这是作者准备的版本，附有附录，并对正文做了少量补充。© 作者，2024。本论文的这一版本（扩展内容除外）经同行评审后已被 ESOP 2024 接收发表，但并非正式出版版本（由 Springer 出版的版本），也未反映录用后的改进或任何勘误。正式出版版本可在线查阅：https://doi.org/10.1007/978-3-031-57267-8_12。

arXiv:2302.13051v2 [cs.PL] 5 Apr 2024

面向通用概率编程语言的挂起分析与选择性延续传递风格

DanielLundén[1](✉)、Lars Hummelgren[2],JanKudlicka[3],Oscar Eriksson[2],和 David Broman[2,4]

[1] Oracle，瑞典斯德哥尔摩，daniel.lunden@oracle.com[2] 电气工程与计算机科学系及 Digital Futures，瑞典斯德哥尔摩，KTH 皇家理工学院，{larshum,oerikss,dbro}@kth.se[3] 数据科学与分析系，BI挪威商学院，挪威奥斯陆，jan.kudlicka@bi.no[4] 计算机科学系，斯坦福大学，美国加利福尼亚州，broman@stanford.edu

摘要。 通用概率编程语言（PPL）使统计推断问题的编码和自动求解相对容易。为解决推断问题，概率编程语言实现通常会应用依赖执行挂起的蒙特卡洛推断算法。当前最先进的解决方案通过以下方式之一实现执行挂起：（i）延续传递风格（CPS）变换，或（ii）高效但相对复杂的底层方案，而这类方案通常无法在高级语言中使用。CPS 变换会因不必要的闭包分配而产生开销——概率编程领域普遍忽视了这一问题。为降低开销，我们为概率编程语言开发了一种新颖高效的选择性 CPS 方法。具体而言，我们设计了一种新颖的静态挂起分析技术，可针对特定推断算法确定程序中需要挂起的部分。该分析使程序仅在必要之处进行选择性 CPS 变换。我们从形式上证明了该分析的正确性，并在 Miking CorePPL 编译器中实现了分析和变换。我们针对大量蒙特卡洛推断算法，在系统发育学、流行病学和主题建模领域的现实世界模型上评估了该实现。评估结果表明，在所有模型和推断算法上均取得了显著改进。

关键词： 概率编程 · 静态分析 · 续延传递风格。

1 引言

概率编程语言（PPL），例如 Anglican [49], Birch [35],WebPPL [18], Stan [10], Pyro [6], 和 Gen [11], ，使得编码并求解统计推断问题成为可能。这类推断问题在许多研究领域都备受关注，包括系统发育学 [42], 计算机视觉 [25],

主题建模 [7], 逆向图形学 [20],和认知科学 [19]。概率编程语言尤为吸引人的一个特点，是将推断问题规范（语言）与用于求解问题的推断算法（语言实现）分离开来。这种分离使概率编程语言的用户能够专注于编码推断问题，而推断算法专家则负责处理推断实现中的复杂细节。

概率编程语言实现会采用许多不同的推断算法。蒙特卡洛推断算法——例如马尔可夫链蒙特卡洛（马尔可夫链蒙特卡洛） [16]和序贯蒙特卡洛（序贯蒙特卡洛） [12]——因其渐近正确性，以及在通用[5] 概率编程语言中相对容易实现而广受欢迎。概率编程语言中所有蒙特卡洛方法背后的核心思想，都是多次执行概率程序，以生成样本 来近似编码的推断问题所对应的目标分布。然而，重复执行开销很大，因此概率编程语言实现必须避免不必要的开销。

蒙特卡洛算法通常需要挂起 执行。例如，马尔可夫链蒙特卡洛算法可以在程序中的随机抽样 处挂起，以避免在提出新的执行方案时进行不必要的重新执行；序贯蒙特卡洛算法则可以在似然更新 处挂起，以重新采样 执行。WebPPL [18] 、Anglican [49],等语言，以及 Ritchie 等人 [40], 描述的方法，会应用延续传递风格（*CPS*）变换 [3]，以便在执行期间任意挂起。CPS 变换的主要优点是，在函数式编程语言中相对容易实现。然而，CPS 变换的一个缺点是，不支持高阶函数的高性能底层语言无法支持这类变换。因此，也有一些比 CPS 更直接的底层替代方案，包括非抢占式多任务处理（例如协程 [15]）和概率编程语言控制流图 [30]。这些更直接的替代方案还可以避免 CPS[6], 带来的大部分开销，但实现起来更复杂。

我们探讨如何弥合基于 CPS 的概率编程语言与较低层级的概率编程语言之间的性能差距，后者依赖于例如直接实现协程。我们探讨 CPS 变换层面的优化，而非将基于 CPS 的概率编程语言转换为较低层级表示的过程。CPS 开销源于续延的闭包分配。我们提出一个重要观察：概率编程语言并不需要完整 CPS 变换所提供的任意挂起。大多数蒙特卡洛推断算法只需要在程序中非常特定的部分挂起。目前最先进的基于 CPS 的概率编程语言尚未考虑针对推断的挂起要求来减少 CPS 开销。

我们为概率编程语言设计了一种全新的静态挂起分析和一种全新的选择性CPS 变换，二者结合可显著降低运行时开销，相

[5] 这是 Goodman 等人最早提出的术语， [17], 用于指代表达能力强的概率编程语言，其中随机变量的数量和类型并非总能在静态时确定。[6] 请注意，只有当程序在运行时具体化续延（例如，为了挂起计算）时，CPS 才会带来开销。传统的基于 CPS 的编译器通常只在编译过程中将 CPS 用作中间形式，因此不会产生运行时开销。

较于传统的完整 CPS 转换。当前最先进的函数式概率编程语言使用 CPS 来实现执行挂起，因此能够从我们的新方法中获益良多。挂起分析会识别程序中所有可能因应用特定推断算法而需要挂起的部分。我们使用一种配备大步操作语义的核心概率编程语言演算，将挂起分析算法形式化。具体而言，难点在于如何在存在高阶函数的情况下，刻画挂起要求如何在程序中传播。此外，我们还将选择性 CPS 变换形式化，并论证在挂起分析的指导下，该变换的正确性。此前针对通用编程语言的选择性 CPS 研究，例如 Nielsen [37] 以及 Asai和 Uehara [4],的工作，侧重于基于类型系统和类型推断的分析。相比之下，我们采用 0-CFA [45] 构建挂起分析，并使其直接作用于无类型演算。

总的来说，我们（i）证明了挂起分析是正确的；（ii）表明，与使用完整 CPS 变换相比，由此得到的选择性 CPS 变换能显著提升性能；（iii）表明这一整体方法可直接应用于大量推断算法。具体而言，我们针对以下推断算法评估了该方法：似然加权、SMC 自举粒子滤波器、SMC 存活粒子滤波器 [24], 对齐轻量级 MCMC [29,48],和粒子独立 Metropolis–Hastings [39]。对于每种推断算法，我们都考察了来自系统发育学、流行病学和主题建模领域的四个现实世界模型。

我们在 Miking CorePPL [30,9]中实现了挂起分析和选择性 CPS 变换。与 WebPPL 和 Anglican 类似，该实现支持在同一程序中共存多个推断问题，并针对这些问题应用推断算法。不过，与完整 CPS 相比，使用选择性 CPS 处理此类程序更具挑战性，因为推断问题的 CPS 变换还取决于所应用的推断算法——不同的推断算法通常需要不同的挂起。更复杂的是，不同的推断问题可能共享部分代码，或者概率编程语言的用户可能对同一个推断问题应用两种不同的推断算法。此时，编译器必须对程序的不同部分应用不同的 CPS 变换，有时甚至要对程序中同一部分的不同副本应用多种不同的 CPS 变换。为解决这一问题，我们开发了一种方法，能够针对任意给定的 Miking CorePPL 程序，提取所有可能的推断问题及其对应的推断算法应用。此提取过程使选择性 CPS 能够在整个程序中得到正确应用。

总而言之，我们作出了以下贡献。

– 我们针对概率编程语言设计、形式化并证明了一种挂起分析的正确性，其中挂起要求来自给定的推断算法（第 4 节）。– 我们为概率编程语言设计并形式化了一种新的选择性 CPS 变换。与完整 CPS 相比，根据挂起分析对概率编程语言程序进行选择性 CPS 变换，可以

显著减少不必要的闭包分配所导致的运行时开销（第 5 节）。

– 我们在 Miking CorePPL 编译器中实现了挂起分析和选择性 CPS 变换。与完整 CPS 不同，选择性 CPS 对包含许多推断问题和推断算法应用的概率程序提出了挑战。我们通过提取各个推断问题（第 6 节），实现了一种能够正确地对这类程序应用选择性 CPS 的方法。

第 7 节介绍了 Miking CorePPL 中各项实现的评估及其结果，第 8 节更详细地讨论了相关工作，第 9 节则总结全文。我们首先在第 2 节考察一个动机示例，并在第 3 节介绍其底层概率编程语言演算。

2 动机示例

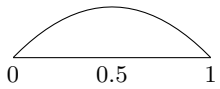
本节介绍图 1 中贯穿全文的示例，并借此阐述概率编程语言背后的基本思想，以及序贯蒙特卡洛和马尔可夫链蒙特卡洛等推断算法如何利用 CPS 挂起执行。最重要的是，我们将说明概率编程语言采用选择性 CPS 的动机和关键思想。

考虑图 1a 中以函数式概率编程语言编写的概率程序。该程序编码了一个推断问题：根据四次实验抛硬币的结果——真、真、假和真（真 = 表示正面，假 =表示反面）——估计硬币偏置的概率分布。在第 1 行，我们使用概率编程语言特有的 assume 构造来定义对硬币偏置的先验信念 a_1 。我们将该先验信念设为 Beta(2,2) 概率分布，如图 1b 所示。在图示中，0 表示总是出现假面的硬币，1 表示总是出现正面的硬币，0.5 则表示公平硬币。可以看到，我们的先验信念分布相当均匀，但公平硬币对应的概率质量更大。为了根据观测到的抛硬币结果对该先验分布进行条件化，我们从概念上无限次执行图 1a 中的程序：每次在 assume（第 1 行）处从先验贝塔分布中抽取值，并将作为副作用累积权重的乘积，这些权重作为参数传递给概率编程语言特有的 weight 构造（第 4 行）。我们使用递归函数 $iter$ 连续调用四次 weight，分别为 weight ($f_{\text{Bernoulli}}\ a_1$ true)、weight ($f_{\text{Bernoulli}}\ a_1$ true)、weight ($f_{\text{Bernoulli}}\ a_1$ false) 和 weight ($f_{\text{Bernoulli}}\ a_1$ true)[7]，。函数应用 $f_{\text{Bernoulli}}\ a_1\ o$ 给出在硬币偏置为 a_1 时，结果 o 出现的概率。也就是说，$f_{\text{Bernoulli}}\ a_1$ true $= a_1$ 和 $f_{\text{Bernoulli}}\ a_1$ false $= 1 - a_1$。例如，样本 $a_1 = 0.4$ 会得到累积权重 $0.4 \cdot 0.4 \cdot 0.6 \cdot 0.4$，而样本 $a_1 = 0.7$ 会得到累积权重 $0.7 \cdot 0.7 \cdot 0.3 \cdot 0.7$。最终得到 a_1 的无限个带权样本（程序在第 8 行返回 a_1 ），它们近似于图 1a 中的后验分布或目标分布，如图 1c 所示。注意

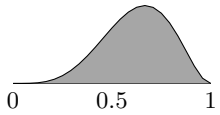
[7] 概率编程语言也常使用类似的内置函数 observe 来更新权重。例如，observe (伯努利分布 a_1) true 等价于 weight ($f_{\text{Bernoulli}}\ a_1$ true)。

```
let a1 = assume (Beta 2 2) in
递归 let iter = λobs.
  条件表达式 null obs then () else
    权重 (f_Bernoulli a1 (head obs));
    iter (tail obs)
in
iter [真, 真, 假, 真];
a1
```

(a) 程序 **t**example。



(b) Beta(2,2)。



(c) **t**example 的概率分布。

```
挂起 assume (Beta 2 2, λa1.
  递归 let iter = λobs.
    条件表达式 null obs then () else
      权重 (f_Bernoulli(a1)
            (head obs));
      iter (tail obs)
  in
  iter [真,真,假,真];
  a1)
```

(d) assume 处的挂起。

```
let a1 = assume (Beta 2 2) in
递归 let iter = λk. λobs.
  条件表达式 null obs then k ()
  else
    挂起 权重 (
      f_Bernoulli(a1) (head obs),
      (λ_. iter k (tail obs)))
in
iter (λ_. a1)
  [真,真,假,真];
```

（e）在权重处挂起。

```
let k7 = λt6.
 let k8 = λt7.
  挂起 assume (t7, λa1.
   递归 let iter = λk1. λobs.
    let k2 = λt1.
     条件表达式 t1 then k1 () else
      let k3 = λt2.
       let k4 = λt3.
        let k5 = λt4.
         挂起weight(t4, λ_.
          let k6 = λt5. iter k1 t5 in
          tail_CPS k6 obs)
        在 t2 k5 t3
       在 head_CPS k4 obs
      在 f_BernoulliCPS k3 a1
    在 null_CPS k2 obs
   在 iter (λ_. a1)
      [真, 真, 假, 真])
  在 t6 k8 2
在 Beta_CPS k7 2
```

(f) 完整 CPS。

图 1：一个用于建模硬币偏差的概率程序 **t**example 。图 (a) 给出了该程序。函数 $f_{\text{Bernoulli}}$ 是伯努利分布的概率质量函数。图 (b) 展示了 (a) 中第 1 行 a_1 对应的概率分布。图 (c) 展示了概念上无限次运行 **t**example 所得到的（加权）样本集合。图 (d) 和图 (e) 分别展示了在 assume 和 weight 处挂起所需的选择性 CPS 变换。图 (f) 给出了处于完整 CPS 形式的 **t**example ，并在 assume 和 weight 处挂起。下标 CPS 表示内置函数的 CPS 版本，例如 头部 和 尾部。

由于我们观测到了三个真结果，只有一个假结果，权重会将概率质量向 1 偏移，并使其略微变窄，因为我们现在更加确信硬币的偏差。增加实验中的抛硬币次数，会使图 1c 越来越窄。

我们可以通过运行程序大量（但有限）次，来近似无限数量的样本。这种基本的推断算法称为 似然加权。似然加权的问题在于，它对于简单模型而言才足够准确。对于复杂模型，通常只有少数似然加权样本（往往只有一个）会获得相对于其他样本大得多的权重，从而大幅降低推断准确性。现实世界模型需要更强大的推断算法，例如基于 SMC 或马尔可夫链蒙特卡洛的算法。在 SMC 和马尔可夫链蒙特卡洛中，一个关键要求是能够在调用 weight 和/或 assume 时挂起概率程序的执行。一种实现挂起的方法是使用 CPS 编写程序。我们首先在图 1d 中展示如何简单地使用 CPS 在 assume 处挂起。此时，程序会立即返回一个 Suspension$_{\texttt{assume}}$(Beta 2 2, k) 对象，表示执行在一次 assume 处停止，其参数为 Beta 2 2，并带有一个 k 续延（即抽象绑定 a_1），用于执行程序的剩余部分。使用似然加权时，我们只需从 Beta 2 2 概率分布中采样一个值 a_1，然后通过调用 $k\ a_1$ 恢复执行。该调用随后会运行程序直至终止，并得到程序的实际返回值，即 a_1。许多马尔可夫链蒙特卡洛推断算法会在 Suspension$_{\texttt{assume}}$ 处复用先前执行中的样本，因此，挂起有助于避免不必要的重新执行 [40]。

作为第二个示例，我们以图 1e 中的序贯蒙特卡洛（SMC）推断为例，说明如何在权重处挂起。这里，我们需要在递归调用 $iter$ 的中途挂起，因此用 CPS 编写程序更具挑战性。我们改写$iter$ 函数，使其将续延 k 作为参数，并在第 3 行调用续延并传入返回值 ()，而不是像图 1a 第 3 行那样直接返回 ()。这个续延参数 k 恰好使我们能够在第 5 行构造并返回 Suspension$^{\texttt{weight}}$ 对象。为说明这些挂起，考虑使用似然加权执行该程序。首先，程序返回对象 Suspension$^{\texttt{weight}}$($f_{\text{Bernoulli}(a_1)}$ true k')，其中 k' 是第 7 行构造的续延。随后，似然加权会根据值 $f_{\text{Bernoulli}(a_1)}$ true 更新此次执行的权重，并通过调用 k' () 恢复执行。同样，下一次执行会在第二次递归调用 $iter$ 时返回 Suspension$^{\texttt{weight}}$($f_{\text{Bernoulli}(a_1)}$ true k'')，我们再次更新权重，并通过调用 k'' () 恢复执行。在最后一次调用 k'''' () 使程序运行至终止并产生实际返回值 a_1 之前，我们还会遇到 Suspension$^{\texttt{weight}}$($f_{\text{Bernoulli}(a_1)}$ false k''') 和 Suspension$^{\texttt{weight}}$($f_{\text{Bernoulli}(a_1)}$ true k'''')。在 SMC 中，我们并发运行多次执行，并等待它们都返回 Suspension$^{\texttt{weight}}$ 对象。此时，我们根据权重对这些执行进行重采样（即 Suspension$^{\texttt{weight}}$中的第一个值），舍弃权重较低的执行，并复制权重较高的执行。重采样后，我们通过调用续延继续运行，直至到达下一个挂起点并再次重采样。

概率编程语言实现可通过自动的完整 CPS 变换，在 assume 和/或权重处实现挂起。图 1f 展示了对图 1a 进行此类变换的示例。我们用下标 $_{\text{CPS}}$ 标示内置函数的 CPS 版本。请注意，与图 1d 和图 1e 相比，完整 CPS 变换会产生大量额外的闭包分配。因此，运行时开销会显著增加。本文的贡献是一种静态分析，可实现对程序的自动选择性 CPS 变换，如图 1d 和图 1e 所示。通过选择性转换，我们可以避免许多不必要的闭包分配，并显著降低运行时开销，同时仍能按给定推断算法的需要实现挂起。

3 语法与语义

本节介绍用于形式化第 4 节挂起分析和第 5 节选择性 CPS 变换的概率编程语言演算。第 3.1 节给出抽象语法，第 3.2 节给出大步操作语义。第 3.3 节介绍 A-范式——这是挂起分析和选择性 CPS 变换的前提。

3.1 语法

我们以标准的无类型 λ 演算为基础，它代表了函数式通用概率编程语言，例如 Anglican、WebPPL 和 Miking CorePPL。下面定义抽象语法。

定义 1（项、值和环境）。我们定义项 $\mathbf{t} \in T$和值 $\mathbf{v} \in V$ 如下

$$\begin{aligned}\mathbf{t} ::= {} & x \mid c \mid \lambda x.\ \mathbf{t} \mid \mathbf{t}\ \mathbf{t} \mid \texttt{let}\ x = \mathbf{t}\ \texttt{in}\ \mathbf{t} \qquad \mathbf{v} ::= c \mid \langle \lambda x.\ \mathbf{t}, \rho \rangle \\ & \mid \texttt{if}\ \mathbf{t}\ \texttt{then}\ \mathbf{t}\ \texttt{else}\ \mathbf{t} \mid \texttt{assume}\ \mathbf{t} \mid \texttt{weight}\ \mathbf{t} \\ & x, y \in X \quad \rho \in P \quad c \in C \quad \{\text{false}, \text{true}, ()\} \cup \mathbb{R} \cup D \subseteq C.\end{aligned} \tag{1}$$

可数集合 X 包含变量名称、C 内置值和运算，以及 $D \subset C$ 内置概率分布。集合 P 包含求值环境，即从 X 中的变量到 V中值的映射。

定义 2（目标语言项）。作为第 *5* 节中选择性 *CPS* 变换的目标语言，我们进一步将定义 *1* 扩展为目标语言项$\mathbf{t} \in T^+$，其形式为

$$\mathbf{t} \mathrel{+}= \text{Suspension}_{\texttt{assume}}(\mathbf{t}, \mathbf{t}) \mid \text{Suspension}_{\texttt{weight}}(\mathbf{t}, \mathbf{t}). \tag{2}$$

图 1a 给出了 T中一个项的示例，而图 1d 和图 1e 给出了T^+中项的示例。不过请注意，图 1 中的程序还使用了列表构造器 `[...]`（不属于上述定义的一部分），以使示例更加有趣。

除了无类型 λ 演算中的标准变量、抽象和应用项之外，为了方便起见，我们还加入了显式的 let 表达式。

此外，我们使用语法糖递归 let $f = \lambda x.\mathbf{t}_1$ in $\mathbf{t}_2$ 来定义递归函数（将其转换为按值调用定点组合子的应用）。我们使用 $\mathbf{t}_1; \mathbf{t}_2$ 作为 $(\lambda\_.\mathbf{t}_2)\ \mathbf{t}_1$，的简写，其中 _ 表示我们不使用该参数。也就是说，我们只为副作用而求值$\mathbf{t}_1$ 。

我们纳入一组 C 内置操作和常量，它们对于以概率编程语言编码的推断问题至关重要。内置函数集合包括布尔真值、单位值、实数和概率分布。我们还可以向 C添加更多操作和常量。例如，我们可以令 $+ \in C$ 支持实数加法。为了使控制流能够依赖于内置值，我们纳入以内置布尔值作为条件的 if 表达式。

我们在第 2 节中看到了 assume 和 weight 构造的示例。assume 构造将概率分布 $D \subset C$ 作为参数，并生成服从这些概率分布的随机变量。例如，我们可以令$\mathcal{N} \in C$ 为一个用于构造正态分布的函数。那么，assume ($\mathcal{N}$0 1)，其中 $\mathcal{N}\ 0\ 1 \in D$，定义了一个服从标准正态分布的随机变量。部分构造的概率分布，例如 $\mathcal{N}\ 0$，也属于 C，但不属于 D （它们尚不是适当的概率分布）。正如我们在第 2 节中所见，weight 构造会根据作为参数给出的实数更新似然，并允许基于数据进行条件化（例如图 1 中的四次抛硬币）。

3.2 语义

我们基于 Lundén等人 [29], 构建了按值调用大步操作语义，用以描述如何对项 $\mathbf{t} \in T$进行求值。此类语义是在形式化定义与项 $\mathbf{t} \in T$ 相对应的概率分布（例如，与图 1a 中程序对应的图 1c 中的概率分布）时的关键组成部分，也是在证明概率编程语言及其推断算法的各种性质（例如推断正确性）时的关键组成部分。有关完整的形式化论述，参见例如 Borgström等人 [8]和 Lundén等人 [28] 的工作。

我们使用该语义形式化定义挂起，并利用此定义陈述第 4 节中挂起分析的可靠性（定理 1）。我们采用大步语义，因为不需要小步语义所提供的额外控制。例如，我们不关注终止的细节，因为该分析的可靠性仅与终止的执行有关。图 2 将完整语义表示为元组 $(P, T, S, \{\mathsf{false}, \mathsf{true}\}, \mathbb{R}, V)$ 上的关系 $\rho \vdash \mathbf{t}\ {}^{s}\Downarrow_{u}^{w}\ \mathbf{v}$ 。S是一组 轨迹 ，用于捕获求值期间 assume 处的随机抽样。直观而言，当且仅当 $\mathbf{t}$ 在环境 ρ 中、使用轨迹 s 并具有总概率密度（即累积权重） w求值为 $\mathbf{v}$ 时， $\rho \vdash \mathbf{t}\ {}^{s}\Downarrow_{u}^{w}\ \mathbf{v}$ 成立。本节稍后将介绍挂起标志 u 。

大多数规则都是标准的，我们重点解释与概率编程语言和挂起相关的关键性质。我们首先考虑规则（常量-应用），该规则使用 δ-函数来求值内禀操作。

定义 3（内禀元数与 δ-函数）。对于每个 $c \in C$，我们令 $|c| \in \mathbb{N}$ 表示其元数。我们还假设存在一个偏函数$\delta\colon C \times C \to C$，使得若 $\delta(c, c_1) = c_2$，则 $|c| > 0$ 且 $|c_2| = |c| - 1$。

$$\frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}\Downarrow_{u_1}^{w_1} \ \langle \lambda x.\mathbf{t}, \rho' \rangle \quad \rho \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow_{u_2}^{w_2} \ \mathbf{v}_2 \quad \rho', x \mapsto \mathbf{v}_2 \vdash \mathbf{t} \ {}^{s_3}\Downarrow_{u_3}^{w_3} \ \mathbf{v}}{\rho \vdash \mathbf{t}_1 \ \ \mathbf{t}_2 \ {}^{s_1 \| s_2 \| s_3}\Downarrow_{u_1 \vee u_2 \vee u_3}^{w_1 \cdot w_2 \cdot w_3} \ \mathbf{v}} (\textsc{App})$$

$$\frac{}{\rho \vdash x \ {}^{[]}\Downarrow_{\text{false}}^{1} \ \rho(x)} (\textsc{Var}) \qquad \frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}\Downarrow_{u_1}^{w_1} \ c_1 \quad \rho \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow_{u_2}^{w_2} \ c_2}{\rho \vdash \mathbf{t}_1 \ \ \mathbf{t}_2 \ {}^{s_1 \| s_2}\Downarrow_{u_1 \vee u_2}^{w_1 \cdot w_2} \ \delta(c_1, c_2)} (\textsc{Const-App})$$

$$\frac{}{\rho \vdash \lambda x.\mathbf{t} \ {}^{[]}\Downarrow_{\text{false}}^{1} \ \langle \lambda x.\mathbf{t}, \rho \rangle} (\textsc{Lam}) \qquad \frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}\Downarrow_{u_1}^{w_1} \ \mathbf{v}_1 \quad \rho, x \mapsto \mathbf{v}_1 \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow_{u_2}^{w_2} \ \mathbf{v}}{\rho \vdash \texttt{let} \ \ x = \mathbf{t}_1 \ \ \texttt{in} \ \ \mathbf{t}_2 \ {}^{s_1 \| s_2}\Downarrow_{u_1 \vee u_2}^{w_1 \cdot w_2} \ \mathbf{v}} (\textsc{Let})$$

$$\frac{}{\rho \vdash c \ {}^{[]}\Downarrow_{\text{false}}^{1} \ c} (\textsc{Const}) \qquad \frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}\Downarrow_{u_1}^{w_1} \ \text{true} \quad \rho \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow_{u_2}^{w_2} \ \mathbf{v}_2}{\rho \vdash \texttt{if} \ \ \mathbf{t}_1 \ \ \texttt{then} \ \ \mathbf{t}_2 \ \ \texttt{else} \ \ \mathbf{t}_3 \ {}^{s_1 \| s_2}\Downarrow_{u_1 \vee u_2}^{w_1 \cdot w_2} \ \mathbf{v}_2} (\textsc{If-True})$$

$$\frac{\rho \vdash \mathbf{t} \ {}^{s}\Downarrow_{u}^{w} \ d \quad w' = f_d(c)}{\rho \vdash \texttt{assume} \ \ \mathbf{t} \ {}^{s \| [c]}\Downarrow_{suspend_{\texttt{assume}} \vee u}^{w \cdot w'} \ c} (\textsc{Assume}) \quad \frac{\rho \vdash \mathbf{t} \ {}^{s}\Downarrow_{u}^{w} \ w'}{\rho \vdash \texttt{weight} \ \ \mathbf{t} \ {}^{s}\Downarrow_{suspend_{\texttt{weight}} \vee u}^{w \cdot w'} \ ()} (\textsc{Weight})$$

图 2：$\mathbf{t} \in T$的大步操作语义。为简洁起见，我们省略了规则 (If-False)；它与 (If-True) 类似。环境 $\rho, x \mapsto \mathbf{v}$ 表示在 ρ的基础上扩展一个针对 x的绑定 $\mathbf{v}$。对于每个 $d \in D$，函数 f_d 是其概率密度或概率质量函数。例如,—$f_{\mathcal{N}(0,1)}(x) = e^{x^2/2}/\sqrt{2\pi}$是标准正态分布的密度函数。我们使用以下记法：$\|$表示序列拼接，$\cdot$ 表示乘法，$\vee$ 表示逻辑析取。

例如，$\delta((\delta(+,1)),2) = 3$。我们利用内置函数的元数性质来形式化定义轨迹。

定义 4（轨迹）。对于所有 $s \in S$，s 都是一个由元数为 *0* 的内置函数组成的序列，称为轨迹。我们用 $s = [c_1, c_2, \ldots, c_n]$ 表示一个含有 n 个元素的轨迹 s。

规则 (Assume) 对随机抽样进行了形式化，并消耗轨迹中的元素。具体而言，(Assume) 会根据传递给 assume 的分布，使用轨迹首个元素的概率密度 w' 更新求值的总概率密度 $w \in \mathbb{R}$。此外，规则 (Weight) 会根据 weight 参数直接修改总概率密度。

现在我们来考虑推导 $\rho \vdash \mathbf{t} \ {}^{s}\Downarrow_{u}^{w} \mathbf{v}$ 中的特殊挂起标志 u。

定义 5（挂起要求）。当挂起标志 u 为真时，推导 $\rho \vdash \mathbf{t} \ {}^{s}\Downarrow_{u}^{w} \mathbf{v}$ 需要挂起。

例如，规则 (App) 在 $u_1 \vee u_2 \vee u_3$ 时要求挂起——也就是说，任何子推导都要求挂起时。为体现 SMC 和马尔可夫链蒙特卡洛推断中的特定挂起要求，我们将挂起要求的来源限制为 assume 和 weight。我们通过图 2 中的布尔变量 $suspend_{\texttt{assume}}$ 和 $suspend_{\texttt{weight}}$，分别开启或关闭这些来源。本文其余部分的示例中，我们令 $suspend_{\texttt{weight}} =$ 为 true，$suspend_{\texttt{assume}} =$ 为 false（即只有 weight 要求挂起，如 SMC推断中所示）。

```
1let t1 = 2 in2let t2 = 2
in3let t3 = Beta in4let
 t4 = t3 t1 in5let t5 = t4 t2
in6let a1 = assume t5 in 7let
rec iter = λobs.8 let
 t6 = null in9 let t7 = t6 obs
in10 let t8 =11 条件表达式 t7
then12 let t9 = () in13 t9
14 else15 let
 t10 = fBernoulli in16 let
 t11 = t10 a1 in17 let
 t12 = head in18 let
 t13 = t12 obs in
19 let t14 = t11 t13 in
20 let w1 =权重 t14 in
21 let t15 = tail in
22 let t16 = t15 obs in
23 let t17 = iter t16 in
24 t17
25 in
26 t8
27 在
28let t18 = 真 在
29let t19 = 假 在
30let t20 = 真 在
31let t21 = 真 在
32let t22 = [t21,t20,t19,t18] 中
33let t23 = iter t22 in
34 a1
```

图 3：图 1a 中的运行示例 $\mathbf{t}^{\text{example}}$ 转换为 A 范式。

为阐释语义，再次考虑图 1a 的 $\mathbf{t}^{\text{example}}$。由于 $\mathbf{t}^{\text{example}}$恰好计算一个 assume，$\mathbf{t}^{\text{example}}$ 的唯一有效轨迹是单元素轨迹 $[a_1]$，其中 $a_1 \in \mathbb{R}_{[0,1]}$，这是由于 a_1的贝塔先验所致。首先将ρ 设置为空环境 $\varnothing$，然后按照图 2 中的规则，我们推导出 $\varnothing \vdash$

$$\mathbf{t}^{\text{example}} \overset{[a_1]}{\Downarrow} f_{\text{Beta}(2,\ 2)} \quad \overset{(a_1)\ \cdot a_1^{3}}{1} \quad -a_1)$$

为真 a_1. 请注意，每次对 $\mathbf{t}^{\text{example}}$ 求值时，$u =$都为真，因为求值过程中总会调用 weight 四次。也就是说，该推导需要挂起。然而，对 $\mathbf{t}^{\text{example}}$ 的许多子推导不需要挂起。例如，子推导 assume (Beta 2 2) 和空观测 就不需要（即，$u =$ 为假）。第 4 节介绍了一种挂起分析，用于保守地近似哪些子推导需要挂起。例如，该分析可用于图 1e 中的选择性 CPS 变换。

3.3 A-范式

为简化第 4 节中的挂起分析和第 5 节中的选择性 CPS 变换，我们要求项采用A-范式（ANF） [13]。

定义 6（A-范式）。我们将 A-范式项 $\mathbf{t}_{ANF} \in T_{ANF}$ 定义如下。

$$\begin{aligned} \mathbf{t}_{\text{ANF}} &::= x \mid \texttt{let}\ x = \mathbf{t}'_{\text{ANF}}\ \texttt{in}\ \mathbf{t}_{\text{ANF}} \\ \mathbf{t}'_{\text{ANF}} &::= x \mid c \mid \lambda x.\ \mathbf{t}_{\text{ANF}} \mid x\ y \\ &\quad \mid \texttt{if}\ x\ \texttt{then}\ \mathbf{t}_{\text{ANF}}\ \texttt{else}\ \mathbf{t}_{\text{ANF}} \mid \texttt{assume}\ x \mid \texttt{weight}\ x \end{aligned} \tag{3}$$

该性质成立：$T_{\text{ANF}} \subset T$。此外，存在标准变换，可将 T 中的项转换为 T_{ANF}。图 3 展示了将图 1a 转换为 ANF 的结果。第 4 节和第 5 节将以图 1a 为贯穿示例。

将程序限制为 A 范式，可以显著简化挂起分析和选择性 CPS 变换。从现在起，我们要求所有变量

在程序中的绑定都是唯一的；结合 A 范式，这意味着程序中的每个表达式 $\mathbf{t} \in T_{\mathrm{ANF}}$ 都由 let 表达式中的一个变量名称唯一标记。这一性质对于第 4 节中的处理至关重要。

4 挂起分析

本节介绍主要的技术贡献：挂起分析。分析的目标是识别按照定义 5 的含义可能需要挂起的程序表达式。识别这类表达式可引出第 5 节中的选择性 CPS 变换，从而实现图 1e 所示的变换。

挂起分析建立在 0-CFA 算法 [45,38], 之上，我们依据 Lundén等人 [29]对算法进行了形式化。我们要解决的主要挑战是如何在存在高阶函数的情况下对挂起的传播进行建模。0-CFA 中的 0 表示 上下文无关性——该分析在一个全局上下文中考察程序的每个部分。与 k-CFA 等上下文敏感方法相比，上下文无关性使分析更加保守；在这类方法中，$k \in \mathbb{N}$ 表示上下文敏感性的级别 [32]。我们使用 0-CFA 有两个原因：(i) 该分析的最坏情况时间复杂度为多项式，而 k-CFA 的时间复杂度为指数级，且从 $k = 1$起便是如此；(ii) 在实际的概率编程语言应用中，0-CFA 的局限性鲜少造成影响。例如，对于第 7 节中的程序，k-CFA 并不比 0-CFA 更有优势。

我们假设 $\langle \lambda x.\, \mathbf{t}\ \rho \rangle \notin C$（请回忆，$C$ 是内置函数的集合）。也就是说，我们假设闭包不属于内置函数。特别地，这禁止使用内置操作（包括使用 assume d、$d \in D \subset C$）生成闭包；否则会在没有任何收益的情况下不必要地使分析复杂化。

考虑图 3 中的程序，并假设权重需要挂起。显然，第 20 行由 w_1 标记的表达式随后需要挂起。此外，w_1 作为第 10 行由 t_8 标记的更大表达式的一部分进行求值。因此，对 t_8 的求值也需要挂起。另外，t_8 作为第 7 行对绑定抽象 obs 的应用的一部分进行求值。特别地，绑定抽象 obs 绑定到 $iter$，而我们在第 23 行和第 33 行应用 $iter$。因此，由 t_{17} 和 t_{22} 命名的表达式需要挂起。总之，我们得到 w_1、t_8、t_{17} 和 t_{22} 需要挂起，并且还注意到，对绑定抽象 obs 的所有应用都需要挂起。

接下来我们将进行形式化，并首先介绍标准的抽象值。

定义 7（抽象值）。我们将抽象值$\mathbf{a} \in A$ 定义为$\mathbf{a} ::= \lambda x.y \mid$常量$_x\, n$，其中 x、$y \in X$ 和 $n \in \mathbb{N}$。

抽象值 $\lambda x.y$ 表示在运行时程序中，例如由项 $\lambda x.$let y = 1 in y in 产生的所有闭包（回想一下，我们假设变量 x 和 y 唯一）。注意，y 表示函数体返回的名称（由算法 1 中的名称函数形式化定义）。抽象值 $\mathsf{const}_x\, n$ 表示所有源自 x、元数为 n 的内置函数。例如，$\mathsf{const}_x\ 2$ 源自例如项 let x = + in $\mathbf{t}$。

算法 1 挂起分析的约束生成。我们将
算法本身的函数式风格伪代码用无衬线字体书写，以便与
T 中的项区分开来。

函数 生成约束(**t**): $T_{\mathrm{ANF}} \to \mathcal{P}(R)$ =

```
1  匹配 t，条件表达式为
2  | x → ∅
3  | let x =t1 在 t2 →
4    生成约束(t2) ∪
5      匹配 t1 与
6      | y → {S_y ⊆ S_x}
7      | c → 条件表达式 |c| > 0 则 {常量_x |c| ∈ S_x}
8              否则 ∅
9      | λy.t_b → 生成约束(t_b)
10       ∪ {λy. 名称 t_b ∈ S_x}
11       ∪ {suspend_n ⇒ suspend_y
12               | n ∈suspendNames(t_b)}
13     | lhs rhs → {
14         ∀z∀y λz.y ∈ S_lhs
15           ⇒ (S_rhs ⊆ S_z) ∧ (S_ y ⊆ S_x),
16         ∀y∀n 常量_y n ∈ S_lhs ∧ n > 1
17           ⇒ 常量_y n − 1 ∈ S_x,
18         ∀y λy._ ∈ S_lhs
19           ⇒ (suspend_y ⇒ suspend_x),
20         ∀y 常量_y _ ∈ S_lhs
21           ⇒ (suspend_y ⇒ suspend_x),
22         suspend_x ⇒
23           (∀y λy._ ∈ S_lhs ⇒ suspend_y)
24           ∧(常量_y _ ∈ S_lhs ⇒ suspend_y)
25       }
26     | assume _ →
27       条件表达式 suspend assume 那么 {suspend_x} 否则 ∅
28
29     | 权重 _ →
30       条件表达式 suspend 权重 那么 {suspend_x}否则 ∅
31   |条件表达式 y 那么 t_t 否则 t_e →
32     生成约束(t_t)
33       ∪ 生成约束(t_e)
34       ∪ {S_NAME t_t ⊆ S_x, S_NAME t_e ⊆ S_x}
35       ∪ {suspend_n ⇒ suspend_x
36           | n ∈ 挂起名称(t_t)
37                 ∪ 挂起名称(t_e)}
38
39 函数 名称(t): T_ANF → X =
40   匹配 t:
41   | x → x
42   | let x = t1 中 t2 → 名称(t2)
43
44 函数 suspendNames(t): T_ANF → P(X) =
45   匹配 t:
46   | x → ∅
47   | let x = t1 在 t2 →
48     suspendNames(t2) ∪
49       匹配 t1 与
50       | lhs rhs → {x}
51       | 条件表达式 y 则 t_t 否则 t_e → {x}
52       | assume _ →
53         条件表达式 suspend assume 则 {x} 否则 ∅
54       | 权重 _ →
55         条件表达式 suspend 权重 则 {x}否则 ∅
56       | _ → ∅
```

分析中的核心对象是集合 $S_x \in \mathcal{P}(A)$，以及对所有 $x \in X$ 而言的布尔值 挂起$_x$。集合 S_x 包含所有可能流向标记为 x 的表达式的抽象值，而 挂起$_x$ 表示该表达式是否需要挂起。一个简单却无用的解是 $S_x = A$，并且对程序中的所有变量 x，挂起$_x$ = 均为 true。为了获得更精确的挂起信息，我们希望找到满足 S_x 和 挂起$_x$ 的更小解。

为了形式化 S_x 和 挂起$_x$ 的可靠解集，我们为程序生成约束 $\mathbf{c} \in R$（约束的正式定义见附录 A.1）。算法 1 对程序$\mathbf{t} \in T_{\mathrm{ANF}}$ 所需的约束进行了形式化，其中辅助函数 generateConstraints 通过递归遍历程序 **t** 来生成约束集。由于采用 A 范式，顶层匹配（第 1 行）只有两种情况：变量不会生成任何约束，而关键情况是第 3–30 行的 let 表达式。该算法使用辅助函数名称（第 39 行）来确定 ANF 表达式的名称，并使用挂起名称函数 suspendNames（第 44 行）来确定表达式中所有可能挂起的顶层表达式的名称（即应用、if 表达式以及 assume 和/或 weight）。

接下来，我们通过考察约束集 generateConstraints(**t**^{example}) 来说明并阐释生成的约束，其中 **t**^{example} 是图 3 中的程序的

一部分。许多约束都是标准的，因此我们重点关注本文引入的新挂起约束。具体而言，难点在于正确捕获挂起要求在函数应用和高阶函数之间的传播。首先可以看到，定义别名（第 6 行）会生成形如 $S_y \subseteq S_x$ 的约束，常量会引入常量抽象值（例如 $\mathsf{const}_{t_6} 1 \in S_{t_6}$），而 assume 和 weight 会引入挂起要求，例如 挂起$_{w_1}$（挂起$_{w_1}$ = true 的简写）。

首先，我们考虑通过算法 1 第 9–12 行的情形，为 λ观测值.（图 3 第 7 行）生成的约束。为使示例简单，在分析中我们将尚未展开的递归 let 视为普通 let（对于这个特定示例，分析结果不受影响）。省略为抽象体递归生成的约束后，生成的约束如下：

$$\{\lambda obs.\, t_8 \in S_{iter}\} \cup \{suspend_n \Rightarrow suspend_{obs} \mid n \in \{t_7, t_8\}\}. \tag{4}$$

第一个约束是标准约束，表示抽象值 λ观测值.t_8 流向S_{iter}，因为命名该观测值表达式的变量在图 3 第 26 行是 t_8（由于分栏换行，这一点不容易注意到）。其余约束是新的，用于建立挂起要求的传播。具体而言，如果抽象体中由顶层 let 绑定的任何表达式需要挂起，那么该抽象观测值 本身也需要挂起。为提高效率，我们只为可能挂起的表达式建立依赖关系（由算法 1 中的 suspendNames 形式化定义）。请注意，这里我们不添加约束 挂起$_{w_1}$ ⇒挂起$_{obs}$，因为 w_1不在观测值的函数体顶层。相反，我们随后添加约束 挂起$_{w_1}$ ⇒ 挂起$_{t_8}$，而挂起$_{w_1}$ ⇒ 挂起$_{obs}$ 则可由传递性推出。

在算法 1 中，通过第 31–37 行的 case 处理第 10 行绑定到 t_8 的条件表达式时生成的约束如下（省略递归生成的约束）

$$\begin{aligned}&\{S_{t_9} \subseteq S_{t_8}, S_{t_{17}} \subseteq S_{t_8}\} \\ &\qquad \cup \{suspend_n \Rightarrow suspend_{t_8} \mid n \in \{t_{11}, t_{13}, t_{14}, w_1, t_{16}, t_{17}\}\}.\end{aligned} \tag{5}$$

前两个约束是标准约束，表示两个分支结果中的抽象值都会流向结果 S_{t_8}。最后一组约束是新增的，与抽象挂起约束类似。这些约束表明，两个分支中所有位于顶层且需要挂起的表达式，也会导致 t_8 需要挂起。

考虑图 3 第 23 行的应用。算法 1 第 13–25 行的条件表达式所生成的约束为

$$\begin{aligned}\{\ &\forall z \forall y\ \ \lambda z.y \in S_{iter} \Rightarrow (S_{t_{16}} \subseteq S_z) \wedge (S_y \subseteq S_{t_{17}}), \\ &\forall y \forall n\ \ \mathtt{const}_y\, n \in S_{iter} \wedge n > 1 \Rightarrow \mathtt{const}_y\, n-1 \in S_{t_{17}}, \\ &\forall y\ \ \lambda y.\_ \in S_{iter} \Rightarrow (suspend_y \Rightarrow suspend_{t_{17}}), \\ &\forall y\ \ \mathtt{const}_y\ \ \_ \in S_{iter} \Rightarrow (suspend_y \Rightarrow suspend_{t_{17}}), \\ &suspend_{t_{17}} \Rightarrow (\forall y\ \ \lambda y.\_ \in S_{iter} \Rightarrow suspend_y) \\ &\qquad\qquad\qquad \wedge (\forall y\ \ \mathtt{const}_y\ \ \_ \in S_{iter} \Rightarrow suspend_y)\ \}.\end{aligned} \tag{6}$$

前两个约束是标准约束，说明抽象值如何作为应用的结果流动。后三个约束是新的，涉及挂起。第三个和第四个约束说明，如果需要挂起的抽象或内置函数流向 $iter$，则应用的结果 t_{17} 也需要挂起。第五个约束说明，如果结果 t_{17} 需要挂起，那么流向 $iter$ 的所有抽象和常量都需要挂起。严格来说，后续证明分析的可靠性（定理 1）并不要求最后一个约束；但是，正如我们将在第 5 节看到的，它是选择性 CPS 变换所必需的。

我们通过附录 A.1 中的算法 3 找到了约束的一个解。该算法根据约束传播抽象值，直到达到不动点，这一过程相当标准。不过，我们扩展了该算法，以支持新的挂起约束。该算法是一个函数 analyzeSuspend:$T_{\mathrm{ANF}} \to ((X \to \mathcal{P}(A)) \times \mathcal{P}(X))$。该函数返回一个映射 data : $X \to \mathcal{P}(A)$，为所有 S_x 分配抽象值集合，并返回一个集合 suspend : $\mathcal{P}(X)$，当且仅当 $x \in$ suspend 时，将 $suspend_x =$ 赋值为 true。重要的是，对 S_x 和 suspend$_x$ 的赋值满足所有生成的约束。为说明该算法，下面给出 analyzeSuspend(**t**^{example}) 的分析结果：

$$
\begin{gathered}
S_{iter} = \{\lambda obs.t_8\} \quad S_{t_6} = \{\texttt{const}_{t_6} 1\} \quad S_{t_{10}} = \{\texttt{const}_{t_{10}} 2\} \\
S_{t_{11}} = \{\texttt{const}_{t_{10}} 1\} \quad S_{t_{12}} = \{\texttt{const}_{t_{12}} 1\} \quad S_{t_{15}} = \{\texttt{const}_{t_{15}} 1\} \\
S_n = \varnothing \mid \text{all other } n \in X \\
suspend_n = \text{true} \mid n \in \{obs, w_1, t_8, t_{17}, t_{22}\} \\
suspend_n = \text{false} \mid \text{all other } n \in X.
\end{gathered} \tag{7}
$$

上述结果证实了我们之前的推理：标记为 obs、w_1、t_8、t_{17} 和 t_{22} 的表达式可能需要挂起。

现在我们考虑该分析的可靠性。首先，0-CFA 的可靠性已得到充分确立（例如，参见 Nielson 等人 [38]），并且这一结论可扩展到我们的新约束；因此，我们将以下引理视为无需证明即可成立。

引理 1（0-CFA 可靠性）。对于每个**t** $\in T_{ANF}$，*analyzeSuspend*(**t**) 给出的 S_x 和 *suspend*$_x$, $x \in X$ 的解，都满足约束 *generateConstraints*(**t**)。

接下来，我们必须证明这些约束本身是可靠的。考虑对任意项**t** $\in T_{\mathrm{ANF}}$的求值。对于**t**的每个子推导（由于采用 A 范式，其标签为名称 x），如果该子推导需要挂起，则必须满足$suspend_x =$ 为真。否则，该分析就不可靠。定理 1 对可靠性作了形式化描述。请注意，该分析是保守的（即不完备的），因为即使 x 的子推导不需要挂起，它也可能判定$suspend_x = true$ 。

定理 1（挂起分析可靠性）。令 **t** $\in T_{ANF}$, $s \in S$, $u \in$*{false, true}*, $w \in \mathbb{R}$, 且 **v** $\in V$，使得 $\varnothing \vdash$ **t** $^{s}\Downarrow_{u}^{w}$ **v**。现在，令 S_x 和 *suspend*$_x$，其中 $x \in X$ 根据 *analyzeSuspend*(**t**) 得出。对于 ($\varnothing \vdash$ **t** $^{s}\Downarrow_{u}^{w}$ **v**) 的每个子推导 ($\rho \vdash$ `let` $x =$ **t**$_1$ `in` **t**$_2$ $^{s_1 \| s_2}\Downarrow_{u_1 \vee u_2}^{w_1 \cdot w_2}$ **v**′)，$u_1 =$ *true* 意味着 *suspend*$_x =$ *true*。

算法 2 选择性续延传递风格转换。我们定义 $\mathbf{t}_{\mathrm{id}} = \lambda x.x$。项 c_{CPS} 是 c 的 CPS 版本。我们用函数式风格的伪代码表示该算法本身，以无衬线字体将它与项区分开来

在 T 中。

函数 $\mathrm{CPS}(\mathsf{vars}, \mathbf{t})$: $\mathcal{P}(X) \times T_{\mathrm{ANF}} \to T^{+} =$

```
返回 CPS′(t_id, t)

函数 CPS′(续延,t): T × T_ANF → T⁺ =
  匹配 t
  | x → 条件表达式 续延 = t_id 则 t 否则 续延 t
  | let x = t1 在 t2 →
    let t′2 = CPS′(续延,t2) 于
    匹配 t1 与
    | y → let x = t1 在 t′2
    | c → let x =
              (条件表达式 x ∈ 变量 则 c_CPS 否则 c) 于 t′2
    | λy.t_b →
      let t′1 = 条件表达式 y ∈变量
        则 λk.λy. CPS′(k,t_b)
        否则 λy. CPS′(t_id,t_b)
      in
      let x = t′1 英寸 t′2
    | lhs rhs →
      条件表达式 x ∈ 变量 则
        条件表达式 尾调用(t)
        则 lhs 续延 rhs
        else lhs (λx.t′2) rhs
      否则 let x = t1 其中 t′2

  |条件表达式 y 则 t_t 否则 t_e →
    条件表达式 x ∈变量 则
      条件表达式尾调用(t) 则
        条件表达式 y 则 cps′(续延, t_t)
        否则 cps′(续延, t_e)
      else
        let k = λx.t′2中
        条件表达式 y 则 cps′(k, t_t) 否则 cps′(k, t_e)
    否则 let x = 条件表达式 y 则 cps′(t_id,t_t)
                  否则 cps′(t_id,t_e) 于 t′2
  | assume y → let x = t1 在 t′2
    条件表达式 x ∈ 变量 则
      条件表达式 尾调用(t)
      则 挂起 assume (y, 续延)
      否则 挂起 assume (y,λx.cps′(续延, t2))
    否则let x = t1 in t′2
  | 权重 y → let x = t1 条件表达式 t′2
    条件表达式 x ∈ 变量 then
      条件表达式 尾调用(t)
      then 挂起 权重 (y, 续延)
      否则挂起 权重 (y,λx.cps′(续延, t2))
    否则 let x = t1 in t′2

函数 尾调用(t): T_ANF → {假, 真} =
  匹配 t, 其中
  | let x = _ in x →true
  | _ → false
```

证明。由附录 A.2 中的引理 2 及 $\rho = \varnothing$ 可直接得出。

引理 2 的证明使用引理 1，并对推导$\varnothing \vdash \mathbf{t} \ {}^{s}\Downarrow_{u}^{w} \mathbf{v}$进行结构归纳。

接下来，我们使用挂起分析对程序进行选择性 CPS 变换。

5 选择性 CPS 转换

本节介绍第二项技术贡献：选择性 CPS 转换。转换本身是标准的，难点在于如何正确利用挂起分析结果进行选择性转换。

算法 2 是完整算法。使用 A 范式中的项作为输入可显著降低算法的复杂度。主函数 cps 的输入包括一组变量 vars: $\mathcal{P}(X)$，用于指示哪些表达式需要进行 CPS 转换，以及一个待转换的程序 $\in T_{\mathrm{ANF}}$。新增的 vars 参数使该转换区别于标准 CPS 转换。就本文而言，我们始终使用 vars $= \{x \mid \text{挂起}_x = \text{true}\}$，其中 挂起_x 来自 analyzeSuspend(t)。也可以使用 vars $= X$ 进行标准的完整 CPS

```
1let t1 = 2 in
2let t2 = 2 in
3let t3 = Beta in
4let t4 = t3 t1 in
5let t5 = t4 t2 in
6let a1 = assume t5 in
7递归 let iter = λk. λobs.
8   let t6 = null in
9   let t7 = t6 obs in
10  条件表达式 t7 then
11    let t9 = () in
12    t9
13  else
14    let t10 = f_Bernoulli 中
15    let t11 = t10 a1 中
16    let t12 = head 中
17    let t13 = t12 obs 中
18    let t14 = t11 t13 中
19    挂起          权重 (t14,
20      λ_.
21        let t15 = tail in
22        let t16 = t15 obs in
23        iter k t16)
24 中
25let t18 = 真 in
26let t19 = 假 in
27let t 20 =真 in
28let t21 = 真 in
29let t22  = [t21,t20,t19,t18] 其中
30let k′  = λ_. a1 其中
31 iter k′ t22
```

图 4：图 3 中的运行示例经过选择性 CPS 变换后的结果。该程序在语义上等价于图 1e。

转换（例如图 1f），或为其他应用领域使用其他变量集合 vars。cps 函数返回的值是一个类型为 T^+ 的（非 A 范式）项。第 1 行首次调用的辅助函数 cps′ 以一个延续项 cont 作为输入，该项表示要在尾位置应用的续延。初始时，该延续项为 $\mathbf{t}_{\mathrm{id}}$，表示没有续延。与算法 1 类似，第 4 行的顶层匹配有两种情形：变量的简单情形（第 5 行）以及 let 表达式的复杂情形（第 6–49 行）。为支持尾调用优化，辅助函数 tailCall 用于判断 A 范式表达式是否为尾调用（即形如 let x = $\mathbf{t}'$ in x）。

现在，我们通过计算 cps(vars$_{\text{example}}$, $\mathbf{t}^{\text{example}}$) 来说明算法 2，其中 vars$_{\text{example}} = \{obs\ w_1,\ t_8,\ t_{17},\ t_{22}\}$ 来自（7），而 $\mathbf{t}^{\text{example}}$ 来自图 3。图 4 展示了最终结果。首先请注意，变换不会改变 vars^{example} 中未标记名称的表达式，因为这些表达式不需要挂起。因此，接下来我们只关注经过变换的表达式。首先，考虑图 3 第 7 行定义的抽象obs，它由算法 2 第 12 行的情形处理。由于 obs ∈属于 vars^{example}，我们对抽象应用标准 CPS 变换：为抽象添加一个延续参数，并使用该延续递归地变换函数体。接下来，考虑图 3 第 20 行的权重表达式 w_1 的变换，它由算法 2 第 44 行的情形处理。该表达式不在尾位置，因此我们构造一个包含后续 let 表达式的新延续，递归地变换该延续的函数体，然后将最终结果包装在一个 Suspension 对象中。图 3 第 10 行的 if 表达式 t_8 由算法 2 第 28 行的情形处理，且处于尾位置（其后直接是返回 t_8）。因此，我们递归地变换两个分支。最后，考虑图 3 第 23 行和第 33 行的应用 t_{17} 和 t_{22}，它们由算法 2 第 18 行的情形处理。应用 t_{17} 处于尾位置，我们通过将当前延续作为参数添加进去来变换它。位置 t_{22} 处的应用

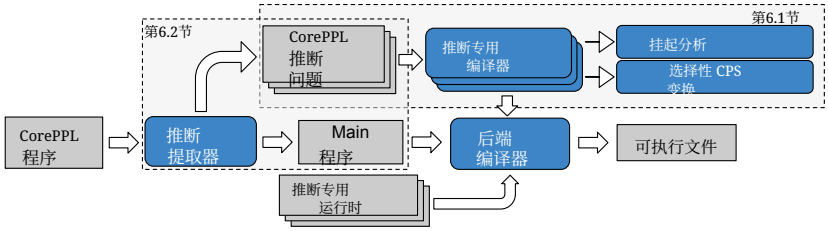


图 5：Miking CorePPL 编译器实现概览。我们将整个编译器分为两部分：（i）挂起分析与选择性 *CPS*（第6.1节）以及（ii）推断问题提取 （第6.2节）。图中以灰色矩形框表示制品，以蓝色圆角框表示变换单元和库。请注意，推断提取器 变换将程序分成两条不同的路径，这两条路径在推断专用编译之后重新合并。白色继承箭头（指向挂起分析 和 选择性 *CPS* 变换）表示这些库在推断专用编译器变换中使用。

不在尾位置，因此我们构造一个返回最终值a_1 的延续 k' （图 3 第 34 行），然后将其作为参数添加到该应用中。

算法 2 不一定会产生正确结果。具体而言，对于所有应用$lhs\ rhs$，我们必须确保：（i）如果对该应用进行 CPS 变换，也必须对可能出现在lhs处的所有抽象进行 CPS 变换；（ii）如果不 对该应用进行 CPS 变换，就不 得对可能出现在lhs处的任何抽象进行 CPS 变换。我们通过参数 vars 控制这一点。特别是，根据挂起分析为 vars 赋值可产生正确结果。为此，请再次考虑算法 1 第 13–25 行中的应用约束，并注意，如果任何需要挂起的抽象或内禀操作出现在lhs处，则挂起$_x$ = true。此外，最后一条应用约束确保，如果挂起$_x$ = true，那么出现在lhs 处的所有 抽象和内禀操作都需要挂起。因此，对于所有 $\lambda y.\ \_$ 和 $const_y\ \_$，要么挂起$_y$ = true，要么挂起$_y$ = false。

6 实现

我们在 Miking CorePPL [30]，中实现了挂起分析和选择性 CPS 变换。Miking CorePPL 是在领域特定语言构造框架 Miking [9] 中实现的核心概率编程语言。我们选择在 Miking CorePPL 中实现，而非采用其他基于 CPS 的概率编程语言，因为该语言实现包含现成的 0-CFA 基础实现，可简化挂起分析的实现。图 5 展示了 CorePPL 编译器的组织结构。输入是一个 CorePPL 程序，其中可能包含许多推断问题和推断算法的应用，这与 WebPPL 和 Anglican 类似。输出是由某个 Miking 后端编译器生成的可执行文件。第6.1节

介绍挂起分析和选择性 CPS 的实现细节，尤其是与第 3 节核心演算的差异。第 6.2 节介绍推断提取器，以及它与选择性 CPS 结合时的工作方式。挂起分析、选择性 CPS 变换和推断提取的实现总计约 1500 行代码（这是本文的一项贡献）。代码可在 GitHub [2] 上获取。

6.1 挂起分析与选择性 CPS

Miking CorePPL 在定义 1 的抽象语法基础上，扩展了标准函数式数据结构和特性，例如代数数据类型（记录、元组和变体）、列表及模式匹配。Miking CorePPL 中的挂起分析和选择性 CPS 实现扩展了算法 1 和算法 2，以支持这些语言特性。此外，与图 2 中的挂起$_{\texttt{weight}}$ 和挂起$_{\texttt{assume}}$ 相比，该实现允许任意配置挂起源。具体而言，该实现将这种任意配置与 Lundén 等人 [29]提出的对齐分析结合使用。对于对齐版本的序贯蒙特卡洛和马尔可夫链蒙特卡洛推断算法，这种组合允许选择性地将 CPS 变换应用于一部分 assume 或权重处的挂起。

Miking CorePPL 还提供了一个用于实现推断算法的框架。具体而言，要实现新的推断算法，用户需要实现推断专用编译器 和推断专用运行时。图 5 展示了不同的编译器和运行时。每个推断专用编译器都会应用挂起分析和选择性 CPS 变换，以满足相应推断算法的特定挂起要求。

接下来，我们将展示 Miking CorePPL 如何处理包含多个推断问题的程序，这些推断问题由不同的推断算法求解。

6.2 推断问题提取

图 5 展示了推断提取编译器的处理流程。首先，编译器对输入程序应用推断提取器。结果是一组推断问题，以及一个包含剩余胶水代码的主程序。其次，编译器对每个推断问题应用推断专用编译器。最后，编译器将主程序和经过编译的推断问题与推断专用运行时相结合，并将结果提供给后端编译器。

请看图 6a 中的示例。我们在第 7–10 行定义了函数 m，用一次 assume 调用和一次 observe 调用构造一个最简推断问题（修改执行权重的方式与 weight 类似）。该函数将初始概率分布 d 和数据点 y 作为输入。我们通过第 12–16 行的 infer 构造，对该推断问题应用对齐的轻量级MCMC推断。infer 的第一个参数指定推断算法的配置，第二个参数则是推断问题。推断问题是 thunk（即带有虚设 unit 参数的函数）。我们通过将 m 应用于一个均匀分布来构造推断问题 thunk

```
1 mexpr2let 数据 = [3 248, 4 34
0, 42 2, 96 7, 9 2, 85 2, 41 7, 53 4, 85
6, 45 4.5 ] in67let m = lam d. lam
y. lam.8 let x = assume in9
observe y(Gaussian x 0.1);10
xin11 12let d0 =13
infer(LightweightMCMC14
{iterations = 100,15 aligned =
true}) 16 (m(Uniform 0.0 4.0)
1.0) in17
18 =
19 递归 let
repeat lam data. lam d.20 条件表达式
data with [y] ++data then21 let
posterior =22 推断 (BPF{particles = 100})
23 (m d y) in24 repeat
data posterior25 else d26let d1 =
repeat data d0 in27条件表达式
distEmpiricalSamples d128with (样本, 权
重) in29 迭代30 (lam s.31 print32
(concat (float2string s) "\n"))33 样本
```

(a) Miking CorePPL 程序。

```
1let m = lam d. lam y. lam.2 let
x = assume d in3 observe y (高
斯分布 x 0.1);4 x in5m (均匀分布
0.0 4.0) 1.0 ()
```

（b）从（a）第 13 行提取的推断问题。

```
1let m = lam d. lam y. lam.2 let
x =assume d in3 observe y (高
斯分布 x 0.1);4 x in5m d y ()
```

(c) 从 (a) 第 22 行提取的推断问题。

图 6：(a) 中的 Miking CorePPL 程序示例，其中有两处非平凡的 infer 用法。(b) 和 (c) 分别展示了从 (a) 的第 13 行和第 22 行提取并经过选择性 CPS 转换的推断问题。编译器会在后续阶段处理 (c) 中的自由变量 d 和 y。

初始分布和数据点 1。0。推断结果 d0 是另一个概率分布，我们将其用作递归 repeat 函数（第 19–24 行）中的第一个初始分布。该函数反复使用 SMC 自举粒子滤波器执行推断（第 21–23 行），并再次使用函数 m 构造推断问题序列。每次 infer 应用都会将上一次迭代得到的结果分布用作初始分布，并从数据序列中消耗数据点。我们在第 29–33 行提取并打印最终结果分布 d1 中的样本。当前提取方法的一个局限是，我们尚不支持嵌套推断。

编译器设计中的一个关键挑战，是如何在同一个概率程序中处理不同的推断算法。具体而言，推断算法需要不同的选择性 CPS 变换，并将其应用于代码的不同部分。为了分别处理推断算法，我们对 infer 应用采用 Hummelgren 等人的提取方法 [22]，为 infer 的每次出现生成独立的推断问题。尽管编译器设计主要涉及相当全面的工程工作，但在变换和组合不同代码实体时，必须特别注意处理名称绑定这一非平凡问题。例如，编译器必须对图 6b 进行选择性 CPS 变换，使其在 assume 处挂起（这是 MCMC 所要求的）

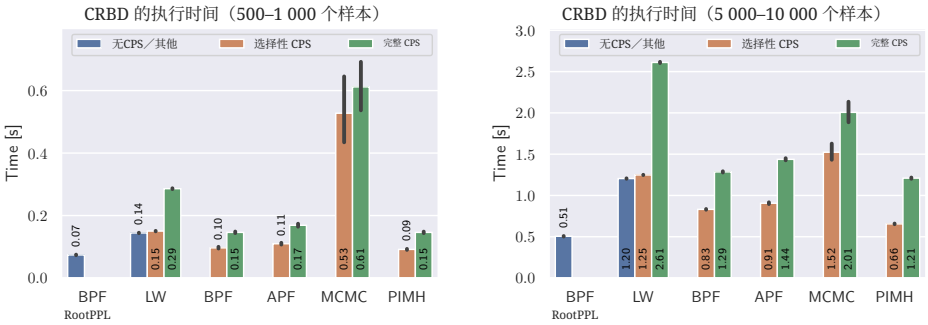
，并对图 6c 进行选择性 CPS 变换，使其在 observe 处挂起（这是 SMC 所要求的）。我们设计了一种稳健且模块化的解决方案，从而可以轻松添加新的推断算法，而无需担心名称冲突。

7 求值

本节介绍挂起分析和选择性 CPS 实现的评估。我们的主要论点是：（i）与传统的完整 CPS 相比，选择性 CPS 方法显著提升了性能；（ii）这一优势适用于一系列重要的推断算法，这些算法在真实的推断问题上进行了评估。我们使用了 Miking 基准测试代码库中的四种概率编程语言模型及其对应的数据集，该代码库可在 GitHub 上获取 [1]。这些模型包括：第 7.1 节中的恒定速率出生-死亡模型（CRBD）、第 7.2 节中的分支发生多样化率转变（ClaDS）、第 7.3 节中的潜在狄利克雷分配（LDA），以及第 7.4 节中的媒介传播疾病（VBD）。所有模型都具有重要意义，并在不同研究领域得到积极应用：CRBD 和 ClaDS 用于进化生物学和系统发育学 [36,42,31], ；LDA 用于主题建模 [7],；VBD 用于流行病学 [14,33]。除了 Miking 基准测试中的 Miking CorePPL 模型外，我们还分别用 WebPPL 和 Anglican 实现了 CRBD。

我们在 Miking CorePPL 中加入了多种支持选择性 CPS 的常用推断算法。首先是标准的似然加权（LW），如第 2 节所述。LW 并不严格要求使用 CPS，但我们在权重处挂起以实现它，从而凸显无 CPS、选择性 CPS 和完整 CPS 之间的差异。LW 可以直接衡量 CPS 的开销，因为该算法只需反复执行程序。在 LW 中，权重处挂起也有助于提前停止权重为 0 的执行（即无用的样本）。不过，为了隔离 CPS 对执行时间的影响，我们不采用提前停止。接下来，我们加入自举粒子滤波器（BPF）和存活粒子滤波器（APF）。这两种都是在权重处挂起以便重采样执行的序贯蒙特卡洛算法。BPF 是一种常用于概率编程语言的标准算法，而 APF 是一种相关算法，由 Kudlicka 等人 [24] 在概率编程语言领域提出。我们加入的最后两种推断算法是对齐轻量级 MCMC（简称 MCMC）和粒子独立 Metropolis–Hastings。对齐轻量级 MCMC [29] 是对 Wingate 等人 [48], 提出的标准概率编程语言 Metropolis–Hastings 方法的扩展，并在部分 assume 调用处挂起。粒子独立 Metropolis–Hastings（PIMH）是一种 MCMC 算法，它在 Metropolis–Hastings MCMC 算法 [39] 中反复使用 BPF（在权重处挂起）。我们的研究范围仅限于单核 CPU 推断。

除了 Miking CorePPL 中的推断算法外，我们还使用了三种用于 CRBD 的其他先进概率编程语言：Anglican、WebPPL，以及专为 Miking CorePPL 打造的高性能 RootPPL 编译器 [30]。对于 Anglican，我们采用似然加权、BPF 和粒子独立 Metropolis–Hastings 推断。对于 WebPPL，我们采用 BPF 和（非对齐的）轻量级马尔可夫链蒙特卡洛。对于 RootPPL 版本的 Miking CorePPL，我们采用 BPF 推断（这是唯一受支持的推断算法）。



	Anglican 似然加权	Anglican BPF	WebPPL BPF	WebPPL 马尔可夫链蒙特卡洛
1000 个样本	11.6 ± 0.36 秒	5.65 ± 2.71 秒	2.42 ± 0.20 秒	1.42 ± 0.07 秒
10000 个样本	90.4 ± 2.12 秒	29.1 ± 2.35 秒	53.9 ± 4.03 秒	3.10 ± 0.77 秒

图 7：CRBD 模型的平均执行时间。误差棒表示 95% 置信区间（使用 Seaborn 条形图中的选项（'ci', 95））。表格显示标准差。

我们为每个模型考虑两种配置：1000 个和 10000 个样本。CRBD 和 ClaDS 是例外；为使推断准确度与相应的 BPF 相当，我们将 APF 的样本数调整为 500 和 5000。每项实验运行 300 次（另加一次预热运行），并测量执行时间（不包括编译时间）。为说明作为编译器组成部分的挂起分析和选择性 CPS 变换的效率，我们在此指出，这两者合计在所有模型上的运行时间仅为 1–5 毫秒。

这些实验并未比较不同推断算法的性能。要进行这样的比较，还需要考虑在给定执行时间下推理结果的准确度。不同推断算法与模型的组合，其准确度差异很大。我们分别评估了选择性 CPS 和完整 CPS 在单个推断算法中的执行时间。选择性 CPS 仅是一种执行时间优化，不会改变算法本身及其准确度（我们在附录 B 中针对似然加权、BPF 和 APF 验证了这一点）。

对于 Miking CorePPL，我们使用 OCaml 4.12.0 作为第 6 节实现的后端编译器，并使用 GCC 7.5.0 作为独立 RootPPL 编译器。我们使用了 Anglican 1.1.0（OpenJDK 11.0.19）和 WebPPL 0.9.15（Node.js 16.18.0）。我们在搭载 64 GB 内存的英特尔至强 Gold 6148 CPU 上，使用 Ubuntu 18.04.6 运行实验。

7.1 恒定速率出生-死亡

CRBD 是一种多样化模型，进化生物学家使用它来推断被称为 系统发育树 的物种群体观测进化树中出生率和死亡率的分布。对于 CRBD 实验，我们使用翠鸟科系统发育树（翠鸟，54 个现存物种） [42,23]。我们将 CRBD 与

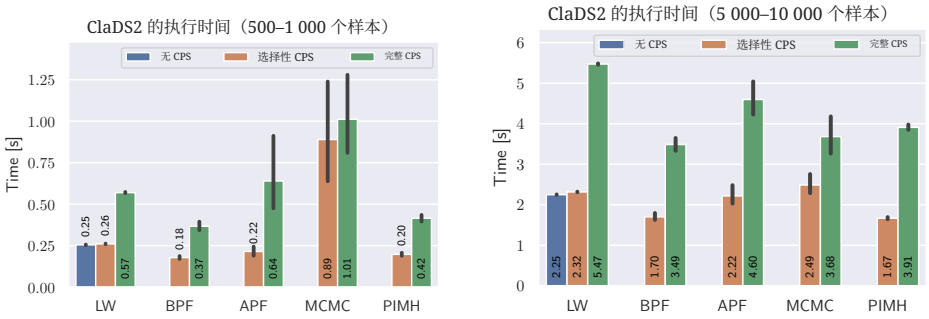


图 8：ClaDS模型的平均执行时间。误差线表示 95% 置信区间（在 Seaborn 的条形图中使用选项（'ci'，95））。

Miking CorePPL（55 行代码）、Anglican（129 行代码）和 WebPPL（66 行代码）进行比较。源代码见附录B.1。实验总执行时间为 9 小时。

图 7 展示了结果。我们注意到，在所有情况下，选择性 CPS 都比完整 CPS 更快。与完整 CPS 不同，对于似然加权（LW），选择性 CPS 相比无 CPS 的开销微乎其微。早期马尔可夫链蒙特卡洛（MCMC）样本的执行时间对初始条件很敏感，因此，与其他算法相比，MCMC 的结果方差更大。当我们将样本数增加到 10000 时，方差会减小。除 WebPPL 中的 MCMC 外，Anglican 和 WebPPL 的执行时间都比 Miking CorePPL 中相应算法慢一个数量级。不过请注意，此比较仅供参考，并不完全公平，因为与 Miking CorePPL 相比，Anglican 和 WebPPL 使用了不同的执行环境。最后，我们注意到，采用选择性 CPS 的 Miking CorePPL BPF 实现并不比将 Miking CorePPL 编译为 RootPPL BPF 慢多少；后者是一款专为提升效率而设计的编译器（但也存在其他限制，例如不支持垃圾回收）。RootPPL 不使用 CPS，而是通过一种利用概率编程语言控制流图概念的底层转换来实现挂起 [30]。

7.2 分支形成多样化率转变

ClaDS 是进化生物学中使用的另一种多样化模型 [31,42]。与 CRBD 不同，它允许出生率和死亡率随时间变化。我们再次使用翠鸟科系统发育树。完整源代码（72 行代码）见附录B.2。整个实验的执行时间为 3 小时。图 8 展示了结果。值得注意的是，在所有情况下，选择性 CPS 都比完整 CPS 快。

7.3 潜在狄利克雷分配

潜在狄利克雷分配 [7] 是自然语言处理中用于将文档归类为主题的一种模型。我们使用了一个规模与该数据

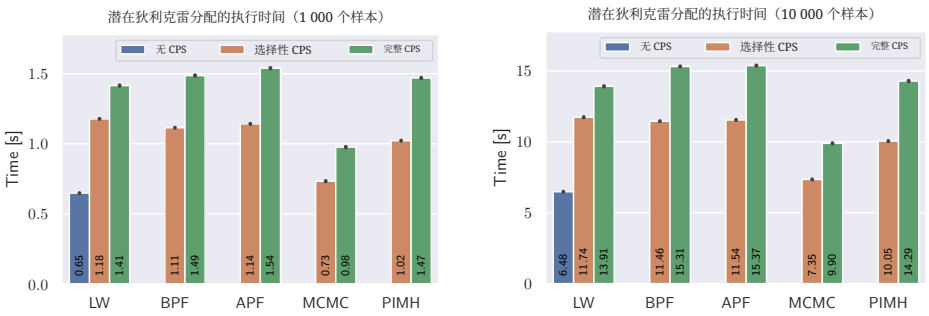


图 9：LDA 模型的平均执行时间。误差线表示 95% 置信区间（使用 Seaborn 条形图中的选项（'ci'，95））。

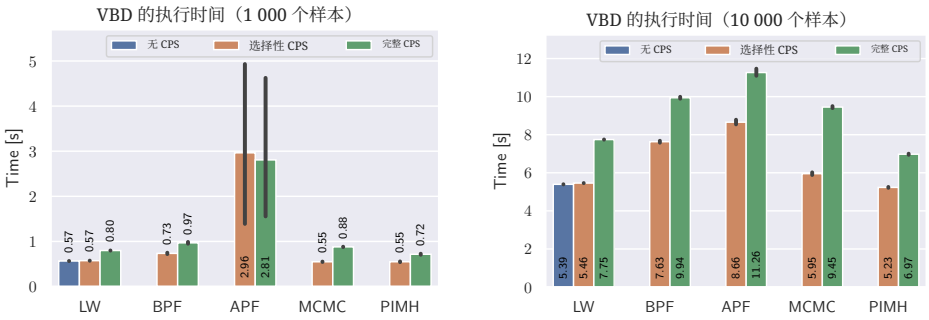


图 10：VBD 模型的平均执行时间。误差线表示 95% 置信区间（在 Seaborn 的条形图中使用选项（'ci', 95））。

集相当的合成数据集，具体参见 Ritchie 等人 [40]：包含 100个词的词汇表、10个主题和 25份观测文档（每份包含 30 个词）。我们不采用任何优化技术，例如折叠吉布斯采样 [21]。因此，即使对于小规模数据集，使用概率编程语言解决推断问题也已经颇具挑战性。完整的源代码（26 行代码）见附录B.3。实验总执行时间为 12 小时。

图 9 展示了结果。我们注意到，在所有情况下，选择性CPS都比完整 CPS 更快。有趣的是，与完整 CPS 相比，似然加权的开销降低并不那么显著。原因在于，对于该模型，在权重处挂起需要对计算量最大的递归应用 CPS 变换。

7.4 媒介传播疾病

我们采用 Funk等人 [14] 以及后来 Murray等人 [33]提出的 VBD 模型。背景是密克罗尼西亚的一次登革热暴发，以及疾病在蚊子和人类之间的传播。推断问题是：根据卫生中心每天报告的新病例数量，求出每天易感者、暴露者、感染者和康复者（SEIR模型）的真实人数。完整源代码（140 行）见附录B.4。总执行时间为 8 小时。

图 10 展示了结果。我们再次注意到，除 APF 在 1 000 个样本时看起来例外外，所有情况下选择性 CPS 都比完整 CPS 快。这很可能是统计异常，因为 APF 在 1000 个样本时的方差相当大。与 BPF 相比，APF 使用重采样方法；当样本数过少时，其执行时间会有很大波动 [24]。图中清楚地显示，与 1 000 个样本相比，10 000 个样本时的方差已降低至与 BPF 相当的水平。总之，求值结果表明，对于通用概率编程语言，选择性 CPS 相较于完整 CPS 具有明显优势。

8 相关工作

有一些通用概率编程语言需要进行非平凡的挂起。其中一种语言是 Anglican [49], , 它使用 CPS 解决挂起问题。Anglican 执行完整 CPS 转换，但有一个例外：某些静态已知的函数称为原语过程，它们包括一部分常规 Clojure 函数（Anglican 的宿主语言），这些函数保证不会执行概率编程语言代码，因此 Anglican 不会对它们进行 CPS 转换 [46]。然而，Clojure 库中的高阶函数不能是原语过程，因此 Anglican 必须手动重新实现这类函数（例如 map 和 fold）。Anglican 不考虑对概率编程语言代码进行选择性 CPS 转换，而是始终对 Anglican 程序中的概率编程语言部分执行完整 CPS 转换。

WebPPL [18] 和 Ritchie 等人提出的方法 [40] 也使用 CPS 变换来实现概率编程语言推断。不过，他们没有考虑选择性 CPS 变换。Ścibior 等人 [43]提出了一种基于单子和单子变换器的概率函数式编程库架构设计（以及 Ścibior 等人 [44]中的相应理论）。特别是，他们使用协程单子变换器来挂起序贯蒙特卡洛推断。这种方法与我们的方法相似，因为它利用高阶函数式语言特性来实现挂起。不过，他们没有考虑选择性转换。

概率编程语言 Pyro [6], Stan [10,5],Gen [11,27], 和 Edward [47] 要么实现不需要挂起的推断算法（例如哈密顿蒙特卡罗），要么以这样的方式限制语言，使得挂起成为显式操作，并且可以由语言实现轻松处理。例如，Pyro 中的序贯蒙特卡洛 [8] 以及较新版本的 Birch 要求用户将程序显式编写为步进函数，由序贯蒙特卡洛实现反复调用。重采样只发生在步进调用之间，因此挂起也就变得十分简单。

通用选择性 CPS 变换方面的研究包括 Nielsen [37],Asai 和 Uehara [4], Rompf 等人 [41],以及 Leijen [26]。他们研究的是有类型语言，而本文研究的是无类型语言。Nielsen 的早期工作 [37]探讨了如何通过选择性 CPS 变换高效实现 call/cc。该变换需要手动添加用户注释，而本文的方法则是全自动的。较新的一种方法由 Asai 和 Uehara 提出 [4], , 他们通过选择性 CPS 变换，利用 shift 和 reset 高效实现分隔延续——

[8] 请注意，Pyro 中的主要推断算法是随机变分推断，该算法不需要挂起。

——shift 和 reset。与我们的方法类似，他们会自动确定程序中需要进行选择性 CPS 变换的位置。他们采用基于类型推断的方法，而我们的方法则基于 0-CFA。Rompf 等人 [41] 针对 Scala 采用了与 Asai 和 Uehara [4], 类似的方法，但还需要用户注释。Leijen [26] 则使用类型导向的选择性 CPS 变换来编译代数效应处理器。

在概率编程语言中，除了 CPS 之外，还有一些用于实现挂起的底层替代方案。具体而言，有多种语言和方法直接实现了对非抢占式多任务处理（例如协程）的支持。Turing [15]和较早版本的 Birch [35,34] 实现了协程，以支持任意挂起，但没有详细讨论其实现。Lundén等人 [30] 引入并使用概率编程语言控制流图这一概念，将 Miking CorePPL 编译到低层 C++ 框架 RootPPL。编译器显式插入代码来维护特殊的执行调用栈，使其与 C++ 的隐式调用栈相互独立。这种实现带来了出色的性能，但既不支持垃圾回收，也不支持高阶函数。另一种底层方法由 Paige和Wood [39], 提出，他们利用互斥锁和 fork系统调用来挂起并重新采样序贯蒙特卡洛执行。理论上，如果能高效实现，上述许多 CPS 底层替代方案都能凭借更精细的底层控制，将开销降至最低。不过，与 CPS 转换相比，这些方法需要投入多得多的实现工作。相比之下，选择性 CPS 变换是一种出人意料地简单、抽象层次高且易于实现的替代方案，能够使 CPS 的开销更接近底层方法。

9 结论

本文提出了一种选择性 CPS 变换，用于实现概率编程语言中的执行挂起。为支持该变换，我们开发了一种静态挂起分析，用于确定由于推断算法的挂起要求而需要进行 CPS 变换的程序部分。我们在 Miking CorePPL 中实现了挂起分析、选择性 CPS 变换，以及推断问题提取过程（该过程是选择性 CPS 变换所需的）。此外，我们还使用系统发育学、主题建模和流行病学领域的现实世界模型对该实现进行了评估。结果表明，与标准的完整 CPS 挂起方法相比，该实现针对大量蒙特卡洛推断算法显著提升了速度。

致谢。 本项目获得瑞典战略研究基金会（FFL15-0032 和 RIT15-0012）的资助、瑞典研究理事会（资助编号：2018-04329）的部分资助，以及 Digital Futures（DLL 项目）的资助。本研究也是 KTH皇家理工学院 Vinnova可信边缘计算系统与应用能力中心的一部分。我们感谢 Gizem Çaylak 提供潜在狄利克雷分配的实现，并感谢 Viktor Senderov 提供 ClaDS 的实现。

数据可用性声明。 本文附有一个用于支持求值的资料包：
https://zenodo.org/doi/10.5281/zenodo.10454311。

参考文献

1. Miking基准测试套件。https://github.com/miking-lang/miking-benchmarks（2023 02 2 2023 01），访问日期：--。Miking DPPL。https://github.com/miking-lang/miking-dppl（2023），访问日期：2023-01-02 3. Appel, A.W.：《带续延的编译》。剑桥大学出版社（1991）4. Asai, K., Uehara, C.：用于shift和reset的选择性 CPS 变换。载于：ACM SIGPLAN部分求值与程序操作研讨会论文集。第40–52页。计算机协会（2017）5. Baudart, G., Burroni, J., Hirzel, M., Mandel, L., Shinnar, A.：将 Stan 编译为生成式概率语言，并扩展到深度概率编程。载于：第42届ACM SIGPLAN编程语言设计与实现国际会议论文集。第497–510页。计算机协会（2021）6. Bingham, E., Chen, J.P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., Goodman, N.D.：Pyro：深度通用概率编程。《机器学习研究杂志》**20**(28)，1–6（2019）7. Blei, D.M., Ng, A.Y., Jordan, M.I.：潜在狄利克雷分配。《机器学习研究杂志》**3**，993–1022（2003）8. Borgström, J., Dal Lago, U., Gordon, A.D., Szymczak, M.：通用概率编程的 λ 演算基础。载于：第21届ACM SIGPLAN 函数式编程国际会议论文集。第33–46页。计算机协会（2016）9. Broman, D.：Miking 愿景：交互式程序化建模、声音语言组合与自学习编译。载于：第12届ACM SIGPLAN软件语言工程国际会议论文集。第55–60页。计算机协会（2019）10. Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., Riddell, A.：Stan：一种概率编程语言。《统计软件杂志》，论文 **76**(1)，1–32（2017）11. Cusumano-Towner, M.F., Saad, F.A., Lew, A.K., Mansinghka, V.K.：Gen：一种具备可编程推断功能的通用概率编程系统。载于：第40届ACM SIGPLAN编程语言设计与实现国际会议论文集。第221–236页。计算机协会（2019）12. Doucet, A., de Freitas, N., Gordon, N.：《序贯蒙特卡洛方法实践》。信息科学与统计，施普林格纽约（2001）13. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.：带续延编译的本质。载于：ACM SIGPLAN 1993年编程语言设计与实现会议论文集。第237–247页。计算机协会（1993）14. Funk, S., Kucharski, A.J., Camacho, A., Eggo, R.M., Yakob, L., Murray, L.M., Edmunds, W.J.：登革热与寨卡病毒疫情的比较分析揭示了不同环境与病毒之间的差异。《PLOS被忽视的热带病》**10**(12)，1–16（2016）15. Ge, H., Xu, K., Ghahramani, Z.：Turing：一种用于灵活概率推断的语言。载于：第二十一届人工智能与统计国际会议论文集。第84卷，第1682–1690页。机器学习研究论文集（2018）

16. Gilks, W., Richardson, S., Spiegelhalter, D.：《马尔可夫链蒙特卡洛实践》。Chapman & Hall/CRC 跨学科统计学，Taylor & Francis（1995）17. Goodman, N.D., Mansinghka, V.K., Roy, D., Bonawitz, K., Tenenbaum, J.B.：Church：一种用于生成模型的语言。载于：第二十四届人工智能不确定性会议论文集。第220–229页。AUAI Press（2008）18. Goodman, N.D., Stuhlmüller, A.：概率编程语言的设计与实现。http://dippl.org（2014 31 19 2022 10），访问日期：--。Goodman, N.D., Tenenbaum, J.B., Contributors, T.P.：《认知概率模型》。http://probmods.org/v2 10 20 2016 2022 06（），访问日期：--。Gothoskar, N., Cusumano-Towner, M., Zinberg, B., Ghavamizadeh, M., Pollok, F., Garrett, A., Tenenbaum, J., Gutfreund, D., Mansinghka, V.：3DP3：基于概率编程的三维场景感知。载于：神经信息处理系统进展。第34卷，第9600–9612页。Curran Associates, Inc.（2021）21. Griffiths, T.L., Steyvers, M.：发现科学主题。《美国国家科学院院刊》**101**(suppl_1)，5228–5235（2004）22. Hummelgren, L., Wikman, J., Eriksson, O., Haller, P., Broman, D.：表达式加速：带类型的高级语言的无缝并行化。arXiv 电子预印本，第 arXiv:2211.00621 页（2022）23. Jetz, W., Thomas, G.H., Joy, J.B., Hartmann, K., Mooers, A.O.：鸟类在时空中的全球多样性。《自然》**491**(7424)，444–448（2012）24. Kudlicka, J., Murray, L.M., Ronquist, F., Schön, T.B.：利用带延迟采样的存活粒子滤波器对进化出生—死亡模型进行概率编程。载于：人工智能不确定性会议（2019）25. Kulkarni, T.D., Kohli, P., Tenenbaum, J.B., Mansinghka, V.：Picture：一种用于场景感知的概率编程语言。载于：2015年IEEE计算机视觉与模式识别会议（CVPR）。第4390–4399页（2015）26. Leijen, D.：行类型代数效应的类型引导编译。载于：第44届ACM SIGPLAN编程语言原理研讨会论文集。第486–499页。计算机协会（2017）27. Lew, A.K., Matheos, G., Zhi-Xuan, T., Ghavamizadeh, M., Gothoskar, N., Russell, S., Mansinghka, V.K.：SMCP3：基于概率程序提议的序贯蒙特卡洛。载于：第26届国际人工智能与统计会议论文集。第7061–7088页。机器学习研究论文集（2023）28. Lundén, D., Borgström, J., Broman, D.：概率编程语言中序贯蒙特卡洛推断的正确性。载于：编程语言与系统。第404–431页。Springer 国际出版公司（2021）29. Lundén, D., Çaylak, G., Ronquist, F., Broman, D.：高阶概率编程语言中的自动对齐。载于：编程语言与系统。第535–563页（2023）30. Lundén, D., Öhman, J., Kudlicka, J., Senderov, V., Ronquist, F., Broman, D.：采用高效并行序贯蒙特卡洛推断编译通用概率编程语言。载于：编程语言与系统。第29–56页。Springer 国际出版公司（2022）31. Maliet, O., Hartig, F., Morlon, H.：一种利用许多小幅变化估算物种特异性多样化速率的模型。《自然·生态与进化》**3**(7)，1086–1092（2019）32. Midtgaard, J.：函数式程序的控制流分析。《ACM计算调查》**44**(3)（2012）

33. Murray, L., Lundén, D., Kudlicka, J., Broman, D., Schön, T.：概率程序的延迟采样与自动 Rao-Blackwell 化。载于：第二十一届人工智能与统计国际会议论文集。第 84 卷，第 1037–1046 页。机器学习研究论文集（2018）34. Murray, L.M.：惰性对象复制：基于总体的概率编程平台。arXiv 电子预印本，第 arXiv:2001.05293 页（2020）35. Murray, L.M., Schön, T.B.：使用概率编程语言 Birch 进行自动化学习。《控制年度评论》**46**，29–43（2018）36. Nee, S.：宏观进化中的出生—死亡模型。《生态学、进化与系统学年度评论》**37**(1)，1–17（2006）37. Nielsen, L.R.：一种选择性 CPS 变换。《理论计算机科学电子札记》**45**，311–331（2001）38. Nielson, F., Nielson, H.R., Hankin, C.：《程序分析原理》。Springer-Verlag（1999）39. Paige, B., Wood, F.：概率编程语言的编译目标。载于：第 31 届机器学习国际会议论文集。第 32 卷，第 1935–1943 页。机器学习研究论文集（2014）40. Ritchie, D., Stuhlmüller, A., Goodman, N.：C3：利用延续和调用点缓存实现概率程序的轻量级增量化 MCMC。载于：第 19 届人工智能与统计国际会议论文集。第 51 卷，第 28–37 页。机器学习研究论文集（2016）41. Rompf, T., Maier, I., Odersky, M.：通过类型导向的选择性 CPS 变换实现一等多态分隔延续。载于：第 14 届 ACM SIGPLAN 函数式编程国际会议论文集。第 317–328 页。计算机协会（2009）42. Ronquist, F., Kudlicka, J., Senderov, V., Borgström, J., Lartillot, N., Lundén, D., Murray, L., Schön, T.B., Broman, D.：通用概率编程为统计系统发育学提供了强有力的方法。《生物学通讯》**4**(1)，244（2021）43. Ścibior, A., Kammar, O., Ghahramani, Z.：用于模块化贝叶斯推断的函数式编程。《ACM 编程语言论文集》**2**(ICFP)（2018）44. Ścibior, A., Kammar, O., Vákár, M., Staton, S., Yang, H., Cai, Y., Ostermann, K., Moss, S.K., Heunen, C., Ghahramani, Z.：高阶贝叶斯推断的指称语义验证。《ACM 编程语言论文集》**2**(POPL)（2017）45. Shivers, O.G.：高阶语言的控制流分析，或驯服 lambda。卡内基梅隆大学（1991）46. Tolpin, D., van de Meent, J.W., Yang, H., Wood, F.：概率编程语言 Anglican 的设计与实现。载于：第 28 届函数式编程语言实现与应用研讨会论文集。计算机协会（2016）47. Tran, D., Kucukelbir, A., Dieng, A.B., Rudolph, M., Liang, D., Blei, D.M.：Edward：用于概率建模、推断和批评的库。arXiv 电子预印本，第 arXiv:1610.09787 页（2016）48. Wingate, D., Stuhlmueller, A., Goodman, N.：通过变换式编译实现概率编程语言的轻量级实现。载于：第十四届人工智能与统计国际会议论文集。第 15 卷，第 770–778 页。机器学习研究论文集（2011）

30 D.Lundén 等人。

49. Wood, F., Meent, J.W., Mansinghka, V.：一种新的概率编程推断方法。载于：第十七届人工智能与统计国际会议论文集。第 33 卷，第 1024–1032 页。机器学习研究论文集（2014）

A 挂起分析，续

本节提供有关挂起分析的更多细节。具体而言，A.1 节描述挂起分析算法，A.2 节给出定理 1 的证明。

A.1 算法

在介绍算法之前，我们先对约束作出形式化定义。

定义 8（约束）。我们将约束 $\boldsymbol{c} \in R$ 定义如下。

$$
\begin{aligned}
\mathbf{c} ::= \ & \mathbf{a} \in S_x \mid S_x \subseteq S_y \mid \mathbf{a} \in S_x \Rightarrow \mathbf{a} \in S_y \\
\mid \ & suspend_x \mid suspend_x \Rightarrow suspend_y \\
\mid \ & \forall x \forall y \ \ \lambda x.y \in S_{lhs} \Rightarrow (S_{rhs} \subseteq S_x) \wedge (S_y \subseteq S_{app}) \\
\mid \ & \forall x \forall n \ \ (\texttt{const}_x \, n \in S_{lhs}) \wedge (n > 1) \Rightarrow \texttt{const}_x \, n-1 \in S_{app} \\
\mid \ & \forall x \ \ \lambda x.\_ \in S_{lhs} \Rightarrow (suspend_x \Rightarrow suspend_{res}) \\
\mid \ & \forall x \ \ \lambda x.\_ \in S_{lhs} \Rightarrow suspend_x \\
\mid \ & \forall x \ \ \texttt{const}_x \ \ \_ \in S_{lhs} \Rightarrow (suspend_x \Rightarrow suspend_{res}) \\
\mid \ & \forall x \ \ \texttt{const}_x \ \ \_ \in S_{lhs} \Rightarrow suspend_x \\
\mid \ & suspend_{res} \Rightarrow (\forall x \ \ \lambda x.\_ \in S_{lhs} \Rightarrow suspend_x) \\
& \qquad\qquad \wedge (\forall x \ \ \texttt{const}_x \ \ \_ \in S_{lhs} \Rightarrow suspend_x) \\
x, y, lhs, rhs, app, res \in X.
\end{aligned}
\tag{8}
$$

算法 3 是完整的挂起分析。该算法使用算法 1 生成的工作列表和约束，在整个程序中传播抽象值，直到达到不动点。具体而言，该算法会传播新的挂起相关约束。

A.2 正确性证明

引理 2 直接推出定理 1。

引理 2（挂起分析可靠性）。设$\mathbf{t}' \in T_{ANF}$ 是$\mathbf{t}$的一个子项， $\rho \in P$, $s \in S$, $u \in \{$ *false, true*$\}$,$w \in \mathbb{R}$， 且$\mathbf{v} \in V$满足

$$
\rho \vdash \mathbf{t}' \ {}^{s}\Downarrow_{u}^{w} \ \mathbf{v}. \tag{9}
$$

且对于每个 $x \in X$,

(C1) 若 $\rho(x) = \langle \lambda y.\mathbf{t}_y, \rho_y \rangle$，则 $\lambda y.$名称$(\mathbf{t}_y) \in S_x$，且 ρ_y 也满足 ***(C1)***。那么，

(R1) 对于每个子推导 $(\rho' \vdash \texttt{let} \ \ x = \mathbf{t}_1 \ \texttt{in} \ \mathbf{t}_2 \ {}^{s_1 \| s_2}\Downarrow_{u_1 \vee u_2}^{w_1 \cdot w_2} \mathbf{v}')$ ***of*** $(\rho \vdash \mathbf{t}' \ {}^{s}\Downarrow_{u}^{w} \mathbf{v})$， $u_1 =$ *true* 意味着 $suspend_x =$ *true*。

算法 3 挂起分析。我们将函数式风格伪代码写为无衬线字体，以便将算法本身与 T 中的项区分开来。

函数 analyzeSuspend(**t**): $T_{\mathrm{ANF}} \to ((X \to \mathcal{P}(A)) \times \mathcal{P}(X))$

```
1  工作列表: [X] := []
2  数据:  X → P(A) := {(x, ∅) | x ∈ X}
3  挂起: P(X) := ∅
4  边:  X → P(R) := {(x, ∅) | x ∈ X}
5  对于 c ∈生成约束(t):
6    初始化约束(c)
7  迭代(); 返回 (数据, 挂起)
8
9  函数 迭代:  →() =
10   匹配工作列表，其中
11   | [] →()
12   | x :: 工作列表' →
13     工作列表 :=工作列表'
14     对于 c ∈ 边(x):
15     约束传播(c)
16     迭代 ()
17
18 函数 初始化约束(c):  R →() =
19   匹配 c
20   | a ∈ S_x → 添加数据(x, {a})
21   | S_x ⊆ S_y →
22     初始化约束'(x, c)
23   | a_1 ∈ S_x ⇒a_2 ∈ S_y →
24     初始化约束'(x, c)
25   | suspend_x → 添加挂起(x)
26   | suspend_x ⇒ suspend_y →
27     初始化约束'(x, c)
28   | ∀x∀y λx.y ∈ S_lhs
29       ⇒ (S_rhs ⊆ S_x) ∧ (S_ y ⊆ S_app) →
30     初始化约束'(lhs, c)
31   | ∀x∀n (常量_x n ∈ S_lhs) ∧(n > 1)
32       ⇒ 常量_x n − 1 ∈ S_app →
33     初始化约束'(lhs, c)
34   | ∀x λx._ ∈ S_lhs
35       ⇒ (suspend_x ⇒ suspend_res) →
36     初始化约束'(lhs, c)
37   | ∀x λx._ ∈ S_lhs ⇒ suspend_x →
38     初始化约束'(lhs, c)
39   | ∀x 常量_x _ ∈ S_lhs
40       ⇒ (suspend_x ⇒ suspend_res) →
41     初始化约束'(lhs, c)
42   | ∀x常量_x  _ ∈ S_lhs ⇒ suspend_x →
43     初始化约束'(lhs, c)
44   | suspend_res ⇒
45       (∀x λx._ ∈ S_lhs ⇒ suspend_x)
46       ∧(∀x 常量_x  _ ∈ S_lhs ⇒ suspend_x)
47     → 初始化约束'(res,c)
48 函数 初始化约束'(x,c):  X →() =
49   边(x) := 边(x) ∪ {c}
50   约束传播(c)
51
52 函数 添加数据(x, A):  X × P(A) → () =
53   条件表达式 A ⊈数据(x) 则
54     数据(x) := 数据(x) ∪ A
55     工作列表 := x :: 工作列表
56
57 函数 添加挂起(x):  X →() =
58   条件表达式 x ∉ 挂起 则
59     挂起 :=挂起 ∪{x}
60     工作列表 := x :: 工作列表
61
62 函数 约束传播(c):  R →() =
63   匹配 c 与
64   | 甲 ∈ S_x → ()
65   | S_x ⊆ S_y → 添加数据(y, 数据(x))
66   | a_1 ∈ S_x ⇒a_2 ∈ S_y →
67   条件表达式a_1 ∈ 数据(x) then 添加数据(y,{a_2})
68   | suspend_x → ()
69   | suspend_x ⇒ suspend_y →
70   条件表达式 x ∈ 挂起 then 添加挂起(y)
71   | ∀x∀y λx.y ∈ S_lhs ⇒
72       (S_rhs ⊆ S_x) ∧(S_y ⊆ S_app) →
73   for λx.y ∈数据(lhs):
74     初始化约束(S_rhs ⊆ S_x)
75     初始化约束(S_y ⊆ S_app)
76   | ∀x∀n (常量_x n ∈ S_lhs) ∧(n > 1)
77       ⇒ 常量_x n − 1 ∈ S_app →
78   对于常量_x n ∈ 数据(lhs):
79     条件表达式 n > 1 然后 添加数据(app, {常量_x n − 1})
80   | ∀x λx._ ∈ S_lhs ⇒(suspend_x  ⇒ suspend_res) →
81   用于 λx._ ∈ 数据(lhs):
82     初始化约束(suspend_x ⇒ suspend_res)
83   | ∀x λx._ ∈ S_lhs ⇒ suspend_x →
84   用于 λx._ ∈ 数据(lhs): 添加挂起(x)
85   | ∀x常量_x  _ ∈ S_lhs
86       ⇒ (suspend_x ⇒ suspend_res) →
87   对于 常量_x n ∈ 数据(lhs):
88     初始化约束(suspend_x ⇒ suspend_res)
89   | ∀x 常量_x _ ∈ S_lhs ⇒ suspend_x  →
90   对于 常量_x n ∈ 数据(lhs): 添加挂起(x)
91   | suspend_res ⇒(∀x λx._ ∈ S_lhs ⇒ suspend_x)
92     ∧(∀x 常量_x  ∈ S_lhs ⇒ suspend_x) →
93   条件表达式 res ∈ 挂起 then
94     初始化约束(∀x λx._ ∈ S_lhs ⇒ suspend_x)
95     初始化约束(∀x 常量_x _ ∈ S_lhs ⇒ suspend_x)
```

(R2) 若 $u = \mathit{true}$，则存在一个 $y \in \mathit{suspendNames}(\mathbf{t}')$，使得 $\mathit{suspend}_y = \mathit{true}$。**(R3)** 若 $\mathbf{v} = \langle \lambda y.\mathbf{t}_y, \rho_y \rangle$，则 $\lambda y.\mathit{name}(\mathbf{t}_y) \in S_{\mathit{NAME}(\mathbf{t}')}$，并且 **(C1)** 对$\rho_y$成立。

证明。 我们对 (9) 进行结构归纳。首先，假设 $\mathbf{t}' = x$ 以及相应的推导

$$\frac{}{\rho \vdash x \ {}^{[]}\Downarrow^{1}_{\text{false}} \rho(x)}(\textsc{Var}).$$

那么 **(R1)** 和 **(R2)** 立即成立，因为没有子推导，且$u =$ 为假。此外，根据 **(C1)** 和 name(**t′**) $= x$，**(R3)** 也成立。因此，我们只需考虑 $\mathbf{t}' =$ (let $x = \mathbf{t}_1$ in $\mathbf{t}_2$) 这一情形，其推导为

$$\frac{\rho \vdash \mathbf{t}_1 \ {}^{s_1}\Downarrow^{w_1}_{u_1} \mathbf{v}_1 \quad \rho, x \mapsto \mathbf{v}_1 \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow^{w_2}_{u_2} \mathbf{v}}{\rho \vdash \texttt{let } x = \mathbf{t}_1 \texttt{ in } \mathbf{t}_2 \ {}^{s_1 \| s_2}\Downarrow^{w_1 \cdot w_2}_{u_1 \vee u_2} \mathbf{v}}(\textsc{Let}).$$

为证明 **(R1)**，我们需要证明

(R1$_{\mathbf{t}_1}$) 对于推导 $\rho \vdash \mathbf{t}_1 \ {}^{s_1}\Downarrow^{w_1}_{u_1} \mathbf{v}_1$,**(R1$_{\mathbf{t}_2}$)** 而言，**(R1)** 的等价表述成立；对于推导 $\rho, x \mapsto \mathbf{v}_1 \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow^{w_2}_{u_2} \mathbf{v}$而言，**(R1)** 的等价表述成立；并且**(R1$_x$)** $u_1 =$ true 蕴含 $\mathit{suspend}_x =$ true。

当 $\mathbf{t}_2 \in T_{\text{ANF}}$ 时，我们通过证明对 $\rho, x \mapsto \mathbf{v}_1 \vdash \mathbf{t}_2 \ {}^{s_2}\Downarrow^{w_2}_{u_2} \mathbf{v}$ 而言等价于 **(C1)** 的命题（记为 **(C1$_{\mathbf{t}_2}$)**），并应用归纳假设，来确立 **(R1$_{\mathbf{t}_2}$)**。对于 **(R2)**，考虑 $u_2 =$ true 的情况。若我们确立 **(C1$_{\mathbf{t}_2}$)**，则可通过归纳假设得到 **(R2$_{\mathbf{t}_2}$)**。**(R2)** 随即由 suspendNames($\mathbf{t}_2$) $\subseteq$suspendNames(**t′**) 得出。因此，在下文中，只需先确立 **(C1$_{\mathbf{t}_2}$)**，然后在证明 **(R2)** 时假设 $u_2 =$ false。另请注意，若 **(C1$_{\mathbf{t}_2}$)** 成立，则由 **(R3$_{\mathbf{t}_2}$)** 可得 **(R3)**，因为 name($\mathbf{t}_2$) $=$ name(**t′**)。现在我们逐一考虑 $\mathbf{t}_1$ 的各个情形。综上，只要在假设 $u_2 =$false 的条件下确立 **(R1$_{\mathbf{t}_1}$)**、**(C1$_{\mathbf{t}_2}$)**、**(R1$_x$)** 和 **(R2)**，即可完成证明。

情形 $\mathbf{t}_1 = y$
$\mathbf{t}_1$ 的推导是

$$\frac{}{\rho \vdash y \ {}^{[]}\Downarrow^{1}_{\text{false}} \rho(y)}(\textsc{Var})$$

(R1$_{\mathbf{t}_1}$) 由此立即得出，因为没有子推导。
(C1$_{\mathbf{t}_2}$) 我们将环境 ρ 扩展为包含绑定 $x \mapsto \rho(y)$。对 ρ 应用 **(C1)**即可得出结果。

(R1$_x$) 因为 $u_1 =$ 为 false，所以立即得出。
(R2) 因为 $u_1 =$ 为 false，所以立即得出。

情形 $\mathbf{t}_1 = c$
$\mathbf{t}_1$ 的推导是

$$\frac{}{\rho \vdash c \ {}^{[]}\Downarrow^{1}_{\text{false}} c}(\textsc{Const})$$

($\mathbf{R1_{t_1}}$) 由于没有子推导，因此可立即得出。
($\mathbf{C1_{t_2}}$) 我们用一个绑定 $x \mapsto c$ 扩展环境 ρ 。由于 c 不是抽象，结论由 (**C1**) 得出。

($\mathbf{R1}_x$) 由于 $u_1 =$ 为假，立即可得。
(**R2**) 由于 $u_1 =$ 为假，立即可得。

情形 $\mathbf{t}_1 = \lambda y.\mathbf{t}_y$
$\mathbf{t}_1$ 的推导为

$$\frac{}{\rho \vdash \lambda y.\mathbf{t}_y \ {}^{[]}\Downarrow^1_{\text{false}} \langle \lambda y.\mathbf{t}_y, \rho \rangle}(\textsc{Lam})$$

($\mathbf{R1_{t_1}}$) 由于没有子推导，立即可得。
($\mathbf{C1_{t_2}}$) 我们将绑定 $x \mapsto \langle \lambda y.\mathbf{t}_y, \rho \rangle$加入环境 ρ 。根据假设，$\mathbf{t}'$ 是$\mathbf{t}$的子项，因此由引理 1 可得 $\lambda y.\mathsf{name}(\mathbf{t}_y) \in S_x$。此外，根据假设，(**C1**) 对 rho 成立。

($\mathbf{R1}_x$) 显然成立，因为 $u_1 =$为假。
(**R2**) 显然成立，因为 $u_1 =$为假。

情况 $\mathbf{t}_1 = y \ z$
$\mathbf{t}_1$ 的可能推导为

$$\frac{\rho \vdash y \ {}^{[]}\Downarrow^1_{\text{false}} \langle \lambda y'.\mathbf{t}_{y'}, \rho_{y'} \rangle \quad \rho \vdash z \ {}^{[]}\Downarrow^1_{\text{false}} \rho(z) \quad \rho_{y'}, y' \mapsto \rho(z) \vdash \mathbf{t}_{y'} \ {}^{s_1}\Downarrow^{w_1}_{u_1} \mathbf{v}'}{\rho \vdash y \ z \ {}^{s_1}\Downarrow^{w_1}_{u_1} \mathbf{v}'}(\textsc{App})$$

$$\frac{\rho \vdash y \ {}^{[]}\Downarrow^1_{\text{false}} c_1 \quad \rho \vdash z \ {}^{[]}\Downarrow^1_{\text{false}} c_2}{\rho \vdash y \ z \ {}^{[]}\Downarrow^1_{\text{false}} \delta(c_1, c_2)}(\textsc{Const-App})$$

首先，考虑（常量-应用）情况。

($\mathbf{R1_{t_1}}$) 成立，因为没有子推导会挂起。($\mathbf{C1_{t_2}}$) 我们将绑定 $x \mapsto \delta(c_1, c_2)$加入环境 ρ 。由于 $\delta(c_1, c_2)$不是抽象，结论由(**C1**)得出。($\mathbf{R1}_x$) 显然成立，因为 $u_1 =$为假。(**R2**) 显然成立，因为 $u_1 =$为假。

现在，考虑（应用）。首先，注意到 (**C1**) 根据 (**C1**) 对 $\rho_{y'}$ 成立。此外，根据 (**C1**)， $\rho(z)$ 满足必要条件。也就是说，对于推导 $\rho_{y'}, y' \mapsto \rho(z) \vdash \mathbf{t}_{y'} \ {}^{s_1}\Downarrow^{w_1}_{u_1} \mathbf{v}'$，($\mathbf{C1_{t_{y'}}}$) 成立，因此我们应用归纳假设得到($\mathbf{R1_{t_{y'}}}$)、($\mathbf{R2_{t_{y'}}}$)和($\mathbf{R3_{t_{y'}}}$)。

($\mathbf{R1_{t_1}}$) 由($\mathbf{R1_{t_{y'}}}$)可得。($\mathbf{C1_{t_2}}$) 我们将绑定 $x \mapsto \mathbf{v}'$加入环境 ρ。根据($\mathbf{R3_{t_{y'}}}$)，若$\mathbf{v}' = \langle \lambda y''.\mathbf{t}_{y''}, \rho_{y''} \rangle$，则 $\lambda y''.\mathsf{name}(\mathbf{t}_{y''}) \in S_{\text{NAME}(\mathbf{t}_{y'})}$。同样根据($\mathbf{R3_{t_{y'}}}$)，(**C1**)对 $\rho_{y''}$成立。最后，根据引理 1，我们有 $S_{\text{NAME}}(\mathbf{t}_{y'}) \subseteq S_x$。结论得证。

$(\mathbf{R1}_x)$ 假设 $u_1 =$成立。根据$(\mathbf{R2}_{\mathbf{t}_{y'}})$，存在一个 $y'' \in$suspendNames$(\mathbf{t}_{y'})$，使得 $suspend_y =$成立。根据引理 1，$\{suspend_n \Rightarrow suspend_{y'} \mid n \in$suspendNames$(\mathbf{t}_{y'})\}$。因此，$suspend_{y'} =$ 成立。此外，根据$(\mathbf{C1})$，$\lambda y'.$name$(\mathbf{t}_{y'}) \in S_y$。根据引理 1，$\forall y' \lambda y'.\_ \in S_{y'} \Rightarrow (suspend_{y'} \Rightarrow suspend_x)$。由此可得$suspend_x =$ 成立，证毕。

$(\mathbf{R2})$ 如前所述，我们可以安全地假设 $u_2 =$false。现在，假设 $u_1 =$true。根据 $(\mathbf{R1}_x)$，结果可直接得出，因为 $x \in$suspendNames$(\mathbf{t}')$。

情形 $\mathbf{t}_1 =$条件表达式 y then $\mathbf{t}_t$ else $\mathbf{t}_e$

$\mathbf{t}_1$ 的可能推导为

$$\frac{\rho \vdash y \ {}^{[]}\Downarrow^1_{\text{false}} \text{true} \quad \rho \vdash \mathbf{t}_t \ {}^{s_1}\Downarrow^{w_1}_{u_1} \mathbf{v}_t}{\rho \vdash \texttt{if } y \texttt{ then } \mathbf{t}_t \texttt{ else } \mathbf{t}_e \ {}^{s_1}\Downarrow^{w_1}_{u_1} \mathbf{v}_t}(\textsc{If-True})$$

$$\frac{\rho \vdash y \ {}^{[]}\Downarrow^1_{\text{false}} \text{false} \quad \rho \vdash \mathbf{t}_e \ {}^{s_1}\Downarrow^{w_1}_{u_1} \mathbf{v}_3}{\rho \vdash \texttt{if } y \texttt{ then } \mathbf{t}_t \texttt{ else } \mathbf{t}_e \ {}^{s_1}\Downarrow^{w_1}_{u_1} \mathbf{v}_e}(\textsc{If-False})$$

不失一般性，我们只考虑 (If-True)。我们直接对 $\rho \vdash \mathbf{t}_t \ {}^{s_1}\Downarrow^{w_1}_{u_1} \mathbf{v}_t$ 应用归纳假设，依据 $(\mathbf{C1})$，得到 $(\mathbf{R1}_{\mathbf{t}_t})$、$(\mathbf{R2}_{\mathbf{t}_t})$ 和$(\mathbf{R3}_{\mathbf{t}_t})$。

$(\mathbf{R1}_{\mathbf{t}_1})$ 由 $(\mathbf{R1}_{\mathbf{t}_t})$可得。

$(\mathbf{C1}_{\mathbf{t}_2})$ 我们将环境 ρ 扩展，加入绑定 $x \mapsto \mathbf{v}_t$。根据 $(\mathbf{R3}_{\mathbf{t}_t})$，若$\mathbf{v}_t = \langle \lambda y'.\mathbf{t}_{y'}, \rho_{y'} \rangle$，则 $\lambda y'.$name$(\mathbf{t}_{y'}) \in S_{\text{NAME}(\mathbf{t}_t)}$。同样根据 $(\mathbf{R3}_{\mathbf{t}_t})$，$(\mathbf{C1})$对 $\rho_{y'}$成立。最后，根据引理 1，我们有 $S_{\text{NAME}}(\mathbf{t}_t) \subseteq S_x$。结论由此得出。

$(\mathbf{R1}_x)$ 假设 $u_1 =$ 为真。根据 $(\mathbf{R2}_{\mathbf{t}_t})$，存在一个 $y' \in$ suspendNames$(\mathbf{t}_t)$，使得 $suspend_y =$ 为真。根据引理 1，$\{suspend_n \Rightarrow suspend_x \mid n \in$suspendNames$(\mathbf{t}_t)\}$。因此，$suspend_x =$ 为真。

$(\mathbf{R2})$ 假设 $u_1 =$为真。由于 $x \in$suspendNames$(\mathbf{t}')$，结论可由 $(\mathbf{R1}_x)$ 得出。

情况 $\mathbf{t}_1 =$assume y

$\mathbf{t}_1$ 的推导为

$$\frac{\rho \vdash y \ {}^{[]}\Downarrow^1_{\text{false}} d \quad w' = f_d(c)}{\rho \vdash \texttt{assume } y \ {}^{[c]}\Downarrow^{w'}_{suspend_{\texttt{assume}}} c}(\textsc{Assume})$$

$(\mathbf{R1}_{\mathbf{t}_1})$ 由于没有子推导，结论立即成立。

$(\mathbf{C1}_{\mathbf{t}_2})$ 我们将环境 ρ 扩展为包含一个绑定 $x \mapsto c$。由于 c 不是抽象，结论由 $(\mathbf{C1})$ 得出。$(\mathbf{R1}_x)$ 由引理 1 得出。$(\mathbf{R2})$ 如果 $u_1 =$ 为真，则 挂起$_{\texttt{assume}} =$ 为真，且 $x \in$ suspendNames$(\mathbf{t}')$。

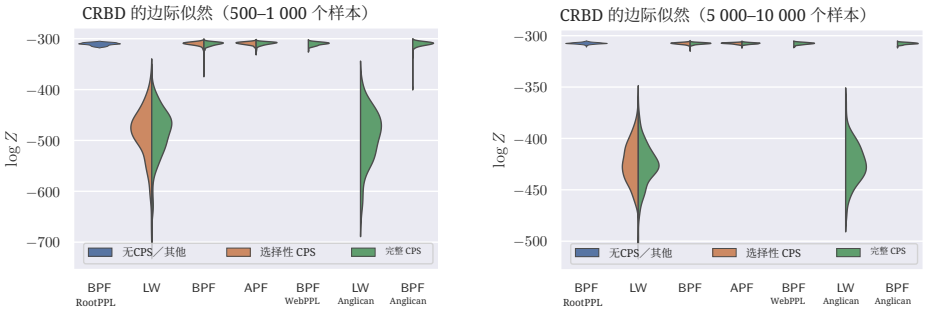


图11：恒定速率出生-死亡模型的边际似然

情形 $\mathbf{t}_1$ =权重 y
$\mathbf{t}_1$ 的推导为

$$\frac{\rho \vdash y \ {}^{[]}\Downarrow^{1}_{\text{false}} \ w'}{\rho \vdash \texttt{weight} \ \ y \ {}^{s}\Downarrow^{w'}_{suspend_{\texttt{weight}}} \ ()}\,(\textsc{Weight})$$

($\mathbf{R1}_{\mathbf{t}_1}$) 由于不存在子推导，结论立即成立。
($\mathbf{C1}_{\mathbf{t}_2}$) 我们将环境 ρ 扩展一个绑定 $x \mapsto ()$。由于 $() \in C$ 不是抽象，结果由**(C1)**得出。

($\mathbf{R1}_x$) 由引理 1 得出。**(R2)** 如果 $u_1 =$ 为真，则$suspend_{\texttt{weight}} =$ 为真，并且 $x \in$ suspendNames(**t′**)。

B 求值（续）

本节进一步介绍第 7 节中的求值细节。

B.1 恒定速率出生-死亡

图11展示了 CRBD 的 log 边际似然（也称为归一化常数 Z）的过高估计小提琴图。似然加权、BPF 和 APF 推断都会生成边际似然估计，我们可以将其用作衡量推断准确性的直接指标，并以此验证实现的正确性。我们从图11中可以看到，对于不同的概率编程语言以及选择性/完整 CPS，每种推断算法的边际似然估计分布都是等价的，从而证明了 Miking CorePPL 实现的正确性。

清单1、清单2和列表 3分别给出了 Miking CorePPL、Anglican 和 WebPPL 中 CRBD 的源代码。

清单1：第 7.1 节中 CRBD 实验的 CorePPL源代码

```
1包含 "phylo.mc" 2包含 "
tree-alcedinidae.mc"
```

3包含 "math.mc" 4 5 mexpr 67-- 先验分布8let lambda = assume (伽马分布 1.0 1.0) in9let mu = assume (伽马分布 1.0 0.5) in 10 11递归 let 存活 = lam tBeg. 12 let t = subf tBeg (assume (指数分布 (addf lambda mu))) in 13 if ltf t 0. then 14 assume (伯努利分布 rho) 15 else 16 if assume (伯努利分布 (divf lambda (addf lambda mu))) then 17 if 存活 t then 18 true 19 else 20 存活 t 21 else 22 false 23 in 24 25递归 let 遍历 = lam 节点. lam parentAge. 26 let nodeAge = getAge 节点 in 27 递归 let simHiddenSpeciation =lam tBeg. 28 let t = subf tBeg (assume (指数分布 lambda)) in 29 if gtf t nodeAge then 30 if 存活 t then 31 权重 (negf inf) 32 else 33 权重 (log 2.); 34 simHiddenSpeciation t 35 else () 36 in 37 simHiddenSpeciation parentAge; 38 observe 0 (泊松分布 (mulf mu (subf parentAge nodeAge))); 39 match 节点 with 节点 n then 40 observe 0. (指数分布 lambda); 41 resample; 42 遍历 n.left nodeAge; 43 遍历 n.right nodeAge 44 else match 节点 with 叶节点 _ then 45 observe true (伯努利分布 rho); 46 resample 47 else never 48 in 49 50let 叶节点数 = 计数叶节点 树 in 51权重 (subf (mulf (subf (int2float 叶节点数) 1.) (log 2.)) 52 (对数阶乘 叶节点数)); 53match 树 with 节点 root in 54遍历 root.left root.age; 55遍历 root.right root.age; 56 lambda

清单2：第 7.1 节 CRBD 实验的 Anglican 源代码

1(ns crbd.core 2 (:require [clojure.tools.cli :refer [parse-opts]])3 (:use [anglican [core :exclude [-main cli-options parse-options]]4 emit runtime]5 clojure.pprint)6 (:gen-class))78(def~:const tree9 ;;为简洁起见，省略定义10)

```
11 12(def~:const rho 0.5684210526315789) 13 14(defdist id 15 "用于使 factor 和 condition
函数正确工作的变通方法"16 [] [17 18 ;; 不允许采样19 [ ] (sample* this
(throw (Exception. "id-dist 不支持采样"))) 21 22 ;; 观测值的“对数概率”就是其本身 23
(observe* [this value]value)) 24 25(with-primitive-procedures [id]26 ;; 权重 27 (defm
factor [x]28 "类似 WebPPL 的因子函数" 29 (observe (id) x)) 30 31 (defm condition [b]32 "
类似 WebPPL 的条件函数" 33 (if b (factor 0) (factor Double/NEGATIVE_INFINITY))) 34 35
(defm count-leaves [tree]36 (case (:type tree) 37 :node (+(count-leaves (:left tree))
(count-leaves (:right tree))) 38 :leaf 1)) 39 40 (defm log-factorial [n] (if (= n 1) 0 (+ (log
n) (log-factorial (- n 1)))))) 41 42(defquery crbd 43 (let [lambda (sample (gamma 1 1)) ;注
意：gamma 的参数化方式为形状/速率 44 mu (sample (gamma 1 2)) 45 存活 (fn survives [
t-beg]46 (let [t (- t-beg (sample (指数分布 (+ lambda mu))))]47 (if (< t 0) 48 (sample (伯努
利翻转 rho)) 49 (if (sample (伯努利翻转 (/ lambda (+lambda mu)))) 50 (or (survives t)
(survives t)) 51 false)))) 52 遍历 (fn walk [tree parent-age]53 (let [
sim-hidden-speciation 54 (fn sim-hidden-speciation [t-beg]55 (let [t (- t-beg (sample (指
数分布 lambda)) )]56 (if (>t (:age tree)) 57 (if (survives t) 58 Double/NEGATIVE_INFINITY
59 (+(log 2) (sim-hidden-speciation t))) 60 0))) 61 score (+(sim-hidden-speciation
parent-age) 62 (observe* 63 (泊松分布 (* mu (- parent-age (:age tree)))) 64 0))]65 (case
(:type tree) 66 :node (do 67 (factor (+score (observe* (指数分布 lambda) 0))) 68 (walk (:l
eft tree) (:age tree)) 69 (walk (:right tree) (:age tree))) 70 :leaf (factor (+score (observe* (伯
努利分布 rho) 1)))))) 71 num-leaves (count-leaves tree)]72 73 (factor (- (* (- num-leaves 1)
(log 2)) (log-factorial num-leaves))) 74 (walk (:left tree) (:age tree)) 75 (walk (:right tree)
(:age tree)) 76 lambda)) 77 78(defn norm-const [weights]
```

```
(let [最大权重 (apply max weights)] (if (=最大权重 Double/NEGATIVE_INFINITY) 最大权重 (let [总和 (reduce + (map #(exp (- % 最大权重)) weights))] (- (+最大权重 (log 总和)) (log (count weights)))))))

(def cli-options [["-m" "--method _METHOD" "推断方法，可选 :importance、:pimh、:lmh 或 :smc" :default :importance :parse-fn #(case % ":importance" :importance ":pimh" :pimh ":lmh" :lmh ":smc" :smc) :validate [some? "必须是已知算法"]] ["-p" "--particles _COUNT" "粒子数量" :default 10 :parse-fn #(Integer/parseInt %) :validate [#(< 0 %) "必须是大于 0 的数字"]] ["-o" "--output" "将样本输出到标准输出"] ["-h" "--help"]])

(defn int-or-nil [number-string] (try (Integer/parseInt number-string) (catch Exception e nil)))

(defn parse-args [args] (if (= (count args) 1) (int-or-nil (first args))))

(defn -main [& args] (let [opts (parse-opts args cli-options) nsamples (parse-args (:arguments opts))] (if (some? (:errors opts)) (do (println (:errors opts)) (println (:summary opts)) (System/exit 1)) (if (not (some? nsamples)) (do (println (:summary opts)) (System/exit 1)) (let [opts (:options opts) samples (doall (take nsamples (case (:method opts) :smc :pimh) (doquery (:method opts) crbd nil :number-of-particles (:particles opts) :drop-invalid false) (doquery (:method opts) crbd nil :drop-invalid false)))) norm-const? (case (:method opts) (:smc :importance) true false)] (when norm-const? (println (norm-const (map :log-weight samples)))) (when (some? (:output opts)) (run! #(printf "%f %f\n" (:result %) (:log-weight %)) samples) (flush)))))))
```

列表 3：第 7.1 节 CRBD 实验的 WebPPL 源代码

```
let tree = // 为简洁起见省略定义
let rho = 0.5684210526315789
let countLeaves = function(tree) {
  return tree.type =='节点' ? countLeaves(tree.left) + countLeaves(tree.right): 1
}
let logFactorial = function(n) {
  return n == 1? 0: Math.log(n) + logFactorial(n - 1)
}
```

```
12 13let model = function() { 14 // 先验分布 15 let lambda = 伽马分布({shape: 1.0,
scale: 1.0}) 16 let mu = 伽马分布({shape: 1.0, scale: 0.5}) 17 18 let survives =
function(tBeg) { 19 let t =tBeg - 指数分布({a: lambda +mu}) 20 条件表达式 (t < 0) {
21 return 伯努利翻转(rho) 22 } 23 if (伯努利翻转(lambda/(lambda +mu))) { 24 ret
urn survives(t) ||survives(t) 25 } 26 return false 27 } 28 29 let walk =
function(node, parentAge) { 30 let simHiddenSpeciation =function(tBeg) { 31 let t =
tBeg - 指数分布({a: lambda}) 32 if (t > node.age) { 33 return survives(t)? 34 -
Infinity: Math.log(2) +simHiddenSpeciation(t) 35 } 36 return 0. 37 } 38 let score =
simHiddenSpeciation(parentAge) 39 + 泊松分布({mu:
mu*(parentAge - node.age)}).score(0) 40 if (node.type == '节点') { 41 factor(score +
指数分布({'a': lambda}).score(0)) 42 walk(node.left, node.age) 43 walk(node.right,
node.age) 44 } else { 45 factor(score + 伯努利分布({p: rho}).score(true)) 46 } 47 } 48
49 let numLeaves =计数叶节点(tree) 50 factor((numLeaves - 1)*Math.log(2) - 对数阶
乘(numLeaves)) 51 walk(tree.left, tree.age) 52 walk(tree.right, tree.age) 53 return
lambda 54 } 55 56var myArgs = process.argv.slice(3); 57var obj = { 58 parseInt:
parseInt 59 } 60var method = myArgs[0]; 61var particles = obj.parseInt(
myArgs[1], 10); 62if (method == '序贯蒙特卡洛') { 63 var dist = 推断({method:
method, particles: particles, model: model}) 64 dist.normalizationConstant 65 }
66else if (method=='马尔可夫链蒙特卡洛') { 67 var dist = 推断({method: method,
samples: particles, model: model}) 68 }
```

B.2 支系形成多样化率 shift

图 12 展示了 ClaDS 对数边际似然估计值的高估小提琴图。可以看到，在选择性 CPS 和完整 CPS 下，估计值保持不变，这证明了 Miking CorePPL 实现的合理性。

代码清单 4 给出了 Miking CorePPL 的 ClaDS 源代码。

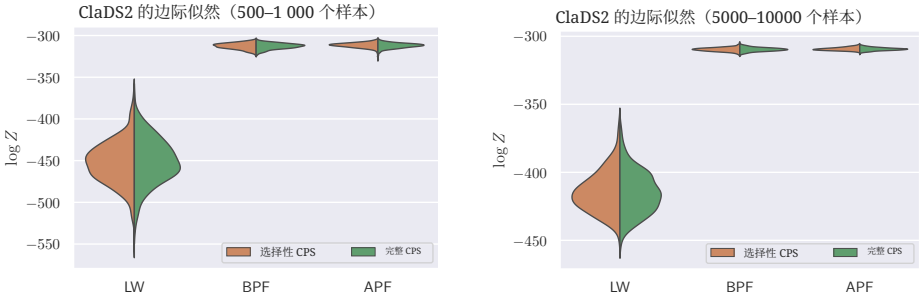


图 12：分支形成多样化速率转变模型的边际似然

代码清单 4：第 7.2 节中 ClaDS 实验的 CorePPL源代码

```
1 包含 "../../crbd/coreppl/phylo.mc" 2 包含 "../../crbd/coreppl/tree-alcedinidae.mc"
3 包含 "math.mc"4包含 "bool.mc" 5 6 mexpr 7-- 先验分布 9 let lambda = assume (伽
马分布 1.0 1.0) in 10let mu = assume (伽马分布 1.0 0.5) in 11let sigma = sqrt (divf
0.2 (assume (伽马分布 1.0 1.0))) in 12let log_alpha = assume (高斯分布 0. sigma) in
13 14递归 let 存活 = lam tBeg. lam multiplier. 15 let multiplier =16 mulf
multiplier (exp (assume (高斯分布 log_alpha sigma))) in 17 条件表达式 or (ltf
multiplier 1e-5) (gtf multiplier 1e5) then 18 true 19 else 20 let t =21 subf tBeg 22
(assume (指数分布 (mulf multiplier (addf lambda mu)))) in 23 条件表达式 ltf t 0. then
24 assume (伯努利分布 rho) 25 else 26 if assume (伯努利分布 (divf lambda (addf
lambda mu))) then 27 -- 物种形成 28 if 存活 t multiplier then 29 true 30 else 31 存活 t
multiplier 32 else 33 -- 灭绝 34 false 35 in 36 37递归 let 遍历 = lam 节点. lam
parentAge. lam multiplier. 38 let nodeAge = getAge 节点 in 39 递归 let
simHiddenSpeciation =lam tBeg. lam multiplier. 40 let multiplier =41 mulf
multiplier (exp (assume (高斯分布 log_alpha sigma))) in 42 条件表达式 or (ltf
multiplier 1e-5) (gtf multiplier 1e5) then 43 权重 (negf inf); 44 multiplier 45 else 46
let t = subf tBeg (assume (指数分布 (mulf multiplier lambda))) in 47 if gtf t
nodeAge then
```

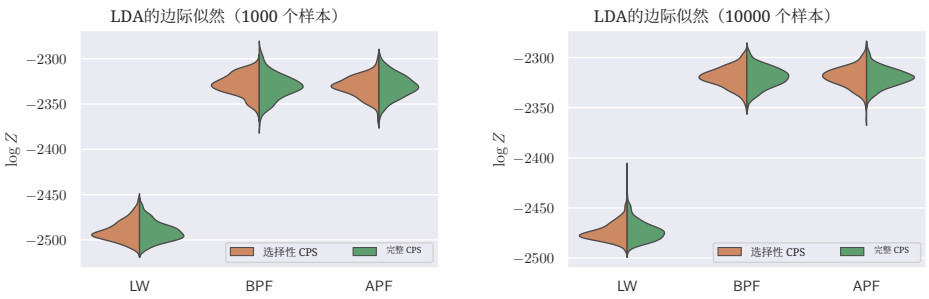


图13：潜在狄利克雷分配模型的边际似然

```
48 if 存活t multiplier then49 权重 (negf inf);50 multiplier51 else52
observe0 (泊松分布 (mulf (mulf mu multiplier) (subf tBeg t))); 53 权重 (log2.);54
simHiddenSpeciation t multiplier55 else56 observe 0(泊松分布 (mulf (mulf mu
multiplier) (subf tBeg nodeAge)));57 multiplier58 in59 let multiplier =
simHiddenSpeciation parentAge multiplier in60 match 节点 with Node n then61
observe 0. (指数分布 (mulf multiplier lambda));62 resample; 63 遍历 n.leftnodeAge
multiplier; 64 遍历 n.right nodeAge multiplier65 else match 节点with Leaf _ then66
observe true (伯努利分布 rho);67 resample68 else never69 in7071let nu
mLeaves = 计数叶节点 tree in72权重 (subf (mulf (subf (int2float numLeaves) 1.) (log
2.))73 (对数阶乘 numLeaves));74match tree with Node root in75遍历 root.left
root.age 1.; 76遍历 root.right root.age 1.;77 lambda
```

B.3 潜在狄利克雷分配

图13展示了 LDA 的对数边际似然高估值的小提琴图。再次可以看到，选择性 CPS 与完整 CPS 得出的估计值相同，这证明了 Miking CorePPL 实现的正确性。

代码清单5给出了 Miking CorePPL 的 LDA 源代码。

代码清单5：第7.3节中 LDA 实验的 CorePPL源代码

```
1include "common.mc"2
include "string.mc" 3i
nclude "seq.mc" 4include "
ext/dist-ext.mc"5
```

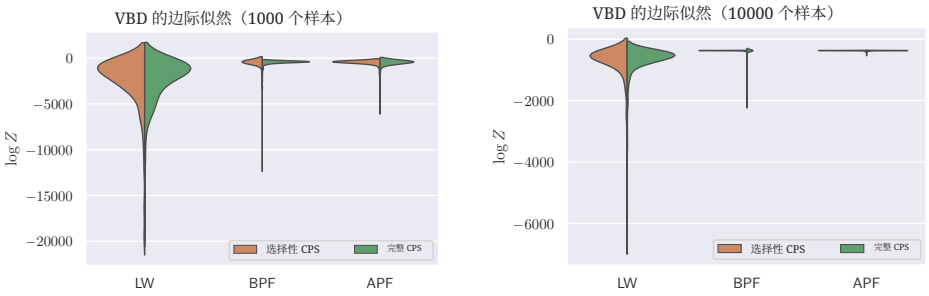


图14：媒介传播疾病模型的边际似然

```
6include "data-c3.mc" 7 8 mexpr 9 10let alpha: [Float] = make numtopics 1. in
11let beta: [Float] = make vocabsize 1. in 12let phi = create numtopics (lam. ass
ume (狄利克雷分布 beta)) in 13let theta = create numdocs (lam. assume (狄利克雷分布
alpha)) in 14repeati (lam w. 15 let word = get docs w in 16 let counts = assume (多项分
布 word.1 (get theta (get docids w))) in 17 iteri (lam z. lam e. 18 权重 (mulf (int2float e)
19 (伯努利分布LogPmf (get (get phi z) word.0) true)) 20 ) counts; 21 resample 22 ) (le
ngth docs); 23 24-- 我们只比较执行时间，因此返回 unit（无内容）作为 25-- 结果 26 ()
```

B.4 媒介传播疾病

图14展示了 VBD 对数边际似然估计值的小提琴图。我们再次看到，在选择性 CPS 和完整 CPS 下，估计值均未发生变化，这证明了 Miking CorePPL 实现的正确性。

代码清单6给出了 Miking CorePPL 的 VBD源代码。

代码清单6：第7.4节中 VBD实验的 CorePPL源代码

```
1 ----------------------------------------------------------------------------
2-- 来自 https://docs.birch.sh/examples/VectorBorneDisease/ 的 SEIR模型 --3 ----------------------------------------------------------------------------
45-- 引入 pow 和 exp 6include "math.mc"78-- 引入数据9include "data.mc"1011
mexpr1213-- 人类参数 14let hNu: Float = 0. in
```

15let hMu: Float = 1. in 16let hLambda: Float = assume (Beta 1. 1.) in 17let hDelta: Float =18 assume (Beta (addf 1. (divf 2. 4.4)) (subf 3. (divf 2. 4.4))) in 19let hGamma: Float =20 assume (Beta (addf 1. (divf 2. 4.5)) (subf 3. (divf 2. 4.5))) in 21 22-- 蚊子参数 23let mNu: Float = divf 1. 7. in 24let mMu: Float = divf 6. 7. in 25let mLambda: Float = assume (Beta 1. 1.) in 26let mDelta: Float =27 assume (Beta (addf 1. (divf 2. 6.5)) (subf 3. (divf 2. 6.5))) in 28let mGamma: Float = 0. in 29 30-- 其他参数 31let rho: Float = assume (Beta 1. 1.) in 32let z: Int = 0 in 33 34-- 人类SEIR组件 35let n: Int = 7370 in 36let hI: Int = addi 1 (assume (Poisson 5.0)) in 37let hE: Int = assume (Poisson 5.0) in 38let hR: Int =39 floorfi (assume (Uniform 0. (int2float (addi 1 (subi (subi n hI) hE))))) in 40let hS: Int = subi (subi (subi n hE) hI) hR in 41 42-- 人类初始增量 43let hDeltaS: Int = 0 in 44let hDeltaE: Int = hE in 45let hDeltaI: Int = hI in 46let hDeltaR: Int = 0 in 47 48-- 蚊子SEIR组件 49let u: Float = assume (Uniform (negf 1.) 2.) in 50let mS: Int = floorfi (mulf (int2float n) (pow 10. u)) in 51let mE: Int = 0 in 52let mI: Int = 0 in 53let mR: Int = 0 in 54 55-- 蚊子初始增量 56let mDeltaS: Int = 0 in 57let mDeltaE: Int = 0 in 58let mDeltaI: Int = 0 in 59let mDeltaR: Int = 0 in 60 61-- 条件化函数 62let condition: Int -> Int -> Int -> Int =63 lam t: Int. lam zP: Int. lam hDeltaI: Int. 64 let z: Int =addi zP hDeltaI in 65 let y: Int =get ys t in 66 let z: Int =if neqi (negi 1) y then 67 observe y (Binomial z rho); 0 68 else z in 69 resample; 70 z 71 in 72 73-- 初始条件化 74let z = condition 0 z hDeltaI in 75 76-- 模拟函数 77recursive let simulate: 78 Int ->Int ->Int ->Int ->Int ->Int ->Int ->Int ->Int ->Int ->Int -> ()79 =80 lam t: Int. 81 lam hSP: Int. 82 lam hEP: Int.

83 lam hIP: Int. 84 lam hRP: Int. 85 lam mSP: Int. 86 lam mEP: Int. 87 lam mIP: Int. 88 lam mRP: Int. 89 lam zP: Int. 90 91 -- 人类 92 let hN: Int =addi (addi (addi hSP hEP) hIP) hRP in 93 let hTau: Int =94 assume (二项分布 hSP (subf 1. (exp (negf (divf 95 (int2float mIP) 96 (int2float hN)))))) in 97 let hDeltaE: Int = assume (二项分布 hTau hLambda) in 98 let hDeltaI: Int =assume (二项分布 hEP hDelta) in 99 let hDeltaR: Int =assume (二项分布 hIP hGamma) in 100 let hS: Int = subi hSP hDeltaE in 101 let hE: Int =subi (addi hEP hDeltaE) hDeltaI in 102 let hI: Int =subi (addi hIP hDeltaI) hDeltaR in 103 let hR: Int =addi hRP hDeltaR in 104 105 -- 蚊子 106 let mTau: Int =107 assume (二项分布 mSP (subf 1. (exp (negf (divf 108 (int2float hIP) 109 (int2float hN)))))) in 110 let mN: Int =addi (addi (addi mSP mEP) mIP) mRP in 111 let mDeltaE: Int =assume (二项分布 mTau mLambda) in 112 let mDeltaI: Int =assume (二项分布 mEP mDelta) in 113 let mDeltaR: Int =assume (二项分布 mIP mGamma) in 114 let mS: Int =subi mSP mDeltaE in 115 let mE: Int =subi (addi mEP mDeltaE) mDeltaI in 116 let mI: Int = subi (addi mIP mDeltaI) mDeltaR in 117 let mR: Int =addi mRP mDeltaR in 118 119 let mS: Int =assume (二项分布 mS mMu) in 120 let mE: Int =assume (二项分布 mE mMu) in 121 let mI: Int =assume (二项分布 mI mMu) in 122 let mR: Int = assume (二项分布 mR mMu) in 123 124 let mDeltaS: Int =assume (二项分布 mN mNu) in 125 let mS: Int =addi mS mDeltaS in 126 127 let z: Int =condition t zP hDeltaI in 128 129 -- 递归 130 let tNext: Int =addi t 1 in 131 if eqi (length ys) tNext then 132 -- 此处不返回任何内容，而只是运行模型 133 -- 以估计归一化常数。 134 () 135 else 136 simulate tNext hS hE hI hR mS mE mI mR z 137 in 138 139-- 开始递归 140simulate 1 hS hE hI hR mS mE mI mR z