证明与计算：无限鸽巢原理与可数选择

Zena M.Ariola Paul Downen俄勒冈大学 马萨诸塞大学洛厄尔分校

Hugo Herbelin巴黎西岱大学，Inria，法国国家科学研究中心，IRIF

结构递归是现代编程语言中程序员常用的一种技术，也是计算机科学入门课程教授的内容。那么，它的对偶——结构余递归又如何呢？结构余递归是一种优雅的技术，受到 Haskell 等语言以及 Rocq 或 Agda 等证明助手的支持。它通过将潜在无限或规模庞大的数据集合的生成与消费解耦，使组合式算法的设计成为可能。尽管有这些优点，结构余递归通常仍被视为比结构递归更高级的技术，并且主要在纯函数式编程的语境下进行研究。

我们的目标是阐明在经典推理的背景下，不同结构余递归概念的表达能力。更具体地说，我们研究余迭代和余递归与经典 callcc 运算符的结合；该运算符为经典推理提供了计算解释。这种结合能够实现有趣的流处理算法。作为应用，我们给出一个基于控制的无限鸽巢原理余递归证明，并将其与 Escard´o 和 Oliva 在 Agda 中的延续传递证明进行比较。为了进一步展示余递归与控制相结合的能力，我们给出可数选择公理的一种实现。与该公理通常的延续传递实现不同，后者依赖一般递归并由外部论证其终止性；我们的方法仅凭余迭代即可证明终止性。

1 引言

递归。对递归数据类型的结构进行递归，是设计实用程序的一种常见原则。这一概念植根于自然数归纳，数十年来一直卓有成效地应用于程序设计。如今，人们对它的理解已足够深入，因此它常被作为通用算法设计方法教授给计算机科学专业的初学者 [11]。结构递归的优势在于，它提供了一个处理任意大小输入数据的模板：将输入较小子部分所得的结果组合起来，同时保证这一过程最终会以一个答案结束。例如，最基本的结构递归形式是对数字进行“原始递归”——它对应于“原始归纳”，本文简称为“递归”——规定递归调用只能针对输入的直接前驱。它不同于“一般递归”：后者对递归调用不作任何限制，因此既不为程序设计提供指导，也不保证终止性。此外，结构递归技术远不止适用于数字，还能用于任何归纳定义的数据类型上的算法，从列表到几乎任何形状的有限树皆可。

余递归。那么，结构递归的自然对偶——结构余递归又如何呢？余递归已用于余代数的若干应用中 [21, 28]。但这与将结构递归本身作为一种独立的实用程序设计技术来介绍，大不相同。递归的结构直接遵循程序输入的结构，而

T. Altenkirch、F. Barbanera、F. Damiani 和 U. de’Liguoro（编）：《逻辑与类型论：献给 Stefano Berardi 的论文集》，献于其 64 岁生日之际，EPTCS441，2026年，第 1–18 页，
doi:10.4204/EPTCS.441.1

© Z. M. Ariola、P. Downen 和 H. Herbelin 本作品依据知识共享署名许可协议授权。

余递归的结构则直接遵循程序输出的结构 [12]。我们旨在展示余递归如何提供恰当的抽象，以便计算无限序列。

余递归的模式。结构递归有许多不同的表述方式，各有利弊，会影响易用性、表达能力和计算复杂度等方面。例如，与原始递归相对应的递归方案有时称为“参数形态”[25], ，而“折叠态射” [19]有时用于指普通迭代。识别并研究这些不同的递归方案，可以发掘出一系列定律和定理，供编译器使用，例如折叠态射融合律，它可以消除中间结构 [24]。同样，结构余递归也有不同的表述方式，各有类似的利弊。从给定种子展开的原始余迭代称为“展开态射” [26]；可以随时停止计算的原始余递归称为“阿波形态” [30]。这些余递归概念通常在 Haskell [14, 20, 13], 等纯函数式语言的语境下研究，在这类语言中，无限对象与有限对象的类型被混为一谈。本文则转而在 Rocq 证明助手的语境下研究各种余递归方案的表达能力，尤其是在存在经典原理的情况下。

无限鸽巢原理。作为余递归和控制的一个应用，我们关注无限鸽巢原理的计算内容。该原理断言，任何由布尔值 true 和 false 构成的序列都存在一个常值子序列。这个证明是经典的，也就是说，在计算上需要使用 callcc 之类的控制算子。证明利用余递归逐步并行地构造 true 的子流和 false 的子流：由于原序列是无限的，至少有一个子流会产生任意多个元素。作为比较，一个仅用余迭代表述的类似证明无法得出相同结果。在这两种情况下，我们都遵循 Berardi 等人研究的程序提取方法 [2, 6] ，从而得到一个程序。

经典可数选择。可数选择公理断言，对于可数域上的任何全关系，都可以提取出一个函数，为每个输入选择一个见证，以证明该关系是全的。经典逻辑的存在增强了可数选择公理的力量。事实上，当通过双重否定翻译来解释经典逻辑，以保留直觉主义逻辑的构造性特征时，可数选择公理会扩展为更强的双重否定的可数选择公理：

$$\forall n.\neg\neg\exists x^{A}.R(n,x) \to \neg\neg\exists f^{\mathsf{nat}\to A}.\forall n.R(n,f(n)) \ .$$

从 Spector [29], 开始，已有多种实现形式给出了可数选择公理的这种双重否定形式，其中尤以 Berardi 等人 [5] 的实现为代表。这些实现依赖一种其终止性被假定成立的一般递归。相比之下，若通过控制算子直接实现经典逻辑，则无需一般递归，只需余迭代即可！这一点最初由 Griffin [15] 展示，后来也由 Berardi 等人 [2, 3, 4]等人进一步研究。

概述。我们首先介绍 Rocq 中流的定义（第2节），接着讨论不同的余递归模式（第3节）。我们从余迭代开始，然后介绍一种允许停止余递归的概念。接下来，我们讨论一种表达能力更强的概念，它借助控制，允许保存一个控制点，以便稍后恢复。作为应用，我们给出一个程序：它接收一个布尔流并返回一个常值子流；我们还将说明，该程序对应于我们对

无限鸽巢原理证明的计算内容（第4节）。随后，我们将我们的证明与 Escard´o 和 Oliva 的 Agda 证明 [10] 进行比较，并提供一种基于余迭代而非余递归的替代表述（第5节）。我们通过给出可数选择公理的直接证明，展示余迭代／余递归与控制的威力（第6节）。在总结之前，我们将探讨表述无限鸽巢原理的不同方式（第7节）。所有 Rocq 代码均可在代码仓库 infinite-pigeonhole 中找到。

这项工作涉及 Stefano Berardi 的三项重要贡献：用经典逻辑进行计算、用可数选择公理进行计算，以及从证明中提取程序。

2 Rocq 中的流

在 Rocq 中，我们根据流的观察或解构方式来定义流：

```
余归纳 流 A: = { 头部: A; 尾部: 流 A }。
```

符合上述类型的零无限流如下：

```
CoFixpoint zeroes: = {| 头部: = 0; 尾部: = zeroes  |}。
```

无论深入多少层，(head (tail...(tail (tail zeroes))...))始终会返回0。我们还可以将zeroes 推广到任何一个恒定流 返回相同值x的流：

```
CoFixpoint always {X} (x : X): = {| 头部: =x; 尾部: = always x  |}。
```

或者通过反复repeat应用某个函数 f 到一个初始值 x来构造一个流：

```
CoFixpoint repeat {X} f (x : X): = {| 头部: = x; 尾部: = repeat f (f x)  |}。
```

我们还可以构造一个从给定数字n 开始递增计数的流（即n (n+1) (n+2)...），或者递减计数的流（即 n (n-1) ... 1 0 0 ...）：

```
CoFixpoint countUp n: = {|头部: =n; 尾部: = countUp (n+1)  |}。CoFixpoint
```

间奏 1。由于我们关注的是提取，因此要指出当前 Rocq 针对按值调用语言的提取机制存在的一个问题。我们聚焦于下面这个简单的自然数流示例，并使用对 4 和 4 求阿克曼函数的值，来代替像 Rocq 这样仅包含终止函数的语言中的无限计算：

```
不动点定义 ack n: = 匹配 n与
                    | 0   => S
                    | S n  => 固定 ackSn m: =匹配 m 与
                                          | 0   => ack n 1
                                          |S m => ack n（ackSn m）
                                        end
                  end.
CoFixpoint z n: ={| 头部: = n; 尾部: = z (ack 4 4)  |}。
```

流作为一种无限对象，无法直接观察。不过，我们可以使用一个取前缀操作来观察它的有限部分，该操作会取出 个元素：

```
不动点定义 取前缀 {A} (s : 流 A) n: =匹配n 时
```

我们原本预计，提取并运行 `take (z 0) 1` 后会立即返回 `0`，因为我们只关心流的头部，而不关心尾部，但实际上它计算了很长时间。这是因为我们的 `take` 函数即使在 `n` 等于 1 时，也会强制计算尾部。通过将 `stream` 的共归纳定义提取为 OCaml 代码，我们可以更好地理解其中发生了什么：

```
类型 'a 流 = 'a __流 Lazy.t以及'a __流 = | 构造_流 的'a * 'a 流递归定义 z n = 惰性 (构造_流
```

正如预期，流是一种延迟结构。然而，根据 OCaml 的按值调用语义，对 `(z 0)` 求值会强制计算 构造_流 的两个组成部分；这解释了观察到的行为。实际上，如果将 `z` 的提取结果改为下面这样，`take (z 0) 1`就会立即返回：

```
令 rec z n =
  lazy (Build_流 (n, lazy (Lazy.force (z (ack (S (S (S (S O)))) (S (S (S (S O))))))))                    )
```

我们该如何继续？在按值调用的环境中，我们习惯于延迟尾部的计算，因此会自行确保惰性，并将流定义为：

```
余归纳 流 A: = { 头部: A; 尾部: 单位类型 -> 流 A }。CoFixpoint z
```

我们还将取前缀 函数重新定义为

```
不动点定义 取前缀 A (s: 单位类型 -> 流 A) n: =匹配n 与| 0 =>
```

现在，我们的 取前缀 `(fun _ => z 0) 1` 会按预期执行。作为 取前缀 的另一种变通方法，我们也可以避免延迟参数，改用以下定义：

```
不动点定义 取前缀 {A}(s : 流 A) n: =匹配n 与
```

注意 `n=1` 的特殊情况；否则，流的尾部就会被强制求值，即使我们只关心头部，也会遇到同样的问题：即便只想取得头部，也会求值 `(ack 4 4)`。

更一般地，我们可能会问，在构造流时，头部 的求值是否也应该延迟。的确，既然余归纳类型是一种计算类型 [23],，按照这一思路，我们是否会预期，当只请求尾部的 头部 时，如下述 `z'` 的定义所示，`head (tail (z' 4))` 的求值会跳过对 `z' 4` 的头部进行代价高昂的求值：

```
CoFixpoint z'n: = {|头部: = ack n n;尾部:
```

由于我们的应用会严格求值头部，因此我们不会延迟对它的求值。

3 Rocq 中的余递归模式

余迭代。 借助 CoFixpoint 和自引用，我们可以创建各种各样的无限流。然而，我们的目标是抽象出不同的递归方式。在前面的所有示例中，我们递归时要么使用相同的实际参数值（如 zeroes 流中那样），要么使用更新后的值。生成流的这种常见模式由余迭代捕获，它使用内部状态 来即时生成 头部。为了生成尾部，状态 会被更新，然后余迭代继续进行。

```
定义 coiter {X} {Y} (base:X -> Y)(next:X -> X) : X -> 流 Y: =
 余不动点 f s : 流: ={| 头部: = base s;
                         尾部 := f (next s)
                      |}.
```

因此，我们可以将目前为止看到的一些流定义改写如下：

```
定义 恒定流 {X} (x : X) := coiter (函数 x=>x)(函数 x=>x) x.定义 countUp
```

此外，将映射 函数 f 应用于流的所有元素这一操作，是coiteration 的一个实例：

```
定义 maps {X} {Y} (f:X -> Y)xs := coiter (函数 xs=>f(头部 xs))(函数 xs=>尾部 xs) xs.
```

最小余递归。但并非每个流都能仅通过coiteration 生成。事实上，我们很想把countDown 表示为

```
定义 倒计数' n :=coiter (fun x=>x)(fun x=>if Nat.eqb x 0 then 0 else x-1) n.
```

countDown 与 倒计数' 在外延上等价（仅考虑外部观察者所见），但在内涵上并不等价（计入实现内部所用的其他因素）。具体而言，倒计数'会在每一步持续检查内部种子的值，即便在种子变为 0 且不再进一步变化之后仍是如此，而 countDown 则会停止余递归并返回 zeroes 流。

为了更准确地反映 countDown 的代价，我们需要一种更一般的生成流的方式。这正是极小余递归子（亦称阿波形态 [30]）发挥作用之处：一旦流的剩余部分事先已完全可知，它便提供一条结束余递归的路径。我们可以通过让余归纳结构返回两种可能选项的和类型（亦即不交并 流+ X），从而允许余递归提前结束：要么返回类型为 X 的新种子以继续余递归，要么返回一个先前已定义的流（类型为 流）作为剩余部分。

```
定义 corecM {X} {Y} (base:X -> Y)(next:X -> (流 Y) + X): X -> 流 Y :=余不动点 f (s:X):=
```

现在，countDown 的定义就成为最小余递归的一个实例：

```
定义 zeroes := coiter (fun x=>x)(fun x=>x) 0.定义
                        (fun x=>x)
                        (fun x=>if Nat.eqb x 0 then inl zeroes else inr (x-1))
                        n.
```

另一个例子表明需要停止余递归，该函数将一个有限的列表追加到无限流前面：

```
不动点定义 append l xs := {| 头部 :=match l with
                                  | []   => head xs
                                  | x::y => x
                              end;
                       tail := match l with
                                  | []   => tail xs
                                  | _::y => append y xs
                              end
                    |}.
```

这个append 函数也可以定义为最小corec的一个实例：

```
定义 append' {X} (l:list X) xs :=corecM(fun l=>match l with  | [] => 头部 xs  | x::y  =>
```

注意，append，中两个函数的函数体与append所用的头部和尾部分支几乎相同。区别在于余归纳步骤；当l 为空时，我们不是直接返回，而必须给结果打上标记，以表明循环已经结束。而当l非空时，我们不是通过调用自身来继续循环，而只是返回更新后的状态y。

经典余递归。上文我们已经看到，借助和类型，程序员可以控制何时结束余递归。但余递归循环一旦结束，便彻底结束了。我们还想更进一步：利用一等控制，我们可以在余递归循环的tail 步骤中提供两个延续；第一个（隐式）延续让tail 步骤更新循环状态并继续余递归，而第二个（显式）延续则捕获请求该流tail的调用者。换言之，我们希望余递归器具有如下类型：

```
corecC : (X -> Y) -> (X -> (流 Y ->B) -> X) ->X -> 流 Y
```

基础函数与之前相同；不过，下一个函数可以访问一个延续。我们如何用 corecM 来定义 corecC？有趣的是，在经典逻辑下，我们可以证明以下判断

$$(\neg S \to X) \vdash S + X$$

如下

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(S+X)\vdash\neg(S+X)}\quad\dfrac{}{S\vdash S+X}\vee_I}{\neg(S+X),S\vdash\bot}\to_E}{\neg(S+X)\vdash\neg S}\to_I}{\neg S\to X\vdash\neg S\to X}\quad\dfrac{}{\neg(S+X)\vdash\neg S}\to_E}{\neg(S+X),\neg S\to X\vdash X}}{\neg(S+X),\neg S\to X\vdash S+X}\vee_I}{\neg S\to X\vdash\neg(S+X)\to S+X}\to_I \qquad \vdash(\neg(S+X)\to S+X)\to S+X}{\neg S\to X\vdash S+X}\to_E$$

该证明依赖于非构造性公理

$$(\neg(S+X)\to S+X)\to S+X$$

它是皮尔士律的一个实例（在下文证明中记作 PL）

$$(\neg A\to A)\to A$$

其计算解释是`callcc` 控制算子，如 [1]中所述。上述推导变为以下程序：

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{disjret}:\ \neg(S\ +X)\vdash\text{disjret}:\ \neg(S\ +X)}\quad\dfrac{}{\text{ret}:S\vdash\text{ret}:S}\over{\text{ret}:\ S\vdash\text{inl ret}:\ S\ +X}\vee_I}{\text{disjret}:\ \neg(S\ +X),\text{ret}:S\vdash\text{disjret(inl ret)}:\ \bot}\to_E}{\text{disjret}:\ \neg(S\ +X)\vdash\lambda\text{ret.disjret(inl ret)}:\ \neg S}\to_I}{\text{下一个}:\ \neg S\to X\vdash\text{下一个}:\ \neg S\to X}\quad\dfrac{}{\text{disjret}:\ \neg(S\ +X),\text{next}:\ \neg S\to X\vdash\text{next}\ (\lambda\text{ret.disjret(inl ret)}):X}\to_E}{\text{disjret}:\ \neg(S\ +X),\text{next}:\ \neg S\to X\vdash\text{inr(next}\ (\lambda\text{ret.disjret(inl ret)})):S\ +X}\vee_I}{\text{next}:\ \neg S\to X\vdash\lambda\text{disjret.inr(next}\ (\lambda\text{ret.disjret(inl ret)})):\ \neg(S\ +X)\to S\ +X}\to_I \qquad \vdash\text{callcc}:\text{PL}}{\text{next}:\ \neg S\to X\vdash\text{callcc}(\lambda\text{disjret.inr(next}\ (\lambda\text{ret.disjret(inl ret)})):\ S\ +X}\to_E$$

我们得到了经典`corec`的以下定义[1]：

```
定义 cont A := A -> Empty_set。参数 callcc : forall {A}, (cont A -> A) ->A。参数 throw : forall {A B},
```

与会一劳永逸地彻底结束循环的和类型不同，延续可以被多次调用。这使得我们能够随意“暂停”和“恢复”循环。与上文最简的 `corec` 的区别在于，在余归纳步骤中，`corecC` 会将尾投影的调用者捕获到延续 `disjret` 中，并将其连同当前的 `state` 一起提供给 `next` 函数。使用经典余递归器，我们可以给出 `countDown` 的另一种定义：

```
定义 countDown1 n: =corecC
                        (fun x=>x)
                        (fun x=>fun k=>if Nat.eqb x 0
                                       then throw k zeroes
                                       else x-1)
                        n.
```

[1]关键字 Parameter 只是一个类型声明，它推迟了实际的定义。

然而，正如下一节所示，经典余递归器的表达能力不仅限于停止余递归。

注2。经典余递归器不能像下面这样通过将 `coiter` 与`callcc` 结合起来定义：

```
定义 corecC {X} {Y} (base:X -> Y)(next:X -> (stream Y -> B) -> X) : X -> stream Y:=
```

请注意，延续`k` 捕获的是更新`coiter`状态的上下文，而不是`next`的调用者。相反，`corecC` 恰好向`next`提供了这一额外信息。

4 使用余递归的无限鸽巢原理

我们首先（第4.1节）编写一个程序，用于在布尔流中查找某个布尔值的无限次重复。然后，我们将程序提取为扩展了分界控制的 OCaml[2] 。接着，我们证明（第4.2节）该程序确实计算出了无限鸽巢原理。

4.1 计算

在这里，我们展示经典余递归相对于余迭代的表达能力：它捕获尾部分支，并为其提供一等延续，而这在纯函数式语言中原本无法做到。特别是，访问指向尾部调用者的延续，比仅仅停止余迭代提供了更大的灵活性。

我们将编写一个函数 `infinite_布尔类型`：给定布尔流 `bs`，它返回一个指向 `bs` 的索引流，使得所有索引都指向同一个布尔值。例如，给定以下流：

```
定义 恒定流_真 := coiter (fun x=>x)(fun x=>X) 真.定义 almost_恒定流_真 := {| 头部 :=假; 尾部 :

                                                                    尾部 := 恒定流_真
                                                                |}
                                |}.
```

我们期望 `infinite_布尔类型` `恒定流_真` 被绑定到自然数流（即 `countUp0`），而 `infinite_布尔类型` `almost_恒定流_真` 被绑定到流 `countUp 2`。然而，如果我们有：

```
CoFixpoint alternate: = coiter (函数 x=>x) (函数 x=>negb x)true
```

那么`无限_布尔类型` alternate 可能会返回偶数流，也可能会返回奇数流！这说明答案取决于请求，这并不奇怪，因为我们无法观察整个流。例如，考虑一个由 100 个`false`组成、后接 100 万个`true`、然后再接无限多个`false`的流。如果我们要查找出现 10 次的元素，可能会说`false`。但如果接着要查找出现 1000 次的元素，我们可能会想说`true`，因为在看到 101st `false` 之前，会找到更多这样的元素。然而，如果我们要查找出现 10 亿次的元素，就只能说`false`；`bs` 中`true` 的出现次数实在不足以满足请求。

借助一等控制，我们可以有效地提供一个流，它看起来能给出无限次出现的布尔值的所有索引。然而，归根结底，我们只需实现它的一个近似，因为每个终止程序在结束前只能检查流中的有限个元素。经典余递归中的一等控制让我们能够在程序运行时自动推断这个有限数量，无需程序员明确知晓或介入。

[2]在代码仓库中，我们还提供了 Scheme 代码

```
定义 无限_布尔类型 (bs : 布尔流) : IPP_流： =
  令 b0： = 头部 bs 在                                    (* （头部 bs）无限次出现 *)
  callcc（函数start： 延续 IPP_流  =>
    coiter
      {ns : {n : 自然数 & 布尔流} & 延续 ipp_布尔流}      (* 内部状态的类型 *)
      （函数 '(深度； _； _)  => 深度)                    (* 头部 *)
      （函数 '(深度； 剩余部分； 切换) =>                 (* 尾部 *)
          如果布尔类型_dec (head rest) b0
          那么 (S 深度； 尾部 rest； 切换)
          否则callcc （函数 restart  => throw switch
   (* (head bs) 不会无限次出现 *)
        (corecC
          {n : 自然数 & 布尔流}                          (* 内部状态的类型 *)
          （函数 s => projT1 s）                         (* 头部 *)
          （fun '(深度； 剩余部分) ret  =>               (* 尾部 *)
              如果布尔类型_dec (头部 剩余部分) b0
              则 throw restart (S 深度; 尾部 剩余部分; ret)
              否则 (S 深度; 尾部 剩余部分)
          )
          (S 深度; 尾部 剩余部分)))                      (* 余递归器的状态          *)
      )
      (0; 尾部 bs; start)).                              (* 余迭代器的状态 *)
```

图 1：在布尔流中搜索某个布尔值的无限次重复。

callcc 运算符通过将观察者捕获到一个延续中来创建检查点；我们可以多次调用这个延续，回溯到调用callcc 时的时刻，从而向同一个观察者提供多个不同的答案。

在这个应用中，我们首先猜测，流的头部元素可能与无限次出现的那个布尔值相同。然后，我们用callcc创建一个检查点。只要我们不断找到该头部布尔值的更多重复项，这个猜测看起来就是正确的，我们就可以继续提供该布尔值重复出现位置的更多索引。然而，如果我们在流中找到另一个布尔值，那么我们的猜测可能就是错的。在这种情况下，我们可以回溯到起点，将答案改为另一个布尔值，并继续搜索流中余下的部分。如果我们再次找到第一个布尔值的重复项，就可以再次回溯到之前停下的位置，而不是从头开始。

如图 1 所示，该算法可以用 Rocq 实现。输入中的布尔流只会被观察，因此我们实现时不会延迟其尾部。相比之下，为了防止callcc在提取后被急切求值，必须延迟输出流的尾部。为此，我们采用插曲部分讨论过的修改版本，并将其称为ipp_流。组合子coiter 和corecC 也作了相应修改[3]。顶层infinite_bool 函数接受任意布尔流bs。 infinite_bool 的任务是返回一个自然数索引流，其中的索引指向bs中同一个布尔值。首先， infinite_bool 调用callcc ， 保存首次调用时的检查点，以便在需要时能够从头开始重新开始遍历整个流。接着， infinite_bool 猜测索引 0 处的布尔值（记作b0）会重复出现足够多次，并返回通过余迭代创建的索引流，其状态为：

– 当前深入原始流的深度 （从 0 开始）， 以及– 要搜索的剩余部分 bs 流，并且

[3]完整的 Rocq 程序可在线获取，网址为 IPP program.v。

```
令无限的_布尔类型 bs  =令 b0 = 为 bs 的头部，然后调用 callcc（fun start ->coiter（fun pat ->
```

图 2：使用分界控制提取为 OCaml 的程序。

– 在初始猜测错误的情况下要切换到的延续（从名为 `start` 的顶层延续开始）

如果下一个元素等于输入流的头部，那么我们只需使用更新后的状态继续`coiter`：

– 深度加一，– 流的剩余部分的尾部，
以及– 相同的延续

否则，我们找到了一个不同于头部 bs的布尔值，因此必须切换搜索模式，以寻找更多neg (头部 bs)的出现。不过在切换之前，我们会记住自己在生成该流时所处的位置，以便稍后恢复。然后，我们调用`switch` 延续，并传入一个索引流，其中所有索引都指向neg (头部 bs)。有趣的是，这个内部流是经典余递归器的一个实例，其状态为：

– 原始布尔流中的当前深度 （从 1 开始），以及– 我们正在搜索的流的
尾部

与前一个流一样，获取头部 会返回当前深度。计算尾部时，如果下一个布尔值与`neg b0`相同，则会更新余递归的状态。否则，控制必须切换 回外层循环，重新启动它。为此，它会传递更新后的状态，其中包含新的延续；该延续会返回给调用尾部的调用者。

使用分界控制将程序提取为 OCaml

为了运行 Rocq 程序，我们将代码提取为 OCaml 代码，该语言提供了带分隔符的控制[4]，，并将 callcc 定义为 shift p (`fun` k -> k (c (`fun` x -> abort p (k x)))，其中 p 表示用于界定的提示。提取出的代码 [5] 见图 2。我们使用名为test的流来测试程序。该流由append将一些不规则变化添加到恒定流always true上得到：true false

[4]使用 delimcc 库[5]完整提取出的 OCaml 程序可在线获取，网址为 IPP program.ml。我们也提供 Scheme 代码，网址为 IPP program.scm。

`false true false true true true ...` 通过使用类型为 `stream A → nat →list A`的函数 `take`，我们可以观察流的有限前缀。例如，如果我们只要求流 `test`中重复出现的前三次，就应该能观察到`false`首次出现的三次：

```
push_提示 p (fun () -> 取前缀 (无限_布尔类型 test) 3) - : 自然数列表
```

然而，如果我们要求在同一条流中重复五次，那么其中可找到的`false`就远远不够了。因此，无限_布尔类型 会指出五个不同的索引，使其值为`true`：

```
push_提示 p (fun () -> 取前缀 (无限_布尔类型 test) 5) - : 自然数列表
```

现在，让我们对无限_布尔类型所返回的同一条流运行上面的两个测试：

```
push_提示 p (fun () -> let ix = (无限_布尔类型 test) in [取前缀 ix 3; 取前缀 ix 5]);; - : (自然数列表) 列表
```

每次我们都会得到一致近似[6] 。请注意，尽管答案可能取决于观察结果，但对无限_布尔类型 的单次调用，在其整个生命周期内，无论结果被检查多少次，都会恒定流地生成一致的结果。尽管第一次测试（只请求三个索引）最初可能会得到上面显示的近似结果 `[1; 2; 4]`，但为了与要求更高的第二次测试（请求五个索引）保持一致，它会自动更新为 `[0; 3; 5]` 。

4.2 证明

我们成功编写了无限_布尔类型 函数并运行了提取出的代码，但我们无法保证它会返回正确的索引序列；它也可能返回由`zeroes` 组成的流，因为这同样满足每个索引都指向相同布尔值这一性质。我们的目标是证明，前面的程序构成了无限鸽巢原理的计算内容。该原理断言，在任何由真和假组成的布尔值序列中，都存在一个常值子序列。

定理 3（无限鸽巢原理）。对于每个无限布尔流 bs，都存在一个递增的索引流 s，使得每个索引在 bs 中都指向同一个真值：

$$\forall bs.\exists s.\forall n.(s(n) < s(n+1) \wedge bs(s(n+1)) = bs(s(n))) \ .$$

我们的目标是将这个不变量嵌入流的定义中。同时，我们希望确保从证明中提取的代码与原始程序一致；换句话说，程序就是证明的擦除。在类型论中，通常将命题视为子单元素类型，即至多只有一个元素的类型所构成的子集。在此语境下，类型 A 的命题截断`trunc A`是一种类型论联结词，它能精确地阻止从 A 中提取计算内容。也就是说，`trunc A` 是一个至多只有一个元素的类型，用来表达 A 有元素，但无法得知具体是哪个元素。因此，我们对流的定义变为：

```
归纳 截断 A: 命题 := 截断: A -> 截断 A。定义 ok(bs:截断 (流
```

[6]运行上述测试需要定义一个新的提示，因为返回类型现在是 (nat list) list，而不是 nat list。

```
令 infinite_布尔类型 bs  =令 b0 = 头部 bs in callcc (fun start ->coiter (fun _ pat -> 令 ExistT
```

图 3：从证明中提取的 OCaml 程序

请注意，该流以输入的（截断）布尔流作为参数，名为bs，并以先前的索引作为另一个参数，名为 n。起初，索引为 None，这就是我们使用（截断）选项类型的原因。深度 指的是程序中布尔流的索引，而剩余部分 指的是流的尾部。假设 形式化了这个不变量；除了查看盒子内部（即ok定义中的 ...）之外，它还保证所有索引都指向同一个布尔值（indexdepth bs’ = index n’ bs’），并且当前索引（即深度）严格大于先前的索引（n’ < 深度）。因此，我们的目标变成证明：

```
无限_布尔类型 bs : ipp_流 (截断 bs) (截断 None)
```

Rocq 证明可在线获取，文件名为 IPP proof.v。我们在图 3 中展示从该证明提取出的 OCaml 程序，以强调我们最初的程序确实构成了无限鸽巢原理证明的计算内容[7]。

5 使用余迭代的无限鸽巢原理

与 Escard´o 和 Oliva 的间接证明比较。Escard´o 和 Oliva [9, 10] 在 Agda[8] 中给出了无限鸽巢原理双重否定表述的直觉主义证明：

$$\forall f^{\mathbb{N}\to bool}.\neg\neg\exists b.\exists g^{\mathbb{N}\to\mathbb{N}}.\forall n.(g(n) < g(n+1) \wedge f(g(n)) = b)$$

让我们固定一个值 b_0。在 Escard´o 和 Oliva 的证明中，它是一个固定值（例如 false），但我们也可以选择，例如 b_0 :=bs(0)，正如我们在第4节中所做的那样。从直接风格来看，该程序通过调用排中律进行推理：要么存在一个以与 b_0，相同的值开头的无限常值子流，要么在某个位置之后，该流变为值相反的常值流

[7]完整的 OCaml 代码可在线获取，文件名为 IPP_proof.ml；Scheme 代码则可在 IPP proof.scm 获取。在 Rocq 提取 coiter 和 corecC 时，类型为截断自然数的证明无关参数仍未被擦除。我们不知道原因，但这解释了为何会多出 _s.[8]他们更精确地证明了 $\forall f^{\mathbb{N}\to\text{布尔值}}.\neg_R\neg_R\exists b.\exists g^{\mathbb{N}\to\mathbb{N}}.\forall n.(g(n) < g(n+1) \wedge \neg_R\neg_R f(g(n)) = b)$，其中 $\neg_R A := A \to R$；对于 $R := \bot$，由于布尔值上的等式可判定，这一结论可以简化。

b_0。令 A 为命题 $\exists n.\forall n'.(n' \geq n \rightarrow bs(n') \neq b_0)$。从计算角度看，排中律设置一个回溯延续，例如 start，假设 $\neg A$；如果最终给出了 A 的证明，则将其返回给该回溯延续。无论 $\neg A$ 还是 A 成立，都必须找到一个子流。A 的情形很直接，取 $g(k) = n + k$。$\neg A$ 的情形依赖于可数选择公理[9]，以提取一个函数 f，使得 $\forall n.(f(n) \geq n \wedge bs(f(n)) = b_0)$。随后可以通过迭代 f 来构造适当的 g。

比较这两个证明可以发现，使用流而非从 $\mathbb{N}$ 到 $\mathbb{N}$的函数来表示子序列会产生影响。事实上，流支持参数，因此下一个元素可以依赖于先前的元素；而函数对某个输入的输出只取决于输入本身[10]。例如，要证明函数 f 是递增的——正如本节开头的无限鸽笼原理陈述所要求的那样——需要一个外部性质来联系 f (n) 和 f (n + 1)，然后再证明：

$$\exists f.\forall n.f(n) < f(n+1)$$

相较之下，只需用最后访问的值作为参数（开头使用 `None`），就能证明一个流是递增的；递增条件可以直接封装在流中：

```
余归纳 封装的_流 (prev:option 自然数) : 类型 :={ depth : 自然
```

然后证明 封装的_流 None。在第4节使用余递归的证明中，利用了向流传递参数的能力：余递归所给出的延续会被一路传递下去。在 Escard´o 和 Oliva 的证明中，每当发现一个不同于 b_0 的值时，就会从头重建一个新的子序列。Escard´o 和 Oliva 的证明并不对称：如果等于 b_0,的值有无限多个，就返回由所有这些值构成的子序列；如果只有有限多个，则返回从 b_0, 最后一次出现后立即开始、由连续的 b $\neg b_0$, 构成的尾随子序列。特别是，所有后面跟着一个 b_0 的 $\neg b_0$ 最终都会被跳过（前提是我们请求足够多的元素来观察到这一点）。下一节将介绍 Escard´o 和 Oliva 的一个变体证明，该证明使用流且只采用余迭代。

5.1 使用余迭代的直接证明

在图 4 中，我们给出一个使用余迭代和控制的程序，它从由 true 或 false 组成的流 bs 中提取一个常值子流。与图 1 中给出的程序一样，我们使用 IPP_流 作为输出。该程序受 Escard´o 与 Oliva 的 Agda 证明 [10], 启发，其中使用布尔类型而非 $\{0,1\}$，并用流代替从 $\mathbb{N}$ 到布尔类型的函数。现在我们可以观察这两个程序之间的差异。如果我们令 `test` 为该流：

false true false true true true true...

如果我们观察前四个索引，我们得到这些索引

1 3 4 5

使用余递归编写程序，但我们得到

3 4 5 6

[9]其双重否定形式在 Escard´o 与 Oliva 的证明[10]中具有使用一般递归的计算解释，除非朝高度依赖函数的方向发展 [18], 流在函数语言中的对应物

```
定义 infinite_布尔类型_coiter (bs : 流 布尔类型) : ipp_流 :=let b0 := 流的头部
```

图 4：通过余迭代实现的无限鸽巢原理。

通过余迭代实现。第二个真值（索引为 1）被忽略。证明及其对应程序可分别在线获取：IPP Escardo proof.v 和 IPP Escardo program.v。提取出的 OCaml 代码可在 IPP Escardo proof.ml 获取，提取出的 Scheme 代码可在 IPP Escardo proof.scm 获取。

6 通过余迭代实现可数选择

一般的选择公理表述如下：

$$\mathsf{AC} \ : \ \forall x^A.\exists y^B.R(x,y) \to \exists f^{A\to B}.\forall x.R(x,f(x))$$

构造性选择。上述内容仍需作一些澄清：其中的 $\exists$ 和 $\to$ 应如何解释？作为一种逻辑与编程语言，依赖类型论通常足够精细，能够表达这些差异。例如，在类型论中，$\exists$ 可以解释为一种强构造性存在量化，即一个 Σ-类型，并配有投影 $\pi_1 : \Sigma x^A B(x) \to A$ 和 $\pi_2 : \forall z^{\Sigma x^A B(x)} B(\pi_1 z)$；在这种情况下，选择公理（称为 Martin-L¨of 选择公理，或内涵选择公理）可由 $\forall$ 对 Σ 的分配律直接推导得出：

$$\begin{array}{rcl} \mathsf{AC}^{\mathsf{int}} & : & \forall x^A.\Sigma y^B.R(x,y) \to \Sigma f^{A\to B}.\forall x.R(x,f(x)) \\ & := & \lambda H.(\lambda x.\pi_1(H(x)),\lambda x.\pi_2(H(x))) \end{array}$$

经典选择。得益于命题截断，$\exists$在类型论中可以解释为 Σ的命题截断，即：

$$\exists x^A.B(x) := ||\Sigma x^A.B(x)||$$

在这种情况下，$\exists$取自直觉主义逻辑中通常的命题含义，而选择公理则不再可导出。不过，借助 Martin-L¨of 的选择公理，我们可以证明命题 $\forall x^A.\exists y^B.R(x,y) \to \exists f^{A\to B}.\forall x.R(x,f(x))$ 等价于：

$$||\ ||\text{–Shift} \ : \ \forall x^A.||B(x)|| \to ||\forall x^A.B(x)||$$

这让人想起熟悉的弱经典双重否定移位原理，通常记作 DNS，但与之相对，我们在此将其写作 $\mathsf{DNS}^{\mathsf{Prop}}$：

$$\mathsf{DNS}_{\mathsf{Prop}} \ : \ \forall x^{A}.\neg\neg B(x) \to \neg\neg\forall x^{A}.B(x)$$

当 B 是一个命题时。

在经典逻辑成立的情况下，也就是说，在命题上满足双重否定消去或排中律时，命题截断等价于双重否定：

$$\neg\neg A \leftrightarrow ||A||$$

因此，在经典逻辑成立的情况下，选择公理等价于：

$$\mathsf{DNS}_{\mathsf{Type}} \ : \ \forall x^{A}.\neg\neg B(x) \to \neg\neg\forall x^{A}.B(x)$$

这与通常的 DNS 不同之处在于，它允许考虑任意类型 B(x)，而不局限于命题 .

经典可数选择。在 A 为可数集的情况下，Spector [29]以递归程序的形式给出了 $\mathsf{DNS}^{\mathsf{Type}}$的计算内容，该程序逐步生成 ∀n.B(n) 中某个元素的越来越精细的近似。从这一点可以直接得到一个用于可数选择的双重否定形式的递归程序：

$$\mathsf{AC}^{\neg\neg}_{\mathsf{nat}} \ : \ \forall n.\neg\neg\Sigma x^{A}.R(n,x) \to \neg\neg\Sigma f^{\mathsf{nat}\to A}.\forall n.R(n,f(n))$$

Spector 当时是在 G¨odel 的“Dialectica”函数解释的背景下进行推理的。后来，Berardi、Beze m 和 Coquand [5]在 Kreisel 的修正实现背景下给出了该解释的一个变体。Berger、Escard´o 和 Oliva [7, 8] 等人也研究了类似原则的计算实现。这些解释在如何解释 $\mathsf{DNS}^{\mathsf{Type}}$ 方面有所不同（例如，参见 [9] 中的 Agda 文件 K-Shift，其中实现了这些解释）。但是，在所有情况下，它们都依赖于一般递归，其终止性由依赖选择公理或与之在经典逻辑下等价的公理（例如 Bar 归纳）加以证明。

在 [17], 第三位作者中，他描述了一种结合余归纳、 [22], 控制和 Σ-类型的形式语言，即 dPA_{ω}（避免了 [16] 中所述的不一致性），该语言为 $\mathsf{AC}^{\neg\neg}_{\mathsf{nat}}$ 提供了一种基于控制的计算解释，而这一解释仅由余归纳[11]在内部予以证明。

事实上，在 $\mathsf{DNS}^{\mathsf{Type}}$的表述中， ∀n.¬¬B(n) 可以解释为如下的流：

$$\{\mathtt{head} : \neg\neg B(0); \mathtt{tail} : \{\mathtt{head} : \neg\neg B(1); ...\}\}$$

而 ¬¬∀n.B(n) 则可解释为如下的流：

$$\neg\neg\{\mathtt{head} : B(0); \mathtt{tail} : \{\mathtt{head} : B(1); ...\}\}$$

s因此，$\mathsf{DNS}^{\mathsf{Type}}$ 可以改述为双重否定在流上的分配律。

转而采用直接风格，也就是使用 callcc， [17] 中探讨的余归纳与控制相结合的方法，可用于给出 $||\ ||_{\vdash}$移位（更一般地说，对于强单子 T，可给出 T –移位）的余迭代证明，最终不仅能证明 $\mathsf{AC}^{\neg\neg}_{\mathsf{nat}}$，还能证明 $\mathsf{AC}^{T}_{\mathsf{nat}}$，其中 T 可以是任意强单子；特别地，还能证明 $\mathsf{AC}^{||\ ||}$ 自然数。Rocq 证明可在线查阅，文件为 ac coinductive.v。对该证明的详细分析留待后续工作。

[11]另请参阅 Miquey [27]，了解二阶算术中余归纳、控制与 Σ-类型组合的交叉论证。

7 无限鸽巢原理的不同表述 e

研究表明，无限鸽巢原理可以用许多不同的方式来表述和证明。使用流时，一个可变之处是在流的定义中打包其规格，如 打包的_流 的定义那样，这样便无需调用可数选择公理。另一个可变之处是在存在量词前加入双重否定，以避免使用经典推理，Escard´o 和 Oliva 采用的正是这种做法。此外，在 Rocq 中进行程序提取时（如我们所做的那样），可以使用命题截断，将陈述中选定的部分标记为非计算性的。还有一种可变之处：无论子序列表示为函数还是流，其恒定性都可以像 Escard´o 和 Oliva 的形式化那样，以全局方式表达为：

$$\exists b.\exists g.\forall n.(g(n) < g(n+1) \wedge f(g(n)) = b)$$

或者在局部，像我们的证明中那样，采用以下形式：

$$\exists g.\forall n.(g(n) < g(n+1) \wedge f(g(n)) = f(g(n+1)))$$

谈到对称证明与非对称证明之间的区别，更强的要求是子序列没有空洞（无论 g 是函数还是流）：

$$\exists \text{g.}\forall \text{n.(g}\ \ (n) < g(n+1) \wedge f(g(n)) = b \wedge \forall n'.((n' < g(0) \vee g(n) < n' < g(n+1)) \to f(n') \neq b))$$

而子序列从某个时刻起为常数这一更强的陈述则是 ：

$$\exists g.\forall n.(g(n)+1 = g(n+1) \wedge f(g(n)) = b)$$

事实上，值 b 的无限性也可以不借助子序列，用纯一阶逻辑来表达，如下：

$$\forall n.\exists n'.(n' \geq n \wedge f(n') = b)$$

特别是，我们似乎有理由认为，“对称”的陈述

$$\exists b.\forall n.\exists n'.(n' \geq n \wedge f(n') = b \wedge \forall n''.(n \leq n'' < n' \to f(n'') \neq b))$$

无需余递归即可证明，因此最终有可能利用可数选择，仅通过余迭代给出定理 3 的对称证明。

8 结论

我们用余递归和经典推理给出了无限鸽巢原理的直接风格证明，并开始将其与 Escard´o 和 Oliva 通过双重否定翻译给出的间接证明进行比较。我们还提供了一个程序，仅使用控制和余迭代来解释命题截断对流的分配律，从而解释可数选择公理。控制、余迭代和余递归能够组合成一致逻辑系统的一般条件尚待澄清，这将留待未来研究。还应对无限鸽巢原理的不同表述及其证明进行更严谨的研究。

参考文献

[1] Zena M. Ariola & Hugo Herbelin (2003)：极小经典逻辑与控制算子。载于：自动机、语言与程序设计：第30届国际学术研讨会，ICALP 2003，施普林格柏林海德堡，柏林，海德堡，第871–885页，doi:10.1007/3-540-45061-0_68.[2] Franco Barbanera & Stefano Berardi (1993)：通过类控制归约从经典逻辑中提取构造性内容。载于 Marc Bezem & Jan Friso Groote 编：类型化λ演算及其应用，类型化λ演算及其应用国际会议，TLCA '93，荷兰乌得勒支，1993年3月16–18日，论文集，《计算机科学讲义》664，施普林格，第45–59页，doi:10.1007/BFB0037097.[3] Franco Barbanera & Stefano Berardi (1995)：经典逻辑的强规范化结果。《纯数学与应用逻辑年刊》76(2)，第99–116页，doi:10.1016/0168-0072(95)00004-Z.[4] Franco Barbanera & Stefano Berardi (1996)：用于经典程序提取的对称λ演算。《信息与计算》125(2)，第103–117页，doi:10.1006/INCO.1996.0025.[5] Stefano Berardi, Marc Bezem & Thierry Coquand (1999)：选择公理的计算内容。载于：类型化λ演算及其应用（TLCA 1999），《计算机科学讲义》1581，施普林格，第56–72页，doi:10.1007/3-540-48959-2_5.[6] Stefano Berardi, Mario Coppo, Ferruccio Damiani & Paola Giannini (2000)：基于类型的函数式程序无用代码消除。载于 Walid Taha 编：《程序生成的语义、应用与实现》，SAIG 2000 国际研讨会，蒙特利尔，加拿大，2000年9月20日，论文集，《计算机科学讲义》1924，施普林格，第172–189页，doi:10.1007/3-540-45350-4_13.[7] Ulrich Berger & Paulo Oliva (2006)：修正的bar递归。《数学结构与计算机科学》16(2)，第163–183页，doi:10.1017/S0960129506005093.[8] Mart´ín H¨otzel Escard´o & Paulo Oliva (2010)：选择函数、bar递归与逆向归纳。《数学结构与计算机科学》20(2)，第127–168页，doi:10.1017/S0960129509990351.[9]Mart´ín Escard´o (2011)：在 Agda 中运行经典鸽巢原理。可通过链接获取：https://martinescardo.github.io/pigeon.[10] Mart´ín Escard´o & Paulo Oliva (2011)：从证明得到程序 III：通过选择函数的乘积实现经典可数选择。在程序语义的数学基础（MFPS 2011）上的报告。[11] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2018)：如何设计程序：程序设计与计算导论。麻省理工学院出版社，doi:10.5555/369273.[12]Jeremy Gibbons (2021)：函数式珍珠：如何设计协程序。《函数式编程期刊》，第e15页，doi:10.1017/S0956796821000113.[13] Jeremy Gibbons & Graham Hutton (2005)：余递归程序的证明方法。《Fundamenta Informaticae》66，第353–366页，doi:10.5555/2370162.2370165.[14]Jeremy Gibbons & Geraint Jones (1998)：未获充分重视的展开。载于：第三届 ACM SIGPLAN 函数式编程国际会议论文集，ACM出版社，第273–279页，doi:10.1145/289423.289455.[15] Timothy G. Griffin (1990)：公式即类型的控制观念。载于：第17届 ACM 程序设计语言原理研讨会会议记录，POPL '90，美国加利福尼亚州旧金山，1990年1月17–19日，计算机协会，第47–57页，doi:10.1145/96709.96714.[16] Hugo Herbelin (2005)：计算经典逻辑存在时 Σ 类型的退化。载于 Pawel Urzyczyn 编：第七届国际会议，TLCA '05，日本奈良，2005 年 4 月，论文集，《计算机科学讲义》3461，施普林格，第209–220页，doi:10.1007/11417170_16.[17] Hugo Herbelin (2012)：与经典逻辑相容的依赖选择构造性证明。载于：第27届年度 ACM/IEEE 逻辑与计算机科学研讨会论文集，LICS 2012，2012年6月25–28日，克罗地亚杜布罗夫尼克，IEEE计算机学会，第365–374页，doi:10.1109/LICS.2012.47。

[18] Jason J. Hickey (1996)：使用高度依赖类型的类型论形式对象。载于：FOOL 3：第三届面向对象语言基础国际研讨会。可访问 https://api. semanticscholar.org/CorpusID:14983194。[19] Ralf Hinze、Nicolas Wu 和 Jeremy Gibbons (2013)：统一结构化递归方案。载于：第 18 届 ACM SIGPLAN 函数式编程国际会议论文集，ICFP '13，计算机协会，纽约，美国纽约州，第 209–220 页，doi:10.1145/2500365.2500578。[20] Graham Hutton (1998)：程序语义中的折叠与展开。载于：第三届 ACM SIGPLAN 国际函数式编程会议论文集，ICFP '98，计算机协会，纽约，美国纽约州，第 280–288 页，doi:10.1145/289423.289457。[21] Bart Jacobs 和 Jan Rutten (1997)：（余）代数与（余）归纳教程。EATCS 通报 62，第 62–222 页。[22] Dexter Kozen 和 Alexandra Silva (2017)：实用余归纳。《计算机科学数学结构》27(7)，第 1132–1152 页，doi:10.1017/S0960129515000493。[23] Paul Blain Levy (2004)：调用即推值：函数式／命令式综合。计算中的语义结构 2，施普林格，doi:10.1007/978-94-007-0954-6。[24] G. Malcolm (1990)：数据结构与程序变换。《计算机程序科学》14(2)，第 255–279 页，doi:10.1016/0167-6423(90)90023-7。[25] Lambert Meertens (1992)：参数递归。《形式化方面与计算》4，doi:10.1007/BF01211391。[26] Erik Meijer、Maarten Fokkinga 和 Ross Paterson (1991)：使用香蕉、透镜、信封和带刺铁丝网进行函数式编程。载于：第 5 届 ACM 函数式编程语言与计算机体系结构会议论文集，施普林格出版社，柏林、海德堡，第 124–144 页，doi:10.5555/645420. 652535。[27] ´Etienne Miquey (2017)：经典可实现性与副作用。博士论文，索邦大学巴黎城´校区 - 巴黎第七大学´；乌拉圭共和国大学 - 蒙得维的亚，乌拉圭。可访问 https://inria.hal.science/tel-01653733。[28] Jan Rutten (2019)：余代数方法：余归纳练习。CWI，阿姆斯特丹，荷兰。[29] Clifford Spector (1962)：可证明递归的分析泛函：通过有限类型泛函解释分析的一致性证明。载于：递归函数论：纯数学专题讨论会论文集，第 5 卷，美国数学会，第 1–27 页，doi:10.2307/2271288。[30] Varmo Vene 和 Tarmo Uustalu (1998)：使用阿波形态进行函数式编程（余递归）。载于：《爱沙尼亚科学院学报：物理学、数学》，第 147–161 页，doi:10.3176/phys. math.1998.3.01。