《函数式编程杂志》，?? 页，2022年。© 剑桥大学出版社2022年 doi:10.1017/xxxxx

arXiv:2303.09358v1 [cs.PL] 16 Mar 2023

函 数 式 珍 珠

免费的完全抽象

MARCO PAVIOTTI
肯特大学，（电子邮件：m.paviotti@kent.ac.uk）
NICOLAS WU
伦敦帝国理工学院，（电子邮件：n.wu@imperial.ac.uk）

摘要

折叠和展开等结构化递归方案，已广泛用于构造函数式程序和程序语义。在此背景下，人们通常将指称语义实现为对归纳数据类型的折叠，以确保终止性和组合性。此外，操作模型可以通过展开给出，而显然并非所有操作模型都能以有意义的方式与给定的指称语义相一致。

为确保这些语义协调一致，考虑完全抽象这一关联指称模型与操作模型的性质十分重要。本文展示了如何构造组合语义，使其天然具备完全抽象性。我们通过使用分配律来实现这一点，并由此生成操作模型和指称模型。分配律确保语义在类型层面上完全抽象，从而免去程序员证明这些性质的负担。

1 引言

长期以来，人们一直使用结构化递归方案来确保函数定义良好（Meijer 等人，1991年）。特别是，折叠一直用于确保终止性，而展开则用于确保递归定义函数的生产性。关于递归算子，一个或许鲜为人知的事实是，当它们用于编程语言和语义的语境中时，还能提供额外的语义性质。Hutton（1998年）普及了这样一种观点：折叠可用于定义这类语言的指称语义，而展开可用于定义其操作语义。

既然折叠和展开提供了不同的语义，一个自然而然的问题是二者之间有何关联。完全抽象的概念回答了这个问题：在任何语境中，操作行为都相同的语法形式不同的程序，在指称语义上应被视为完全相同的实体。

为了解释这一点，我们将 Hutton剃刀的语法定义为一个归纳集（左），并在 Haskell 中将其实现为一种数据类型（右）：

$$L := \mathbb{N} \mid L + L \qquad\qquad \textbf{data}\ L = \mathit{Val}\ \mathit{Nat} \mid \mathit{Add}\ L\ L$$

其中 Nat 是自然数类型。可以通过将语法的结构折叠到语义域的结构中来定义这种语言的语义，该语义域由基本情形 n ::Nat 和二元运算 Add :: Nat $\to$ Nat 给出：

$$\begin{aligned}
&[\![\cdot]\!] :: L \to \mathit{Nat} \\
&[\![\mathit{Val}\ n]\!] \quad\ = n \\
&[\![\mathit{Add}\ t_1\ t_2]\!] = [\![t_1]\!] + [\![t_2]\!]
\end{aligned}$$

这种语义通常称为指称语义，其中项的含义由其子项来定义。从这个意义上说，指称语义是将语法解释到给定论域中的模块化解释。恰好，通用折叠的运作方式正是如此：它接受一个处理基本情形的函数和另一个处理归纳情形的函数，二者合称为代数，并利用它们将数据结构归约为最终的值。对语法进行折叠，恰好对应于指称语义的结构。

不过，我们也可以用许多其他方式来定义语义，这些方式同样对应于折叠。例如，将每个程序都映射为 ()，即 $[\![\_]\!] = ()$；或者将每个程序都映射为其自身，即 $[\![\cdot]\!] = \mathrm{id}$。一个或许更极端的例子是，将语法构造子加法解释为语义上的乘法。

鉴于语义有多种不同的定义方式，一个值得思考的问题是如何提供一种在它们之间进行评判的方法。将数字相加得到 Nat 似乎合理，而另外三种则有些极端且不尽如人意。

然而，判断一个语义模型好坏的唯一方法，就是精确定义程序应该“做什么”，例如为该语言给出一个操作模型。在这里，我们用规则定义一个归约关系，记作 t $\to$t′，适用于程序 t 和 t′；每条规则中，若横线上方的归约成立，则横线下方的归约也成立：

$$\frac{}{\mathit{Add}\ (\mathit{Val}\ m)\ (\mathit{Val}\ n) \rightsquigarrow \mathit{Val}\ (m+n)} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathit{Add}\ t_1\ t_2 \rightsquigarrow \mathit{Add}\ t_1'\ t_2} \qquad \frac{t_2 \rightsquigarrow t_2'}{\mathit{Add}\ t_1\ t_2 \rightsquigarrow \mathit{Add}\ t_1\ t_2'}$$

定义好操作模型后，我们就可以说明对指称模型的要求。尤其是，一个好的指称模型必须是可靠的，这意味着它必须不受单步归约的影响；此外，它还必须至少在计算上是充分的，也就是说，如果两个程序在模型中表示同一个对象，那么它们在操作上应当不可区分。进一步而言，如果模型恰好将本应不可区分的程序等同起来，那么这种语义就称为完全抽象。因此，好的指称语义应当以某种方式与操作语义相关联。

问题在于，现有用于给出语义解释的递归方案（如折叠）并未对用户可以提供的代数类型施加任何限制。换言之，折叠算子既不保留也不反映操作语义所提供的附加信息。

在余代数系统中，我们可以通过使用余代数，为当前的语法树结构关联一种操作语义。余代数是一个函数，它以程序作为输入，并输出目标程序以及关于所产生行为的一些信息。例如，上文定义的语言的操作语义也可以等价地

定义为函数 opsem :: L $\rightarrow$ [L][1]，其中值映射到空程序集合，形式为 Add (Val n) (Val m) 的程序映射到 Val(n + m)，而形式为 Add t_1 t_2 的程序则映射到形式为 Add t_1' t_2 和 Add t_1 t_2' 的程序，只要 t_1 归约为 t_1'；对于 t_2，情况也类似。

现在我们已经掌握了所需的操作信息，接下来要确保在解释语法时以某种方式体现这些信息。正如前文所述，折叠算子不适合这项工作，而展开算子则适合。具体而言，展开算子是一个函数：给定一个程序，它会生成在该程序上“展开”或“运行”操作模型所产生的“轨迹”。换言之，当且仅当两个程序在操作层面行为相同时，它们的展开结果也相等，此时展开算子就是从语法到轨迹域的唯一完全抽象翻译。

本文将展示如何利用 分配律 来确保从语法到语义域的解释函数既是折叠（即具有组合性），也是展开（即完全抽象）。这一方法背后的理论早已为人所知（Rutten 和 Turi，1993；Turi 和 Plotkin，1997年；Jacobs 和 Rutten，2012；Rutten，2000），许多人也曾运用这一理论来构建函数式编程（Hinze 和 James，2011；Jaskelioff 等，2011）。然而，这些研究都没有明确关注递归算子在语义性质方面的用途。

本文通过重新阐释 Hutton剃刀和 Milner的CCS 来实现这一点。尤其是后者，实质上催生了一种用于 Haskell 流式编程的（正确）领域特定语言（DSL）。我们的工作可视为对 Hutton（1998年）工作的扩展；他提出了将指称语义表达为折叠、将操作语义表达为展开的思想。然而，两者并不一定一致；就 CCS 而言，这种对应关系需要通过纸笔证明。

在我们的方法中，分配律的参数化类型结构用于引导程序员免费获得完全抽象语义。我们发现，虽然 CCS 语义自然适用于分配律的框架，但 Hutton 的算术语言则需要稍作调整。按照惯例（Danielsson 等，2006），我们在 Haskell 的全片段中实现文献中的理论。具体而言，我们使用递归类型实现数据类型，并采用适当的递归方案，以保持初始代数与终余代数的区别；最后，我们只使用谓词多态，因此本文中的 Haskell 程序可以在集合范畴中解释。

我们首先介绍折叠和展开的背景知识（第2节），随后运用它们，为流的简单算术语言（第3节）以及带有非确定性的算术语言（第4节）实现完全抽象语义。作为一个更复杂的示例，我们将展示如何为 Milner 的值传递 CCS 提供完全抽象语义（第5节）。最后，我们列出相关工作（第6节），并进行讨论（第7节）。

[1] 为简化起见，集合被建模为列表。

2 背景：折叠与展开

折叠和展开是最基本的递归模式。前者用于确保递归定义能够终止，后者则用于确保无限数据结构的定义具有生产性。当折叠用于组织语义时，它们能提供模块化的指称语义，正如 Hutton（1998年）所推广的那样。同样，展开通过构造即可提供完全抽象解释，但这一事实在函数式编程文献中鲜少得到探讨。本节概述这两种递归模式的基础知识。在第2.1 节中，我们介绍折叠；在第2.2 节中，我们介绍展开，并解释为何它们是迁移系统上的完全抽象解释。

2.1 折叠

折叠是一种递归算子，它会遍历归纳数据结构，并通过使用给定的组合算子，重新组装归纳结构中经过递归变换的各个子部分的结果。

折叠使用函子的概念来描述数据类型的结构，并通过该函子的最小不动点定义归纳数据类型。在下面的代码中， Σ 是一个变量，我们将用它把函子作为参数传递给 μ·。后者将据此计算 Σ 的最小不动点，并提供构造子 In 和析构子 in°，以证明 μΣ 与 Σ(μΣ)[1]同构。

新类型 μΣ = In {in° :: Σ (μΣ)}

折叠是一个递归函数，其类型为 μΣ to any type a，并配有一个函数 alg:: Σ a → a。折叠以函数 alg 为输入，并使用 alg 递归地将归纳定义的数据结构转换为类型 a 的元素，同时合并递归调用的结果

(|·|) :: 函子 Σ ⇒ (Σ a → a) → μΣ → a
(|alg|) = alg · fmap (|alg|) · in°

例如，回想一下我们在第1节前面介绍的语言。这个语言的签名函子Σ 可定义如下：

数据 ValAddF x = Val Nat | 加法 x x 推导 函子

其中，Val :: Nat → Σ x 和 Add :: x → x → Σ x 是该语言的构造子。 Σ 所描述的语言记作 μΣ。为简化说明并避免在代码中因 In 运算符而显得杂乱，我们也可以定义智能构造子

加法 :: μValAddF → μValAddF → μValAddF val:: ℕ → μ
ValAddF 加法 x y = In (Add x y) val = In · Val

现在可以将解释函数定义为对一个 Σ-代数进行折叠。例如，如果我们想将这种语言解释到自然数域中，就可以在 Nat 上使用以下 Σ-代数。

[1] 这里我们假设归纳数据类型和余归纳数据类型将通过关键字 μ·and 区分

desem :: ValAddF Nat →
Nat desem (Val n) = n
desem (加法 n m) = n +m

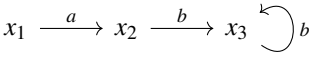
那么，对*desem*的折叠，即 *desem* ⦇ 根据定义等于解释函数 $[\![\cdot]\!]$ 我们已在第 1节中证明了这一点。

2.2 展开

折叠会析构数据，而与之对偶的展开则构造数据。具体而言，给定一个种子值和一次观测，展开会从初始种子开始，余递归地“运行”这些观测。

例如，对于状态集 X 和字母表集合 L，我们可以将 X 上的迁移系统定义为函数 c : X →L ×X，以实现迁移映射。具体而言，对于状态 $x_1 \in X$，$c(x_1)$ 会返回一对 $(l, x_2) \in L \times X$，其中 $l \in L$ 是可观测动作，而 $x_2 \in X$ 是下一状态。

现在考虑下面的迁移系统，它由三个状态 x_1,x_2 和 x_3 组成，迁移的标签来自标签集合 $L = \{a, b\}$ c_X，定义如下：$x_1 \mapsto (a, x_2)$, $x_2 \mapsto (b, x_3)$ 和 $x_3 \mapsto (c, x_3)$。



可以通过定义行为函子来实现，记作 BHV，用来表示每次迁移中观测到的信息形态；并定义观测映射 obs :: States → BHV States，将起始状态映射到下一状态，同时附带进程中产生的可观测行为的额外信息。

数据 BHV k = Nat :< k 数据
状态 = X_1 | X_2 | X_3opsem :: 状
态 → BHV 状态 opsem X
$_1 = 1$:<X_2opsem $X_2 = 2$:<X_3
opsem $X_3 = 2$:<X_3

我们可以引入一些记号，明确表示这个迁移映射实际上就是我们迁移系统的操作语义。

$$x \xrightarrow{l}_X x' \stackrel{\triangle}{\Longleftrightarrow} obs(x) = (l, x')$$

在迁移系统的语境中，展开函数会从起始状态开始执行操作语义，同时收集每次迁移中出现的可观测行为。因此，我们需要先定义行为集合的类型。

2.2.1 余归纳数据类型

由于执行操作语义可能不会终止，我们关注的是（可能）无限的数据类型。在上面的示例中，BHV 被定义为标签与目标状态的配对，我们关注的是无限的标签集合。因此，一旦确定了行为 B 的形状，我们就希望得到 B 的最大不动点，记作 νB，并按如下方式实现[1]

newtype νb = 输出° {out :: b (νb)}

现在，对于每个类型为 x 的起始元素，展开都会将其映射到 νb 中满足以下方程的唯一元素

⟦·⟧: 函子b ⇒ (x → b x) → x → νb
⟦coalg⟧ = Out° ·fmap ⟦coalg⟧ ·coalg

让我们使用上文定义的行为函子 BHV 来举例说明。它的最大不动点是数字流（无限列表）这一数据类型

类型 Stream = νBHV

现在，从一个状态开始，例如 X_2，，通过对迁移映射 opsem 无限次迭代，同时收集可观察行为，即可得到展开；对于状态 X_2，这会产生无限轨迹“22222222222222...”。

2.2.2 展开的完全抽象

很容易看出，尽管 X_2 和 X_3 是不同的状态，但它们的轨迹相同。也就是说，$[\![opsem\ X_2]\!] = [\![opsem\ X_3]\!]$。这意味着，当且仅当两个状态映射到相同的流时，它们会产生相同的轨迹。为此，再次考虑状态 X_2，并注意到 *opsem* 将 X_2 以标签 2 迁移到 X_3，那么 *opsem* X_2 必须将 X_2 映射到一个无限流，其中第一个标签是 2，其余部分则是 X_3 所映射到的流。因此，展开映射到 ν*b* 时会保留行为；所以，如果两个状态（在操作语义上）产生相同的轨迹，它们的展开也必然相同。反过来则基于这样一个事实：展开会将每个状态映射到其唯一的轨迹。换言之，不存在其他轨迹能够表示某个状态，因此，如果两个状态的轨迹相同，那么它们在操作语义上的行为也相同。关于这一现象的更正式解释，请参阅 Hinze（2008，附录，定理 1）。

2.2.3 其他行为函子

根据我们选择的函子 B，我们可以对不同类型的语义进行建模。

另一个例子是大步操作语义，它由常值函子 Big x = V 建模，其中 V 是某种语言的值集合。与之相关的 V-余代数 $\cdot\Downarrow\cdot :: \mu\Sigma \to V$ 将程序发送到它们规约后得到的 V 中的值。我们可以使用经典记号来表示大步语义，如下所示

$$t \Downarrow v \overset{\triangle}{\Longleftrightarrow} \cdot\Downarrow\cdot(t) = v$$

[1] Noti请注意，在 Haskell 中，归纳数据类型与余归纳数据类型相同。这里我们假设 μ·和 ν·不同 t.

到目前为止，我们讨论的都是确定性语言的语义。然而，当语言具有非确定性时，我们需要一种将程序映射到多个程序的方法。为此，我们可以使用有限幂集函子 $\mathscr{P}_{\text{fin}}(-)$，在函数式编程中可用列表类型实现。在这种情况下，操作语义是一个类型为 $\mathrm{x} \to [\mathrm{x}]$ 的函数，甚至可以是 $\mathrm{x} \to [(\mathrm{L}, \mathrm{x})]$，其中每次迁移都标注了集合 L 中的一个标签。

与前面的情况一样，我们可以用如下记号表示：如果状态列表中存在 x_2，且 x_1 映射到 x_2，那么 x_1 会归约为 x_2。

$$x_1 \to_X x_2 \stackrel{\triangle}{\iff} x_2 \in opsem(x_1)$$

现在我们已经介绍了折叠和展开，以及如何在迁移系统上使用展开，接下来需要将这两部分结合起来；这就是下一节的主题。

程序语义的 3 种简单递归方案

上一节中，我们展示了如何通过折叠（第 2.1 节）为一种小型编程语言给出组合式指称语义，说明了如何用余代数描述操作语义，以及展开如何构成迁移系统的完全抽象解释（第 2.2 节）。

我们的目标是使用同一个解释函数实现组合性和完全抽象。为此，我们使用语法与语义之间的分配律。本节首先通过算术语言举例说明分配律（第 3.1 节），随后将这一框架推广到一般的签名和行为函子（第 3.2 节）。

3.1 分配律

要理解这如何有助于实现我们的目标，可以考虑我们在第 2 节中定义的签名和行为函子，特别是*ValAddF* 和 *BHV*。

假设我们想为由μValAddF 描述的语言定义操作语义，特别是，我们希望语义的行为符合以下推理规则：

$$\frac{}{val\ n \overset{n}{\rightsquigarrow} val\ n} \qquad \frac{t1 \overset{n}{\rightsquigarrow} t1' \qquad t_2 \overset{m}{\rightsquigarrow} t_2'}{add\ t_1\ t_2 \overset{n+m}{\rightsquigarrow} add\ t_1'\ t_2'}$$

从数学上说，$_n \to$ 是一个归纳定义的关系族，同时它也恰好是一个函数。具体而言，它按照语言结构归纳定义：将一个值 val n 映射为 (n, val n)，并在两个子程序 t_1 和 t_2 分别映射为 $(\mathrm{n}, \mathrm{t}_1')$ 和 $(\mathrm{m}, \mathrm{t}_2')$ 时，将程序 add $\mathrm{t}_1\ \mathrm{t}_2$ 映射为 $(\mathrm{n} + \mathrm{m}, \mathrm{add}\ \mathrm{t}_1'\ \mathrm{t}_2')$。

很容易看出，上述规范对应于一个定义在μValAddF 上的观测函数，即一个类型为 μValAddF $\to$BHV (μValAddF) 的函数，它将一个程序映射为另一个程序以及一个观测，如以下递归程序所示：

```
opsemSimple :: μValAddF → BHV (μValAddF)
opsemSimple (In (Val n)) = n :< In (Val n)
```

opsemSimple（在（加法 $t_1\, t_2$））$=$let（$n :< t_1'$）$=$opsemSimple t_1

$$(m :< t_2') = opsemSimple\ t_2$$

in

$$(n + m) :< add\ t_1'\ t_2'$$

由于这是一个递归定义，其中递归调用只用于严格更小的输入，因此我们可以等价地将 opsem 表示为对 BHV (μValAddF) 上一个 ValAddF-代数的折叠，即一个映射 ValAddF (BHV (μValAddF)) →BHV (μValAddF)，如下面的程序所示：

opsemAlg :: ValAddF (BHV (μValAddF)) → BHV (μ
ValAddF) opsemAlg (Valn) $=$ n :< In (Val n) opsemAlg (
加法 (n :< t_1') (m :< t_2')) $=$ (n $+$ m) :< In (加法 t_1' t_2')

接下来我们注意到，通过观察 In 应用于类型为 ValAddF(μValAddF) 的目标程序，可以进一步分解这个代数。事实上，我们可以去掉 In，从而得到 opsemAlg 的另一种定义，我们称之为 opsemDistr

opsemDistr :: ValAddF (BHV (μValAddF)) → BHV (ValAddF (μ
ValAddF)) opsemDistr (Valn) $=$ n :< Val n opsemDistr (加法 (n :< t_1')
(m :< t_2')) $=$(n $+$ m) :< 加法 t_1' t_2'

它与 fmapIn 后组合即可得到 opsemAlg。现在，只需将 μValAddF 替换为一个全称量化的类型变量，就能轻松地将 opsemDistr 参数化，从而得到以下参数化函数：

opsemDistr :: ∀x. ValAddF (BHV x) → BHV (ValAddF x)

注意，这不会改变 opsemDistr 的实现，因为我们定义 opsemDistr 时并未依赖 μ ValAddF 的结构。

当然，上述案例也可以很好地对偶化。事实上，指称语义是类型 νB 上的一个 ValAddF- 代数，由 dsem $=$ opsemDistr $\cdot$ fmap out 定义。

3.2 一般情况

在上面的例子中，opsemDistr 是签名函子 ValAddF 与行为函子 BHV 之间的分配律。

一般而言，给定一个签名和行为函子 Σ 以及 b，分配律是一个参数化函数，按如下方式将 Σ 分配到 b 上：

$\lambda :: (Functor\, \Sigma, Functor\, b) \Rightarrow \forall x.\, \Sigma\, (b\, x) \to b\, (\Sigma\, x)$

此时，给定 λ在 Σ 与 b 之间的分配律，以下事实成立

- 操作模型是一个 b-余代数，其类型为 $\mu\Sigma \to b\, (\mu\Sigma)$ 由对以下内容进行折叠得到Σ-代数 $(fmap\ In\ \cdot \lambda) :: \Sigma\, (b\, (\mu\Sigma)) \to b(\mu\Sigma) opsem\, \lambda = fmap\ In\ \cdot \lambda$

  $(\!|\quad\quad|\!)$

- 对偶地，指称模型是一个类型为 Σ (νb) → νb 的 Σ-代数，由余代数 (λ · fmap out) :: Σ (νb) → b (Σ (νb)) 上的展开给出

$$desem\ \lambda = [\![\lambda \cdot fmap\ out]\!]$$

- 指称模型上的折叠是组合式的
- 操作模型上的展开是完全抽象的

此外，指称模型上的折叠与操作模型上的展开相对应。这个映射称为普遍语义 (Turi 和 Plotkin（1997年）。

$$\begin{aligned}&\mathsf{sem} :: \forall \Sigma\ b.\ (\text{函子}\ \Sigma, \text{函子}\ b) \Rightarrow (\forall x.\ \Sigma\ (b\ x) \to b\ (\Sigma\ x)) \to \mu\Sigma \to \nu\\ &b\ \mathsf{sem}\ \lambda t = \mathsf{desem}\ \lambda\ [\![t]\!] = \mathsf{opsem}\ \lambda\ [\![t]\!]\end{aligned}$$

作为简写，我们用语义括号表示这个定义：$[\![t]\!]\lambda = \mathsf{sem}\ \lambda t$。

作为本节的最后一点，值得注意的是，分配律与某些规则格式之间存在对应关系。特别是，我们在本节中看到的这类分配律，对应于结构操作语义（SOS）（Plotkin（2004年））。例如，对于行为函子BHV和一个一般的签名函子，该分配律会导出如下形式的操作语义：

$$\frac{x_1 \overset{l_1}{\rightsquigarrow} x_1' \quad \dots \quad x_n \overset{l_n}{\rightsquigarrow} x_n'}{\sigma\,(x_1, \cdots, x_n) \overset{l}{\rightsquigarrow} \sigma'(y_1, \cdots, y_m)} \tag{3.1}$$

对于一组元变量 $\{y1 \dots, y_m\} \subseteq \{x_1', \dots, x_n'\}$、签名$\sigma$中的构造子 $\sigma' \in \Sigma$ 以及标签 $\{l1, \dots, l_n, l\} \in L$。请注意，分配律并不要求结论中的标签必须属于前提中的标签，但目标中的变量必须来自前提中的目标，这一点确实受到限制。我们将在接下来的章节中看到，这些限制会限制我们所能建模的语言的表达能力。

4 简单非确定性算术语言的完全抽象

我们在 第3节 介绍的语言是一种用于讨论流的基础语言。现在，我们希望为 第1节 中介绍的语言实现一种完全抽象的解释。这种语言是由 Hutton 最初提出的算术表达式语言（1998年），也称为 Hutton剃刀。

在 第4.1节 中，我们介绍 Hutton剃刀；在 第4.2节 中，我们展示如何给出完全抽象语义；在 第4.3节 中，我们抽象到余指函子，以定义这类语言的行为。这里最一般的情形将留待 第5节讨论。

4.1 Hutton剃刀

这种语言的语法与我们在 第3节 中看到的相同，因此我们可以直接重用签名函子 $ValAddF$。Hutton 的 razor 的操作语义由以下归纳关系定义 $(\to) \subseteq \mu\Sigma \times \mu\Sigma$:

$$\frac{}{\mathtt{add}(\mathtt{val}(n), \mathtt{val}(m)) \rightsquigarrow \mathtt{val}(n+m)} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathtt{add}(t_1, t_2) \rightsquigarrow \mathtt{add}(t_1', t_2)}$$

$$\frac{t_2 \rightsquigarrow t_2'}{\mathtt{add}(t_1, t_2) \rightsquigarrow \mathtt{add}(t_1, t_2')}$$

这种语言的语义显然是非确定性的，因为当内层项不是 Val 时，加法运算可能会执行两种不同的归约。因此，为了对其建模，我们使用函子 $\mathscr{P}_{\text{fin}}(-)$ 的观测函数，并通过有限个程序组成的列表来实现。比较这些列表时，应忽略元素的重复和顺序。基于这一点，上述推理规则的实现如下：

$$
\begin{aligned}
&\mathit{smallstep} :: \mu\, \mathit{ValAddF} \to [\mu\, \mathit{ValAddF}] \\
&\mathit{smallstep}\ (\mathit{In}\ (\mathit{Val}\ n)) &&= [\,] \\
&\mathit{smallstep}\ (\mathit{In}\ (\mathit{Add}\ (\mathit{In}\ (\mathit{Val}\ n))\ (\mathit{In}\ (\mathit{Val}\ m)))) &&= [(\mathit{val}\ (n+m))] \\
&\mathit{smallstep}\ (\mathit{In}\ (\mathit{Add}\ t_1\ t_2)) &&= [\mathit{add}\ t_1'\ t_2 \mid t_1' \leftarrow \mathit{smallstep}\ t_1] \\
&&&\quad +\!\!+ [\mathit{add}\ t_1\ t_2' \mid t_2' \leftarrow \mathit{smallstep}\ t_2]
\end{aligned}
$$

注意，val 的基本情形为空，因为 val 情形没有归约。

这个实现本质上存在两个问题。首先，对该观测函数进行展开，会得到有限分支的空 树。例如，只需展开小步语义 (加法 (值 n)(值 m))的定义，就很容易算出结果。直观来说，这是因为计算一步之后，我们会得到值 [值 $(n+m)$] 这是根据第二种情形的定义得出的。如果再对列表中的程序应用小步语义 进行下一步计算，那么根据第一种情形，这些程序（此处只有一个）会归约为空列表。

第二个问题是，要想从 Turi 和 Plotkin 的方法中获益，我们需要一个参数化分配律。这相当棘手，因为在下方的实现中，尚不清楚“ ”这些空位应该填入什么：

λ:: ValAddF ([x]) → [(ValAddF x)]λ(加法 (n :< xs) (m :< ys)) =
([加法 x | x ← xs] ++ [加法 y | y ← ys])

事实上，在 Add (n :<xs) (m :< ys) 的情况下，两个子程序已经生成了类型为 x 的输出列表。由于 x 是全称量化的，我们不知道究竟是 x 中的哪个元素生成了这次归约，而这正是我们补全这些空缺所需要的信息。

4.2 razor 的完全抽象

第一个问题可以通过调整行为函子的定义来解决。其思路是明确语义何时继续、何时停止。具体来说，我们希望将 valn 映射为数字 n，这样值的归约语义就不会是空的。

因此，第一个调整是使用行为函子 $\mathbb{N} + \mathscr{P}_{\text{fin}}(-)$：当语义以一个数字停止时，使用和的左注入；当语义（以非确定性方式）归约为一个程序时，则使用右注入。可以按如下方式实现：

数据 StopAndGo k = Nat | Step [k] 派生 函子

请注意，这里的下划线是该数据类型的构造子。

第二处调整影响更为深远，因为我们需要修改分配律的类型。具体来说，我们希望在输入被解构之前保留一份副本。为此，我们将类型 StopAndGox 与另一个 x 配对，而这个 x

用于保留输入的原始副本。请注意，得到的类型 (x, StopAndGox) 也是关于 x 的一个函子。

ρ:: ∀x. ValAddF (x, StopAndGo x) → StopAndGo (ValAddF x)

我们刚刚插入的这个新结构确保了当 ρ-规则需要原样重复输入时，只需使用左投影。

不过，需要注意的是，这已不再是分配律，因为 (x, StopAndGo x) 和 StopAndGo x 并不相同。为避免混淆，从此我们将这种参数化函数称为 ρ-规则（遵循 Turi 和 Plotkin 的原始工作）。还要注意，这个 ρ-规则不是分配律，这带来了一些理论上的后果，我们需要对此加以处理。暂且先看看如何快速实现我们所需的操作语义。因此，首先考虑以下 ρ-规则的实现

$\rho(\mathrm{Val}\ n) = n\rho(\mathrm{Add}\ (\ ,\ n)\ (\ ,\ m)) = n\ +\underline{m}\rho(\mathrm{Add}\ (x_1, (\mathrm{Step}\ xs_1))\ (x_2,$
$(\mathrm{Step}\ xs_2))) = \mathrm{Step}\ ([\mathrm{Add}\ x_1\, x_2' \mid x_2' \leftarrow xs_2\,] +\!\!+ [\mathrm{Add}\ x_1'\, x_2 \mid x_1' \leftarrow xs_1\,])$
$\rho(\mathrm{Add}\ (x_1,)\ (\ ,\ (\mathrm{Step}\ xs_2))) = \mathrm{Step}\ [\mathrm{Add}\ x_1\, x_2' \mid x_2' \leftarrow xs_2\,]\rho(\mathrm{Add}\ (\ ,$
$(\mathrm{Step}\ xs_1))\ (x_2,)) = \mathrm{Step}\ [\mathrm{Add}\ x_1'\, x_2 \mid x_1' \leftarrow xs_1\,]$

请注意，这个额外参数起到了输入副本的作用。例如，在该定义的第三行中，x_1 是原始程序，它可以化简为程序列表 xs_1；在这种情况下，由于我们不关心这条信息，因此可以保留原始程序。与此 ρ-规则对应的关系定义如下：

$$\frac{}{val\ n \rightsquigarrow \underline{n}} \qquad \frac{t_1 \rightsquigarrow \underline{n} \qquad t_2 \rightsquigarrow \underline{m}}{add\ t_1\ t_2 \rightsquigarrow \underline{n+m}} \qquad \frac{t_1 \rightsquigarrow t_1'}{add\ t_1\ t_2 \rightsquigarrow add\ t_1'\ t_2} \qquad \frac{t_2 \rightsquigarrow t_2'}{add\ t_1\ t_2 \rightsquigarrow add\ t_1\ t_2'}$$

值得注意的是，我们从一组规则——Hutton 的算术表达式语言——出发，而使用分配律迫使我们考虑一个略有不同版本的操作语义。这种形式对应于余指结构操作语义（余指SOS），因为分配律中使用了余指函子。

4.3 关于余指函子

类型 (x, b x) 是关于 x 的函子，特别是 b 的自由余指函子。这一术语借自它的对偶概念，即自由指函子 Either x (b x)，因为 b 增广了一个额外的点。

type $b_\times\ x = (x, b\ x)$

类型 $\cdot_\times$ 显然是一个函子，具有显而易见的 pmap 函数：

$pmap :: Functor\ b \Rightarrow (x \to y) \to b_\times\ x \to b_\times\ y$
$pmap\ f\ (x, y) = (f\ x, fmap\ f\ y)$

此外，还有一个显然的余单位映射，它只是返回输入的副本：

```
ε:: b× a →
aε(x,_) = x
```

给定函子 b 的一个观测函数，$g :: a \to b\,a$，余指函子 $\mathrm{b}_\times$ 的观测函数是一个类型为 $id \wedge g :: a \to b_\times\, a$ 的函数，它将 g 与 a上的恒等映射配对。换句话说，余指函子的观测与其对应函子的观测之间存在一一对应关系[1]。这通过 $\lfloor\cdot\rfloor_\times$ 和 $\lceil\cdot\rceil_\times$

```
⌊·⌋× :: (函子 g) ⇒ (a → g× a) → a → g a⌊
余代数⌋× x = 令 (_,c') = 余代数x in c'⌈·⌉× ::
( 函子 g) ⇒ (a → g a) → a → g× a⌈g⌉× =
恒等映射 ∧g
```

其中 $\wedge$ 是

$$(\wedge) :: (x \to a) \to (x \to b) \to (x \to (a, b))$$
$$(f \wedge g)\, x = (f\, x, g\, x)$$

现在我们要重复 第3节 中的推导过程。为此，我们首先需要将 ρ 规则转化为分配律。然而，为了能够对更复杂的语言建模，我们需要所谓的 GSOS（即守卫结构操作语义），它涵盖了余指 ρ 规则。因此，在介绍构造语义的一般方法之前，我们将先介绍 GSOS。

5 CCS的完全抽象

作为一个更大的案例研究，我们为 Milner 的 通信系统演算 （CCS）（Milner, 1980年）实现完全抽象语义。我们首先回顾 CCS（第5.1节）。然后，我们解释选择行为函子（第5.2节）的原因，并将 CCS 的操作规则编码为 ρ-规则（第5.3节），从而得到 CCS 的完全抽象语义。最后，在第5.5节 中，我们通过一些编程示例展示如何使用该语义进行流式编程。

5.1 CCS的语法和操作规则

值传递 CCS 的语法如下（为简单起见，省略了 CCS 的重标记和条件语句（Milner, 1980年），但可以轻松地将其纳入）：

$$P := \overline{c}(v).P \mid c(x).P_x \mid P_1 + P_2 \mid \mathtt{nil} \mid P_1 \parallel P_2 \mid \mathtt{rep}(P) \mid P\backslash c \tag{5.1}$$

CCS 程序通常称为进程。c(v).P 将值 v $\in \mathbb{V}$ 发送到通道集合 Chan 中某个固定通道 $\in$c，并继续执行 P。类似地，对于进程族 $\{P_x\}_{x\in\mathbb{V}}$，进程 c(x).P_x 从通道集合 Chan 中的通道 $\in$ c 接收一个值（该值绑定到变量 x），并继续执行 P_x。和 $P_1 + P_2$ 可以分别像 P_1 和 P_2 那样动作，而非活动进程 nil 什么也不做。并行组合 $P_1 \parallel P_2$ 使 P_1 和 P_2 并发运行，并可能彼此通信。进程

[1] 这是由于范畴 B-代数同构于余指函子的代数范畴 Id × B-代数

$$\frac{}{\overline{c}(v).P \overset{\overline{c}(v)}{\rightsquigarrow} P} \qquad \frac{v \in \mathbb{V}}{c(x).P_x \overset{c(v)}{\rightsquigarrow} P_v} \qquad \frac{i \in \{0,1\} \qquad P_i \overset{a}{\rightsquigarrow} P}{P_0 + P_1 \overset{a}{\rightsquigarrow} P} \qquad \frac{P \overset{a}{\rightsquigarrow} P'}{P \parallel Q \overset{a}{\rightsquigarrow} P' \parallel Q}$$

$$\frac{Q \overset{a}{\rightsquigarrow} Q'}{P \parallel Q \overset{a}{\rightsquigarrow} P \parallel Q'} \qquad \frac{P \overset{c(v)}{\rightsquigarrow} P' \qquad Q \overset{\overline{c}(v)}{\rightsquigarrow} Q'}{P \parallel Q \overset{\tau}{\rightsquigarrow} P' \parallel Q'} \qquad \frac{P \overset{\overline{c}(v)}{\rightsquigarrow} P' \qquad Q \overset{c(v)}{\rightsquigarrow} Q'}{P \parallel Q \overset{\tau}{\rightsquigarrow} P' \parallel Q'}$$

$$\frac{P \overset{a}{\rightsquigarrow} P'}{\mathtt{rep}(P) \overset{a}{\rightsquigarrow} P' \parallel \mathtt{rep}(P)} \qquad \frac{P \overset{a}{\rightsquigarrow} P' \qquad a \notin \{c(v), \overline{c}(v)\}}{P \backslash c \overset{a}{\rightsquigarrow} P' \backslash c}$$

图1：CCS 的操作语义

rep(P) 表示任意多个 P 副本的并行复制。最后，P\c 限制 P 与外部环境在通道 c 上的通信。

CCS 的（小步）操作语义归纳定义于 图1 中，是一个三元关系 $(\rightarrow) \subseteq P \times$动作 $\times P$，其中 动作 是以下动作集合：

$$\mathsf{Act} = \{\overline{c}(v) \mid c \in \mathit{Chan}, v \in \mathbb{V}\} \cup \{c(v) \mid c \in \mathit{Chan}, v \in \mathbb{V}\} \cup \{\tau\} \tag{5.2}$$

直观地说，$P \overset{\overline{c}(v)}{\rightsquigarrow} Q$ 表示 P 将值 v 发送到通道 c，然后变为 Q；类似地，$P \overset{c(v)}{\rightsquigarrow} Q$ 表示 P 接收 v，然后变为 Q；最后，还有一种特殊动作τ，称为静默动作，对于它，$P \overset{\tau}{\rightsquigarrow} Q$ 表示 P 内部的某次通信成功了，而这一过程对外部环境不可见。

5.2 行为函子的选择

图1 中操作语义的非确定性源于非确定性（$P + Q$）和并行（$P \parallel Q$）运算符。因此，我们需要使用有限幂集函子 $\mathscr{P}_{\text{fin}}(-)$ 来对迁移至有限程序集合进行建模，正如我们在上一节所做的那样。

我们使用有限幂集函子（或某个类型上的有限列表）的原因更多是理论上的：幂集函子 $\mathscr{P}(-)$ 没有不动点，而我们需要它来构造解释函数[1]的陪域。

此外，我们还需要捕捉形如 $P \xrightarrow{a} Q$ 的转移规则所产生的动作。因此，B 的显而易见的选择是 $\mathscr{P}_{\text{fin}}(\mathsf{Act} \times -)$。

然而，考虑接收一个值的进程 $c(x).P_x$ 。它可能的迁移$\{(c(v), P_v) \mid v \in \mathbb{V}\}$ 是 $\mathsf{Act} \times P$ 的一个无限子集。然而， $\mathscr{P}_{\text{fin}}(-)$ 是有限子集函子，因此不适合我们的用途。

为了解决这个问题，我们改为考虑以下行为函子：

$$\mathscr{P}_{\text{fin}}((\mathit{Chan} \times \mathbb{V} \times -) + (\mathit{Chan} \times (-)^{\mathbb{V}}) + 1)$$

直观来说，可观测行为有余积所给出的三种可能形式。“发送”规则会产生一个通道—值对，以及目标进程 $(\mathsf{Chan} \times \mathbb{V} \times \mathsf{X})$。

[1] 这一结果归功于康托尔，他证明了对于每个集合 X，$\mathsf{X} \ncong \mathscr{P}(\mathsf{X})$。

“接收”规则会产生一个通道，以及一个将每个值 v 映射到其进程 P_v (Chan $\times$ $X^{\mathbb{V}}$)) 的函数。最后，两个进程之间的“同步”由类型 1 表示。

5.3 规则格式的选择

现在，我们需要像前几节那样，用 图1 中的 ρ-规则来编码操作规则。然而，简单规则格式（3.1）和带余单位的规则格式（5.3）都不适合在此实现 CCS 的规则。具体来说，这两种规则格式都有限制：结论中的目标必须恰好有一个 构造子作用于元变量 σ′，而不能是多个嵌套的构造子。

第3节和第4节中的算术语言操作规则恰好满足这一要求，但图1中的许多 CCS 规则并不满足。例如，c(v).P $_{c(v)}$ P 的规则中没有构造子作用于 P，而 rep(P) $\overline{a}$ P′ ||rep(P) 的规则中则有两个构造子分别作用于 P 和 P′，即 || 和 rep。

为了克服这一限制，我们需要再次推广 ρ-规则的类型，允许迁移目标中出现任意多个构造子。具体做法是将 ρ 目标中的 Σ 替换为签名 Σ 上的自由单子。最终，ρ 的类型签名应如下所示：

$$\rho :: \forall x.\ \Sigma\ (b_\times\ x) \to b\ (\Sigma^*\ x)$$

其中 Σ^* 是自由单子。现在我们简要回顾它的运算。

5.3.1 Free Monads

给定一个函子 Σ，自由单子 Σ^* 在 Σ 之上定义为以下数据类型：

$$\textbf{data}\ \Sigma^*\ a = Var\ a \mid Op\ (\Sigma\ (\Sigma^*\ a))$$

它配备了以下递归方案：

eval :: 函子 Σ ⇒ (Σ b → b) → (a → b) → Σ^* a →
b evalalg gen (Var v) = gen v evalalg gen (Op
c) = 代数 (fmap (eval alg gen) c)

利用 eval，Σ^* 的单子实例定义如下：

$$return :: a \to \Sigma^*\ a \qquad (\gg\!=) :: Functor\ \Sigma \Rightarrow \Sigma^*\ a \to (a \to \Sigma^*\ b) \to \Sigma^*\ b$$
$$return = Var \qquad x \gg\!= f = eval\ Op\ f\ x$$

上述递归方案的依据是：签名函子 Σ 上的自由单子等价于 ΓX = A + ΣX 满足 μΓ = A + Σ(μΓ) 的最小不动点。

对于每个 Σ-代数，我们都可以通过使用恒等函数和 Σ-代数对语法求值来构造一个 Σ^*-代数，反之亦然

$$\lfloor\cdot\rfloor_* :: Functor\ \Sigma \Rightarrow (\Sigma^*\ a \to a) \to \Sigma\ a \to a \qquad \lceil\cdot\rceil_* :: Functor\ \Sigma \Rightarrow (\Sigma\ a \to a) \to \Sigma^*\ a \to a$$
$$\lfloor f \rfloor_* = f \cdot Op \cdot fmap\ Var \qquad \lceil g \rceil_* = eval\ g\ id$$

映射 $\lceil\cdot\rceil_*$ 和 $\lfloor\cdot\rfloor_*$ 表明， Σ-代数与单子 Σ^*的 Σ^* -代数一一对应[1]。

5.3.2 CCS 的 ρ 规则

首先，我们需要使用签名函子来实现该语言的语法。它将以一种类型为参数： ℂ 通道标签。与往常一样，μ(*CCSl*) 将是归纳生成的 CCS 程序类型（5.1）。

type $\mathbb{V} = \mathbb{N}$type $\mathbb{C}$ =String 数据 CCS x = 发送
$\mathbb{C}\,\mathbb{V}$ x | 接收 $\mathbb{C}\,(\mathbb{V} \to$ x) | 和 x x| 空进程 | 并
行 x x |rep x |限制 $\mathbb{C}$ x

其次，我们需要定义动作集

data *Act x* = *ActS* ℂ 𝕍 *x* | *ActR* ℂ (𝕍 → *x*) | *Silent x*
newtype 动作集 x = 动作集 {unActs :: [动作 x]}

以下两个定义分别是将 concatentation (++) 和 filter 提升到 typeActs 类型上的结果

bapp:: 动作集 x → 动作集 x → 动作集 x
bapp bs bs′ = 动作集 (unActs bs ++unActs
bs′) bfilter :: (动作 x → 2) → 动作集 x → 动
作集 x bfilter p = 动作集 ·filter p ·unActs

现在， 图1 中的 CCS 操作规则可以编码为一个 ρ类型的规则

$\rho_{CCS} :: CCS\,(x, Acts\,x) \to Acts\,(CCS^*\,x)$

发送和接收的规则如下

$\rho_{CCS}\,(Send\,c\,v\,(x, \_)) = Acts\,[ActS\,c\,v\,(return\,x)]$
$\rho_{CCS}\,(Recv\,c\,k) \quad = Acts\,[ActR\,c\,(fmap\,(\lambda\,(x, \_) \to return\,x)\,k)]$

其中，发送进程只有一个动作，即发送动作以及通道、值和剩余计算的副本；接收进程也只有一个动作，即接收动作以及通道和一个续延，该续延总是在接收到值 v 后返回该进程。

0（空进程）的情形就是空动作集

ρ_{CCS} 空进程 = 动作集 []

选择进程 $(P_0 + P_1)$ 是将第一个进程的动作列表与第二个进程的动作列表串接起来，而限制算子 $(P\backslash c)$ 需要一些额外的函数。第一个是 检查函数 ，用于实现条件 $a \notin \{c(v), \overline{c}(v)\}$ (图1)，另一个是 bfilter ，用于筛除所有满足由 检查函数 实现的条件的动作

[1] 这里我们的意思是，单子的代数 Σ^* 与 Σ的代数同构

ρ_{CCS} (和(, b) (, b′)) = fmap return b $+\!\!+_B$ fmap return b′ρ_{CCS} (限制算子 c (, b)) = bfilter 检查函数 (fmap (λy → Op (限制算子 c (return y))) b) where 检查函数 (ActS c′) = c ≠ c′检查函数 (ActR c′) = c ≠ c′检查函数 (Silent) =True

并行情况（||）将四条规则合并为一条，因此其定义稍显复杂也许并不奇怪

ρ_{CCS} (Par xb xb′) = lmerge xb xb′ $+\!\!+_B$ lmerge xb′ xb 其中 lmerge (x, b) (x′, b′) = fmap (λy → Op (Par (return y) (return x′))) b $+\!\!+_B$Acts [Silent (Op (Par (return y) (return (k m)))) | (ActS c m y) ← unActs b, (ActR c′ k) ← unActs b′, c ≡ c′]
这里，lmerge 接收初始抽象进程 x 和 x′ 以及它们各自的动作列表 b 和 b′，并生成一个集合：其中包含由 x 单步归约得到的动作列表，以及由 x 发送与 x′接收配对产生的静默动作列表。在顶层，该规则输出合并 x 和 x′ 的输出与其镜像情况所得结果的并集。

rep 的规则更易于实现，但关键在于它需要自由单子中的结构，因为它返回两层语法：第一层是 Par 构造子，第二层构造子是 Rep

ρ_{CCS} (Rep (x, b)) = fmap (λy → Op (并行 (return y) (Op (Rep (return x))))) b

此时，我们需要从 GSOS 规则中恢复分配律，由此可以构造普遍语义。

5.4 一般情形：恢复分配律

CCS 的 ρ-规则可以按以下方式推广。对于签名函子 Σ 和行为函子 b， ρ-规则具有以下类型：

$\rho :: \forall x.\ \Sigma\ (b_\times\ x) \to \Sigma^*\ (b\ x)$

这个参数化函数产生了一种称为 GSOS 的规则格式

$$\frac{x_1 \overset{l_1}{\rightsquigarrow} x_1' \ldots x_n \overset{l_n}{\rightsquigarrow} x_n'}{\sigma(x_1, \cdots, x_n) \overset{a}{\rightsquigarrow} t(y_1, \cdots, y_m)} \tag{5.3}$$

对于元变量 $\{y1 \ldots, y_m\} \subseteq \{x1, \ldots, x_n\} \cup \{x_1', \ldots, x_n'\}$ 以及构造子 $\sigma, \in \Sigma$ 和程序 $t\ \in \Sigma^*$。

现在，我们需要根据 ρ，为自由单子 Σ^* over $B_\times$ 恢复一个分配律。我们可以使用自由单子求值 eval 的递归方案来实现：

rhoToLambda :: ∀Σ b. (函子 Σ, 函子 b) ⇒ (∀x. Σ ($b_\times$ x) → b (Σ^* x)) → (∀x. Σ^* ($b_\times$ x) → $b_\times$ (Σ^* x))

rhoToLambda ρ = 求值 代数 (pmap return) where

代数 :: Σ ($b_\times$ (Σ^* x)) → $b_\times$ (Σ^* x) 代数 =

(Op ·(fmap ε)) ∧ ((fmap (⩥id)) ·ρ)

借助分配律 rhoToLambda ρ，由 ρ 所诱导的通用操作语义 opsem 和指称语义 desem，可以按照前几节的做法类似地定义。这里只需使用运算符 $\lceil\cdot\rceil_*$、$\lfloor\cdot\rfloor_*$、$\lceil\cdot\rceil_\times$ 和 $\lfloor\cdot\rfloor_\times$ 对代数进行提升和降级，使类型相匹配

$$
\begin{aligned}
&opsem :: \forall\Sigma\, b.\,(\text{函子 } \Sigma, \text{函子 } b) \Rightarrow (\forall x.\ \Sigma\,(b_\times\, x) \to b\,(\Sigma^*\, x)) \to \mu\Sigma \to b\,(\mu\Sigma)\\
&sem\ \rho = \rho\, alg\ \lfloor\ \text{where}\ \rfloor_*\\
&\quad alg :: \Sigma\,(b_\times\,(\mu\Sigma)) \to b_\times\,(\mu\Sigma)\\
&\quad alg = \lfloor pmap\,(\lceil In\rceil_*) \cdot (rhoToLambda\,\rho)\rfloor_*
\end{aligned}
$$

指称语义同理：

$$
\begin{aligned}
&desem :: \forall\Sigma\, b.\,(\text{函子 } \Sigma, \text{函子 } b) \Rightarrow (\forall x.\ \Sigma\,(b_\times\, x) \to b(\Sigma^*\, x)) \to \Sigma\,(\nu b) \to \nu\\
&b\ desem\ \rho = \lfloor\ \text{coalg}\ \rfloor_*\ \text{where}\ coalg :: \Sigma^*\,(\nu b) \to b\,(\Sigma^*\,(\nu b))\ coalg = \lfloor\\
&rhoToLambda\,\rho \cdot fmap\,(\lceil out\rceil_\times)\rfloor_\times
\end{aligned}
$$

5.4.1 完全抽象论证

由于 λ具有分配性，函数 $desem\ \rho$ 和 $opsem\ \rho$ 总是相同（Hinze 和 James，2011）；并且由于 $opsem\ \rho$ 是一个展开，因此它总是完全抽象的。以下递归方案在给定抽象规则 μΣ 的情况下解释一种语言 ρ（照例，我们将 t ρ 表示为 $sem\ \rho t$）：

$$
\begin{aligned}
&sem :: \forall\Sigma\, b.\,(\text{函子 } \Sigma, \text{函子 } b) \Rightarrow (\forall x.\ \Sigma\,(b_\times\, x) \to b\,(\Sigma^*\, x)) \to \mu\Sigma \to \nu\\
&b\ sem\ \rho t = desem\ \rho\ t \text{ -- 或 } opsem\ \rho\ t
\end{aligned}
$$

将 sem 应用于 ρ_{CCS} 后，我们得到以下函数，它将 CCS 程序的（全局）行为作为其语义：

$$
\begin{aligned}
&sem_{CCS} :: \mu CCS \to \nu\text{动}\\
&\text{作集 } sem_{CCS} = sem\ \rho_{CCS}
\end{aligned}
$$

在本节余下部分中，我们继续通过这个递归方案展示 CCS 示例，并说明如何将其用于函数式编程。

5.5 编程示例

在本节余下部分中，我们通过 CCS 中的编程示例，展示由 ρ-规则所导出的语义如何发挥作用。为方便起见，我们定义以下辅助函数，用 In 包装构造子：

$$
\begin{aligned}
&p \parallel q = \text{属于 (并行 } p\ q);\ \text{空进程} = \text{属于 空进程}\ send\ c\ m\ p =\\
&\text{属于 (发送 } c\ m\ p);\ \text{和}\ p\ q = \text{属于 (求和 } q)\ rep\ p = \text{属于 (rep}\\
&p);\ \text{限制}\ c\ p = \text{属于 (限制 } c\ p)\ \text{接收}\ c\ k = \text{属于 (接收 } c\ k);\ res'\\
&ls\ p = \text{用 } res \text{ 对 } p \text{ 和 } ls \text{ 进行右折叠}
\end{aligned}
$$

这些只是后置应用于 In 的语言构造子，而 res′ is 则是将通道列表ls 限制到进程 p 上。

例 5.1。我们可以使用 CCS 进行流式编程。以下程序将所有自然数逐个发送到通道“output”：

nats :: μCCS nats = res "i" (发送 "i" 0 空进程 || rep iter)
where iter = 接收 "i" (λi → 发送 "output"i (发送 "i" (i + 1)
空进程))

进程 iter 从通道 "i" 接收当前值，并将其发送到通道 "output"。rep 将进程 iter 复制为无限多个副本，因此 iter 可以将 i + 1 发送给自身的另一个副本，以继续迭代。最后，通过 send "i" 0 nil 启动迭代，而 res "i" 限制进程内部通道 "i" 上的通信，使其他进程无法干扰迭代，只能观察通道 "output"。

对该进程应用语义函数，可得到 sem_{CCS} nats:: ν动作集。直观来说，这是一个（余归纳）树，其边以动作 Act 为标签，不同路径代表非确定性的不同可能性。我们可以使用以下函数，将这棵树中的所有输出收集为一个列表：

outputs :: ν动作集 → [ℕ]outputs (Out◦ (动作
集 [])) = []outputs (Out◦ (动作集 bs)) =
concat (map f bs) where f (ActS m bs′) =
m : outpūts bs′f (ActR) = []f (Silent bs′) =
outputs $\overline{\mathrm{bs}}^{\frown}$

并且确实，outputs (sem_{CCS} nats) = [0, 1, 2, 3, ...]。

示例 5.2。sem_{CCS} 语义对于探索并发系统的非确定性行为很有用，因为所有非确定性的可能性都记录在语义 ν动作集中。举例来说，下面的程序使用本节开头定义的子程序实现了一个计数器：

counter = res "init" (send "init" 0 空进程 || iter) where
iter = rep (recv "init" (λv → 和 (recv "rd" (_ → send "
count" v (send "init" v 空进程))) (recv "wt" (λi → send "
init"i 空进程))))

直观来说，计数器是一个存储名为 init 的值的可移动程序，并会响应其他进程发出的读取或写入请求；这些进程希望读取或覆盖该存储单元的内容。

可以通过访问通道“rd”、“wt”和“count”来读取和修改该单元，具体如下：

read :: (ℕ → μCCS) → μCCS write :: ℕ → μCCS → μCCS read k
= 发送 "rd" 0 (接收 "count"k) write v p = 发送 "wt" v p

假设有两个进程在不使用锁的情况下并发递增计数器，如以下程序所示：

incr :: μCCS → μCCS incr p =读取 (λv → 写入 (v + 1) p) 计数器测试 = 结果′ ["读取", "写入", "计数"](计数器 || 递增 (递增 空进程)|| 递增 (递增 (读取 (λv → 发送 "输出" v 空进程))))

然后，执行该语义便能得到所有可能的结果：

$$nub\,(outputs\,(sem_{CCS}\,counterTest)) = [4, 3, 2, 1]$$

其中，nub 是一个用于移除列表中重复项的函数。

6 相关工作

从范畴论的角度来看，余代数系统理论已得到深入研究（Rutten 和 Turi，1993；Turi 和 Plotkin，1997年；Jacobs 和 Rutten，2012；Rutten，2000；Staton，2011），而其与结构操作语义（SOS）（Plotkin，2004年）之间的联系也已由 Klin（2011）作了详尽阐述。

在函数式编程领域，Turi 和 Plotkin 的分配律（Turi 和 Plotkin，1997年）已被证明有广泛的应用：证明唯一不动点原理正确（Hinze 和 James，2011）；以模块化方式定义操作语义（Jaskelioff 等人，2011）；以及证明排序算法的等价性（Hinze 等人，2012）。Hutton 的工作（Hutton，1998年）旨在推广将折叠和展开用于程序语义，而 razor 也一直被用于文献中，以阐释编程语言设计、语义和编译器中的关键思想（Hutton，1998年，2021；Bahr 和 Hutton，2015；Hutton，2021）。我们的工作采用了略有不同的方法。我们对 razor 做了调整，使其符合这样一种观点：指称语义是定义在语义域上的代数，由展开定义；而操作语义是一个余代数，由语法上的折叠定义。依此观点，对指称语义进行折叠，等价于对操作语义进行展开；利用分配律，完全抽象便可自然得出。Hinze 和 James（2011）以及 Hinze 等人（2012）已对这些结论的证明作了详尽阐述；但另一方面，他们并未讨论完全抽象方面的结果。

7 讨论

递归方案的发明是为了确保递归定义在数学上是良定义的。然而，递归方案的作用远不止于此。折叠是组合式解释，而展开则是完全抽象解释。具体而言，它们可以“运行”一个迁移系统（或自动机），并为某个状态构造其轨迹，使得轨迹相同的状态对应于操作等价状态。

当迁移系统（例如操作语义）和指称语义由某个分配律导出时，对指称语义进行折叠所得的结果，等于对操作语义进行展开所得的结果。

参考资料

Bahr, P. & Hutton, G.（2015）计算正确的编译器。《函数式编程杂志》25。Danielsson, N. A., Hughes, J., Jansson, P. & Gibbons, J.（2006）快速且宽松的推理在道德上是正确的。第33届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集，POPL 2006，美国南卡罗来纳州查尔斯顿，2006年1月11—13日。ACM。Hinze, R.（2008）函数式珍珠：流与唯一不动点。第13届 ACM SIGPLAN 函数式编程国际会议论文集，ICFP 2008，加拿大不列颠哥伦比亚省维多利亚，2008年9月20—28日。ACM。Hinze, R. & James, D. W. H.（2011）证明唯一不动点原理正确：一次范畴论探险。第16届 ACM SIGPLAN 函数式编程国际会议论文集，ICFP 2011，日本东京，2011年9月19—21日。ACM。Hinze, R., J ames, D. W. H., Harper, T., Wu, N. & Magalh˜aes, J. P.（2012）利用双代数和分配律排序。第8届 ACM SIGPLAN 泛型编程研讨会论文集，WGP@ICFP 2012，丹麦哥本哈根，2012年9月9—15日。ACM。第69—80页。Hutton, G.（1998）程序语义中的折叠与展开。第三届 ACM SIGPLAN 函数式编程国际会议（ICFP ’98）论文集，美国马里兰州巴尔的摩，1998年9月27—29日。ACM。第280—288页。Hutton, G.（2021）简单如一二三。未发表手稿。Jacobs, B. & Rutten, J.（2012）关于（余）代数与（余）归纳的导论。载于《互模拟与余归纳高级主题》，Sangiorgi, D. & Rutten, J. J. M. M.（编）。《剑桥理论计算机科学论丛》第52卷。剑桥大学出版社。第38—99页。Jaskelioff, M., Ghani, N. & Hutton, G.（2011）数学操作语义的模块化与实现。《电子理论计算机科学纪事》229(5)，75—95。Klin, B.（2011）结构操作语义的双代数：导论。《理论计算机科学》。Meijer, E., Fokkinga, M. M. & Paterson, R.（1991）使用香蕉、透镜、信封和带刺铁丝网进行函数式编程。第五届 ACM 函数式编程语言与计算机体系结构会议论文集，美国马萨诸塞州剑桥，1991年8月26—30日。施普林格。第124—144页。Milner, R.（1980）《通信系统演算》。计算机科学讲义第92卷。施普林格。Plotkin, G. D.（2004）操作语义的结构化方法。《逻辑与程序代数方法杂志》。Rutten, J. J. M. M.（2000）泛余代数：一种系统理论。《理论计算机科学》。Rutten, J. J. M. M. & Turi, D.（1993）并发的初始代数与终余代数语义。《并发十年：回顾与展望》，REX 学校／研讨会，荷兰诺德韦克豪特，1993年6月1—4日，论文集。施普林格。Staton, S.（2011）关联余代数互模拟概念。《逻辑方法与计算机科学》。Turi, D. & Plotkin, G. D.（1997）迈向数学操作语义。第 12 届 IEEE 计算机科学逻辑研讨会论文集，波兰华沙，1997年6月29日—7月2日。IEEE 计算机学会。