arXiv:2302.01415v1 [cs.PL] 2 Feb 2023

高阶效应与处理器框架

Birthe van den Berg 和 Tom Schrijvers
鲁汶大学，比利时

摘要
代数效应与处理器是函数式编程中建模副作用的一种模块化方法。其语法由效应操作的签名定义，这些操作被编码为函子并接入自由单子；其指称语义则由折叠式处理器定义，处理器只解释语法中属于自己的部分，并将其余部分转发出去。然而，并非所有效应都是代数效应：有些效应需要访问内部计算。例如，作用域效应区分作用域内和作用域外的计算；并行效应对计算进行并行化，潜伏效应则延迟计算。针对这些高阶效应及其对应的处理器，人们提出了彼此独立的定义，这往往导致权宜而复杂的单子定义。在本文中，我们提出了一个用于高阶效应的通用框架，将代数效应与处理器加以推广：一种具有高阶效应签名的通用自由单子，以及与之对应的解释器。对这种高阶语法进行特化，可以导出此前定义的多种效应（作用域效应、并行效应、潜伏效应），以及新颖的效应（写入效应、括号效应）的定义。此外，我们从形式上证明了该框架在理论上是正确的，也为不同的效应实例奠定了形式化基础；这对并行效应、潜伏效应、写入效应和括号效应而言是一项重要贡献。

关键词：代数效应与处理器、高阶效应与处理器、自由单子、可组合数据类型 `ala carte

1. 引言

自九十年代以来，单子 [16, 17, 29] 一直是纯函数式环境中对效应进行建模的标准方法。代数效应与处理器 [21, 23],近年来却日益受到广泛关注 [25, 10, 11]。后者为组合不同种类的效应提供了一种模块化方法。尤其是，代数操作（语法）与处理器（语义）的清晰分离，使其极具吸引力。使用这些操作的余积，按照特定顺序组合不同的代数效应

[27] 意味着处理器的顺序也不同，进而导致不同的解释 [34]。处理器只了解语法中属于自己的部分，并将未知效应转发给其他处理器。

尽管这种代数效应与处理器的模块化技术适用于所有带效应的程序，但并非所有效应都适合这种方法。代数性指效应操作与顺序执行可交换，但并非所有类型的效应都满足这一性质。Plotkin 和 Power 提出 [23] 将非代数效应建模为处理器，但事实证明 [31, 33]这种做法会损害模块化。

在这项工作中，我们为不同的效应提出一种通用语法和指称语义，从而保留语法与语义分离所带来的理想模块化特性。具体而言，我们关注高阶效应，即能够访问内部计算的效应。例如，作用域效应与处理器 [33, 20, 31] （如捕获异常、局部变量）区分作用域内的操作与作用域外的延续；潜伏效应与处理器 [1] （如惰性求值、分阶段计算）会将内部计算推迟到之后的执行时刻；并行效应与处理器 [32]（如循环）则会并行化内部计算，以提升性能或效率。

我们通过扩展代数效应与处理器的自由单子定义来实现所需的模块化，使其支持高阶函子；这些函子会将内部计算纳入其效应签名。再结合通用的折叠式解释器，便构成了用于建模不同效应的框架。具体而言，本文的贡献可总结如下：

- 我们定义了一个具有高阶效应签名的通用自由单子 [24]，并给出了相应的折叠式解释器，以表示能够对内部计算进行推理的效应（第 3 节）。
- 我们将文献中的不同效应建模为该框架的实例，包括代数效应与处理器、作用域效应与处理器、并行效应与处理器，以及潜伏效应与处理器（第 4 节）。
- 我们建模了两类新颖的效应：Writer 效应与处理器，以及括号效应与处理器，分别用于表示写入日志和安全处理资源的功能（第 4 节）。
- 我们证明，每一类（已有和新颖的）效应都可以通过一个自由单子及其对应的递归方案来建模（第 4 节）。
- 我们为所有效应提供实用的编程示例，包括不同效应的组合（第4节）。

- 我们通过基于自由-遗忘伴随的范畴模型，为自由单子的定义提供支撑（第5节）。
- 我们提供了一个完整的 Haskell 实现，并附有更多详细示例（补充材料）。

本文始终使用 Haskell 作为载体来展示我们的研究成果。

2. 背景与动机

本节介绍必要的预备知识，首先讲解函子、自由单子及其与代数效应与处理器的关系。

2.1. 代数效应与处理器

代数效应与处理器 [21, 23] 使用自由单子和解释器，将效应的语法与其语义分离。在语法层面，我们区分由其签名决定的代数操作，以及计算：通过自由单子在这些操作之上构造的递归结构。此外，递归方案（处理器）会解释这些计算。

代数操作：签名。遵循 Plotkin 和 Pretnar 的方法 [22], ，我们通过效应操作的签名 σ 对其进行建模。效应签名被建模为函子。

$$\textbf{class } \mathit{Functor}\ \sigma \textbf{ where}$$
$$\quad \mathsf{fmap} :: (a \to b) \to \sigma\ a \to \sigma\ b$$

效应签名将多个效应操作组合在一起。例如，状态效应的签名包含两个操作：用于读取并返回状态的 Get，以及用于修改状态、用给定值覆盖它的 Put。

$$\textbf{data } \mathit{State}\ s\ a = \mathit{Get}\ (s \to a) \mid \mathit{Put}\ s\ a$$

计算：自由单子。使用函子作为签名，我们可以构造一种递归数据类型来表示计算，这些计算可以是纯计算，也可以是效应ful计算。随后，计算可以调用签名中包含的操作。这种递归数据类型称为自由单子：

$$\mathsf{data\ Free}\ (\sigma :: * \to *)\ \mathsf{a\ where}$$
$$\mathsf{Var} :: \mathsf{a} \to \mathsf{Free}\ \ \sigma\ \mathsf{a\ Op} :: \sigma\ (\ \mathsf{Free}\ \ \sigma\ \mathsf{a}) \ \to \mathsf{Free}\ \ \sigma\ \mathsf{a}$$

instance Functor σ ⇒Monad
(Free σ) return =Var Var x $\gg$=k =
k x Op t $\gg$=k =Op (fmap ($\gg$=k) t)

这里，Var 构造器表示纯计算，而 Op 则以递归方式组合计算，其分支结构由签名 σ 决定。等价地，我们可以将 Op 构造器表示为接受两个参数：一个操作和一个延续 [9]。

$$Op :: \forall\, b\,.\ \underbrace{\sigma\ b}_{operation} \rightarrow \underbrace{(b \rightarrow Free\ \sigma\ a)}_{continuation} \rightarrow Free\ \sigma\ a$$

例如，单子 Free ($State$ String) 整数表示状态类型为 String、结果类型为整数的计算。通常，我们会定义构造器，以便于编写带效应的计算。例如，对于 State s，我们将 get 和 put 分别定义为读取和设置状态的计算。

$$\begin{aligned} &\mathsf{get} :: Free\ (State\ s)\ s && \mathsf{put}\ \ :: s \rightarrow Free\ (State\ s)\ () \\ &\mathsf{get} = Op\ (Get\ \mathsf{return}) && \mathsf{put}\ s = Op\ (Put\ s\ (\mathsf{return}\ ())) \end{aligned}$$

我们使用单子绑定来顺序执行带效应的计算。

解释：折叠。我们使用递归方案来解释自由单子：编写一个折叠式解释器，通过生成元和代数为自由单子中的计算赋予语义。具体来说，要将自由单子 Free σ a 解释到语义域 b 中，需要一个生成元 gen :: a → b，用于将纯计算转换到语义域 b；还需要一个代数 alg :: σ b → b，用于解释签名的效应操作。

$$\begin{aligned} &\mathsf{fold}_{\mathsf{Alg}} :: Functor\ \sigma \Rightarrow (a \rightarrow b) \rightarrow (\sigma\ b \rightarrow b) \rightarrow Free\ \sigma\ a \rightarrow b \\ &\mathsf{fold}_{\mathsf{Alg}}\ gen\ alg\ (Var\ x) = gen\ x \\ &\mathsf{fold}_{\mathsf{Alg}}\ gen\ alg\ (Op\ op) = alg\ (\mathsf{fmap}\ (\mathsf{fold}_{\mathsf{Alg}}\ gen\ alg)\ op) \end{aligned}$$

例如，以下处理器使用此结构递归方案，将 Free (State s) a 解释为语义域 s → (s,a)。

$$\begin{aligned} &h_{\mathsf{State}} :: Free\ (State\ s)\ a \rightarrow (s \rightarrow (a, s)) \\ &h_{\mathsf{State}} = \mathsf{fold}_{\mathsf{Alg}}\ (,)\ alg\ \mathbf{where} \\ &\quad alg\ (Get\ k) \quad = \lambda s \rightarrow k\ s\ s \\ &\quad alg\ (Put\ s'\ k) = \lambda_ \rightarrow k\ s' \end{aligned}$$

可以将上述状态处理器应用于一个计算：该计算会递增状态并返回原始状态，初始值为 0。

$$\text{>>>}\ h_{\mathsf{State}}\ (\mathsf{get} \gg\!= \lambda s \to \mathsf{put}\ (s+1) \gg \mathsf{return}\ s)\ 0$$
$$(0, 1)$$

结果为 (0, 1)，其中 0 是结果值，1 是（递增后的）状态 .

模块化组合效应。人们常常需要组合多种效应。此类复合通常通过“Datatypes [27]a la Carte” [27] 方法建模，即使用函子的余积；在此处，也就是效应签名的余积（记作 $+$）[1]。

$$\mathbf{data}\ (\sigma_1 + \sigma_2)\ a = Inl\ (\sigma_1\ a) \mid Inr\ (\sigma_2\ a)$$

例如，我们可以将有状态计算与非确定性组合起来。非确定性操作可以失败，也可以分支。

$$\mathbf{data}\ Choice\ a = Fail \mid Or\ a\ a \qquad\qquad Free\ (State\ s + Choice)\ a$$

这种模块化组合允许不同的效应交互 [3, 34], 例如，通过交换状态与非确定性的处理器，实现局部状态语义与全局状态语义 [19], 。

也可以将效应操作与未知效应组合起来，随后将未知效应转发给其他处理器。分隔符 # 用于区分要解释的代数和要转发的效应。我们重新定义上述状态处理器，使其包含“其他”（未知）效应，并将这些效应转发给其他处理器解释，同时把状态传递给这些处理器。

$$\begin{aligned}
&h'_{\mathsf{State}} :: Functor\ \sigma \Rightarrow Free\ (State\ s + \sigma)\ a \to (s \to Free\ \sigma\ (a, s))\\
&h'_{\mathsf{State}} = \mathsf{fold}_{\mathsf{Alg}}\ gen\ (alg\ \#\ fwd)\ \mathbf{where}\\
&\quad gen\ x \qquad\quad = \lambda s \to \mathsf{return}\ (x, s)\\
&\quad alg\ (Get\ k) \quad = \lambda s \to k\ s\ s\\
&\quad alg\ (Put\ s'\ k) = \lambda_ \to k\ s'\\
&\quad fwd\ op \qquad = \lambda s \to Op\ (\mathsf{fmap}\ (\$s)\ op)
\end{aligned}$$

2.2. 非代数效应

并非所有效应都是代数性的；有些效应表示更高级的效应ful 计算 [1, 31, 33]。如果效应满足代数性，即代数计算与顺序执行可交换，那么这些效应就是代数性的：

[1]为提高可读性， [27], 我们在示例中省略了 Inl 和 Inr。

$$(\gg\!= k) \circ Op \equiv Op \circ \mathsf{fmap}\ (\gg\!= k)$$

尽管许多效应都满足这一性质，但并非所有效应都满足。

例如，作用域效应与处理器 [20, 31, 33] 区分作用域计算（表示程序中处于作用域内的部分）与延续（指程序中其余位于作用域外的部分）。这一复杂性要求重新表述代数性，并使用更复杂的自由单子来表示这些效应。作用域效应与处理器的示例包括局部变量、捕获异常、局部剪枝非确定性等。此外，并行效应与处理器 [32]会并行执行带有代数效应的计算，例如使用循环。这些效应包含一个待并行执行的计算，以及一个延续。此外，潜伏效应与处理器 [1]会延迟效应程序的某些部分，使其在执行过程中稍后求值。它们的语法也由一个待延迟的计算和一个延续组成。借助这种方法，可以对存在效应时的惰性求值策略、分阶段计算等进行建模。

这些效应类别中的每一类[2] 都已通过自由单子的专门化版本进行指称语义建模，使得它们的表示具有临时性且本质上彼此不同。此外，用于对并行效应和潜伏效应建模的单子尚未被证明是自由结构。

这项工作建立在一个基本洞见之上：这些效应有一些共同之处，即它们都具有内部计算。确切地说，对于作用域效应，这个内部计算就是作用域中的计算；对于并行效应，它是要进行并行化的计算；对于潜伏效应，它是要延迟执行的计算。我们将这些效应称为高阶效应。基于这一洞见，我们认为可以将代数效应与处理器的框架加以泛化，从而也能对更高级的效应进行建模。这一通用框架保留了将语法（借助自由单子）与语义（借助折叠）分离所带来的模块化特性。我们以范畴论基础为该框架提供支持，证明了：（1）它确实是一个自由单子；（2）它适用于一系列示例。

3. 用于建模效应的通用自由单子

我们用于建模高阶效应的框架采用通用自由单子和解释器。这种表示有意不偏离代数效应与处理器的方法（第 2.1 节）太远；该方法的模块化已通过代数效应在

[2]我们将在第 4 节中更详细地讨论它们。

不同编程语言 [2, 5, 12, 13] 和库 [10, 11, 25]中的采用，证明了其重要性。具体而言，我们再次使用自由单子，并通过高阶函子将内部计算纳入框架，同时还引入了余积。

效应签名。高阶效应需要访问内部计算。为了在签名中体现这一点，函子表示的能力并不足够。与 Poulsen 和 van der Rest [24], 一样，我们将效应签名推广为包含一个高阶函子 k : (∗ → ∗) → (∗ → ∗)，它是函子之间的映射。在这里，函子参数（种类为 ∗ → ∗）表示内部计算，因此 k f 具有熟悉的代数效应签名形式（一个普通函子）。我们据此对其进行建模 [31]:

$$\textbf{class}\ (\forall f\,.\,Functor\ f \Rightarrow Functor\ (k\ f)) \Rightarrow HFunctor\ k\ \textbf{where}$$
$$\quad \mathsf{hmap} :: (Functor\ f, Functor\ f') \Rightarrow f \rightsquigarrow f' \rightarrow k\ f \rightsquigarrow k\ f'$$

在这里，( ) 表示两个函子之间的自然变换[3]。例如，异常的高阶签名如下:

$$\textbf{data}\ Exc\ f\ r = Throw \mid \forall\, a\,.\,Catch\ (f\ a)\ (Maybe\ a \rightarrow r)$$

捕获异常时的延续（Maybe a → r）取决于内部计算 f a 中是否抛出了异常。

计算。与用于代数效应与处理器的自由单子类似，现在我们可以构造一个使用上述签名推广形式的自由单子。

数据 $Free_{\mathsf{H}}$ k a，其中Var_{H} :: a → $Free_{\mathsf{H}}$ k aOp_{H} :: k ($Free_{\mathsf{H}}$ k) ($Free_{\mathsf{H}}$ k a) → $Free_{\mathsf{H}}$ k a 实例 $HFunctor$ k ⇒单子 ($Free_{\mathsf{H}}$ k)，其中 return = $Var_{\mathsf{H}}Var_{\mathsf{H}}$ x >>=k = k xOp_{H} op >>= k = Op_{H} (fmap (>>=k) op)

在第5节中，我们将证明这确实是一个自由单子。同样，利用余 Yoneda 引理，我们重写 Op_{H} 构造器：它包含一个操作、一个内部计算和一个延续。

$$Op_{\mathsf{H}} :: \forall \mathsf{f\ b}\,.\,(\underbrace{\mathsf{k\ f\ b}}_{operation}) \rightarrow (\underbrace{\mathsf{f}\ Free_{\mathsf{H}}\ \mathsf{k}}_{internal\ computation}) \rightarrow (\underbrace{\mathsf{b} \rightarrow Free_{\mathsf{H}}\ \mathsf{k\ a}}_{continuation}) \rightarrow Free_{\mathsf{H}}\ \mathsf{k\ a}$$

[3]类型 f g = ∀a . f a → g a

例如， $prog_{\mathsf{Exc}}$ 如果其参数小于 0，就会抛出异常。

$prog_{\mathsf{Exc}}$ :: 整数 $\rightarrow Free_{\mathsf{H}}$ Exc 字符串$prog_{\mathsf{Exc}}$x $= Op_{\mathsf{H}}$(Catch (if x $\geqslant 0$then return x else Op_{H}Throw) k) where k Nothing $=$ return `"Too small"`k (Just x) $=$return (show x)

解释。我们为这个自由单子配备一个折叠式解释器:

$$
\begin{aligned}
&\mathsf{fold} :: \forall k\ g\ a\ b\,.\,(HFunctor\ k, Pointed\ g)\\
&\quad \Rightarrow (a \rightarrow g\ b) \rightarrow (\forall x\,.\,k\ g\ (g\ x) \rightarrow g\ x) \rightarrow (Free_{\mathsf{H}}\ k\ a \rightarrow g\ b)\\
&\mathsf{fold}\ gen\ alg\ (Var_{\mathsf{H}}\ x) \ = gen\ x\\
&\mathsf{fold}\ gen\ alg\ (Op_{\mathsf{H}}\ \ op) = alg\ (\mathsf{hmap}\ \mathsf{fold}_2\ (\mathsf{fmap}\ (\mathsf{fold}\ gen\ alg)\ op))\\
&\quad \textbf{where}\ \mathsf{fold}_2 :: Free_{\mathsf{H}}\ k \rightsquigarrow g\\
&\qquad\quad \mathsf{fold}_2\ (Var_{\mathsf{H}}\ x) = \eta\ x\\
&\qquad\quad \mathsf{fold}_2\ (Op_{\mathsf{H}}\ t) \ = alg\ (\mathsf{hmap}\ \mathsf{fold}_2\ (\mathsf{fmap}\ \mathsf{fold}_2\ t))
\end{aligned}
$$

这个折叠由两部分组成：一部分将 $Free_{\mathsf{H}}$ k a 解释到语义域 g b，另一部分解释内部计算： $Free_{\mathsf{H}}$k g。因此，人们会预期这两个折叠对应两个生成器和两个代数。然而，fold_2 依赖于 g 是一个带基函子这一事实，它隐含地带有一个生成元 $\forall$ a . a $\rightarrow$ g a，因此只需一个显式生成元即可。

$$
\begin{aligned}
&\textbf{class}\ Functor\ g \Rightarrow Pointed\ g\ \textbf{where}\\
&\quad \eta :: a \rightarrow g\ a
\end{aligned}
$$

此外，为了简洁起见，我们选择在两个折叠中复用同一个代数（其定义域是全称量化的），以一致地解释内部计算和延续。这大大降低了处理器的复杂性，但也意味着某些效应不受支持（第4.2节）。例如，我们的异常处理器使用 Maybe 来解释结果。

h_{Exc} :: $Free_{\mathsf{H}}$ Exc a $\rightarrow$ Maybe a
$h_{\mathsf{Exc}} =$ 折叠 Just alg，其中 alg
Throw =Nothing alg (Catch c k)
$=$ k c

>>> h_{Exc} ($prog_{\mathsf{Exc}}$ 5) Just `"5"`>>> h_{Exc} ($prog_{\mathsf{Exc}}$ (-5))
Just `"Too small"`

模块化组合。为了以“Datatypes `a la Carte” [27], 的风格组合不同的效应，我们需要一个高阶

序函子 k_1 和 k_2。这个余积[4] （记作 $\oplus$，分隔符为 $\#$ ）的作用方式与函子的余积 $\textcircled{+}$（第 2 节）类似。

$$\begin{aligned}
&\mathbf{data}\ (k_1 \oplus k_2)\ f\ a = In\ (k_1\ f\ a) \mid Out\ (k_2\ f\ a)\\
&(\textcircled{\#}) :: (k_1\ f\ a \to g\ b) \to (k_2\ f\ a \to g\ b) \to (k_1 \oplus k_2)\ f\ a \to g\ b\\
&(lft \textcircled{\#} rht)\ (In\quad op) = lft\ op\\
&(lft \textcircled{\#} rht)\ (Out\ op) = rht\ op
\end{aligned}$$

4. 不同效应作为我们框架的实例

不同的效应（代数效应和高阶效应）都可以视为我们通用框架的实例，只需定义一个适当的高阶函子，将它们的签名映射到通用设置中。具体而言，实例化该框架包含四个步骤:

步骤 1 通过定义高阶函子 K，将效应签名映射为高阶表示。该函子将函子 F 和类型 A 映射到种类为 $*$ 的类型。

步骤 2 证明这个 K 确实是一个高阶函子。

步骤 3 将它代入通用自由单子 $Free_{\mathsf{H}}$，并证明它与专用效应定义同构。

步骤 4 使用通用折叠函数为这些效应编写处理器，并证明它与专用效应处理器同构（如果存在的话）。

接下来，我们通过以下四个步骤，将框架实例化以适用于不同类别的效应。

4.1. 代数效应与处理器

我们针对代数效应遵循这些步骤，并说明其框架特化与第 2.1 节中的定义同构。

步骤 1 我们的映射忽略函子参数 F，因为代数效应没有内部计算。 Σ 是代数运算函子。

$$K_{\Sigma}^{\mathsf{Alg}}\ F\ A = \Sigma\ A \qquad \mathbf{data}\ K^{\mathsf{Alg}}\ \sigma\ f\ a\ \mathbf{where}\ \mathsf{Op} :: \sigma\ a \to K^{\mathsf{Alg}}\ \sigma\ f\ a$$

[4]为便于阅读，我们在示例中省略 In 和 Out。

步骤 2 $K_{\Sigma}^{\mathsf{Alg}}$ 的这个定义是一个高阶函子：

$$\begin{aligned}&\textbf{instance}\ Functor\ \sigma \Rightarrow HFunctor\ (K^{\mathsf{Alg}}\ \sigma)\ \textbf{where}\\&\quad \mathsf{hmap}\ k\ (\mathsf{Op}\ x) = \mathsf{Op}\ x\end{aligned}$$

步骤 3 我们可以证明以下同构成立（附录 C）：

$$Free\ \sigma\ a\ \cong\ Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ \sigma)\ a$$

步骤 4 代数效应的通用处理器定义为 h_{Alg}。我们在附录 C 中证明，h_{Alg} gen alg 与 fold^{Alg} gen (alg . Op) 同构。

$$\begin{aligned}&h_{\mathsf{Alg}} :: (Functor\ \sigma, Pointed\ g)\\&\qquad \Rightarrow (a \to g\ b) \to (\forall x\ .\ K^{\mathsf{Alg}}\ \sigma\ g\ (g\ x) \to g\ x) \to Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ \sigma)\ a \to g\ b\\&h_{\mathsf{Alg}} = \mathsf{fold}\end{aligned}$$

示例：状态。我们已在第 2.1 节中将状态效应引入为一个函子。状态的处理器以模块化方式用 $Free_{\mathsf{H}}$ 表示。

$$\begin{aligned}&h_{\mathsf{State}} :: Functor\ \sigma\\&\qquad \Rightarrow Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ (State\ s + \sigma))\ a \to (s \to Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ \sigma)\ (a, s))\\&h_{\mathsf{State}} = h_{\mathsf{Alg}}\ \eta\ alg_{\mathsf{Alg}}\ \textbf{where}\\&\quad alg_{\mathsf{Alg}}\ (\mathsf{Op}\ op) = (alg\ \#\ fwd)\ op\ \textbf{where}\\&\qquad alg\ \ (Get\ k)\quad = \lambda s \to k\ s\ s\\&\qquad alg\ \ (Put\ s'\ k) = \lambda_ \to k\ s'\\&\qquad fwd\ op\qquad\ \ = \lambda s \to Op_{\mathsf{H}}\ (\mathsf{Op}\ (\mathsf{fmap}\ (\$s)\ op)))\end{aligned}$$

我们定义智能构造器 get 和 put，以便使用状态效应编写程序，并重新讨论第 2.1 节中的示例。

$$\begin{aligned}&\texttt{>>>}\ h_{\mathsf{State}}\ (\textbf{do}\ x \leftarrow \mathsf{get}; \mathsf{put}\ (x+1); \mathsf{return}\ 5)\ 0\\&(5, 1)\end{aligned}$$

示例：非确定性。非确定性的处理器会将计算解释为包含所有可能结果的列表。

$$\begin{aligned}&h_{\mathsf{ND}} :: Functor\ \sigma\\&\qquad \Rightarrow Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ (Choice + \sigma))\ a \to Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ \sigma)\ [a]\\&h_{\mathsf{ND}} = h_{\mathsf{Alg}}\ \eta\ alg_{\mathsf{Alg}}\ \textbf{where}\\&\quad alg_{\mathsf{Alg}}\ (\mathsf{Op}\ op) = (alg\ \#\ fwd)\ op\ \textbf{where}\\&\qquad alg\ Fail\qquad = \mathsf{return}\ [\,]\end{aligned}$$

$$
\begin{aligned}
&alg\ (Or\ p\ q) = (+\!\!+)\ \langle\$\rangle\ p\ \langle*\rangle\ q \\
&fwd\ \ \ \ \ \ \ \ \ \ \ \ = Op_{\mathsf{H}}\ .\ \mathsf{Op}
\end{aligned}
$$

构造子 fail 和 or 支持将非确定性作为一种效应进行编程（类似于 get 和设置）。例如，请看下面的示例。

```
>>> h_ND (or (return 1) (or (or (return 2) (return 3)) fail))
[1, 2, 3]
```

4.2. 作用域效应与处理器

定义。作用域效应与处理器用于对限定某个作用域的效应进行建模，最典型的示例包括异常、带 Once 的非确定性以及局部状态变量。为了保持语法与语义之间的分离，并实现与代数效应相同的模块化，文献 [20, 31, 33] 提出使用递归数据类型 Prog 对作用域效应进行建模，该类型同时捕获代数效应与作用域效应。代数操作由函子 σ表示，而另一个函子 γ 则用于表示作用域操作。

```
data Prog σ γ a, 其中
  Var :: a                                  →Prog σ γ a
  Op  :: σ (程序 σ γ a)                     →Prog σ γ a
  Enter :: γ (Prog σ γ (Prog σ γ a)) → Prog σ γ a
```

这里，Op 对应代数效应，而 Enter 则进入一个作用域，表示作用域效应。利用余 Yoneda 引理，Enter 可以改写为一个程序，其中包含 (1) 表示作用域内程序的作用域计算，以及 (2) 位于作用域之外的延续。

$$
Enter :: \forall\, b\ c\ .\ \gamma\ b \to \underbrace{(b \to Prog\ \sigma\ \gamma\ c)}_{scoped\ computation} \to \underbrace{(c \to Prog\ \sigma\ \gamma\ a)}_{continuation} \to Prog\ \sigma\ \gamma\ a
$$

我们只聚焦于作用域效应，以便之后再将它们与代数效应组合起来。 $Free_{\mathsf{Sc}}$ 与不包含代数效应的 Prog Op 同构。

```
data Free_Sc γ a where
  Var   :: a                                  → Free_Sc γ a
  Enter :: γ (Free_Sc γ (Free_Sc γ a)) → Free_Sc γ a
```

解释。为了诠释作用域效应，Yang 等人 [33] 提出了函子代数，作为一种结构化的作用域效应处理方式，与 Pir´og 等人 [20] 采用索引代数的较繁琐方法形成对比。函子代数 [33] 由两部分组成：内代数 (Alg_E) 诠释作用域内的程序部分，而基代数 (Alg_B) 诠释延续。

$$\begin{array}{l}\textbf{data}\ Alg_E\ \gamma\ f = Alg_E\ \{ \\ \quad return_E :: \forall x\,.\,x \to f\ x, \\ \quad enter_E\ \ :: \forall x\,.\,\gamma\ (f\ (f\ x)) \to f\ x\}\end{array} \qquad \begin{array}{l}\textbf{data}\ Alg_B\ \gamma\ f\ a = Alg_B\ \{ \\ \quad enter_B :: \gamma\ (f\ a) \to a\}\end{array}$$

几乎总是，这两个代数（自同态代数和基代数）采用相同的实现，这大大简化了代码，但也不允许在作用域内外采用不同的解释。例如，在抽象求值策略时，采用不同的解释会很有用，例如深度优先搜索、广度优先搜索或深度界限搜索；后者可以建模为作用域效应 [33]。出于简化考虑，在我们的通用框架中，我们要求这两个代数相同。结构递归方案折叠$_{\mathsf{Sc}}$ 专门用于解释 $Free_{\mathsf{Sc}}$，即仅包含作用域效应的自由单子。

$$\begin{array}{l}\mathsf{fold}_{\mathsf{Sc}} :: Functor\ \gamma \\ \qquad \Rightarrow (a \to b) \to Alg_E\ \gamma\ f \to Alg_B\ \gamma\ f\ b \to Free_{\mathsf{Sc}}\ \gamma\ a \to b \\ \mathsf{fold}_{\mathsf{Sc}}\ gen\ alg_E\ alg_B\ (Var\ x) \quad = gen\ x \\ \mathsf{fold}_{\mathsf{Sc}}\ gen\ alg_E\ alg_B\ (Enter\ sc) = enter_B\ alg_B\ (\mathsf{fmap}\ endo\ sc) \\ \quad \textbf{where}\ endo \qquad\qquad = h_{cata}\,.\,\mathsf{fmap}\ (\mathsf{fold}_{\mathsf{Sc}}\ gen\ alg_E\ alg_B) \\ \qquad\quad h_{cata}\ (Var\ x) \quad = return_E\ alg_E\ x \\ \qquad\quad h_{cata}\ (Enter\ sc) = (enter_E\ alg_E\,.\,\mathsf{fmap}\ (h_{cata}\,.\,\mathsf{fmap}\ h_{cata}))\ sc\end{array}$$

通用框架。作用域效应与处理器适用于我们的通用框架。步骤 1 我们定义一个映射，使用函子 Γ 来表示作用域操作 .

$$K_\Gamma^{\mathsf{Sc}}\ F\ A = \Gamma\ (F\ A) \qquad \begin{array}{l}\textbf{data}\ K^{\mathsf{Sc}}\ \gamma\ f\ a\ \textbf{where} \\ \quad \mathsf{Enter} :: \gamma\ (f\ a) \to K^{\mathsf{Sc}}\ \gamma\ f\ a\end{array}$$

步骤 2 该映射 K_Γ^{Sc}是一个高阶函子。

$$\begin{array}{l}\textbf{instance}\ Functor\ \gamma \Rightarrow HFunctor\ (K^{\mathsf{Sc}}\ \gamma)\ \textbf{where} \\ \quad \mathsf{hmap}\ k\ (\mathsf{Enter}\ sc) = \mathsf{Enter}\ (\mathsf{fmap}\ k\ sc)\end{array}$$

步骤 3 我们称以下同构成立（附录 C）：

$$Free_{\mathsf{Sc}}\ \gamma\ \mathsf{a}\ \ \cong\ \ Free_{\mathsf{H}}\ (K^{\mathsf{Sc}}\ \gamma)\ \mathsf{a}$$

步骤 4 我们用通用递归方案来编写处理器。在附录 C 中，我们证明 $\mathsf{fold}_{\mathsf{Sc}}$ gen alg alg 与 h_{Sc} gen (λ(Enter sc) $\to enter_E$ alg sc) 同构，其中 $return_E$ alg $= \eta$。注意，我们对内代数和基代数使用相同的实现。

$$\begin{aligned}
&h_{\mathsf{Sc}} :: (Functor\ \gamma, Pointed\ g)\\
&\quad \Rightarrow (a \to g\ b) \to (\forall x\,.\,K^{\mathsf{Sc}}\ \gamma\ g\ (g\ x) \to g\ x) \to Free_{\mathsf{H}}\ (K^{\mathsf{Sc}}\ \gamma)\ a \to g\ b\\
&h_{\mathsf{Sc}} = \mathsf{fold}
\end{aligned}$$

我们使用代数效应和作用域效应的高阶函子的余积来重构 Prog 数据类型，并定义相应的处理器。

$$\begin{aligned}
&Prog\ \sigma\ \gamma\ a\ \ \cong\ \ Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ \sigma + K^{\mathsf{Sc}}\ \gamma)\ a\\
&h_{\mathsf{Prog}} :: (Functor\ \sigma, Functor\ \gamma, Pointed\ g)\\
&\quad \Rightarrow (a \to g\ b) \to (\forall x\,.\,(K^{\mathsf{Alg}}\ \sigma \oplus K^{\mathsf{Sc}}\ \gamma)\ g\ (g\ x) \to g\ x)\\
&\quad \to Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ \sigma \oplus K^{\mathsf{Sc}}\ \gamma)\ a \to g\ b\\
&h_{\mathsf{Prog}} = \mathsf{fold}
\end{aligned}$$

示例：带 Once 的非确定性。为展示这种方法，我们使用带 once 的非确定性，其中 once 是一种只返回非确定性程序第一个结果的操作。我们区分非确定性代数操作 Fail 和 Or，以及作用域操作 Once。

$$\mathsf{data\ Once\ a\ \ =Once\ a}$$

同样，我们用一个保留所有已找到结果的列表来解释非确定性程序。带 Once 的非确定性处理器同时包含代数操作（Or 和 Fail）与作用域操作（Once）。

$$\begin{aligned}
&h_{\mathsf{Once}} :: (Functor\ \sigma, Functor\ \gamma)\\
&\quad \Rightarrow Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ (Choice + \sigma) \oplus K^{\mathsf{Sc}}\ (Once + \gamma))\ a\\
&\quad \to Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ \sigma \oplus K^{\mathsf{Sc}}\ \gamma)\ [a]\\
&h_{\mathsf{Once}} = \mathsf{fold}\ gen\ (alg_{\mathsf{Alg}} \mathbin{⊕} alg_{\mathsf{Sc}})\ \mathbf{where}\\
&\quad gen\quad x\qquad\qquad = \mathsf{return}\ [x]\\
&\quad alg_{\mathsf{Alg}}\ (\mathsf{Op}\ op)\quad = (alg_{\mathsf{Choice}}\ \#\ fwd_{\mathsf{Choice}})\ op\ \mathbf{where}\ ...\\
&\quad alg_{\mathsf{Sc}}\ \ (\mathsf{Enter}\ sc) = (alg_{\mathsf{Once}}\ \ \#\ fwd_{\mathsf{Once}}\ )\ sc\ \mathbf{where}\\
&\qquad alg_{\mathsf{Once}}\quad (Once\ y) = \mathsf{join}\ (\mathsf{fmap\ head}\ y)\\
&\qquad fwd_{\mathsf{Once}}\qquad\qquad = Op_{\mathsf{H}}\,.\,\mathsf{Enter}\,.\,\mathsf{fmap}\ (\mathsf{fmap\ lift}_{\mathsf{Once}})\\
&\qquad\quad \mathbf{where}\ \mathsf{lift}_{\mathsf{Once}} = foldr\ (\lambda x\ xs \to (+\!\!+)\ \langle\$\rangle\ x\ \langle*\rangle\ xs)\ (\mathsf{return}\ [\,])
\end{aligned}$$

我们按惯例定义智能构造器 fail、or 和 once，以便编写非确定性程序。例如，比较下面这个包含作用域操作 once 的程序与不含 once 的等价程序：前者会在找到第一个结果后继续执行程序的其余部分。

```
>>> h_Once ( e (or (return 1) (return
5)) >>=λx → or (return x) (return (x + 1)))
[1,2]>>> h_Once (or (return 1) (return 5) >>=
λx → or (return x) (return (x + 1)))
[1,2,5,6]
```

4.3. 并行效应与处理器

定义。一般而言，代数效应是按顺序执行的。不过，在某些情况下，例如出于性能考虑，可能需要并行执行效应。Xie 等人 [32] 定义了并行效应与处理器，并给出了同时支持代数效应和并行效应的操作语义。具体来说，他们通过 for 关键字迭代可并行化计算，以此表示并行效应。他们的工作还提供了 Dex [18] 和 Haskell 实现。他们使用 $Free_{\mathsf{Par}}\ \rho$ a 表示并行效应，其中 ρ 是用于表示并行操作的可迭代函子[5]。

$$
\begin{aligned}
&\textbf{data}\ Free_{\mathsf{Par}}\ \rho\ a\ \textbf{where}\\
&\quad Var :: a \rightarrow Free_{\mathsf{Par}}\ \rho\ a\\
&\quad For :: \rho\ (Free_{\mathsf{Par}}\ \rho\ b) \rightarrow (\rho\ b \rightarrow Free_{\mathsf{Par}}\ \rho\ a) \rightarrow Free_{\mathsf{Par}}\ \rho\ a
\end{aligned}
$$

这里， Var 是纯计算，而 For 表示可并行化的计算。For 接受两个参数：一个由计算组成的可迭代结构（由函子 ρ表示）和一个延续。

Xie 等人 [32] 并未证明这种并行效应表示是一个自由单子。

通用框架。并行效应可以用我们的框架来表达。

步骤 1 我们定义一个用于映射效应签名的高阶函子 $K_{\mathrm{P}}^{\mathsf{Par}}$ ：

$$K_{\mathrm{P}}^{\mathsf{Par}}\ F\ A = \mathrm{P}\ (F\ B) \times (\mathrm{P}\ B \Rightarrow A)$$

[5]这与 Xie 等人 [32] 的实现有两点不同：它省略了代数效应，并使用通用函子 ρ 而非列表来遍历计算。

$$\textbf{data}\ K^{\mathsf{Par}}\ \rho\ f\ a\ \textbf{where}$$
$$\quad \mathsf{For} :: \rho\ (f\ b) \to (\rho\ b \to a) \to K^{\mathsf{Par}}\ \rho\ f\ a$$

步骤 2 的确，$K_{\mathrm{P}}^{\mathsf{Par}}$是一个高阶函子：

$$\textbf{instance}\ Functor\ \rho \Rightarrow HFunctor\ (K^{\mathsf{Par}}\ \rho)\ \textbf{where}$$
$$\quad \mathsf{hmap}\ k\ (\mathsf{For}\ iters\ c) = \mathsf{For}\ (\mathsf{fmap}\ k\ iters)\ c$$

步骤 3 现在，以下同构成立（附录 C）：

$$Free_{\mathsf{Par}}\ \rho\ a\ \cong\ Free_{\mathsf{H}}\ (K^{\mathsf{Par}}\ \rho)\ a$$

这意味着 $Free_{\mathsf{Par}}$ 也是一个自由单子。
步骤 4 现在可以通用地定义并行效应的处理器。在附录 C 中，我们证明 [32]的处理器（在我们调整后的版本中）与 h_{Par}gen (λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) 同构。

$$h_{\mathsf{Par}} :: (Functor\ \rho, Pointed\ g)$$
$$\quad \Rightarrow (a \to g\ b) \to (\forall x\,.\,K^{\mathsf{Par}}\ \rho\ g\ (g\ x) \to g\ x) \to Free_{\mathsf{H}}\ (K^{\mathsf{Par}}\ \rho)\ a \to g\ b$$
$$h_{\mathsf{Par}} = \mathsf{fold}$$

示例：并行累积。我们重新讨论 Xie 等人提出的示例 [32]，该示例模仿了 Dex 的累积效应；它类似于状态，但只能递增，并且隐式地以该递增操作的单位元初始化。累积由代数操作 Accum m 表示，其中 m 是一个幺半群[6]。

$$\textbf{data}\ Accum\ m\ a = Accum\ m\ a$$

用于并行累积的处理器具有代数效应和并行效应。

$$h_{\mathsf{Accum}} :: (Monoid\ m, Functor\ \sigma)$$
$$\quad \Rightarrow Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ (Accum\ m + \sigma) \oplus K^{\mathsf{Par}}\ [\,]\ a$$
$$\quad \to Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ \sigma)\ (m, a)$$
$$h_{\mathsf{Accum}} = \mathsf{fold}\ gen\ (alg_{\mathsf{Alg}} \circledast alg_{\mathsf{Par}})\ \textbf{where}$$
$$\quad gen\quad x \qquad\qquad = \mathsf{return}\ (\epsilon, x)$$
$$\quad alg_{\mathsf{Alg}}\ (\mathsf{Op}\ op) \qquad = (alg_{\mathsf{Accum}}\ \#\ fwd_{\mathsf{Accum}})\ op\ \textbf{where}$$
$$\quad\quad alg_{\mathsf{Accum}}\ (Accum\ m'\ k) = \mathsf{fmap}\ (\lambda(m, x) \to (m' \diamond m, x))\ k$$
$$\quad\quad fwd_{\mathsf{Accum}} \qquad\qquad = Op_{\mathsf{H}}\,.\,\mathsf{Op}$$

[6]幺半群定义为一个集合，配备一个结合二元运算 $\diamond$和一个单位元 ϵ。

alg_{Par} (For iters k) = 执行 (ms, xs) ← fmap unzip (sequence iters) 令 append (m, x) =(foldr ($\diamond$) m ms, x) 其中 fmap append (k xs)

我们定义 accum 和 for 构造器，分别用于累积计算和遍历计算。考虑以下示例，它计算一个整数列表的总和。我们使用 Sum 幺半群作为累加器。

```
>>> h_Accum (for (fmap (accum . Sum) [1, 2, 10, 4]))
17
```

4.4. Writer 效应与处理器

为了定义写入器效应与处理器，我们以 Writer 单子为例。Writer 单子会同时跟踪返回值和输出消息，例如日志文件中的内容。根据 MonadWriter 库[7]，，Writer 效应的最小完整定义需要三个操作：tell、listen 和 pass。第一个是代数操作，用于生成类型为 w 的输出消息。

```
tell :: w → m ()
```

listen 和 pass 都不符合此前定义的任何效应。我们重新表述它们的定义，使其适用于我们的通用框架。listen 执行一项计算，并返回由结果值和输出消息组成的元组。它用于检查子计算向输出中写入了什么。

$$
\begin{aligned}
& \mathsf{listen} :: m\ a \to m\ (a, w) \\
\cong\ & \mathsf{listen} :: m\ a \to ((a, w) \to m\ b) \to m\ b \\
\cong\ & \mathsf{listen} :: m\ (w \to m\ b) \to m\ b \\
\cong\ & \mathsf{listen} :: m\ (\varphi_{\mathsf{listen}}\ (m\ b)) \to m\ b \quad \text{with } \varphi_{\mathsf{listen}}\ w = ((\to)\ w)
\end{aligned}
$$

此外，pass 会执行一项计算，得到一个值和一个函数，随后将该函数应用于输出消息。它用于修改写入输出的内容。

$$
\begin{aligned}
& \mathsf{pass} :: m\ (a, w \to w) \to m\ a \\
\cong\ & \mathsf{pass} :: m\ (\varphi_{\mathsf{pass}}\ a) \to m\ a \quad \text{with } \varphi_{\mathsf{pass}}\ w = ((,)\ (w \to w))
\end{aligned}
$$

现在，我们对 listen 和 pass 的这两个定义进行抽象，定义一种新颖的效应，并证明它们是我们的通用框架的一个特例。

[7] https://hackage.haskell.org/package/mtl-2.2.2/docs/Control-Monad-Writer-Class.html

定义。我们将这些效应称为写入器效应，并记作 $Free_{\mathsf{Write}}\ \varphi\ \mathsf{a}$，其中 φ 是用于 Writer 效应操作的函子。

$$
\begin{aligned}
&\text{data } Free_{\mathsf{Write}}\ \varphi\ \mathsf{a} \text{ where}\\
&\quad Var :: \mathsf{a} \to Free_{\mathsf{Write}}\ \varphi\ \mathsf{a}\\
&\quad \mathsf{Exec} :: Free_{\mathsf{Write}}\ \varphi\ (\varphi\ (Free_{\mathsf{Write}}\ \varphi\ \mathsf{a})) \to Free_{\mathsf{Write}}\ \varphi\ \mathsf{a}
\end{aligned}
$$

这里，Var 表示纯计算，而 Exec 是 Writer 计算。事实上，可以（使用余 Yoneda 引理）重写 Exec，使其由一个 Writer 操作、一个由某个函子 φ 修饰的结果，以及一个延续组成。

$$
Exec :: \forall\, b\,.\ \underbrace{Free_{\mathsf{Write}}\ \varphi\ (\varphi\ b)}_{\text{writer } computation} \to \underbrace{(b \to Free_{\mathsf{Write}}\ \varphi\ a)}_{continuation} \to Free_{\mathsf{Write}}\ \varphi\ a
$$

特别是，对于 listen，这个函子 φ 是 ((→) w)，因为 Writer 操作的结果类型是 w → m a。类似地，pass 由函子 ((,) (w → w)) 修饰，因为其内部作用的结果类型是 (a, w → w)。

通用框架。我们展示写入器效应符合我们的通用框架。

步骤 1 我们选择一个恰当的映射 $K_{\Phi}^{\mathsf{Write}}$，以根据我们的理论模型来表达写入器效应。请注意，这与作用域效应与处理器类似，只是函子的顺序相反。

$$
K_{\Phi}^{\mathsf{Write}}\ F\ A = F\ (\Phi\ A)
\qquad\qquad
\begin{aligned}
&\mathbf{data}\ K^{\mathsf{Write}}\ \varphi\ f\ a\ \mathbf{where}\\
&\quad \mathsf{Exec} :: f\ (\varphi\ a) \to K^{\mathsf{Write}}\ \varphi\ f\ a
\end{aligned}
$$

步骤 2 $K_{\Phi}^{\mathsf{Write}}$是一个高阶函子：

$$
\begin{aligned}
&\mathbf{instance}\ Functor\ \varphi \Rightarrow HFunctor\ (K^{\mathsf{Write}}\ \varphi)\ \mathbf{where}\\
&\quad \mathsf{hmap}\ k\ (\mathsf{Exec}\ x) = \mathsf{Exec}\ (k\ x)
\end{aligned}
$$

步骤 3 此外，以下同构成立（附录 C）：

$$
Free_{\mathsf{Write}}\ \varphi\ a \ \cong\ Free_{\mathsf{H}}\ (K^{\mathsf{Write}}\ \varphi)\ a
$$

这意味着 $Free_{\mathsf{Write}}$ 也是一个自由单子。
步骤 4 现在可以根据我们的框架定义一个用于写入器效应的处理器：

$$
\begin{aligned}
&h_{\mathsf{Wr}} :: (Functor\ \varphi, Pointed\ g)\\
&\qquad \Rightarrow (a \to g\ b) \to (\forall x\,.\, K^{\mathsf{Write}}\ \varphi\ g\ (g\ x) \to g\ x) \to Free_{\mathsf{H}}\ (K^{\mathsf{Write}}\ \varphi)\ a \to g\ b\\
&h_{\mathsf{Wr}} = \mathsf{fold}
\end{aligned}
$$

为这个处理器编写同构没有意义，因为不存在针对 $Free_{\mathsf{Write}}$ 的专用处理器。

示例：重置日志。Writer 单子同时使用代数效应和写入器效应。Tell 用于构造输出消息，其中 w 是一个用于跟踪输出消息的幺半群。

```
数据 Tell w a  =Tell w a
```

Listen 可以查看某个（子）计算写入输出的内容。Pass 能够调整写入输出的消息。这两个操作分别由与 listen 和 pass 的参数类型相对应的函子表示。

$$\textbf{type}\ Listen\ w = ((\to)\ w) \qquad\qquad \textbf{type}\ Pass\ w = ((,)\ (w \to w))$$

写入操作的处理器是基于我们的通用框架定义的：

$$\begin{aligned}
&h_{\mathsf{Write}} :: (Functor\ \sigma, Monoid\ w)\\
&\quad \Rightarrow Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ (Tell\ w + \sigma) \oplus K^{\mathsf{Write}}\ (Listen\ w + Pass\ w + \varphi))\ a\\
&\quad \to Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ \sigma \oplus K^{\mathsf{Write}}\ \varphi)\ (a, w)\\
&h_{\mathsf{Write}} = \mathsf{fold}\ gen\ (alg_{\mathsf{Alg}} \mathbin{⊕} alg_{\mathsf{Write}})\ \textbf{where}\\
&\quad gen\ x = \mathsf{return}\ (x, \epsilon)\\
&\quad alg_{\mathsf{Alg}}\ (\mathsf{Op}\ op) = (alg_{\mathsf{Tell}}\ \#\ fwd_{\mathsf{Tell}})\ op\ \textbf{where}\\
&\quad\quad alg_{\mathsf{Tell}}\ (Tell\ w\ k) = \textbf{do}\ (x, w') \leftarrow k; \mathsf{return}\ (x, w \diamond w')\\
&\quad\quad fwd_{\mathsf{Tell}} = Op_{\mathsf{H}}\ .\ \mathsf{Op}\\
&\quad alg_{\mathsf{Write}}\ (\mathsf{Exec}\ k) = k \gg\!= \lambda\textbf{case}\\
&\quad\quad (f, w) \to f\ w\\
&\quad\quad ((f, mx), _) \to \mathsf{fmap}\ (\mathsf{fmap}\ f)\ mx\\
&\quad\quad (op, _) \to Op_{\mathsf{H}}\ (\mathsf{Exec}\ (\mathsf{return}\ op))
\end{aligned}$$

我们定义 tell、listen 和 pass 构造子，以定义函数 reset，该函数会重置日志，使其清空。

$$\begin{aligned}
&reset :: (Functor\ \sigma, Functor\ \varphi, Monoid\ w)\\
&\quad \Rightarrow Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ (Tell\ w + \sigma) \oplus K^{\mathsf{Write}}\ (Listen\ w + (Pass\ w + \varphi)))\ ()\\
&reset = \mathsf{pass}\ (\mathsf{return}\ ((), const\ \epsilon))
\end{aligned}$$

例如，考虑以下程序：它先记录日志 `"pre"`，然后重置日志，接着记录日志 `"post"`。

$$\begin{aligned}
&\texttt{>>>}\ h_{\mathsf{Write}}\ (\mathsf{tell}\ \texttt{"post"} \gg reset \gg \mathsf{tell}\ \texttt{"pre"})\\
&((), \texttt{"post"})
\end{aligned}$$

或者，也可以用 censor 来定义这个 reset 函数。censor 是 MonadWriter 库中的一个派生方法，它接收一个函数 w → w 和一个计算 m a，并在计算完成后修改 writer，同时保持返回值不变。censor 既可以定义为作用域效应，也可以视为 pass 的特例（附录 B）。

4.5. 潜伏效应与处理器

定义。潜伏效应与处理器 [1] 指的是这样一类效应：其控制流会将某些计算推迟到程序解释的后续阶段才进行求值。潜伏效应的示例包括函数体带有效应的函数抽象、按名调用或按需调用等求值策略，以及分阶段计算。潜伏效应表示为 $Free_{\mathsf{Lat}}\ \zeta\ \ell$ a，其中ζ 是常规效应函子，而 ℓ 是潜在效应函子，表示被推迟的效应[8]。

$$
\begin{aligned}
&\mathbf{data}\ Free_{\mathsf{Lat}}\ \zeta\ \ell\ a\ \mathbf{where}\\
&\quad Leaf\ ::\ a \to Free_{\mathsf{Lat}}\ \zeta\ \ell\ a\\
&\quad Node :: \zeta\ p\ c \to \ell\ () \to (\forall x\ .\ c\ x \to \ell\ () \to Free_{\mathsf{Lat}}\ \zeta\ \ell\ (\ell\ x))\\
&\qquad\quad \to (\ell\ p \to Free_{\mathsf{Lat}}\ \zeta\ \ell\ a) \to Free_{\mathsf{Lat}}\ \zeta\ \ell\ a
\end{aligned}
$$

这里，Leaf 是纯计算，而 Node 是一个内部节点，其中包含：(1) 一个操作 ζ p c，其结果类型为 p，并包含若干子计算 c；(2) 节点中的效应状态 ℓ ()；(3) 一个函数 (∀ x . c x $\to$ ℓ () $\to Free_{\mathsf{Lat}}\ \zeta\ \ell$ (ℓ x))，用于在给定当前效应状态的情况下，解释结果类型为 x 的子计算 c；以及 (4) 一个延续 (ℓ p $\to$ $Free_{\mathsf{Lat}}\ \zeta\ \ell$ a)，用于解释程序的剩余部分。

van den Berg 等人 [1] 没有证明 $Free_{\mathsf{Lat}}\ \zeta\ \ell$ a 是一个自由单子。

通用框架。潜伏效应是我们的通用自由单子的一个实例。步骤 1 我们选择一种映射[9] $K^{\mathsf{Lat}}_{\zeta,\mathcal{L}}$，用以表示具有效应函子 ζ 和潜在效应函子 $\mathcal{L}$的潜伏效应与处理器。它包含一个 Node 构造器来表示内部节点，类似于 $Free_{\mathsf{Lat}}$的 Node 构造器。

$$
\begin{aligned}
&K^{\mathsf{Lat}}_{\zeta,\mathcal{L}}\ F\ A = \int^{P,C} \zeta\ P\ C \times \mathcal{L}\ 1 \times (\int_X C\ X \times \mathcal{L}\ 1 \Rightarrow F\ (\mathcal{L}\ X)) \times (\mathcal{L}\ P \Rightarrow A)\\
&\mathbf{data}\ K^{\mathsf{Lat}}\ \zeta\ \ell\ f\ a\ \mathbf{where}\\
&\quad \mathsf{Node} :: \zeta\ p\ c \to \ell\ () \to (\forall x\ .\ c\ x \to \ell\ () \to f\ (\ell\ x))\\
&\qquad\quad \to (\ell\ p \to a) \to K^{\mathsf{Lat}}\ \zeta\ \ell\ f\ a
\end{aligned}
$$

步骤 2 它对应一个高阶函子实例。

实例 $HFunctor$ ($K^{\mathsf{Lat}}\ \zeta\ \ell$) where hmap k (Node sub ℓ st c) = Node sub ℓ (fmap k . st) c

步骤 3 由此可得以下同构（附录 C）：

[8]这个数据类型等价于 van den Berg 等人的 Tree 数据类型 [1]。[9]使用类型代数中的 ends 和 coends，它们分别对应 Haskell 中的全称量化和存在量化。

$$Free_{\mathsf{Lat}}\ \zeta\ \ell\ a\ \cong\ Free_{\mathsf{H}}\ (K^{\mathsf{Lat}}\ \zeta\ \ell)\ a$$

这意味着 $Free_{\mathsf{Lat}}$ 也是一个自由单子。
步骤 4 我们现在可以为潜伏效应编写一个通用处理器。

$$h_{\mathsf{Lat}} :: Pointed\ g \Rightarrow (a \to g\ b) \to (\forall x\,.\,K^{\mathsf{Lat}}\ \zeta\ \ell\ g\ (g\ x) \to g\ x) \to Free_{\mathsf{H}}\ (K^{\mathsf{Lat}}\ \zeta\ \ell)\ a \to g\ b$$
$$h_{\mathsf{Lat}} = \mathsf{fold}$$

为此处理器编写同构并没有意义，因为文献中不存在针对 $Free_{\mathsf{Lat}}$ 的专用处理器。

示例：带记忆化的惰性求值。我们通过记忆化实现按需调用和按值调用。此示例改编自 van den Berg 等人的惰性求值示例。[1]考虑以下程序：

$$prog_{\mathsf{Lazy}} = \mathsf{app}\ (\mathsf{abs}\ (\mathsf{return}\ 3))\ (\mathbf{do}\ \mathsf{put}\ 42; \mathsf{return}\ 5)$$

它将一个常量函数应用于一个会改变程序状态的参数。在按需调用的求值策略下，我们预计状态保持不变，因为该参数永远不会被需要，因而也永远不会被执行。在急切策略下，我们预计程序的状态为 42。

我们需要代数效应、作用域效应和潜伏效应来对此示例建模。首先，代数状态效应（第 4.1 节）可以获取状态并设置新状态。其次，我们需要一个环境来了解哪些变量处于作用域内。我们使用作用域读取器效应对此功能进行建模，该效应包含两个操作：询问（ask）是代数操作，用于获取当前环境；局部（local）是作用域操作，用于在修改后的环境中执行计算之前修改环境。有关 ask 和 local 的更多细节，请参阅 Haskell 的 MonadReader[10] 库。最后，用于带记忆化的惰性求值的潜伏效应包含两个操作：thunk 用于延迟某个操作的求值，force 用于强制其进行求值。

data Thunk 化 v :: $* \to (* \to *) \to *$
where Thunk :: Thunk 化 v Ptr (OneSub
v) Force :: Ptr $\to$ Thunk 化 v v NoSub
data NoSub :: $* \to *$where data OneSub
v :: $* \to *$where One :: OneSub v v

[10]https://hackage.haskell.org/package/mtl-2.3/docs/Control-Monad-Reader.htm 1

这里，Ptr 是指向环境的指针（整数），该环境用于跟踪 thunk 和记忆化值。NoSub 和 OneSub 分别表示 Thunk 和 Force 没有或有一个潜在子计算。

为简洁起见，我们省略未知效应的转发，并定义一种表达式数据类型，其中包含上述代数效应、有作用域效应和潜伏效应。

$$\begin{aligned}\textbf{type}\ Expr\ v\ a = Free_{\mathsf{H}}\ \ (\ & K^{\mathsf{Alg}}\ (State\ v + Ask\ [v]) \\ \oplus\ & K^{\mathsf{Sc}}\ \ (Local\ [v]) \\ \oplus\ & K^{\mathsf{Lat}}\ (Thunking\ v)\ Id)\ a\end{aligned}$$

我们的处理器的语义域是 $State_L$ s l a，其中在本例中，状态 s 由一个元组 (v, [Thunk v]) 构成，而 l 是恒等函子。

$$\textbf{newtype}\ State_L\ s\ l\ a = State_L\ \{unStateL :: (s, l\ a)\}$$

惰性求值的处理器包含三个环境：(1) 类型为 v 的状态；(2) 类型为 [v] 的求值环境，用于确定哪些变量处于作用域内；以及 (3) 类型为 [Thunk v] 的 thunk 和记忆化值环境。Thunk 是一种未求值操作。

$$\begin{aligned}\textbf{type}\ Thunk\ v = Either\ & (Id\ () \to v \to [v] \to [Thunk\ v] \\ & \to State_L\ (v, [Thunk\ v])\ Id\ v)\ v\end{aligned}$$

环境 (1) 和 (3) 是解释后的表达式结果的一部分。有了这些，我们便可以定义 h_{Lazy}，它会对有状态程序 [11] 进行惰性解释。

h_{Lazy} :: 表达式 v a → v → [v] → [Thunk v] → $State_L$ (v, [Thunk v]) Id a
h_{Lazy} 程序 s nv th = 折叠 生成 ($alg_{\mathsf{Alg}} \oplus alg_{\mathsf{Sc}} \oplus alg_{\mathsf{Lat}}$) 程序，其中

$$\begin{aligned}&gen\ \ x = State_L\ ((s, th), Id\ x) \\ &alg_{\mathsf{Alg}}\ (\mathsf{Op}\ op) = (alg_{\mathsf{St}}\ \#\ alg_{\mathsf{Ask}})\ op\ \textbf{where}\ ... \\ &alg_{\mathsf{Sc}}\ (\mathsf{Enter}\ sc) = alg_{\mathsf{Local}}\ sc\ \textbf{where}\ ... \\ &alg_{\mathsf{Lat}}\ (\mathsf{Node}\ Thunk\ l\ st\ k) = k\ (\mathsf{length}\ th <\$\ l)\ s\ nv \\ &\qquad\qquad (th +\!\!+ [Left\ (st\ One)]) \\ &alg_{\mathsf{Lat}}\ (\mathsf{Node}\ (Force\ p)\ l\ st\ k) = \textbf{case}\ (th\ !!\ p)\ \textbf{of} \\ &\quad Left\ t \to \textbf{let}\ State_L\ ((s', th'), Id\ lv) = (t\ l)\ s\ nv\ th \\ &\qquad \textbf{in}\ unId\ \$\ \mathsf{fmap}\ (\lambda v \to (k\ lv)\ s'\ nv\ (replace\ p\ (Right\ v)\ th'))\ lv \\ &\quad Right\ v \to (k\ (v <\$\ l))\ s\ nv\ th\end{aligned}$$

代数效应和作用域效应的代数并没有什么令人意外之处。对操作进行 thunk 化时，我们会调用延续，并将当前操作添加

[11] 我们使用一个紧凑的数据类型 data Id a = Id {unId :: a } 来表示恒等函子。

到 thunk 列表中。强制执行操作时，我们会从 thunk 列表中获取 thunk 或值。如果获取到的是 thunk，就对其求值，然后用其值替换 thunk 列表中的对应项，以便进行记忆化，并调用延续。如果获取到的是值，就立即将该值传给延续并调用它。

急切处理器对代数效应和作用域效应具有相同的类型签名和实现，但会交换 Thunk 和 Force 的行为：

```
h_Eager prog s nv th = ...where
  alg_Lat (Node Thunk l st k) =
    let State_L ((s', th'), lv) = (st One l) s nv th
    in unId $
       fmap (λv → k (length th' <$ l) s' nv (th' ++ [Right v])) (unId lv)
  alg_Lat (Node (Force p) l st k) = case (th !! p) of
    Right v → k (v <$ l) s nv th
```

我们定义构造器 get、put、ask、local、thunk 和 force，并回到示例程序，用这些构造器定义标准的 λ-演算功能。值可以是整数，也可以是函数抽象：

```
data V = Val Int | Abs (Expr V V)
```

我们如下定义变量、函数抽象和函数应用。我们将函数应用的参数 thunk 化，并对其求值

```
var :: Ptr → Expr V V
var x = do nv ← ask; local ([nv !! x]) (force 0)
abs :: Expr V V → Expr V V
abs body = return (Abs body)
app :: Expr V V → Expr V V → Expr V V
app e1 e2 = do vf ← e1; nv ← ask; p ← thunk e2
               case vf of Abs body → local ([nv !! p]) body
```

现在，我们对上述程序 prog 进行惰性求值和急切求值。初始状态为 0，环境为空，thunk 列表也为空。

```
>>> h_Lazy prog_Lazy (Val 0) [] []
(0, [Left "thunk"], 3)

>>> h_Eager prog_Lazy (Val 0) [] []
(42, [Right 5], 3)
```

三元组的第一个元素是当前状态，第二个元素是 thunk 或记忆化值的列表，第三个元素是程序结果。

4.6. 括号效应与处理器

考虑这样一种情况：在使用某个资源（例如文件）时，引发了一个（内置）异常。在这种情况下，应该如何处理并不总是显而易见，但无论如何，我们都希望避免代码无法访问该资源。函数 Bracket [15] 获取资源，对其进行操作，然后安全地释放资源。即使发生异常，Bracket 也会正确释放资源并重新引发异常。这体现在它的类型中：m r 获取资源 r，函数 r → m b 释放资源，函数 r → m a 使用资源，而 m a 是结果。

$$
\begin{aligned}
& \mathsf{bracket} :: m\ r \to (r \to m\ b) \to (r \to m\ a) \to m\ a \\
\cong\ & \mathsf{bracket} :: m\ r \to (r \to (m\ b, m\ a)) \to m\ a \\
\cong\ & \mathsf{bracket} :: m\ (m\ b, m\ a) \to m\ a
\end{aligned}
$$

通常，b 是单元类型 ()[12]，而单子 m 是 IO 单子。例如，考虑程序 firstTwo：它打开一个文件 `"foo.txt"`，读取并打印其中的前两个字符。如果发生异常，则释放文件（通过打印 `"released"` 表示）。

```
firstTwo =bracket (openFile "foo.txt" ReadMode) (λ_ →print
"released") (λh →do x ←读取字符 h; y ←读取字符 h; print (x , y))
```

下面的程序会打印 `"foo.txt"` 的内容，然后执行 firstTwo。

```
>>> 读取文件 "foo.txt" ≫= print ≫firstTwo"HELLO, WORLD!" "H"
('H', 'E')"released"
"released" "***Exception: foo.txt hGetChar end of file"
```

有两种可能的情况：要么 `"foo.txt"` 中包含足够多的字符（左图），要么由于无法再读取更多字符而发生异常（右图）。注意，在这两种情况下都会释放资源；但如果发生错误，则会在释放资源后重新抛出异常。

定义。bracket 不符合前述任何高阶效应。因此，我们定义 $Free_{\mathsf{Res}}$ a 来表示括号效应与处理器。

$$
\begin{aligned}
&\mathbf{data}\ Free_{\mathsf{Res}}\ a\ \mathbf{where} \\
&\quad Var\ \ :: a \to Free_{\mathsf{Res}}\ a \\
&\quad Bracket :: Free_{\mathsf{Res}}\ (Free_{\mathsf{Res}}\ (), Free_{\mathsf{Res}}\ a) \to Free_{\mathsf{Res}}\ a
\end{aligned}
$$

[12]接下来，我们将用 () 替换此类型 b。

此处， Var 表示纯计算，而 Bracket 的结构与上文改写后的 bracket 相同。

通用框架。我们展示括号效应符合我们的框架。步骤 1 我们选择适当的映射 K^{Res} 来表示括号。

$$K^{\mathsf{Res}}\ F\ A = F\ (F\ (), F\ A) \qquad \begin{array}{l}\mathbf{data}\ K^{\mathsf{Res}}\ f\ a\ \mathbf{where}\\ \quad \mathsf{Bracket} :: f\ (f\ (), f\ a) \to K^{\mathsf{Res}}\ f\ a\end{array}$$

步骤 **2** K^{Res} 是一个高阶函子：

instance $HFunctor\ K^{\mathsf{Res}}$ where hmap k (Bracket res) =Bracket (k (f map (λ(rel, use) $\to$(k rel , k use)) res))

步骤 3 此外，以下同构成立（附录 C）：

$$Free_{\mathsf{Res}}\ a \ \cong\ Free_{\mathsf{H}}\ K^{\mathsf{Res}}\ a$$

步骤 4 可以根据我们的框架定义括号效应的处理器：

$$\begin{array}{l}h_{\mathsf{Res}} :: (Pointed\ g)\\ \qquad \Rightarrow (a \to g\ b) \to (\forall x\,.\,K^{\mathsf{Res}}\ g\ (g\ x) \to g\ x) \to Free_{\mathsf{H}}\ K^{\mathsf{Res}}\ a \to g\ b\\ h_{\mathsf{Res}} = \mathsf{fold}\end{array}$$

为此处理器编写同构没有意义，因为不存在针对 $Free_{\mathsf{Res}}$ 的专用处理器。

示例：打印文件的前两个字符。我们重新讨论上面的示例，该示例会在可能的情况下打印文件的前两个字符。要用我们的框架对这个示例建模，我们需要用于打开（OpenFile）和读取文件（ReadFile）、从文件中读取字符（HGetChar ）以及将字符打印到标准输出（Print）的代数效应。此外，我们还需要括号效应来编码括号行为，以便安全地释放资源。

$$\begin{array}{lllll}\mathbf{data}\ Teletype\ a = & HGetChar & Handle & (Char \to a)\\ \quad \mid & Print & String & a\\ \quad \mid & ReadFile & FilePath & (String \to a)\\ \quad \mid & OpenFile & FilePath\ IOMode & (Handle \to a)\end{array}$$

括号处理器是用我们的通用框架定义的：

$$\begin{array}{l}h_{\mathsf{Bracket}} :: Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ Teletype \oplus K^{\mathsf{Res}})\ a \to \mathsf{IO}\ a\\ h_{\mathsf{Bracket}} = \mathsf{fold}\ \eta\ (alg_{\mathsf{Tele}} \mathbin{\#} alg_{\mathsf{Res}})\ \mathbf{where}\end{array}$$

alg_{Tele} (Op (HGetChar h k)) $=$ 读取字符 h $\gg=$ kalg_{Tele} (Op (Print s k)) $=$ print s $\gg$ kalg_{Tele} (Op (ReadFile fp k)) $=$ 读取文件 fp $\gg=$ k alg_{Tele} (Op (OpenFile fp mode k)) $=$ 打开文件 fp mode $\gg=$ kalg_{Res} (Bracket res) $=$ 执行 (rel , use) $\leftarrow$ res; bracket (return ()) (const rel) (const (join use))

我们定义构造子读取字符、打印、资源管理括号、读取文件和打开文件，以重新审视这个示例。

```
>>> h_Bracket (读取文件 "foo.txt" >>= 打印 >>firstTwo)"HELLO,
 WORLD!"   "H"'('H','E')"       "released"
"released"        "***Exception: foo.txt hGetChar end of file"
```

5. 自由单子的形式化

接下来，我们将介绍通用自由单子的范畴论基础。对于不熟悉范畴、伴随和自由结构的读者，请参阅附录 A。

概括而言，我们的框架基于 Free $\circ$ (Id $\rtimes \overline{-}$) $\dashv$ App $\circ$ U 伴随，它由两个较简单的伴随 Free $\dashv$U 和 Id $\rtimes - \dashv$ App 复合而成。这个复合伴随导出了我们的单子 $Free_{\mathsf{H}}$ in$\mathbb{C}$。

$$H-Alg(\mathbb{P} \rtimes \mathbb{C}) \underset{\overline{U}}{\overset{\overline{Free}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{P} \rtimes \mathbb{C} \underset{App}{\overset{Id \rtimes -}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{C}$$

我们希望得到一个表示高阶效应的 $\mathbb{C}$ 中的单子。然而，我们的操作签名不再是 $\mathbb{C}$ 中的自函子（如代数效应中的情况），而是高阶自函子。因此，在构造自由-遗忘伴随之前，必须先从 $\mathbb{C}$ 过渡到一个自定义的积范畴。

下面我们将先解释这两个伴随及其对应的范畴和函子，再将它们结合起来。

5.1. Id $\rtimes - \dashv$ App 伴随

我们首先关注概览图中最右侧的伴随。它连接基范畴 $\mathbb{C}$ 与范畴 $\mathbb{P} \rtimes \mathbb{C}$，后者

被定义为范畴 $\mathbb{P}$ 上带基点的自函子与基范畴 $\mathbb{C}$的乘积的一种变体。该范畴 $\mathbb{P} \rtimes \mathbb{C}$ 表示效应操作的高阶签名，由一个内部计算（位于 $\mathbb{P}$ 中）和一个延续（位于 $\mathbb{C}$中）组成。下表总结了不同范畴的对象和态射。

符号对象		态射
$\mathbb{C}$	$A,\ B$	$\mathsf{f} : \mathsf{A} \to \mathsf{B}$
$\mathbb{P}$	$F,\ G$	$\alpha : \mathsf{F} \to \mathsf{G}$
$\mathbb{P} \times \mathbb{C}$	$\langle \mathsf{F}, \mathsf{A} \rangle, \langle \mathsf{G}, \mathsf{B} \rangle$	$\langle \alpha : \mathsf{F} \to \mathsf{G}, \mathsf{f} : \mathsf{A} \to \mathsf{B} \rangle$
$\mathbb{P} \rtimes \mathbb{C}$	$\langle \mathsf{F}, \mathsf{A} \rangle, \langle \mathsf{G}, \mathsf{B} \rangle$	$\langle \alpha : \mathsf{F} \to \mathsf{G}, \mathsf{f} : \mathsf{F}\ \mathsf{A} \to \mathsf{G}\ \mathsf{B} \rangle$

$\mathbb{P} \rtimes \mathbb{C}$ 中的对象形式为 $\langle \mathsf{F}, \mathsf{A} \rangle$，其中 F 是 $\mathbb{C}$ 上的带点自函子，而 A 是 $\mathbb{C}$ 的一个对象。态射由一对 $\langle \alpha, \mathsf{f} \rangle$ 表示，其中包含一个保结构自然变换 $\alpha : \mathsf{F} \to \mathsf{G}$ ($\alpha \circ \eta^F = \eta^G$)，以及一个态射 $\mathsf{f} : \mathsf{F}\ \mathsf{A} \to \mathsf{G}\ \mathsf{B}$。复合与恒等态射按分量定义：

$$\langle \beta, g \rangle \circ \langle \alpha, f \rangle = \langle \beta \circ \alpha, g \circ f \rangle \qquad id_{\langle F, A \rangle} = \langle id_F, id_{F\ A} \rangle$$

左伴随函子 $\mathsf{Id} \rtimes -$将对象和态射从 $\mathbb{C}$ 映射到$\mathbb{P} \rtimes \mathbb{C}$。此外，$\mathsf{Id}$ 本身是一个带点自函子，其 $\eta^{Id} = id$。右伴随函子 App 则相反：它将对象和态射从 $\mathbb{P} \rtimes \mathbb{C}$ 映射到 $\mathbb{C}$，把带点自函子作用于 $\mathbb{C}$ 上的对象 $\mathbb{C}$，并在态射中遗忘自然变换。

$$\begin{array}{ll} (Id \rtimes -)\ A = \langle Id, A \rangle & App\ \langle F, A \rangle = F\ A \\ (Id \rtimes -)\ f\ = \langle \eta^{Id}, Id\ f \rangle = \langle id, f \rangle & App\ \langle \alpha, f \rangle\ = f \end{array}$$

以下同构体现了这个伴随，其中左伴随 ψ 和右伴随 ψ^{-1} 定义如下：

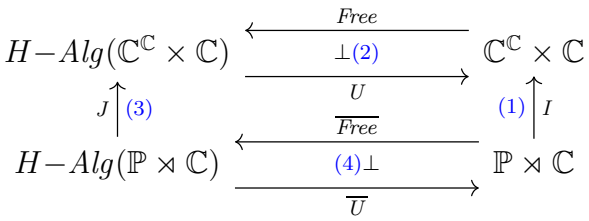
$$\begin{array}{ll} \psi : \mathbb{P} \rtimes \mathbb{C}\ \Big(\ (Id \rtimes -)\ A, \langle F, B \rangle\Big)\ \cong\ \mathbb{C}\ \Big(\ A, App\ \langle F, B \rangle\Big) : \psi^{-1} & \\ \psi \langle \alpha, f \rangle = f & \psi^{-1}\ g = \langle \eta^F, g \rangle \end{array}$$

这些同构的见证满足必要的往返性质。

5.2. Free $\dashv$ U 伴随

现在，我们来解释图左侧的自由-遗忘伴随。事实上，范畴 $\mathbb{P} \rtimes \mathbb{C}$ 是积范畴的一个特例，而自由-遗忘伴随是一种成熟的构造。我们利用

这一情况，将我们的自由-遗忘伴随定义为积范畴的一个特例。



具体而言，我们分四步进行：(1) 我们将范畴 $\mathbb{P} \rtimes \mathbb{C}$ 包含到 $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$；(2) 我们讨论函子 $H : \mathbb{C}^{\mathbb{C}} \times \mathbb{C} \to \mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ 的自由-遗忘伴随；(3) 我们将 $H-Alg(\mathbb{P} \rtimes \mathbb{C})$ 呈现为 H-代数范畴的一个~~受限~~版~~本~~；以及 (4)我们展示 $Free \dashv U$ 是如何由 $Free \dashv U$ 与两个表示子范畴关系的包含函子复合而成。

范畴的推广。 $\mathbb{P} \rtimes \mathbb{C}$ 与积范畴 $\mathbb{P} \times \mathbb{C}$中的一个子范畴同构。事实上，每个对象 $\langle \mathbf{F}, \mathbf{A} \rangle$ 属于 $\mathbb{P} \rtimes \mathbb{C}$，都对应于对象 $\langle \mathbf{F}, \mathbf{F}\ \mathbf{A} \rangle$ 属于 $\mathbb{P} \times \mathbb{C}$。此外，$\mathbb{P} \times \mathbb{C}$ 是$\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ 的一个子范畴[13]，且 $\mathbb{P} \times \mathbb{C}$ 中的每个对象都是 $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ 中的对象，只是忽略了 F 的带基点性质。

$$\begin{aligned} \mathbb{P} \rtimes \mathbb{C} \left(\langle F, A \rangle, \langle G, B \rangle \right) &= \mathbb{P} \times \mathbb{C} \left(\langle F, F\ A \rangle, \langle G, G\ B \rangle \right) \\ &\subseteq \mathbb{C}^{\mathbb{C}} \times \mathbb{C} \left(\langle F, F\ A \rangle, \langle G, G\ B \rangle \right) \end{aligned}$$

通过传递地组合这两个关系，我们得出 $\mathbb{P} \rtimes \mathbb{C}$ 同构于 $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ 的一个子范畴，可由包含函子 $\mathbf{I} : (\mathbb{P} \rtimes \mathbb{C}) \to (\mathbb{C}^{\mathbb{C}} \times \mathbb{C})$ 表示，其中 $\mathbf{I}\ \langle \mathbf{F}, \mathbf{A} \rangle = \langle \mathbf{F}, \mathbf{F}\ \mathbf{A} \rangle$，且 $\mathbf{I}\ \langle \mathbf{h}_1, \mathbf{h}_2 \rangle = \langle \mathbf{h}_1, \mathbf{h}_2 \rangle$。

广义伴随。我们将关注点转向 $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ 范畴。假设 $\mathbb{C}$ 具有自函子的不动点，那么 $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ 也具有。因此，我们可以基于自函子 $\mathbf{H} : (\mathbb{C}^{\mathbb{C}} \times \mathbb{C}) \to (\mathbb{C}^{\mathbb{C}} \times \mathbb{C})$，在 $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ 上实例化自由 - 遗忘伴随。这里，$\mathbf{H}^{\star}$ 是一个自由单子，定义为 $\mathbf{H}^{\star} = \mathbf{U}\ \mathbf{Free}$。此外，这个不动点还带有一个折叠递归方案。

特化伴随。接下来，我们施加五项限制，使该伴随作用于范畴 H–Alg($\mathbb{P} \rtimes \mathbb{C}$) 和 $\mathbb{P} \rtimes \mathbb{C}$ 之间。

(1) 我们将范围限制在 $\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$ 的子范畴中，其中对象的形式为 $\langle \mathbf{F}, \mathbf{F}\ \mathbf{A} \rangle$。

[13]不存在同构，因为并非每个自然变换都会保留 η。

(2) 我们只考虑由高阶双函子 K 构造出的自函子 H: $\mathbb{C}^{\mathbb{C}} \times \mathbb{C} \to \mathbb{C}$，使得 H ⟨F，A⟩ = ⟨K F ∘F，K F A⟩。此外，对于载体为 ⟨F，F A⟩ 的代数，在 Free:$\mathbb{C}^{\mathbb{C}}\times\mathbb{C}$ →H–Alg($\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$) 下，其代数作用的形式为 ⟨$\alpha$: K F ∘F → F，f : K F (F A) →F A⟩。(3) 我们进一步将范围限制在代数的子范畴中，其中 f = α_A，并且只考虑由其自变换 α唯一确定的作用 ⟨α, α_A⟩。(4) 我们只研究 $\mathbb{C}^{\mathbb{C}}$ 对象 F，且 F 是带基点函子，也就是说，它同时也是 $\mathbb{P}$ 中的对象，并带有一个关联的 η^F: Id → F。(5) 最后，我们只考虑满足 h_1 是带基点函子同态的态射 ⟨h_1,h_2⟩。

结果表明，在 (1) 和 (2) 的条件下，H ⋆⟨F , F A⟩ 具有如下形式：⟨$Free_{\mathsf{H}}$ K F , $Free_{\mathsf{H}}$ K F A⟩，其中 H ⋆是广义自由单子。

$$Free_{\mathsf{H}}\ K\ F = \Lambda\ X\ .\ F\ X + K\ Free_{\mathsf{H}}\ (Free_{\mathsf{H}}\ X)$$

因此，H ⋆⟨F , F A⟩ 属于 (1) 的子范畴。此外，由 Free (2) 创建的代数作用也属于这个子范畴 (3)。注意，$Free_{\mathsf{H}}$ 不是 $\mathbb{P} \rtimes \mathbb{C}$中的单子，但它是带基点的 (4)，因为 F 也是如此。此外，(5) 成立，因为函子 Free 保持它：以生成元 ⟨h_1, h_2⟩ 和代数 ⟨α, α_A⟩ 进行折叠后，其第一个分量是一个带基函子态射。根据对范畴 H–Alg ($\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$) 的这五项限制，我们可以定义范畴 H–Alg($\mathbb{P} \rtimes \mathbb{C}$)。$H$–Alg ($\mathbb{P} \rtimes \mathbb{C}$) 中的对象是元组⟨⟨F , A⟩，a : K F ∘F →F⟩，其中载体 ⟨F , A⟩ 是 $\mathbb{P} \rtimes \mathbb{C}$ 中的一个对象，而作用 a 是 $\mathbb{C}^{\mathbb{C}}$中的一个态射。H–Alg ($\mathbb{P} \rtimes \mathbb{C}$) 中的态射 ⟨$h_1, h_2$⟩: ⟨⟨F , A⟩, a⟩ → ⟨⟨G, B ⟩, b⟩是 $\mathbb{P} \rtimes \mathbb{C}$中的同态 ⟨$h_1, h_2$⟩: ⟨F , A⟩ → ⟨G, B ⟩:

$$\begin{array}{ccc} \mathsf{K\ F\ \circ F} & \xrightarrow{K\ h_1\ h_1} & \mathsf{K\ G\ \circ G} \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} \\ F & \xrightarrow[h_1]{} & G \end{array} \qquad \begin{array}{ccc} \mathsf{K\ F\ (F\ A)} & \xrightarrow{K\ h_1\ h_2} & \mathsf{K\ G\ (G\ B\ )} \\ {\scriptstyle a_A}\downarrow & & \downarrow{\scriptstyle b_B} \\ F\ A & \xrightarrow[h_2]{} & G\ B \end{array}$$

请注意，⟨⟨F 、A⟩、a⟩ in H–Alg($\mathbb{P} \rtimes \mathbb{C}$) 中的对象，与满足上述限制的 ⟨⟨F 、F A⟩、⟨a、a_A⟩⟩ of H–Alg($\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$) 中的对象之间存在一一对应关系。类似地，⟨$h_1$,$h_2$⟩ in H–Alg($\mathbb{P} \rtimes \mathbb{C}$) 中的态射，也是满足这些限制的 H–Alg($\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$) 中的态射。上述两个对应关系由包含函子 J : H–Alg($\mathbb{P} \rtimes \mathbb{C}$) → H–Alg($\mathbb{C}^{\mathbb{C}} \times \mathbb{C}$) 见证，其中 J ⟨⟨F , A⟩, a⟩ = ⟨⟨F , F A⟩, ⟨a, a_A⟩⟩ and J ⟨h_1, h_2⟩ = ⟨h_1, h_2⟩。

特殊伴随。在范畴论术语中，我们将 Free $\dashv$ U 伴随定义为 $\overline{J^{-1}} \circ \overline{\text{Free}} \circ \text{I} \dashv I^{-1} \circ \text{U} \circ \text{J}$。这意味着，Free $\dashv$ U 伴随是 Free $\dashv$U 伴随与由同构 $J^{-1} \dashv \text{J}$ 和 $\text{I} \dashv I^{-1}$ 导出的两个伴随的复合。此外，Free 的泛性质也传递给 Free。因此，Free $\dashv$ U 也是一个自由-遗忘伴随，而 U $\circ$ Free 是 $\mathbb{P} \rtimes \mathbb{C}$ 中的自由单子。

5.3. 组合两个伴随

组合这两个伴随后，图便完整了。

$$\begin{array}{ccccc} H\text{-}Alg(\mathbb{C}^{\mathbb{C}} \times \mathbb{C}) & \underset{U}{\overset{Free}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} & \mathbb{C}^{\mathbb{C}} \times \mathbb{C} & & \\ \uparrow J & & \uparrow I & & \\ H\text{-}Alg(\mathbb{P} \rtimes \mathbb{C}) & \underset{\overline{U}}{\overset{\overline{Free}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} & \mathbb{P} \rtimes \mathbb{C} & \underset{App}{\overset{Id \rtimes -}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} & \mathbb{C} \end{array}$$

为了在 $\mathbb{C}$ 中获得自由单子，我们使用 Free $\circ$ (Id $\rtimes$ –) $\dashv$App $\circ$ U 伴随。我们从 $\mathbb{C}$ 中的对象 A 开始，将其转换为 $\mathbb{P} \rtimes \mathbb{C}$ 中的 $\langle \text{Id}, \text{A} \rangle$，其中恒等函子 Id 是带基点的。利用自由-遗忘伴随并应用上述限制，我们得到 $\mathbb{P} \rtimes \mathbb{C}$ 中的自由单子 $\langle Free_{\mathsf{H}} \text{K Id}\ Free_{\mathsf{H}} \text{K Id A} \rangle$。然后，如果再应用 App 函子，我们便得到 $\mathbb{C}$ 中的自由单子 $Free_{\mathsf{H}}$ K Id A，这正是我们在实例化中使用的单子。请注意，本节中的 $Free_{\mathsf{H}}$ K Id 与第 3 节中的 $Free_{\mathsf{H}}$ K 同构。

6. 相关工作

模块化组合副作用。存在多种（模块化）组合不同效应的替代方法，其中最突出的当属单子变换器 [4]。Schrijvers 等人 [26] 详细阐述了在何种情形或条件下，本文所基于的代数效应与处理器方法优于或逊于单子变换器方法。通常，单子变换器的表达能力更强，但代数效应与处理器的模块化程度更高。Jaskelioff [6, 7] 研究了如何让单子变换器更具模块化。

非代数效应。我们并非最先意识到，并非所有带效应的操作都适用于代数效应与处理器的方法。Plotkin 和 Power[23] 最早提出将所谓的非代数操作建模为处理器。然而，这种方法牺牲了模块化：它将语法与语义混为一谈；而将两者清晰分离，正是代数效应与处理器备受推崇的特性之一

。Wu 等人 [31]和 Yang 等人 [33] 表明，Plotkin 和 Power 的方法确实会使作用域效应与处理器失去模块化，并提出了一种不同但非常具体的表示方式。Xie 等人 [32] 和 van den Berg 等人 [1] 则分别为并行效应和潜伏效应提出了类似的（特设）表示方式。据我们所知，后两种表示方式尚未建立在形式化基础之上。

代数效应与高阶效应的应用。代数效应与处理器已应用于许多实用编程语言中，例如 Links[5], Koka [12], Effekt [2], 和 Frank [13], ，以及各种库（例如 fused-effects [25],extensible-effects [10], Ocaml 中的 Eff [11]）。高阶效应也呈现出相同的趋势，其在 GitHub [28] 以及 Haskell 库（如 eff [8], polysemy [14], fused-effects [25], 和 in-other-words[30]）中的应用便是明证。

作为高阶效应的作用域效应与处理器。作用域效应与处理器已从指称语义角度得到研究，Wu 等人 [31] 还实现了作用域操作的余积。此外，我们针对作用域效应的框架实例与他们的单子 E 相对应。在他们的工作中，他们为效应签名提出了一种高阶语法，并显式地对处理器进行“交织”（有时也称为“转发”）：让处理器贯穿高阶语法，并将其分发到每个签名。他们的设计专为作用域效应而设；与我们的框架相比，为了实现所需的模块化，该设计需要额外冗长的表达（即交织）。

Yang 等人 [33] 研究了一种用于作用域效应的自由单子，它与我们的泛型单子非常接近。然而，他们的签名仅针对代数效应和作用域效应，并未推广到其他高阶效应。此外，他们选择用两个代数来解释作用域效应：一个用于作用域内的计算，另一个用于延续。他们只有一个示例，其中这些不同的代数似乎有意义：交替搜索策略（例如深度优先、广度优先和深度受限搜索）。

高阶效应演算。Poulsen 和 van der Rest [24]提出了一种用于建模高阶效应的演算，其操作语义与我们工作的指称语义和范畴语义形成对比。尽管 Poulsen 和 van der Rest 使用了类似的编码方式，通过高阶函子和所谓的 Hefty 代数来表示高阶效应，但他们的关注点有所不同。他们的工作旨在将作用域效应和潜伏效应展开为代数效应。我们提供的高阶效应涵盖范围更广（并行效应、写入器效应、括号效应）。而他们只考虑作用域效应和潜伏效应，

此外，我们还为框架提供了范畴模型，并从理论上证明了其正确性。

7. 结论

总而言之，本研究提供了一种通用方法来编码不同的（代数和高阶）效应，同时保留了代数效应与处理器所珍视的模块化特性。

一些设计选择会影响这项工作的可复用性和易用性。例如，我们的单子与“常规”自由单子非常相似，正如我们在代数效应中所熟知的那样，因为我们选择了带基点自函子，并对积范畴作了自定义定义。这一设计选择意味着，我们为内部计算和效应的延续采用了单一且一致的解释器。乍看之下，这似乎限制了表达能力，但我们发现，几乎没有什么有意义的示例需要采用不同的解释方式。

这项工作的一个可能后续研究方向，是探究不同高阶效应复合时的交互作用与规律（例如，如果我们将作用域效应并行化，会怎样？），这与许多研究探讨状态与非确定性交互作用的思路相同[3, 34] 。

参考文献

[1] van den Berg, B., Schrijvers, T., Poulsen, C.B., Wu, N.：可复用语言组件的潜伏效应。载于：Oh, H.（编）《编程语言与系统》。第182–201页。施普林格国际出版，Cham（2021）[2] Brachth¨auser, J.I., Schuster, P., Ostermann, K.：作为能力的效应：效应处理器与轻量级效应多态。《ACM 程序设计语言会议录》4(OOPSLA)（2020年11月）。https://doi.org/10.1145/3428194,https://doi.org/10.1145/3428194[3] Gibbons, J., Hinze, R.：只管去做：简单的单子等式推理。SIGPLAN Not. 46(9), 2–14（2011年9月）。
https://doi.org/10.1145/2034574.2034777,
https://doi.org/10.1145/2034574.2034777[4]Gill, A.：mtl：单子变换器库（2008年），
https://hackage.haskell.org/package/mtl-1.1.0.2[5] Hillerstr¨om, D., Lindley, S.：通过行与处理器释放效应。载于：第1届类型驱动开发国际研讨会论文集。第15–27页。TyDe 2016，计算机协会，纽约，NY，美国（2016）。https://doi.org/10.1145/2976022.2976033,
https://doi.org/10.1145/2976022.2976033[6]Jaskelioff, M.：Monatron：可扩展的单子变换器库。载于：Scholz, S., Chitil, O.（编）《函数式语言的实现与应用——第20届国际研讨会，IFL 2008，英国哈特菲尔德，2008年9月10日至12日。修订精选论文》。《计算机科学讲义》，第5836卷，第233–248页。施普林格（2008年）。
https://doi.org/10.1007/978-3-642-24452-0 13,
https://doi.org/10.1007/978-3-642-24452-0_13

-

[7] Jaskelioff, M.：模块化单子变换器。载于：Castagna, G.（编）《编程语言与系统》，第18届欧洲编程研讨会，ESOP 2009，作为软件理论与实践联合欧洲会议 ETAPS 2009 的一部分举行，英国约克，2009年3月22日至29日。论文集。《计算机科学讲义》，第5502卷，第64–79页。施普林格（2009）。
https://doi.org/10.1007/978-3- 642-00590-9 6,
https://doi.org/10.1007/978-3-642-00590-9_6

[8]King, A.: eff – 极速可扩展效应（2019），
https://github.com/hasura/eff

[9]Kiselyov, O., Ishii, H.: Freer 单子：更多可扩展效应。SIGPLAN Not. 50(12), 94–105（2015 年 8 月）。https://doi.org/10.1145/2887747.2804319, https://doi.org/10.1145/2887747.2804319

[10]Kiselyov, O., Sabry, A., Swords, C., Foppa, B.：可扩展效应：单子变换器的替代方案（2019）, https://hackage.haskell.org/package/extensible-effects

[11]Kiselyov, O., Sivaramakrishnan, K.：直接在 OCaml 中使用 Eff。《电子理论计算机科学论文集》285，23–58（2018 年 12 月）。https://doi.org/10.4204/eptcs.285.2, http://dx.doi.org/10.4204/EPTCS.285.2

[12] Leijen, D.：Koka：使用行多态效应类型进行编程。《电子理论计算机科学论文集》153（2014 年 6 月）。https://doi.org/10.4204/EPTCS.153.8

[13]Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. 载于：第 44 届 ACM SIGPLAN 编程语言原理研讨会论文集。第 500–514 页。POPL 2017，计算机协会，美国纽约州纽约（2017）。https://doi.org/10.1145/3009837.3009897, https://doi.org/10.1145/3009837.3009897[14]Maguire, S.: polysemy：高阶、低样板代码的自由单子（2019）, https://hackage.haskell.org/package/polysemy[15]Marlow, S., Jones, S.P., Moran, A., Reppy, J.: Haskell 中的异步异常。SIGPLAN Not. 36(5), 274–285（2001 年 5 月）。https://doi.org/10.1145/381694.378858, https://doi.org/10.1145/381694.378858[16] Moggi, E.: 编程语言的抽象视角。技术报告 ECS-LFCS-90-113，爱丁堡大学计算机科学系（1989 年 6 月）[17]Moggi, E.: 计算概念与单子。《信息与计算》93(1), 55–92（1991）。https://doi.org/https://doi.org/10.1016/0890-5401(91)90052-4，选自 1989 年 IEEE 计算机科学逻辑研讨会的论文[18]Paszke, A., Johnson, D.D., Duvenaud, D., Vytiniotis, D., Radul, A., Johnson, M.J., Ragan-Kelley, J., Maclaurin, D.: 探索……

关键点：面向点式数组编程的指标集与保持并行性的自动微分。《ACM Program. Lang. 论文集》5(ICFP)（2021 年 8 月）。 https://doi.org/10.1145/3473593, https://doi.org/10.1145/3473593

[19] Pauwels, K.、Schrijvers, T.、Mu, S.C.：使用全局状态处理局部状态。载于：Hutton, G.（编）《程序构造数学》，第 18–44 页。施普林格国际出版，Cham（2019）

[20]Pir´og, M.、Schrijvers, T.、Wu, N.、Jaskelioff, M.：带作用域的操作的语法与语义。载于：《第33届 ACM/IEEE 计算机科学逻辑年会论文集》，第 809–818 页。LICS '18，计算机协会，纽约，NY，美国（2018）。 https://doi.org/10.1145/3209108.3209166, https://doi.org/10.1145/3209108.3209166

[21] Plotkin, G.、Pretnar, M.：代数效应处理器。载于：Castagna, G.（编）《编程语言与系统》，第 80–94 页。施普林格柏林海德堡，柏林，海德堡（2009）。https://doi.org/10.1007/978-3- 642-00590-9 7

[22]Plotkin, G., Pretnar, M.：处理代数效应。Logical Methods in Computer Science 9(4)（2013 年 12 月）。https://doi.org/10.2168/lmcs-9(4:23)2013

[23]Plotkin, G.D., Power, J.：代数操作与泛型效应。Appl. Categorical Struct. 11(1), 69–94（2003）。https://doi.org/10.1023/A:1023064908962, https://doi.org/10.1023/A:1023064908962

[24] Poulsen, C.B., van der Rest, C.：Hefty 代数：高阶代数效应的模块化展开（2023）[25]Rix, R., Thomson, P., Wu, N., Schrijvers, T.：融合效应：一种快速、灵活的融合效应系统（2018）， https://hackage.haskell.org/package/fused-effects[26] Schrijvers, T., Pir´og, M., Wu, N., Jaskelioff, M.：单子变换器与模块化代数效应：将它们联系在一起的是什么。载于：第12届 ACM SIGPLAN Haskell 国际研讨会论文集。第98–113页。Haskell 2019，计算机协会，纽约，纽约州，美国（2019）。 https://doi.org/10.1145/3331545.3342595, https://doi.org/10.1145/3331545.3342595

[27]Swierstra, W.：《自助餐式数据类型》。J. Funct. Program. 18(4), 423–436（2008年7月）。https://doi.org/10.1017/S0956796808006758, https://doi.org/10.1017/S0956796808006758

[28]Thomson, P., Rix, R., Wu, N., Schrijvers, T.：在 GitHub 融合工业界与学术界（经验报告）。载于：第 18 届 ACM SIGPLAN 国际函数式编程会议论文集（2022）， https://arxiv.org/abs/2206.09206

[29]Wadler, P.：《函数式编程的单子》。载于：高级函数式编程，第一届高级函数式编程技术国际暑期学校——教程文本。第24–2页。Springer-Verlag，柏林，海德堡（1995）。https://doi.org/10.5555/647698.734146[30]Waern, L.：in-other-words：一种高阶效应系统，天空才是极限（2022），https://hackage.haskell.org/package/in-other-words[31] Wu, N.、Schrijvers, T.、Hinze, R.：作用域中的效应处理器。SIGPLAN Not. 49(12)，1–12（2014年9月）。https://doi.org/10.1145/2775050.2633358, https://doi.org/10.1145/2775050.2633358[32]Xie, N.、Johnson, D.D.、Maclaurin, D.、Paszke, A.：并行代数效应处理器。CoRR abs/2110.07493（2021），https://arxiv.org/abs/2110.07493[33] Yang, Z.、Paviotti, M.、Wu, N.、van den Berg, B.、Schrijvers, T.：作用域效应的结构化处理。载于：Sergey, I.（编），编程语言与系统。第462–491页。施普林格国际出版，Cham（2022）[34]Yang, Z.、Wu, N.：通过融合推理效应交互。ACM Program. Lang. 5(ICFP)（2021年8月）。https://doi.org/10.1145/3473578， https://doi.org/10.1145/3473578

附录 A. 范畴论背景

本附录将介绍一些基本定义、概念和术语。本小节为第5节提供必要的预备知识，该节将详细阐述我们通用框架的范畴论基础。

定义 1（范畴）。范畴 $\mathbb{C}$由四个部分定义：1. 对象 A、B 的集合。2. 对于 $\mathbb{C}$中的任意两个对象，存在一个由这两个对象之间的态射构成的集合。它通常称为同态集，记作 $\mathbb{C}$(A, B)。3. 对于每个对象，都有一个恒等态射：$\forall$A $\in \mathbb{C}$：$id_A \in \mathbb{C}$(A, A) 4. 对于任意两个态射，它们的复合为：$\forall$ f $\in$ $\mathbb{C}$ (A, A), g $\in \mathbb{C}$(B, C) : f $\circ$g $\in \mathbb{C}$(A, C) 此外，还需满足两个性质：

1. 复合的结合律。

$$\forall A, B, C, D \in \mathbb{C} : \forall f \in \mathbb{C}(A, B), g \in \mathbb{C}(B, C), h \in \mathbb{C}(C, D) : \\ (f \circ g) \circ h \equiv f \circ (g \circ h)$$

2. 复合的左单位和右单位。

$$\forall A, B \in \mathbb{C} : \forall f \in \mathbb{C}(A, B) : id_B \circ f \equiv f \equiv f \circ id_A$$

□

定义 2（同态）。同态是一种保持对象代数结构的态射。

定义 3（始对象）。范畴 $\mathbb{C}$ 的始对象是 $\mathbb{C}$ 中的一个对象，使得对于 $\mathbb{C}$ 中的所有对象 A，都存在一个从该始对象到 A 的唯一态射。

定义 4（函子）。函子 F：$\mathbb{C} \to \mathbb{D}$ 是两个范畴 $\mathbb{C}$ 与 $\mathbb{D}$之间的保结构映射。它将范畴 $\mathbb{C}$ 的对象和同态集都映射到范畴 $\mathbb{D}$中，同时保持单位元和复合：F $id_A = id_{F\,A}$以及 F（f $\circ$g） =F f $\circ$F g。□

定义 5（自函子）。自函子 F :$\mathbb{C} \to \mathbb{C}$ 是从一个范畴到其自身的函子。$\mathbb{C}$ 上的自函子范畴记为 $\mathbb{C}^{\mathbb{C}}$。 □

定义 6（带点自函子）。若自函子 $\mathsf{P}:\mathbb{C}\to\mathbb{C}$ 配备了一个指定的自然变换，从恒等函子 $\eta^P:\mathsf{Id}\to\mathsf{P}$，则称其为带基点的。带基点自函子范畴记为 $\mathbb{P}$。□

定义 7（伴随）。伴随 $\mathsf{L}\dashv\mathsf{R}$ 是两个函子 $\mathsf{L}:\mathbb{C}\to\mathbb{D}$ 和 $\mathsf{R}:\mathbb{D}\to\mathbb{C}$之间的一种关系，其中 $\mathbb{C}$ 和 $\mathbb{D}$ 是范畴，并且它们的同态集之间存在自然同构：

$$\psi:\mathbb{C}(\ L\ A,B\ )\ \cong\ \mathbb{D}(\ A,R\ B\ ):\psi^{-1}$$

这里，ψ 和 ψ^{-1} 是见证该同构的两个自然变换，它们应满足必要的往返性质（$\psi\ \circ\ \psi^{-1}=id=\psi^{-1}\ \circ\ \psi$）。□

通常，伴随关系用图表示，L 和 R 分别称为左伴随和右伴随。

$$\mathbb{D}\ \underset{R}{\overset{L}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\ \mathbb{C}$$

伴随关系会在 $\mathbb{C}$中导出一个单子 $\mathsf{R}\ \circ\mathsf{L}$，它配备了单位算子和顺序算子。

$$\begin{aligned}&\eta\quad:A\to R\ (L\ A)\\&\mathsf{join}:(R\ \circ\ L\ \circ\ R\ \circ\ L)\ A\to R\ (L\ A)\end{aligned}$$

伴随可以组合：我们可以组合 $\mathsf{L}_1\dashv\mathsf{R}_1$ 和 $\mathsf{L}_2\dashv\mathsf{R}_2$, 的函子，使得 $\mathsf{L}_2\circ\mathsf{L}_1\dashv\mathsf{R}_1\circ\mathsf{R}_2$。

$$\mathbb{C}_1\ \underset{R_2}{\overset{L_2}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\ \mathbb{C}_2\ \underset{R_1}{\overset{L_1}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\ \mathbb{C}_3\qquad\cong\qquad\mathbb{C}_1\ \underset{R_1\circ R_2}{\overset{L_2\circ L_1}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\ \mathbb{C}_3$$

此外，我们可以定义伴随的积 $\mathsf{L}_1\dashv\mathsf{R}_1$ 和 $\mathsf{L}_2\dashv\mathsf{R}_2$, ，它们作用于各自的积范畴之间，并取伴随函子的积。

$$\mathbb{C}_1\ \underset{R_1}{\overset{L_1}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\ \mathbb{C}_2\ \text{ and }\ \mathbb{C}_3\ \underset{R_2}{\overset{L_2}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\ \mathbb{C}_4\qquad\text{then}\qquad\mathbb{C}_1\times\mathbb{C}_3\ \underset{R_1\times R_2}{\overset{L_1\times L_2}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\ \mathbb{C}_2\times\mathbb{C}_4$$

定义 8（自由-遗忘伴随）。一般而言，自由-遗忘伴随由一个自由函子 Free 和一个遗忘函子 U 组成，其中 Free 是 U 的左伴随，而 U 会遗忘某些结构。

$$\Sigma\text{-}Alg(\mathbb{C}) \underset{U}{\overset{Free}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{C}$$

这里，我们考虑范畴 $\mathbb{C}$ 中的一个自函子 Σ: $\mathbb{C} \to \mathbb{C}$。左侧的范畴是 Σ-代数在 $\mathbb{C}$ 上构成的范畴，其对象为形如 ⟨A, alg⟩ 的有序对，其中 A 称为代数的载体，而 alg : Σ A → A 是作用。 Σ–Alg(ℂ) 中的同态（图 8）f : A → B 将载体映射起来。

$$\begin{array}{ccc} \Sigma\ A & \xrightarrow{\Sigma f} & \Sigma\ B \\ {\scriptstyle alg_A}\downarrow & & \downarrow{\scriptstyle alg_B} \\ A & \xrightarrow[f]{} & B \end{array}$$

图 8. 作用是保结构的。

□

现在，范畴 Σ–Alg(ℂ) 也有一个始对象。根据 Lambek 定理，始对象的作用是一个同构（Σi ≅ i，其中 i 是始对象）。这意味着始对象是 Σ 的不动点。因此，我们考虑载体为 $\Sigma^\star$ A 的代数；它有一个作用 Op : Σ ($\Sigma^\star$ A) → $\Sigma^\star$ A。此外，始对象的定义指出，从始对象到任何其他 Σ-代数都必须存在唯一的同态。我们称这个唯一的同态为折叠，并用以下两个必须交换的图来描述其唯一性。

$$\begin{array}{ccc} A & \xrightarrow{Var} & \Sigma^\star\ A \\ {\scriptstyle gen}\downarrow & \swarrow{\scriptstyle \text{fold}} & \\ B & & \end{array} \qquad\qquad \begin{array}{ccc} \Sigma\ (\Sigma^\star\ A) & \xrightarrow{Op} & \Sigma^\star\ A \\ {\scriptstyle \Sigma\ \text{fold}}\downarrow & & \downarrow{\scriptstyle \text{fold}} \\ \Sigma\ B & \xrightarrow[alg]{} & B \end{array}$$

这种唯一性称为折叠的泛性质。这个折叠由生成元 gen : A → B 和目标 Σ-代数的作用决定；gen 是 Σ-代数范畴中的一个态射。自由-遗忘伴随导出 $\mathbb{C}$ 中由 Σ 生成的自由单子；这个自由单子通常记作 $\Sigma^\star$。对于生成元为 gen、代数为 alg 的折叠，另一种记法是 ⦇ gen, alg ⦈。请注意，这个伴随与我们在第 2.1 节中的自由单子定义相似：我们将 $\Sigma^\star$ 记作 Free σ，使用 Var 作为单子的 return，使用 Op 作为始对象的作用，并使用 fold_{Alg} gen alg 作为结构递归方案，将 $\Sigma^\star$ A 解释到 B 中，其实现与上述交换图相符。

代数效应与处理器的模型围绕自由单子的定义（效应）及其通过折叠式递归方案进行的解释（处理器）展开。操作由基范畴 $\mathbb{C}$ 上的自函子 Σ : $\mathbb{C} \to \mathbb{C}$ 捕获。这个自函子正是我们在上述自由-遗忘伴随中描述的函子。此外，它表示

效应签名，并决定了调用 Σ- 操作的程序形状，以及解释这些操作的处理器形状。

附录 B. 使用 censor 修改日志的两种方式

我们可以用 pass 来定义 censor。这两个操作的区别在于，censor 单独接收修改函数，而在 pass 中，修改函数是计算结果的一部分。因此，我们可以轻松地用 pass 定义 censor，如下所示：

$$\begin{aligned}&\mathsf{censor} :: (w \to w) \to m\ a \to m\ a\\&\mathsf{censor}\ f\ k = \mathsf{pass}\ \$\ \mathbf{do}\ x \leftarrow k; \mathsf{return}\ (x, f)\end{aligned}$$

根据这个定义，我们可以用 censor 重写 reset 示例。

$$reset = \mathsf{censor}\ (const\ \epsilon)\ (\mathsf{return}\ ())$$

结果保持不变

$$\begin{aligned}&\texttt{>>>}\ h_{\mathsf{Write}}\ (\mathsf{tell}\ \texttt{"post"} \gg reset \gg \mathsf{tell}\ \texttt{"pre"})\\&((), \texttt{"post"})\end{aligned}$$

或者，我们可以把这个函数看作 local 的对偶形式：local 用于从环境中读取时，会先修改环境，再执行计算。与之对偶，censor 会先执行计算，然后修改输出。local 和 censor 都可以建模为作用域计算。例如，我们可以用作用域效应 Censor w 扩展上述处理器。

$$\mathbf{data}\ Censor\ w\ a = Censor\ (w \to w)\ a$$

此外，我们相应地扩展处理器：

$$\begin{aligned}&h_{\mathsf{Censor}} :: (Functor\ \sigma, Functor\ \gamma, Monoid\ w, Functor\ \varphi)\\&\quad \Rightarrow Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ (Tell\ w + \sigma) \oplus K^{\mathsf{Sc}}\ (Censor\ w + \gamma)\\&\qquad\qquad \oplus K^{\mathsf{Write}}\ (Listen\ w + Pass\ w + \varphi))\ a\\&\quad \to Free_{\mathsf{H}}\ (K^{\mathsf{Alg}}\ \sigma \oplus K^{\mathsf{Sc}}\ \gamma \oplus K^{\mathsf{Write}}\ \varphi)\ (a, w)\\&h_{\mathsf{Censor}} = \mathsf{fold}\ gen\ (alg_{\mathsf{Alg}} \mathbin{⊕} alg_{\mathsf{Sc}} \mathbin{⊕} alg_{\mathsf{Write}})\ \mathbf{where}\\&\quad alg_{\mathsf{Sc}}\ (\mathsf{Enter}\ sc) = (alg_{\mathsf{Censor}} \mathbin{\#} fwd_{\mathsf{Censor}})\ sc\ \mathbf{where}\\&\qquad alg_{\mathsf{Censor}}\ (Censor\ f\ k) = \mathbf{do}\ (mx, _) \leftarrow k; (x, w) \leftarrow mx; \mathsf{return}\ (x, f\ w)\\&\qquad fwd_{\mathsf{Censor}} = Op_{\mathsf{H}}\ .\ \mathsf{Enter}\ .\ \mathsf{fmap}\ (\mathsf{fmap}\ fst)\\&\quad \ldots\end{aligned}$$

该示例计算得出的结果与之前相同。

```
>>> h_Censor (tell "post" ≫ reset ≫ tell "pre")
((), "post")
```

附录 C. 同构

附录 C.1. 代数效应与处理器

首先，我们定义这些同构：

iso_1 :: 函子 $\sigma \Rightarrow$ Free σ a $\rightarrow Free_{\mathsf{H}}$ (K^{Alg} σ) aiso_1 (Var x) $= Var_{\mathsf{H}}$ xiso_1 (Op op) $= Op_{\mathsf{H}}$ (Op (fmap iso_1 op))

iso_2 :: 函子 $\sigma \Rightarrow Free_{\mathsf{H}}$ (K^{Alg} σ) a $\rightarrow$ Free σ aiso_2 (Var_{H} x) = Var xiso_2 (Op_{H} (Op op)) $=$ Op (fmap iso_2 op)

接下来，我们证明所需的往返性质成立，即 $iso_1 \,.\, iso_2 = id = iso_2 \,.\, iso_2$。

$$
\begin{array}{ll}
(iso_1 \,.\, iso_2)\ (Var_{\mathsf{H}}\ x) & (iso_2 \,.\, iso_1)\ (Var\ x) \\
\quad \equiv iso_1\ (Var\ x) & \quad \equiv iso_2\ (Var_{\mathsf{H}}\ x) \\
\quad \equiv Var_{\mathsf{H}}\ x & \quad \equiv\ Var\ x \\
(iso_1 \,.\, iso_2)\ (Op_{\mathsf{H}}\ (\mathsf{Op}\ op)) & (iso_2 \,.\, iso_1)\ (Op\ op) \\
\quad \equiv iso_1\ (Op\ (\mathsf{fmap}\ iso_2\ op)) & \quad \equiv iso_2\ (Op_{\mathsf{H}}\ (\mathsf{Op}\ (\mathsf{fmap}\ iso_1\ op))) \\
\quad \equiv Op_{\mathsf{H}}\ (\mathsf{Op}\ (\mathsf{fmap}\ iso_1\ (\mathsf{fmap}\ iso_2\ op))) & \equiv Op\ (\mathsf{fmap}\ iso_2\ (\mathsf{fmap}\ iso_1\ op)) \\
\quad \equiv Op_{\mathsf{H}}\ (\mathsf{Op}\ (\mathsf{fmap}\ (iso_1 \,.\, iso_2)\ op)) & \quad \equiv Op\ (\mathsf{fmap}\ (iso_2 \,.\, iso_1)\ op) \\
\quad \equiv Op_{\mathsf{H}}\ (\mathsf{Op}\ op) & \quad \equiv Op\ op
\end{array}
$$

此外，这两种递归数据类型的处理器满足以下等价关系：fold^{Alg} gen alg $= h_{\mathsf{Alg}}$ gen (λ(Op op) $\rightarrow$ alg op) $\,.\, iso_1$。

h_{Alg} 生成（λ（操作 op） $\rightarrow$ 算法 op）（iso_1 （变量 x ））$\equiv h_{\mathsf{Alg}}$ 生成（λ（操作 op） $\rightarrow$ 算法 op）（Var_{H} x ）≡生成 x≡ 折叠^{Alg} 生成算法（变量 x ）

$$
\begin{array}{l}
h_{\mathsf{Alg}}\ gen\ (\lambda(\mathsf{Op}\ op) \rightarrow alg\ op)\ (iso_1\ (Op\ x)) \\
\quad \equiv h_{\mathsf{Alg}}\ gen\ (\lambda(\mathsf{Op}\ op) \rightarrow alg\ op)\ (Op_{\mathsf{H}}\ (\mathsf{Op}\ (\mathsf{fmap}\ iso_1\ x)))
\end{array}
$$

$\equiv (\lambda(\mathsf{Op}\ op) \to alg\ op)$
$\quad (\mathsf{hmap}\ \mathsf{fold}_2\ (\mathsf{fmap}\ (h_{\mathsf{Alg}}\ gen\ (\lambda(\mathsf{Op}\ op) \to alg\ op))\ (\mathsf{Op}\ (\mathsf{fmap}\ iso_1\ x))))$
$\equiv (\lambda(\mathsf{Op}\ op) \to alg\ op)$
$\quad (\mathsf{hmap}\ \mathsf{fold}_2\ (\mathsf{Op}\ (\mathsf{fmap}\ (h_{\mathsf{Alg}}\ gen\ (\lambda(\mathsf{Op}\ op) \to alg\ op)\ (\mathsf{fmap}\ iso_1\ x)))))$
$\equiv (\lambda(\mathsf{Op}\ op) \to alg\ op)$
$\quad (\mathsf{Op}\ (\mathsf{fmap}\ (h_{\mathsf{Alg}}\ gen\ (\lambda(\mathsf{Op}\ op) \to alg\ op))\ (\mathsf{fmap}\ iso_1\ x))))$
$\equiv alg\ (\mathsf{fmap}\ (h_{\mathsf{Alg}}\ gen\ (\lambda(\mathsf{Op}\ op) \to alg\ op))\ (\mathsf{fmap}\ iso_1\ x)))$
$\equiv alg\ (\mathsf{fmap}\ (h_{\mathsf{Alg}}\ gen\ (\lambda(\mathsf{Op}\ op) \to alg\ op)\ .\ iso_1)\ x))$
$\equiv alg\ (\mathsf{fmap}\ (\mathsf{fold}_{\mathsf{Alg}}\ gen\ alg)\ x))$
$\equiv \mathsf{fold}_{\mathsf{Alg}}\ gen\ alg\ (Op\ x)$

附录 C.2。作用域效应与处理器

首先，我们定义这些同构：

iso_1 :: 函子 $\gamma \Rightarrow Free_{\mathsf{Sc}}\ \gamma\ \mathsf{a} \to Free_{\mathsf{H}}\ (K^{\mathsf{Sc}}\ \gamma)\ \mathsf{a} iso_1\ (Var\ \mathsf{x})$
$= Var_{\mathsf{H}}\ \mathsf{x} iso_1\ (\mathsf{Enter\ sc}) = Op_{\mathsf{H}}\ (\mathsf{Enter\ (fmap}\ (iso_1\ .\mathsf{fmap}\ iso_1)$
sc))

iso_2 :: 函子 $\gamma \Rightarrow Free_{\mathsf{H}}\ (K^{\mathsf{Sc}}\ \gamma)\ \mathsf{a} \to Free_{\mathsf{Sc}}\ \gamma\ \mathsf{a} iso_2\ (Var_{\mathsf{H}}\ \mathsf{x})$
$= Var\ \mathsf{x} iso_2\ (Op_{\mathsf{H}}\ (\mathsf{Enter\ sc})) = \mathsf{Enter\ (fmap}\ (iso_2\ .\mathsf{fmap}\ iso_2)$
sc)

接下来，我们证明所需的往返性质成立，即 $iso_1\ .\ iso_2 = id = iso_2\ .\ iso_2$。

$(iso_1\ .\ iso_2)\ (Var_{\mathsf{H}}\ \mathsf{x})$
$\equiv iso_1\ (Var\ \mathsf{x}) \equiv Var_{\mathsf{H}}$
x

$(iso_1\ .\ iso_2)\ (Op_{\mathsf{H}}\ (\mathsf{Enter\ sc})) \equiv iso_1\ (\mathsf{Enter\ (fmap}\ (iso_2\ .\mathsf{fmap}\ iso_2)$
$\mathsf{sc})) \equiv Op_{\mathsf{H}}\ (\mathsf{Enter\ (fmap}\ (iso_1\ .\mathsf{fmap}\ iso_1))\ (\mathsf{fmap}\ (iso_2\ .\mathsf{fmap}\ iso_2)$
$\mathsf{sc})) \equiv Op_{\mathsf{H}}\ (\mathsf{Enter\ (fmap}\ (iso_1\ .\ \mathsf{fmap}\ iso_1\ .\ iso_2\ .\ \mathsf{fmap}\ iso_2)\ \mathsf{sc}))$
$\equiv Op_{\mathsf{H}}\ (\mathsf{Enter\ (fmap}\ (iso_1\ .\ iso_2\ .\ \mathsf{fmap}\ iso_1\ .\ \mathsf{fmap}\ iso_2)\ \mathsf{sc}))$
$\equiv Op_{\mathsf{H}}\ (\mathsf{Enter\ (fmap}\ (iso_1\ .\ iso_2\ .\ \mathsf{fmap}\ (iso_1\ .\ iso_2))\ \mathsf{sc})) \equiv Op_{\mathsf{H}}$
(Enter sc)

$(iso_2\ .\ iso_1)\ (Var\ \mathsf{x})$
$\equiv iso_2\ (Var_{\mathsf{H}}\ \mathsf{x})$
$\equiv Var\ \mathsf{x}$

($iso_2 \,.\, iso_1$) (Enter sc)≡ iso_2 (Op_{H} (Enter (fmap (iso_1 .fmap iso_1) sc)))≡ Enter (fmap (iso_2 .fmap iso_2) (fmap (iso_1 .fmap iso_1) sc))≡ Enter (fmap ($iso_2 \,.\, \text{fmap}\ iso_2 \,.\, iso_1 \,.\, \text{fmap}\ iso_1$) sc)≡ Enter (fmap ($iso_2 \,.\, iso_1 \,.\, \text{fmap}\ iso_2 \,.\, \text{fmap}\ iso_1$) sc)≡ Enter (fmap ($iso_2 \,.\, iso_1 \,.\, \text{fmap}\ (iso_2 \,.\, iso_1)$) sc)≡Enter sc

此外，这两种递归数据类型的处理器满足以下等价关系：折叠$_{\mathsf{Sc}}$ 生成器 $alg_E\ alg_E = h_{\mathsf{Sc}}$ 生成器 (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc) $.\, iso_1$with $return_E\ alg_E = \eta_\circ$

h_{Sc} 生成器 (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc) (iso_1 (Var x))≡ h_{Sc} 生成器 (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc) (Var_{H} x)≡生成器 x≡ 折叠$_{\mathsf{Sc}}$ 生成器 $alg_E\ alg_E$ (Var x)

h_{Sc} 生成器 (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc) (iso_1 (Enter sc))≡ h_{Sc} 生成器 (λ (Enter sc) $\rightarrow enter_E\ alg_E$ sc) (Op_{H} (Enter (fmap (iso_1 .fmap iso_1) sc)))≡ (λ (Enter sc) $\rightarrow enter_E\ alg_E$ sc) (hmap 折叠$_2$ (fmap (h_{Sc} 生成器 (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc)) (Enter (fmap (iso_1 .fmap iso_1) sc))))≡ (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc) (hmap 折叠$_2$ (Enter (fmap (fmap (h_{Sc} 生成器 (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc)))) (fmap (iso_1 .fmap iso_1) sc)))≡ (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc) (hmap 折叠$_2$ (Enter (fmap (fmap (h_{Sc} 生成器 (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc)) $.iso_1$ .fmap iso_1) sc)))≡ (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc) (Enter (fmap 折叠$_2$(fmap (fmap (h_{Sc} 生成器 (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc)) . iso_1 .fmap iso_1) sc)))≡ $enter_E\ alg_E$ (fmap 折叠$_2$ (fmap (fmap (h_{Sc} 生成器 (λ (Enter sc) $\rightarrow enter_E\ alg_E$ sc)

) .

iso_1 .fmap iso_1) sc))≡ $enter_E\ alg_E$ (fmap (折叠$_2$ .fmap (h_{Sc} gen (λ (Enter sc) $\rightarrow enter_E\ alg_E$ sc)) $.iso_1$ .fmap iso_1) sc))≡ $enter_E\ alg_E$ (f map (折叠$_2$ .fmap (h_{Sc} gen (λ(Enter sc) $\rightarrow enter_E\ alg_E$ sc)) . fmap $iso_1 \,.\, iso_1$) sc))

$\equiv enter_E\ alg_E$(fmap (折叠$_2$ .fmap (h_{Sc} gen (λ(Enter sc) $\to enter_E\ alg_E$ sc) .iso_1) . iso_1) sc))$\equiv enter_E\ alg_E$(fmap (折叠$_2$. fmap (折叠$_{\mathsf{Sc}}$ gen $alg_E\ alg_E$) . iso_1) sc))$\equiv enter_E\ alg_E$(fmap (折叠$_2$. iso_1 . fmap (折叠$_{\mathsf{Sc}}$ gen $alg_E\ alg_E$)) sc))$\equiv enter_E\ alg_E$ (fmap ($h_{cata}\ alg_E$ .fmap (折叠$_{\mathsf{Sc}}$ gen $alg_E\ alg_E$)) sc))$\equiv$折叠$_{\mathsf{Sc}}$ gen $alg_E\ alg_E$ (Enter sc)

我们需要一个辅助引理，说明 $h_{cata}\ \ alg_E\ =$ fold$_2$. iso_1 with$return_E\ alg_E = \eta_\circ$

$h_{cata}\ alg_E$(Var x)
$\equiv return_E\ alg_E$ x$\equiv \eta$
x$\equiv$ 折叠$_2$ (Var_{H} x)$\equiv$ 折
叠$_2$ (iso_1 (Var x))

$h_{cata}\ alg_E\ (Enter\ sc)$
$\equiv (enter_E\ alg_E$. fmap $(h_{cata}\ alg_E$. fmap $(h_{cata}\ alg_E)))\ sc$
$\equiv (enter_E\ alg_E$. fmap $(h_{cata}\ alg_E$. fmap $(h_{cata}\ alg_E)))\ sc$
$\equiv (alg$. Enter . fmap $(h_{cata}\ alg_E$. fmap $(h_{cata}\ alg_E)))\ sc$
$\equiv alg$ (Enter (fmap $(h_{cata}\ alg_E$. fmap $(h_{cata}\ alg_E))\ sc)$)
$\equiv alg$ (Enter (fmap (fold$_2$. iso_1 . fmap $(h_{cata}\ alg_E))\ sc)$)
$\equiv alg$ (Enter (fmap (fold$_2$. fmap $(h_{cata}\ alg_E)$. $iso_1)\ sc)$)
$\equiv alg$ (Enter (fmap fold$_2$ (fmap (fmap $(h_{cata}\ alg_E)$. $iso_1)\ sc)$))
$\equiv alg$ (hmap fold$_2$ (Enter (fmap (fmap $(h_{cata}\ alg_E)$. $iso_1)\ sc)$))
$\equiv alg$ (hmap fold$_2$ (Enter (fmap (fmap (fold$_2$. $iso_1)$. $iso_1)\ sc)$))
$\equiv alg$ (hmap fold$_2$ (Enter (fmap (fmap fold$_2$. iso_1 . fmap $iso_1)\ sc)$))
$\equiv alg$ (hmap fold$_2$ (Enter (fmap (fmap fold$_2$) (fmap $(iso_1$. fmap $iso_1)\ sc)$))))
$\equiv alg$ (hmap fold$_2$ (fmap fold$_2$ (Enter (fmap $(iso_1$. fmap $iso_1)\ sc)$))))
$\equiv$ fold$_2$ (Op_{H} (Enter (fmap $(iso_1$. fmap $iso_1)\ sc)$))
$\equiv$ fold$_2$ ($iso_1\ (Enter\ sc)$)

附录 C.3. 并行效应与处理器

首先，我们定义这些同构：

iso_1 :: 函子 $\rho \Rightarrow Free_{\mathsf{Par}}\ \rho$ a $\to Free_{\mathsf{H}}$ ($K^{\mathsf{Par}}\ \rho$) aiso_1 (Var x)
$= Var_{\mathsf{H}}$ xiso_1(For iters k) $= Op_{\mathsf{H}}$(For (fmap iso_1 iters) (f
map iso_1 k))

iso_2 :: 函子 $\rho \Rightarrow Free_{\mathsf{H}}$ ($K^{\mathsf{Par}}\ \rho$) a $\to Free_{\mathsf{Par}}\ \rho$ aiso_2 (Var_{H} x)
$= Var$ xiso_2(Op_{H}(For iters k)) =For (fmap iso_2 iters) (fmap
iso_2 k)

接下来，我们证明所需的往返性质成立，即 $iso_1 \,.\, iso_2 = id = iso_2 \,.\, iso_2$。

($iso_1 \,.\, iso_2$) (Var_{H} x)≡ iso_1 (Var x)≡ Var_{H} x ($iso_1 \,.\, iso_2$) (Op_{H} (For iters k))≡ iso_1 (For (fmap iso_2 iters) (fmap iso_2 k))≡ Op_{H} (For (fmap iso_1 (fmap iso_2 iters)) (fmap iso_1 (fmap iso_2 k))) ≡ Op_{H}(For (fmap ($iso_1 \,.\, iso_2$) iters) (fmap ($iso_1 \,.\, iso_2$) k))≡ Op_{H} (For iters k) ($iso_2 \,.\, iso_1$) (Var x)≡ iso_2 (Var_{H} x)≡ Var x

($iso_2 \,.\, iso_1$) (For iters k)≡ iso_2 (Op_{H} (For (fmap iso_1 iters) (fmap iso_1 k)))≡ For (fmap iso_2 (fmap iso_1 iters)) (f map iso_2 (fmap iso_1 k))≡For (fmap ($iso_2 \,.\, iso_1$) iters) (fmap ($iso_2 \,.\, iso_1$) k)≡For iters k

我们定义 $Free_{\mathsf{Par}}\ \rho$ a 的处理器如下:

$$
\begin{aligned}
&\mathbf{data}\ Alg_{\mathsf{Par}}\ \rho\ f = Alg_{\mathsf{Par}}\ \{h_{\mathsf{Var}} :: \forall\, a \quad .\ a \to f\ a, \\
&\qquad\qquad\qquad\qquad\qquad h_{\mathsf{For}} :: \forall\, a\ b\ .\ \rho\ (f\ b) \to (\rho\ b \to f\ a) \to f\ a\} \\
&\mathsf{fold}_{\mathsf{Par}} :: (Functor\ \rho, Pointed\ f) \Rightarrow (a \to f\ b) \to Alg_{\mathsf{Par}}\ \rho\ f \to Free_{\mathsf{Par}}\ \rho\ a \to f\ b \\
&\mathsf{fold}_{\mathsf{Par}}\ gen\ alg\ (Var\ x) \qquad = gen\ x \\
&\mathsf{fold}_{\mathsf{Par}}\ gen\ alg\ (For\ iters\ k) = h_{\mathsf{For}}\ alg\ (\mathsf{fmap}\ (\mathsf{fold}_{\mathsf{Par}}\ (h_{\mathsf{Var}}\ alg)\ alg)\ iters) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{fold}_{\mathsf{Par}}\ gen\ alg\ .\ k)
\end{aligned}
$$

此外，我们可以证明处理器h_{Par}与折叠$_{\mathsf{Par}}$之间存在同构：$\mathsf{fold}_{\mathsf{Par}}$gen alg $= h_{\mathsf{Par}}$ gen (λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) $.\ iso_1$.

$$
\begin{aligned}
&h_{\mathsf{Par}}\ gen\ (\lambda(\mathsf{For}\ iters\ k) \to h_{\mathsf{For}}\ alg\ iters\ k)\ (iso_1\ (Var\ x)) \\
&\quad \equiv h_{\mathsf{Par}}\ gen\ (\lambda(\mathsf{For}\ iters\ k) \to h_{\mathsf{For}}\ alg\ iters\ k)\ (Var_{\mathsf{H}}\ x) \\
&\quad \equiv gen\ (Var_{\mathsf{H}}\ x) \\
&\quad \equiv \mathsf{fold}_{\mathsf{Par}}\ gen\ alg\ (Var\ x)
\end{aligned}
$$

h_{Par}gen (λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) (iso_1(For iters k))≡ h_{Par}ge n (λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) (Op_{H}(For (fmap iso_1iters) (iso_1 . k)))≡(λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) (hmap 折叠$_2$(fmap (折叠 gen (λ(For iters k) $\to h_{\mathsf{For}}$alg iters k)) (For (fmap iso_1iters) (iso_1 . k))))≡ (λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) (hmap 折叠$_2$ (For (fmap iso_1 iters) (折叠 gen (λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) . iso_1 . k)))≡(λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) (For (fmap 折叠$_2$ (fmap iso_1 iters)) (折叠 gen (λ (For iters k) $\to h_{\mathsf{For}}$alg iters k) . iso_1 . k))≡ h_{For} alg (fmap 折叠$_2$ (fmap iso_1 iters)) (折叠 gen (λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) . iso_1 . k) ≡ h_{For} alg (fmap (折叠$_2$. iso_1) iters) (折叠 gen (λ(For iters k) $\to h_{\mathsf{For}}$ alg iters k) . iso_1 . k)≡ h_{For} alg (fmap (折叠$_{\mathsf{Par}}$ (h_{Var} alg) alg) iters) (折叠 gen (λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) . iso_1 . k)≡ h_{For} alg (fmap (折叠 $_{\mathsf{Par}}$ (h_{Var} alg) alg) iters) (折叠$_{\mathsf{Par}}$ gen alg . k)≡ 折叠$_{\mathsf{Par}}$ gen alg (For iters k)

我们需要一个辅助引理，说明 fold$_{\mathsf{Par}}$ (h_{Var} alg) alg $=$ fold$_2$. iso_1。

折叠$_2$ (iso_1 (Var x))
≡ 折叠$_2$ (Var_{H} x)
≡ η x

折叠$_2$ (iso_1 (For iters k)≡ 折叠$_2$ (Op_{H} (For (fmap iso_1 iters) (f map iso_1 k)))≡(λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) (hmap 折叠$_2$ (fmap 折叠$_2$(For (fmap iso_1iters) (iso_1 . k))))≡(λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) (hmap 折叠$_2$(For (fmap iso_1 iters) (折叠$_2$. iso_1 . k)))≡(λ(For iters k) $\to h_{\mathsf{For}}$alg iters k) (For (fmap 折叠$_2$(fmap iso_1iters)) (折叠$_2$. iso_1 . k))≡ h_{For} alg (fmap 折叠$_2$(fmap iso_1 it ers)) (折叠$_2$. iso_1 . k)≡ h_{For} alg (fmap (折叠$_2$. iso_1) iters) (折叠$_2$. iso_1 . k)≡ h_{For}alg (fmap (折叠$_{\mathsf{Par}}$(h_{Var}alg) alg) iters) (折叠$_{\mathsf{Par}}$(h_{Var}alg) alg . k)≡折叠$_{\mathsf{Par}}$(h_{Var}alg) alg (For iters k)

附录 C.4. Writer 效应与处理器

首先，我们定义这些同构：

iso_1 :: 函子 $\varphi \Rightarrow Free_{\mathsf{Write}}\ \varphi$ a $\rightarrow Free_{\mathsf{H}}$ ($K^{\mathsf{Write}}\ \varphi$) a$iso_1$ (Var x) = Var_{H} xiso_1 (Exec op) = Op_{H} (Exec ((iso_1 .fmap (fmap iso_1)) op))

iso_2 :: 函子 $\varphi \Rightarrow Free_{\mathsf{H}}$ ($K^{\mathsf{Write}}\ \varphi$) a $\rightarrow Free_{\mathsf{Write}}\ \varphi$ aiso_2 (Var_{H} x) = Var xiso_2 (Op_{H} (Exec op)) = Exec ((iso_2 .fmap (fmap iso_2)) op)

接下来，我们证明所需的往返性质成立，即 $iso_1 \,.\, iso_2 = id = iso_2 \,.\, iso_2$。

($iso_1 \,.\, iso_2$) (Var_{H} x)
≡ iso_1 (Var x)≡ Var_{H}
x

($iso_1 \,.\, iso_2$) (Op_{H} (Exec op))≡ iso_1 (Exec ((iso_2 .fmap (fmap iso_2)) op))
≡ Op_{H} (Exec ((iso_1 .fmap (fmap iso_1)) ((iso_2 .fmap (fmap iso_2)) op)))
≡ Op_{H} (Exec ((iso_1 . fmap (fmap iso_1) . iso_2 . fmap (fmap iso_2)) op))
≡ Op_{H} (Exec ((iso_1 . iso_2 . fmap (fmap iso_1 . fmap iso_2)) op))≡ Op_{H}
(Exec ((iso_1 . iso_2 . fmap (fmap (iso_1 . iso_2))) op))≡ Op_{H} (Exec op)

($iso_2 \,.\, iso_1$) (Var x)
≡ iso_2 (Var_{H} x)
≡ Var x

($iso_2 \,.\, iso_1$) (Exec op)≡ iso_2 (Op_{H} (Exec ((iso_1 .fmap (fmap iso_1))
op)))≡ Exec ((iso_2 .fmap (fmap iso_2)) (iso_1 .fmap (fmap iso_1))
op)≡ Exec ((iso_2 . fmap (fmap iso_2) . iso_1 . fmap (fmap iso_1))
op)≡ Exec ((iso_2 . iso_1 . fmap (fmap iso_2 . fmap iso_1)) op)≡
Exec ((iso_2 . iso_1 . fmap (fmap (iso_2 . iso_1))) op)≡Exec op

附录 C.5. 潜伏效应与处理器

首先，我们定义同构：

iso_1 :: $Free_{\mathsf{Lat}}\ \zeta\ \ell$ a $\to Free_{\mathsf{H}}$ ($K^{\mathsf{Lat}}\ \zeta\ \ell$) a$iso_1$ (Leaf x) = Var_{H} xiso_1 (Node op l sub k) = Op_{H} (Node op l (fmap (fmap iso_1) sub) (fmap iso_1 k))

iso_2 :: $Free_{\mathsf{H}}$ ($K^{\mathsf{Lat}}\ \zeta\ \ell$) 一个 $\to Free_{\mathsf{Lat}}\ \zeta\ \ell$ aiso_2 (Var_{H} x) = Leaf xiso_2 (Op_{H} (Node op l sub k)) = Node op l (fmap (fmap iso_2) sub) (fmap iso_2 k)

接下来，我们证明所需的往返性质成立，即 $iso_1 \,.\, iso_2 = id = iso_2 \,.\, iso_{2\circ}$

($iso_1 \,.\, iso_2$) (Var_{H} x)
$\equiv iso_1$ (Leaf x)
$\equiv Var_{\mathsf{H}}$ x

($iso_1 \,.\, iso_2$) (Op_{H} (Node op l sub k))$\equiv iso_1$ (Node op l (fmap (fmap iso_2) sub) (fmap iso_2 k))$\equiv Op_{\mathsf{H}}$ (Node op l (fmap (fmap iso_1) (fmap (fmap iso_2) sub)) (f map iso_1 (fmap iso_2 k)))$\equiv Op_{\mathsf{H}}$ (Node op l (fmap (fmap $iso_1 \,.\, iso_2$) sub) (fmap ($iso_1 \,.\, iso_2$) k))$\equiv Op_{\mathsf{H}}$ (Node op l sub k)

($iso_2 \,.\, iso_1$) (Leaf x)
$\equiv iso_2$ (Var_{H} x)$\equiv$
Leaf x

($iso_2 \,.\, iso_1$) (Node 运算符 左侧 子项 k)$\equiv iso_2$ (Op_{H} (Node 运算符 左侧 (fmap (fmap iso_1) 子项) (fmap iso_1 k)))$\equiv$ Node 运算符 左侧 (fmap (fmap iso_2) (f map (fmap iso_1) 子项)) (fmap iso_2 (fmap iso_1 k))$\equiv$ Node 运算符 左侧 (fmap (fmap ($iso_2 \,.\, iso_1$)) 子项) (fmap ($iso_2 \,.\, iso_1$) k)$\equiv$Node 运算符 左侧 子项 k

附录 C.6. 括号效应与处理器

首先，我们定义同构：

iso_1 :: $Free_{\mathsf{Res}}$ a $\to Free_{\mathsf{H}}\ K^{\mathsf{Res}}$ aiso_1 (Var x) = Var_{H} xiso_1 (Bracket res) = Op_{H} (Bracket (iso_1 (fmap (λ(x , y) $\to$ (iso_1 x , 返回 (iso_1 y))) res)))

iso_2 :: $Free_{\mathsf{H}}\ K^{\mathsf{Res}}$ a $\to Free_{\mathsf{Res}}$ aiso_2 (Var_{H} x) = Var xiso_2 (Op_{H} (Bracket res)) = Bracket (iso_2 (fmap (λ(x , y) $\to$ (iso_2 x iso_2 (join y))) res))

接下来，我们证明所需的往返性质成立，即 $iso_1 \,.\, iso_2 = id = iso_2 \,.\, iso_2$。

($iso_1 \,.\, iso_2$) (Var_{H} x) $\equiv$ iso_1 (Var x) $\equiv$ Var_{H} x ($iso_1 \,.\, iso_2$) (Op_{H} (Bracket res)) $\equiv$ iso_1 (Bracket (iso_2 (fmap (λ(x , y) $\to$ (iso_2 x iso_2 (join y))) res))) $\equiv$ Op_{H} (Bracket (iso_1 (fmap (λ(x , y) $\to$ (iso_1 x , return (iso_1 y))) (iso_2 (fmap (λ(x , y) $\to$ (iso_2 x iso_2 (join y))) res)))

))

$\equiv$ Op_{H} (Bracket ((iso_1 . fmap (λ(x , y) $\to$ (iso_1 x , return (iso_1 y))) .iso_2 . fmap (λ(x , y) $\to$ (iso_2 x iso_2 (join y)))) res)) $\equiv$ Op_{H} (Bracket (($iso_1 \,.\, iso_2$. fmap (λ(x , y) $\to$ (($iso_1 \,.\, iso_2$) x , (return . $iso_1 \,.\, iso_2$. join) y)))) res)) $\equiv$ Op_{H} (Bracket ((fmap (λ(x , y) $\to$ (x , (return . join) y)))) res)) $\equiv$ Op_{H} (Bracket (fmap (λ(x , y) $\to$ (x , y))) res)) $\equiv$ Op_{H} (Bracket res)

($iso_2 \,.\, iso_1$) (Var x)
$\equiv$ iso_2 (Var_{H} x)
$\equiv$ Var x

($iso_2 \,.\, iso_1$) (Bracket res) $\equiv$ iso_2 (Op_{H} (Bracket (iso_1 (fmap (λ(x , y) $\to$ (iso_1 x , return (iso_1 y))) res)))) $\equiv$ Bracket (iso_2 (fmap (λ(x , y) $\to$ (iso_2 x iso_2 (join y))) (iso_1 (fmap (λ(x , y) $\to$ (iso_1 x , return (iso_1 y))) res)))) $\equiv$ Bracket ((iso_2 . fmap (λ(x , y) $\to$ (iso_2 x iso_2 (join y))) .iso_1 . fmap (λ(x , y) $\to$ (iso_1 x , return (iso_1 y)))) res) $\equiv$ Bracket (($iso_2 \,.\, iso_1$ .fmap (λ(x , y) $\to$ (($iso_2 \,.\, iso_1$) x , (iso_2 . join . return . iso_1) y))) res) $\equiv$ Bracket ((fmap (λ(x , y) $\to$ (x , ($iso_2 \,.\, iso_1$) y))) res) $\equiv$ Bracket (fmap (λ(x , y) $\to$ (x , y)) res) $\equiv$ Bracket res