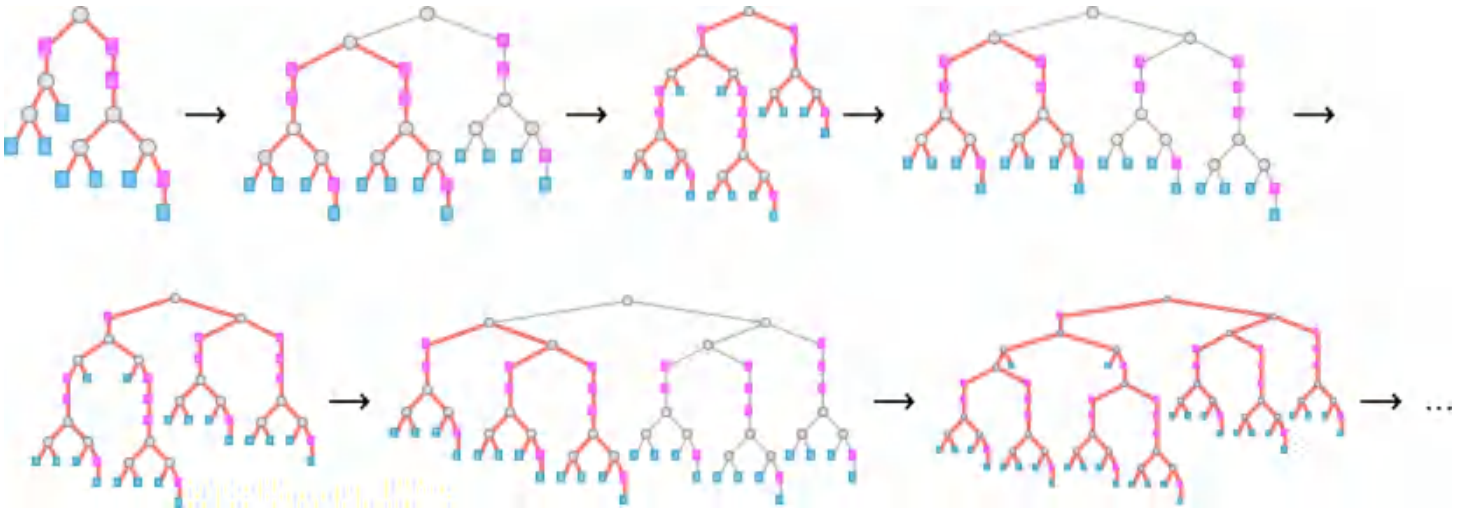
Lambda 的规则学

September 15, 2025



点击任何图表以获取Wolfram 语言代码来重现它。

什么是 Lambda?

这是一个纯粹的、抽象的计算故事。事实上，从历史上看，它是其中最早之一。但尽管我本人已经近半个世纪以来在实践中使用过它，但在我的多年探索简单计算系统和规则学过程中，我从未专门研究过它。是的，它涉及一些繁琐的技术细节。但最终会证明，lambda——就像我所探索的许多系统一样——具有丰富的规则学，这尤其体现在它们与实际计算的联系上。

在 Wolfram 语言中，它是 Function 函数。早在 20 世纪 30 年代阿隆佐·丘奇首次讨论它时，他就称其为λ（lambda）。这个想法是拥有一个能作为“ 纯函数 ”——可以应用于一个参数来给出一个值的东西。例如，在 Wolfram 语言中，可能会有：

```
In[1]:= Function[x, 1 + x]
Out[1]= Function[x, 1 + x]
```

单独来看，它只是一个符号表达式，其值就是自身。但如果我们对它应用一个参数，那么它会将这个参数代入其主体部分，然后计算该主体部分：

```
In[2]:= Function[x, 1 + x][8]
Out[2]= 9
```

在 Wolfram 语言中，我们也可以这样写：

```
In[3]:= (x ↦ 1 + x)[8]
Out[3]= 9
```

或者，使用 λ = Function ，我们可以写成

```
In[4]:= λ[x, 1 + x][8]
Out[4]= 9
```

这接近于丘奇原始的符号表示法（λ*x* .(1 + *x*)) 8。

但如果我们想“更加纯粹”，实际上只用 lambda 来“处理所有事情”——不使用任何预定义的函数，比如 Plus （ + ）呢？要设置 lambda，我们必须要有函数应用的概念（比如 f[x] 表示“将 f 应用于 x ”）。因此，例如，这里有一个 lambda（“纯函数”），其结构仅由函数应用定义：

```
In[5]:= λ[x, x[x[x][x]]][a]
Out[5]= a[a[a][a]]
```

我们能像这样进行熟悉的操作吗？让我们想象有一个符号 z 表示 0，另一个符号 s 表示计算后继的操作。那么，这里就是 0 到 5 的整数的表示：

```
{z, s[z], s[s[z]], s[s[s[z]]], s[s[s[s[z]]]], s[s[s[s[s[z]]]]]}
```

但让我们以 s[s[s[s[z]]]] 为例。我们可以将这个表达式中的 s 和 z “提取出来”，仅用 lambda 来表示其“纯粹结构”部分：

```
In[•]:= λ[a, λ[b, a[a[a[a[b]]]]]][s][z]
Out[•]= s[s[s[s[z]]]]
```

但在某种意义上，我们这里不需要 s 和 z；我们可以完全用 lambda 来设置整数的表示，比如作为（通常被称为“Church numerals”）：

```
{λ[a, λ[b, b]], λ[a, λ[b, a[b]]], λ[a, λ[b, a[a[b]]]], λ[a, λ[b, a[a[a[b]]]]],
 λ[a, λ[b, a[a[a[a[b]]]]]], λ[a, λ[b, a[a[a[a[a[b]]]]]]]}
```

这一切都非常纯粹——而且抽象。至少在最初，它看起来非常简洁。但很快就会遇到一个严重的问题。如果我们写

```
λ[x, λ[x, x[x]]]
```

这意味着什么？在 x[x] 中， x 是与内部 λ 中的 x 相关，还是与外部 λ 相关？将这样的表达式输入 Wolfram 语言， x 会变成红色以表示存在问题：

```
Function[x, Function[x, x[x]]]
```

在这个层面上，解决这个问题的唯一方法是为这两个 x 使用不同的名称——例如：

```
λ[x, λ[y, y[x]]]
```

但最终所使用的特定名称并不重要。例如，交换 x 和 y

```
λ[y, λ[x, x[y]]]
```

它表示与之前完全相同的函数。

如何处理这种情况？一种方法——实际上是在 lambda 之前十多年就被发明了，而且我几年前确实写了一本关于这个主题的整本书 ——是使用所谓的“ 组合子 ”：这些结构实际上抽象地定义了如何重新排列符号表达式，而无需命名任何东西，或者，更不用说，引入任何变量。所以，例如， λ[x, λ[y, y[x]]] 可以写成

```
S[K[S[S[K][K]]]][S[K[K]][S[K][K]]]
```

如可验证：

```
In[•]:= S[K[S[S[K][K]]][S[K[K]][S[K][K]]][y][x] //.
        {S[x_][y_][z_] :> x[z][y[z]], K[x_][y_] :> x}
Out[•]= x[y]
```

在某种程度上，这非常优雅。但是，是的，它极其难以阅读。那么，有没有办法在不需要显式引入带名称的变量等的情况下，至少保留一些λ的可读性？事实证明，这是可以做到的。这始于一个观察，即一个命名变量（比如 x ）基本上只是指回引入该变量的特定λ（比如 λ[x, ...] ）的一种方式。而其思路是，与其使用命名变量来建立这种引用，我们改为通过说明λ相对于被引用对象的结构位置来建立这种引用。

例如，给定

```
λ[x, λ[y, y[x]]]
```

我们可以将其写成“de Bruijn 索引”（发音为“de broin”）的形式

```
λ[λ[1[2]]]
```

其中1表示该位置的变量与表达式中“前一个λ”相关联，而2表示该位置的变量与表达式中“前两个λ”相关联：

```
λ[x, λ[y, y[x]]]
```

所以，例如，我们作为 lambda 的整数版本现在可以写成：

```
{λ[λ[1]], λ[λ[2[1]]], λ[λ[2[2[1]]]], λ[λ[2[2[2[1]]]]],
 λ[λ[2[2[2[2[1]]]]]], λ[λ[2[2[2[2[2[1]]]]]]]}
```

好的，所以 lambda 可以表示事物（比如整数）。但是能否“用 lambda 进行计算”？或者换句话说，纯 lambda 能做什么？

从根本上说，只有一个操作——通常称为 beta 归约——它接收 lambda 给出的任何参数，并明确地“注入”到 lambda 的体中，实际上实现了将函数应用于其参数的“结构”部分。beta 归约的一个非常简单的例子是

```
λ[x, x[x]][a[b[c]]] ⟶ a[b[c]][a[b[c]]]
```

这相当于标准的 Wolfram 语言求值：

```
Function[x, x[x]][a[b[c]]]
a[b[c]][a[b[c]]]
```

在 de Bruijn 索引的术语下，这变成了

```
λ[1[1]][a[b[c]]]
```

这又可以被“计算”（使用 beta 归约）为：

```
a[b[c]][a[b[c]]]
```

这似乎是一个非常简单的操作。但，正如我们将看到的，当反复应用时，它可以导致各种复杂的行为和有趣的结构。而关键在于 lambda 可以作为其他 lambda 的参数出现，并通过 beta 归约被注入到它们的主体中。

但是，一旦将一个 lambda 注入到另一个 lambda 中，就会产生一些棘手的问题。如果用变量显式地写出所有内容，通常需要进行变量重命名，例如

```
λ[a, λ[b, b[a]]][λ[a, λ[b, a]]]
```

计算为：

```
λ[a, a[λ[b, λ[c, b]]]]
```

在 de Bruijn 索引方面，需要重新编号，以便

```
λ[λ[1[2]]][λ[λ[2]]]
```

求值为：

```
λ[1[λ[λ[2]]]]
```

很容易非正式地描述β还原在做什么。用 Wolfram 语言来说，它基本上只是一个转换：

```
λ[x_, body_][arg_] := body /. x → arg
```

换句话说，给定 **λ**[*x* , *body*][*y*] ，β还原是用 *y* 替换 body 中的每一个 *x*，并返回结果。但这并不像看起来那么简单。例如，如果 *y* 包含一个λ——如上例所示——那么我们可能需要进行一定程度的变量重命名（通常称为“α转换”）。是的，可能存在一连串的重命名。你可能需要为这些重命名想出

任意数量的不同新名称。

一旦你完成了替换（“β还原”） **λ**[*x* , *body*][*y*] ，这可能会“暴露”另一个λ，在其中你需要进行更多替换，如此等等。这可能听起来像是一个细节。但实际上它非常基础——正是它让λ能够表示“持续运行”的计算，并由此产生我们将在此讨论的丰富的 ruliology。

使用 de Bruijn 索引进行 beta 归约是不是更容易呢？不是，其实并不容易。是的，你不必为变量发明名称。但相反，每当你在实际插入或删除 lambda 时，你必须进行相当繁琐的索引递归重新排列。

此外还有一个问题。正如我们稍后将详细探讨的，对于给定的 lambda，通常可以进行多次 beta 归约。如果我们追踪所有可能性，就会得到一个完整的多路径图 ，展示了 lambda 求值的各种“路径”。但关于 lambda 的一个基本事实（称为汇聚性或 Church-Rosser 性质 ，与因果不变性相关）是，如果某个 lambda 的 beta 归约序列最终达到一个固定点（正如我们将看到的，这并非总是发生），那么这个固定点是唯一的：换句话说，求值 lambda 总会得到一个确定的结果（或“规范形式”）。

那么，在处理 lambda 时，应该怎么称呼所做的事情呢？在 Wolfram 语言中，我们将将 Function[x, x[x]][a] （或 **λ**[x, x[x]][a] ）转换为 a[a] 的过程称为“求值”。但有时它也被称作，例如，约化、转换、替换或重写。在 Wolfram 语言中，我们可能也将 **λ**[…] 称为一个“λ表达式”，但它有时也被称作一个λ项。那它里面是什么呢？在 **λ**[*x* , *body*] 中，*x* 通常被称为一个“绑定变量”。（如果有一个变量——比如 y ——出现在 *body* 中，并且没有被λ“绑定”（或“作用域”）所限定，那么它被称为一个“自由变量”。我们在这里讨论的λ表达式通常没有自由变量；这种λ表达式通常被称为“闭λ项”）。

将一个 lambda 应用于某物——例如 **λ**[*x* , *东西*] ——通常被称为“λ抽象”，或简称“抽象”，而λ本身有时被称为“抽象器”。在一个 lambda 的“主体”中，将 x 应用于 y 以形成 x[y] ——毫不奇怪——通常被称为“应用”（并且，正如我们稍后会讨论的，它可以表示为 x@y 或 x•y ）。那么对 lambda 进行的操作呢？从历史上看，有三种被识别出来：α简化（或α转换）、β简化（或β转换）和η简化（或η转换）。β简化——正如我们上面讨论的——实际上是核心的“求值风格”操作（例如用 **λ**[x, f[x]][y] 替换 f[y] ）。α和η简化在某种意义上是“符号维护操作”；α简化是变量的重命名，而η简化是将 **λ**[x, f[x]] 简化为 f 单独。 （原则上也可以想象其他简化——比如直接为 **λ**[_][_][_] 定义的转换——但这些对 lambda 来说不是传统的。）

对 lambda 进行归约的实际部分通常被称为“redex”；lambda 的其余部分则是“上下文”——或“带有一个空位的表达式”。在我们这里要进行的操作中，我们主要关注的是逐步评估/归约/转换/…

lambda 的——本质上计算性的——过程。但过去在 lambda 上完成的许多形式工作主要集中在稍有不同的问题上，即“λ转换演算”——或简称“λ演算”，它特别关注识别 lambda 之间的等价性，换句话说，确定一个λ表达式何时可以转换为另一个，或与另一个具有相同的规范形式。

顺便说一句，值得说明的是——正如我上面提到的——lambda 与我几年前详细写过的一些组合子之间存在密切关系。而且确实，我之前讨论过的许多组合子的现象，以各种修改形式出现在我这里关于 lambda 的论述中。一般来说，在形式和计算层面，组合子比 lambda 更容易处理。但 lambda 有一个显著的优势，至少在单个 beta 归约的层面上，它们的作用更容易解释。《组合子 S 以一种令人费解的方式工作》。但 beta 归约本质上只是“将一个函数应用于一个参数”。当然，当进行大量 beta 归约时，结果可能会非常复杂——这也是我们在这里要讨论的内容。

使用 lambda 进行基本计算

如何仅使用 lambda 进行熟悉的计算？首先，我们需要一种纯用 lambda 表示事物的方法。例如，如上所述，我们可以用连续嵌套来表示整数。但现在我们也可以纯用 lambda 定义一个明确的后继函数。有许多可能的方法来做这件事；一个例子是

```
s = λ[λ[λ[2[3[2][1]]]]]
```

这等价于：

```
λ[a, λ[b, λ[c, b[a[b][c]]]]]
```

将零定义为

```
z = λ[λ[1]]
```

或

```
λ[a, λ[b, b]]
```

我们现在可以形成一个表示数字3的表示方法：

```
In[•]:= s[s[s[z]]]
Out[•]= λ[λ[λ[2[3[2][1]]]]][λ[λ[λ[2[3[2][1]]]]][λ[λ[λ[2[3[2][1]]]]][λ[λ[1]]]]]
```

但通过应用β归约来“计算”这个表达式，结果会归约为

```
λ[λ[2[2[2[1]]]]]
```

这正是我们之前所表示的3。

同样地，我们可以定义

```
plus = λ [ λ [ λ [ λ [ 4 [ 2 ] [ 3 [ 2 ] [ 1 ] ] ] ] ] ]
```

```
times = λ [ λ [ λ [ 3 [ 2 [ 1 ] ] ] ] ]
```

（其中这些函数以 “部分应用”形式 ，比如 times[3][2] 等，接受参数）。

那么例如3×2就变成

```
in  |- times [ s [ s [ s [ z ] ] ] ] [ s [ s [ z ] ] ]
out |- λ [ λ [ λ [ 3 [ 2 [ 1 ] ] ] ] ] [ λ [ λ [ λ [ 2 [ 3 [ 2 ] [ 1 ] ] ] ] ] [
          λ [ λ [ λ [ 2 [ 3 [ 2 ] [ 1 ] ] ] ] ] [ λ [ λ [ λ [ 2 [ 3 [ 2 ] [ 1 ] ] ] ] ] [ λ [ λ [ 1 ] ] ] ] ] ] [
        λ [ λ [ λ [ 2 [ 3 [ 2 ] [ 1 ] ] ] ] ] [ λ [ λ [ λ [ 2 [ 3 [ 2 ] [ 1 ] ] ] ] ] [ λ [ λ [ 1 ] ] ] ] ]
```

它在经过 37 步的 beta 归约后，评估为：

```
λ [ λ [ 2 [ 2 [ 2 [ 2 [ 2 [ 2 [ 1 ] ] ] ] ] ] ] ]
```

将其应用于“未评估”的 s 和 z ，我们得到

```
λ [ λ [ 2 [ 2 [ 2 [ 2 [ 2 [ 2 [ 1 ] ] ] ] ] ] ] ] [ s ] [ z ]
```

在 beta 归约下，它给出

```
s [ s [ s [ s [ s [ z ] ] ] ] ]
```

我们可以将其识别为我们表示数字6的表示方法。

而且，是的，我们可以更进一步。例如，这里是一个用纯 lambda 表示的阶乘函数的表示（毫无疑问不是最小的表示）：

```
fac = λ [ λ [ 2 [ 1 [ 1 ] ] ] [ λ [ 2 [ 1 [ 1 ] ] ] [
    λ [ λ [ λ [ 1 ] [ λ [ λ [ 2 ] [ λ [ 1 ] [ 1 ] ] ] [ λ [ λ [ 2 ] [ 1 ] ] ] [ λ [ λ [ 1 ] ] ] ] ] ] [
        λ [ λ [ λ [ 3 ] [ 2 ] [ 1 ] ] ] [ λ [ λ [ λ [ 1 ] ] [ 2 ] [ λ [ λ [ λ [ 3 ] [ λ [ 1 ] [ 2 ] [ 4 ] ] ] ] [ λ [ 2 ] ] ] [ λ [ 1 ] ] ] ] [ 1 ] ] ] ] ]
```

并且根据这个定义

```
fac [ s [ s [ s [ z ] ] ] ]
```

即4!，确实最终计算结果为24：

```
λ[λ[2[2[
    2[2[2[2[2[2[2[2[2[2[2[2[2[2[2[2[2[2[2[2[2[2[1]]]]]]]]]]]]]]]]]]]]]]]]]]
```

稍后我们将讨论进行此类评估的过程，我们会看到，是的，这是复杂的。在我们默认进行此类评估的方式中，这个特定的 lambda 需要 5067 步（对于连续的阶乘，所需的步数是 41，173，864，5067，34470，...）。

编写 Lambda 的方法

到目前为止我们所见到的，关于 lambda 的一个明显观察是——除了琐碎的情况之外——它们很难读懂。但这是必要的吗？有很多方法可以写和渲染它们。但我们会发现，虽然不同的 lambda 有不同的特点和优势（我们之后会利用这些），但没有一种方法能真正“破解密码”，使 lambda 对我们来说普遍容易读懂。

让我们考虑以下用标准文本方式写出的 lambda：

```
λ[x, λ[y, y[y]] [x[x] [λ[z, x[z[z]]]]]]
```

这表示出现变量的相互关系：

是的，因为每个λ“作用域”其绑定变量，我们同样可以用 y 来代替 z：

```
λ[x, λ[y, y[y]] [x[x] [λ[y, x[y[y]]]]]]
```

使这种表达式难以阅读的一个明显因素是其中包含的所有括号。那么我们如何去掉这些括号呢？嗯，在 Wolfram 语言中我们总是可以写 f@x 来代替 f[x]：

```
λ[x, λ[y, y@y] @ (x@x) @ λ[z, x@z@z]]
```

是的，因为 Wolfram 语言中的 @ 运算符是右结合的，我们可以用 f@g@x 来表示 f[g[x]] 。所以，好吧，使用 @ 让我们可以去掉相当多的括号。但它引入了小括号。那么我们该如何处理这些小括号呢？我们可能尝试的是用应用运算符 • (Application)代替 @ ，其中 • 定义为左结合，所以 f•g•x 是 f[g][x] 而不是 f[g[x]] 。所以从•的角度来看，我们的 lambda 变成了：

```
λ[x, λ[y, y•y] • (x•x•λ[z, x•(z•z)])]
```

小括号的位置移动了，但它们并没有消失。

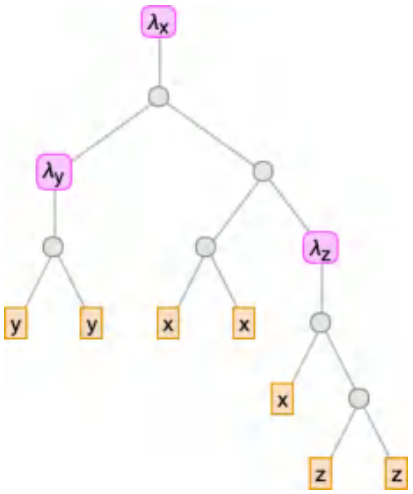
我们可以通过使用空格代替显式的 @ 或 • 来让这些表达式看起来稍微简单一些。但要了解这些表达式的含义，我们必须决定空格是表示 @ 还是 • 。在 Wolfram 语言中， @ 通常比 • 更有用（因为，例如， f@g@x 方便地表示连续应用一个函数，然后是另一个函数）。但在数学逻辑的早期历史中，空格实际上被用来表示•。哦，还有 λ[*x* , *body*] 被写成λ*x* . *body*，为我们的例子 lambda 给出以下形式：

```
(λx.((λy.(yy))((xx)(λz.(x(zz))))))
```

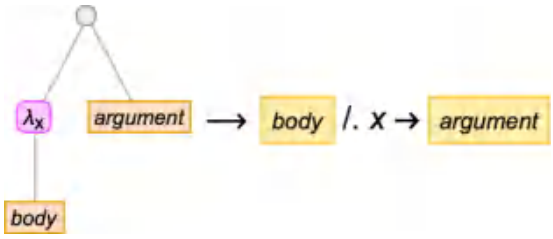
我们一直在尝试以线性形式书写 lambda。但在某种意义上，它们始终真的是树——就像 Wolfram 语言中的表达式是树一样。所以这里，例如是

```
λ[x, λ[y, y[y]][x[x][λ[z, x[z[z]]]]]]
```

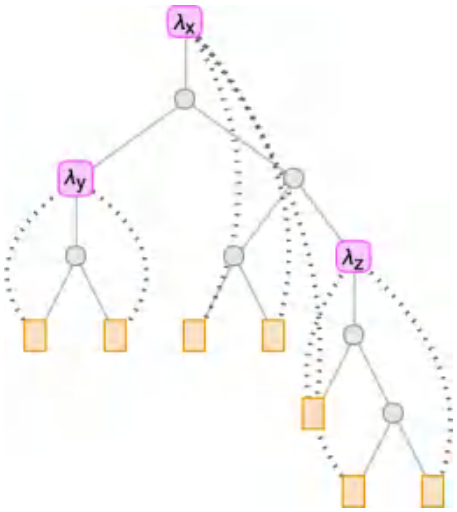
以树的形式显示：



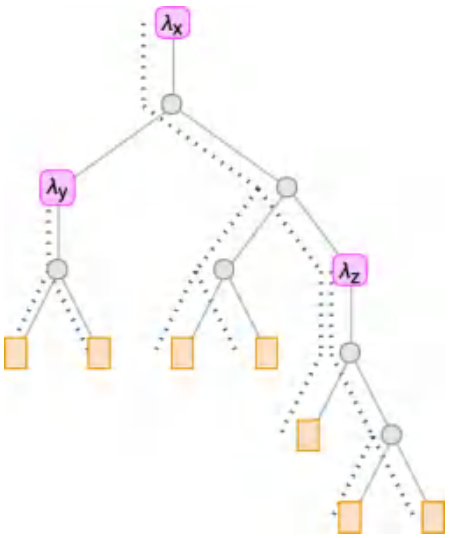
树的叶子是变量。中间节点是 λ[... , ...] （“抽象”）或... [...] （“应用”）。关于 lambda 的所有内容都可以在这个树表示中完成。例如，beta 约简可以被视为对代表 lambda 的树的一部分进行的替换操作：



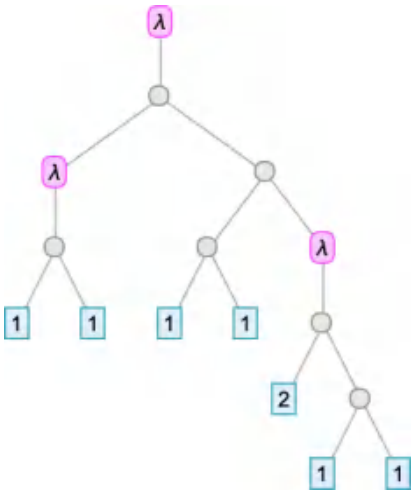
但是，好的，正如我们上面讨论的，变量被赋予的具体名称无关紧要。重要的是它们与哪些 lambda 相关联。我们可以通过在树上叠加适当的连接来表示这一点：



但如果我们将这些连接沿着树中的分支进行路由呢?



要找出给定变量与哪个 λ 匹配，我们只需要计算在从代表该变量的叶节点向上遍历树的过程中，在到达“它的 λ”之前遇到多少个 λ。这意味着我们只需要在每个树的叶节点处填入一个数字，就可以表示 lambda：



这些数字正是我们之前讨论过的 de Bruijn 索引。所以现在我们看到，我们可以用 de Bruijn 索引 lambda 树来替换我们的“命名变量” lambda 树。

将这棵特定的树表示为表达式，我们得到：

```
λ[λ[1[1]][1[1][λ[2[1[1]]]]]]
```

现在的λ没有明确指定任何变量。但是，是的，表达式的连接方式与之前我们明确使用变量时相同：

我们再次可以通过使用•（ Application ）运算符来去掉我们的“应用括号”：

```
λ[λ[1•1]•(1•1•λ[2•(1•1)])]
```

实际上，我们在这里并不需要 • 或者 λ：所有内容都可以仅从括号的序列中推导出来。因此，最终我们可以将我们的 lambda 写成

```
[[11](11[2(11)])]
```

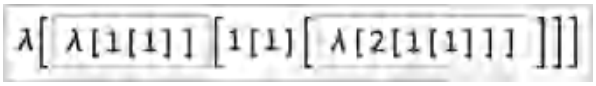
在每一个“ [”字符之前都有一个“隐式λ”，在每一对数字之间都有一个隐式•。（是的，我们只处理所有 de Bruijn 索引都低于 10 的情况。）

现在我们有了将λ写成一串字符的最小方式。它很紧凑，但难以解码。那么我们如何能做得更好呢?

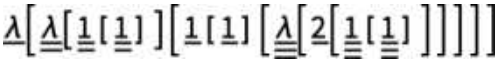
在最简单的层面上，例如，我们可以给每个字符上色（这里包括显式的λ和括号）

确实，我们稍后会发现这种表示法很有用。

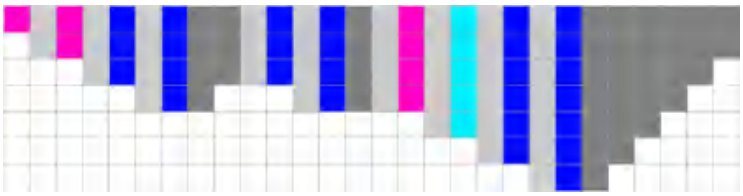
但是任何“纯字符串”都只能相当隐含地显示 lambda 的嵌套结构，在括号序列中。那么我们如何使其更明确呢？嗯，我们可以显式地框定每个嵌套的λ：



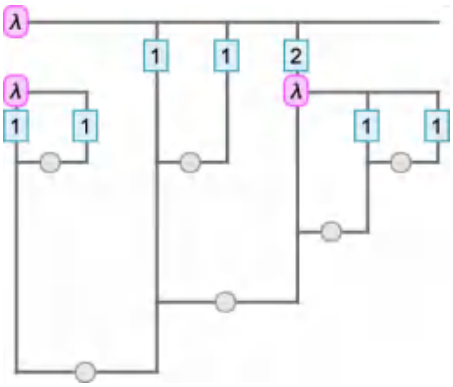
或者我们只需标示每个元素的“深度”——要么“排版上”



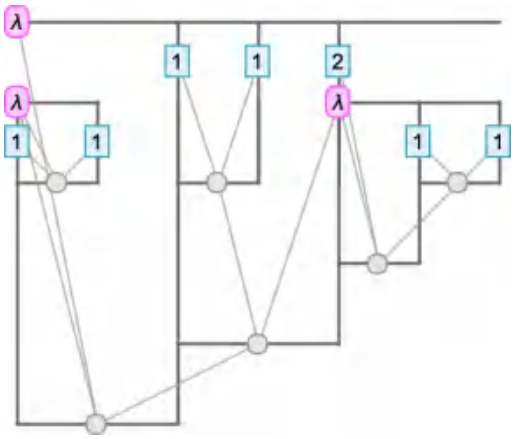
或图形化（其中深度是 TreeDepth ）：



我们可以将其视为λ树表示的一维展开版本。另一种方法实际上是在二维网格上绘制树——形成我们称之为 Tromp 图的结构。

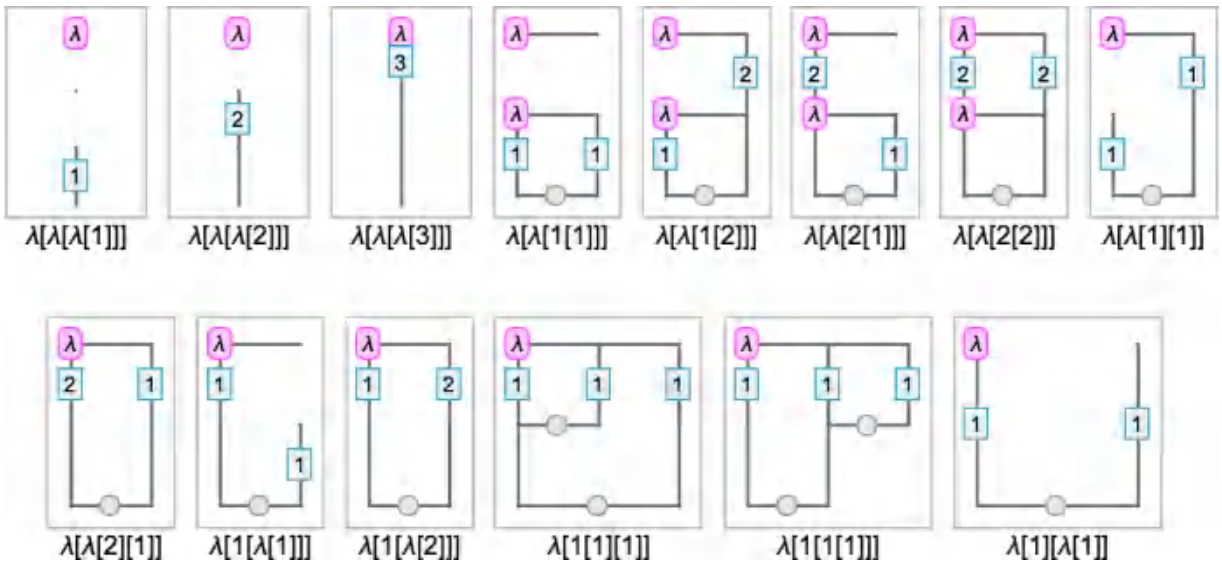


要了解这里发生了什么，我们可以将标准树叠加在 Tromp 图上：



每条垂直线代表一个变量的实例。线的顶端表示与该变量关联的λ——λ由一条水平线表示。当一个变量的实例应用于另一个时，例如 [_*v*] ，有一条水平线从“”垂直线延伸到“*v*”垂直线。

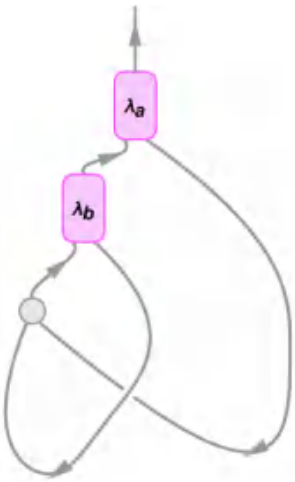
基于此设置，以下是一些小型λ的 Tromp 图：



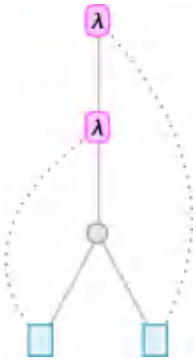
另一种图形方法是使用“字符串图”，它直接展示了 lambda 的元素是如何“连接”的。例如，考虑一个简单的 lambda：

λ [a, λ [b, b [a]]]

这可以用字符串图表示：

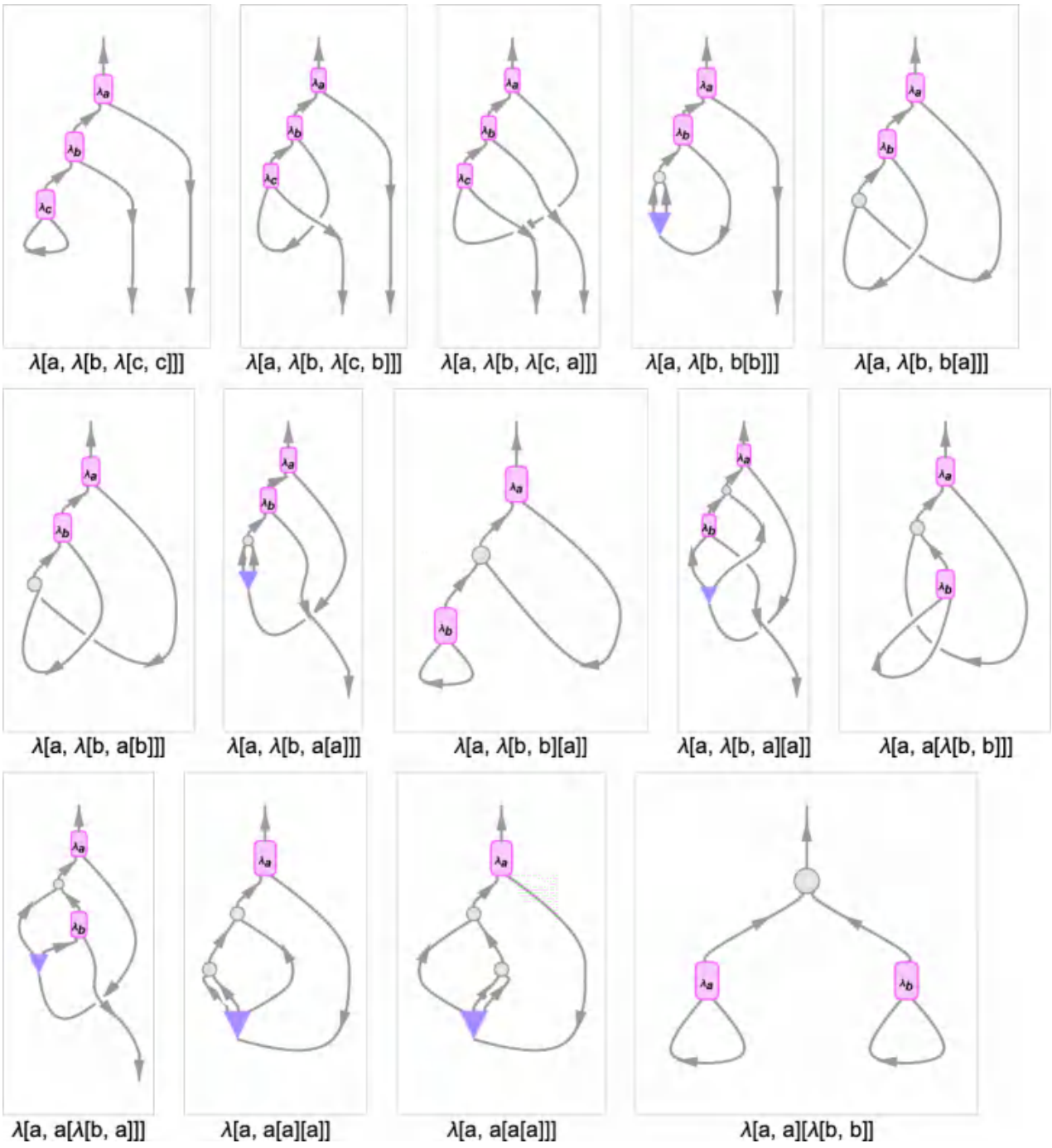


这里的连接性与以下相同：



然而在字符串图中，显式变量被“压缩”掉了，但通过哪些λ的线正确回溯来隐式指示。在每个λ节点处，有一个出边代表λ的参数，有一个入边连接到它的主体。就像我们用树来表示 lambda 一样，lambda 的“内容”从图的顶部“悬挂”下来。

这里是一些小 lambda 的字符串图示例：

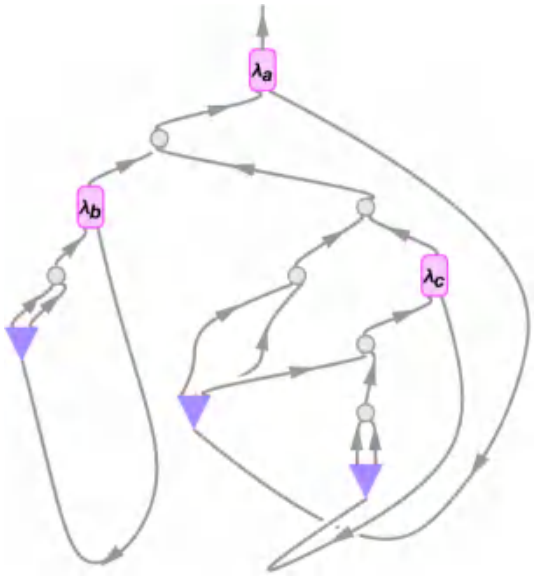


在这些图中，从底部“离开图”的边对应于 lambda 中未使用的变量。图中出现的“杯子”对应于“立即使用的变量”，例如 λ[*x* , *x*] 。那么，例如在 λ[a,λ[b,b[b]]] 中的 ▼ 呢？它表示变量的复制，这里需要这样做是因为 b 在 lambda 中被使用了两次。

回到我们之前的例子（现在使用显式变量书写）

λ [a, λ [b, b [b]] [a [a] [λ [c, a [c [c]]]]]]

这是与此 lambda 对应的（有点复杂）的字符串图：



可以将这种字符串图想象为通过变量关联的“管道”定义了“符号内容的流动”。但是，有没有某种更纯粹的方式来表示这种流动，而无需引入变量呢？当然有，那就是使用组合子 。实际上，组合子表达式和 lambda 之间存在直接的对应关系。例如

```
λ[λ[1[1]][1[1][λ[2[1[1]]]]]]
```

可以用组合子表示为：

```
S[K[S[S[K][K]][S[K][K]]]][S[S[S[K][K]][S[K][K]]]
   S[S[K[S]][S[K][K]]][S[K][K]]][K][S[S[K][K]][S[K][K]]]]
```

正如组合子通常那样，这在某种程度上非常纯粹和统一，但它也非常难以解释。而且，是的，正如我们稍后会讨论的，小的 lambda 表达式可以对应于大的组合子表达式。反过来也是如此，例如，**S[S][K[S]]** 作为一个 lambda 表达式

```
λ[λ[λ[3[1][2[1]]]]][λ[λ[λ[3[1][2[1]]]]][λ[λ[2]][λ[λ[λ[3[1][2[1]]]]]]]]
```

或以简洁形式表示：

```
[[[31(21)]]][[[31(21)]]]([[2]][[[31(21)]]])
```

但是，好吧，这种紧凑形式在某种程度上是一种相当有效的表示。但它有一些粗糙的特点。比如，例如，如果有 de Bruijn 索引的值有多于一位数字，它们就需要用某种“标点”隔开。

但如果我们坚持将 lambda 表达式纯粹表示为 0 和 1 的序列呢？好吧，我们可以做的一件事是采用上述紧凑形式，用固定字符串表示其“标点符号”，然后有效地用一进制数表示其 de Bruijn 索引——

以我们上面的示例为例：

```
00010001101001011010000111001101O
```

这个程序可以被视为一种方法，通过它可以用一个整数来表示任何 lambda（尽管当然不是每个整数都与一个 lambda 相关联）。

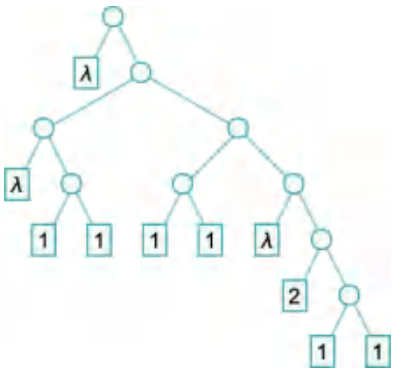
枚举 Lambda

要开始对 lambda 及其作用进行规则学研究，我们需要讨论 lambda 可能存在的形式——换句话说，如何列举 lambda 表达式。一个明显的策略是查看所有规模逐渐增大的 lambda 表达式。但 lambda 表达式的“规模”应该用什么来衡量?

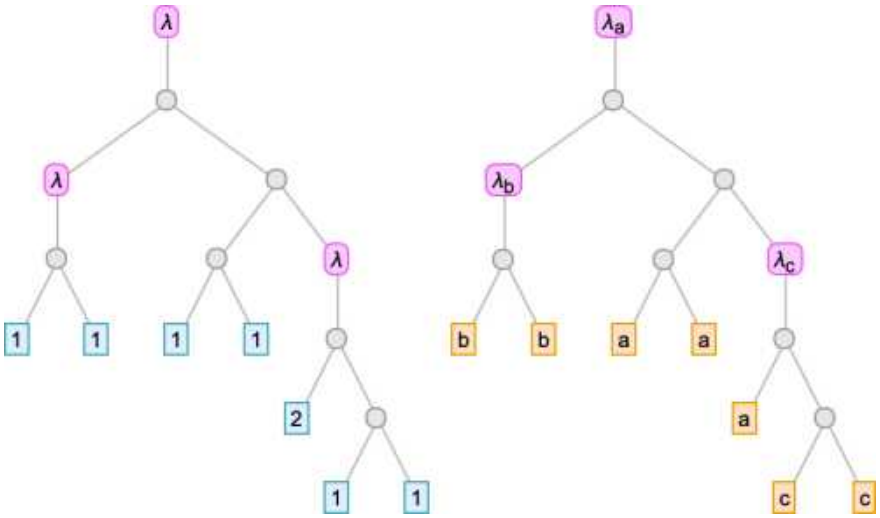
这里我们主要使用 Wolfram 语言 LeafCount 。例如

```
λ[λ[1[1]][1[1][λ[2[1[1]]]]]]
```

将被认为具有“大小”10，因为在最直接的 Wolfram 语言树渲染中



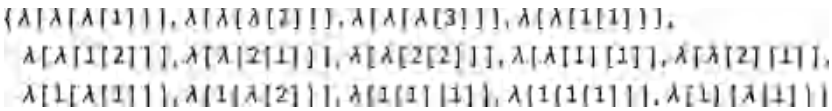
有10个“叶”。我们通常会以类似的形式来渲染λ。



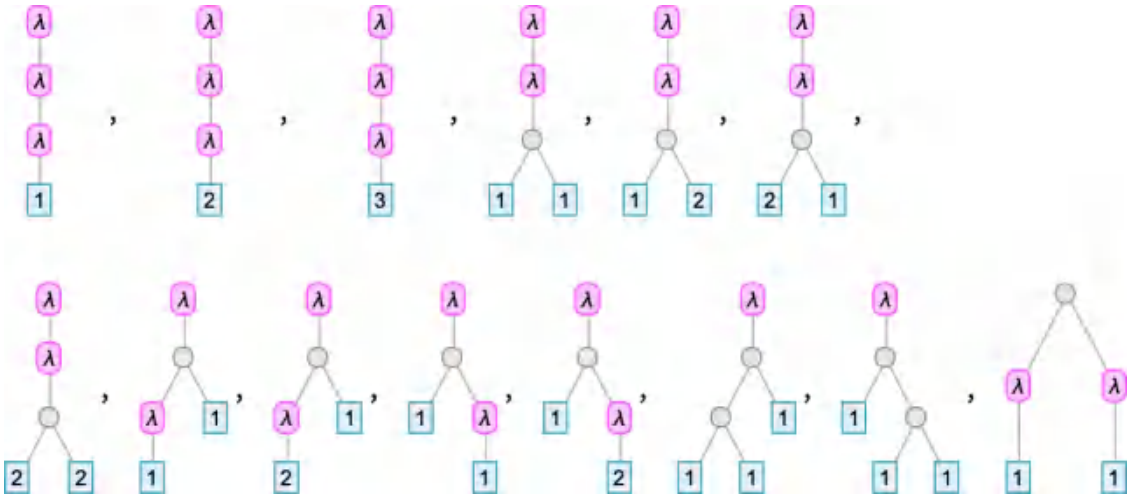
在这种情况下，大小是“de Bruijn 索引叶”或“变量叶”的数量加上λ的数量。

（请注意，我们可以考虑其他大小度量方式，例如，对不同抽象（λ）和应用，或对具有不同值的 de Bruijn 索引“叶子”赋予不同的权重。）

例如，使用 LeafCount 作为度量，就有 14 个大小为 4 的 lambda：



作为树形结构，这些变成



而作为 Tromp 图，它们是：

lambda 的数量 随着规模迅速增长：

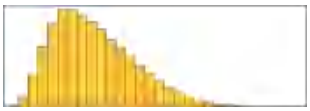
size	2	3	4	5	6	7	8	9	10	11	12
number	1	3	14	82	579	4741	43977	454283	5159441	63782411	851368766

这些数字实际上由 c[n,0] 通过简单的递归计算给出：

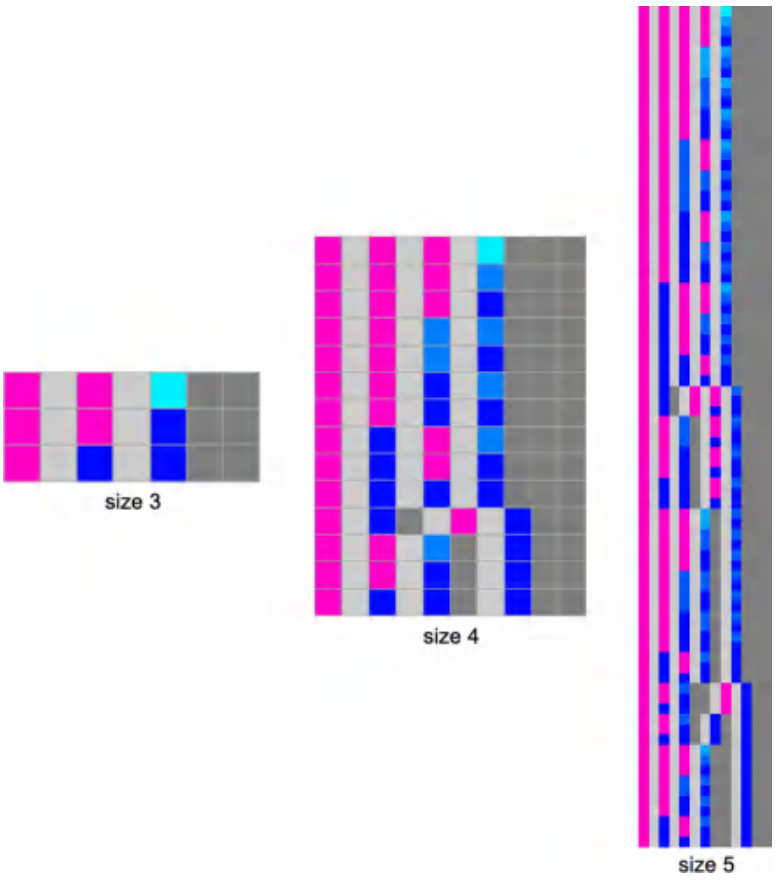
$$c[1, j\_] := j; c[i\_, j\_] := c[i, j] = c[i - 1, j + 1] + \sum_{k=1}^{i-1} c[k, j] \cdot c[i-k, j]$$

对于大的 n，这大致增长如 $n!$，尽管显然稍慢。

在大小为 n 时，表达式树的最大深度是 $n-1$；平均深度约为 $3+n/4$，极限分布为：



我们还可以显示 lambda 的实际形式，一个显著的特点是看到的形式的相对多样性：



Lambda 的评估

“评估”一个 lambda 是什么意思？最明显的解释是进行一系列的 beta 归约，直到达到一个“不动点”，在这个点上不再可以进行进一步的 beta 归约。

所以，例如，可能会有：

$$\lambda[a,\lambda[b,b[b]][\lambda[c,c[a]]]]\longrightarrow$$
$$\lambda[a,\lambda[b,b[a]][\lambda[c,c[a]]]]\longrightarrow\lambda[a,\lambda[b,b[a]][a]]\longrightarrow\lambda[a,a[a]]$$

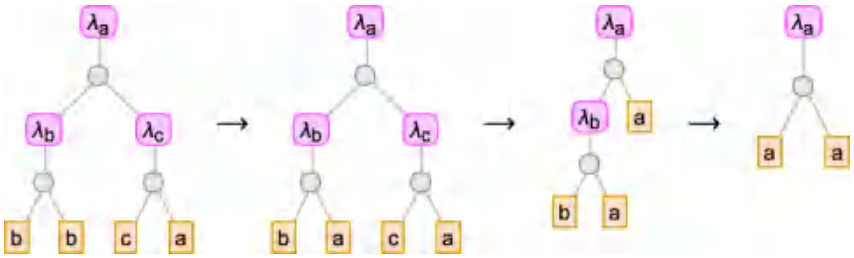
如果使用 de Bruijn 索引，β归约变成一个 λ[_][_] 的转换，我们的例子变成：

$$\lambda[\lambda[1[1]][\lambda[1[2]]]]\longrightarrow\lambda[\lambda[1[2]][\lambda[1[2]]]]\longrightarrow\lambda[\lambda[1[2]][1]]\longrightarrow\lambda[1[1]]$$

在紧凑记法中，这是

$$[[11][12]]\longrightarrow[[12][12]]\longrightarrow[[12]1]\longrightarrow[11]$$

而在树的形式下则是：



通过将 de Bruijn 索引形式的字符渲染为彩色方块，我们也可以在“时空”形式中表示这种演化：

那其他 lambda 呢？3 个 lambda 中的任何一个

$$\{\lambda[\lambda[1]],\lambda[\lambda[2]],\lambda[1[1]]\}$$

包含模式 λ[_][_]，因此它们都不允许贝塔归约。在大小为 4 时，14 种可能的 lambda 中的大多数情况也是如此，但有一个例外，它允许单步贝塔归约：

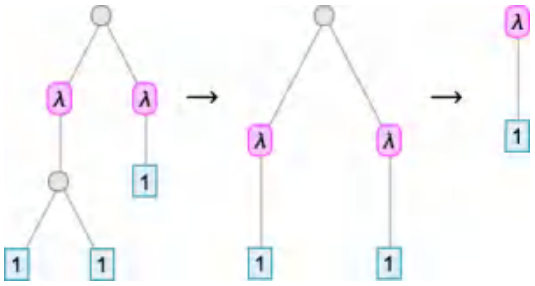
$$\lambda[1][\lambda[1]]\longrightarrow\lambda[1]$$

在大小为 5 时，35%的 lambda 不是惰性的，但它们仍然具有较短的“生命周期”

lifetime	1	2	3
number	53	28	1

其中最长的“求值链”是：

λ[1[1]][λ[1]]⟶λ[1][λ[1]]⟶λ[1]



在大小为 6 时，44%的 lambda 允许 beta 约简，并且不是立即惰性的：

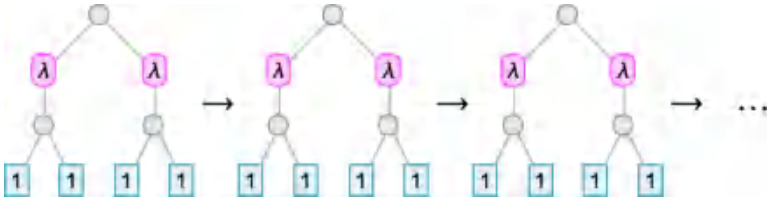
lifetime	1	2	3	4	∞
number	323	224	29	2	1

有几个 lambda 具有长度为 4 的求值链：

λ[1[1][1]][λ[1]]⟶λ[1][λ[1]][λ[1]]⟶λ[1][λ[1]]⟶λ[1]⟶λ[1]
λ[1[1[1]]][λ[1]]⟶λ[1][λ[1][λ[1]]]⟶λ[1][λ[1]]⟶λ[1]⟶λ[1]

现在又有了新东西——一个周期为 1 的“循环 lambda”（或“quine”），它不断通过β还原转化为自身，因此永远不会达到一个无法再应用β还原的固定点：

λ[1[1]][λ[1[1]]]⟶λ[1[1]][λ[1[1]]]⟶λ[1[1]][λ[1[1]]]⟶...

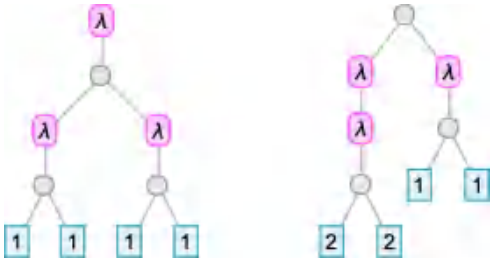


在大小 7 时，超过一半的 lambda 都不是惰性的：

lifetime	1	2	3	4	5	∞
number	2359	1925	402	38	10	7

有几个“自我重复”的 lambda：

λ[λ[1[1]][λ[1[1]]]]→λ[λ[1[1]][λ[1[1]]]]→λ[λ[1[1]][λ[1[1]]]]→...
λ[λ[2[2]]][λ[1[1]]]→λ[λ[2[2]]][λ[1[1]]]→λ[λ[2[2]]][λ[1[1]]]→...

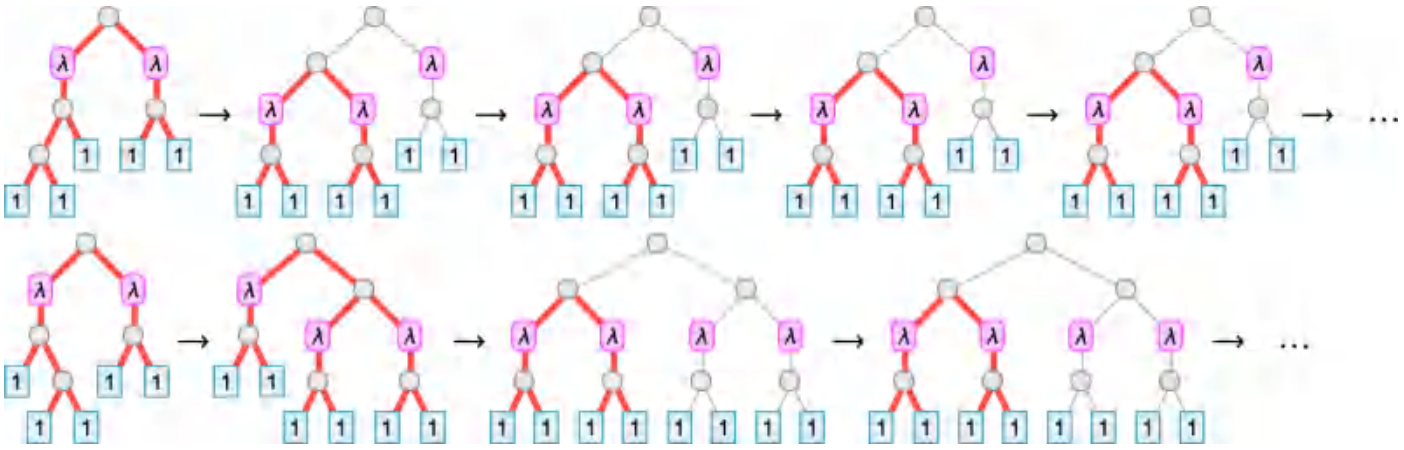


第一个是一个简单的自我重复 lambda 在大小 6 上的扩展；第二个则至少有些不同。

在大小7时，也有几个案例在达到最终重复状态之前有一个小的“瞬时”状态（这里以紧凑形式展示）：

```
[111][11]          [1(11)][11]
[11][11][11]       [11]([11][11])
[11][11][11]       [11][11]([11][11])
 ⋮                 [11][11]([11][11])
                    ⋮
```

这是这种情况下的 lambda 树的变化——现在我们在红色中标记了每棵树中参与每次 beta 归约的部分：

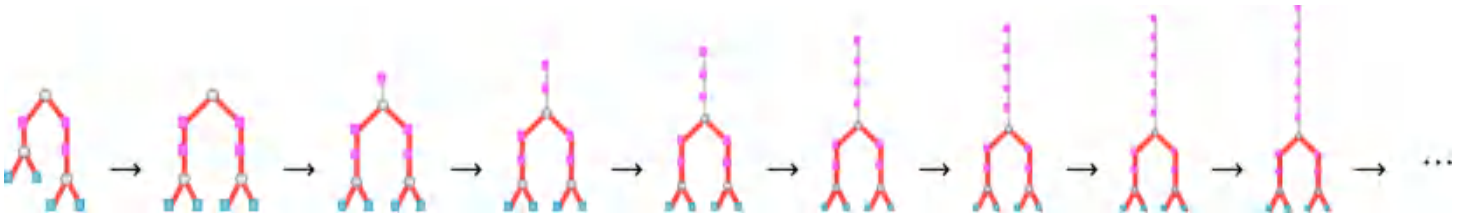


而且还有新的情况：有三种情况下，beta 归约会导致 lambda 表达式无限增长。最简单的情况是

λ[1[1]][λ[λ[2[2]]]]

其大小在每一步增加1：

```
[11] [ [22] ]
[ [22] ] [ [22] ]
[ [ [22] ] [ [22] ] ]
[ [ [ [22] ] [ [22] ] ] ]
[ [ [ [ [22] ] [ [22] ] ] ] ]
[ [ [ [ [ [22] ] [ [22] ] ] ] ] ]
```

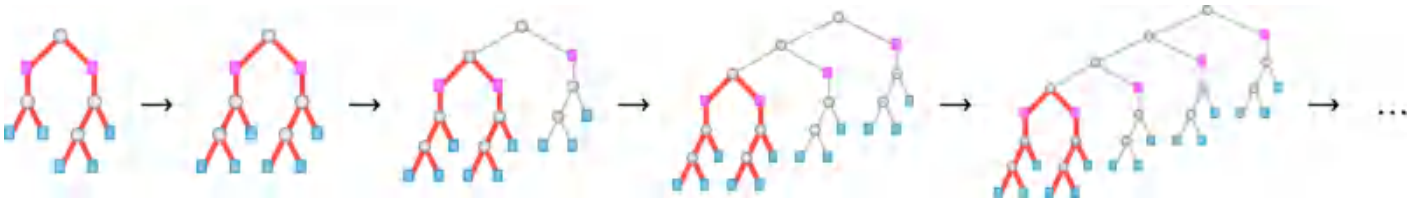


然后是

```
λ [1 [1] ] [λ [1 [1] [1] ] ]
```

在单步瞬态之后，每一步都会增长4倍

```
[11] [111]
[111] [111]
[111] [111] [111]
[111] [111] [111] [111]
[111] [111] [111] [111] [111]
[111] [111] [111] [111] [111] [111]
```

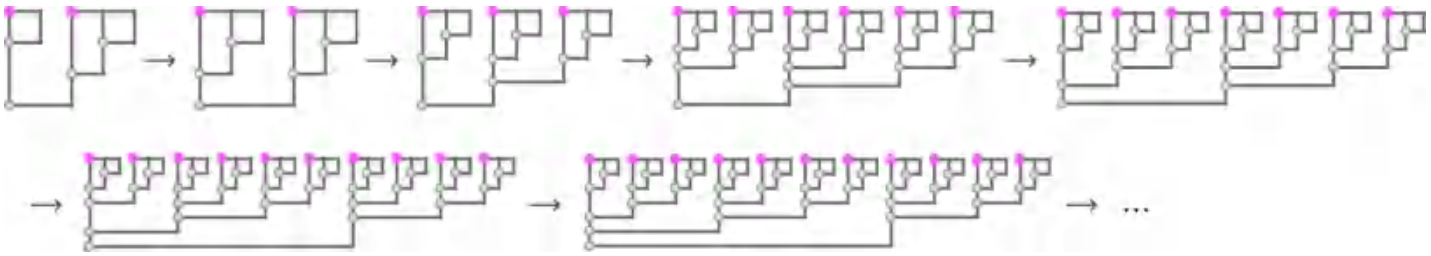


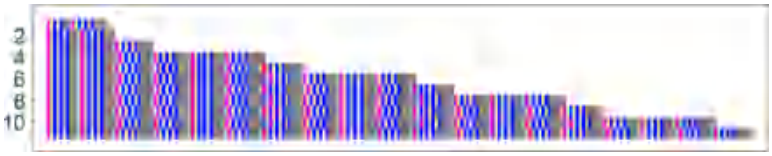
最后还有

```
λ [1 [1] ] [λ [1 [1 [1] ] ] ]
```

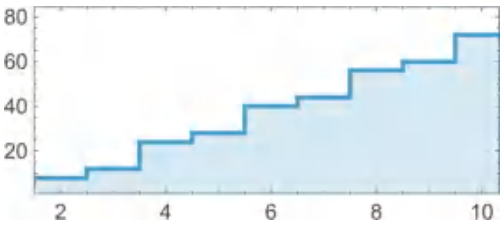
其增长方式稍显复杂

```
[11] [1(11)]
[1(11)] [1(11)]
[1(11)] ([1(11)] [1(11)])
[1(11)] [1(11)] ([1(11)] [1(11)] ([1(11)] [1(11)]))
[1(11)] ([1(11)] [1(11)]) ([1(11)] [1(11)] ([1(11)] [1(11)]))
[1(11)] [1(11)] ([1(11)] [1(11)] ([1(11)] [1(11)])) ([1(11)] [1(11)] ([1(11)] [1(11)]))
```





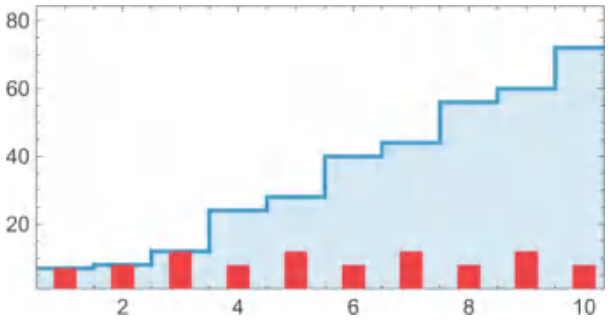
在连续步骤中交替增长 4 和 12（所以在步骤 $t > 1$ 时它是大小为 $8t - 4\ \text{Mod}[t + 1, 2]$）：



如果我们看这里“下面”发生的情况，我们看到β简化只是在每一步反复应用于第一个λ：

```
λ[1[1]][λ[1[1[1]]]]
λ[1[1[1]]][λ[1[1[1]]]]
λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]]]]
λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]]]]]]
λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]]]]]]]
λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]]]]]]]]]
λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]]]]]]]]]]
λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]]]]]]]]]]]
λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]][λ[1[1[1]]]]]]]]]]]]]
```

而且，当我们绘制λ的连续大小时，我们可以方便地表示这一点：



在大小为 8 时，事情开始变得更有趣。在 43,977 个可能的 lambda 中，有 55%不是惰性的：

lifetime	1	2	3	4	5	6	7	8	11	12	∞
number	19877	18337	4839	656	154	19	9	3	1	1	81

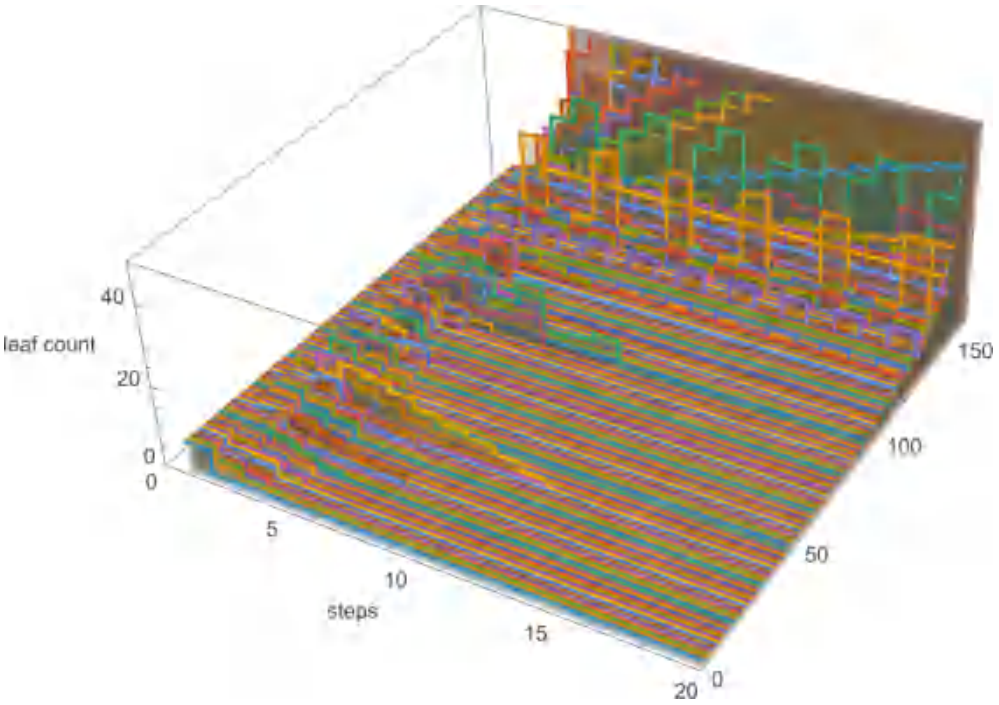
终止的最长求值链发生时

```
λ[1[1]][λ[1[1[λ[1]]]]]
```

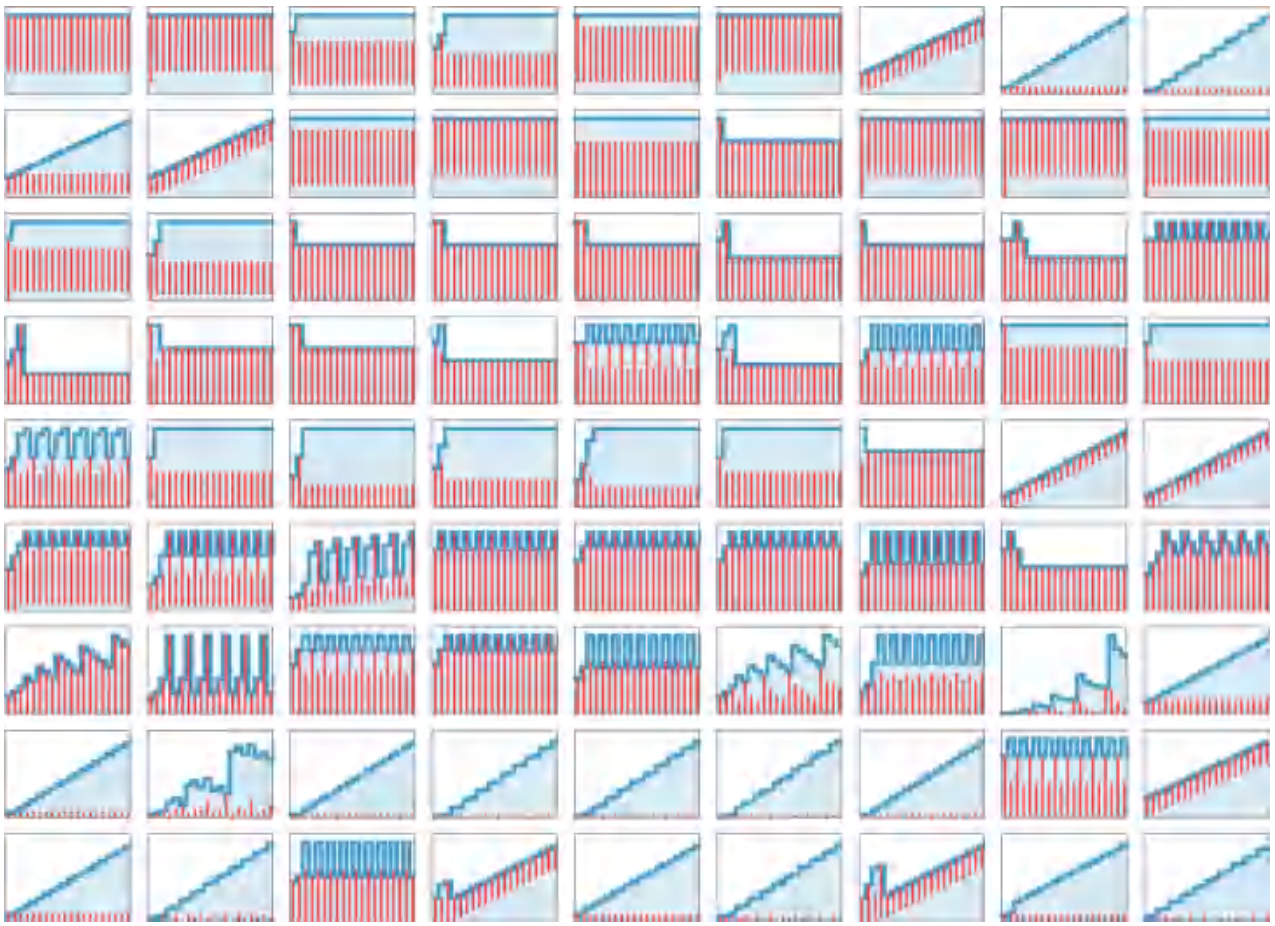
长度为12：

```
[11][1(1[1])]
[1(1[1])][1(1[1])]
[1(1[1])]([1(1[1])][1])
[1(1[1])][1]([1(1[1])][1][1])
[1]([1][1])([1(1[1])][1][1])
[1][1]([1(1[1])][1][1])
[1]([1(1[1])][1][1])
[1(1[1])][1][1]
[1]([1][1])[1]
[1][1][1]
[1][1]
[1]
```

在所有 43,977 个大小为 8 的 lambda 中，只有 164 种不同的增长模式：



那些不终止的 81 个 lambda 具有各种增长模式：



还有一些，如

λ[1[λ[2][1]]][λ[1[1]]]

每隔一步交替：

```
[1([2]1)][11]
[11]([[11]][11])
[[11]][11]([[11]][11])
[11]([[11]][11])
[[11]][11]([[11]][11])
[11]([[11]][11])
[[11]][11]([[11]][11])
[11]([[11]][11])
```

其他一些，如

λ[1[1[λ[2]]]][λ[1[1]]]

具有周期3：

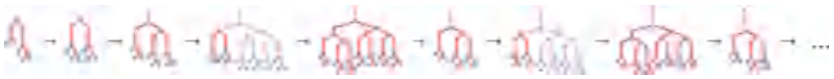
```
[1(1[2])][11]
[11]([11][[11]])
[11][[11]]([11][[11]])
[[11]][[11]]([11][[11]])
[11]([11][[11]])
[11][[11]]([11][[11]])
[[11]][[11]]([11][[11]])
[11]([11][[11]])
```

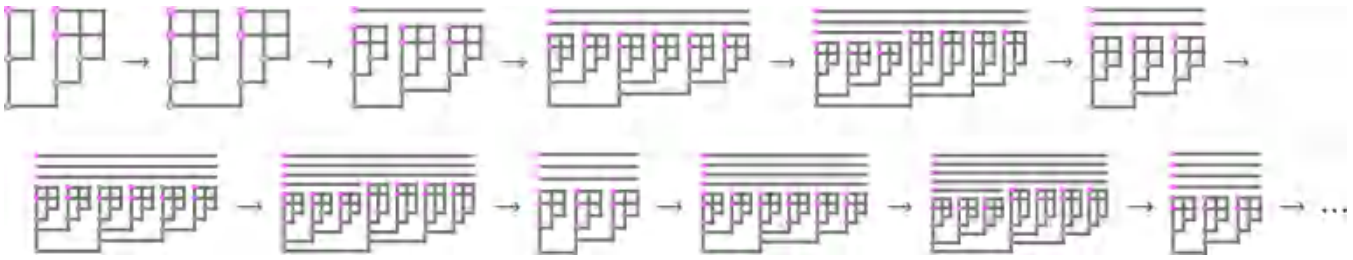
有时，例如

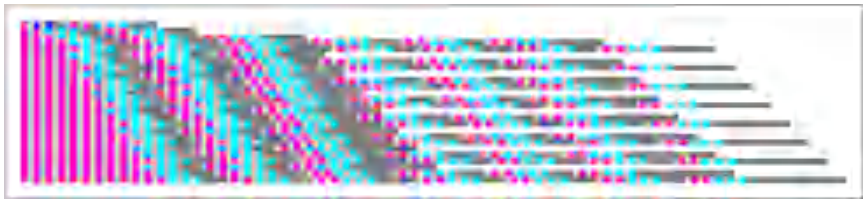
λ[1[1]][λ[λ[2[2[2]]]]]

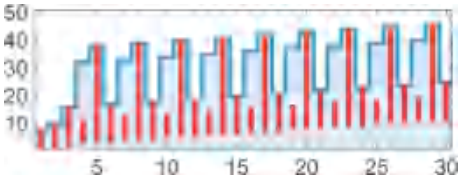
存在增长和周期性的混合

[11] [[2 (22)]]
[[2 (22)]] [[2 (22)]]
[[[2 (22)]] ([[2 (22)]] [[2 (22)]])]
[[[[2 (22)]] [[2 (22)]] ([[2 (22)]] [[2 (22)]] ([[2 (22)]] [[2 (22)]]))]]
[[[[[2 (22)]] ([[2 (22)]] [[2 (22)]])] ([[2 (22)]] [[2 (22)]] ([[2 (22)]] [[2 (22)]]))]]
[[[[2 (22)]] ([[2 (22)]] [[2 (22)]])]]
[[[[[2 (22)]] [[2 (22)]] ([[2 (22)]] [[2 (22)]] ([[2 (22)]] [[2 (22)]]))]]]
[[[[[[2 (22)]] ([[2 (22)]] [[2 (22)]])]
([[2 (22)]] [[2 (22)]] ([[2 (22)]] [[2 (22)]]))]]]



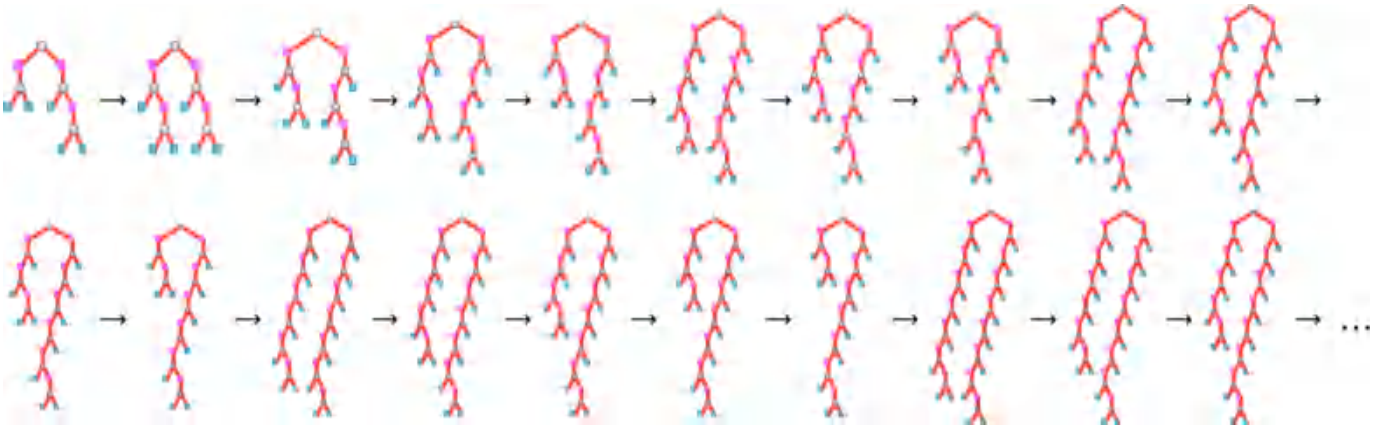


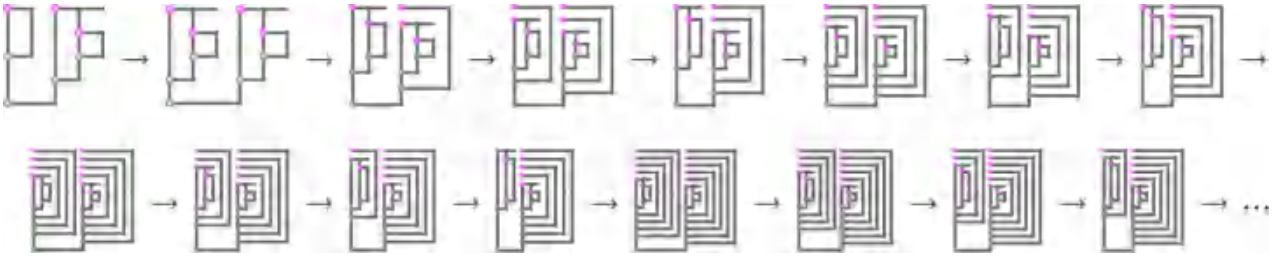


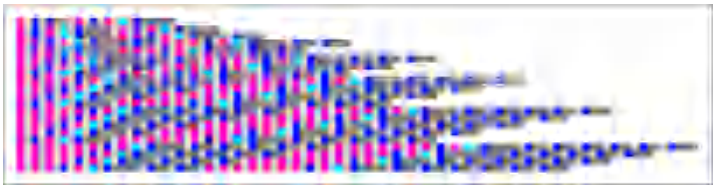


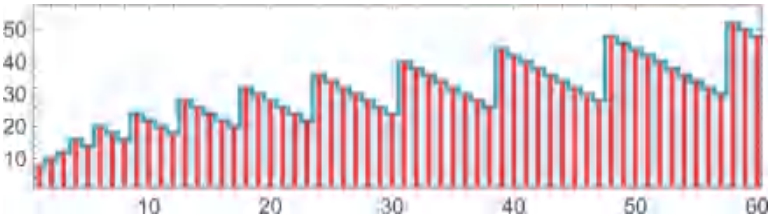
有简单的嵌套行为，例如

λ [1 [1]] [λ [1 [λ [2 [1]]]]]





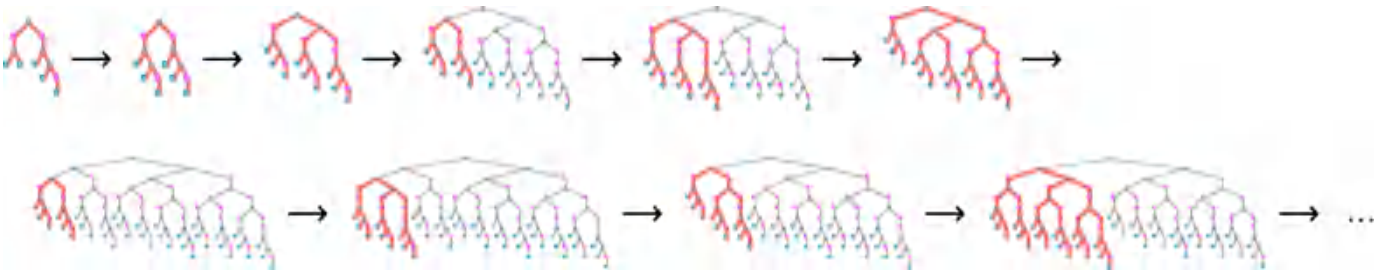


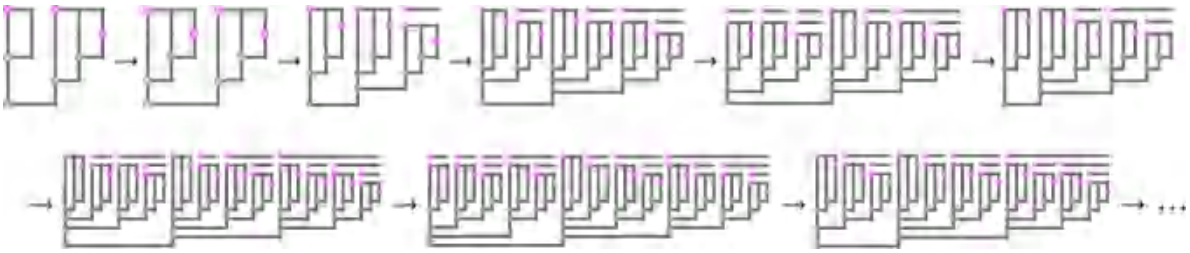


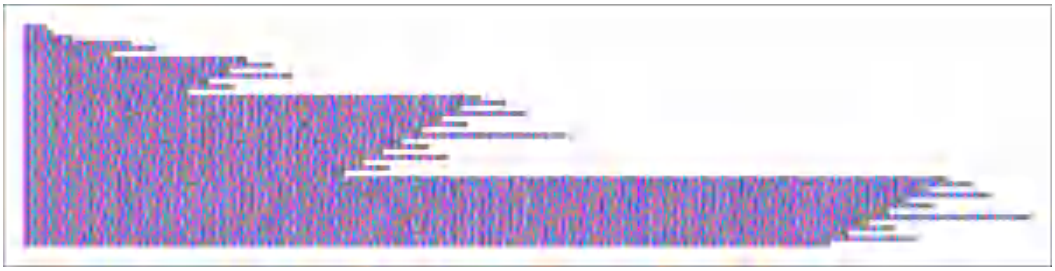
其中，包络线像 $\sqrt{t}$ 一样增长。

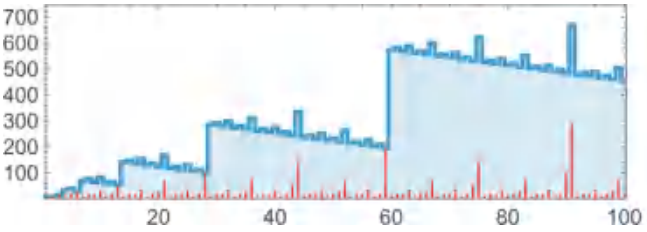
还有更复杂的嵌套行为

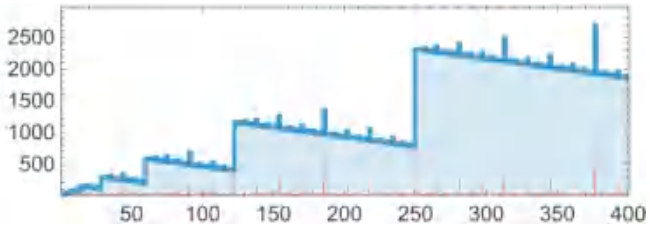
$$\lambda[1[1]][\lambda[1[1[\lambda[2]]]]]$$









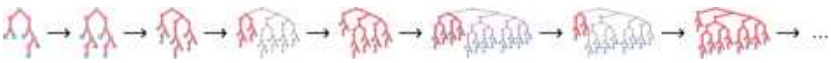


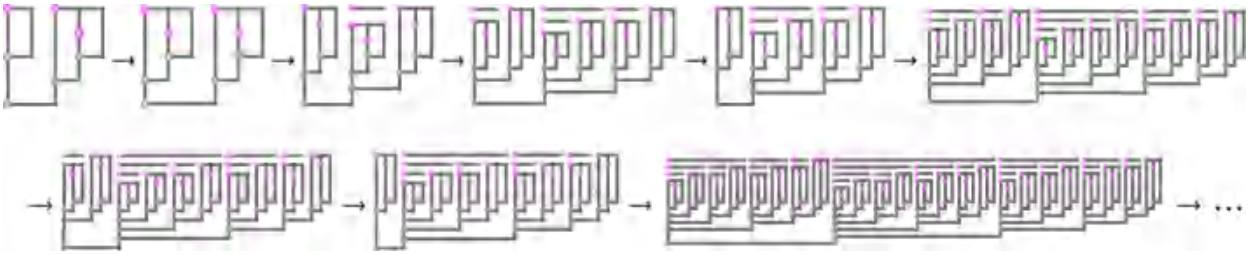
其中，包络线现在随 t 线性增长。

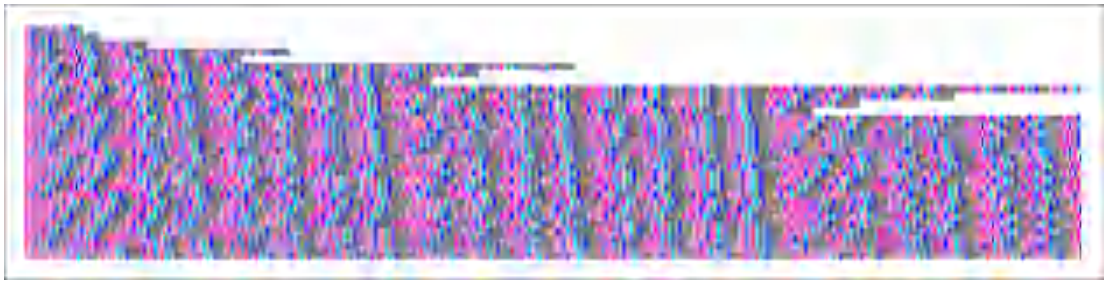
最后，有

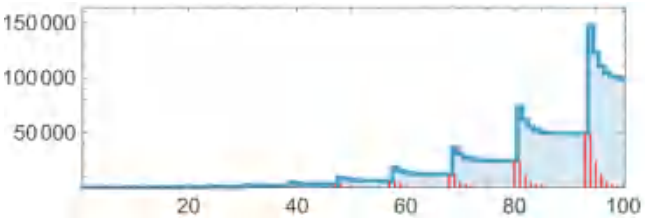
```
λ[1[1]][λ[1[λ[2][1]]]]
```

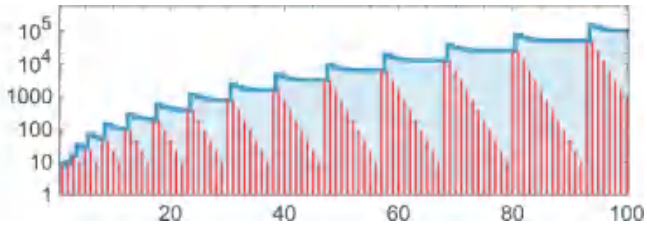
它展示了高度规则的 $2^{\sqrt{2t}}$ 增长：











结论是，对于大小不超过 8 的 lambda，只会发生嵌套行为，没有更复杂的情况。实际上，在这些所有情况下，都可以推导出在步骤 t 时的大小精确公式。

要大致了解这些公式是如何工作的，我们可以从序列开始：

$$\{1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, \ldots\}$$

这个序列最终由 嵌套递归递推关系 给出

```
a[1] = 1; a[t_] := a[t] = a[t - a[t - 1]] + 1
```

而这里的第 t^{th} 项就是：

$$\text{Floor}\left[\frac{1}{2} + \sqrt{2\,t}\right]$$

对于上面第一个嵌套序列，对应的结果是（对于 $t > 1$）：

$$\text{With}\left[\left\{k = \text{Ceiling}\left[\frac{1}{2}\left(\sqrt{8\,t - 15} - 9\right)\right]\right\}, 42 + k\,(11 + k) - 2\,t\right]$$

对于第二个，它是（对于 $t > 3$）：

```
Module[{m = 1, s, p, q}, While[2^(m+2) - 1 - m < t, m++];
  s = t - (2^(m+1) - m);
  {p, q} = Comap[{Floor, Ceiling}, (s - 1) / 2];
  If[s == 0, 6 * 2^m + 4,
    (18 * 2^m - 2) + (12 Sum[2^IntegerExponent[i, 2], {i, p}] - 4 p) +
     (-12 Sum[2^IntegerExponent[i, 2], {i, q}] - 2 q)]]
```

而对于第三个，它是（对于 $t > 1$）：

$$\text{With}\left[\left\{k = \text{Ceiling}\left[\frac{1}{2}\left(\sqrt{1 + 8\,(t-2)} - 1\right)\right] - 1\right\}, \left\{r = t - \left(\frac{1}{2}k\,(k+1) + 2\right)\right\}, 6 * 2^{k-r+1}\,(2^r + 1) - 2\right]$$

而且，确实非常令人惊叹，这些公式存在，却相对复杂。实际上，它们将我们带向了传统数学所能捕捉的边缘，而不是只能通过不可约计算才能发现。

更大的 Lambda 世界

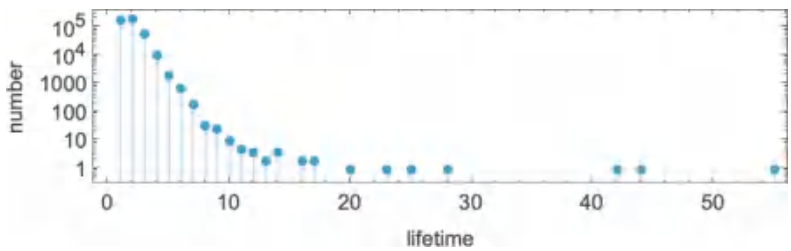
我们已经系统地研究了大小为 8 的 Lambda。那么，更大的 Lambda 会发生什么呢?

大小9

在大小为 9 时，有 454,283 种可能的 lambda。现在寿命的分布是：

lifetime	1	2	3	4	5	6	7	8	9	10	11	12
number	188 591	192 504	59 239	10 178	2039	699	187	35	27	10	5	4

lifetime	13	14	16	17	20	23	25	28	42	44	55	∞
number	2	4	2	2	1	1	1	1	1	1	1	748



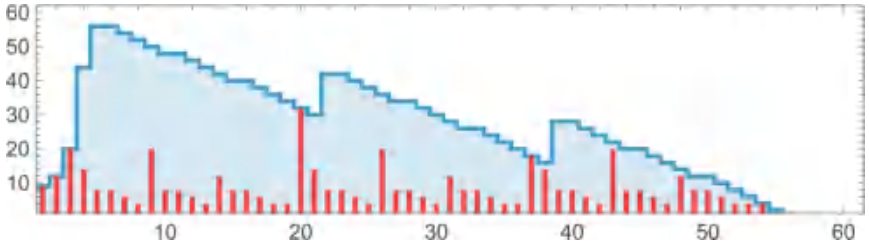
最长的有限寿命——55步——出现在

```
λ[1[1]][λ[1[1[1[λ[1]]]]]]
```

最终演变成：

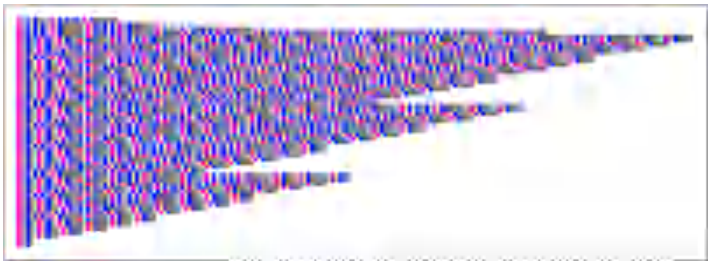
```
λ[1]
```

以下是每一步获得的 lambda 的大小：



但为什么这个过程会终止？这不容易看出。观察生成的树的序列，可以有些迹象表明分支如何重新排列然后消失：

数组图也并不怎么清晰：



正如通常那样，似乎没有什么“可识别的机制”可以解释发生的事情；它“只是那样”。

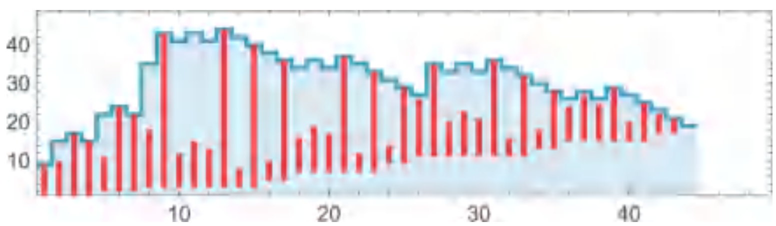
在长度为 9 的 lambda 中，寿命第二长（44 步）的是

λ[1[1][1]][λ[λ[2[2[1]]]]]

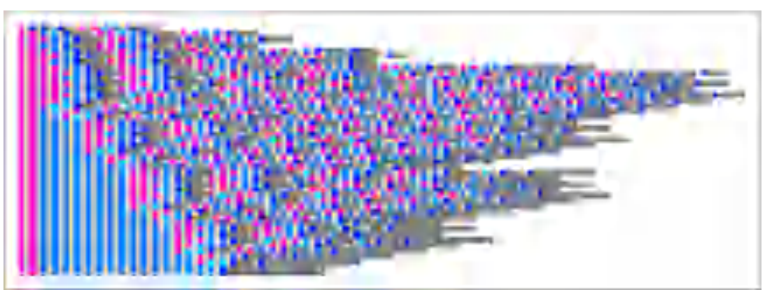
最终演变成

λ[λ[2[2[2[2[2[2[2[2[2[2[2[2[2[2[2[2[1]]]]]]]]]]]]]]]]]]]

（这恰好对应于我们之前讨论的表示中的整数16）：



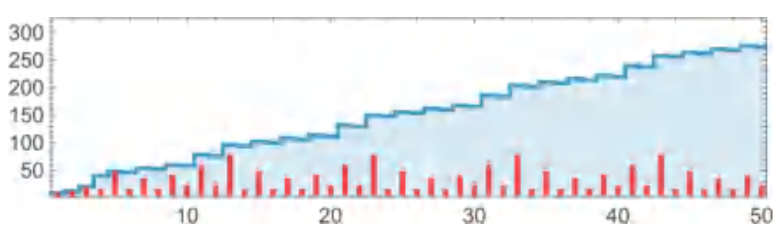
以“数组”形式观察，至少隐约可见“机制”：我们看到在演化过程中，“纯应用”的数量呈稳定（线性）增加，而没有任何λ来“驱动事物”：

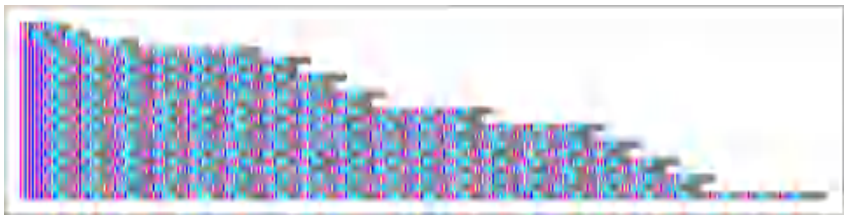


在 748 个永无止境生长的 size-9 lambda 中，大多数最终具有周期性的生长模式。其中生长周期最长（10 步）的是

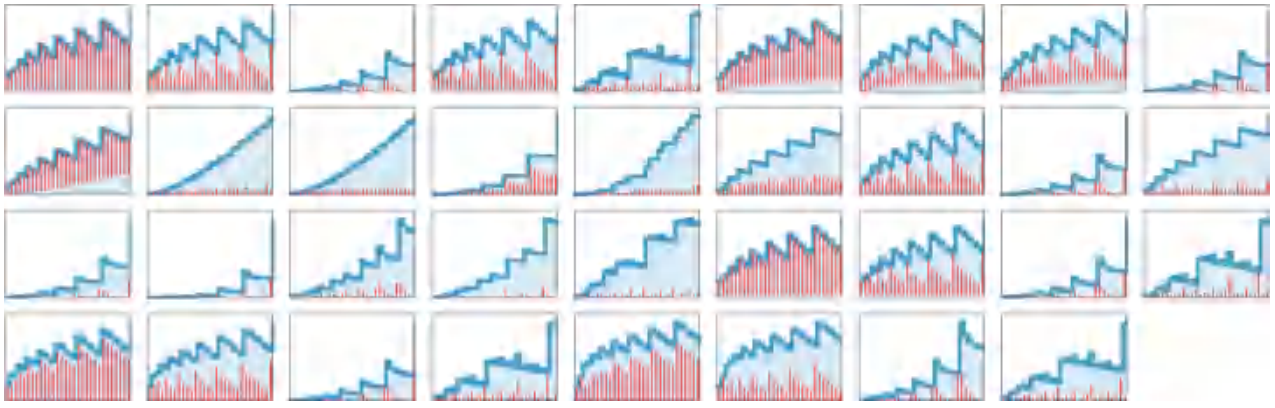
λ[1[1]][λ[λ[1[2[2[2]]]]]]

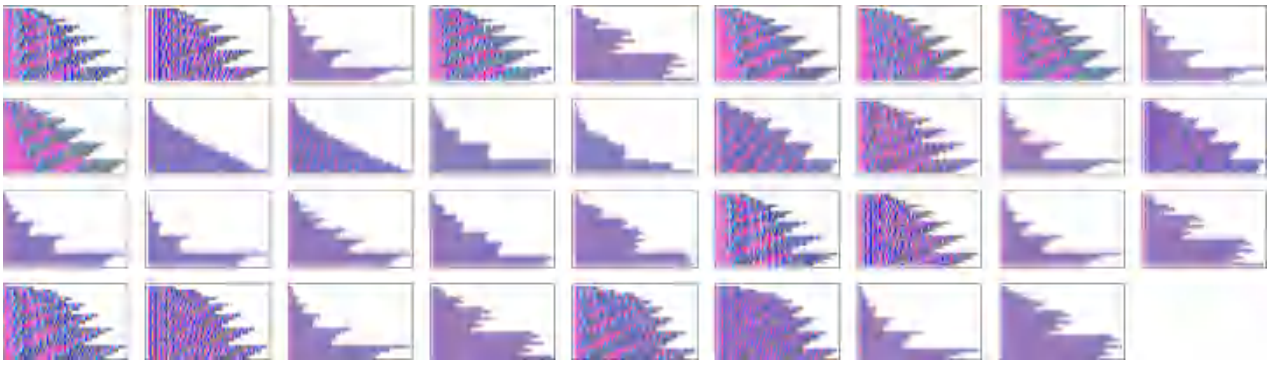
其生长模式相当平凡：



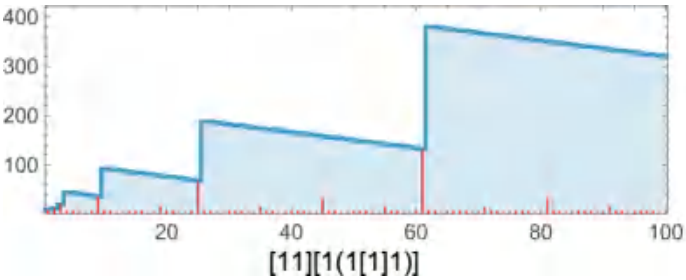


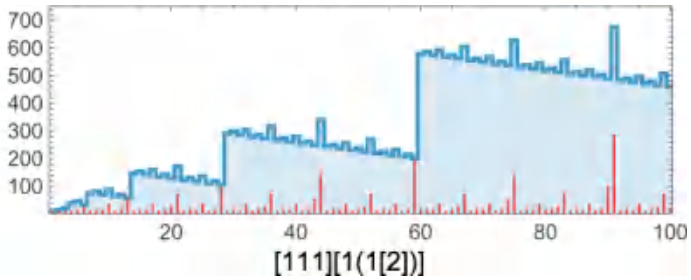
只有 35 个 lambda 具有更复杂的生长模式：



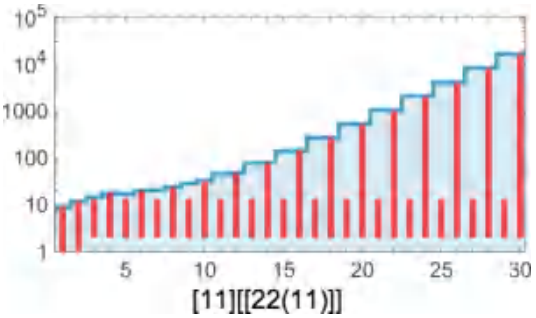


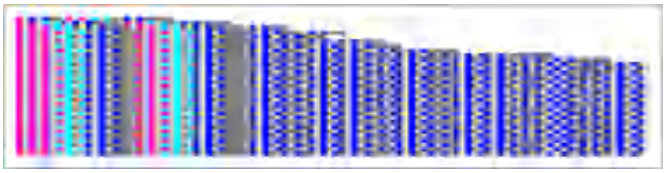
许多的行为与大小8时所见相似。可以有嵌套的生长模式，其包络呈线性：



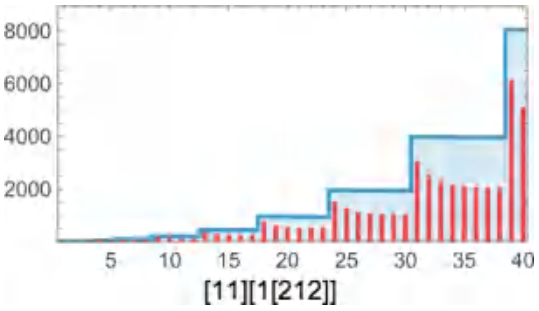


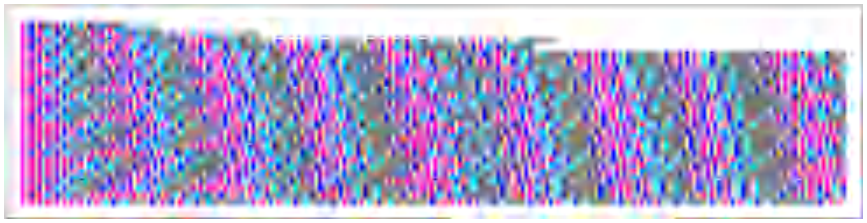
可以有最终纯粹的指数 $2^{t/2}$ 增长：



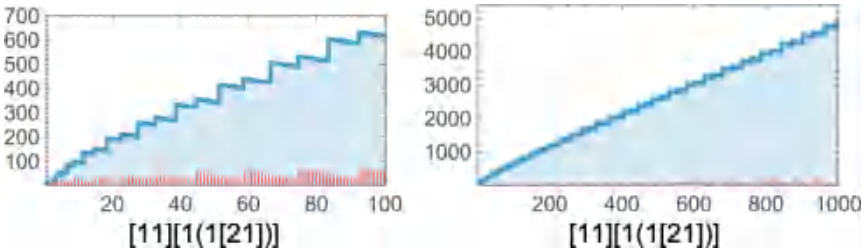


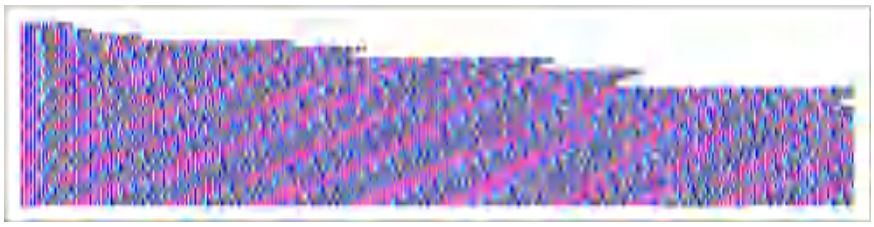
也可能存在增长速度较慢但仍呈指数型的增长，在这种情况下 $2^{\sqrt{2t}}$ ：



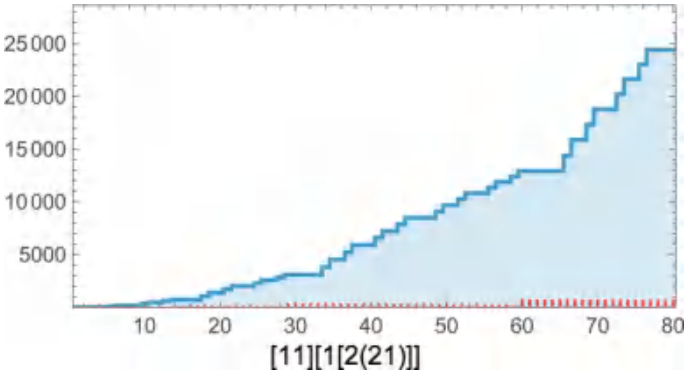


有一种增长，最初看起来可能很复杂，但最终会变成线性：

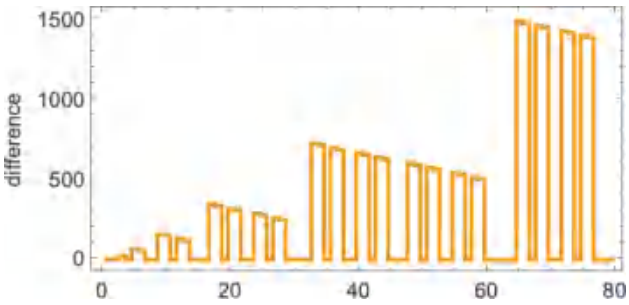




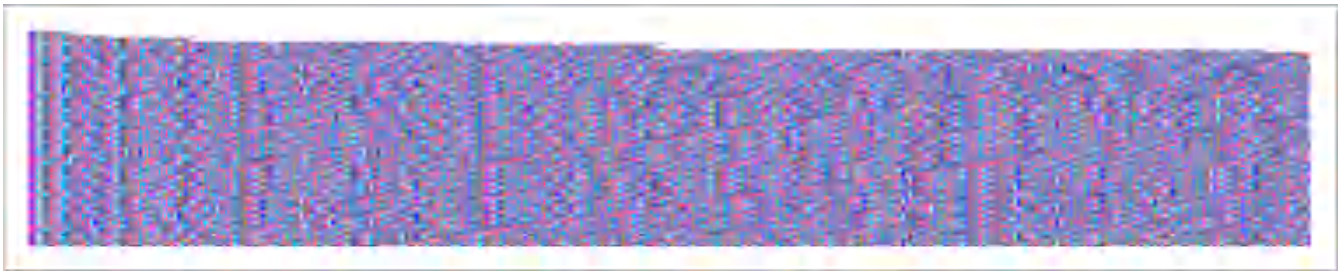
还有一种看似不规则的增长：



但计算差分会揭示出嵌套结构



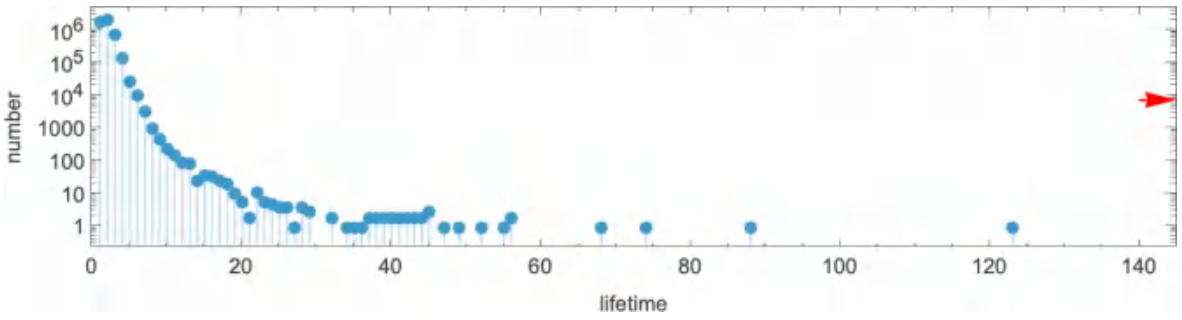
即使在这种情况下，lambda 的详细模式中没有明显的嵌套结构：



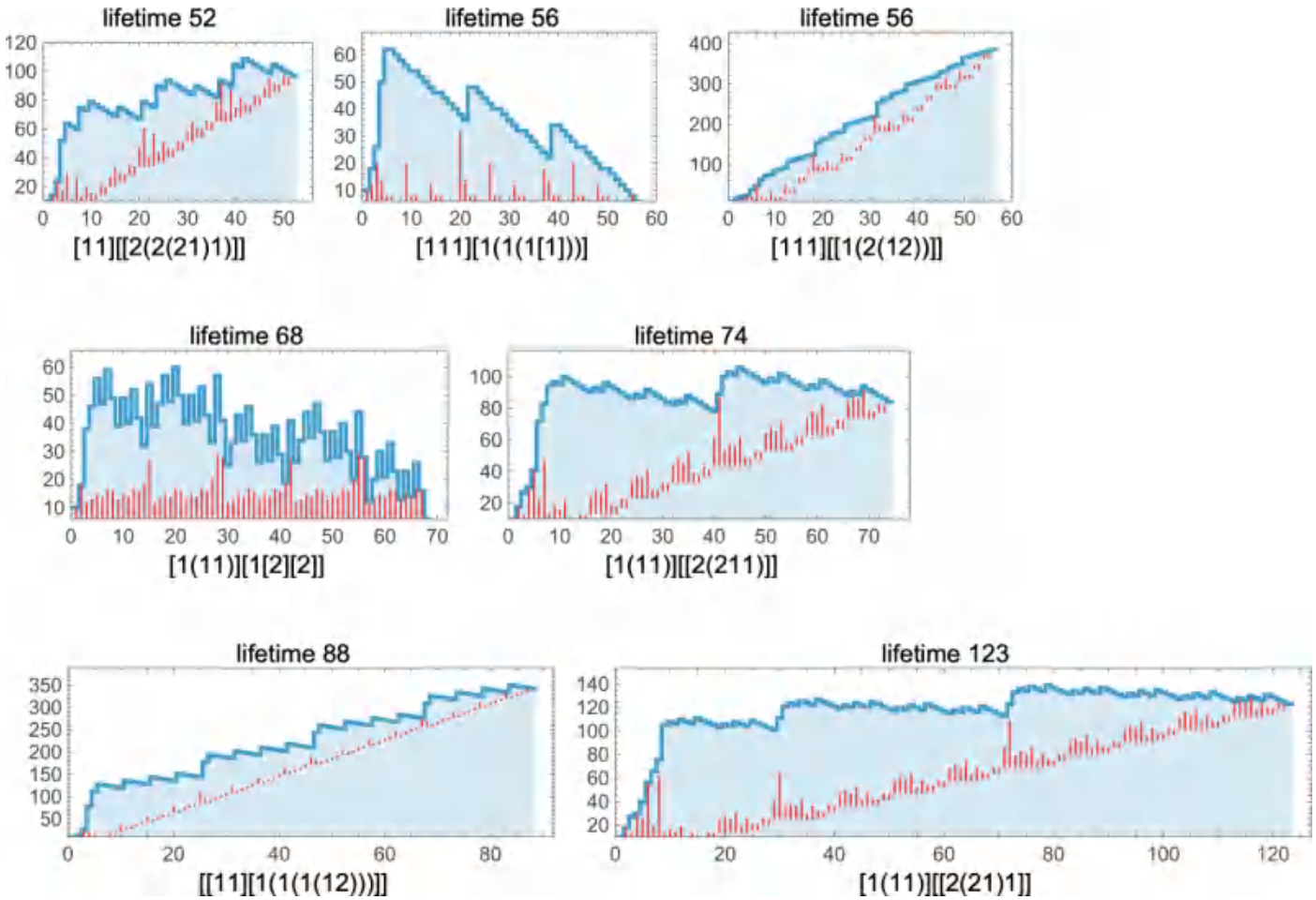
9号尺寸的内容就到此为止。

10号尺寸

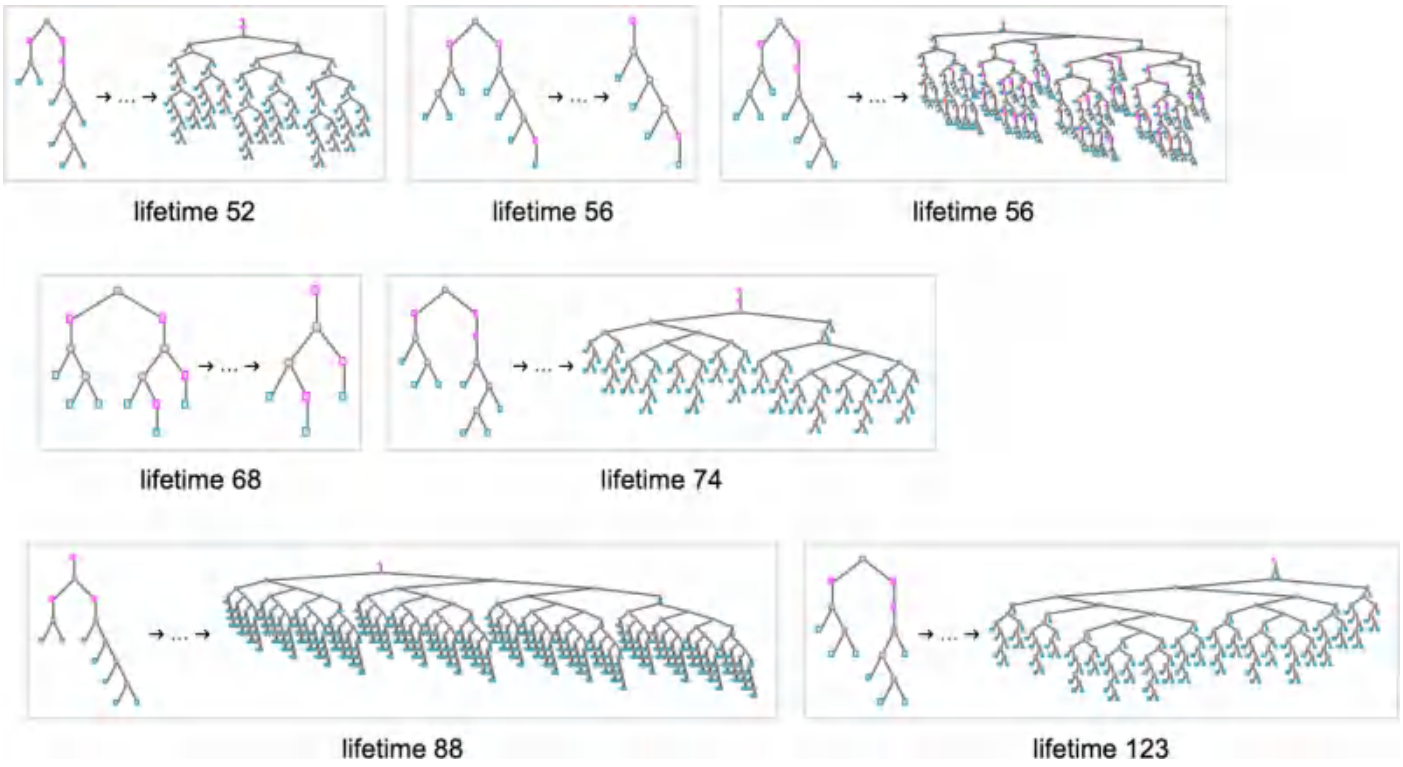
那么 10 号尺寸呢？现在有 5,159,441 种可能的 lambda。其中 38%是惰性的，0.15%不会终止。寿命的分布是：



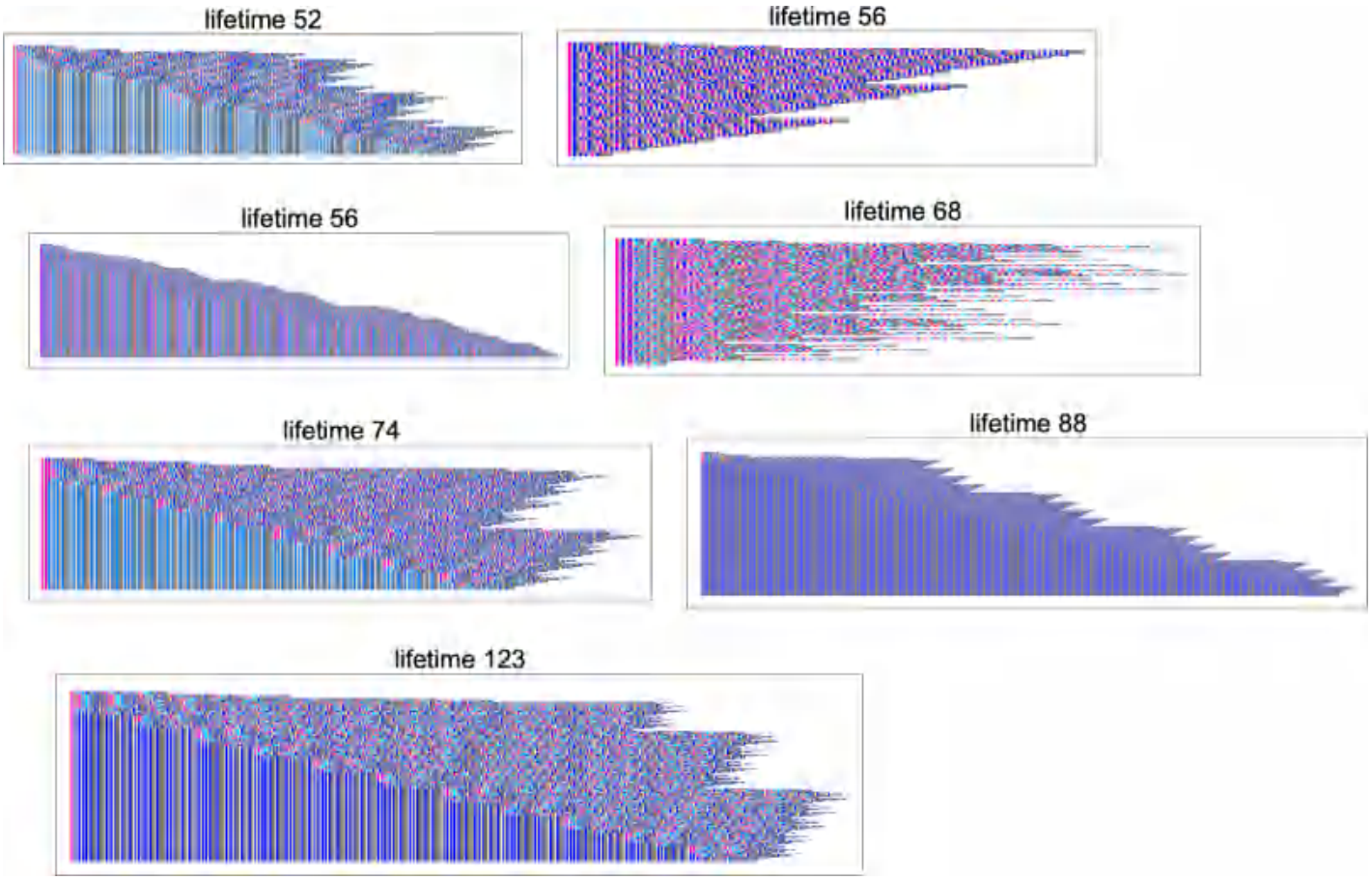
一些具有较长生命周期的 lambda 示例有：



其中一些会以较小的 lambda 终止；生命周期为 88 的那个产生了一个大小为 342 的 lambda，而生命周期为 56 的第二个产生了一个大小为 386 的 lambda：



这些方法在某种程度上有所不同。在某些情况下，行为看起来相当系统化，至少在一定程度上可以确定它们最终会停止；在另一些情况下则完全不明显：

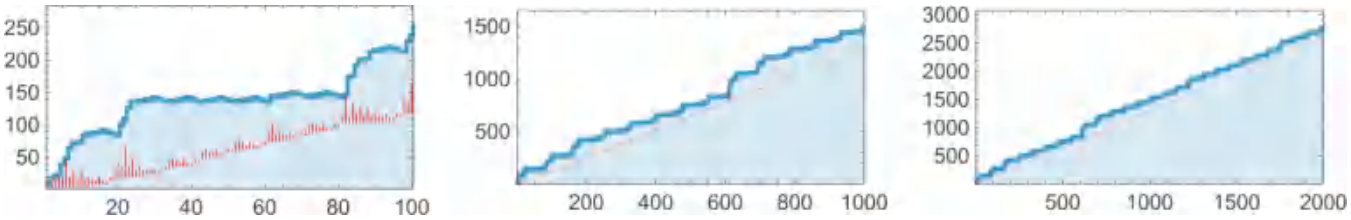


值得注意的是，许多（尽管不是全部）这些演化显示出线性增加的“死亡”区域，这些区域完全由应用组成，没有任何实际的λ——因此，例如，在生命周期-123的情况下，最终的固定点是：

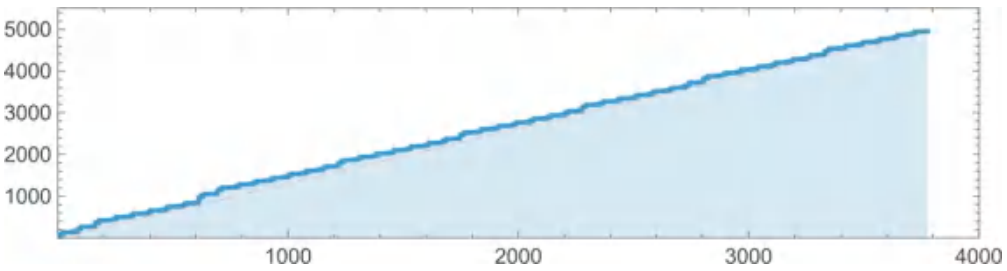
其他情况怎么样？嗯，有惊喜。比如考虑：

```
λ[1[1][1]][λ[λ[1[2[2[1]]]]]]
```

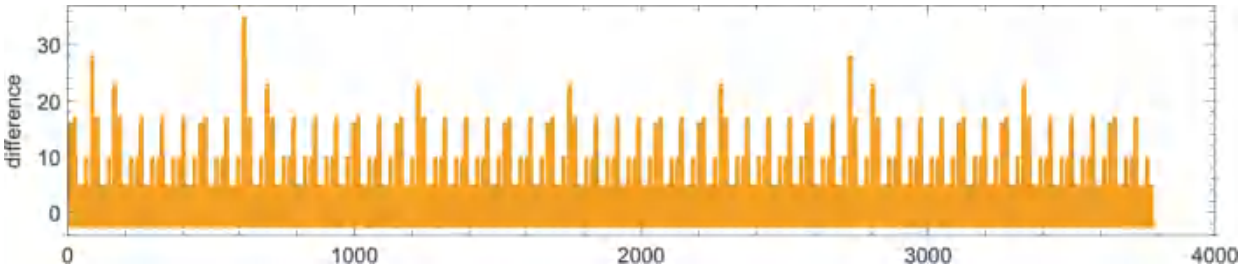
运行这个，比如运行2000步，看起来似乎只是均匀增长：



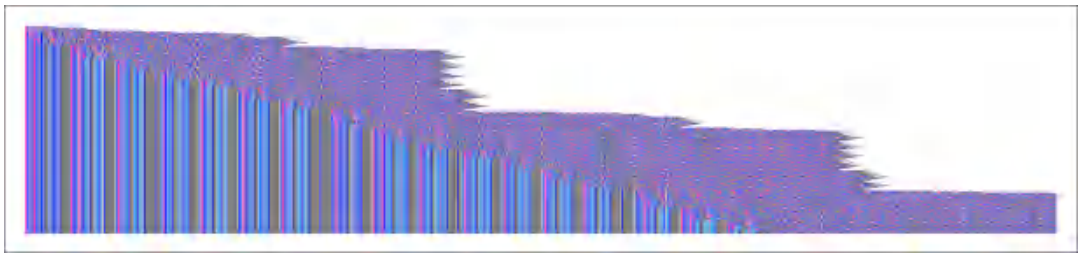
但是，突然之间，在经过 3779 步之后，出现了一个惊喜：它终止了——留下一个大小为 4950 的 lambda：



观察连续的大小差异并没有给出任何明显的“提前预警”终止：

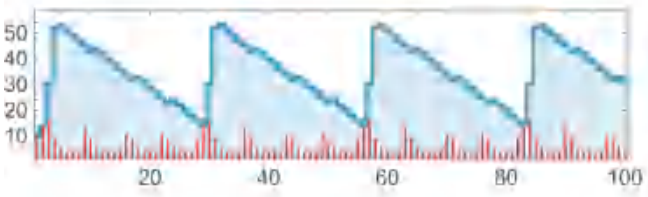


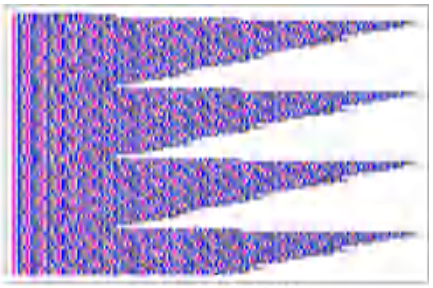
但如果一个人查看实际生成的 lambda 序列（这里仅展示 200 步），我们会看到似乎有些明显的“无效”应用积累：



而且，确实，在下一节中，我们将看到一个更长寿的例子。

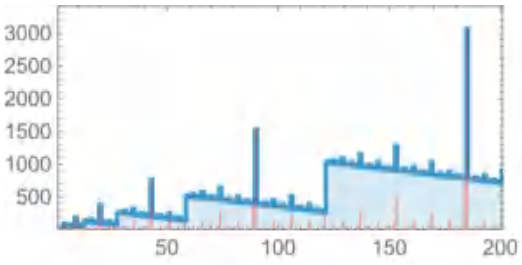
尽管有时需要判断 lambda 的计算是否会终止，但在很多情况下这是显而易见的。一个例子是当计算链最终变得周期性时。在大小为 10 时，周期比之前更长；这里有一个周期为 27 的例子：



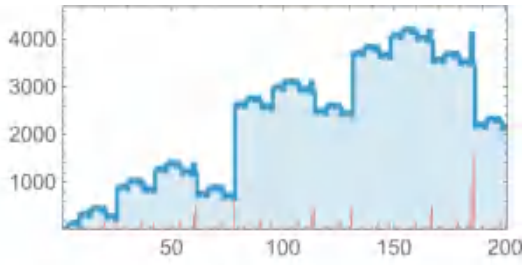


[11][1(1[1])11]

还有一种增长，其大小序列具有明显的嵌套结构，并有一个线性增长的包络：

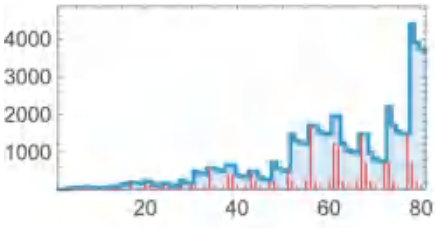


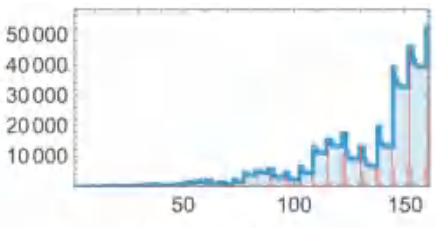
[11][1[2(2[3])]]

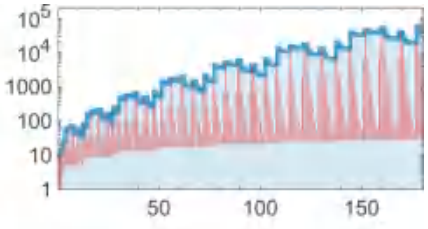


[11][1(1(1[[3]]))]

然后是嵌套结构，但叠加在 $2^{\sqrt{t}}$ 增长之上：





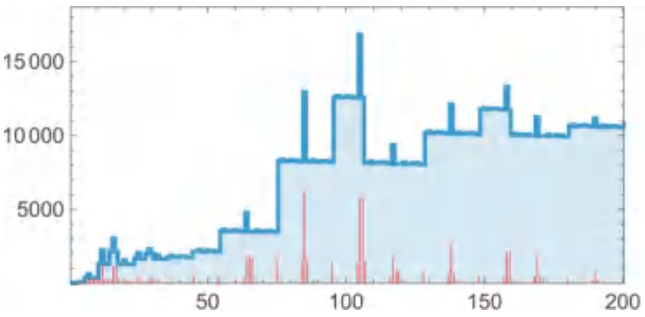


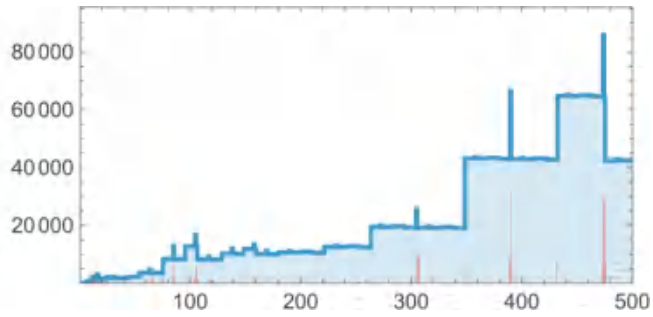
[11][[1(2(2[3]))]]

大小11

在大小为 11 时，存在 63,782,411 种可能的 lambda。大多数行为看起来和之前非常相似。但也有一些惊喜。其中之一是：

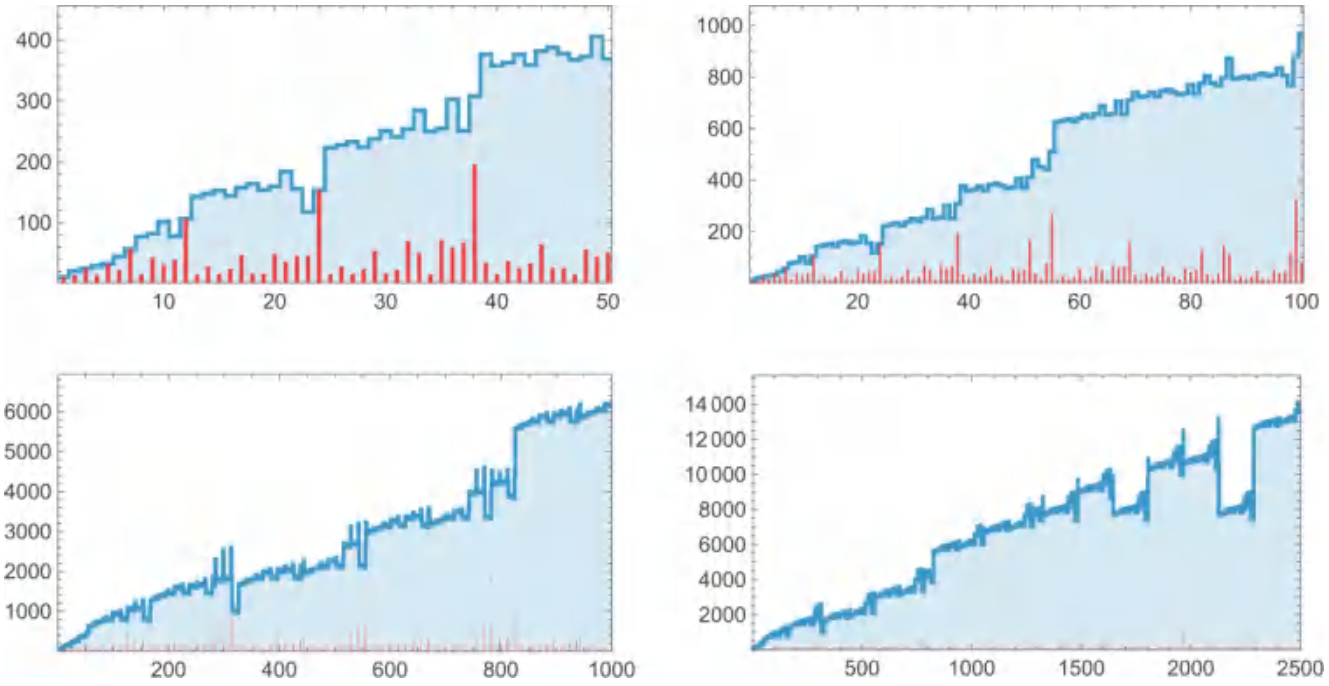
```
λ[1[1]][λ[1[λ[2[2[λ[2]]][2]]]]]
```



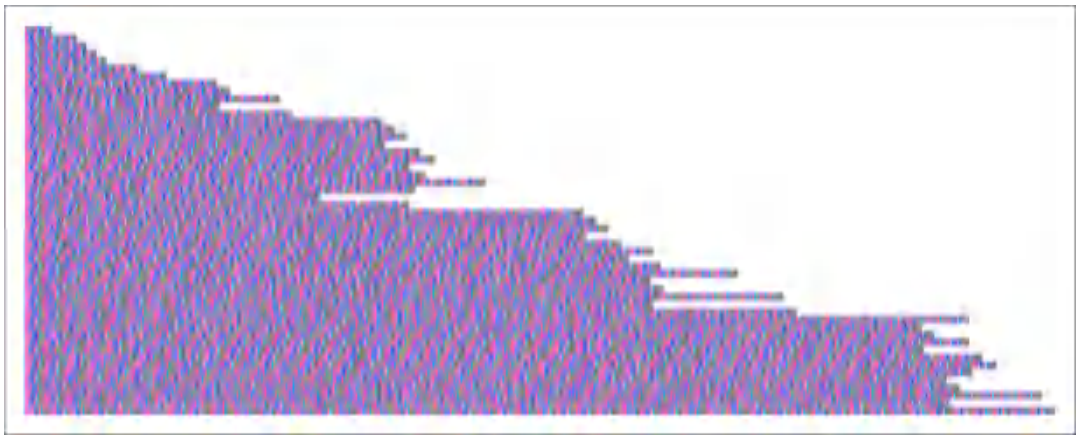


还有：

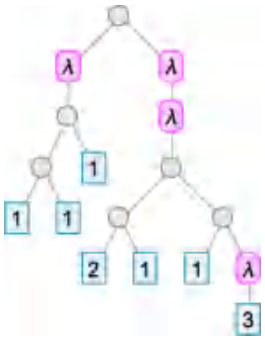
```
λ[1[1][1]][λ[λ[2[1][1[λ[3]]]]]]
```



是的，似乎没有任何整体规律性，在生成的 lambda 序列的详细序列中也没有出现任何规律性：



据我所知， 像许多其他简单的计算系统 —从规则 30 开始—the 相当简单的 lambda



只是不断地产生不可预测的行为。

不可判定性问题

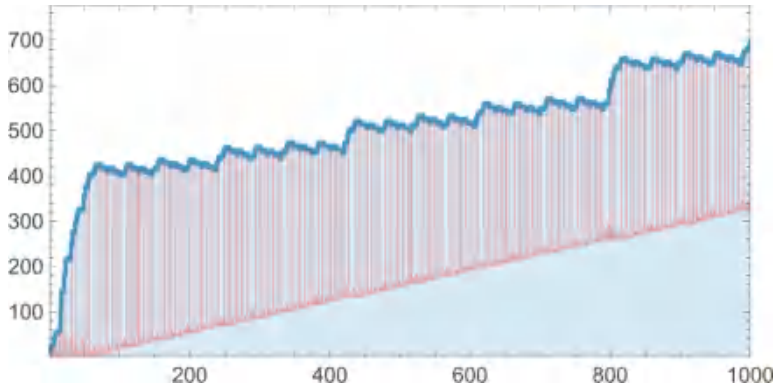
我们如何知道某个特定 lambda 的求值链是否会终止？嗯，我们可以运行一定数量的步骤，看看是否在此之前就终止了。也许到那时，我们会看到明显的周期性或嵌套行为，从而明确知道它永远不会终止。但通常情况下，我们没有办法知道需要运行多长时间才能确定它是否会终止。

这是一个非常典型的例子，展示了即使在简单的计算系统中也普遍存在的计算不可约性 。通常情况下，这会让我们得出结论：判断给定的 lambda 求值是否会终止的问题必须被视为不可解 ，因为不存在长度有限的计算可以保证回答这个问题。

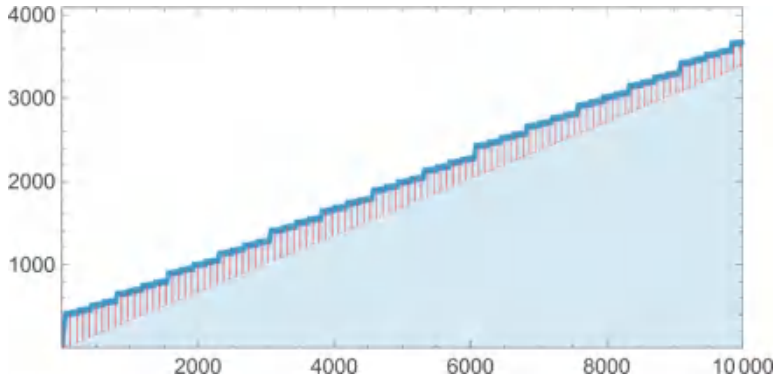
但在实践中会发生什么？我们之前已经看到，至少对于足够小的 lambda 来说，解决终止问题并不太难。但到了 10 的大小就已经出现了问题。例如：

```
λ[1[1[1]][1]][λ[λ[2[2[1]]]]]
```

评估的前 1000 步导致 lambda 出现一系列大小，看起来基本上只是均匀增加的：

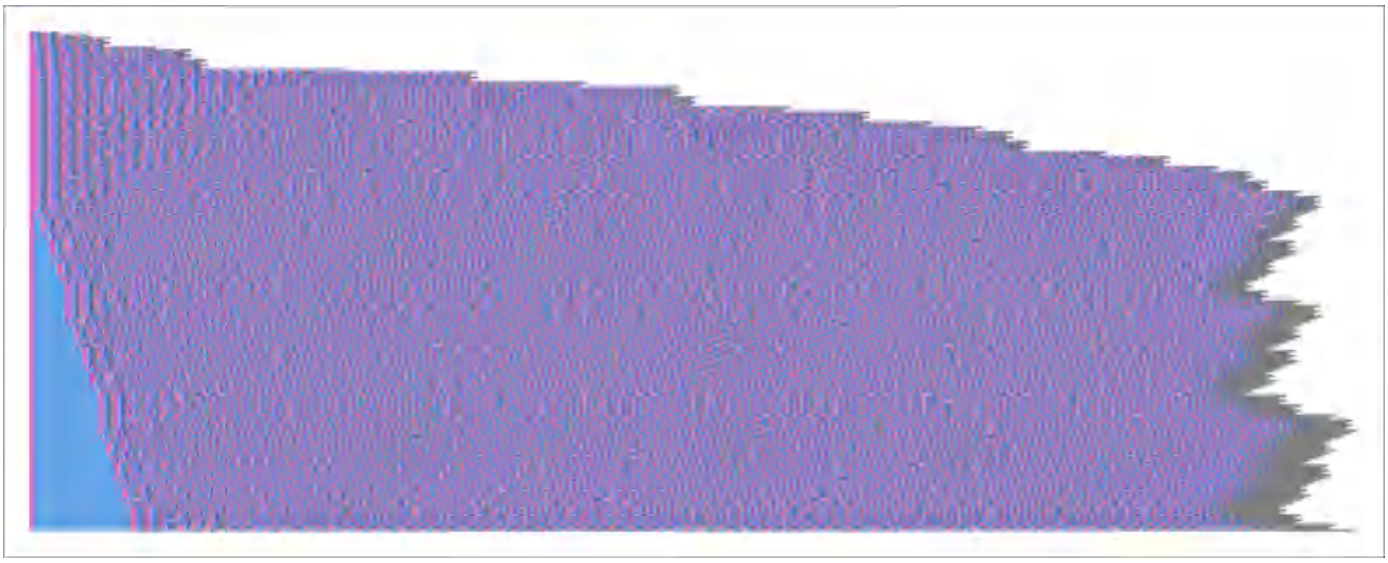


经过 10,000 步后情况相同，其大小大致增长如 $t/3$：



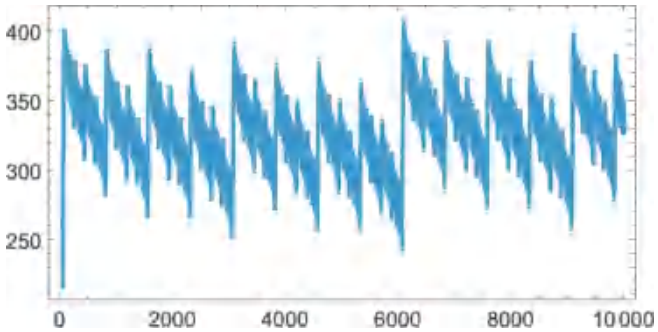
但如果我们观察实际底层 lambda 的结构——即使只是 200 步——我们会看到有一个“死亡区域”开始

形成：



继续到500步，死亡区域继续大致线性增长：

通过对大小序列进行“去趋势”处理，即减去 $t/3$，我们得到：



这里存在明确的规律性，这表明我们实际上可能在这个特定情况下识别出一个“计算可约性区域”——并且能够计算出结果，而无需显式地运行每一步。

是的，事实上这是正确的。让我们再次看看我们正在处理的 lambda 的形式：

λ[1[1[1]][1]][λ[λ[2[2[1]]]]]

第二部分实际上是（"Church 数字"）我们用来表示数字 2 的表示方法。将这个 2 称为整个 lambda，它根据以下方式评估：

λ[1[1[1]][1]][2]⟶2[2[2]][2]

通常你可能认为会做整数上的算术运算。但在这里我们看到的是不同的事物：我们看到整数实际上被应用于整数。那么，例如，

2[2]

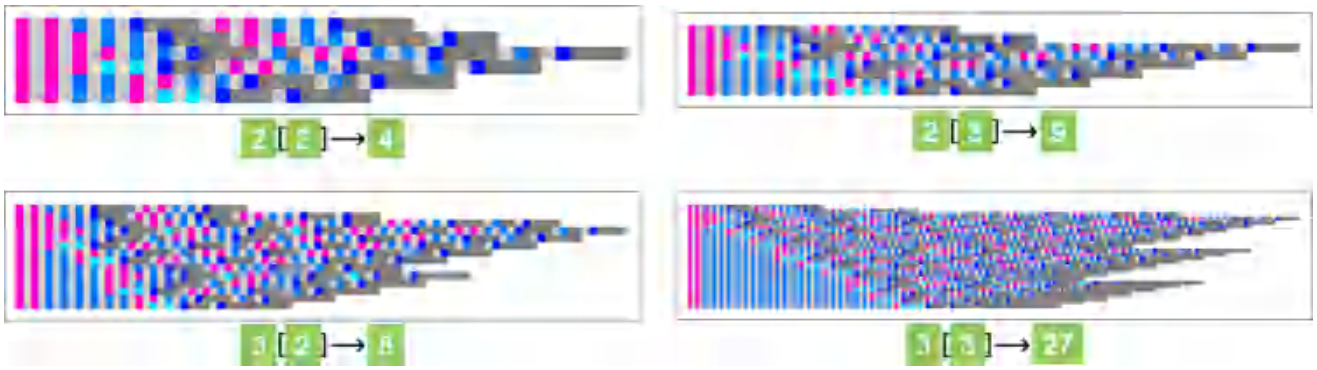
或者

λ[λ[2[2[1]]]][λ[λ[2[2[1]]]]]

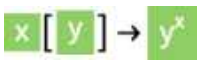
等于什么？反复进行β简化得到：

```
λ[λ[2[2[1]]]][λ[λ[2[2[1]]]]]
λ[λ[λ[2[2[1]]]]][λ[λ[2[2[1]]]]][1]]
λ[λ[λ[λ[2[2[1]]]]][2][λ[λ[2[2[1]]]]][2][1]]]
λ[λ[λ[3[3[1]]]][λ[λ[2[2[1]]]]][2][1]]]
λ[λ[2[2[λ[λ[2[2[1]]]]][2][1]]]]
λ[λ[2[2[λ[3[3[1]]]][1]]]]
λ[λ[2[2[2[2[1]]]]]]
```

换句话说 2[2] 给出 4 。这里有一些其他例子：



一般来说：



现在我们可以“解码”我们最初的 lambda。由于

等于

这意味着

就是：

换句话说，尽管最初看起来可能不是这样，我们原始的 lambda 最终会终止求值，给出一个大小为 65539 的 lambda——而完成这个过程大约需要 200,000 步。（有趣的是，这个整个设置与我在大约 30 年前研究的一种形式为 e[x_][y_]→x[x[y]] 的类似组合系统极其相似我曾研究过的某种东西。）

我们的简单 lambda 终止所需的时间有些令人惊讶。实际上，鉴于我们所看到的，构建能够终止但需要更长时间的 lambda 是直截了当的。所以，例如

计算结果为 *n* 重“tetration”

并且需要大约那么多步来完成。

在下一节中，我们将看到如何设置 lambda，使其不仅能计算指数运算，还能计算 Ackermann 函数

，以及任何级别的超运算，这些超运算递归定义为

```
h[1][x_, y_] := x + y

h[n_][x_, y_] := Nest[h[n - 1][x, #] &, x, y - 1]
```

(where h[1] 是 Plus， h[2] 是 Times， h[3] 是 Power， h[4] 是指数运算，等等。)

好的，所以存在 lambda（实际上甚至相当简单的 lambda）其求值会终止，但需要用超运算来描述其步骤数量。那么对于完全不终止的 lambda 呢？我们无法明确追踪 lambda 在无限步骤中的行为。那么我们如何能确定一个 lambda 不会终止？

嗯，我们必须构建一个证明 。有时候这相当直接，因为我们能轻易地识别出 lambda 内部正在发生一些根本上的简单事情。例如：

```
λ[1[1]][λ[λ[2[2]]]]
```

其求值过程中的连续步骤是（每个 beta 约化涉及的元素被突出显示）：

```
λ[1[1]][λ[λ[2[2]]]]
λ[λ[2[2]]][λ[λ[2[2]]]]
λ[λ[λ[2[2]]][λ[λ[2[2]]]]]
λ[λ[λ[λ[2[2]]][λ[λ[2[2]]]]]]
λ[λ[λ[λ[λ[2[2]]][λ[λ[2[2]]]]]]]
λ[λ[λ[λ[λ[λ[2[2]]][λ[λ[2[2]]]]]]]]
```

这里发生的事情非常明显，而且会永远持续下去。但如果我们愿意，我们可以构建一个正式的证明来展示将要发生什么——比如使用数学归纳法来关联每一步。

确实，只要我们能识别出一个（类似干细胞）的核心“生成器”在重复，我们就可以预期能够使用归纳法来证明不会出现终止。还有许多其他我们见过的案例，我们可以判断存在足够的规律性——比如在连续 lambda 的模式中——以至于“从视觉上明显”不会出现终止。尽管通常会有许多细节需要处理，但在这种情况下，我们可以预期一条直接路径，通过数学归纳法得出形式化证明。

在实践中，“视觉直观性”有时难以识别的一个原因是 lambda 的增长速度过快，以至于难以持续可视化。这种快速增长的根源在于 beta 约简最终会反复复制越来越大的子表达式。如果这些复制在某种程度上形成了一个简单的树，那么人们可以再次期待进行基于归纳的证明（通常基于各种类型的树遍历）。但如果开始攀登超运算的层次 ，正如我们所知 lambda 可以做到的那样，情况可能会变

得更加复杂。例如，最终可能会出现每个树的每个叶节点都在生成新树的情况。

一开始，显然并非显而易见，归纳法的过程会是证明一个“关于无穷的命题”（例如某个特定的 lambda 永远不会终止，即使经过无限多步骤）。但归纳法在数学中有着悠久的使用历史，并且一个多世纪以来一直被确立为 Peano 算术的公理——这个公理系统基本上在基础数学中（至少原则上）被普遍使用。从某种意义上说，归纳法表明，如果你总能在序列中找到下一步（比如在整数数轴上），那么你就能在那个序列中走得更远，甚至无限远。但如果你所处理的东西不能轻易地“展开”成一个简单的（类似整数的）序列呢？例如，如果你所处理的是到处都在生长新树的树，那普通的归纳法就不足以“带你到达所有叶子”。

但在 集合论 （通常被认为是当前数学的终极公理系统）中，存在超越普通归纳法的公理，允许使用超限归纳法的概念。通过超限归纳法（可以“遍历”所有可能的有序集），可以在“树始终生树”系统中“到达叶节点”。

这意味着，虽然足够快速增长的 lambda 的非终止性可能无法在皮亚诺算术中证明（即无法通过普通归纳法证明），但在集合论中仍然可能是可证明的。然而，不可避免地， 集合论也会受到限制 ，存在 lambda 的非终止性它无法证明，这时就需要引入新的公理才能产生证明。

但我们能否明确看到存在这些问题的 lambda 呢？如果这里我们已经考虑的一些 lambda 实际上就是例子，我并不感到意外。但这很可能是一个非常困难的事情要证明。一个更容易（但仍然困难）的方法是明确构造一个 lambda 家族，在皮亚诺算术中无法证明它们都终止。

考虑稍微复杂的定义：

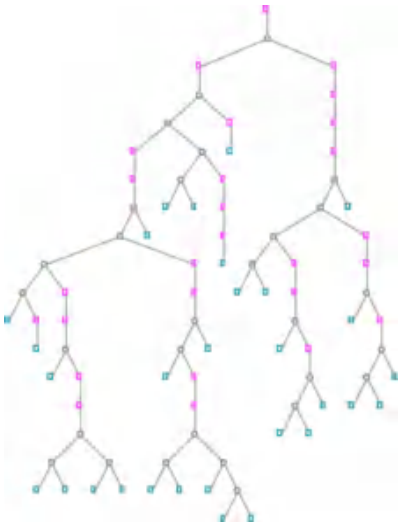
```
g[n_] :=
  Map[First, NestWhileList[{f[#] - 1, Last[#] + 1} &, {n, 3}, First[#] > 0 &]]
```

```
f[{0, _}] = 0;
f[{n_, k_}] :=
  Total[MapIndexed[#1 k^f[{#2[[1]] - 1, k}] &, Reverse[IntegerDigits[n, k - 1]]]]
```

$g[n]$ 是所谓的 Goodstein 序列 ，已知它们最终会达到 0 的命题无法仅通过普通归纳法证明，即无法在皮亚诺算术中证明。

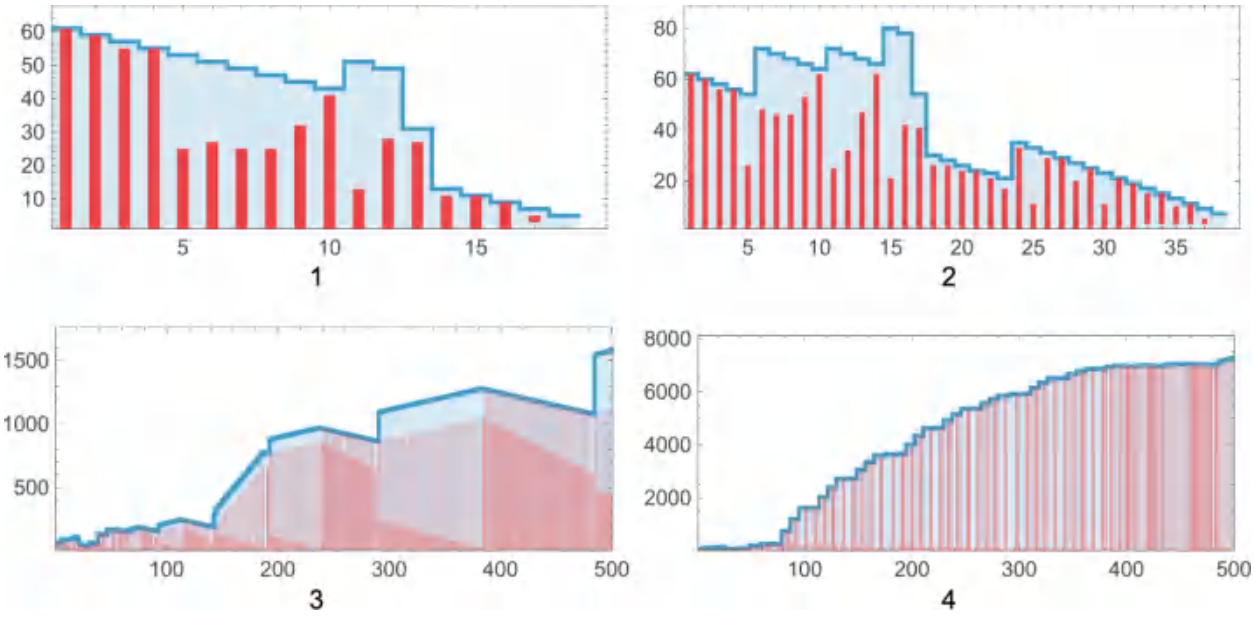
这里有一个 lambda——它甚至相对简单——被构建用来计算 Goodstein 序列的长度：

```
[[ ] [2[1] [ [2[ [32(21) ] ] ] ] [ [2[ [32(2(21) ) ] ]1] ]1] ] (21[ [ [3] ] ] ] [1] ]
  ] ] ] [42] [3[ [32] ] ] ] [ [3[312] ] [3] ] ] ] ]
```



输入一个整数 *n* 的表示，这个 lambda 将会计算相应序列 $g[n]$ 的长度（即需要多少项才能达到 0）。这个长度将是有限的，当且仅当 lambda 的求值终止。换句话说，所有这种类型的 lambda 求值都终止的断言，等价于所有 Goodstein 序列最终都会达到 0 的断言。

但如果我们求值这种类型的 lambda，实际上会发生什么呢？以下是 n 的前几个值的结果：



对于 $n = 1$，该过程在 18 步后终止，最终结果为 λ[λ[2[2[1]]]] ——我们表示整数 2 的方式——反映了 $g[1] = \{1, 0\}$ 的事实。对于 $n = 2$，经过 38 步后得到整数 4，反映了 $g[2] = \{2, 2, 1, 0\}$ 的事实。对于 $n = 3$，需要更多步数，但最终结果是 6（反映了 $g[3] = \{3, 3, 3, 2, 1, 0\}$ 的事实）。但 $n = 4$ 又如何呢？众所周知，在这种情况下，该过程最终也必须终止。但它必须花费非常长的时间——因为最终结果已知为 $3 \times 2^{402\,653\,211} - 1$。如果我们继续进行，规模——以及相应的时间——会迅速变得更大、更大。

这当然不是“通过皮亚诺算术逃避可证明性”的唯一方式，但它展示了这种情况可能的样子。

我们应该记住，一般来说，一个 lambda 表达式的求值是否终止是一个本质上不可判定的问题——这意味着没有任何有限的计算能够保证回答这个问题。但是，证明某个特定的 lambda 表达式的求值不终止的问题则有所不同。如果 lambda 在其无限的生命周期中“没有产生太多新颖性”（比如它只是以一种重复的方式行为），那么我们就能够给出一个相当简单的非终止证明。但是，lambda 所做的事情越基本越多样化，证明就会越复杂。关键在于——本质上由于计算不可约性——必然存在行为任意复杂和多样化的 lambda。但在某种意义上，任何给定的有限公理系统只能期望捕捉到一定的“多样性水平”，因此只能对一定范围内的 lambda 证明其非终止。

为了更具体一些，我们可以设想一种构造，它能引导我们得到一个 lambda，其终止性在一个给定的公理系统中无法被证明。这始于这样一个事实：lambda 的计算普遍性意味着任何计算都可以被编码为某个 lambda 的求值。因此，特别地，存在一个 lambda，它系统地生成给定公理语言中的语句，并确定这些语句是否为该公理系统中的定理。然后我们可以设置它，使得如果任何时候发现不一致——在那种意义上即一个定理及其否定同时出现——那么过程将停止。换句话说，lambda 的求值在某种意义上系统地寻找公理系统中的不一致性，一旦找到就终止。

这意味着证明 lambda 的计算永远不会终止，等同于证明公理系统的一致性。但这是一个基本事实：任何给定的公理系统都无法证明自身的自洽性；需要更强大的公理系统才能做到这一点。因此，这意味着我们永远无法证明我们设定的 lambda 的计算会终止——至少无法在它所基于的公理系统内部证明。

实际上，发生的情况是我们已经设法让我们的 lambda 在其无限的生命周期中扫描了公理系统能够做的一切，结果公理系统无法“进一步”说任何关于 lambda 行为“更大”的事情。这是由于 lambda 的计算通用性，使得我们可以将整个公理系统的“元数学”包装进单个 lambda（无限）的行为中。但结果是，对于任何给定的公理系统，原则上我们可以产生一个 lambda，我们知道该 lambda 的非终止性在公理系统内部是不可证明的。

当然，如果我们扩展我们的公理系统，增加一个额外的公理“这个特定的 Lambda 永远不会终止”，那么我们就能轻易地证明非终止性。但 Lambda 的计算通用性意味着（在哥德尔定理的一个类比中），没有任何一个有限的公理集合能成功地让我们为所有可能的 Lambda 得到有限的证明。

是的，虽然我们提出的构造方法为我们提供了一种特定的方法来生成“逃逸”给定公理系统的 Lambda，但这些肯定不会是唯一能这样做的 Lambda。在比如皮亚诺算术中不可证明非终止性的最小 Lambda 会是什么？我们上面基于 Goodstein 序列的例子已经相当小了。但毫无疑问，仍然存

在更小的例子。尽管一般来说，证明一个特定的 Lambda 确实是一个例子有多么困难并没有上限。

如果我们扫描所有可能的 lambda，不可避免地最终会发现一个能够“逃离”任何可能的公理系统的 lambda。根据计算宇宙的经验，我们可以预期，即使是相当小的 lambda，所需的公理系统的大小和力量也会迅速加速来“控制它们”。是的，原则上我们可以为任何给定的 lambda 添加一个自定义公理，但要点是我们可以预期很快会以某种意义上“任意小的杠杆”：每个添加的公理只涵盖我们达到的 lambda 中微不足起的一小部分。

数值可解释的 lambda

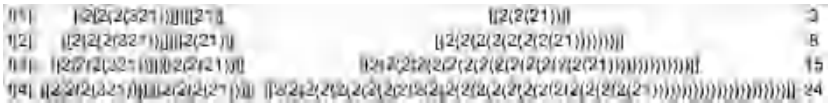
到目前为止，我们讨论了“lambda 自身的求值”。但给定一个特定的 lambda，我们总可以将其应用于“输入”，并查看“它计算了什么”。当然，输入本身也是另一个 lambda，所以“lambda 应用于输入”就是“lambda 应用于一个 lambda”，这本身也是一个 lambda。因此，我们基本上回到了“lambda 自身的求值”。

但如果输入我们给出的东西是“可解释”的？比如说，对应于我们上面使用的表示中的一个整数的 lambda。一个 lambda 应用于这样的东西将——假设它终止——不可避免地只是给出另一个 lambda。而且大多数情况下，那个结果的 lambda 不会是“可解释”的。但有时它可以是。

例如考虑：

$$\lambda\,[\,\lambda\,[\,2\,[\,\lambda\,[\,2\,[\,2\,[\,3\,[\,2\,]\,[\,1\,]\,]\,]\,]\,]\,]\,]$$

如果我们把这个 lambda 应用于表示为 lambda 的整数序列，会发生什么：



换句话说，这个 lambda 可以被解释为从一个整数到另一个整数的数值函数——在这种情况下是函数：

$$n \mapsto 2\,n + n^2$$

那么λ能表示哪些种类的数值函数呢？由于λ能够进行通用计算，它们最终必须能够表示任何整数函数。但所需的λ可能会非常庞大。

早些时候，我们看到阶乘函数的构造表示作为一个 lambda：

```
[[2(11)][2(11)]][[(1[[1]]][(2]]1([[[2(321)]]][[1]])
    ([[[3(21)]]1(2([[[3[1(24)]]][2][1]]]1)}]]]
```

我们还知道，如果我们取代表整数 m 的（"丘奇数"）λ，并将其应用于代表 *n* 的λ，我们会得到代表 n^m 的λ。然后可以很直接地构造一个λ来表示任何可以通过幂运算组合得到的函数：

[1]	n	[1(11)]	n^{n^n}	[1(11)1]	$(n^{n^n})^n$
[11]	n^n	[1111]	$((n^n)^n)^n$	[1(1(11))]	$n^{n^{n^n}}$
[111]	$(n^n)^n$	[1(111)]	$n^{(n^n)^n}$	[11(11)]	$(n^n)^{n^n}$

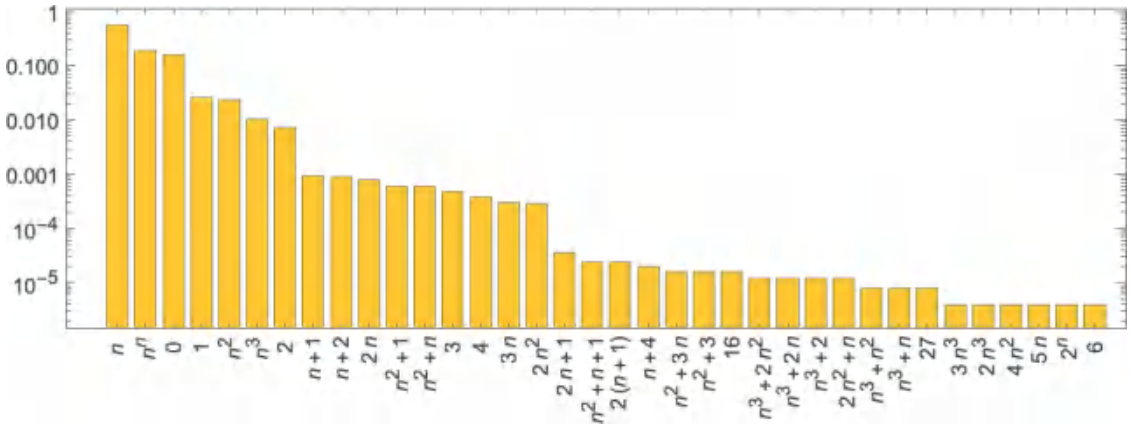
并且可以很容易地继续得到基于超乘方运算和更高阶超运算的函数。例如

```
[[1[[12(2[[21]])]][[[2(321)]]]]11]
```

给出了阿克曼函数的一个对角线，其连续项为：

$$\left\{1, 3, 7, 61, 2^{2^{2^{16}}} - 3, \ldots\right\}$$

但“野生 lambda”又如何呢？它们“数值可解释”的频率有多高？它们会代表哪些函数？例如在大小为 5 的 lambda 中，82 个中有 10 个——大约 12%——是“可解释”的，并代表函数 0、1、*n* 和 n^2。在大小为 10 的 lambda 中，“可解释”的比例下降到大约 4%，不同函数出现的频率如下：



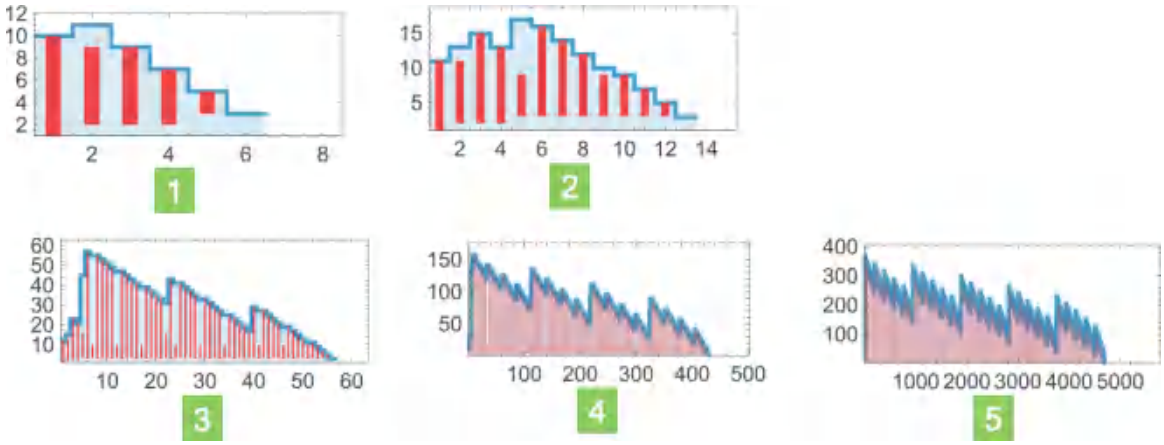
观察逐渐增大的 lambda，这里是一些函数出现的显著“首次”情况：

2	[1]	n
3	[11]	n^n
4	[[[1]]]	0
5	[[[21]]]	1
5	[[2(21)]]	n^2
6	[[[2(21)]]]	2
7	[[[2(321)]]]	$n+1$
7	[[2[2(21)]]]	$2n$
8	[[2[2(321)]]]	n^2+n
9	[[[3(3(32))[1]]]	n^3
9	[[[3(3(3(32)))1]]]	n^4
9	[1[1[3]][1]1]	If[EvenQ[n], n^n, n]
10	[1[2[[2(21)]]]1]	2^n
10	[1[1[[[1]]]2]1]	If[EvenQ[n], n, 0]
11	[1[[[3[2](21)]]]1]	If[EvenQ[n], 1, 0]
11	[1[[[32(321)]]]1]	$2^n n$
11	[1[1[1]][[23]]1]	If[$n = 2, 4, n$]
11	[1(1[1[1]][[3]])1]	If[$n = 2, 16, n$]
11	[[2[[2(2(31))]](21)]]	$2^n n + 2^n - 1$
11	[[2(2[[2(2(31))]]1)]]	$(2^{n+1}-1)n$
11	[1[[[3[2](21)]]]1]	Mod[$1+n$, 2]
11	[1[1[[[1]]](22)]1]	If[EvenQ[n], n^n, 0]

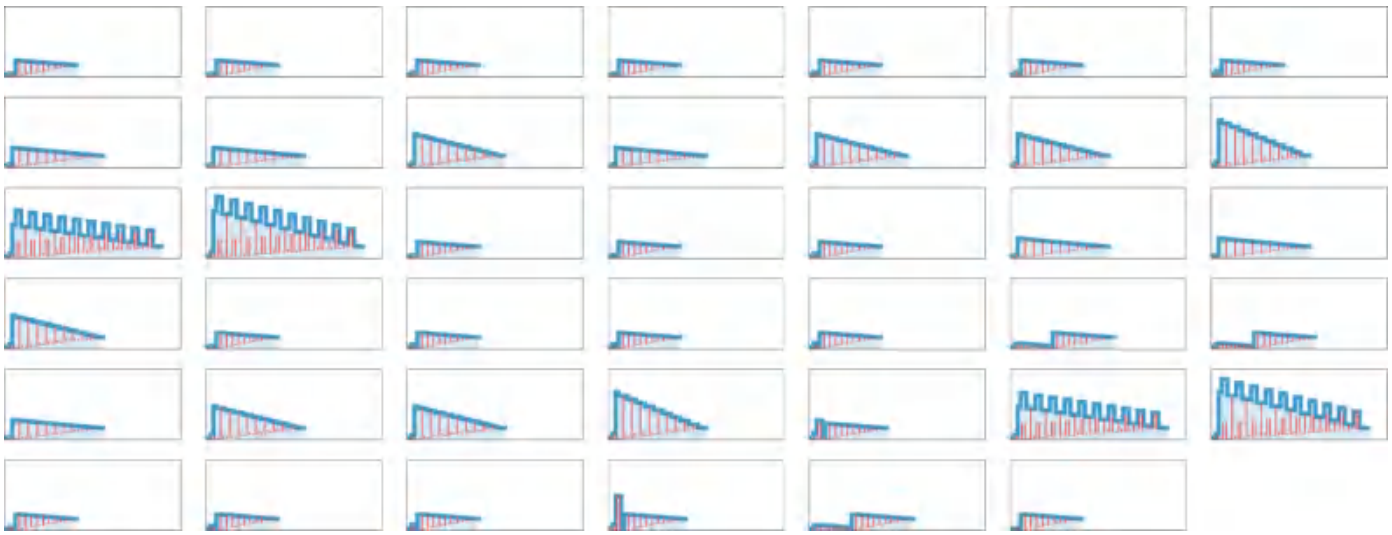
是的，我们可以将给出特定函数的最小 lambda 的大小视为该函数相对于 lambda 的 “算法信息内容”。

不用说，还有一些其他的惊喜。比如 [[22[1]]] ，它对任何整数输入都给出 0，但需要超级指数级

增加的步骤来完成：

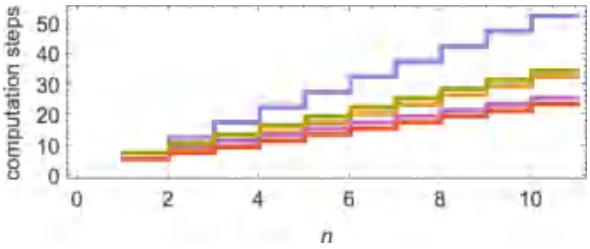


我们一直在讨论由哪些 lambda 可以计算“数值”函数。但我们可以提出另一个问题是这样的计算需要多少时间（即需要多少次 beta 归约）——或者它们需要多少“空间”来存储中间表达式。确实，即使输出相同，不同的 lambda 可能会采用不同的计算方法并消耗不同的计算资源来得到这些输出。所以，例如，在 size-8 的 lambda 中，有 41 个可以计算函数 n^2。这里展示了当 n=10 时，运行这些 lambda 生成的中间表达式的尺寸（所有图像都使用相同的比例绘制）：



这些图片表明，这些不同的 lambda 在计算 n^2时有各种不同的方法。其中一些方法需要更多的“内存”（即中间 lambda 的大小），而另一些则不需要。此外，有些方法需要更多的时间。

对于这个特定的 lambda 集合，所需的时间总是基本上随着 n 线性增长，尽管可能有几种不同的速率：



但在其他情况下，观察到不同的行为，包括一些非常长的“运行时间”。确实，在一般情况下，人们可以想象构建一个关于λ的完整计算复杂性理论，从这类经验性的、规则学探究开始。

我们在这里讨论的所有内容都是针对单参数函数的。但我们也可以查看多参数函数，这些参数可以以“柯里化”形式传递给λ：$f\,[x][y]$，等等。并且在这种设置下，$[[[2(31)]]]$，例如，给出 $x\,y$，而 $[[[2(3(3(31)))]]]$ 给出 $x\,y^3$。

Lambda 求值的多元图

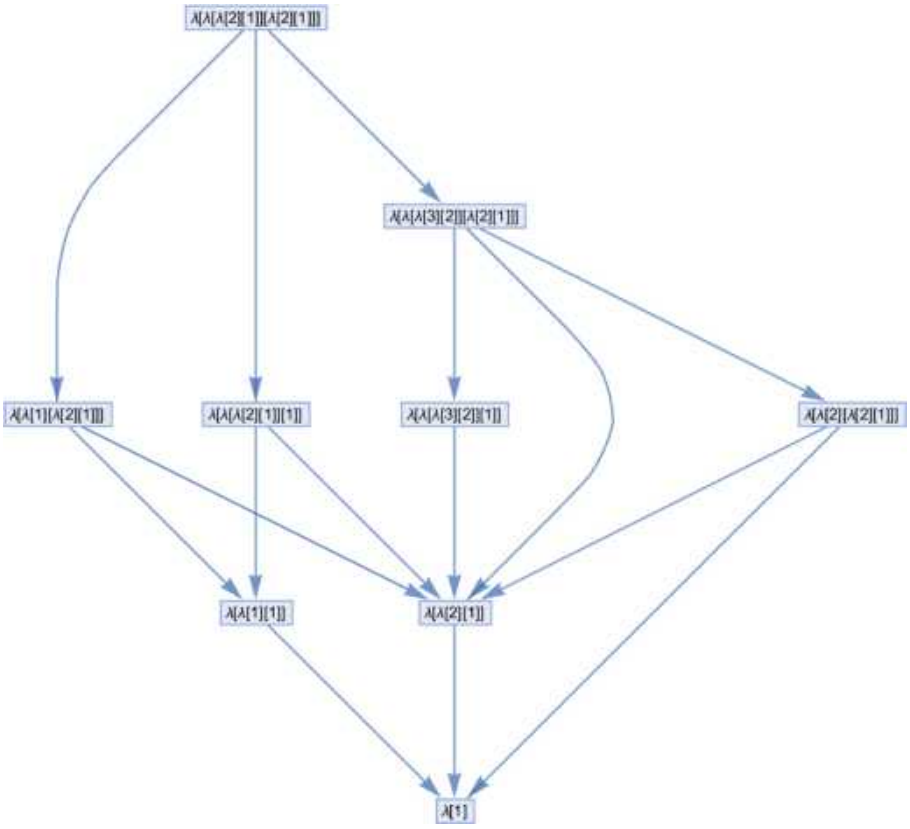
假设你要计算一个 lambda 表达式，例如：

$$\lambda\,[\,\lambda\,[\,\lambda\,[\,2\,]\,[\,1\,]\,]\,[\,\lambda\,[\,2\,]\,[\,1\,]\,]\,]$$

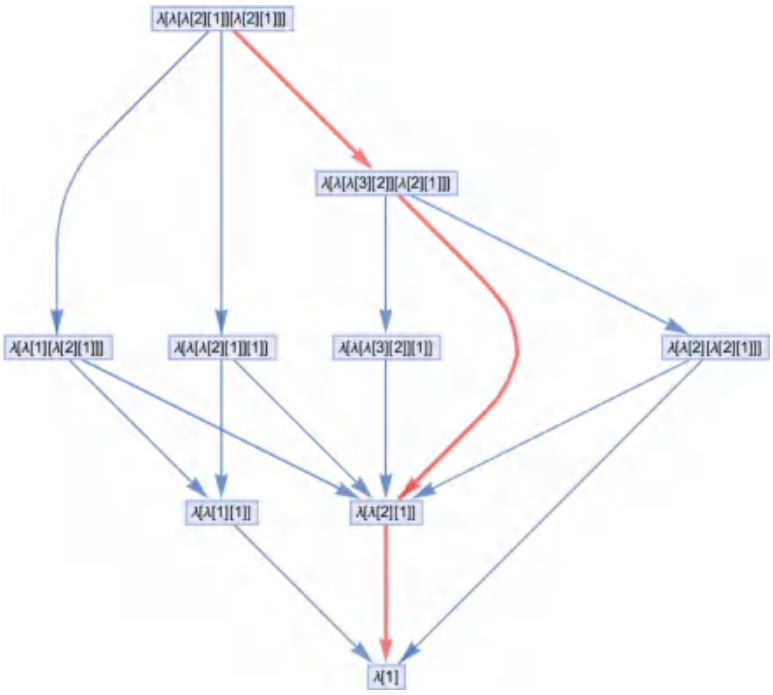
你想进行一个β归约。但哪一个？对于这个表达式，有3个可能的进行β归约的地方，它们都给出不同的结果：

$$\lambda[\,\lambda[\lambda[2][1]][\lambda[2][1]]\,] \longrightarrow \lambda[\lambda[\lambda[3][2]][\lambda[2][1]]]$$
$$\lambda[\lambda[\,\lambda[2][1]\,][\lambda[2][1]]] \longrightarrow \lambda[\lambda[1][\lambda[2][1]]]$$
$$\lambda[\lambda[\lambda[2][1]][\,\lambda[2][1]\,]] \longrightarrow \lambda[\lambda[\lambda[2][1]][1]]$$

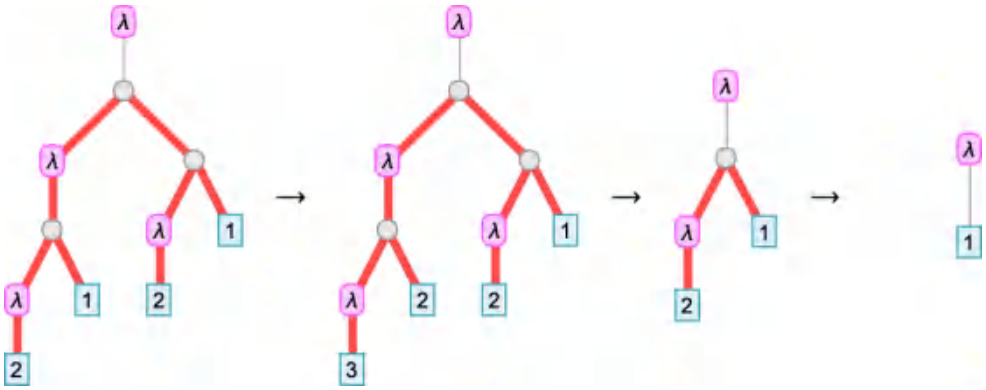
到目前为止我们所做的一切，我们都假设我们在每一步都使用“第一个”β归约（我们很快会讨论“第一个”是什么意思）。但在一般情况下，我们可以查看所有可能的“求值路径”——并从它们形成多路图（双重边反映了两个不同的β归约碰巧得到相同结果的事实）：



那么在这种情况下，我们的标准求值路径是：

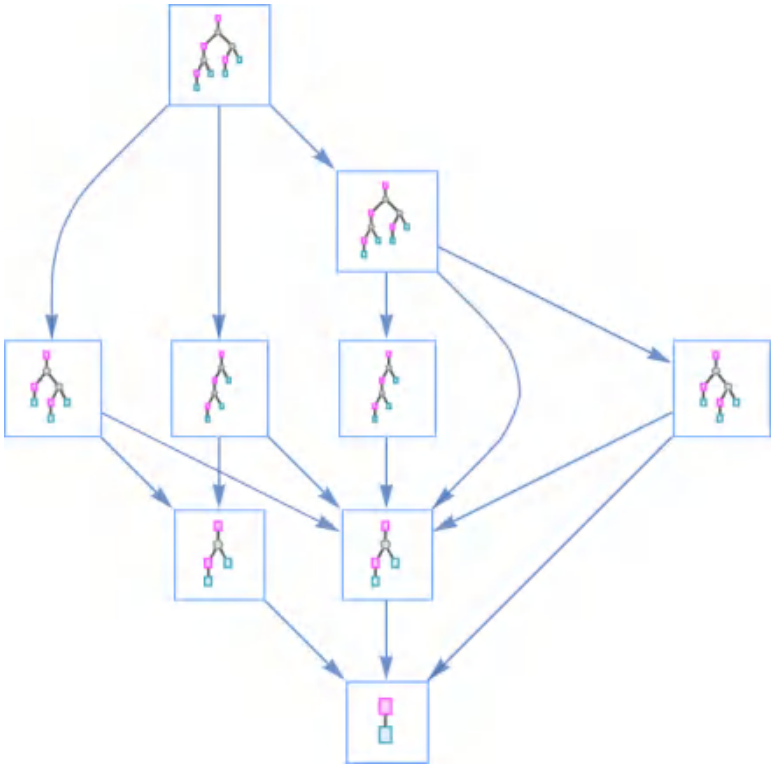


我们通过在每个步骤中选择“ 最左外层 ”可能的 beta 归约来得到这条路径，即涉及表达树中“最高”相关λ的那个（如果有多个选择，则选择最左边的那个）：

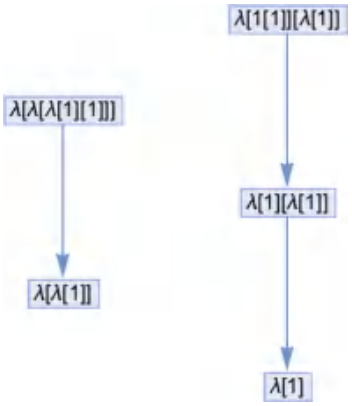


但在上面的多路图中很明显，我们可以选择任何路径，最终都会得到相同的结果。实际上，这是 lambda 的一个普遍性质（所谓的汇聚性或 Church-Rosser 性质），即所有终止的给定 lambda 求值都会以相同的结果终止。

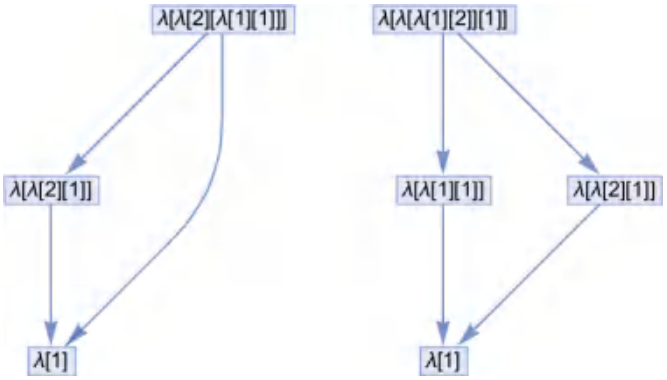
顺便说一下，这里是用树形形式渲染 lambda 的多路图看起来像这样：



在规模为 5 时，所有 lambda 都会产生平凡的多路径图，其中不涉及分支，例如：

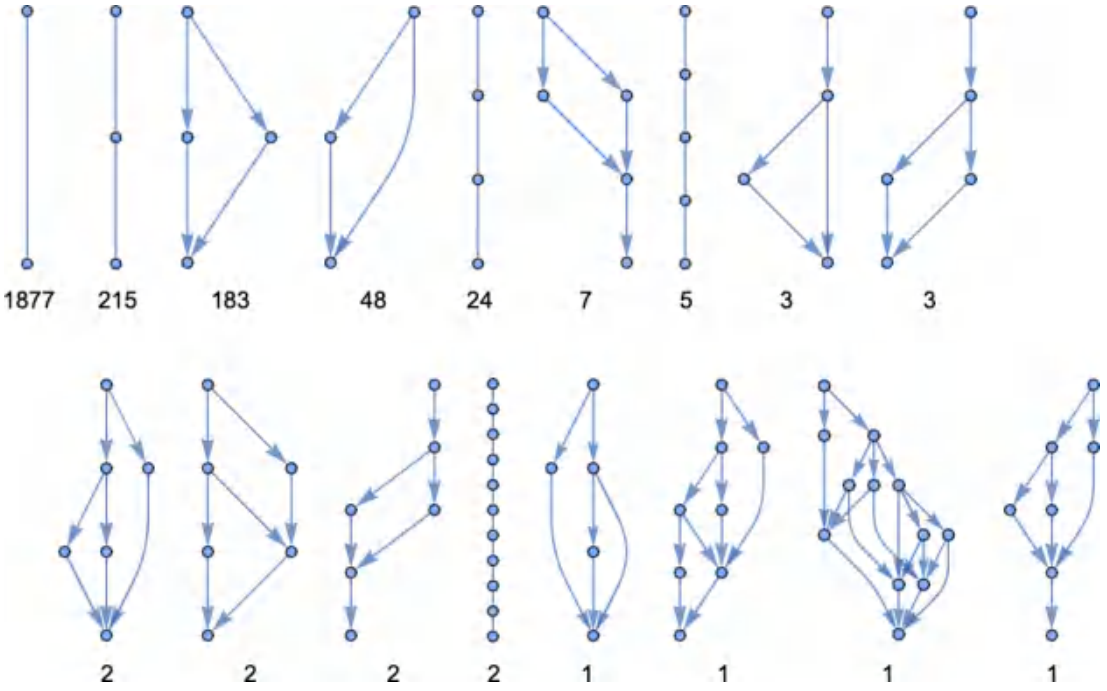


在规模为6时，开始出现分支——以及合并：



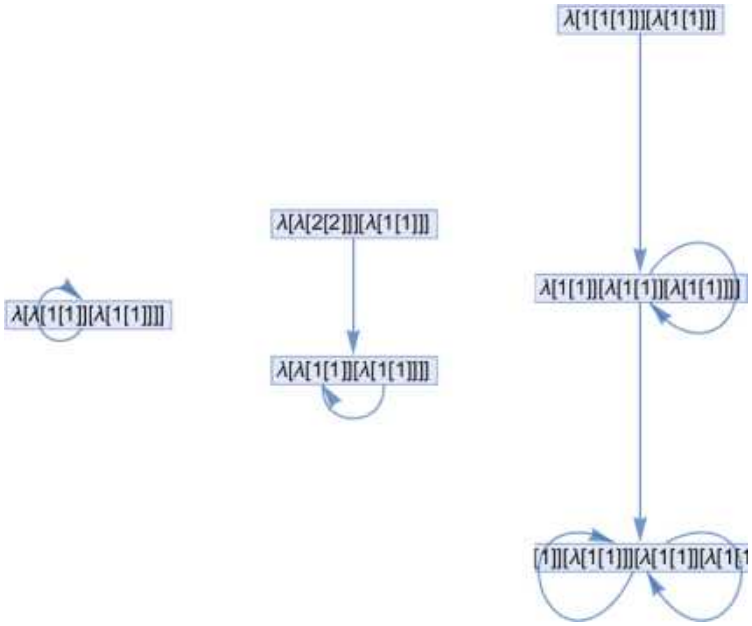
此外，还有另一件事发生：名为“循环 lambda”的 λ[1[1]][λ[1[1]]] 会产生一个多路径图，该图形成了一个环：

在规模为7时，许多不同的多路径图拓扑开始出现。大多数都会导致终止：



（这些多路图中最大的与 λ[1[1]][λ[λ[2][1]]] 相关联，有12个节点。）

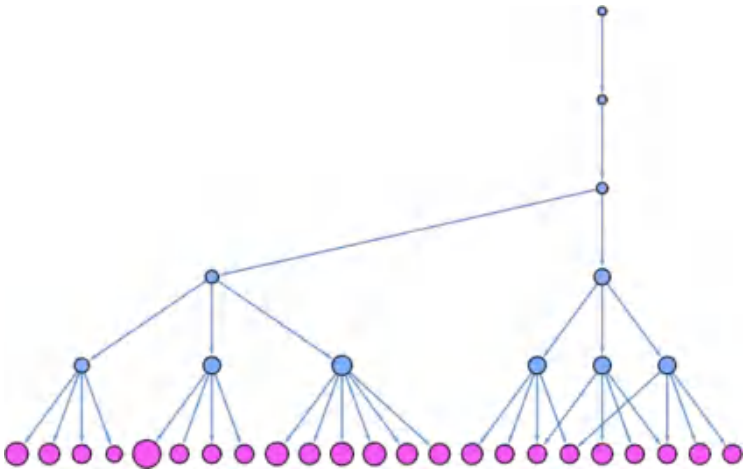
然后还有一些循环的情况：



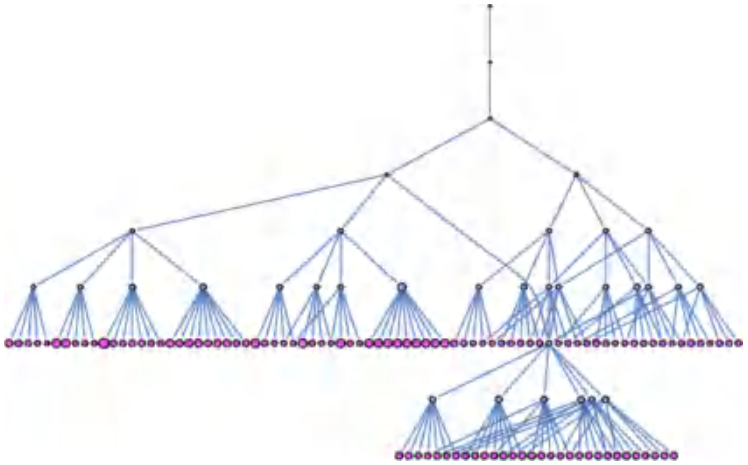
最后是

λ [1 [1]] [λ [1 [1 [1]]]]

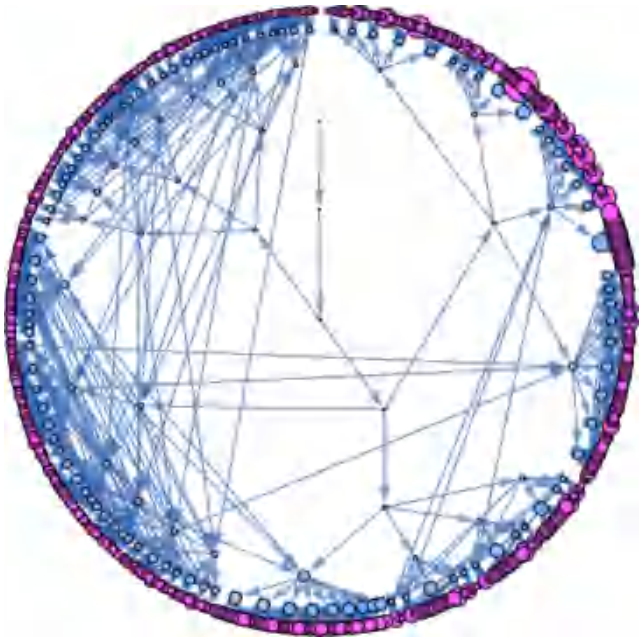
在这里发生了新情况：多路图有许多不同的分支。经过5步之后变成



其中每个节点的尺寸表示它所代表的 lambda 的大小，粉红色节点表示可以进一步进行 beta 约简的 lambda。再进行一步之后，多路图变成



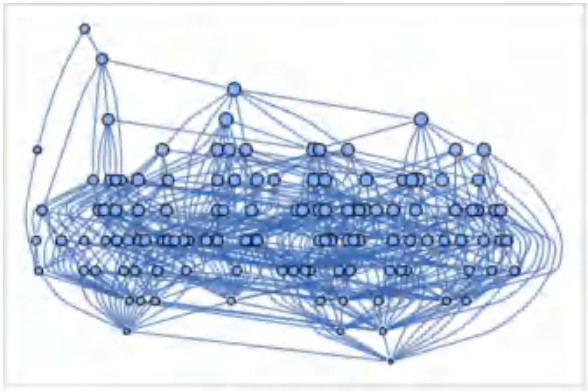
或者，经过另一步之后，在双曲嵌入中：



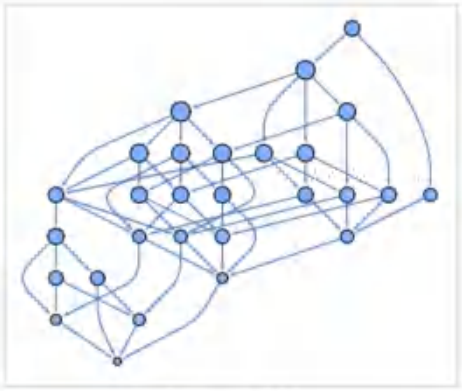
在这个特定情况下，没有任何分支会终结，连续步骤中多路系统的总大小是：

$$\{2, 3, 5, 11, 34, 141, 736, 4636, 34\,568\}$$

使用大小为 8 的 lambda 时，终结的多路图可以更大（第一个情况有 124 个节点）：

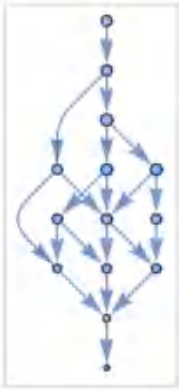


[1(11)][[2]1]



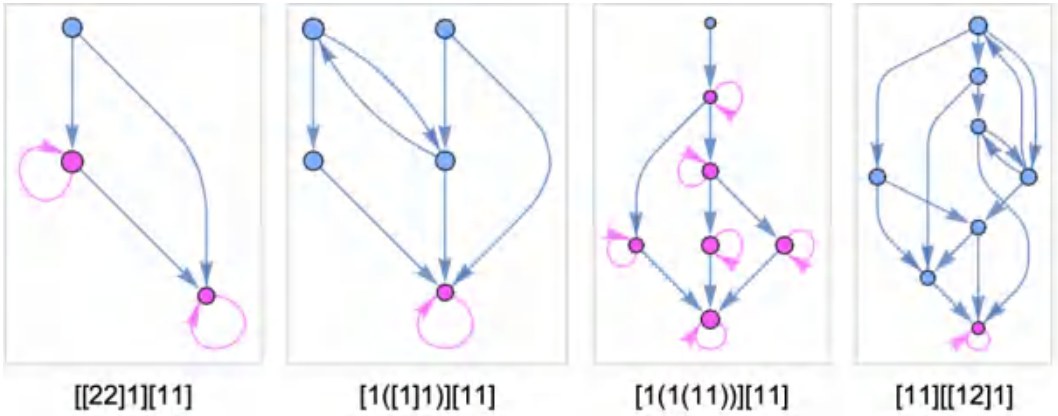
[111][[2]1]

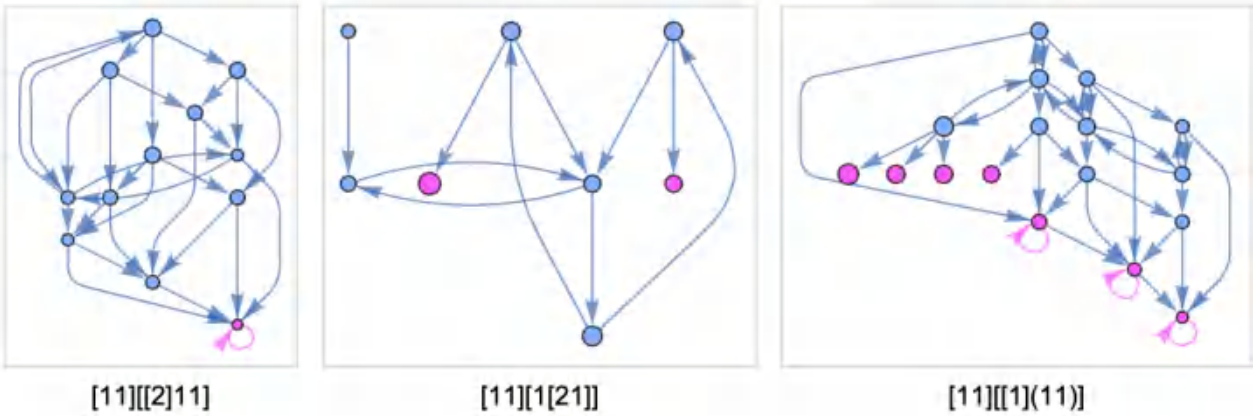
[1(11)][[1]1]

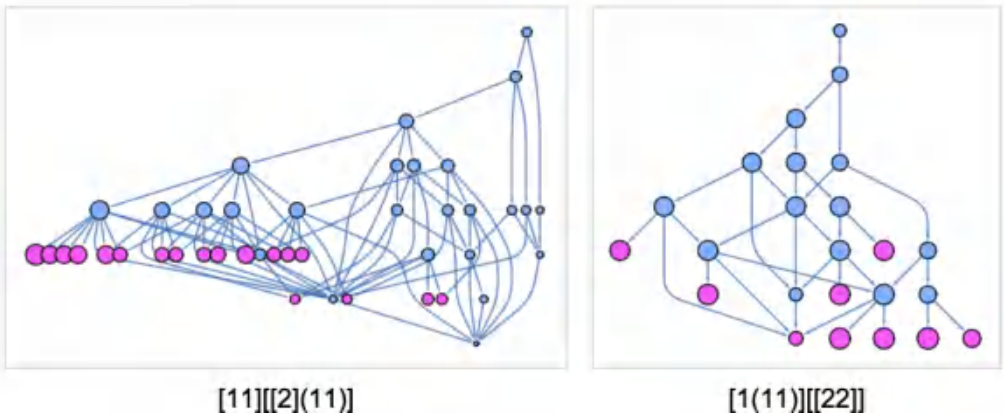


[1(1[1])][11]

并且对于非终结情况，存在各种新的拓扑结构，例如：

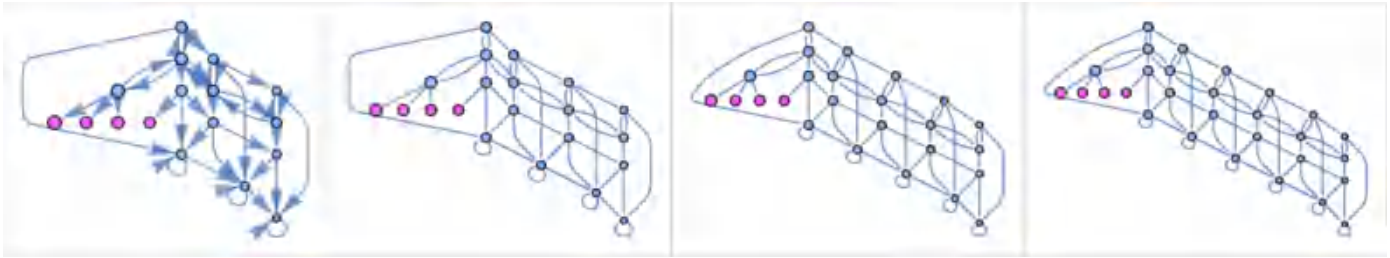






这些图是通过进行 5 次连续的 beta 归约得到的。那些以循环结束的不会因进一步归约而改变。但那些标有粉色节点的会变化。在大多数情况下，最终会得到一个逐渐增加的可以进行的归约数量（通常呈指数增长，但在此处仅为步骤 t 时的 $2t - 1$）：

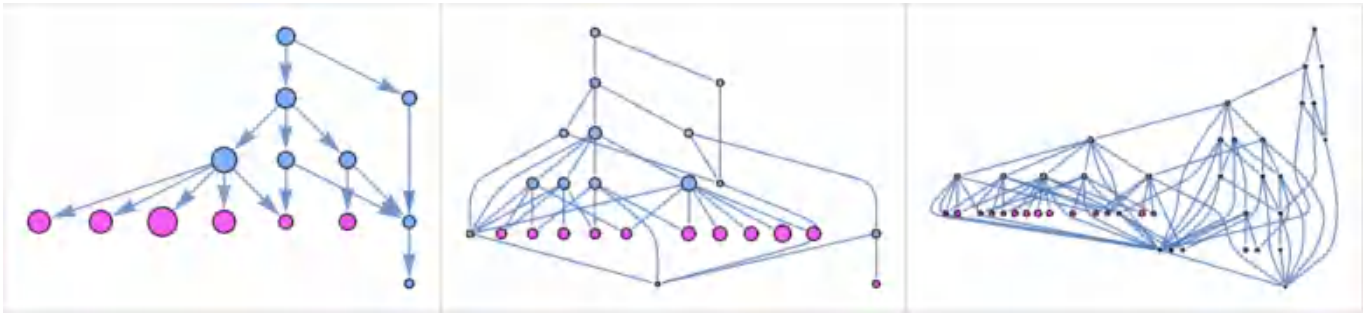
与此同时，有时“未解决的归约”数量会保持不变：



我们知道，如果存在 beta 约化的不动点，它总是唯一的。但能否同时存在一个唯一的不动点，以及永远延伸的分支？结果是可以的。而第一个发生这种情况的 lambda 是：

```
λ[1[1]][λ[λ[2][1[1]]]]
```

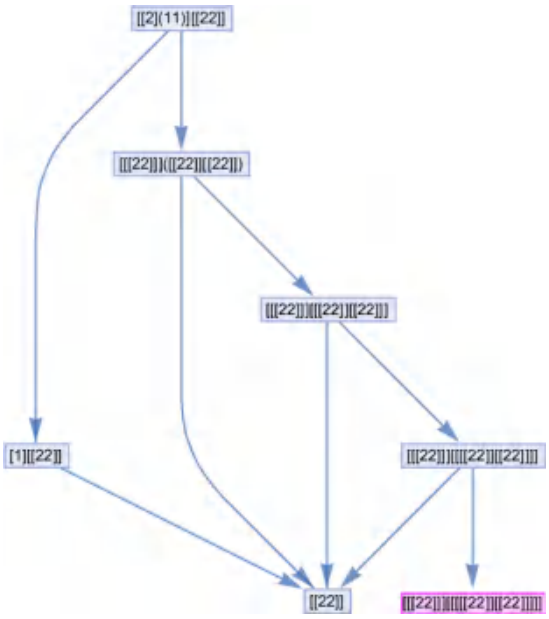
使用我们的标准求值方法，这个 lambda 在 λ[1] 处经过 5 步终止。但还有其他可以遵循的路径，这些路径不会终止（如末尾粉色节点所示）：



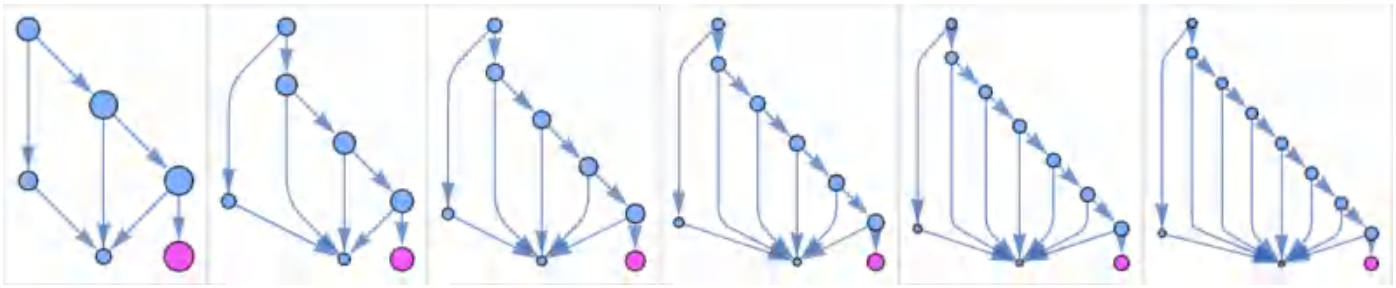
确实，在步骤 t 时，此类路径的数量遵循一个类似斐波那契的递推关系，渐近增长 $\sim\phi^t$ 。

对于大小为 9 的 lambda，情况要简单得多，终止/非终止现象会发生

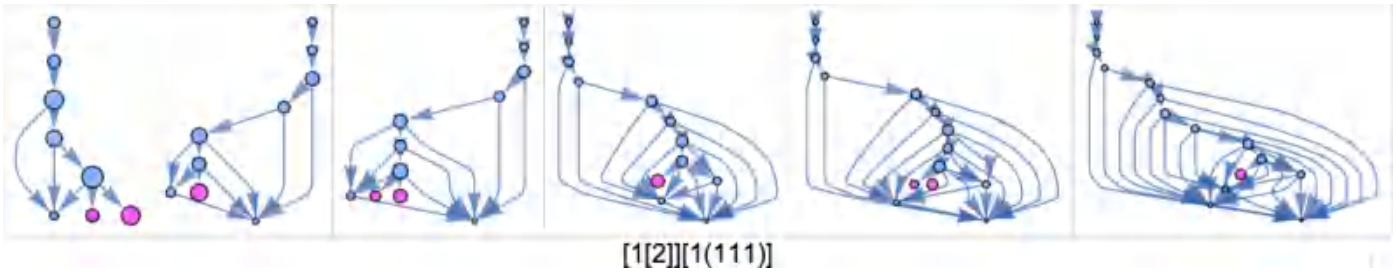
```
λ[λ[2][1[1]]][λ[λ[2[2]]]]
```



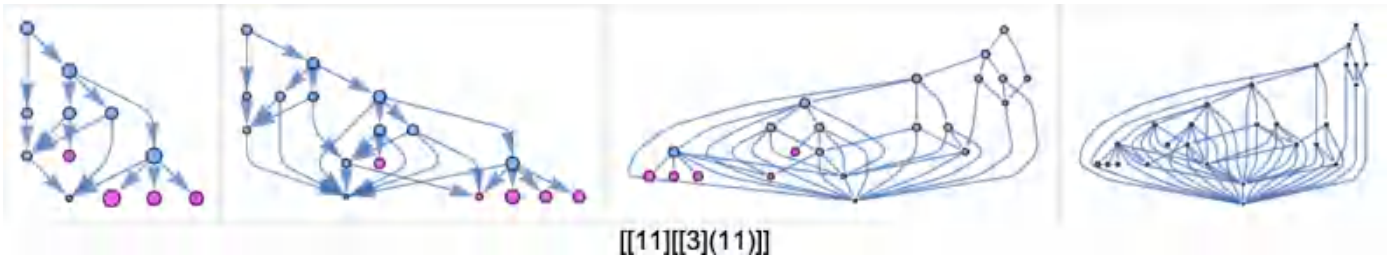
现在每一步都只有一个非终止路径：



还有一个 lambda 在一条和两条非终止路径之间交替：

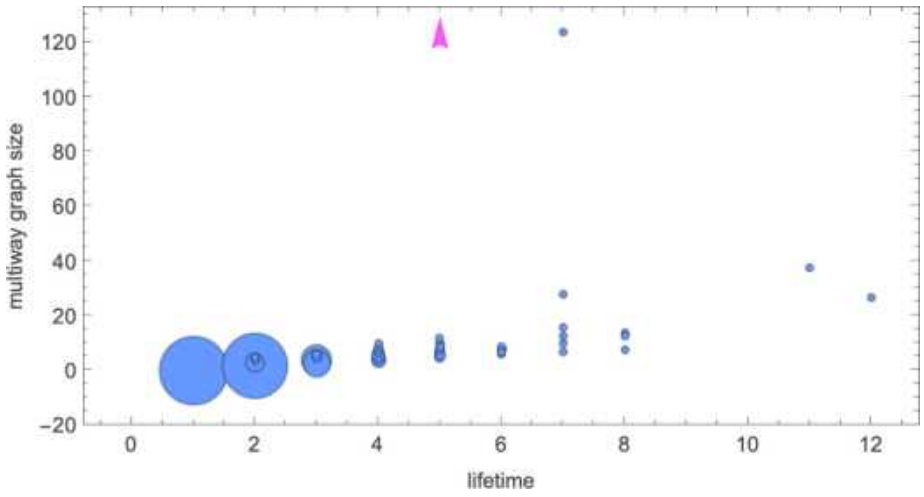


还有一个 lambda 在渐近情况下总是恰好有 5 条非终止路径：

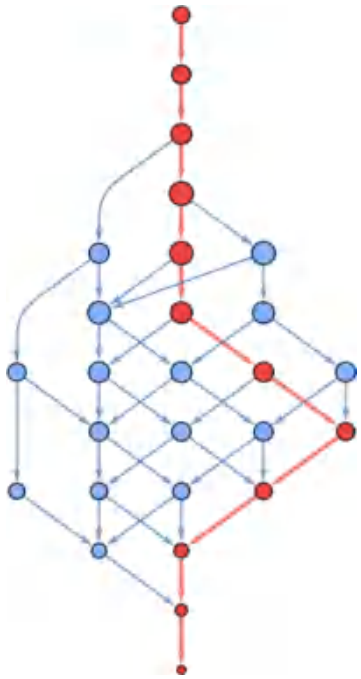


好的，有些 lambda 具有非常“茂密”（或无限）的多路图，而有些则没有。这与我们通过标准求值过程获得的这些 lambda 的生命周期如何相关联?

这是所有有限寿命的 size-8 lambda 的大小与生命周期的关系图：

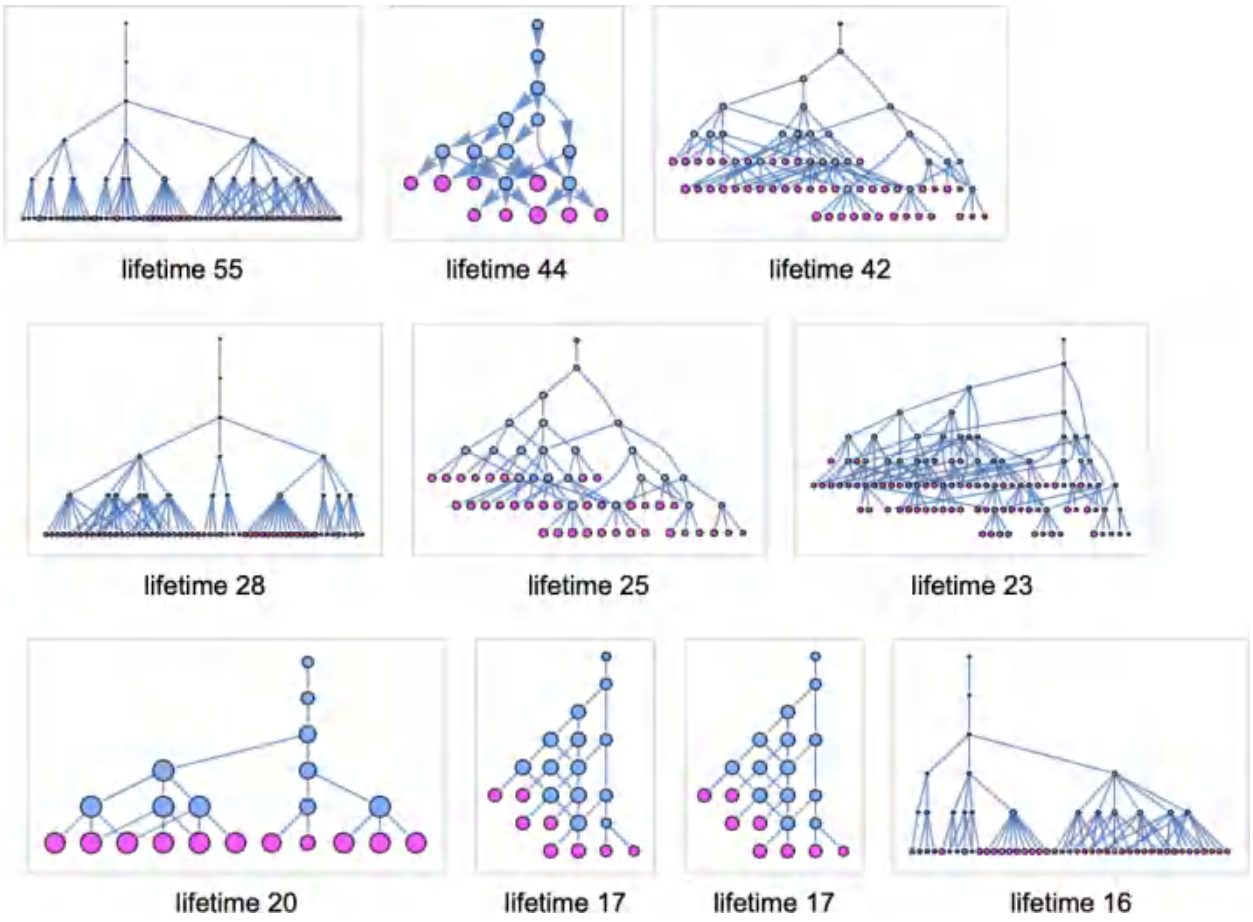


除了上面讨论的一个案例（这里用红色箭头标出）之外，其他所有多路径图都是有限大小的——我们观察到它们的大小与标准评估过程得到的生命周期之间存在一定的相关性。值得注意的是，即使标准评估生命周期最长的 lambda（12）仍然有一个相当小的（27 节点）多路径图。

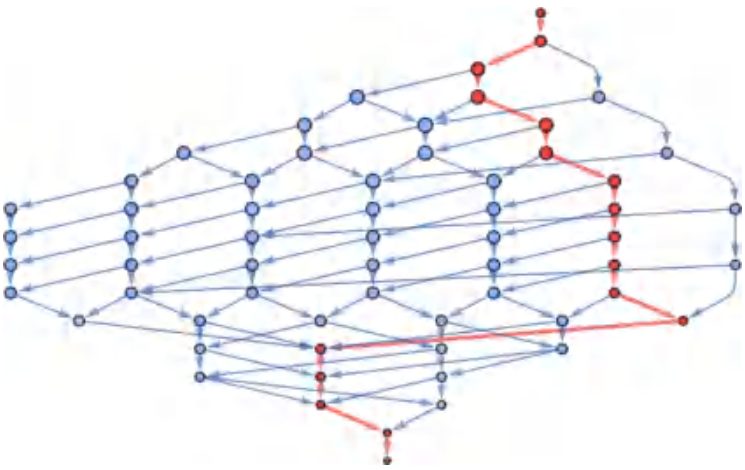


而最大的（有限的）多重路径图（有 124 个节点）出现在一个标准评估生命周期为 7 的 lambda 中。

在大小为 9 时，这里列出了 10 个具有最长标准评估生命周期的 lambda 的多重路径图，计算到 5 步：



在标准求值寿命为17的情况下，多路径图是有限的：



但在所有其他展示的情况中，图可能是无限的。例如，在寿命为 55 的情况下，多路径图中在步骤 *t* 的节点数量增长如下：

{2, 3, 6, 19, 77, 321, 1245, 4323, 13346, 36670, …}

但不知何故，在所有这些不同路径中，我们知道至少有一条是会终止的——至少在55步之后，以给出 λ[1] 。

不同的求值策略

多路径图展示了 lambda 的所有可能的 beta 归约序列。我们的“标准求值过程”选择其中一个特定的序列——通过在每个步骤中采用某种策略来选择要执行的 beta 归约。但如果我们使用不同的策略会怎样呢?

作为一个例子，考虑:

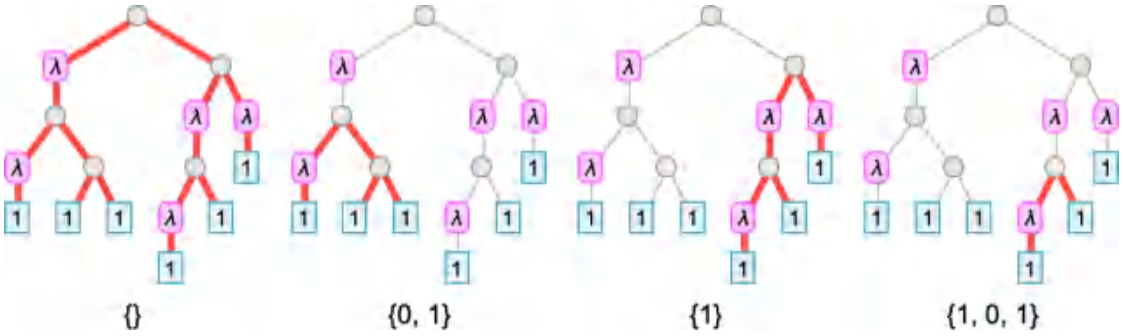
λ[λ[1][1[1]]][λ[λ[1][1]][λ[1]]]

对于这个 lambda，结果表明有 4 个可能的 beta 归约位置（位置使用 Wolfram 语言规范指定）：

tree position	*selection*	*outcome*
{}	λ[λ[1][1[1]]][λ[λ[1][1]][λ[1]]]	λ[1][λ[λ[1][1]][λ[1]][λ[λ[1][1]][λ[1]]]]
{0, 1}	λ[λ[1][1[1]]][λ[λ[1][1]][λ[1]]]	λ[1[1]][λ[λ[1][1]][λ[1]]]
{1}	λ[λ[1][1[1]]][λ[λ[1][1]][λ[1]]]	λ[λ[1][1[1]]][λ[1][λ[1]]]
{1, 0, 1}	λ[λ[1][1[1]]][λ[λ[1][1]][λ[1]]]	λ[λ[1][1[1]]][λ[1][λ[1]]]

我们的标准求值策略执行这里列出的第一个 beta 归约。但它具体是如何确定应该执行这个归约的呢? 嗯，它查看指定 lambda 表达式树中每个可能归约位置的 0 和 1 的列表。然后它选择具有最短位置列表的那个，如果位置列表有相同长度，则按字典序排序。

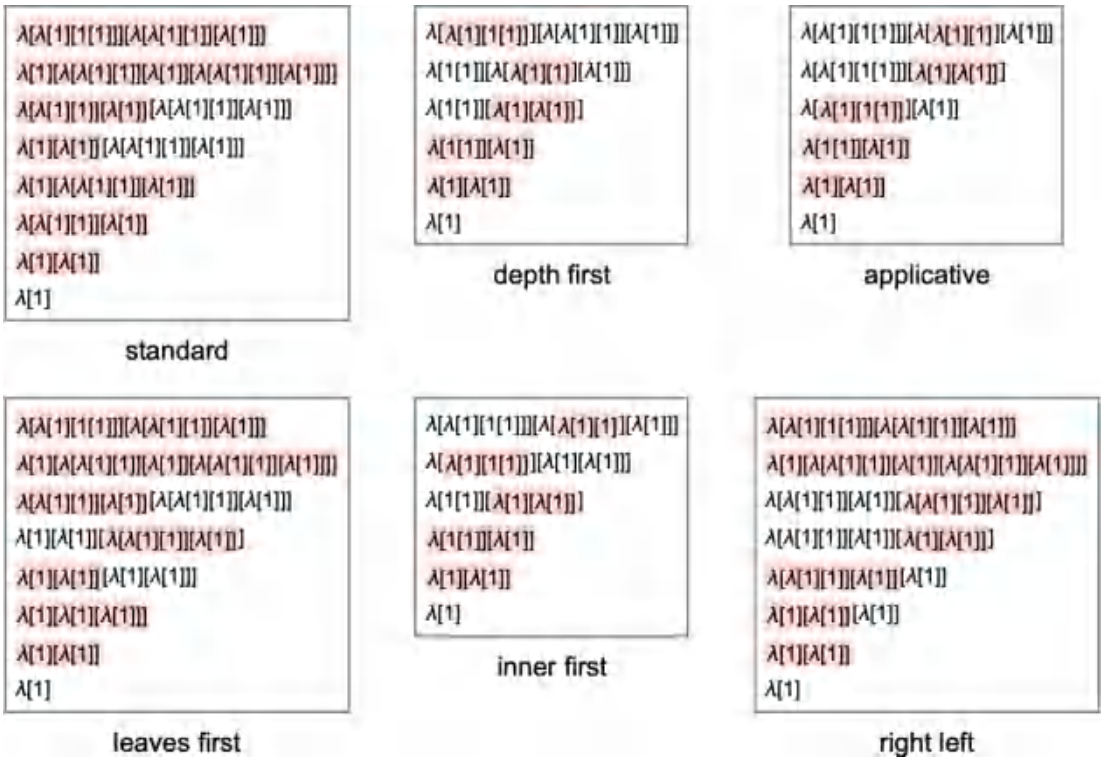
从树的结构来看，这对应于挑选最左外层的 beta 归约，即在我们对树的渲染中是“最高且最靠左”的那个:



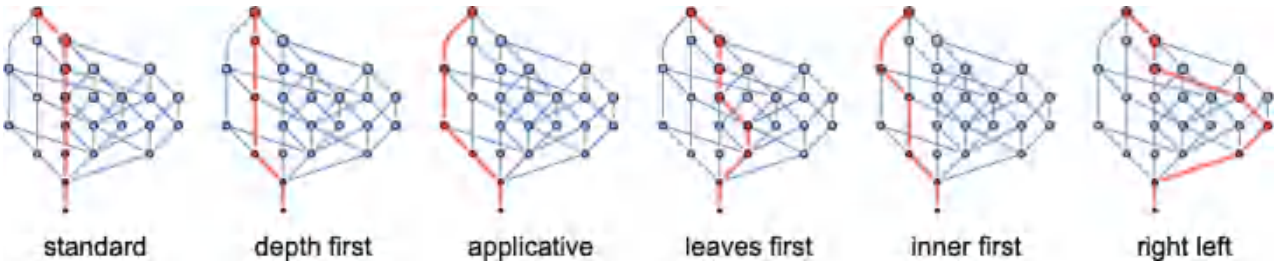
那么我们还能使用哪些其他策略呢？我们可以通过给出一个算法来指定一个策略，该算法能从任何 0 和 1 的列表集合中挑选出一个列表。（目前我们只考虑“顺序策略”，即一次挑选出一个位置列表——即一个单一归约。）但指定策略的另一种方式是定义表达式树上的顺序，然后说明我们进行的 beta 归约是如果我们按该顺序遍历树时最先到达的那个。因此，例如，我们的最左外层策略与树的广度优先遍历相关联。另一种求值策略是最左内层，它与树的深度优先遍历相关联，以及所有 beta 归约树位置的一个纯粹字母顺序，与长度无关。

在 Wolfram 语言中，选项 TreeTraversalOrder 提供了用于函数 TreeMap 等可能遍历顺序的详细参数化，而每个这样的顺序都可以用来定义 lambda 的求值策略。Wolfram 语言中应用于表达式树的标淨 ReplaceAll (/.)操作也定义了一种求值策略——这恰好是我们标准的最外层最左操作。与此同时，在给出赋值 λ[x_][y_]:= …之后使用标准的 Wolfram 语言求值会产生最左内层求值策略（因为在每一层中，头部 λ[…] 都先于参数被求值）。然而，如果λ具有 Hold 属性，那么我们得到的将是所谓的“应用”顺序，其中参数在主体之前被求值。是的，这个可能求值策略的整个设置实际上与我几年前详细讨论过的组合子是相同的 。

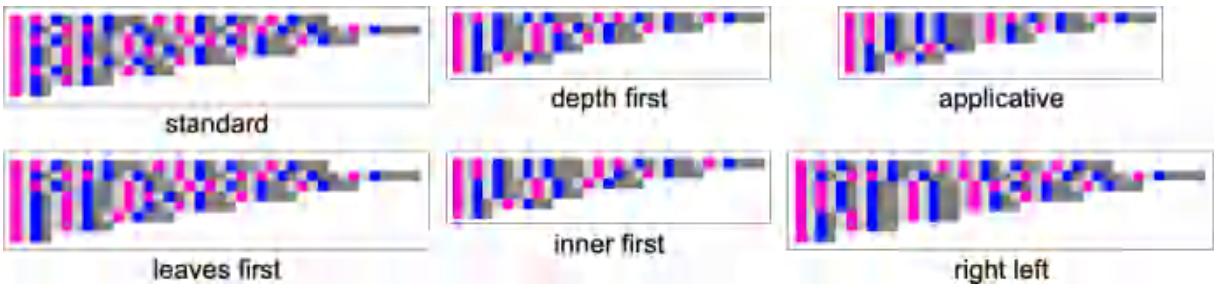
那么，不同策略会发生什么？这里是在不同情况下生成的求值链的例子：



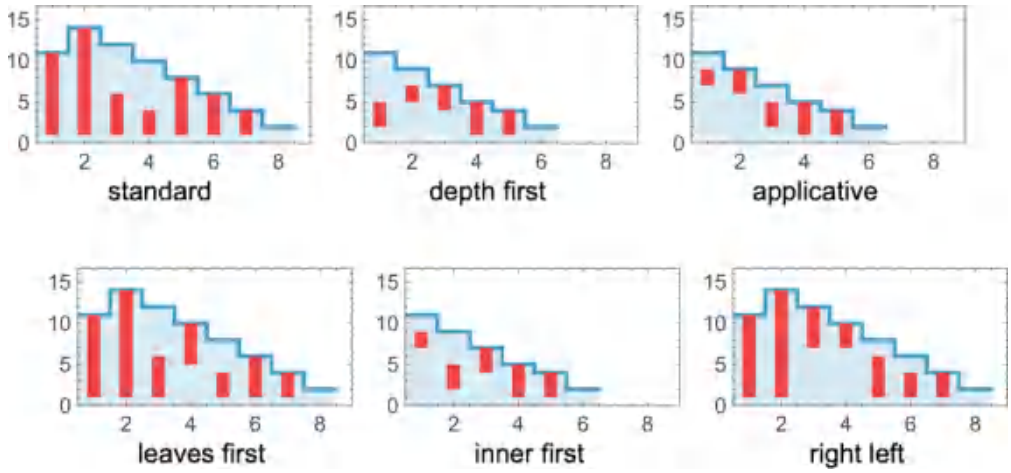
每个策略都对应着在多路径图中遵循不同的路径：



实际上，每个策略都对应着达到最终结果的不同方式：



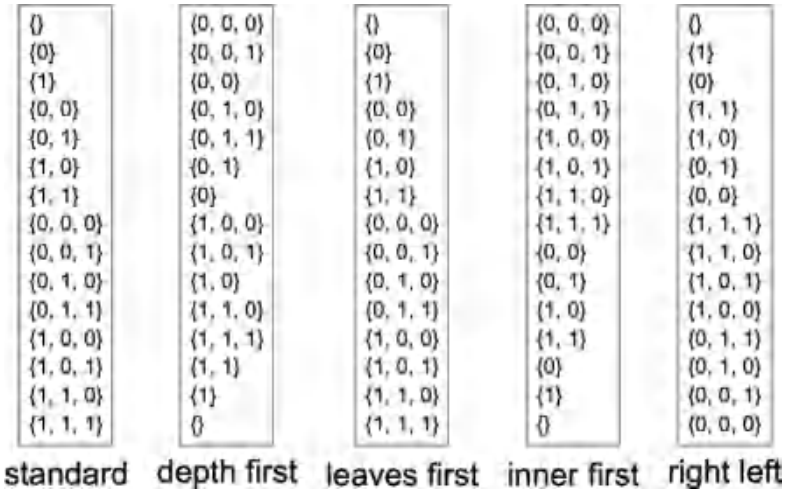
绘制每一步获得的λ的大小，我们看到不同的策略有不同的特征——特别是在这个案例中，标准策略比深度优先等需要更多步骤来达到最终结果：



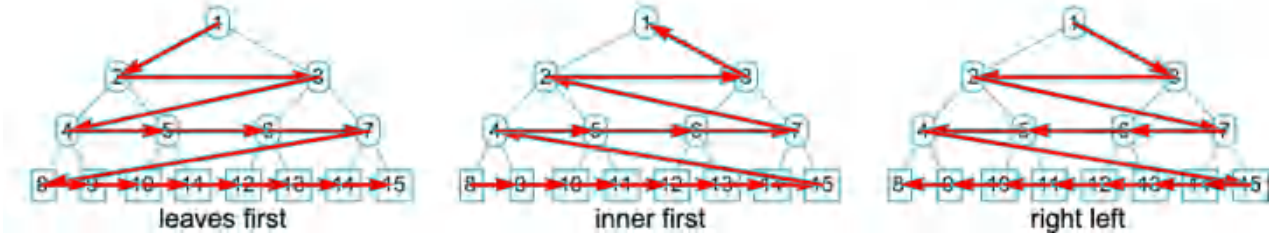
好的，但这些策略具体意味着什么？在 Wolfram 语言函数 TreeTraversalOrder 的术语中，它们是（"applicative"涉及是否为λ或•的依赖，并且工作方式略有不同）：

standard	{BreadthFirst, OuterInner, LeftRight}
depth first	{DepthFirst, InnerOuter, LeftRight}
leaves first	{LeavesFirst, OuterInner, LeftRight}
inner first	{BreadthFirst, InnerOuter, LeftRight}
right left	{BreadthFirst, OuterInner, RightLeft}

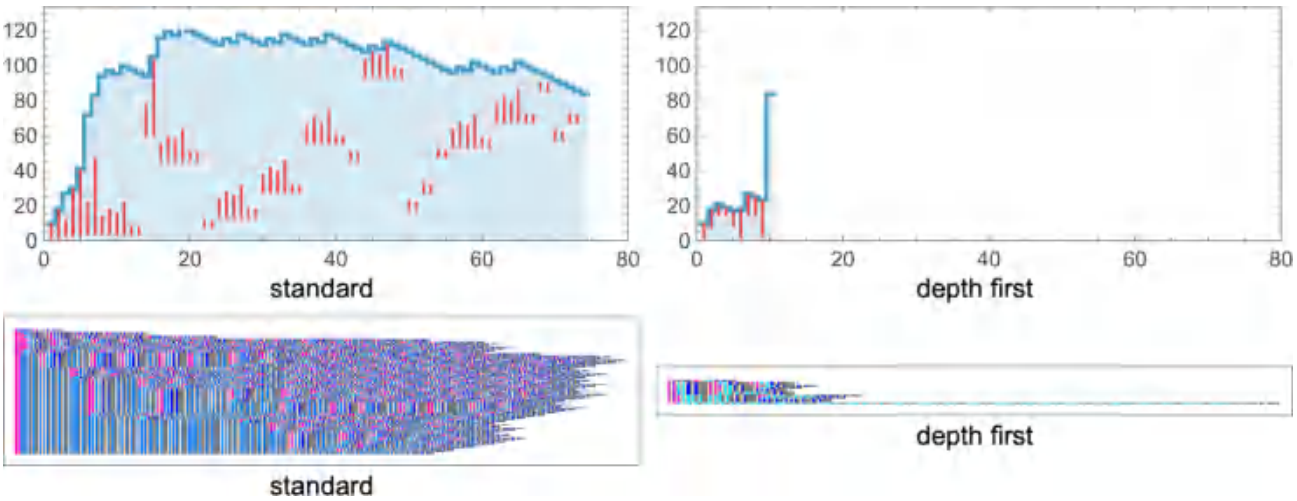
从树位置的角度来看，这些对应于以下排序



这些对应于通过树节点的这些路径：

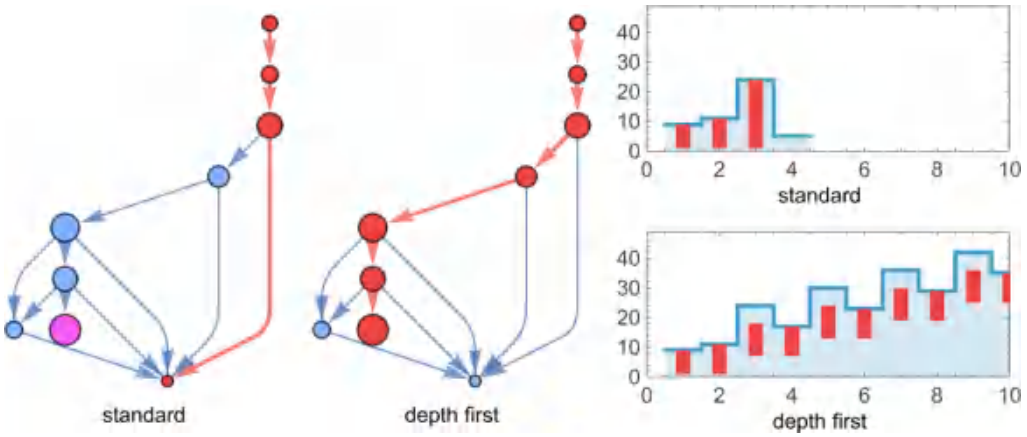


正如我们早前多次提到的，lambda 的一个普遍特征是：如果某个 lambda 的计算终止，那么得到的结果必须始终相同——与所使用的计算策略无关。换句话说，如果多向图中有终止路径，它们必须都收敛到同一个固定点。有些路径可能比其他路径短，这反映了有些计算策略可能比其他策略更高效。我们之前看到了一个这样的例子，但差异可能要大得多。例如，$[1(11)][[2(211)]]$ 使用标准计算策略需要 74 步才能终止，而使用深度优先策略只需要 10 步：

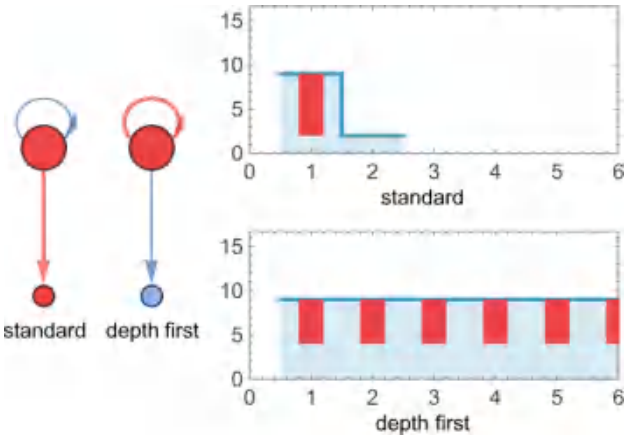


确实，我们经常看到标准求值和深度优先求值产生非常不同的行为的情况；有时前者更高效，有时后者更高效。但尽管“到达结果的方式”有所不同，最终的固定点结果不可避免地总是相同的。

但如果多路图中的只有部分路径终止会怎样呢？这种情况在大小不超过 8 的任何 lambda 中都不会发生。但在大小为 9 的情况下，例如，$[1[2]][1(111)]$ 在标准求值下终止，但在深度优先求值下不终止：

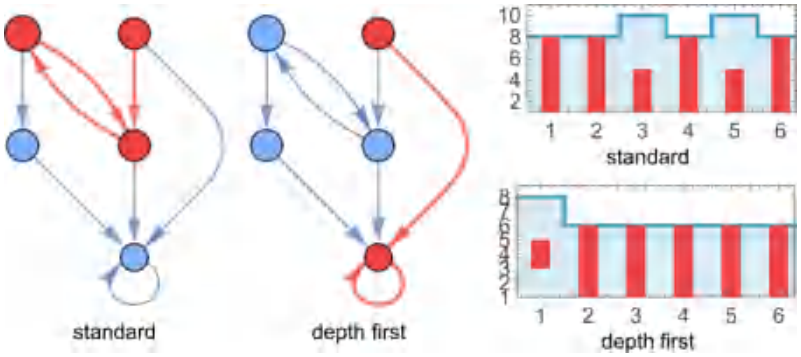


还有一个更简单的例子是 $[[2]([11][11])]$：



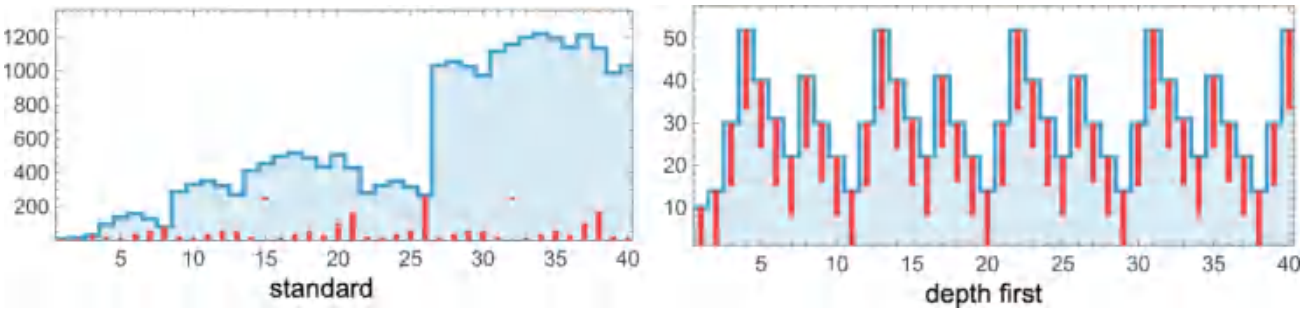
在这两种情况下，标准求值会导致终止，但深度优先求值不会。事实上，这是一个普遍的结果：如果对于某个特定的 lambda 终止是可能的，那么我们的标准（“最左外层”）求值策略将成功实现它。

但如果终止永远不可能呢？不同的求值策略会导致不同的行为。例如，$[1([1]1)][11]$ 可能会陷入周期为 2 或周期为 1 的循环：



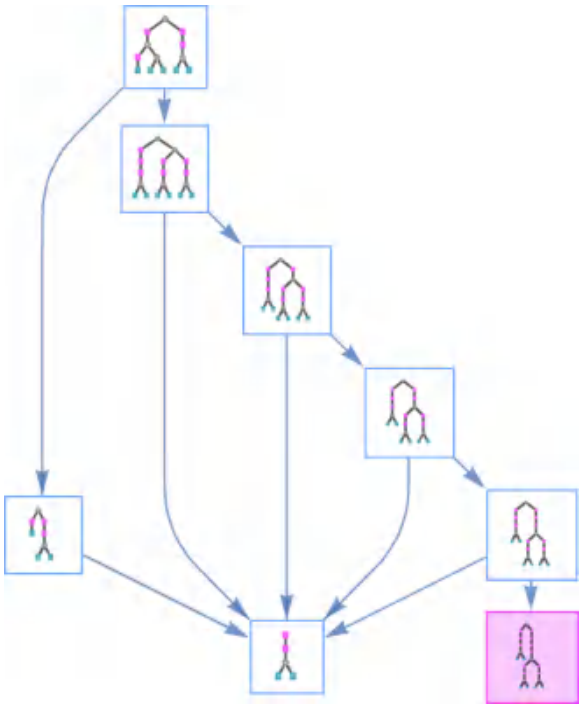
一种策略也可能给出永远增长的 lambda，而另一种策略则给出（如本 $[11][1(1(1[[3]]))]$ 例

中），比如大小周期性的 lambda——实际上展示了两种不同的不终止方式：

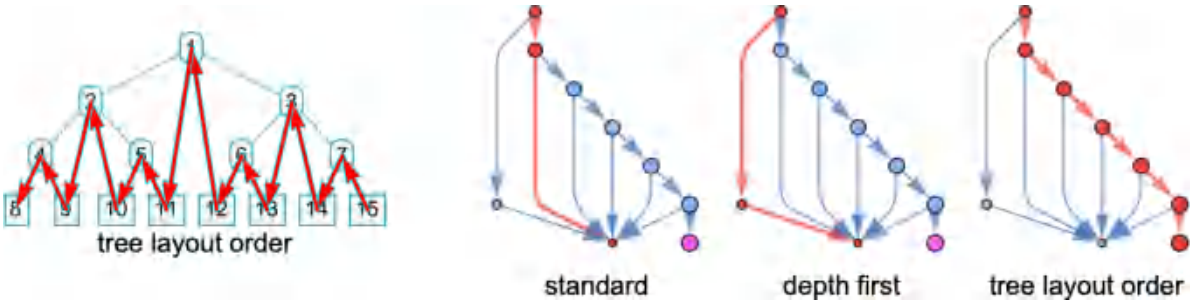


但如果一个人有一整套 lambda 呢？通常会发生什么？在每一个多路图中都可能存在一个固定点，一些循环和一些（树状的）无界增长。每一种策略都定义了多路图中的一条路径——这有可能导致所有这些可能的结果中的任何一个。通常可以看到一种类型的结果比其他类型的结果出现得更加频繁——并且需要某种“特殊调整”才能实现不同类型的结果。

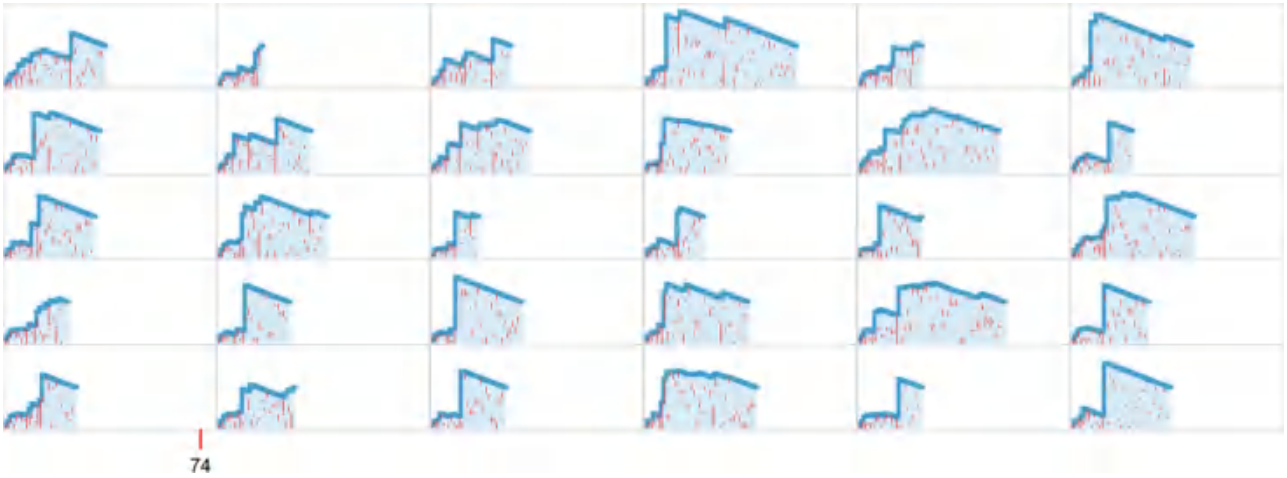
对于这种情况



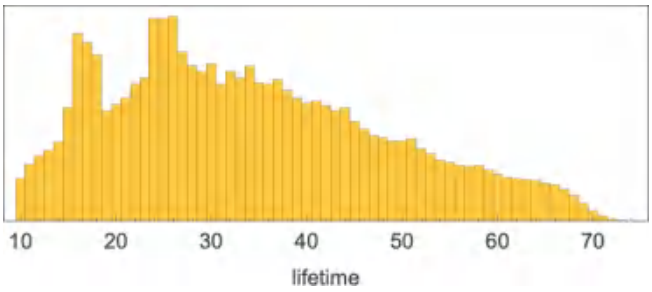
几乎所有求值策略都会导致终止——除了基于我们可能称之为“树形布局顺序”的特定策略：



那么，如果我们使用一种策略，在每一步只是随机选择不同的可能 beta 约简，且概率相等，会发生什么呢？这里有一些对于上面看到的 [1(11)][[2(211)]] lambda 的例子，它使用标准求值策略需要 74 步才能终止：



生成的曲线形状相当多样化。但值得注意的是，没有出现非终止的情况——而且终止通常比标准求值要快得多。事实上，终止时间的分布刚刚达到标准求值情况中看到的74这个值：



对于那些总是能终止求值的 lambda，我们可以认为它们求值过程中生成的表达式大小的序列是“两端固定”的。当求值不终止时，序列的大小可能会更加无序，就像这些 [11][1(1(1[[3]]))] 情况：

Lambda 的等价性

考虑这些 lambda：

```
{λ[λ[1][1]],λ[λ[2][1]],λ[1][λ[1]]}
```

它们每个都有不同的形式。但它们都评估为同一个东西： λ[1] 。特别是如果我们从数学的角度思考 lambda，这意味着我们可以认为所有这些 lambda 在某种程度上“代表同一个东西”，因此我们可以将它们视为“等效”。

在所有 18 个大小为 4 的 lambda 中，这三个实际上是唯一在意义上等效的。这意味着有 15 个“评估不等价”的 lambda 直到大小为 4。

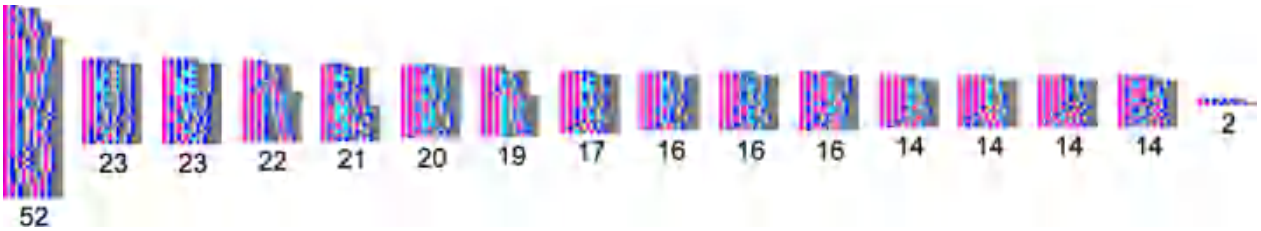
直到大小为 5，总共有 100 种 lambda 形式——进化为 68 个不同的最终结果——并且有 4 个多 lambda 等价类。

```
[11]
[[11]1]
[[12]1]
[[21]1]
[[22]1]
[[1]11]     → [11]
[[1](11)]
[[2]11]
[1([1]1)]
[1([2]1)]
[1][11]

[1]
[[1]1]
[[2]1]
[1][1]
[[2][1]]    → [1]
[[2][2]]
[[2](11)]
[[1]][1]
[11][1]

[[2]]
[[[1]2]]
[[[3]1]]
[[[3]2]]
[[[2]]1]    → [[2]]
[[[3]]1]
[[1][2]]
[1][[2]]

[[1]]
[[[1]1]]
[[[2]1]]
[[[2]2]]
[[[1]]1]    → [[1]]
[[1][1]]
[[2]][1]
[1][[1]]
```

我们可以用图形来概括这一点：

在规模不超过 6 的情况下，总共有 679 种 lambda 形式——这些形式会演变成 392 个不同的最终状

态（以及一个“循环 lambda”）——并且有 16 个多 lambda 等价类：



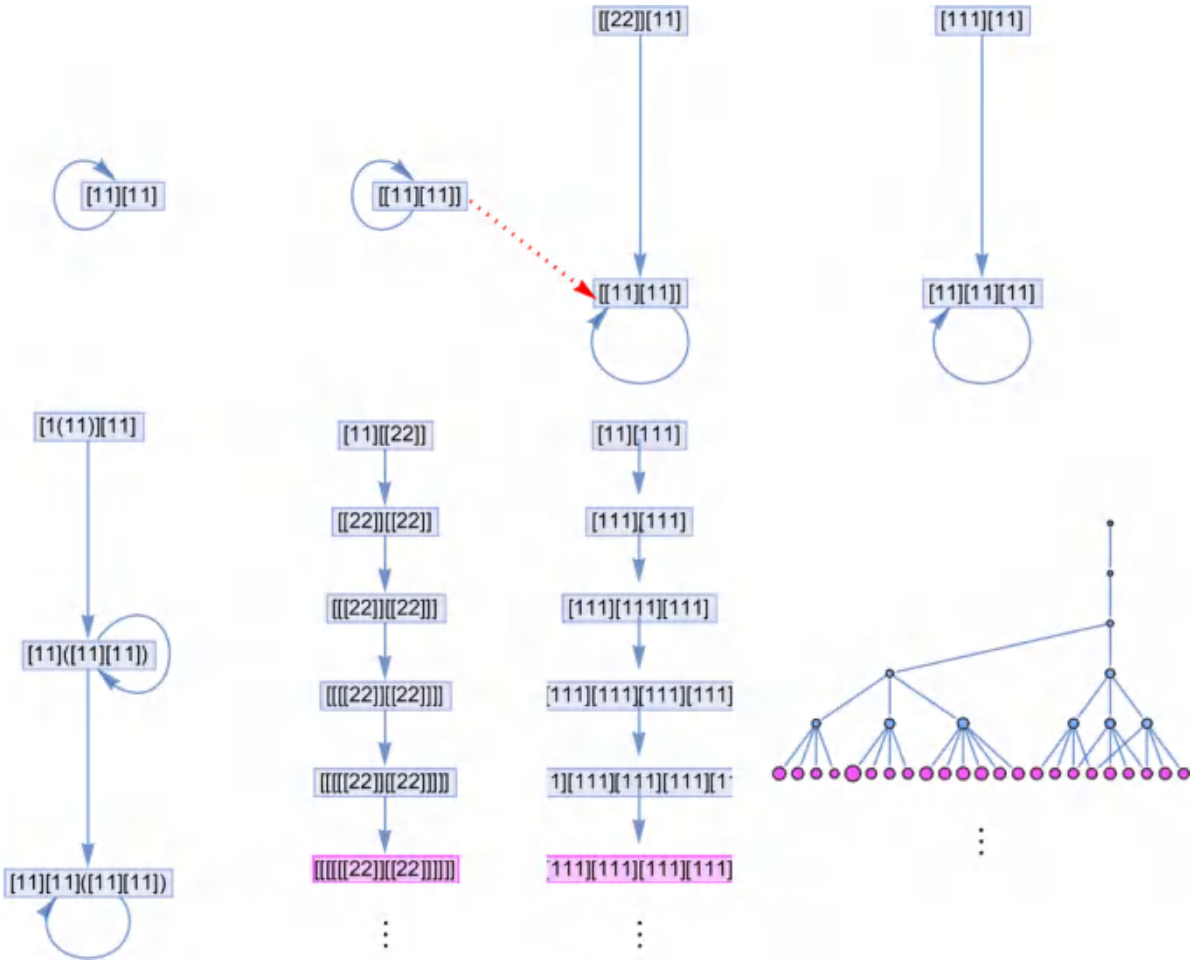
如果我们只考虑那些求值能够终止的 lambda，那么我们有以下情况：

maximum size	2	3	4	5	6	7	8	9
inequivalent lambdas	1	4	15	68	392	2757	22721	212621
total lambdas	1	4	18	100	679	5420	49397	503680
ratio	1	1	0.833	0.68	0.577	0.509	0.46	0.422

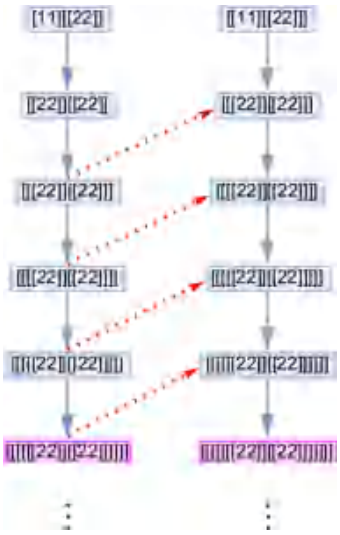
对于较大的规模，这个比率理论上会或多或少地指数级地趋近于一个固定的极限值，大约是1/4。

但这只是针对会终止的 lambda 而言的。那么对于不会终止的 lambda 之间的等价性，我们又能说什么呢？嗯，这取决于我们如何定义“等价性”。一个定义可能是：如果两个 lambda 在其演化过程中生成相同的 lambda 表达式，那么它们是等价的。换句话说，它们的_multiway graphs_之间存在某种重叠。

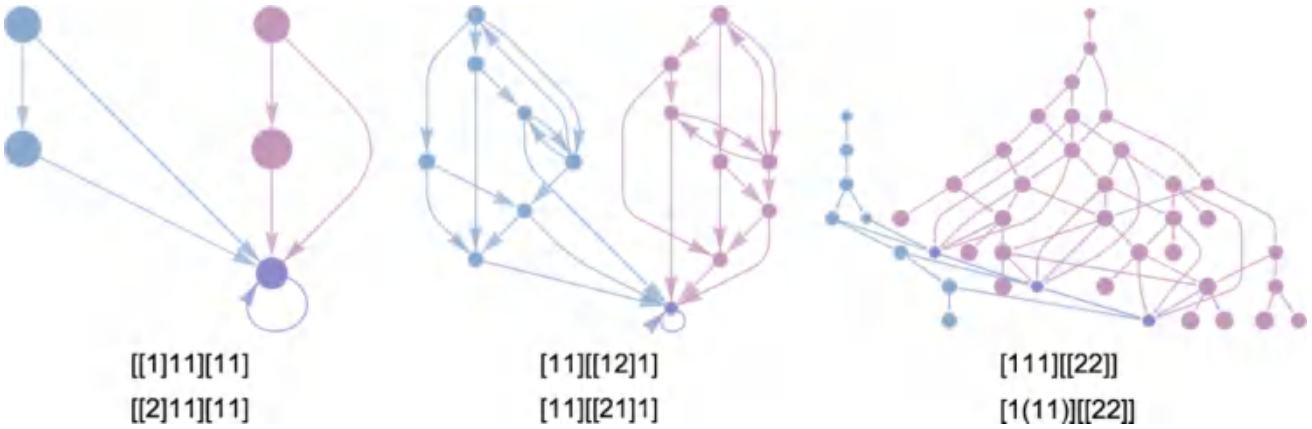
如果我们查看大小为 7 的情况，有 8 个不终止的 lambda，它们的多重系统之间存在一个单一的——相当简单——对应关系：



在大小为 8 之前，有 89 个 lambda 其求值不会终止。在这些 lambda 中，有许多对应关系。一个简单的例子是：



更复杂的例子包括



现在我们展示了用于 lambda 对的多路图组合。在前两种情况下，重叠出现在循环 lambda 上；在第三种情况下，它出现在一系列 lambda 表达式中。

如果我们查看所有 89 个非终止 lambda，这显示了其中 434 对重叠的 lambda 是哪些：

这里有几个微妙之处。首先，两个 lambda 是否“被视为等价”可能取决于所使用的求值策略。例如，一个 lambda 可能在一个策略下终止，但在另一个无法终止的策略中“错过固定点”。

此外，即使两个 lambda 的多向图重叠，也没有保证在任何给定的求值策略下都能“发现”这种重叠。

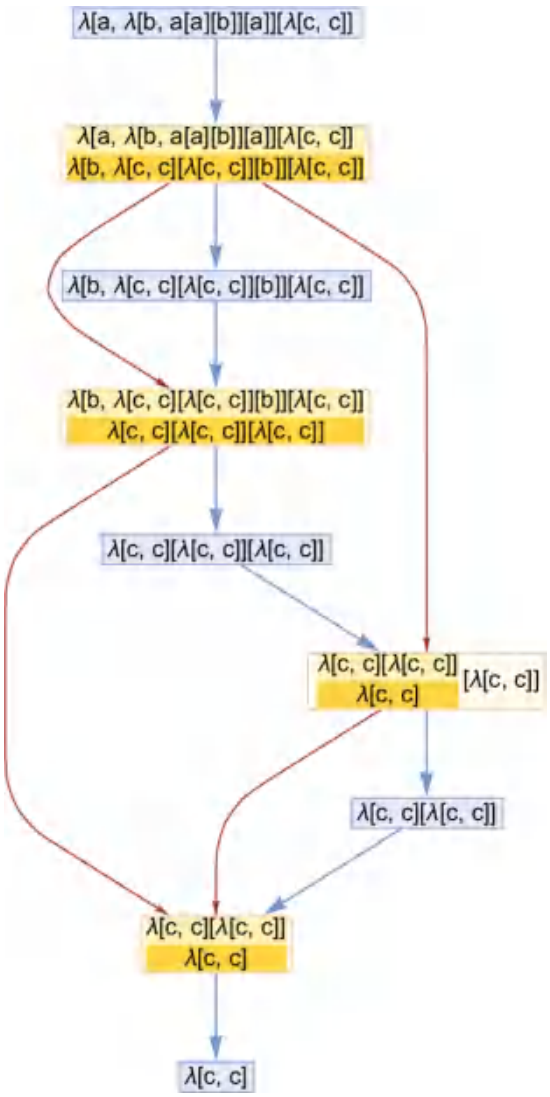
最后，和往常一样，存在一个不可判定性的问题。即使一个 lambda 会终止，它也可能需要任意长的时间来完成。更糟糕的是，如果逐步生成两个 lambda 的多向图，判断它们是否重叠可能非常困难。确定一个 lambda 的单个求值路径可能是计算不可约的；确定这类重叠可能是多计算不可约的。

Lambda 的因果关系

我们可以将 lambda 求值中的每一个 beta 还原视为一个“事件”，在这个事件中，某些输入被转化为特定的输出。如果事件 V 的输入依赖于事件 U 的输出，那么我们可以称 V 在因果关系上依赖于 U。换句话说，事件 V 只有在事件 U 发生之后才能发生。

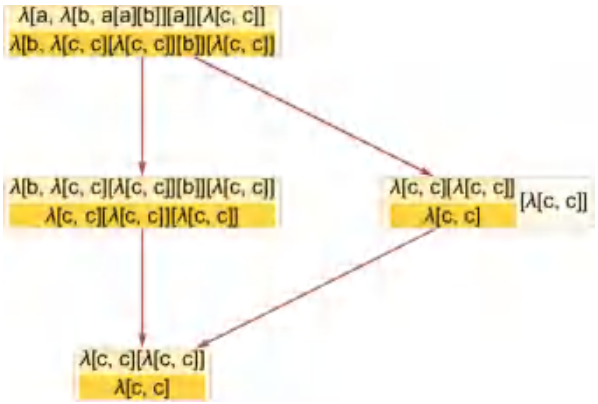
如果我们考虑一个涉及多个 beta 还原的 lambda 求值过程，我们就可以想象构建出一个完整的 “因果图”依赖关系 。

作为一个简单的例子，考虑这个求值链，我们明确显示了 beta 还原事件，并且（在这个简单情况下为了清晰）我们还避免了重命名出现的变量：



这里的蓝色边显示了如何一个 lambda 转化为另一个 lambda——这是由于一个事件的结果。深红色边显示了事件之间的因果关系；每一个都可以被认为是与一个子表达式相关联，这个子表达式是“因果的载体”，从一事件传递到另一事件。

保留事件和因果边，这个演化的因果图是：

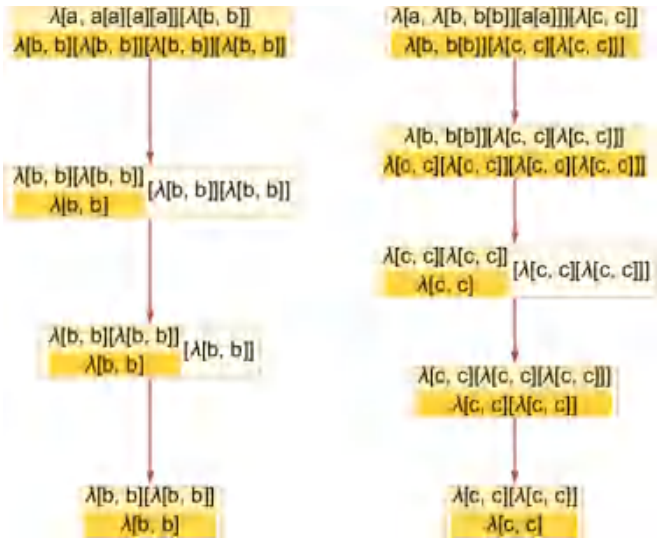


如果我们沿着因果边序列前进，我们会得到一个必须按顺序发生的“类时间”事件链。但在其“横向”上，我们可以（在偏序集的语言中）有一个“反链”，由我们可以认为是“类空间分离”的事件组成，这些事件可以并行发生。在这种情况下，第一个事件（即第一个 beta 归约）导致表达式

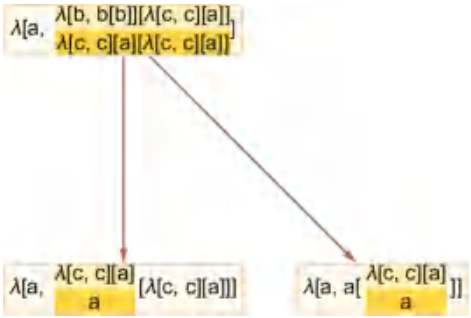
λ [b, λ [c, c] [λ [c, c]] [b]] [λ [c, c]]

其中可以进行两个归约：对于 λ[b, ...] 和对于第一个 λ[c, ...] 。但这些归约完全相互独立；它们可以被认为是“类空间分离”的。在我们的标准求值方案中，这些事件按确定顺序发生。但因果图显示的是，我们得到的结果并不需要这个顺序——实际上允许这些事件“并行”发生。

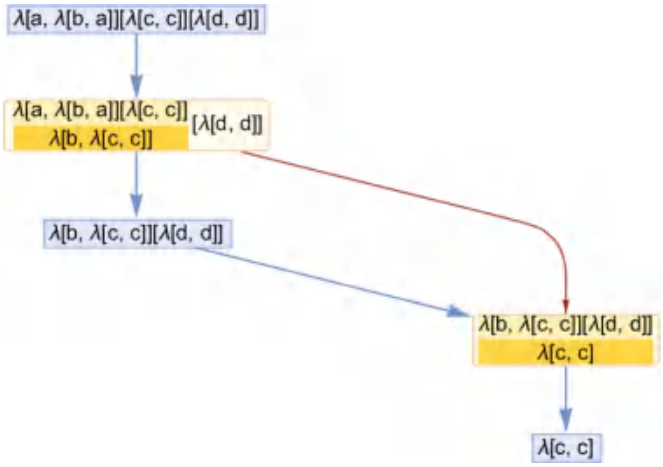
许多 lambda 在其评估中不允许并行化，因此具有简单、线性的因果图：



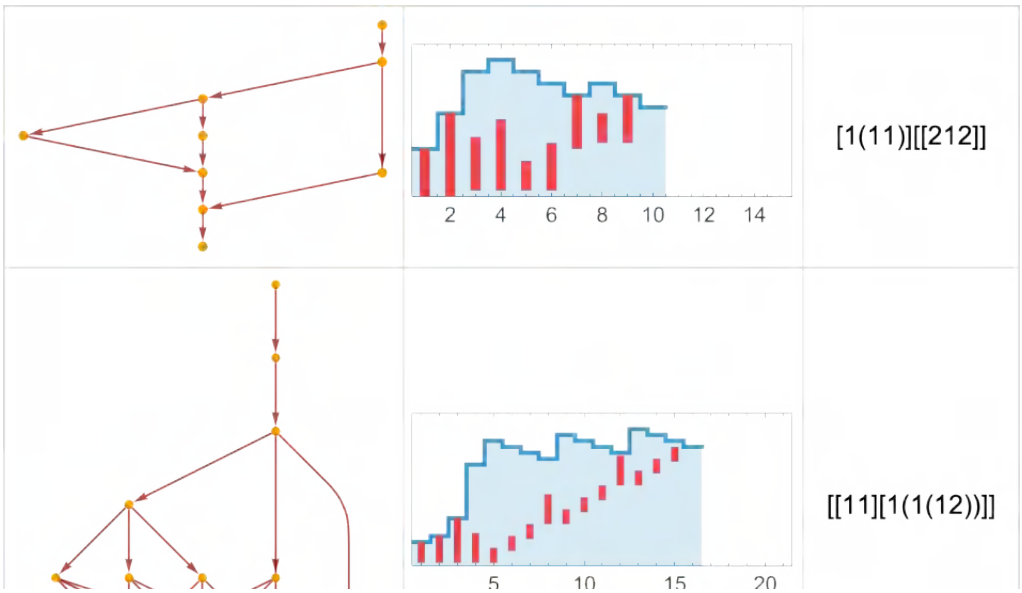
因果图中通常有简单的分支——这与λ表达式中的某些部分有关，这些部分在β归约方面完全分离，并且可以单独评估：

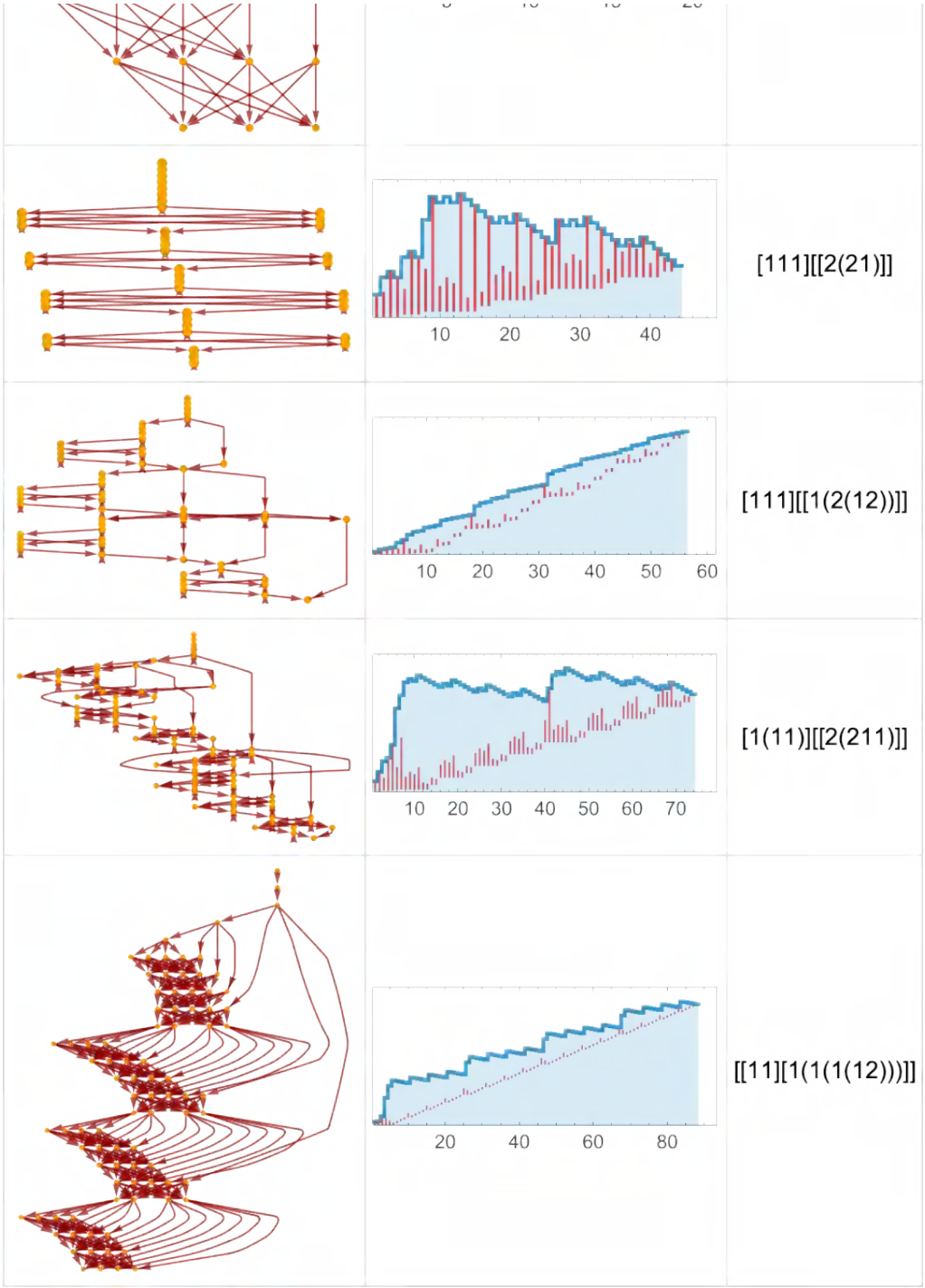


在构建 lambda 的因果图时，一个棘手的问题是弄清楚究竟是什么依赖于什么。在这种情况下，可能看起来这两个事件是独立的，因为第一个事件涉及 λ[a, ...]，而第二个事件涉及 λ[b,...] 。但是，第二个事件不能在第一个事件中通过 λ[a, ...] 的归约“激活” λ[b, ...] 之前发生——因此，第二个事件必须被视为在因果图中对第一个事件有因果依赖：

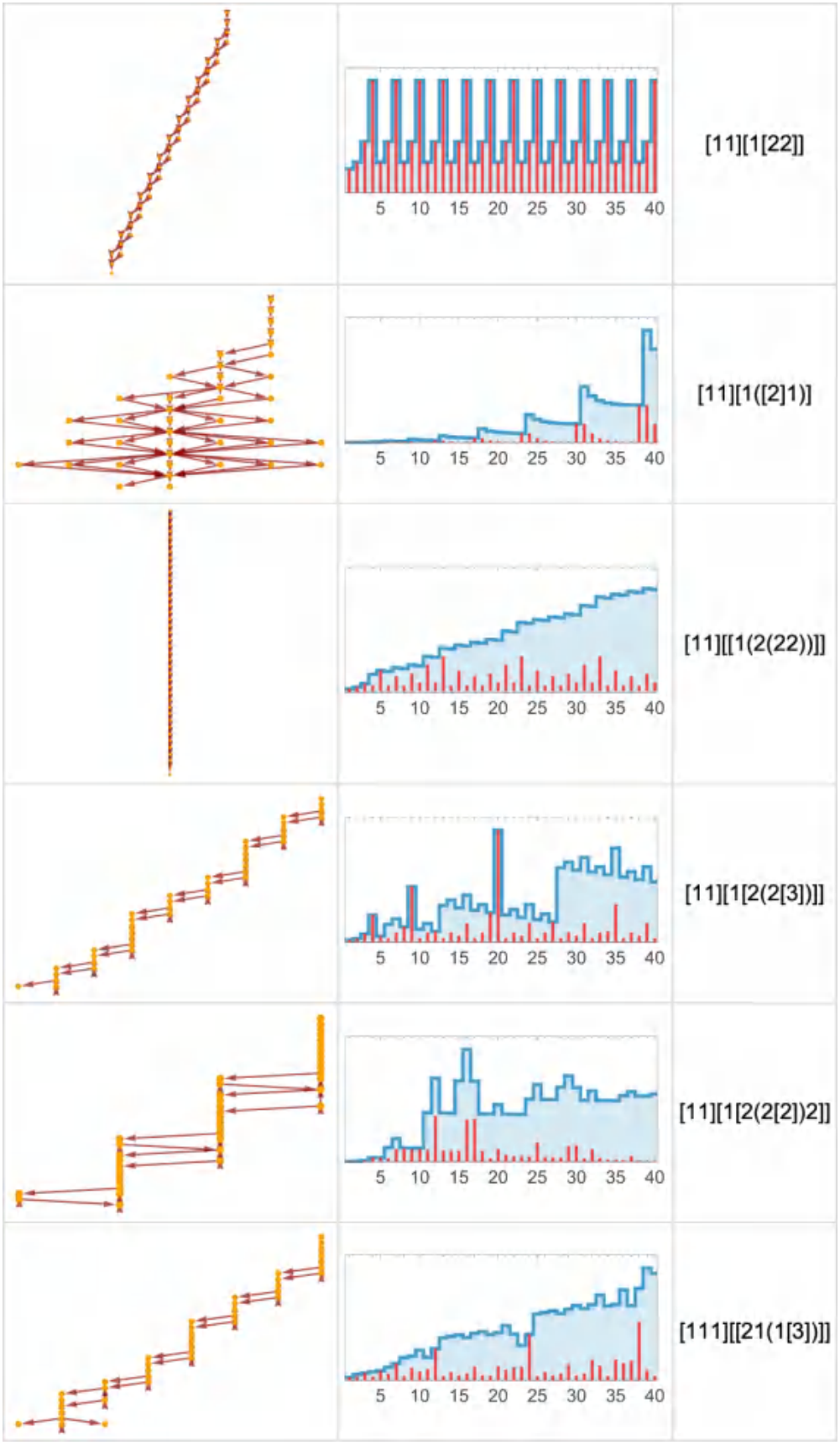


虽然许多 lambda 具有简单、线性或分支的因果图，但许多 lambda 具有更复杂的结构。这里是一些终止 lambda 的例子：





不会终止的 lambda 具有无限长的因果图：



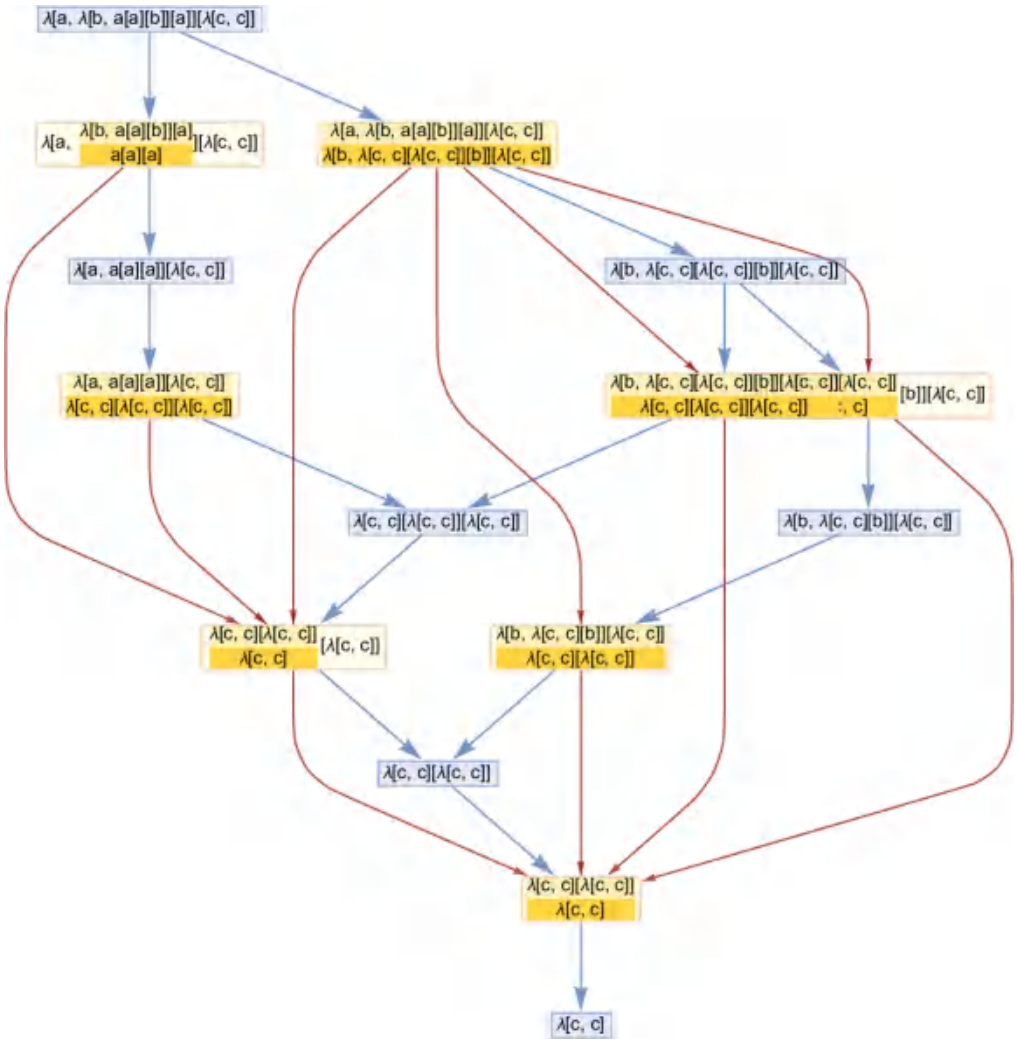
但这里值得注意的是，即使演化——以 lambda 大小的序列形式可视化——并不简单，因果图通常仍然相当简单且重复。是的，在某些情况下，看到这一点可能确实等同于证明 lambda 的演化不会终

止。但需要注意的是，即使对于会终止的 lambda，我们在上文看到它们的因果图仍然可能看起来基本上是重复的——直到它们终止。

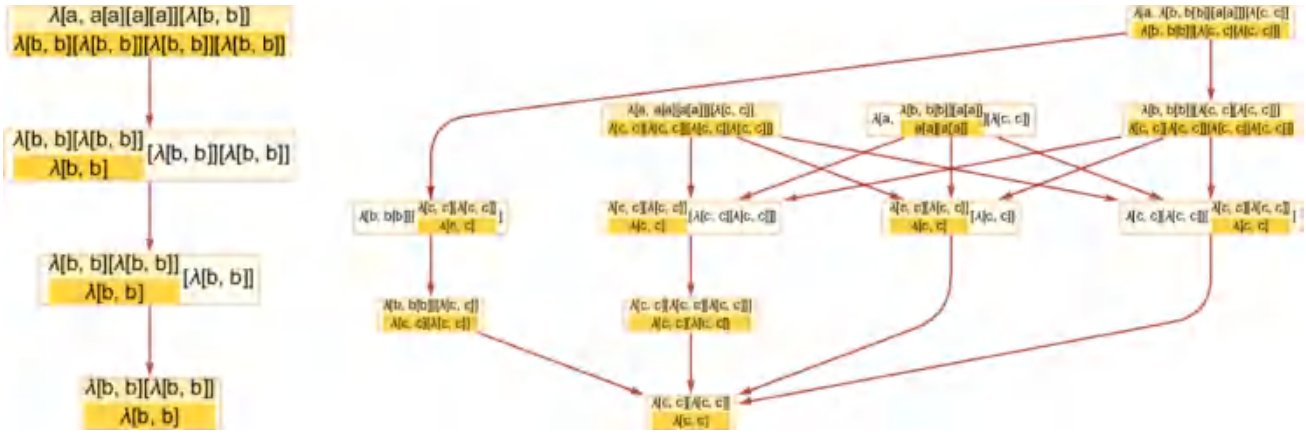
多路因果图

我们在上一节展示的因果图都是基于分析特定 lambda 求值链中的因果依赖关系，这些因果图是通过我们的标准（最左外层）求值方案生成的。但是——就像在我们的物理项目中一样——也可以生成多路因果图，在这些图中，我们展示了多路图中所有可能求值序列所代表的所有状态之间的因果连接。

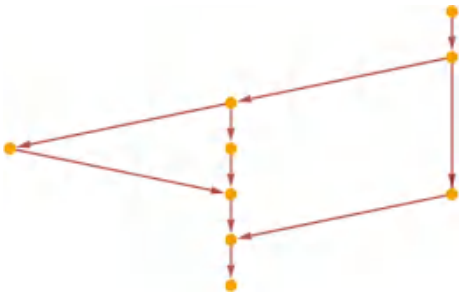
这是我们上面展示的第一个因果图的多元方式版本：



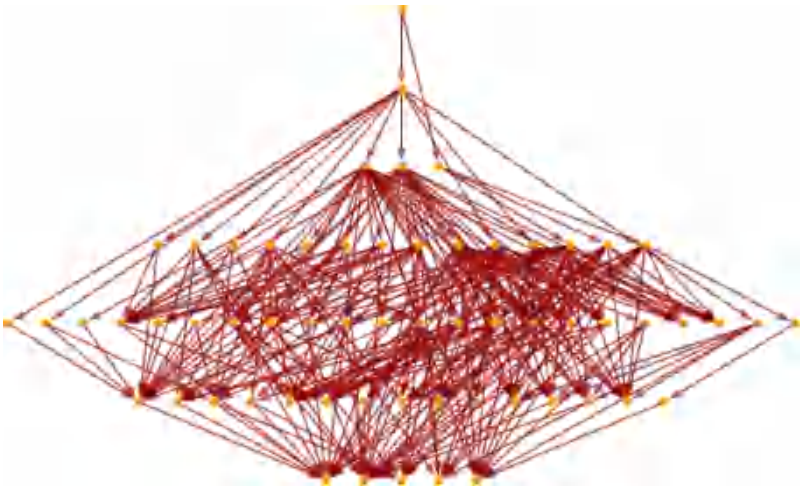
有些因果图即使在多重情况下仍然是线性的，但其他图会分裂成许多路径：



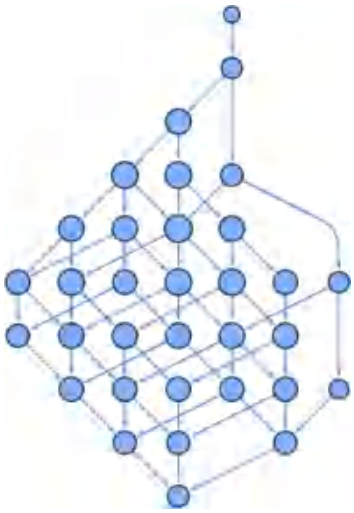
多重因果图通常比普通因果图复杂得多。例如，虽然[1(11)][[212]]的普通因果图是



多重版本是



当状态的多重路径图是：



所以，是的，可以为 lambda 制作多向因果图。但它们意味着什么？这是一个有点复杂的故事。但我不会在这里详细说明——因为它基本上与组合子的故事相同，我几年前讨论过 。尽管 lambda 不允许人们像我们为组合子考虑的那样建立共享子表达式 DAG，但 lambda 中的变量（或 de Bruijn 索引）实际上会导致长程相关性，从而阻止我们为组合子使用的那种模块化树状方法。

线性 lambda 的案例

我们已经看到，一般来说，lambda 可以完成非常复杂的事情。但存在一类特殊的 lambda——"线性 lambda"——它们的行为方式要简单得多，这使得它们可以被相当完整地分析。线性 lambda 是指在其体中每个变量只出现一次的 lambda。这意味着 **λ**[a, a] 是一个线性 lambda，但 **λ**[a, a[a]] 不是。

因此，前几个线性 lambda 是

$$\{\lambda[a,a],\ \lambda[a,\lambda[b,a[b]]],\ \lambda[a,\lambda[b,b[a]]],\\ \lambda[a,\lambda[b,b][a]],\ \lambda[a,a[\lambda[b,b]]],\ \lambda[a,a][\lambda[b,b]],\ \ldots\}$$

或以 de Bruijn 索引形式：

$$\{\lambda[1],\ \lambda[\lambda[2[1]]],\ \lambda[\lambda[1[2]]],\ \lambda[\lambda[1][1]],\ \lambda[1[\lambda[1]]],\ \lambda[1][\lambda[1]],\ \ldots\}$$

线性λ有一个重要的性质，即在β归约下它们的大小永远不会增加。（如果没有一个变量的多次出现，β归约只是移除子表达式，并且永远不会导致子表达式的复制。）实际上，在线性λ的评估的每一步，最多只会移除一个λ：

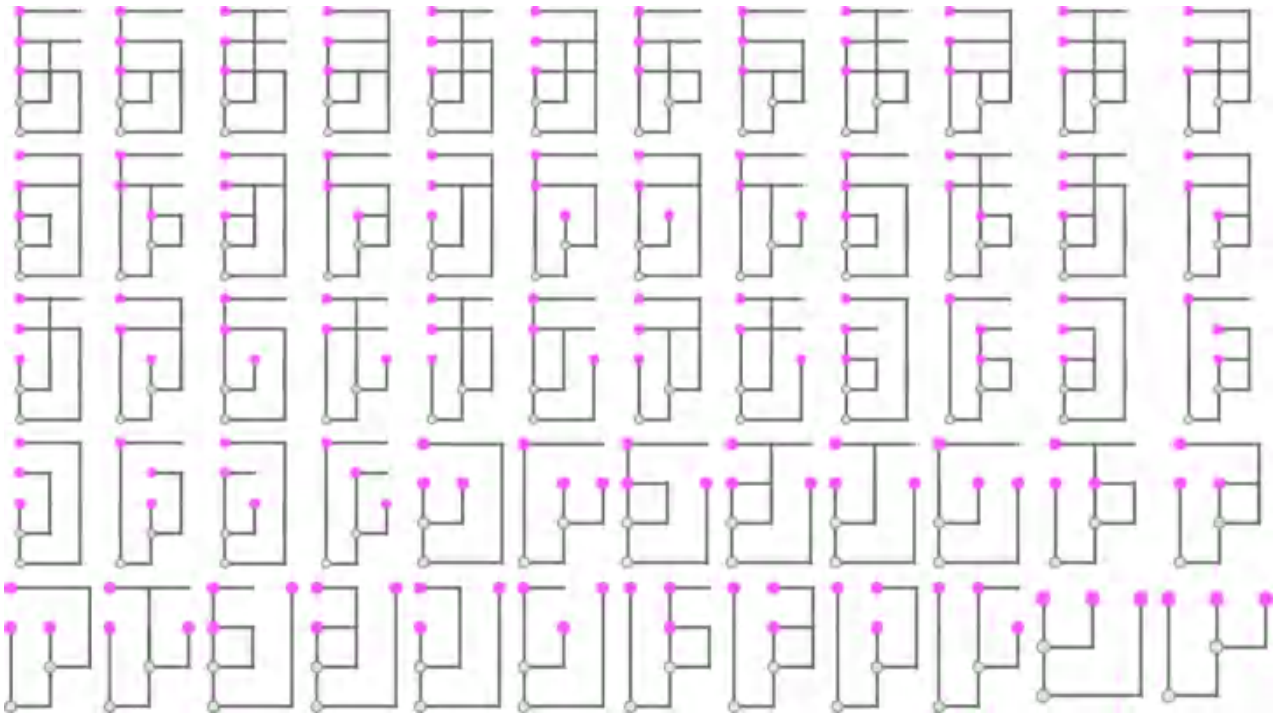
$$\lambda[a,a][\lambda[b,b]] \longrightarrow \lambda[a,a]$$
$$\lambda[a,\lambda[b,b[a]]][\lambda[c,c]] \longrightarrow \lambda[a,\lambda[b,b][a]][\lambda[c,c]] \longrightarrow \lambda[a,\lambda[b,b][a]] \longrightarrow \lambda[a,a]$$
$$\lambda[a,\lambda[b,\lambda[c,c[b]]][a]]][\lambda[d,d]] \longrightarrow \lambda[a,\lambda[b,b[a]]][\lambda[c,c]]] \longrightarrow \lambda[a,\lambda[b,b][a]]] \longrightarrow \lambda[a,a]$$

由于线性λ总是有一个变量对应每个λ，所以它们的大小总是偶数。
与给定大小的λ总数相比， 线性λ的数量很小：

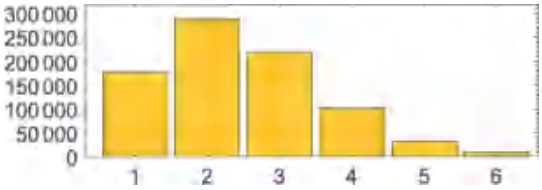
size	2	3	4	5	6	7	8	9	10	11	12
linear	1	0	5	0	60	0	1105	0	27120	0	828250
total	1	3	14	82	579	4741	43977	454283	5159441	63782411	851368766

渐近地，大小为 2*m* 的线性λ的数量增长如同 $3 \times 6^m \, m!$ 。

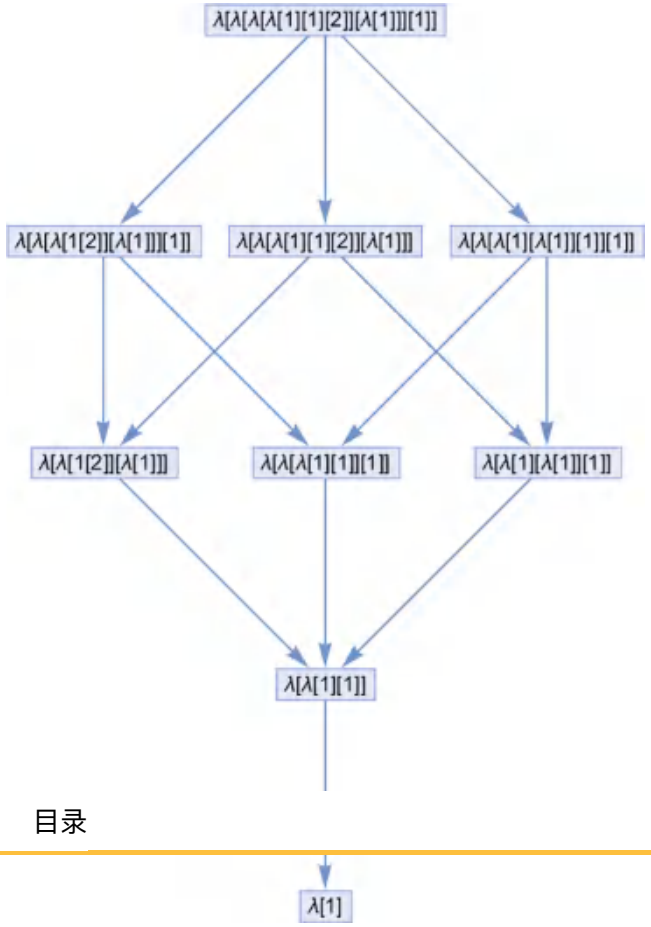
以下是大小为 6 的线性λ的 Tromp 图：



对于线性λ，评估总是会终止；它所需要步骤的数量与评估方案无关，其最大值等于λ的数量；大小为6的线性λ的分布为：



线性λ的多重图相当简单。由于每个λ与其他任何λ都是独立的，因此它们在任何顺序下进行归约都不重要，多重图由一系列菱形组成，例如：



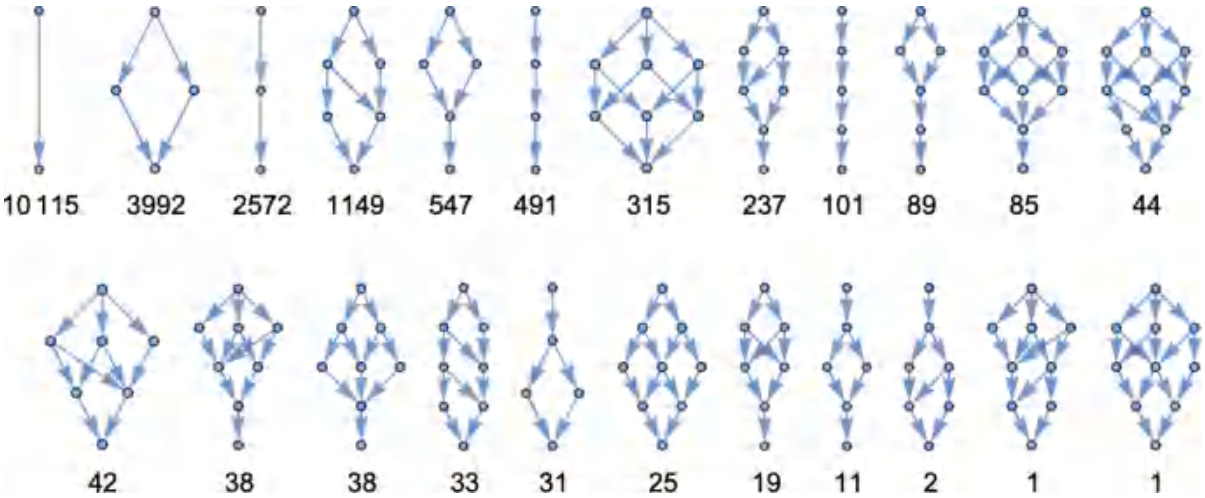
目录

对于大小 6，可能的 Multway 图（及其多重性）是：

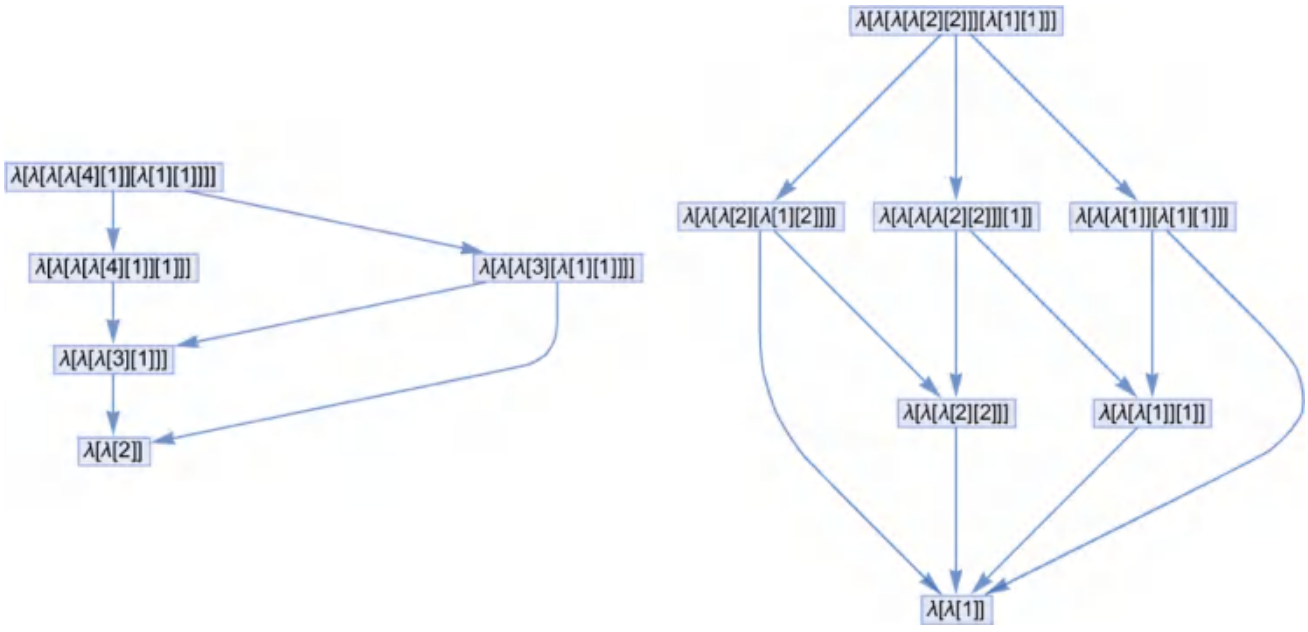
对于大小8，它是：

在最后一个图中可以更明显地对称地绘制为：

而对于尺寸为10的情况是：



除了线性 lambda，我们还可以研究仿射 lambda，仿射 lambda 被定义为允许每个变量为零个或一个实例。仿射 lambda 的行为与线性 lambda 非常相似，只是现在在多路图中出现了一些“退化菱形结构”，例如：



Lambda 与组合子

那么，在我们探索了这么多关于 lambda 的 ruliology 之后，它与我几年前发现的组合子的 ruliology 相比如何呢？许多核心现象当然是非常相似的。事实上，lambda 和组合子之间很容易转换——尽管一个尺寸的 lambda 可能对应一个非常不同尺寸的组合子。

例如，size-4 lambdas 具有这些等价组合子

λ[λ[λ[1]]]	K[K[S[K][K]]]
λ[λ[λ[2]]]	K[S[K[K]][S[K][K]]]
λ[λ[λ[3]]]	S[K[K]][S[K[K]][S[K][K]]]
λ[λ[1[1]]]	K[S[S[K][K]][S[K][K]]]
λ[λ[1[2]]]	S[K[S[S[K][K]]]][S[K[K]][S[K][K]]]
λ[λ[2[1]]]	S[S[K[S]][S[K[K]][S[K][K]]]][K[S[K][K]]]
λ[λ[2[2]]]	S[K[K]][S[S[K][K]][S[K][K]]]
λ[λ[1][1]]	S[K[S[K][K]]][S[K][K]]
λ[λ[2][1]]	S[S[K[K]][S[K][K]]][S[K][K]]
λ[1[λ[1]]]	S[S[K][K]][K[S[K][K]]]
λ[1[λ[2]]]	S[S[K][K]][S[K[K]][S[K][K]]]
λ[1[1][1]]	S[S[S[K][K]][S[K][K]]][S[K][K]]
λ[1[1[1]]]	S[S[K][K]][S[S[K][K]][S[K][K]]]
λ[1][λ[1]]	S[K][K]

而大小为 3 的组合子有这些等价的 lambda：

S[S][S]	[[21[31(21)]]]
S[S[S]]	[[[321(2(32)1)]]]
S[S][K]	[[212]]
S[S[K]]	[[21]]
S[K][S]	[1]
S[K[S]]	[[[[431(21)]]]]
S[K][K]	[1]
S[K[K]]	[[[32]]]
K[S][S]	[[[31(21)]]]
K[S[S]]	[[[[21(321)]]]]
K[S][K]	[[[31(21)]]]
K[S[K]]	[[[1]]]
K[K][S]	[[2]]
K[K[S]]	[[[[31(21)]]]]
K[K][K]	[[2]]
K[K[K]]	[[[[2]]]]

毫不奇怪，对于等价的组合子，lambda 的大小（ LeafCount ）往往略小，因为 de Bruijn 索引的存在意味着 lambda 实际上拥有一个更大的“字母表”来工作。

但最终，lambda 和组合子可以做“所有相同的事情”，就像在这些情况下：

smallest allowing reduction	[1][1]	K[S][S]
smallest looping	[11][11]	S
smallest growing	[11][[22]]	S[S[S]][S][S][S][S]
representation of 4	[[2(2(2(21)))]]	S[S[S[S[K]]]]
representation of addition	[[[[42(321)]]]]	S[K[S]][S[K[S[K[S]][K]]]]
representation of multiplication	[[[3(21)]]]	S[K[S]][K]

《计算等价原理》意味着，在基本层面上，那些行为并非显而易然简单的系统的规则学至少在某种程度上是相同的。

但 lambda 确实给我们带来了一些独特的挑战。首先，存在变量及其所隐含的非局部性问题。这不像细胞自动机或图灵机，在某种程度上每个更新都是局部发生的。甚至也不像组合子，其中事物至少在树上局部化——或者超图重写系统，其中事物在超图上局部化。Lambda 既具有层次性又具有非局部性。Lambda 表达式具有树状结构，但变量以任意复杂的方式将树的各个部分连接在一起。而且 beta 约简实际上可以拉扯任意长的链条。

所有这一切都与另一个问题纠缠在一起：lambda 表达式通常可以变得非常庞大。即使这样也不是一个大问题，如果不是因为没有任何明显的方法来压缩 lambda 表达式所做的事情。结果，尽管 Wolfram 语言提供了一种非常高效的方式来处理符号结构，但我在这里所做的工作经常挑战我的计算资源。

然而，最终我们至少在探索 lambda 的 ruliology 方面完成了基础工作，并且看到计算宇宙的又一个角落充满了非凡和令人惊讶的现象。我最初在 20 世纪 80 年代初就曾思考 lambda 在“野外”会是什么样子。我很高兴，现在终于——这么多年后——我终于能够感知到 lambda 的世界及其 ruliology 的独特性和同一性。

书目注释

在过去的 90 年里，关于 lambda 已经写了很多，尽管大多数情况下其焦点是形式和理论，而不是实证和规则学。当我在 2020 年写关于组合子时，我做了研究来编制一个非常广泛的组合子（规模要小得多）文献书目。对于 lambda 我没有尝试类似的练习。但鉴于组合子和 lambda 之间的重叠——以及等价性——我的大部分组合子书目也适用于 lambda。

组合子书目»

谢谢

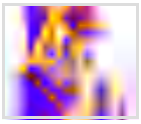
感谢沃尔夫拉姆研究所研究员尼科·穆尔津和威廉·尼尔森的鼎力相助。同时感谢与沃尔夫拉姆研究所相关的其他几位人士，尤其是理查德·阿萨和约瑟夫·斯托克。此外，对于我在这里所描述的具体内容，我要感谢亚历克斯·阿彻曼、乌塔克什·巴贾吉、马克斯·尼德曼、马修·苏迪兹基和维克多·泰莱因。多年来，我与许多人就入演算进行了交流，包括亨克·巴伦德雷特、格雷格·蔡廷、奥斯汀·蒋、扬·威廉姆·克洛普、罗曼·迈德、丹娜·斯科特和约翰·特罗姆普。在实践层面，自 20 世纪 70 年代末以来，我无数次地讨论了沃尔夫拉姆语言和 Mathematica 中的 Function 函数，以及它在 SMP 中的前身。

发布于： 计算科学 , 数学 , 一种新的科学 , 规则学

Join the discussion

相关著作

从半个世纪的“生命游戏”元胞自动机中我们能学到关于工程和创新的什么?
March 18, 2025



迈向医学基础的计算形式化
February 3, 2025

谁能理解这个证明?——正式数学的窗口
January 9, 2025

实用到革命的程度：介绍 Wolfram Notebook Assistant
December 9, 2024

热门分类

人工智能

大局观

公司与商业

计算科学

计算思维

数据科学

教育

未来展望

历史视角

语言与交流

生活与时代

生命科学

Mathematica

数学

科学的新种类

新技术

个人分析

哲学

物理学

规则学

软件设计

Wolfram|Alpha

Wolfram Language

其他

按年份著作

2025 | 2024 | 2023 | 2022 | 2021 | 2020 | 2019 | 2018 | 2017 | 2016 | 2015 | 2014 | 2013 | 2012 | 2011 | 2010 | 2009 | 2008 | 2007 | 2006 | 2004 | 2003 | 所有

© Stephen Wolfram, LLC |
Open content: (code:) W | Terms | RSS