。事实上，根据我的经验，执行规则学时很少会没有惊喜。从某些方面来说，这是一次令人谦卑的经历，同时也是一次鼓舞人心的经历。你想象你知道事情将如何运作。然后意想不到的事情发生了。你会意识到计算宇宙比你想象的要丰富得多。

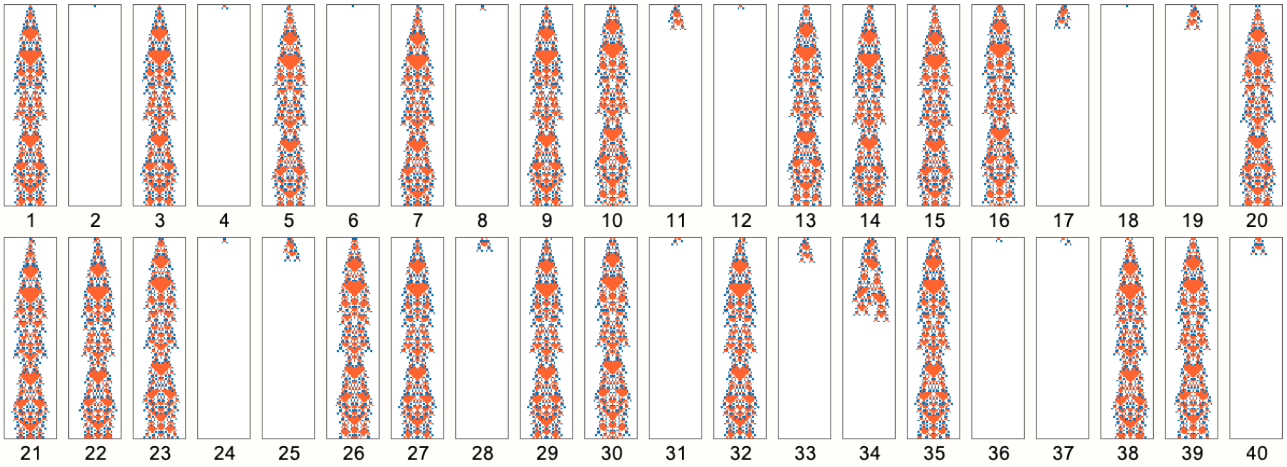
我非常早期且非常重要的规则学中的意外现象之一与最简单的元胞自动机有关。列举可能的（静态）规则，它们似乎都在做非常有规律的事情。但出乎意料的是，出现了规则 30，其复杂性如下：



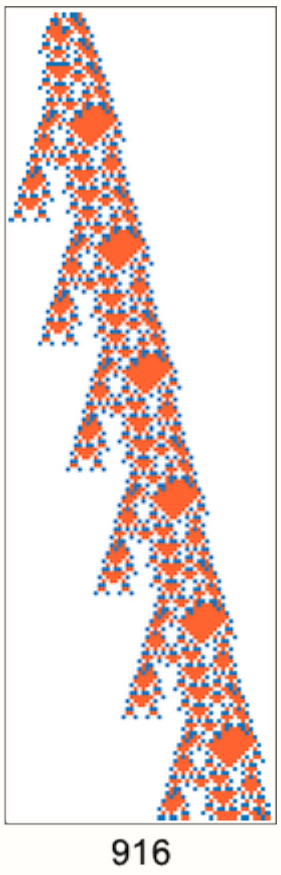
对于图灵机来说，这是一个类似的故事——尽管人们必须走得更远才能发现一个巨大的惊喜。现在，（2 状态，3 颜色）机器 596440 终于做了一些令人惊讶的和复杂的事情：



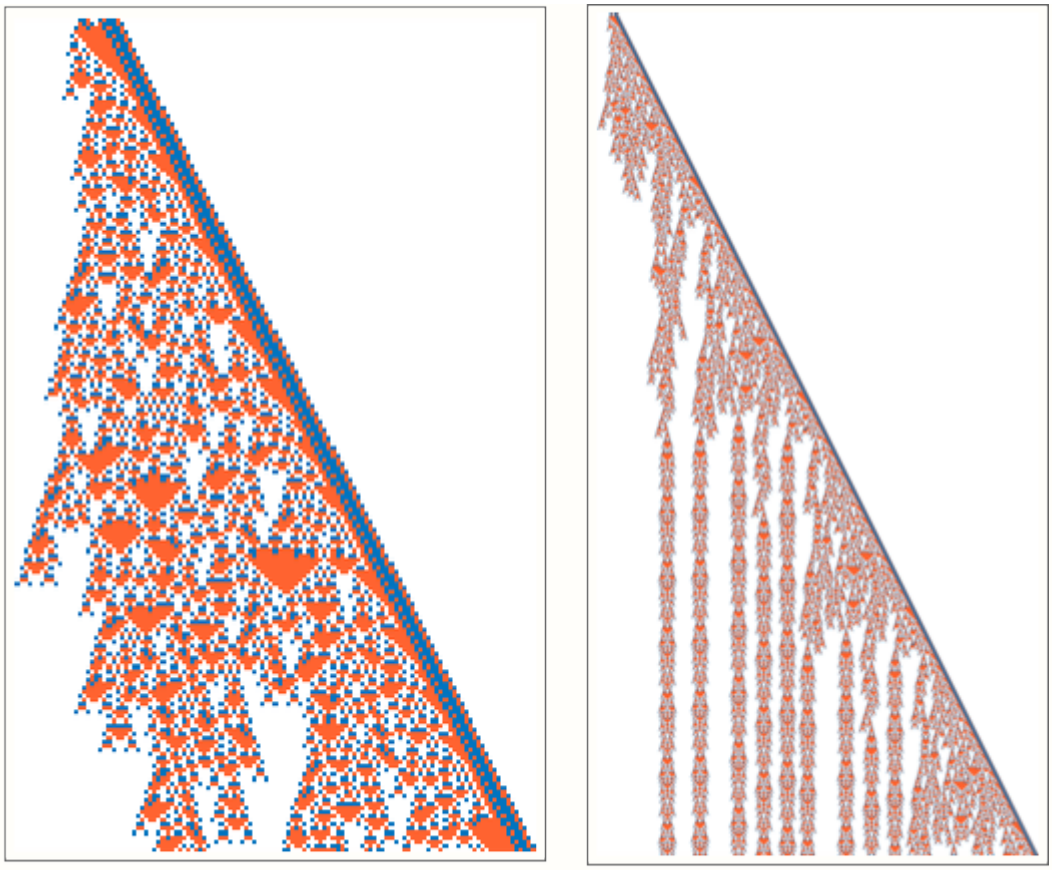
仅仅改变“输入”就会带来惊喜（就像我们上面之前的讨论）？例如，考虑3 色极权代码 1329 元胞自动机。对于许多输入（即初始条件），它会执行一小部分操作：



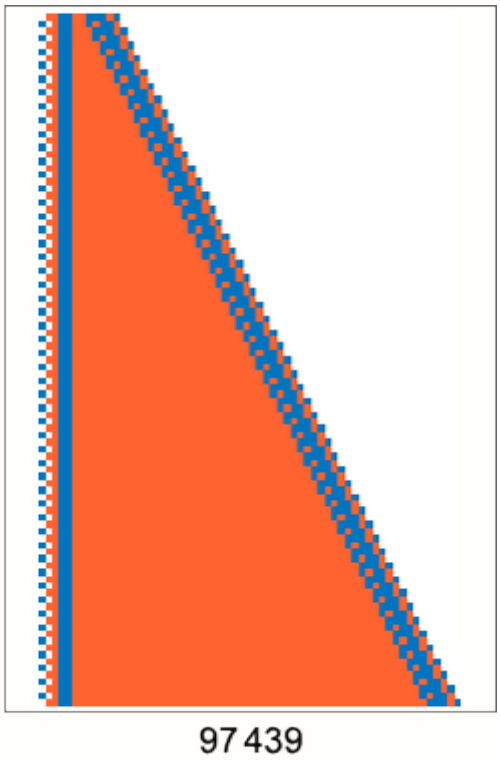
但在初始条件 916 时，突然发生了一些不同的情况：



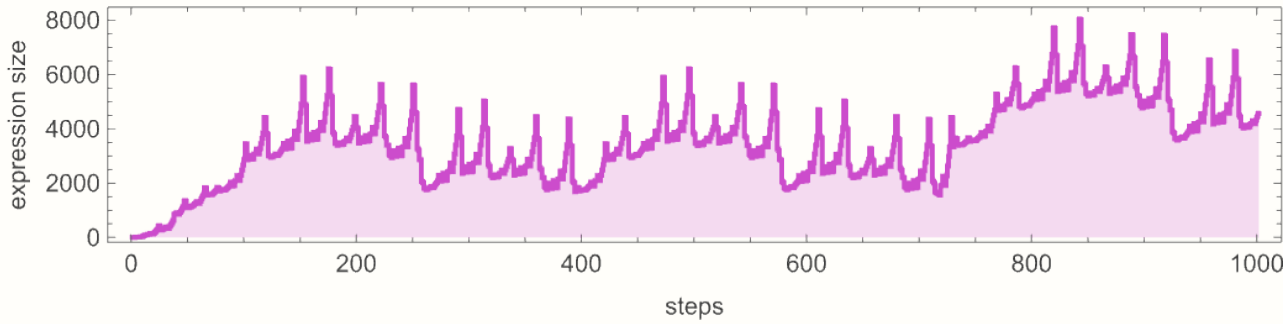
继续往前走，还有另一个惊喜——一个越来越大的模式：



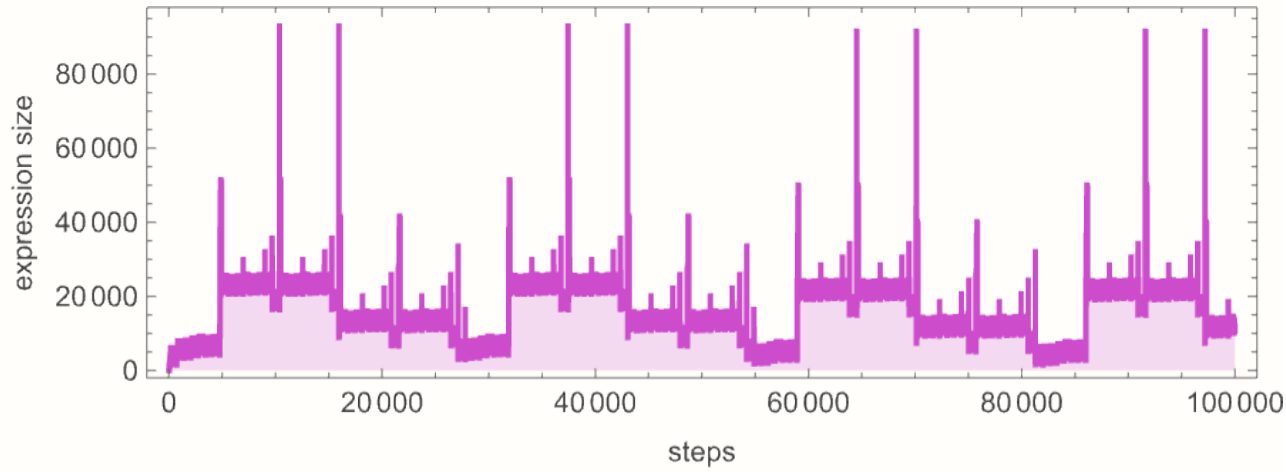
然后，突然间，输入 97,439，出现了人们永远不会猜到这条规则可以做的事情：



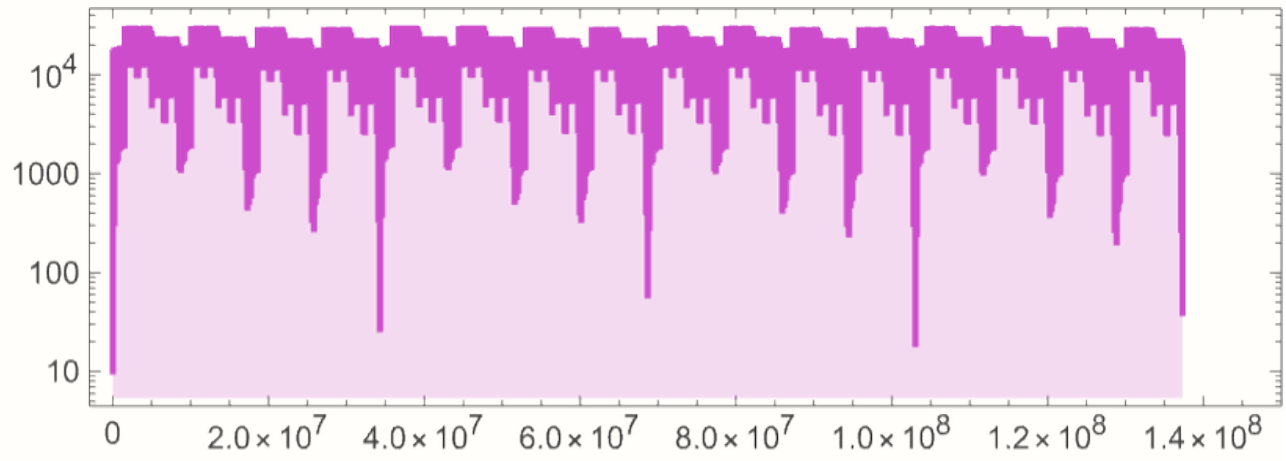
观察某个系统并想知道如果运行足够长的时间它“最终会做什么”是很常见的。例如，它会停止，还是永远继续下去？ 计算不可约性的现象表明，原则上它可能很难判断，但值得注意的是，在实践中，这种情况经常很难判断。作为一个有点极端的例子，请考虑组合器表达式 s[s[s]][s][s][s][s][s][k[k]]] 的标准计算。1000 步之后，它似乎继续产生越来越大的输出：



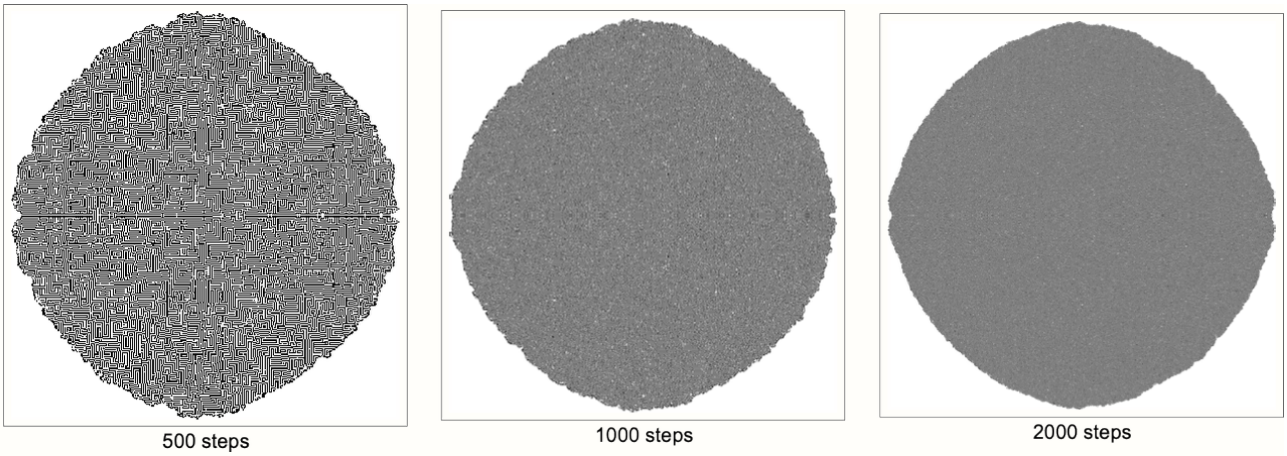
在 100,000 步时，我们可以看到物体在弹跳，但没有证据表明任何物体会停止：



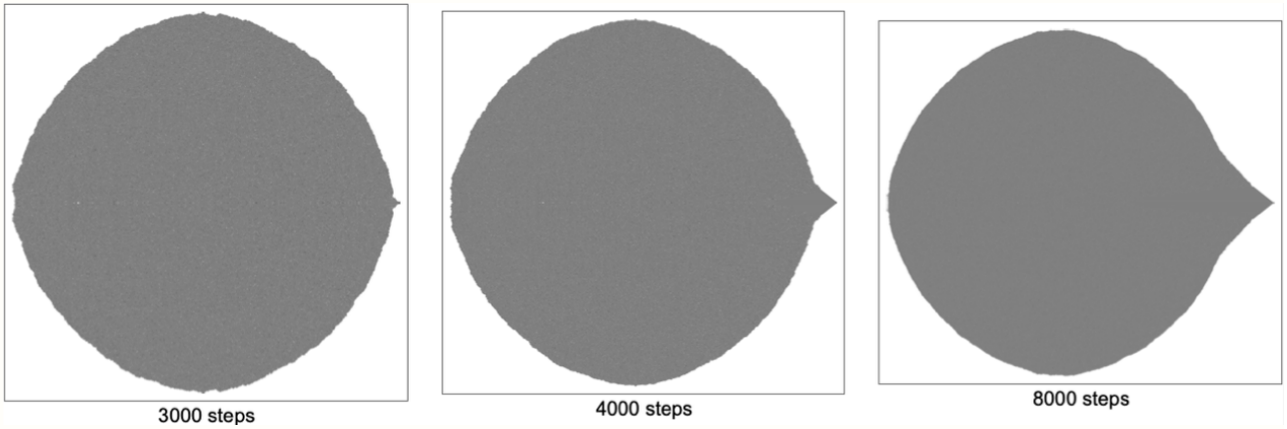
但是，突然之间，在执行了至少 137,356,329 步之后，它实际上停止了：



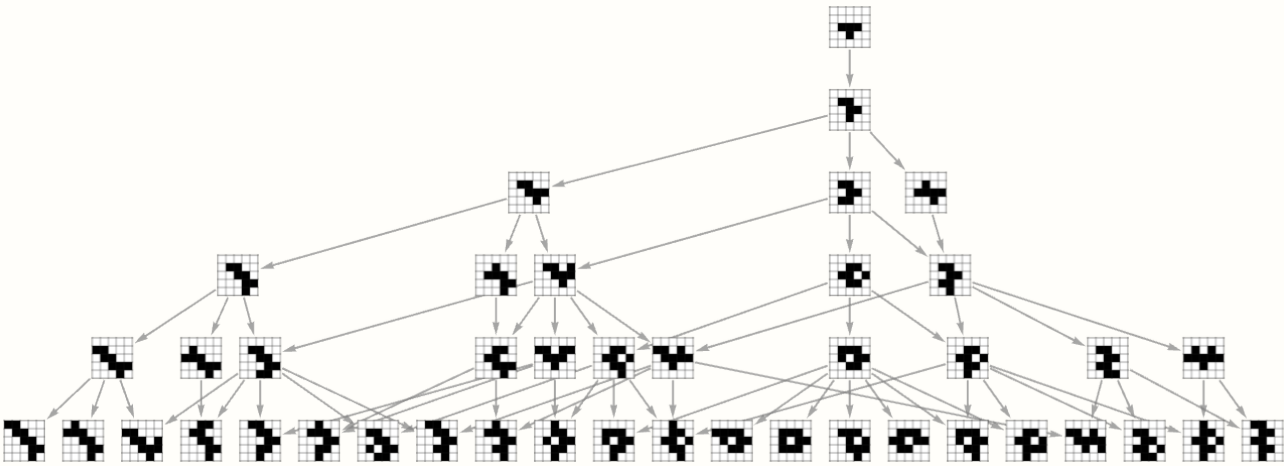
这是一个稍微不同的例子。如果您运行 2D 元胞自动机（外部极权 9 邻域代码 746，从  开始），它似乎会产生大致圆形的图案（尽管其各向异性约为 4%）



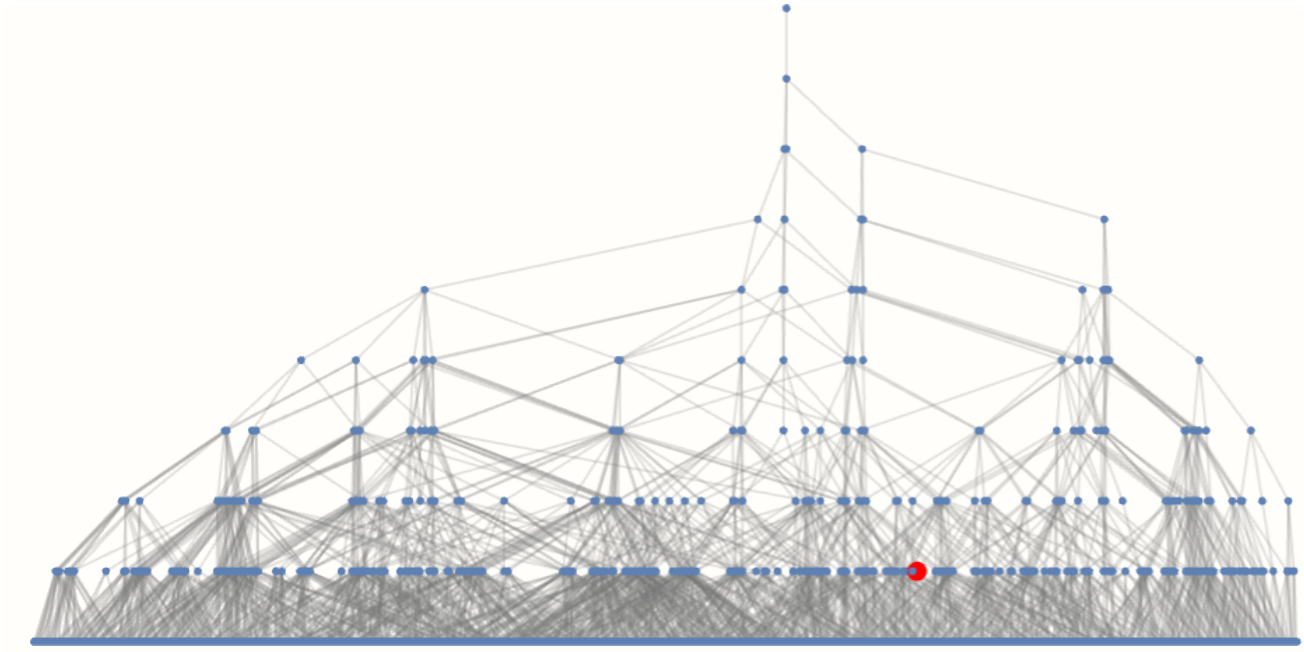
但突然间，在大约 3000 步之后，图案长出了一个“喙”，打破了循环：



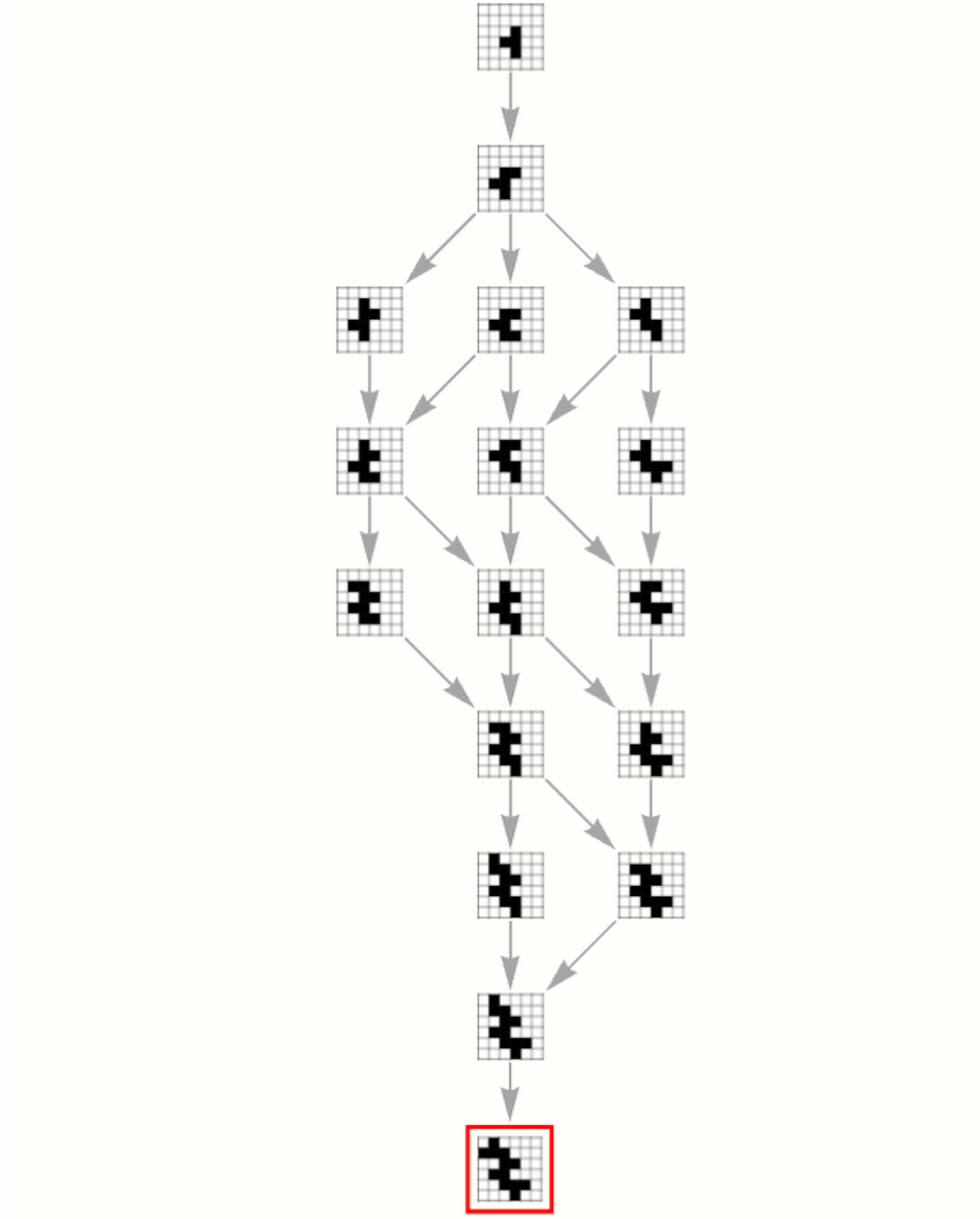
作为另一种示例，考虑一个有多种可能移动的系统，可以用多路图来表示。在此，我们将以所有可能的方式连续添加一个黑色单元格，限制条件是该单元格始终必须有正好 2 个邻居（共 8 个）已经是黑色的。这种系统的“可能历史”多路图中的前几个步骤是：



从这张图中，人们可能会认为增长将永远持续下去。但继续研究多路图，会出现一个令人惊讶的情况：经过 9 个步骤后生成的特定簇根本无法生长：



这有点像拼图或游戏中可能发生的情况：有一组薄弱的动作可以导致一个特殊的结果 - 这里是一个暂停状态：



从某种意义上说，规则学是一个“Bug”是规则而不是例外的领域。事实上，计算不可约性在计算宇宙中的普遍存在使得它基本上不可避免地会出现超出我们预期的“惊喜”，我们可以称之为“Bug”。

数学中的“Bug”

> 如果“Bug”在规则学中如此常见，那么在数学中又如何呢？假设一个人正在研究某个数学系统，并且发现了很多以某种方式工作的例子。人们可能会做出一种假设，认为事情总是会这样进行。也许他们真的这么做了。或者，也许在某个地方有一个反例——实际上是一个假设中的 Bug。

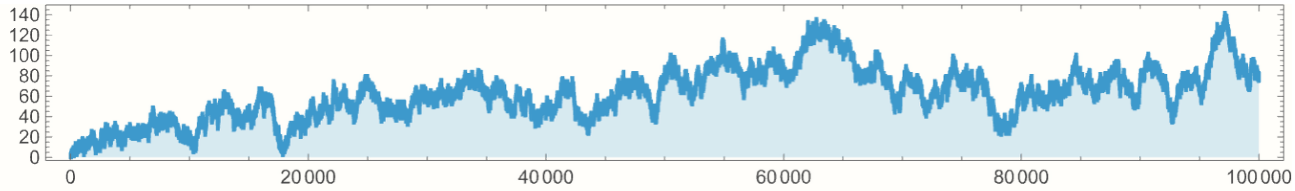
假设您正在查看 $x^n - 1$ 形式的多项式因子：

$x-1$	$x-1$
x^2-1	$(x-1)(x+1)$
x^3-1	$(x-1)(x^2+x+1)$
x^4-1	$(x-1)(x+1)(x^2+1)$
x^5-1	$(x-1)(x^4+x^3+x^2+x+1)$
x^6-1	$(x-1)(x+1)(x^2-x+1)(x^2+x+1)$
x^7-1	$(x-1)(x^6+x^5+x^4+x^3+x^2+x+1)$
x^8-1	$(x-1)(x+1)(x^2+1)(x^4+1)$
x^9-1	$(x-1)(x^2+x+1)(x^6+x^3+1)$
$x^{10}-1$	$(x-1)(x+1)(x^4-x^3+x^2-x+1)(x^4+x^3+x^2+x+1)$

你尝试很多情况。它们似乎都给出了系数±1的结果。但随后你尝试$x^{105}-1$。突然出现了一些不同的情况——系数为2：

$x^{105}-1$	$(x-1)(x^2+x+1)(x^4+x^3+x^2+x+1)(x^6+x^5+x^4+x^3+x^2+x+1)$ $(x^8-x^7+x^5-x^4+x^3-x+1)(x^{12}-x^{11}+x^9-x^8+x^6-x^4+x^3-x+1)$ $(x^{24}-x^{23}+x^{19}-x^{18}+x^{17}-x^{16}+x^{14}-x^{13}+x^{12}-x^{11}+x^{10}-x^8+x^7-x^6+x^5-x+1)$ $(x^{48}+x^{47}+x^{46}-x^{43}-x^{42}-2x^{41}-x^{40}-x^{39}+x^{36}+x^{35}+x^{34}+x^{33}+x^{32}+x^{31}-x^{28}-x^{26}-$ $x^{24}-x^{22}-x^{20}+x^{17}+x^{16}+x^{15}+x^{14}+x^{13}+x^{12}-x^9-x^8-2x^7-x^6-x^5+x^2+x+1)$

这是另一个例子。查看 3m + 2 和 3m + 1 形式的素数数量的累积差异。绘制前 100,000 个素数的结果，看起来这个差异将始终为正：



但事情并没有这样发展。在第 23338590792 个素数处，它翻转并变为负值。

类似的结果有很多，特别是涉及素数、数论等，事实上，对于其中一些结果来说，人们必须走得很远才能找到第一个已知的“Bug”。

但不知怎的，在纯数学的主流中，这样的问题并不常见。从某种意义上说，这似乎反映了我们称之为数学的活动的基本性质。

由哥德尔定理可知，某些问题——甚至只是关于整数算术方程的问题——可以等价于任意计算，因此原则上会遭遇我们在规则学中看到的一切现象。但正如我在其他地方详细讨论的那样，通常所实践的纯数学往往聚焦于更具“人类尺度”的问题；这些问题实际上位于计算可约性的局部区域，从而避开了规则学中普遍存在的计算不可约性。

事实上，正是这一点使得我们有理由期望我们感兴趣的数学可以通过可管理长度的证明来确定。如果我们达到了拥有完全形式化的符号证明的程度，那么它通常很容易检查。当然，就像程序验证一样，从我们认为我们要说的内容到形式化的链接可以再次具有 Bug。然而，在纯数学的实际实践中，这些事情并不像人们想象的那么重要。因为纯数学的核心并不在于证明的细节，而在于创建为人类数学理解提供框架的抽象结构。

但数学并不总是遇到规则学式的 Bug 到底意味着什么呢？从某种意义上说，它直接反映了纯数学发展的每个阶段都强调人类理解这一事实。如果目标是只是为了找出数学中所考虑的结构的真实情况，那么人们可以采用一种不同且强大的方法：实验数学。事实上，严肃的实验数学在方法论上就像规则学一样，尽管是在数学结构上运行，而不仅仅是纯粹的规则和程序。而且，就像规则学一样，实验数学直接接触到计算不可约性——以及它带来的所有惊喜和“Bug”。

实践中的 Bug

> 假设您要创建一个没有 Bug 的程序。你应该怎么做呢？从根本上讲，确保程序始终执行您想要的操作的唯一方法是了解它可以执行的所有操作。但其中隐含着一种悖论：如果你能真正理解你的程序可以做的所有事情，那么基本上意味着该程序正在做一些计算上可简化的事情，并且你最终可能实际上不需要运行该程序的所有步骤来获得你想要的结果。

但是，好吧，在某种程度上，您将不得不掩盖程序正在执行的某些操作。那么什么给你最好的机会让程序做你想做的事呢？我认为，语言设计是关键。因为语言设计就是提供能够满足人类需求并且能够理解的原语。

一个人可以编写无数的程序。每个人都会做一些事情。但他们所做的事情极有可能不是我们人类想要的。设计良好的语言所做的就是提供人类通常真正想要的计算工作的原语块。从某种意义上说，它们定义了人类思想成功计算形式化的框架。

这意味着，如果您用一种设计得足够好的语言编写程序，那么该语言本身的结构将引导您使您的程序成为您想要的程序，而不是您认为的 Bug。好吧，碰巧我在过去几十年里花费了大部分时间试图使Wolfram 语言成为实现这些目标的语言。

事实上，在 Wolfram 语言中，可以用非常小、简单且“可理解”的程序来表达最终非常复杂的任务。实际上，所发生的情况是，大部分计算不可约性已被打包到语言的原语中，因此剩下的部分是我们人类可以访问的部分，并且映射到我们的自然思维模式。

我已经一遍又一遍地看到它：在 Wolfram 语言中，短程序可以做您想做的事，而不是您认为的 Bug。如果程序中有额外的复杂性，它可能会引入您会考虑的 Bug。换句话说，在所有可能的程序空间中，我们人类倾向于想要的是那些可以用 Wolfram 语言表达为短程序的程序。

但是，无论你的语言设计得多么好，任何时候你编写一个实际需要编写的程序（从某种意义上说，它至少实现了一些不可约的计算），它都有可能做一些你没有预料到或预见到的事情——你可能会考虑使用 Bug。

那么你能做什么呢？我一生的经验是，在实践中，最重要的是尝试以尽可能完整但人性化的方式“了解”程序正在做什么。在上面的元胞自动机示例中，可视化整个模式至关重要。事实上，在许多情况下，基本的人类视觉感知足以让人完全相信系统总是“做正确的事”。基于这种“视觉理解”，人们无疑可以（通常需要付出一些努力）构建一个正式的“正确性定理”。但纯粹从系统的描述来做到这一点——而不首先“实际了解它的内部功能”——会遇到计算不可约性的问题，而且通常会困难得多。

但一般来说，人们如何“了解”计算的内部情况呢？一维元胞自动机是一个特别简单的例子，其中计算过程在“时空阵列”中进行，可以立即呈现为我们视觉感知能力可访问的图像。一维图灵机或多或少是相同的方式，特别是如果它（小心，以免丢失重要信息）压缩其时空行为。对于没有这种直接“时空”表示的其他类型的系统，事情可能会更加复杂，并且有一定的艺术可以很好地可视化此类系统的行为。

在传统软件中，事情可能更加复杂。从正在运行的程序内部可视化详细的遥测数据通常可以揭示真相。但往往没有真正的方法来“单独提取程序的各个部分”——因为变量值等中编码了复杂的共享状态。这是 Wolfram 语言中深刻体现的符号范式的强大优势之一。因为在这种范式中，即使是很小的程序片段也可以自行运行，这至少为人们提供了“详细了解程序正在做什么”的原材料。

当程序有 Bug 时，要问的一个问题是“那些 Bug 是如何进入那里的？”在实践中，它们常常是简单的人类弱点（相差一的错误、未能看到某些依赖性等）的结果。有时，诸如类型检查之类的事情会将事情限制在至少可以立即捕获最明显的错误的程度（尽管以程序员的灵活性较低为代价）。现代人工智能通常非常擅长找到遵循人类典型失误模式的 Bug。

然而，我们在这里主要讨论的是更深层次的 Bug。它们是 Bug，不遵循任何与人类相关的模式。从某种意义上说，它们是 Bug，源于计算本身的本质。是的，实际上它们可以由人类或人工智能程序员放置在那里。但即使程序是通过详尽搜索构建的，或者适应性进化——具有一些有限的约束集。（人们可能会想象，在这种情况下，我们可以将“约束”作为“正确性证明”。但正如我们上面所讨论的，能够构造一个完整的这样的证明本质上与程序实现任何“计算上重要”的东西是不兼容的。）

虽然人们可能不认为它们是传统的“程序”，但人们可能会对现代人工智能和神经网络等通过示例进行训练的东西感到好奇。那里的期望肯定要低得多：人们不期望具体的、精确的（甚至经常是可重复的）输出——尽管特别是当存在迭代改进时，人们可能会达到约束可以（至少很有可能）完全满足的程度。

但最终（有点像受过训练的野生动物的情况）我认为人们不能指望能够确定任何事情。是的，机器学习的过程可以以一种似乎满足定义的约束和训练目标的方式将“不可约计算块”组合在一起。所构建的很有可能不是一个“可理解的机制”。不可避免地，计算不可约性的某些部分总有可能“冒出来”，并且系统会以您意想不到的方式运行。现实神经网络中实际发生的情况将很难可视化或“在人类层面上理解”。事实上，即使在我们研究过的非常简单的系统中，所发生的事情的核心基本上也与我们在这里看到的一样。

但是如果我们允许一些 Bug 呢？如果我们的程序是用一种原语非常适合我们想要计算的语言编写的，那么允许 Bug 不太可能让程序变得更短。但是，如果我们的程序是在较低级别编写的，例如图灵机，那么我们可以预期（正如我们在上面看到的）允许一些 Bug 可以让程序更短。在这种情况下，我们可以认为自己正在尝试“利用原始计算”，并且找到大致适合我们想要做的事情的“计算块”往往比找到完全适合我们要做的事情更容易。

假设我们正在运行一个我们认为将以某种方式运行的程序。但事实并非如此。相反，它具有我们可以认为的 Bug。好吧，如果我们尝试进行工程设计，我们会认为这是失败的。但如果我们从事科学研究，我们可能会认为它是成功的。因为实际上我们发现了一些我们没有预料到的东西。事实上，我们可以认为规则学的大部分进展正是关于“Bug”长序列的发现——或者至少Bug 相对于我们关于事物如何运作的正常直觉。但现在，根据我们从规则学中学到的知识，我们可以扭转局面，开始了解 Bug 普遍存在的基本现象的一些基础，无论它们在哪里。

致谢

我要感谢 Wolfram 研究所的几位成员提供的帮助：Richard Assar、Júlia Campolim、Nik Murzin 和 Willem Nielsen。我还要感谢 Wolfram 软件质量保证团队，他们在过去四十年中处理了超过 50 万个 Bug（其中绝大多数现已修复！），他们的工作帮助我培养了对 Bug 现象的直觉。还要感谢多年来在规则学上与我合作的所有人员，以及见证了我对规则学中的意外现象的反复反应的人们：“我们必须认识到计算生物比我们更聪明！”

原文：Stephen Wolfram, “Towards a Theory of Bugs: The Ruliology of the Unexpected”, 2026。