北京大学

博士学位研究生学科综合考试报告

姓　　名：吉如一

学　　号：2001111323

院　　系：计算机学院

专　　业：计算机软件与理论

研究方向：程序合成

导　　师：胡振江教授

协助指导：熊英飞副教授

二〇二二年五月

摘要

本文调研了程序语言领域中的三个研究方向，分别是程序合成技术、程序演算技术与概率编程语言。

程序合成关注的问题是如何让计算机自动地合成一个满足给定规约的程序。经过几十年的发展，大量通用的程序合成算法被提出，这些算法也在各种不同的实际场景中得到了应用。在第一章中，本文首先介绍程序合成问题的一个标准化框架，并接着对程序合成领域的两大流派，演绎程序合成和归纳程序合成，以及这两个流派中细分的程序合成技术进行了系统性的归类与介绍。

程序演算由程序变换这一领域发展而来。与一般性的表达式层面的程序变换系统不同，程序演算关注的是程序组件这一较高层次的变换规则。得益于高层次的视角，程序演算领域中的变换规则通常具有很好的泛用性且往往能用于描述算法层面的优化。在第二章中，本文首先在列表这一具体的数据结构上介绍这一领域以程序组件为单位的研究思想，并依次对该领域中基于范畴论的程序模型、算法层面上的变换规则、以及在自动化方面的尝试进行介绍。

概率编程语言是一种新兴的程序语言类型，其目标是自动化概率模型的推断过程，从而使得用户的精力可以集中于概率模型的设计与调整上。在语法上，概率编程语言在传统语言的基础上引入了概率相关的语法组件，从而使得用户可以便捷地设计、迭代概率模型。在语义上，概率编程语言可以使用通用的推断算法来自动地完成概率推断任务。在第三章中，本文将首先对目前已有的概率编程语言按照其应用场景进行系统性的归类，接着依次介绍概率编程语言中的语义、自动推断算法，以及在概率编程语言上的程序合成技术。

最后，在第四章中，本文将提出作者对三个方向的总结以及对未来探索的设想。

目录

第一章　程序合成技术

1.1　引言

程序合成（Program Synthesis）关注的问题是如何让计算机自动地合成一个满足给定规约的程序。它被图灵奖得主 Amir Pnueli 称为是程序语言理论中最核心的问题之一 [1]。早期对程序合成的研究可以追溯到 1963 年，由 Church 提出的电路合成问题 [2]. 经过半个多世纪的发展，程序合成已经发展成为了程序语言中一个相当活跃的子领域。有大量通用程序合成算法被提出，这些算法也在各种不同的场景下得到了应用，如复杂算法、数据结构的合成 [3–6]，超优化 [7]，代码去混淆 [8, 9]，Execl 表格的自动补全 [10]，SQL 询问的合成 [11] 等。

程序合成技术发展的一个重要节点是语法制导合成（Syntax-Guided Synthesis，缩写为 SyGuS）框架的提出 [12]。该框架将程序合成问题描述为从给定上下文无关文法中找到一个满足给定逻辑表达式的程序的任务。语法制导合成框架作为接口，成功地分离了程序合成器的设计阶段与应用阶段。这一方面降低了程序合成技术在实际问题上的使用难度，另一方面降低了研究过程中数据集的复用难度，使得研究人员可以便捷地评估不同程序合成器的性能。本文主要针对这一框架下的程序合成进展作简单的介绍与部分文献综述。

程序合成技术可以按照对于规约的使用方式分成两类：演绎程序合成 [13] 和归纳程序合成 [14]。演绎程序合成方法使用演绎推理规则对给定的规约进行变换，并从推理过程与变换结果中直接提取目标程序；而归纳程序合成方法从具体的有关程序运行的例子出发（例如输入输出用例），尝试归纳得到一个满足例子的程序。

接下来，本文将首先对语法制导合成的框架进行介绍，并随后对基于演绎的程序合成和基于归纳的程序合成这两类技术中的通用算法作详细的介绍。

1.2　语法制导合成

语法制导合成框架由美国宾夕法尼亚大学的 Alur 等人于 2013 年提出 [12]。该框架使用了三个部分来描述一个程序合成问题：

- 背景理论 T 中定义了程序合成问题中（包括规约、候选程序）涉及的类型、常量、运算符的语义。在一个语法制导合成的问题中，我们通常只关心那些使得模 T 的一阶可满足性问题可以快速求解的背景理论 T，如线性整数运算（LIA）理论、定长布尔表达式（BV）理论等。

- 背景理论 T 下的公式 $\Phi = \exists f_1, \ldots, f_n, \forall \overline{x}, \phi(f_1, \ldots, f_n, \overline{x})$ 定义了程序合成的规约，其中 $f_1, \ldots, f_n$ 是一系列的二阶函数变量，表示所有目标程序，$\overline{x}$ 是一系列的一阶变量，通常用于描述目标函数的输入。
- 对于每一个目标程序 f_i，上下文无关文法 G_i 描述了 f_i 对应的程序空间。

给定一个语法制导合成问题，程序合成器的任务是从每一个 G_i 中找到一个程序 f_i^*，使得公式 $\forall \overline{x}, \phi(f_1^*, \ldots, f_n^*, \overline{x})$ 在模 T 理论下为真。同时，若要判定一组解 $(f_1, \ldots, f_n)$ 的合法性，可以用可满足性模理论（SMT）求解器求解公式 $\exists \overline{x}, \neg\phi(f_1, \ldots, f_n, \overline{x})$。如果该公式不可满足，则解 $(f_1, \ldots, f_n)$ 必然合法。否则，$(f_1, \ldots, f_n)$ 不合法，且 SMT 求解器可以得到一组反例 $\overline{x'}$，表示 $\phi(f_1, \ldots, f_n, \overline{x'})$ 应当为真，但是在当前解上为假。

例 1 考虑如下的语法制导合成问题。

- 背景理论 T 为线性整数运算理论，其中变量类型只包括整数与布尔，表达式只包括布尔运算、整数加（+）、整数比较（≤）与分支操作（ite）。
- 公式 Φ 为 $\exists f, \forall x, y : \mathtt{Int}, \phi(f, x, y)$，其中 $\phi(f, x, y) = f(x, y) \geq x \wedge f(x, y) \geq y \wedge (f(x, y) = x \vee f(x, y) = y)$，即程序 f 的输出需要是两个输入中的较大值。
- 上下文无关文法 G 如下图所示，其中 I 为初始非终结符，表示整数表达式，B 为非终结符，表示布尔表达式。

$$I \to x \mid y \mid 1 \mid I + I \mid ite(B, I, I) \qquad B \to I \leq I$$

程序 $f_1 = ite(x \leq y, y, x)$ 是该问题的一组合法解：公式 $\exists x, y \in \mathtt{Int}, \neg\phi(f_1, x, y)$ 不可满足。而程序 $f_2 = x$ 不是该问题的一组合法解：公式 $\exists x, y \in \mathtt{Int}, \neg\phi(f_2, x, y)$ 可满足，$(x = 1, y = 2)$ 是一组反例。

1.3 演绎程序合成

最早期的程序合成算法大多是基于演绎推理的。早在 1992 年 Manna 与 Waldinger 便明确定义了演绎程序合成的框架 [13]。给定逻辑规约 S，演绎程序合成方法尝试构造命题“存在一个程序满足规约 S”的证明，并从证明中还原出一个满足规约的程序。在给定一个高效的证明系统的情况下，演绎程序合成方法通常可以生成相对复杂的程序。经过三十年的发展，演绎程序合成技术已经有了大量的应用，包括合成高效分治算法 [4, 15, 16]，合成堆操作程序 [17, 18]，合成数字信号变换 [19]，合成并行化数据结构 [20] 等。

在本节中，本文将首先介绍 Manna 与 Waldinger 提出的的经典演绎程序合成框架，并接着介绍近期演绎程序合成方法的一个重要进展，堆操作程序的自动合成 [17, 18]。

表 1.1　演绎推理过程的表格描述。

断言	目标	程序
$\mathcal{A}$		p
	$\mathcal{G}$	p

最后，值得一提的是，在许多实际问题中，尤其是在语法制导合成问题中，演绎程序合成方法存在着两大缺陷。

- 演绎推理系统很难处理文法约束，因此演绎合成方法目前只能用于使用通用文法的问题上，例如包含所有被演绎推理系统使用的操作符的文法。
- 演绎推理系统的合成是拟合到给定规约 S 上的。若要合成正确的程序，用户需要给出其完备的规约，否则合成结果很可能会过拟合到给定的不完备的规约上。然而，在许多应用中，提供完备的规约对于用户来说是困难的。

在程序合成技术的发展过程中，这两大缺陷最终被归纳合成方法所弥补。本文将在第 1.4 节中对这一类技术展开介绍。

1.3.1　演绎程序合成框架

在 Manna 与 Waldinger 定义的演绎程序合成框架 [13] 中，合成的目标是证明 $\exists f, \forall \bar{i}, \phi(f, \bar{i})$ 成立，其中 f 表示目标函数，$\bar{i}$ 表示 f 的输入。因为 f 是一个函数，等价地，该约束可以被视为 $\forall \bar{i}, \exists o, \phi'(o, \bar{i})$，其中 o 对应 $f(\bar{i})$。

例 2 下列公式描述了对应求解两个整数中较大值的程序的证明任务。

$$\forall x, y, \exists o, o \geq x \wedge o \geq y \wedge (o = x \vee o = y)$$

表 1.1 展示了该框架中描述演绎推理过程的表格。表格中的行可以被分为两类：

- 断言行包含一个断言 $\mathcal{A}$，表示已知为真的命题。
- 目标行包含一个目标 $\mathcal{G}$，表示需要生成结果满足的条件。

同时，每一行可能包含一个程序 p，表示根据目前目标程序需要满足的形式。对于所有变量的一个具体赋值 I，程序 p^* 满足一个断言行 $\mathcal{A}$（目标行 $\mathcal{G}$）当且仅当存在一个对变量的替换 θ 满足（1）断言 $\mathcal{A}\theta$ 在 I 下为假（目标 $\mathcal{G}\theta$ 在 I 下为真）且（2）该行的程序 p 为空或者 $p^* = p\theta$。

例 3 程序 y 在赋值 $\langle x = 1, y = 2 \rangle$ 下满足包含目标 $o \geq x$ 与程序 o 的目标行，此时 $\theta = \{o \mapsto y\}$ 是一个合法的替换，$(o \geq x)\theta = y \geq x$，$o\theta = y$。

给定包含行 $r_1, \ldots, r_n$ 的表格 T，程序 p^* 满足表格 T 当且仅当对于任意合法的赋值 I，程序 p^* 均满足表格中的至少一行。根据该定义，给定约束 $\forall \bar{i}, \exists o, \phi(\bar{i}, o)$，程序 p 是一个合法程序当且仅当它满足表 1.2，其中 $\mathcal{A}_1, \ldots, \mathcal{A}_n$ 是一系列已知为真的命题。

表 1.2　演绎程序合成过程中的初始表格。

断言	目标	程序
$\mathcal{A}_1$		
...		
$\mathcal{A}_n$		
	$\phi(\bar{i}, o)$	o

表 1.3　演绎程序合成方法合成求解两个整数中较大值程序的推理过程。

	断言	目标	程序	依据
1	$x = x$			
2	$x \leq x$			
3	$x \leq y \wedge y \leq x$			
4		$x \leq o \wedge y \leq o \wedge (x = o \vee y = o)$	o	
5		$x \leq o \wedge y \leq o \wedge x = o$	o	目标 4
6		$x \leq o \wedge y \leq o \wedge y = o$	o	目标 4
7		$x \leq x \wedge y \leq x$	x	断言 1，目标 5
8		$y \leq x$	x	断言 2，目标 7
9		$x \leq y \wedge y \leq y$	y	断言 1，目标 6
10		$x \leq y$	y	断言 2，目标 9
11		true	$x < y ? y : x$	断言 3，目标 8/10

演绎程序合成器的合成过程依赖预先定义的演绎规则。一条合法演绎规则在使用时都会给表格添加上新的一行，并保证满足表格的程序集合保持不变。从初始表格（表 1.2）出发，演绎程序合成器不断地使用演绎规则，直到出现了以下两种类型的行之一。

- 包含断言 false 与只使用输入变量 $\bar{i}$ 的程序 p 的断言行。
- 包含目标 true 与只使用输入变量 $\bar{i}$ 的程序 p 的目标行。

根据满足关系的定义，程序 p 在任意赋值下一定都满足上述类型的行。因此当表格中出现上述类型的行时，程序 p 一定满足当前表格，从而 p 是一个合法程序。

例 4 *表格 1.3 展示了演绎程序合成器合成求解两个整数中较大值的程序的推理过程。*

1.3.2　堆操作程序自动合成

近几年，Polikarpova 等人发表了一系列将演绎程序合成用于堆操作操作程序的论文 [17, 18, 21–23]，并在程序合成社区中获得了极高的关注度：其中两篇论文分别获评程序语言领域两大顶会 POPL 与 PLDI 的杰出论文奖。

为了描述堆操作程序的规约，这一系列工作使用了分离逻辑来描述程序的语义。分离逻辑由 Reynolds 于 2002 年提出 [24]。它在霍尔三元组的基础上引入了分离逻辑运算符，从而简化了对堆相关程序的验证。分离逻辑中与堆相关的操作符如下所示。

- emp 表示一个空堆，$p \mapsto v$ 表示地址 p 的值为 v。
- 分离与 $*$ 是对逻辑与 $\wedge$ 的扩展，$P_1 * P_2$ 不仅要求 P_1, P_2 均为真，还需要 P_1, P_2 中涉及的地址集合不交。

在分离逻辑中，三元组 $\{P\}p\{Q\}$ 表示在给定满足 P 的堆状态的情况下，程序 p 的运行结果满足 Q。根据分离与的定义，可以得到定理 $\{P\}p\{Q\} \iff \{P * H\}p\{H * P\}$。

例 5 下列公式给出了交换函数 *swap* 的逻辑规约。

$$\{x \mapsto a * y \mapsto b\}swap(x, y)\{x \mapsto b * y \mapsto a\}$$

与一般性的合成问题相比，从分离逻辑规约中合成堆操作程序是困难的。一方面，验证一个程序是否满足由分离逻辑规约是困难的，并不能直接使用 SMT 求解器求解。另一方面，堆操作程序会涉及递归数据结构、递归函数等，通常在规约与控制流方面都更加复杂。

遵循演绎程序合成的框架，Polikarpova 等人于 19 年提出了一个推理系统，用于证明“存在程序 p 满足 $\{P\}p\{Q\}$”，并将其实现为了一个演绎程序器。在该推理系统中 $\Gamma; P \rightsquigarrow Q|p$ 表示在上下文 Γ 的情况下，程序 p 满足三元组 $\{P\}p\{Q\}$。简单起见，本文并不介绍该系统中的具体规则，而是以一个例子展示该系统的推理过程。

例 6 考虑例 5 中的规约，假设目标程序的第一步读取出了地址 x 中的值并将其存储到了临时变量 a_2 中，则可以进行如下推导。

$$\frac{\{x, y, a_2\}; \{x \mapsto a_2 * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a_2\}|c_2}{\{x, y\}; \{x \mapsto a * y \mapsto b\} \rightsquigarrow \{x \mapsto b * y \mapsto a\}|\texttt{let } a_2 = *x;\ c_2}$$

其中上下文中包含的是程序中可以使用的变量的集合：这一次读取操作使得程序可以通过临时变量 a_2 使用地址 x 中存储的值。同样地，假设程序的第二步是读取了 y 中的值存储于临时变量 b_2，则剩下的子问题如下所示。

$$\{x, y, a_2, b_2\}; \{x \mapsto a_2 * y \mapsto b_2\} \rightsquigarrow \{x \mapsto b_2 * y \mapsto a_2\}|c_3$$

接下来，假设程序将 b_2 写入到了地址 x 中，则可以进行如下推导。根据分离逻辑

$$\frac{\{x, y, b_2\}; \{x \mapsto b_2 * y \mapsto b_2\} \rightsquigarrow \{x \mapsto b_2 * y \mapsto a_2\}|c_4}{\{x, y, a_2, b_2\}; \{x \mapsto a_2 * y \mapsto b_2\} \rightsquigarrow \{x \mapsto b_2 * y \mapsto a_2\}| * x = b_2; c_4}$$

中的定理 $\{P\}p\{Q\} \iff \{P * H\}p\{Q * H\}$，可以消去两侧的 $x \mapsto b_2$，得到如下子问题。

$$\{y, b_2\}; \{y \mapsto b_2\} \rightsquigarrow \{y \mapsto a_2\}|c_5$$

通过同样的操作,在将 a_2 写入地址 x 且消去 $y \mapsto a_2$ 后,可以得到子问题 $\{\};\{emp\} \rightsquigarrow \{emp\}||c_7$。此时 c_7 取空程序显然是该问题的一个解。综上，可以得到如下程序。

$$\texttt{let } a_2 = *x; \texttt{let } b_2 = *y; *x = b_2; *y = a_2;$$

根据推理规则的正确性，该系统保证通过证明构造得到的程序一定正确，从而避免了验证的问题。此外，在搜索证明的过程中，该系统使用了一系列的搜索策略，从而使得合成器足以用于合成复杂的堆操作程序。下面列举了其中几条较为重要的策略。

- 在推理系统中包含了一系列的等价变化，例如例 6 中的读取操作。对于任意子问题 $\mathcal{T}_1$，假设在应用某条等价变换后得到的子问题为 $\mathcal{T}_2$。如果 $\mathcal{T}_2$ 的合成问题失败了，则可以 $\mathcal{T}_1$ 必然失败，可以直接跳过。
- 在遇到递归谓词时，搜索过程只关心先展开所有递归谓词，再化简的证明过程。
- 推理系统中的一些规则是可交换的，例如例 6 中读取 x 的规则与读取 y 的规则。因此，该系统在规则上定义了一个全序。如果当前证明序列中有两条规则可交换，且它们逆序，则当前搜索分支可以直接被跳过。

1.4 归纳程序合成

归纳程序合成方法从具体的例子（例如输入输出样例）出发，尝试归纳得到一个满足所有例子的程序。尽管归纳程序合成器以例子为约束，它们可以通过预言制导下的归纳程序合成这一框架来求解通用的语法制导程序合成问题 [25]。与演绎程序合成相比，归纳程序合成放弃了规约中的语法信息，从而增加了程序合成的难度。相对应的，这一限制为归纳程序合成方法带来了两方面的好处。

- 首先，归纳程序合成方法具有更好的通用性。它们可以通过预言制导下的程序合成框架应用到大量不同场景、具有不同类型规约的程序合成任务中
- 其次，归纳程序合成方法的使用代价更低。在实际的程序合成任务中，例子通常是最容易获取的规约形式。

按照技术特点，已有的归纳程序合成方法可以分成四类：基于枚举的合成方法 [12, 26]，基于约束求解的合成方法 [7, 27]，基于空间表示的合成方法 [10, 28–30] 和基于合一化的合成方法 [31–33]。同时，近期也有一些工作尝试将这些合成方法与概率模型相结合，从而进一步地提升合成效率 [34–36]。

在这一节中，本文将首先介绍预言制导下的归纳程序合成框架，并接着对归纳程序合成的几类方法分别展开讨论。

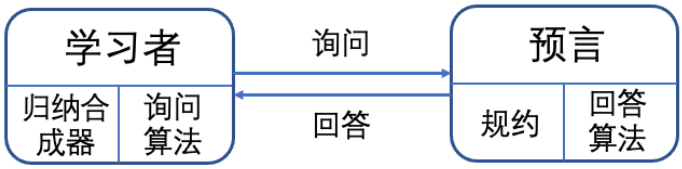

图 1.1　预言制导下的归纳程序合成框架

1.4.1　预言制导下的归纳程序合成

预言制导下的归纳程序合成（Oracle guided inductive synthesis, OGIS）框架由 Jha 等人于 2015 年整理提出 [25]。如图 1.1 所示，该框架描述了一个由两个组件，*学习者*和*预言*，构成的程序合成器。

- 学习者是一个封装后的归纳程序合成器。它不断地通过向预言提出询问来获得例子，并用归纳程序合成器来从这些例子中生成程序。
- 预言接收学习者的询问，并根据程序合成问题的规约来生成适当的回答。

在实践中，预言制导下的归纳程序合成有两类值得注意的实例，分别对应*反例制导的程序合成*与*交互式程序合成*。

1.4.1.1　反例制导的程序合成

反例制导的程序合成（Counterexample-guided inductive synthsis, CEGIS）可以处理一般性的语法制导合成问题 [37]。给定约束为 $\Phi = \exists f_1, \ldots, f_n, \forall \overline{x}, \phi(f_1, \ldots, f_n, \overline{x})$ 的语法制导程序合成问题，在 CEGIS 框架中学习者与预言的行为如下所示。

- 学习者从已有的例子中生成候选程序 $(f_1, \ldots, f_n)$，并向预言提出询问“使候选程序 $(f_1, \ldots, f_n)$ 不满足规约的一组反例”。如果预言的答案是一个反例 e，则学习者将这个反例纳入考虑，并生成下一组候选程序。否则说明反例不存在，即 $(f_1, \ldots, f_n)$ 合法。这时 CEGIS 过程结束，$(f_1, \ldots, f_n)$ 被作为结果返回。
- 预言接受询问 $(f_1, \ldots, f_n)$，并调用 SMT 求解器求解公式 $\exists \overline{x}, \neg\phi(f_1, \ldots, f_n, \overline{x})$。如果该公式不可满足，则说明 $(f_1, \ldots, f_n)$ 是一组合法解。否则预言得到一组 $\overline{x}$ 的赋值 e，对应当前候选程序的一组反例。这时 e 会被作为答案提供给学习者。

例 7 *表 1.4 展示了用 CEGIS 求解例 1 中的语法制导合成问题时的一个求解过程。*

1.4.1.2　交互式程序合成

在交互式程序合成 [38] 中，学习者与预言的行为如下所示。

表 1.4　CEGIS 对例 1 中的合成问题的求解过程。

轮次	例子集合	候选程序	反例
1	$\emptyset$	x	$e_1 = \langle x = 0, y = 1\rangle$
2	$\{e_1\}$	y	$e_2 = \langle x = 1, y = 0\rangle$
3	$\{e_1, e_2\}$	1	$e_3 = \langle x = 0, y = 0\rangle$
4	$\{e_1, e_2, e_3\}$	$x + y$	$e_4 = \langle x = 1, y = 1\rangle$
5	$\{e_1, e_2, e_3, e_4\}$	$ite(x \le y, y, x)$	$\perp$

.

- 学习者选定一个输入 I 并向预言提出询问“目标程序在输入 I 时的输出”。输入 I 可以是随机选定的，也可以由特定的询问选择算法 [39] 给出。当输入输出对数量足够时，学习者调用归纳程序合成器，并将合成结果返回。
- 预言接受询问 I，并将对应的输出作为答案提供给学习者。

交互式程序合成通常用于完整规约很难获得但是目标程序的语义容易获得的场景，例如代码去混淆 [8, 9]，或者与用户直接交互的合成任务 [40–42]。

1.4.1.3　逐点的语法制导合成问题与关系的语法制导合成问题

因为本章专注于对语法制导合成问题的求解算法，所以在本节的剩余部分中，本文假设归纳程序合成器的例子均是由 CEGIS 框架产生的。此时，语法制导合成问题可以按照例子的形式分成*逐点*和*关系*两类 [43]。

- 一个语法制导合成问题是逐点的当且仅当 (1) 它只涉及一个目标程序 f，且 (2) 约束 Φ 中所有对 f 的调用对应的输入都相同，即 Φ 可以写成如下的形式。

$$\Phi = \exists f, \forall \overline{x}, \phi'(f(p_1(\overline{x}), \ldots, p_k(\overline{x}), \overline{x})) \tag{1.1}$$

- 一个语法制导合成问题是关系的当且仅当它涉及多个目标程序或者约束 Φ 涉及 f 在多个不同输入下的输出。

例 8 *例 1 中的语法制导合成问题是逐点的，因为它只涉及一个目标程序 f 且约束 Φ 中只涉及 f 在输入 (x, y) 时的输出。如果将该问题中的约束修改成 $\exists f, \forall x, y \in Int, f(x, y) = f(y, x)$，则这一问题会变为关系的，因为在约束中涉及到了 f 在两个不同输入下的输出 $f(x, y)$ 和 $f(y, x)$。*

在对逐点的语法制导合成问题使用 CEGIS 框架时，所有产生的例子都可以转化成关于目标程序 f 的输入输出样例。假设约束形式如公式 1.1 所示，则反例 e 对应了关于 f 的一个输入输出样例 $(p_1(e), \ldots, p_k(e)) \mapsto y$，其中 y 是可满足性问题 $\exists y, \phi'(y, e)$ 的一个解，可以通过调用 SMT 求解器得到。

而对于关系的语法制导合成问题，使用 CEGIS 框架与归纳程序合成器进行求解通常被认为是更困难的。一方面，关系的语法制导合成问题涉及了多个目标程序，导致解空间大小产生了指数爆炸。另一方面，在对关系的语法制导合成问题使用 CEGIS 框架时，产生的例子的形式往往比输入输出样例更加复杂，从而导致一些针对程序合成的优化算法难以使用。

因此，在本节的剩余部分介绍归纳合成方法时，本文先在逐点的语法制导合成问题上介绍合成方法的主要思想，即假设问题中涉及单一目标程序、单一文法以及输入输出样例，然后再讨论这些合成方法在关系的语法制导合成问题上的扩展。

1.4.2　基于枚举的合成方法

1.4.2.1　朴素的枚举方法

给定上下文无关文法 G 与一系列输入输出样例 $I_i \mapsto O_i$，基于枚举的合成方法依次遍历 G 中的所有程序，直到遇到一个满足所有输入输出样例的程序。其中，遍历 G 中所有程序的方法可以分为自顶向下和自底向上两类。

- 自顶向下的枚举从文法 G 的起始非终结符出发，依次枚举所有可能的最左展开。在遍历过程中，如果当前的展开结果只包含终结符，则表示当前遍历到了一个程序。否则该方法会选择最左边的非终结符出发，依次尝试所有可能的规则，并将展开结果放入遍历队列的尾端。
- 自底向上的枚举按照大小枚举所有非终结符上的所有程序。对于每一个大小，每一个非终结符，该方法会尝试所有可能的展开式，所有将程序大小分配到展开后的非终结符上的可能性，并用这些非终结符对应大小的程序集合构造当前非终结符上当前大小的程序集合。

例 9 考虑例 1 中的上下文无关文法 G。下面展示了用自顶向下的方法遍历 G 中程序时，最开始遍历到的一些展开式，其中蓝色表示遍历到的程序。

- 零次展开：I。
- 一次展开：$x, y, 1, I + I, ite(B, I, I)$。
- 两次展开：$x + I, y + I, 1 + I, (I + I) + I, ite(I \le I, I, I)$。
- 三次展开：$x + x, x + y, x + 1, x + (I + I), x + ite(B, I, I), y + x, y + y, y + 1 \ldots$

下面展示了用自底向上的方法遍历 G 中程序时，最开始遍历到的一些程序，其中蓝色表示枚举到的由起始非终结符派生出的程序。

- （大小为一）I 派生出 $x, y, 1$，B 无派生出的程序。
- （大小为二）I 和 B 均无派生出的程序。

- （大小为三）I 派生出 $x+x, x+y, x+1, y+x, y+y, y+1, 1+x, 1+y, 1+1$，$B$ 派生出 $x \le x, x \le y, x \le 1, y \le x, y \le y, y \le 1, 1 \le x, 1 \le y, 1 \le 1$。

两种遍历方法都是按照大小递增的顺序遍历 G 中的程序。同时，例 9 可以看出，两种遍历方法在遍历 G 中的程序以外，都存在着额外的开销。

- 自顶向下的遍历方法无法保证遍历到的每一个展开式都是 G 中的程序，其中一部分包含着 G 中的非终结符。
- 自底向上的遍历方法无法保证遍历到的每一个程序都是从初始非终结符派生而来的，其中有一部分是由其他非终结符派生而来的。

一般来说，自底向上的遍历方法使用小程序来不断构造大程序，因此更适合于剪枝：当一个冗余的程序被剪枝策略跳过时，所有包含该程序的程序也被一并跳过了。而自顶向下的遍历方法可以建模成图上的寻路问题，因此更容易与概率模型相结合。本文会在第 1.4.6 中对这一点进行展开讨论。

1.4.2.2 剪枝策略

因为上下文无关文法中存在着语义等价的程序（如例 9 中的 $x+y$ 和 $y+x$），所以直接遍历 G 中的所有程序往往会带来不必要的开销。在实践中，基于枚举的合成方法的性能往往可以通过剪枝策略得到很大的提升。下面是几种常见的用于自底向上遍历的剪枝策略。

- 规则法。求解器可以利用预先设计的规则来跳过冗余等价的程序。以加法的交换律为例，因为对于任意两个程序 p_1, p_2，p_1+p_2 与 p_2+p_1 始终语义等价，所以给定程序上的全序 $<_p$，求解器可以跳过所有形如 p_1+p_2 且 $p_2 <_p p_1$ 的程序。
- 约束求解法。求解器可使用 SMT 求解器来直接检测两个程序的语义等价性。给定两个程序 p_1, p_2，如果 $\exists \overline{x}, p_1(\overline{x}) \neq p_2(\overline{x})$ 不可满足，则说明 p_1, p_2 语义等价，其中一个程序可以被求解器跳过。
- 观察等价法 [26]。因为求解器的目标是找到一个满足所有输入输出样例 $I_i \mapsto O_i$ 的程序，所以一旦有两个程序 p_1, p_2 在所有输入 I_i 的输出均相同，它们就可以被视为等价，从而求解器可以跳过它们中的任何一个。

比较三种剪枝策略，从剪枝效果的角度来说，这三种策略从弱到强的顺序为规则法、约束求解法和观察等价法，其中规则法只根据预先设定的规则跳过了一部分语义等价的程序，约束求解法跳过了所有语义等价的程序，而观察等加法在语义等价的程序外，还跳过了那些虽然一般情况下语义不等价，但是在当前问题的所有输入 I_i 下输出相同的程序。而从额外开销的角度来说，三种策略从小到大的顺序通常为规则法、观察等加法和约束求解法，其中规则法只需要利用程序的语法信息，观察等加法需要在

所有输入下计算当前程序的输出，而约束求解法需要调用 SMT 求解器。综合两点，在实践中，输入输出样例数量较少时，往往是观察等价法效果最好，而当样例数量较多时，规则法的效果会更好一些。

1.4.2.3　对关系的语法制导合成问题的扩展

两种遍历方法都可以简单地从逐点的语法制导合成问题推广到关系的语法制导合成问题中。假设当前问题涉及的目标程序为 $f_1, \ldots, f_n$，上下文无关文法为 $G_1, \ldots, G_n$，对应的初始非终结符为 $S_1, \ldots, S_n$。

- 自顶向下的遍历方法从非终结符元组 $(S_1, \ldots, S_n)$ 出发，枚举所有可能的最左展开，从而可以遍历程序元组 $(f_1, \ldots, f_n)$ 所有可能的取值。
- 自底向下的遍历方法引入额外规则 $S \to (S_1, \ldots, S_n)$，并以 S 为初始非终结符。这时，S 派生出的“程序”对应了程序元组 $(f_1, \ldots, f_n)$ 所有可能的取值。

关于剪枝策略，规则法与约束求解法都可以用于扩展后的自底向上遍历过程。而观察等加法由于依赖于输入输出用例，因此无法用于一般性的关系的语法制导合成问题。

1.4.3　基于约束求解的合成方法

给定上下文无关文法 G 与一系列输入输出样例 $I_i \mapsto O_i$，基于约束求解的合成方法将 G 中程序的语法与语义编码成约束，并使用 SMT 求解器解出满足所有输入输出样例的程序。具体来说，该方法编码了两种类型的约束。

- 结构约束 $\phi_s(\overline{x_s})$，其中 $\overline{x_s}$ 是一系列的结构变量。该约束表示结构变量 $\overline{x_s}$ 对应的程序是否是一个合法的程序，且是否在上下文无关文法 G 中。
- 语义约束 $\phi_v(\overline{x_s}, \overline{x_i}, x_o, \overline{x_t})$，其中 $\overline{x_s}$ 是结构变量，$\overline{x_i}$ 是一系列代表输入的变量，x_o 是一个代表输出的变量，$\overline{x_t}$ 是一系列的临时变量。该约束表示结构变量 $\overline{x_s}$ 对应的程序在输入 $\overline{x_i}$ 时的输出是否是 x_o。

给定编码方法，满足下列约束的一组对结构变量 $\overline{x_s}$ 的赋值就对应了一个合法程序，而该赋值可以通过调用 SMT 求解器解出。

$$\exists \overline{x_s}, \left(\phi_s(\overline{x_s}) \wedge \bigwedge_{I_i \mapsto O_i} \exists \overline{x_t^i}, \phi_v(\overline{x_s}, I_i, O_i, \overline{x_t^i}) \right)$$

1.4.3.1　基于电路的编码方法

Jha 等人于 2010 年提出了基于程序的电路表示的编码方法 [7]。图 2.1 展示了该编码方法的一个实例。该编码方式假设用户给定了一系列可用的组件，包括所有的输入、可用的常量以及可用的操作符，并将每一个程序视为连接这些组件的一个电路。

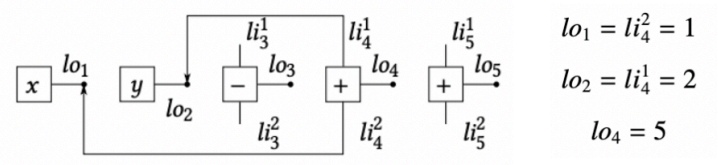

图 1.2　基于电路表示编码程序 $x + y$ 的图示，其中左图展示了电路的连接，右图展示了对应的一组对结构变量的（部分）赋值。

假设给定了 n 个组件 $c_1, \ldots, c_n$，其中第 i 个组件接受 t_i 个输入。基于电路的编码方法假设存在 n 个连接点，编号分别为 1 到 n，其中编号为 n 的连接点表示了程序的输出。该方法对于每一个组件引入了两类结构变量来表示电路的连接。对于第 i 个组件，li_i^j 表示了它的第 j 个输入从哪一个连接点处获得，而 lo_i 表示了它输出到哪一个连接点。为了保证电路合法，基于电路的编码方式引入了如下两类约束 $\phi_s^1(\overline{x_s})$ 与 $\phi_s^2(\overline{x_s})$，并定义结构约束 $\phi_s(\overline{x_s}) \coloneqq \phi_s^1(\overline{x_s}) \wedge \phi_s^2(\overline{x_s})$。

- $\phi_s^1(\overline{x_s})$ 保证了每一个组件都输出到不同的连接点。

$$\phi_s^1(\overline{x_s}) \coloneqq \bigwedge_{i=1}^{n} (lo_i \geq 1 \wedge lo_i \leq n) \wedge \bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} \left(lo_i \neq lo_j\right)$$

- $\phi_s^2(\overline{x_s})$ 保证了电路无环，即每个组件的输入不会依赖自己的输出。

$$\phi_s^2(\overline{x_s}) \coloneqq \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{t_i} \left(li_i^j \geq 1 \wedge li_i^j < lo_i\right)$$

给定结构变量，即程序的电路表示，语义约束 ϕ_v 可以直接从电路的结构以及组件的语义得到。具体来说，在语义约束中，基于电路表示的编码方法对每一个组件引入了两类临时变量来编码电路的语义。对于第 i 个组件，vi_i^j 表示了它第 j 个输入的值，vo_i 表示了它输出的值。根据电路结构，基于电路的编码方式引入了三类语义约束 $\phi_v^1(\overline{x_s}, x_o, \overline{x_t})$, $\phi_v^2(\overline{x_s}, \overline{x_t})$ 和 $\phi_v^3(\overline{x_i}, \overline{x_t})$，并定义结构约束 $\phi_v(\overline{x_s}, \overline{x_i}, x_o, \overline{x_t})$ 为它们的合取。

- $\phi_v^1(\overline{x_s}, x_o, \overline{x_t})$ 保证了程序的输出等于连接点 n 处的输出。

$$\phi_v^1(\overline{x_s}, x_o, \overline{x_t}) \coloneqq \bigwedge_{i=1}^{n} (lo_i = n \rightarrow x_o = vo_i)$$

- $\phi_v^2(\overline{x_s}, \overline{x_t})$ 保证了组件的运算结果按照电路的连接情况在不同组件之间传递。

$$\phi_v^2(\overline{x_s}, \overline{x_t}) \coloneqq \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \bigwedge_{k=1}^{t_i} \left(lo_i = li_j^k \rightarrow vo_i = vi_j^k\right)$$

- $\phi_v^3(\overline{x_i}, \overline{x_t})$ 保证了组件的运算结构符合组件的语义，其中 $c_i[\overline{x_i}](v_1, \ldots, v_{t_i})$ 表示第

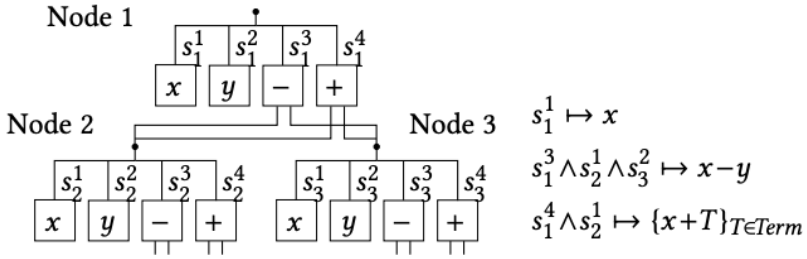

图 1.3　基于语法树编码程序的图示，其中左图展示了符号化的抽象语法树，右图展示了不同的对结构变量 s_i^j 的赋值对应的程序。

i 个组件在（1）输入 $v_1, \ldots, v_{t_i}$ 且（2）变量取值为 $\overline{x_i}$ 时的输出。

$$\phi_v^3(\overline{x_i}, \overline{x_t}) \coloneqq \bigwedge_{i=1}^{n} \left(vo_i = c_i[\overline{x_i}] \left(vi_i^1, \ldots, vi_i^{t_i} \right) \right)$$

1.4.3.2　基于语法树的编码方法

在基于电路的编码方法中，结构变量 $\overline{x_s}$ 是一系列的整型变量，表示了每一个组件的输入输出分别与哪一个连接点相连。然而，对于约束求解来说，处理整型约束的时间开销是要高于处理布尔约束的。受到这点的启发，Mechtaev 等人提出了一种基于语法树的编码方法，其中只涉及布尔类型的结构变量 [27]。该方法假设目标程序的语法树结构已经给定，并使用一系列的布尔变量来表示语法树上每一个节点分别对应哪一个组件。图 1.3 展示了基于语法树编码程序的一个实例。

假设语法树上存在 n 个节点，根节点编号为 1，第 i 个节点的第 j 个子节点编号为 t_i^j，且第 i 节点上可用的组件集合是 C_i。基于语法树的编码方法对每一个节点 i，以及该节点上每一个可用的组件 c 均引入了一个布尔类型的结构变量 s_i^c，表示第 i 个节点使用的组件是否为 c。为了保证语法树合法，该方法的结构约束 $\phi_s(\overline{x_s})$ 保证了每一个节点上有且只有一个组件被使用，其定义如下。

$$\phi_s(\overline{x_s}) \coloneqq \bigwedge_{i=1}^{n} exactlyOne\left(\{ x_i^c | c \in C_i \} \right)$$

其中 $exactlyOne(\overline{x})$ 表示 $\overline{x}$ 中的布尔变量有且只有一个为真。使用排序网络，它可以被编码成一个使用了 $O(|\overline{x}|)$ 个临时变量与 $O(|\overline{x}|)$ 个子句的逻辑表达式 [44]。

给定结构变量，即语法树上每一个节点使用的组件，语义约束 ϕ_v 可以直接从语法树结构与每一个组件的语义得到。具体来说，在语义约束的中，基于语法树的编码方法为每一个节点 i 引入了一个临时变量 vo_i，表示该节点的子树对应程序片段的输出。根据语法树结构，该方法的语义约束 $\phi_v(\overline{x_s}, \overline{x_i}, x_o, \overline{x_v})$ 定义如下，其中 $k(c)$ 表示组件 c 需要的输入数量，$c[\overline{x_i}](v_1, \ldots, v_{k(c)})$ 表示组件 c 在（1）输入 $v_1, \ldots, v_{k(c)}$ 且（2）变量取值

为 $\overline{x_i}$ 的输出。

$$\phi_v(\overline{x_s}, \overline{x_i}, x_o, \overline{x_v}) \coloneqq x_o = vo_1 \wedge \bigwedge_{i=1}^{n} \bigwedge_{c \in C_i} \left(s_i^c \rightarrow vo_i = c[\overline{x_i}] \left(vo_{t_i^1}, \ldots, vo_{t_i^{k(c)}} \right) \right)$$

1.4.3.3 对关系的语法制导合成问题的扩展

基于约束求解的合成方法可以用于求解关系的语法制导合成问题。假设当前合成问题涉及 n 个目标程序 $f_1, \ldots, f_n$，约束为 $\Phi = \exists f_1, \ldots, f_n, \forall \overline{x}, \phi(f_1, \ldots, f_n, \overline{x})$，以及给定的例子集合为 E。给定编码方法，令 $\overline{x_s^i}$ 表示第 i 个程序的结构变量，则满足下列约束的一组对所有结构变量的赋值对应了一组满足所有例子的合法程序 $(f_1, \ldots, f_n)$，而该赋值可以通过调用 SMT 求解器解出。

$$\exists_{i=1}^{n} \overline{x_s^i}, \left(\bigwedge_{i=1}^{n} \phi_s(\overline{x_s^i}) \wedge \bigwedge_{e \in E} \exists \overline{x_t}, \Phi_v(\overline{x_s^1}, \ldots, \overline{x_s^n}, e, \overline{x_t}) \right)$$

其中 $\Phi_v(\overline{x_s^1}, \ldots, \overline{x_s^n}, e, \overline{x_t})$ 表示将 ϕ 中所有对 f_i 的调用都替换成使用结构变量 $\overline{x_s^i}$ 的语义约束 ϕ_v，所有对 $\overline{x}$ 的使用都替换成例子 e 中的赋值后，得到的约束，其中 $\overline{x_t}$ 表示在这个过程中使用到的所有中间变量。

例 10 考虑约束为 $\Phi = \exists f_1, f_2, \forall x, y \in \mathit{Int}, f_1(f_2(x)) = f_1(y) + x$ 的合成问题，以及例子 $e = \langle x = 1, y = 2 \rangle$。此时，约束 Φ_v 的定义如下所示，其中 $\overline{x_t} = \{x_o^1, x_o^2, x_o^3\} \cup \overline{x_t^1} \cup \overline{x_t^2} \cup \overline{x_t^3}$，变量 x_o^1, x_o^2 与 x_o^3 分别表示函数调用 $f_1(f_2(x)), f_2(x)$ 与 $f_1(y)$ 的输出，变量集合 $\overline{x_t^1}, \overline{x_t^2}$ 与 $\overline{x_t^3}$ 分别表示对应这三个调用的语义约束中使用的临时变量。

$$\Phi_v(\overline{x_1}, \overline{x_2}, e, \overline{x_t}) \coloneqq \phi_v\left(\overline{x_1}, x_o^2, x_o^1, \overline{x_t^1}\right) \wedge \phi_v\left(\overline{x_2}, 1, x_o^2, \overline{x_t^2}\right) \wedge \phi_v\left(\overline{x_3}, 2, x_o^3, \overline{x_t^3}\right) \wedge x_o^1 = x_o^3 + 1$$

1.4.4 基于空间表示的合成方法

给定上下文无关文法 G 与一系列输入输出样例 $I_i \mapsto O_i$，基于空间表示的合成方法使用变形代数空间 (version space algebra, VSA) [45] 表示 G 中满足输入输出样例的程序集合，并将该空间中的任意一个程序作为生成结果返回。

1.4.4.1 变形代数空间

变形代数空间可以视为一种特殊的上下文无关文法，它包含了如下三类节点。

- 第一类节点直接表示了一个程序集合 $\{p_1, \ldots, p_k\}$。它可以被视为上下文无关文法中具有展开式 $S \rightarrow p_1, \ldots, S \rightarrow p_k$ 的非终结符 S。
- 第二类节点 $\mathbf{U}(N_1, \ldots, N_k)$ 表示了一些其他节点对应的程序集合的并。它可以被视为上下文无关文法中具有展开式 $S \rightarrow S_1, \ldots, S \rightarrow S_k$ 的非终结符 S，其中 S_i 是对应节点 N_i 的非终结符。

- 第三类节点 $f(N_1,\ldots,N_k)$ 表示所有形如 $f(p_1,\ldots,p_k)$ 的程序形成的集合，其中 p_i 在节点 N_i 对应的程序集合中。它可以被视为上下文无关文法中具有展开式 $S \to f(S_1,\ldots,S_k)$ 的非终结符 S，其中 S_i 是对应节点 N_i 的非终结符。

例 11 定义上下文无关文法 G 如下，其中 S 为文法 G 的初始非终结符，操作符 $if(p_1,p_2)$ 的语义为 $ite(p_1 \le p_2, x, y)$。

$$S \to if(E,E) \mid E \qquad E \to 0 \mid x \mid y$$

文法 G 对应的程序集合等加入如下变形代数空间 G' 中节点 S 对应的程序集合。

$$S \coloneqq \boldsymbol{U}(S_1,E) \qquad S_1 \coloneqq if(E,E) \qquad E \coloneqq \{0,x,y\}$$

基于空间表示的合成方法要求合成问题中的上下文文法 G 是以变形代数空间的形式给出的。给定变形代数空间 G 和输入输出样例 $e = I \mapsto O$，包含 G 中所有满足样例 e 的程序的代数空间（记作 $G[e]$）可以通过在节点上标记预期输出来构造得到。简单来说，$G[e]$ 中的每一个节点都形如 $\langle S,v\rangle$，其中 S 是 G 中的一个节点，v 是一个可能的输出：该节点对应的程序集合等于（1）在节点 S 对应的程序集合中且（2）满足输入输出样例 $I \mapsto v$ 的程序形成的集合。$G[e]$ 中节点的结构可以通过 G 的结构以及操作符的语义构造得到。例 12 展示了这一构造过程的一个实例。

例 12 考虑例 11 中的变形代数空间 G' 与输入输出样例 $e = \langle x=0, y=1\rangle \to 0$。如下变形代数空间 $G'[e]$ 中的节点 $\langle S,0\rangle$ 描述了 G' 中所有满足样例 e 的程序集合。

$$\langle S,0\rangle \coloneqq \boldsymbol{U}(\langle E,0\rangle,\langle S_1,0\rangle) \qquad \langle S_1,0\rangle \coloneqq \boldsymbol{U}(\langle S_1,0\rangle_1,\langle S_1,0\rangle_2,\langle S_1,0\rangle_3)$$

$$\langle S_1,0\rangle_1 \coloneqq if(\langle E,0\rangle,\langle E,0\rangle) \qquad \langle S_1,0\rangle_1 \coloneqq if(\langle E,0\rangle,\langle E,1\rangle) \qquad \langle S_1,0\rangle_1 \coloneqq if(\langle E,0\rangle,\langle E,0\rangle)$$

$$\langle E,0\rangle \coloneqq \{x,0\} \qquad \langle E,1\rangle \coloneqq \{y\}$$

当给定多个输入输出样例 $e_1,\ldots,e_n$ 时，对应同时满足所有样例的变形代数空间 $G[e_1,\ldots,e_n]$ 可以通过对满足单一样例的变形代数空间 $G[e_1],\ldots,G[e_n]$ 依次求交得到。变形代数空间中两个节点的求交操作 $N \cap N'$ 定义如下。

$$\mathbf{U}(N_1,\ldots,N_k) \cap N' \coloneqq \mathbf{U}(N_1 \cap N',\ldots,N_k \cap N')$$

$$f(N_1,\ldots,N_k) \cap f(N'_1,\ldots,N'_k) \coloneqq f(N_1 \cap N'_1,\ldots,N_k \cap N'_k)$$

$$f(N_1,\ldots,N_k) \cap f'(N_1,\ldots,N'_{k'}) \coloneqq \emptyset$$

$$f(N_1,\ldots,N_k) \cap \{p'_1,\ldots,p'_{k'}\} \coloneqq \left\{p'_i = f(p_1,\ldots,p_k) \mid \forall j \in [1,k], p_j \in N_j\right\}$$

$$\{p_1,\ldots,p_k\} \cap \{p'_1,\ldots,p'_{k'}\} \coloneqq \left\{p_i \mid \exists j \in [1,k'], p_i = p'_j\right\}$$

上述求交操作会增加变形代数空间中的节点数量。给定两个分别包含 n 个节点和 m 个节点的变形代数空间，可以证明按照上述定义构造得到的求交结果中，至多包含 $O(nm)$ 个节点。

根据以上描述，可以发现变形代数空间的效率与如下两个因子有关。

- 可能的中间结果数量。给定上下文无关文法 G 与输入输出样例 $e = I \mapsto O$，令 P_e 为 G 中满足样例 e 的所有程序形成的集合，令 S_e 为 P_e 中程序的所有子表达式形成的集合，令 O_e 为 S_e 中表达式在输入为 I 时的输出形成的集合，即 $\{p(I)|p \in S_e\}$。直观上来说，O_e 包含了任意一个满足 e 的程序中用到的所有中间结果。根据变形代数空间 $G[e]$ 的构造过程，可以证明 $G[e]$ 中至多包含 $O(|G| \times |O_e|)$ 个节点。因此可能的中间结果数量越少，变形代数空间描述程序空间的效率就越高。
- 因为对多个变形代数空间求交可能会让节点数量指数级的上升，所以程序合成问题中使用的输入输出样例数量越少，变形代数空间描述程序空间的效率越高。

综上，基于空间描述的程序合成方法通常适用于（1）可能的中间结果数量较少（2）输入输出样例数量较少的场景，例如字符串变换 [10]，表格格式化 [46] 等。相比之下，这一类合成方法在合成整数表达式时效果较差，原因如下。

- 对于任意样例 $I \mapsto O$，任何中间结果 x 都可以通过加上 $O - x$ 来得到期望输出 O。因此合成整数表达式时会涉及大量可能的中间结果。
- 整数表达式通常会用到分支语句。为了准确描述不同分支的条件与行为，在合成整数表达式的过程中可能会引入大量的输入输出样例。

1.4.4.2 变形代数空间的构造方法概述

注意到在上一节中，虽然我们定义了用于表示满足样例 e 的程序的变形代数空间 $G[e]$，但是我们并未给出一个高效的构造 $G[e]$ 的方法。给定变形代数空间 G 和输入输出样例 $e = I \mapsto O$，下面描述了源自 $G[e]$ 定义的一个朴素的构造方法。

1. 对于 G 中的每一个节点 S 和每一个可能的输出 v，创建节点 $\langle S, v\rangle$，表示节点 S 对应的集合中在输入为 I 时输出为 v 的程序形成的集合。
2. 令 S_0 为 G 的初始节点，将 $G[e]$ 的初始节点设为 $\langle S_0, v\rangle$。
3. 根据节点的定义与操作符的语义构造 $G[e]$ 中节点的结构。
4. 删去所有冗余节点，包括对应程序集合为空集的*空节点*与无法从初始节点 $\langle S_0, v\rangle$ 到达的*不可达节点*。

因为可能的输出的数量通常很大（甚至无穷），所以朴素的构造方法会在第一步中引入大量的冗余节点，从而带来大量的额外开销。为了减少冗余节点数量，已有工作使用了不同方式来优化变形代数空间 $G[e]$ 的构造。与基于枚举的合成方法类似，变形代数空间 $G[e]$ 的基本构造方法可以分为*自顶向下* [28] 与*自底向上* [30] 两类。此外，这

自底向上的构造方法的效率还可以通过抽象精化的方法得到进一步地提升 [29, 47]。

1.4.4.3　自顶向下的变形代数空间构造方法

Polozov 等人于 2015 年提出了 FlashMeta 程序合成框架，其中使用了自顶向下的方法来构造变形代数空间 $G[e]$ [28]。FlashMeta 使用见证函数 (witness function) 来将期望输出 O 分解成可能用到的中间结果，并只对这些中间结构构造节点。直观上来说，见证函数描述了一个操作符的逆函数。见证函数以一个操作符的输出为约束，并输出一系列的输入组合，表示操作符在给定这些输入时会产生给定的输出。

定义 1 (见证函数) *给定全局输入 I 与操作符 $f(p_1, \ldots, p_k)$，见证函数 $\omega_f[I](v)$ 输出一个以 k 元组为元素的集合，满足如下条件。*

$$\forall (w_1, \ldots, w_k) \in \omega_f[I](v), f[I](w_1, \ldots, w_k) = v$$

其中 $f[I](v_1, \ldots, v_k)$ 表示 f 在（1）输入 $v_1, \ldots, v_k$ 且（2）变量取值为 I 时的输出。

例 13 *变量 x 与常量 c 的见证函数 $\omega_x[I](A)$ 和 $\omega_c[I](A)$ 如下所示。*

$$\omega_x[I](v) \coloneqq \begin{cases} \{()\} & I(x) = v \\ \emptyset & I(x) \neq v \end{cases} \qquad \omega_c[I](v) \coloneqq \begin{cases} \{()\} & c = v \\ \emptyset & c \neq v \end{cases}$$

令 + 表示字符串链接操作符。给定两个字符串 s_1, s_2，$s_1 + s_2$ 的结果为 s_1, s_2 连接后得到的字符串。该操作符的一个见证函数 $\omega_+[I](s)$ 如下所示。

$$\omega_+[I](s) \coloneqq \left\{ \Big(s[1, i], s[i + 1, |s|] \Big) \;\middle|\; i \in [1, |s|) \right\}$$

其中 $|s|$ 表示字符串 s 的长度，$s[l, r]$ 表示由 s 中第 l 个至第 r 个字符形成的字符串。下面是该见证函数在给定输出为 `J.Jonathan` 时的输出。

$$\{(\texttt{J}, \texttt{.Jonathan}), (\texttt{J.}, \texttt{Jonathan}) \ldots, (\texttt{J.Jonatha}, \texttt{n})\}$$

给定变形代数空间 G 与输入输出样例 $e = I \mapsto O$，令 S_0 为 G 的初始节点，FlashMeta 通过如下方式递归构造 $G[e]$ 中的初始节点 $\langle S_0, O \rangle$。假设当前需要构造的节点为 $\langle S, v \rangle$，FlashMeta 根据 S 的类型分类讨论。

- 如果 $S = \{p_1, \ldots, p_k\}$，则 $\langle S, v \rangle \coloneqq \{p \in S \mid p(I) = v\}$。
- 如果 $S = \mathbf{U}(N_1, \ldots, N_k)$，则 $\langle S, v \rangle \coloneqq \mathbf{U}(\langle N_1, v \rangle, \ldots, \langle N_k, v \rangle)$。
- 如果 $S = f(N_1, \ldots, N_k)$，令 W 为见证函数 $\omega_f[I](v)$ 的输出。
  - 如果 $W = \emptyset$，则 $\langle S, v \rangle$ 为空节点。
  - 如果 W 只包含一个元组 $(w_1, \ldots, w_k)$，则 $\langle S, v \rangle \coloneqq f(\langle N_1, w_1 \rangle, \ldots, \langle N_k, w_k \rangle)$。

– 否则，对 W 中的元组 $(w_1^i, \ldots, w_k^i)$，构造 $\langle S, v\rangle_i := f(\langle N_1, w_1^i\rangle, \ldots, \langle N_k, w_k^i\rangle)$。最后令 $\langle S, v\rangle := \mathbf{U}(\langle S, v\rangle_1, \ldots, \langle S, v\rangle_{|W|})$。

例 14 考虑如下的以 S 为初始节点的变形代数空间 G 与输入输出样例 $(FS = \mathit{John}, LS = \mathit{Jonathan}) \mapsto \mathit{J.Jonathan}$。

$$S := \boldsymbol{U}(S_x, S_+, S_C) \quad S_x := \{FS, LS\} \quad S_+ := +(S, S) \quad S_C := CharAt(S, I) \quad I := \{0, 1\}$$

其中操作符 $CharAt$ 以字符串 s 与整数 i 为输入，输出 s 中的第 i 个字符。

下面展示了 *FlashMeta* 构造变形代数空间 $G[e]$ 的部分过程。

$$\langle S, \mathit{J.Jonathan}\rangle := \boldsymbol{U}\Big(\langle S_x, \mathit{J.Jonathan}\rangle, \langle S_+, \mathit{J.Jonathan}\rangle, \langle S_C, \mathit{J.Jonathan}\rangle\Big)$$

$$\langle S_x, \mathit{J.Jonathan}\rangle := \emptyset \quad \langle S_C, \mathit{J.Jonathan}\rangle := \emptyset$$

$$\langle S_+, \mathit{J.Jonathan}\rangle := \boldsymbol{U}\Big(\langle S_+, \mathit{J.Jonathan}\rangle_1, \langle S_+, \mathit{J.Jonathan}\rangle_2, \ldots\Big)$$

$$\langle S_+, \mathit{J.Jonathan}\rangle_1 := +\Big(\langle S, \mathit{J}\rangle, \langle S, \mathit{.Jonathan}\rangle\Big)$$

$$\langle S_+, \mathit{J.Jonathan}\rangle_2 := +\Big(\langle S, \mathit{J.}\rangle, \langle S, \mathit{Jonathan}\rangle\Big)$$

直观上来说，自顶向下的构造方法通过使用见证函数，让构造过程关注于可能用于产生目标输出的那些中间结果，从而极大地减少了需要考虑的中间结果的数量。

1.4.4.4 自底向上的变形代数空间构造方法

王新宇等人于 2017 年提出了基于有限树自动机 (Finite Tree Automata) 的自底向上的变形代数空间构造方法 [30]。本文只介绍该方法的构造过程，并不对其理论模型有限树自动机作过多展开。更多理论结果可以参见论文原文。

为了表述方便，本文引入如下类型的复合节点 $N = \mathbf{U}(P_1, \ldots, P_k)$，其中 P_i 要么是一个程序集合，要么是一次调用 $f(N_1, \ldots, N_{k'})$。虽然 N 并不是一个变形代数空间中一个合法的节点，但是我们可以通过为每一个 P_i 创建一个中间节点来将 N 变成一个合法节点。本文用*并入*来表示对复合节点的修改。给定程序集合（或调用）P 和复合节点 $N = \mathbf{U}(P_1, \ldots, P_k)$，如果 P 等于某一个 P_i，则操作“将 P 并入 N”不会对 N 作出修改，否则该操作会将复合节点 N 修改为 $\mathbf{U}(P_1, \ldots, P_k, P)$。

给定变形代数空间 G 与输入输出样例 $e = I \mapsto O$，自底向上的方法通过如下方式构造变形代数空间 $G[e]$。

1. 对于 G 中每一个集合节点 $N = \{p_1, \ldots, p_n\}$，将集合 $\{p_i\}$ 并入节点 $\langle N, p_i(I)\rangle$。
2. 对于 G 中的每一个节点 $N = \mathbf{U}(N_1, \ldots, N_k)$ 与每一个值 v，如果 $\langle N, v\rangle$ 未被创建且每一个 $\langle N_i, v\rangle$ 均非空，则创建节点 $\langle N, v\rangle := \mathbf{U}(\langle N_1, v\rangle, \ldots, \langle N_k, v\rangle)$。

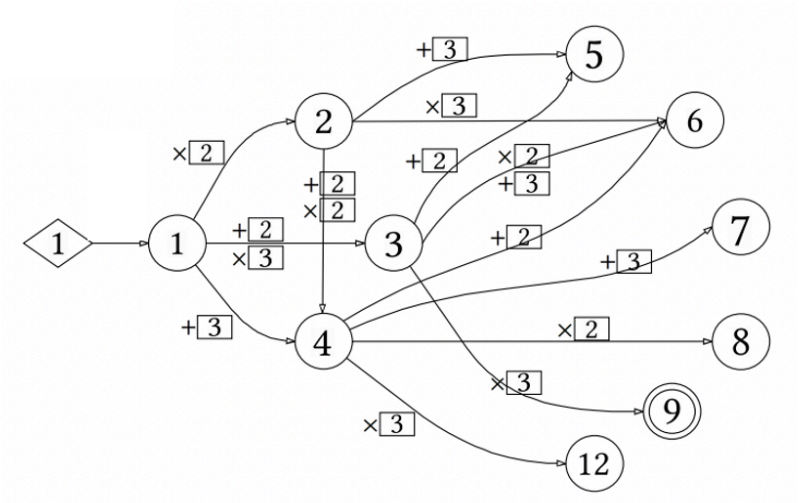

图 1.4　使用自底向上的方法构造变形代数空间的图示。

3. 对于 G 中的每一个节点 $N = f(N_1, \ldots, N_k)$ 与每一组值 $(v_1, \ldots, v_k)$，如果每一个 $\langle N_i, v_i \rangle$ 均非空，则将 $f(\langle N_1, v_1 \rangle, \ldots, \langle N_k, v_k \rangle)$ 并入节点 $\langle N, f[I](v_1, \ldots, v_k) \rangle$。
4. 重复步骤 2 和 3，直到变形代数空间的结构不再改变或者迭代轮数到达上限。

例 15 *考虑一个只包含整数输入 x 与单目运算符 $+2, +3, \times 2, \times 3$ 的程序空间 G 以及输入输出样例 $1 \mapsto 9$。图 1.4 展示了使用自底向上的方法构造变形代数空间 $G[e]$ 的过程，其中菱形节点代表了程序集合 $\{x\}$，标记为 i 的圆形节点代表了对应输出为 i 的复合节点，箭头代表了构造过程中的并入操作。*

联系上一节中介绍的自顶向下的构造方法，两种构造方法分别使用了样例中的输入与输出信息来减少构造过程中创建的冗余节点数量。

- 从构造效率来说，因为样例中的输出信息与目标程序的语义相关，而输入信息与目标程序无关，所以自顶向下的构造效率一般优于自底向上的构造效率。
- 从通用性来说，因为自顶向下的方法需要每一个操作符的见证函数，而自底向上的方法只需要语义，所以自底向上方法的通用性一般优于自顶向下的方法。

此外，自底向上的变形代数空间构造方法与使用观察等价法的自底向上枚举的合成方法存在一定的相似性。给定样例 $I \mapsto O$，对于每一个节点（非终结符）与每一个在输入 I 时可能的输出，前者只会创建一个节点，而后者只考虑第一个枚举到的程序。实际上，两个算法可以被相似的方法实现，唯一的区别在于前者会创建一个变形代数空间来描述所有可能的解，而后者只会保留枚举到的第一个合法程序。

1.4.4.5 基于抽象精化的变形代数空间构造方法

给定变形代数空间 G 与输入输出样例 e，朴素的基于空间表示的合成方法在构造变形代数空间 $G[e]$ 时，会为每一个具体的中间结果构造一个新的节点，因此朴素方法效率的一个瓶颈是这些中间结果的数量。为了减少中间结果的数量，王新宇等人于 2018 年将抽象精化的方法用于加速基于空间表示的合成方法 [29]。简单来说，该方法使用抽象语义来构造变形代数空间，从而将不同的中间结果合并成为相同的抽象值。因为使用抽象后的变形代数空间可能会生成实际上不满足样例 e 的程序，所以该方法不断根据错误的生成结果精化抽象语义，直到生成满足样例的程序。

因为基于抽象精化的构造方法的形式化描述相对复杂，本文使用一个具体的例子来介绍该方法的求解过程。为了减少不必要的细节，在本节中本文使用与上下文无关文法相同的方式来描述变形代数空间。本节中涉及的上下文无关文法均可以通过简单的变形得到等价的变形代数空间。延续例 15，本文考虑关于如下文法 G 的合成问题。

$$S \to x \mid S + C \mid S \times C \qquad C \to 2 \mid 3$$

基于抽象精化的合成方法假设给定了一个元抽象域 $\mathcal{U}$，包含了所有可用的抽象值，其中每一个抽象值都对应着一个具体值的集合。元抽象域 $\mathcal{U}$ 需要满足（1）具体值的全集 $[-\infty, \infty]$ 是一个抽象值，（2）任何两个抽象值的交集是一个抽象值，且（3）每个单独的具体值 $\{i\}$ 是一个抽象值。一个抽象域 $\mathcal{P}$ 是一个满足上述前两个条件的元抽象域 $\mathcal{U}$ 的子集。对于任意抽象值 $\varphi \in \mathcal{U}$，定义其在抽象域 $\mathcal{P}$ 上的取值 $\alpha^{\mathcal{P}}(\varphi)$ 为 $\mathcal{P}$ 中满足 $\varphi' \supseteq \varphi$ 的最小的抽象值。根据抽象域 $\mathcal{P}$ 定义中的两个条件，可以证明该函数是良定义的。

假设给定的元抽象域 $\mathcal{U}$ 中包含抽象值 $[-\infty, \infty], [1 + 8k, 8 + 8k], [1 + 4k, 4 + 4k], [1 + 2k, 2 + 2k], \{k\}$，其中 k 是任意正整数。可以验证 $\mathcal{U}$ 是一个合法的元抽象域。考虑抽象域 $\mathcal{P} = \{[-\infty, \infty], [1, 4]\}$，下面是几个关于函数 $\alpha^{\mathcal{P}}(\varphi)$ 的例子。

$$\alpha^{\mathcal{P}}(\{3\}) = \alpha^{\mathcal{P}}([1, 2]) = [1, 4] \qquad \alpha^{\mathcal{P}}([5, 8]) = \alpha^{\mathcal{P}}([1, 8]) = [-\infty, \infty]$$

对于 k 元操作符 f，从抽象值的 k 元组到抽象值的函数 $[\![f(\varphi_1, \ldots, \varphi_k)]\!]^{\#}$ 是 f 的一个抽象语义当且仅当它描述了 f 语义的一个超集，即满足如下条件。

$$\forall(\varphi_1, \ldots, \varphi_k) \in \mathcal{U}^k, \forall v_1 \in \varphi_1, \ldots, \forall v_k \in \varphi_k, f(v_1, \ldots, v_k) \in [\![f(\varphi_1, \ldots, \varphi_k)]\!]^{\#}$$

利用函数 $\alpha_{\mathcal{P}}$，f 在抽象域 $\mathcal{P}$ 上的抽象语义可以直接定义为 $\alpha^{\mathcal{P}}([\![f(\varphi_1, \ldots, \varphi_k)]\!]^{\#})$。

在本节的例子中，操作符 + 和 × 的抽象语义可以按照如下方式定义。

$$[\![+(\varphi_1, \varphi_2)]\!]^\# := \alpha^{\mathcal{U}}\Big(\{v_1 + v_2 \mid v_1 \in \varphi_1, v_2 \in \varphi_2\}\Big)$$
$$[\![\times(\varphi_1, \varphi_2)]\!]^\# := \alpha^{\mathcal{U}}\Big(\{v_1 \times v_2 \mid v_1 \in \varphi_1, v_2 \in \varphi_2\}\Big)$$

下面是几个关于上述抽象语义的具体例子。

$$[\![+([1,4],1)]\!]^\# = [1,8] \quad [\![+([1,4],[1,8])]\!]^\# = [-\infty,\infty]$$
$$[\![\times([1,4],[1,2])]\!]^\# = [1,8] \quad [\![\times([1,4],[0,1])]\!]^\# = [-\infty,\infty]$$

给定输入输出样例 $1 \mapsto 9$，G 中的一个合法程序为 $\times(x+2)\times 3$。为了合成这一程序，基于抽象精化的方法首先选取朴素的抽象域 $\mathcal{P}_0 = \{[-\infty,\infty]\}$，并在对应的抽象样例 $[-\infty,\infty] \mapsto [-\infty,\infty]$ 上构造变形代数空间如下。

$$\langle S,[-\infty,\infty]\rangle \to x \mid \langle S,[-\infty,\infty]\rangle + \langle C,[-\infty,\infty]\rangle \mid \langle S,[-\infty,\infty]\rangle \times \langle C,[-\infty,\infty]\rangle$$
$$\langle C,[-\infty,\infty]\rangle \to 2 \mid 3$$

该方法从这一空间中采样得到程序 x，作为这一轮生成的结果。

然而，根据在具体语义上的运算，x 在输入为 1 时的输入为 1，并不满足输入输出样例，因此基于抽象精化的构造方法会尝试精化 $\mathcal{P}_0$，即在 $\mathcal{P}_0$ 中加入更多的抽象值，从而将错误结果 x 排除。一般性地，给定从抽象域 $\mathcal{P}$ 生成的错误结果 p，该方法会用通过如下方式为 $\mathcal{P}$ 添加抽象值.

- 首先，该方法找到 $\mathcal{P}$ 中一个极大的抽象值 φ_o 满足 $p(I) \in \varphi_o$ 且 $O \notin \varphi_o$。如果 $\mathcal{P}$ 中不存在这样的抽象值，则从 $\mathcal{U}$ 中选取合适的抽象值加入 $\mathcal{P}$ 中。
- 接着，该方法递归地限制 p 的抽象语义的输出值为 φ_o。假设当前程序片段为 $f(p_1,\ldots,p_k)$，要求的抽象输出为 φ，该方法会从 $\mathcal{P}$ 中选取一个极大的抽象值列表 $\varphi_1,\ldots,\varphi_k$ 满足（1）$p_i(I) \in \varphi_i$ 且（2）$[\![f(\varphi_1,\ldots,\varphi_k)]\!]^\# \subseteq \varphi$，并递归地限制程序片段 p_i 的抽象输出符合 φ_i。如果 $\mathcal{P}$ 中不存在这样的抽象值列表，则从 $\mathcal{U}$ 中选取合适的抽象值加入 $\mathcal{P}$ 中。

对于初始抽象域 $\mathcal{P}_0 = \{[-\infty,\infty]\}$ 与错误程序 x，该方法会在第一步中引入抽象值 $[1,8]$，从而得到抽象域 $\mathcal{P}_1 = \{[-\infty,\infty],[1,8]\}$ 与下列变形代数空间。

$$\begin{aligned}\langle S,[-\infty,\infty]\rangle \to &\langle S,[-\infty,\infty]\rangle + \langle C,[1,8]\rangle \mid \langle S,[-\infty,\infty]\rangle \times \langle C,[1,8]\rangle \\ &\mid \langle S,[1,8]\rangle + \langle C,[1,8]\rangle \mid \langle S,[1,8]\rangle \times \langle C,[1,8]\rangle \\ \langle S,[1,8]\rangle \to &x \qquad \langle C,[1,8]\rangle \to 2 \mid 3\end{aligned}$$

该方法从这一空间中采样得到程序 $x+2$，作为这一轮的生成结果。根据精化过程，该

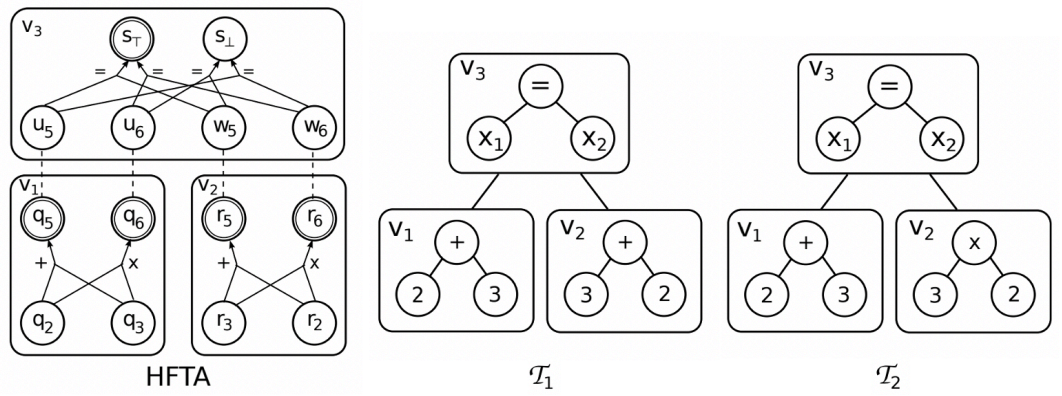

图 1.5　左图展示了对应公式 1.2 的层次有限树自动机图示，其中每个节点 f_v 表示对应函数 f 的输出为 v 的变形代数空间节点，$\top$ 和 $\bot$ 分别表示布尔值 true 和 false。中间与右侧两图展示了被该层次有限树自动机接受的两棵层次树，分别对应了解 $(q(x,y) = x + y, r(x,y) = x + y)$ 与 $(q(x,y) = x + y, r(x,y) = x \times y)$。

方法会在第二步中为了保证 $x+2$ 的抽象值为 $[1,8]$，会引入抽象值 $[1,4]$，并得到 $\mathcal{P}_2 = \{[-\infty,\infty],[1,8],[1,4]\}$。通过在 $\mathcal{P}_2$ 上构建变形代数空间，该方法能找到目标程序 $(x+2)\times 3$。

与朴素的两种变形代数空间的构造方法相比，基于抽象精化的构造方法同时利用了样例中的输入信息与输出信息：在自底向上构造抽象空间时用到了输入信息，在精化抽象域时用到了输出信息。实验表明，该方法的效率相比于朴素的构造方法有着 90 倍以上的效率提升。然而，因为基于抽象精化的方法依赖于抽象域与抽象语义的定义，因此它的通用性不如朴素的自底向上的构造方法。

1.4.4.6　对关系的语法制导合成问题的扩展

基于空间表示的方法并不能直接推广至关系的语法制导合成问题，因为第 1.4.4.1 中变形代数空间 $G[e]$ 的构造依赖于输入输出样例。王新宇等人于 2018 年提出了*层次有限树自动机* (hierarchical finite tree automata, HFTA)，从而将自底向上的构造方法拓展到了关系的语法制导合成问题上。

本文使用约束 $\exists f, \forall x, y \in \texttt{Int}, f(x,y) = f(y,x)$ 来展示该方法的合成过程。总体来说，该方法分为两个阶段。在第一阶段中，该方法考虑如下的弱化约束，其中原约束中对 f 的调用被认为是对两个不同的函数 q, r 进行的。

$$\exists q, r, \forall x, y \in \texttt{Int}, q(x,y) = r(y,x) \tag{1.2}$$

该方法将公式 1.2 分为三部分：$q(x,y), r(y,x), s(u,w)$，其中 $s(a,b) \coloneqq a = b$，u, v 分别对应 q 和 r 的输出。给定例子 $e = \langle x = 2, y = 3\rangle$，该方法会按照这三部分的层次为公式 1.2 构建类似变形代数空间的结构，成为层次有限树自动机，如图 1.5 所示。

1. 两个最底层的表达式为 $q(x,y)$ 和 $r(y,x)$，它们的输入 x, y 在例子 e 中已经给出，

因此可以用自底向上的变形代数空间方法为 q 和 r 构建变形代数空间 v_1, v_2。

2. 上层的表达式为 $s(u, v)$，其中 u 和 v 的可能值由变形代数空间 v_1 和 v_2 中的节点给出。因此，对于 $v_1(v_2)$ 中的每一个节点，为 $u(v)$ 构造一个具有对应值的节点，并继续通过自底向上的变形代数空间构造方法为 s 构建变形代数空间 v_3。

通过对该层次有限树自动机采样，可以得到一些满足样例 e 的程序对 (q, r)。图 1.5 中的层次树 $\mathfrak{T}_1$ 与 $\mathfrak{T}_2$ 展示了两个可能的采样结果。

因为公式 1.2 中，函数 q 和 r 实际上均对应原约束中的目标程序 f，所以在第二阶段中，该方法尝试从层次有限树自动机中得到一棵满足 $q = r$ 的层次树（例如图 1.5 中的树 $\mathfrak{T}_1$），从而得到一个满足原约束的程序 f。该方法使用搜索与回溯的方法来找到这样的一棵层次树，其中每一轮的过程如下。

- 选择原公式中的一个还未被赋值的目标程序 f。假设该程序在层次有限树自动机中对应变形代数空间 $v_1, \ldots, v_k$。
- 枚举被 $v_1, \ldots, v_k$ 共同包含的程序集合。对于其中的每一个程序 p，该方法构造一个新的层次有限树自动机，其中 $v_1, \ldots, v_k$ 只包含程序 p。
- 如果新的层次有限树自动机为空，则回溯。如果所有目标程序均被赋值，则找到一组合法解。否则该方法递归地未其他目标程序搜索赋值。

在本节的例子中，因为只有一个目标程序 f，所以该方法相当于在 v_1, v_2 的交集中遍历程序，并找到一个使得固定 f 后新的层次有限树自动机非空的程序。

基于层次有限树自动机的合成方法第二阶段的搜索过程与第 1.4.2.3 中基于枚举的求解关系的合成问题的方法存在一定的相似性，两者都在枚举所有目标程序的可能取值。相比之下，基于层次有限树自动机的方法存在两个优点。

- 第一阶中，该方法过滤掉了一部分即使在弱化约束上也不合法的取值。
- 第二阶段搜索的过程中，通过维护部分解确定下的层次有限树自动机，该方法可以有效地剪去一些已经确定不合法的搜索分支。

1.4.5　基于合一化的合成方法

Kroening 等人于 2015 年提出了基于合一化的程序合成框架 (synthesis through unification, STUN) [31]。给定程序合成问题，该框架首先合成一系列可能用于目标程序的表达式，再使用一些特殊的合一操作符将这些表达式合并成目标程序。常见的合一操作符包括分治操作符 ite、逻辑与 $\wedge$、逻辑或 $\vee$ 等。本文只对其中最复杂的操作符——分支操作符 ite 进行介绍。

1.4.5.1 基于合一化的分支程序合成框架

基于合一化的合成方法在合成分支程序时，要求文法 G 以 $S \to T \mid ite(B,T,T)$ 的形式给出，其中 S 是初始非终结符，非终结符 B 与 T 分别代表分支条件与分支语句。

基于合一化的合成器由语句合成器 $\mathfrak{T}$ 与合一器 $\mathfrak{U}$ 组成。给定输入输出样例集合 E，合成器 $(\mathfrak{T},\mathfrak{U})$ 通过如下两步合成分支程序。

- 语句合成器 $\mathfrak{T}$ 生成语句集合 T 覆盖所有样例，即 $\forall(I \mapsto O) \in E, \exists t \in T, t(I) = O$。
- 合一器 $\mathfrak{U}$ 为每一个 T 中的语句合成合适的条件，并将它们合一成一个满足所有样例的分支程序。

例 16 给定样例 $e_1 = \langle x = 0, y = 1\rangle \mapsto 1$ 和 $e_2 = \langle x = 2, y = 0\rangle \mapsto 2$，语句合成 $\mathfrak{T}$ 的一个可能结果为 $\{x, y\}$，其中样例 e_1 被 x 满足，样例 e_2 被 y 满足。给定该语句集合，$\mathfrak{U}$ 可能合成分支条件 $x \le y$，并返回程序 $ite(x \le y, y, x)$。

一般情况下，上述框架是不完备的。因为文法 G 中的条件可能不足以将语句合成器 $\mathfrak{T}$ 合成的语句集合合并成一个合法程序，上述框架可能无法解出一些存在合法解的合成问题。Kroening 证明了当文法 G 分支封闭时，上述框架是完备的。

定义 2 给定以 $S \to T \mid ite(B,T,T)$ 的形式给出的文法 G，G 是分支封闭的当且仅当可用的分支条件足以描述任意两个分支语句的相等条件，即满足如下公式。

$$\forall t_1, t_2 \in P_T, \exists c \in P_C, \forall I, t_1(I) = t_2(I) \iff c(I)$$

其中 P_B 与 P_T 分别表示非终结符 B 和 T 对应的条件集合与语句集合。

对于其他情况，可以通过回溯来将上述框架变得完备：每当合一器求解失败时，合成框架回溯至第一步并让语句合成器返回其他的合法语句集合。

1.4.5.2 基于合一化的分支表达式合成器

Alur 等人于 2017 年提出了 Eusolver，一个基于合一化的分支表达式合成器 [33]。该合成器使用的语句合成器 $\mathfrak{T}_E$ 与合一器 $\mathfrak{U}_E$ 如下所示。

- $\mathfrak{T}_E$ 从小到大枚举可用的分支语句。对于每一个语句，如果它满足的样例集合与已有的语句均不相同，则把它加入到语句集合中。如果语句集合中的所有程序已经覆盖了所有样例，则 $\mathfrak{T}_E$ 将它作为结果返回。
- $\mathfrak{U}_E$ 从小到大枚举可用的条件语句，并使用决策树学习算法 ID3 来将语句集合合一成满足所有样例的分支程序 [48]。

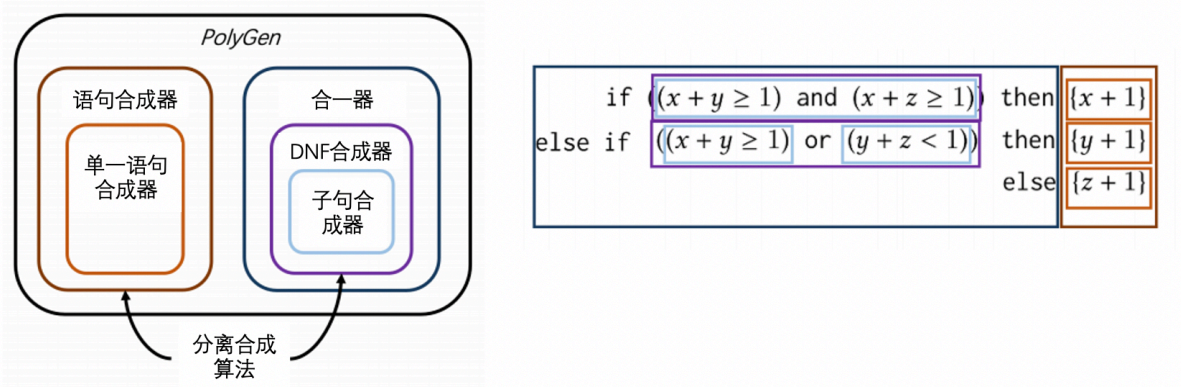

图 1.6　PolyGen 结构的图示，其中右图的每一个矩形表示了由左图对应颜色合成器合成的部分。

Eusolver 的语句合成器与合一器均非常高效，因此给定样例集合，它总是可以快速地合成满足所有样例的分支程序。然而在 CEGIS 框架下，它的效率却不理想：Eusolver 经常需要大量的反例才能合成一个满足逻辑规约的合法程序。直观上来说，因为 Eusolver 并未对结果程序的规模作出限制，$\mathcal{T}_E$ 可能会返回较大的语句集合且 ID3 可能会学到复杂的决策结构，所以 Eusolver 很容易过拟合到给定的样例集合上。

为了提高合成器的泛化能力，北京大学的熊英飞团队于 2021 年基于奥卡姆学习理论 [49] 提出了*奥卡姆合成器*的概念，并为分支程序设计了基于合一化的奥卡姆合成器 PolyGen [32]。简单来说，一个奥卡姆合成器保证合成结果的大小一定不超过最小合法程序大小的多项式倍。根据奥卡姆学习理论，给定关于最小合法程序大小多项式数量的输入输出用例，奥卡姆合成器便能保证合成结果的错误率是可控的。

图 1.6 展示了 PolyGen 的结构。PolyGen 的核心是分离合成算法，它进一步第将语句集合的合成问题拆分成单一语句的合成问题，将析取范式（disjunctive normal form, DNF）的合成问题拆分成单一子句的合成问题。为了限制合成结果的大小，分离合成算法迭代进行。以语句集合的合成问题为例，在每一轮中，分离合成算法限制使用的语句数量与每个语句的大小，并用搜索的方法寻找满足限制的语句集合。如果这样的语句集合不存在，则分离合成算法会放宽语句数量与每个语句大小的限制，并进入下一轮的迭代。实验表明，在 CEGIS 框架下，PolyGen 表现显著地优于 Eusolver。Eusolver 合成的耗时平均是 PolyGen 的 6.14 倍，使用的反例数量平均是 PolyGen 的 2.33 倍。

1.4.6　与概率模型相结合的合成方法

近期，随着机器学习技术的发展，有一些工作尝试使用概率模型来加速已有的程序合成技术 [34–36, 50–52]。在本节中，本文将介绍其中的一些典型工作。

1.4.6.1 概率模型引导的枚举方法

Menon 等人于 2013 年提出了使用概率上下文无关文法加速基于枚举的合成方法 [51]。概率上下文无关文法 (probabilistic context-free grammar, PCFG) [53] 是一种经典的概率模型，它可以被视为由一个上下文无关文法 G 与概率函数 γ 组成的二元组，其中 γ 为 G 中的每一条展开式赋予了一个概率值，表示其被选用的概率。为了保证概率是良定义的，γ 需要保证对 G 中的每一个非终结符 S，从 S 出发的所有展开式的概率和为 1。给定概率上下文无关文法 $\mathcal{G} = \langle G, \gamma \rangle$，对于 G 中任意一个程序 p，假设 p 由展开式 $r_1, \ldots, r_n$ 得到，则 p 在模型 $\mathcal{G}$ 下的概率为 $\prod_{i=1}^{n} \gamma(r_i)$。

例 17 *对应例 1 中上下文无关文法的一个概率上下文无关文法如下所示，其中每条规则右侧括号内的数值表示了该规则被赋予的概率。*

$$I \to x(0.5) \mid y(0.4) \mid 1(0) \mid I + I(0) \mid ite(B, I, I)(0.1) \quad B \to I \le I(1)$$

考虑程序 $ite(x \le y, y, x)$。要展开得到该程序，需要使用规则 $I \to x$ 与 $I \to y$ 各两次，规则 $I \to ite(B, I, I)$ 与 $B \to I \le I$ 各一次。因此，该程序在这一上下文无关文法中的概率为 $0.5^2 \times 0.4^2 \times 0.1 \times 1 = 0.004$

概率上下文无关文法可以方便地与自顶向下的基于枚举的合成方法相结合（第 1.4.2.1）。在不引入概率模型时，自顶向下的枚举方法使用队列维护所有可能的展开结果，每次扩展队首的展开结果，从而便保证按照大小从小到大的顺序枚举所有程序。而在引入概率模型后，只需要使用优先队列维护所有的展开结果，每次扩展概率最大的展开结果，即可保证按照概率从小到大的顺序枚举所有的程序。

例 18 *给定例 17 中的概率上下文无关文法，自顶向下的枚举方法在该模型的引导下的枚举过程如下所示，其中每个展开结果右侧括号内的数值表示了该结果的概率。*

$$S(1) \quad x(0.5) \quad y(0.4) \quad ite(B, I, I)(0.1) \quad ite(I \le I, I, I)(0.1) \quad ite(x \le I, I, I)(0.05)$$
$$ite(y \le I, I, I)(0.04) \quad ite(x \le x, I, I)(0.025) \quad ite(x \le y, I, I)(0.02) \quad ite(y \le x, I, I)(0.02)$$
$$ite(y \le y, I, I)(0.016) \quad \ldots \quad ite(x \le x, x, x)(0.00625) \quad \ldots$$

Lee 等人于 2018 年提出了求解器 Euphony [35]，在以上朴素的概率模型引导枚举算法的基础上，作出了两点优化。首先，Euphony 使用了 A* 算法来枚举展开结果。对于每一个非终结符，定义其启发值为它对应的程序中概率的最大值。对于每一个扩展结果，定义其启发值为其概率乘以所有用到的非终结符的启发值的乘积。可以证明每一个扩展结果的启发值一定是它能扩展得到的程序的概率的上界。因此，Euphony 在

```
TCOND ::= ϵ | WriteOp TCOND| MoveOp TCOND
WriteOp ::= WriteValue | WritePos | WriteType
MoveOp ::= Up | Left | Right | DownFirst | DownLast |
        NextDFS | NextLeaf | PrevDFS | PrevLeaf |
        PrevNodeType | PrevNodeValue | PrevNodeContext
```

图 1.7　用于描述上下文函数的领域特定语言。

使用优先队列维护展开结果时，每次扩展启发值最大的展开结果。这样在仍然保证按照概率从小到大枚举的同时，还及大地减少了枚举到的包含非终结符的展开结果数量。

例 19 在例 *17* 中的概率上下文无关文法中，非终结符 I 与 B 的启发值分别为 0.5 与 0.25。此时，A^* 算法的枚举过程如下所示，其中每个展开结果右侧括号内的数值代表了改结果的启发值。

$$S(0.5) \quad x(0.5) \quad y(0.4) \quad ite(B, I, I)(0.00625) \quad ite(I \le I, I, I)(0.00625)$$
$$ite(x \le I, I, I)(0.00625) \quad ite(x \le x, I, I)(0.00625) \quad ite(x \le x, x, I)(0.00625)$$
$$ite(x \le x, x, x)(0.00625) \quad \ldots$$

与例 *18* 中的枚举顺序相比，A^* 算法极大地提前了枚举序列中第三个完整程序 $ite(x \le x, x, x)$ 的出现位置。

其次，Euphony 使用了概率高阶文法而不是概率上下文无关文法来知道枚举过程。概率高阶文法 (probabilistic higher-order grammar, PHOG) [54] 是一个由上下文无关文法 G、上下文函数 α 与概率函数 γ 组成的三元组。

- 给定包含非终结符的展开结果 p，$\alpha(p)$ 表示 p 中最左非终结符的上下文。
- 给定上下文 c 与展开式 r，$\gamma(c, r)$ 表示了展开式 r 在上下文为 c 时被选用的概率。

类似上下文无关文法，为了保证概率是良定义的，γ 需要对于任意上下文与任意 G 中的非终结符，从该非终结符所有展开式的概率和为 1。给定概率高阶文法 $\mathcal{H} = \langle G, \alpha, \gamma \rangle$，对于 G 中任意一个程序 p，假设其最左展开使用的展开式为 $r_1, \ldots, r_n$，且对应的上下文为 $c_1, \ldots, c_n$，则 p 在模型 $\mathcal{H}$ 下的概率为 $\prod_{i=1}^{n} \gamma(c_i, r_i)$。

在概率高阶文法中，上下文函数 α 定义为如图 1.7 所示的领域特定语言中的一个程序。在该语言中，一个程序为一系列移动操作 (MoveOp) 与写操作 (WriteOp) 的序列。从最左的非终结符出发，移动操作在展开结果的抽象语法树上移动，而每个写操作则将当前节点的一些信息写入到上下文中。

例 20 考虑 `Up WriteValue DownFirst WriteType DownLast WritePos` 在展开结果 $ite(x \le x, x, I)$ 上的输出。从节点 I 出发，该程序上移到对应 ite 的节点，写入其取值 ite，下移到第一个子节点 $\le$，写入其类型 $Bool$，下移到最后一个子节点 x，写入其相对父节点的位置 2。因此，该程序输出的上下文为 $ite\ Bool\ 2$。

概率高阶文法的表达能力严格强于概率上下文无关文法，因为后者可以被视为前者在上下文函数为常函数时的特例。因此，概率高阶文法通常能更精确地描述程序分布，从而更准确地引导枚举过程。实验结果表明，Euphony 在使用概率高阶文法时比使用概率上下文无关文法时多解出了 77.4% (103 个) 合成任务。

除了引入更复杂的模型，选择适当的程序展开顺序同样可以提升概率模型引导枚举的效率。其原因在于对于不同的展开方式，概率模型的预测精度可能不同：概率模型的预测精度越高，则使用对应展开方式的枚举合成方法的效率也就越高。

例 21 考虑展开结果 $f(E, E)$ 与如下四个对应的程序，其中括号内的数值表示其概率。

$$f(x,x)(0.25) \quad f(x,y)(0.3) \quad f(y,x)(0.4) \quad f(y,y)(0.05)$$

考虑如下两种展开策略枚举到最可能的程序 $f(y,x)$ 的耗时。

- 如果优先展开左侧的 E，则枚举过程会先尝试分支 $f(x,E)(0.55)$，然后再尝试对应目标程序的分支 $f(y,E)(0.45)$。
- 如果优先展开右侧的 E，则枚举过程会直接尝试进入目标分支 $f(E,x)(0.65)$。

更进一步地，在一些问题中，高效的展开方式甚至可能不是自顶向下的。

例 22 考虑程序 $hours(> 12)$，如果自顶向下的展开方法，因为 (> 12) 并不是一个常用的操作符，所以枚举过程可能会先尝试其他的操作符，例如算术运算符。但是，如果尝试自底向上的展开方法，在已知子表达式 $hours$ 的情况下，(> 12) 作为一个时间比较操作，它在概率模型中的优先级可能就被大幅提升了。

受到上述两个例子的启发，北京大学的熊英飞团队于 21 年提出了扩展规则的概念 [52]。在上下文无关文法的展开式的基础上，扩展规则显式地指定了展开顺序。具体地，每个扩展规则由展开式 r 与位置符 p 组成，表示在已知展开式 r 中第 $(p+1)$ 个非终结符的情况下，可以扩展出展开式右侧的结果。因此当 $p = 0$ 时，扩展规则定义了一次自顶向下的展开；而当 $p > 0$ 时，它定义了一次自底向上的展开。特殊地，$p = \perp$ 表示初始规则，它定义了枚举过程的起点。

例 23 如下展示了对应例 22 中程序 $hours > 12$ 的一组扩展规则集合。

$$\langle E \to hours, \perp \rangle \quad \langle E \to E > 12, 1 \rangle$$

在应用第一条规则后，得到枚举起点 *hour*。接着，在应用第二条规则后，*hour* 被填入 $E > 12$ 中的非终结符 E，得到目标程序 $hour + E$。

实验结果表明，当使用合适的扩展规则时，概率模型引导下的枚举方法的合成效率可以提高一倍以上。

1.4.6.2　概率模型引导的合一化方法

概率模型引导的枚举方法可以直接与基于合一化的求解器 Eusolver 相结合。回顾在第 1.4.5.2 中介绍的求解器 Eusolver 的求解过程：

- Eusolver 从小到大枚举所有分支语句直到已有的语句集合足以覆盖所有的程序。
- Eusolver 从小到大枚举所有分支条件直到已有的条件足以学习将语句集合合一成一个合法程序。

Lee 等人通过将 Eusolver 中的两个枚举过程替换成 Euphony，从而将 Euphony 扩展为一个基于合一化的求解器 [35]。实验结果表明，Euphony 比 Eusolver 多完成了接近两倍的合成任务，并且在共同解出的合成任务中，达到了两倍以上的求解效率提升。

1.4.6.3　概率模型引导的空间表示方法

通过与基于迭代加深的 A* 算法（即 IDA* [55]）结合，概率模型可以被用于引导自顶向下的基于空间表示的合成方法的求解过程 [36]。该方法维护了一个概率的下界 t，并在构建变形代数空间 $G[e]$ 时，只要求 $G[e]$ 包含那些概率大于等于 t 程序并允许其忽略其他概率较小的程序。如果在限制 t 下不存在满足所有样例的程序（即构建的所有变形代数空间 $G[e]$ 的交为空），则该方法会减小下界 t 的值，并开始下一轮的合成。

并不是所有概率模型都可以用于引导基于空间表示的合成方法。回顾第 1.4.4.3 节，自顶向下的构造方法是一个递归的过程，其中每一次递归调用中给定的信息只有原变形代数空间中的一个节点 N 与一个期望的输出值 v。因此，若要使用概率模型引导该方法的构造过程，必须要保证每一个展开式的选取概率可以从节点 N 与 v 计算得到。根据这个限制，下面讨论了第 1.4.6.1 中涉及的两个概率模型，概率上下文无关文法与概率高阶文法，对于基于空间表示的合成方法的适用性。

- 在概率上下文无关文法中，每条展开式被赋予了一个固定的概率值，因此它可以被用于引导基于空间表示的合成方法。
- 在概率高阶文法中，每条展开式的概率值域一个从当前展开结果中提取得到的上下文有关。因此该模型用到了除 $\langle N, v\rangle$ 以外的信息，一般情况下无法被用于引导基于空间表示的合成方法。

因为概率上下文无关文法的精度有限，所以为了进一步地提升合成效率，已有工作集中在设计与基于空间表示的合成方法相兼容且具有更高精度的概率模型 [34, 36]。

Kalyan 等人于 2018 年探索了合成字符串程序时的情况 [34]。在该场景下构造变形代数空间时，期望的输出值 v 是一个字符串。相比于其他的基本类型例如整数、布尔值等，字符串中包含了大量信息，因此可以从 v 的内容预测需要使用的展开式。

例 24 *考虑例 14 中的文法与输入输出样例* $(FS = \texttt{John}, LS = \texttt{Jonathan}) \mapsto \texttt{J.Jonathan}$。

- *当* $v = \texttt{J.Jonathan}$ *时，使用的展开式只可能为连接操作* $+$。
- *当* $v = \texttt{J}$ *时，使用的展开式只可能为访问操作 CharAt*。

基于以上观察，Kalyan 等人用长短期记忆网络 (long short-term memory, LSTM) [56] 构建了概率模型。给定输入输出样例 $I \mapsto O$，程序 p 的概率为 $\prod_{s \in sub(p)} \gamma(I, s(I), top(s))$，其中 $sub(p)$ 返回 p 的所有子程序形成的集合，$top(s)$ 表示程序 s 使用的第一条展开式，γ 由长短期神经网络定义，$\gamma(I, O, r)$ 表示输入为 I 输出为 O 时，第一条展开式是 r 的概率。该模型可以用于引导自顶向下的变形代数空间构造方法。给定样例 $I \mapsto O$，规则 r 用于展开节点 $\langle N, v\rangle$ 的概率为 $\gamma(I, v, r)$。实验表明，该方法求解效率相比于朴素的自顶向下构造方法有着 93% 的平均提升。

尽管将神经网络引入概率模型提升了概率模型的精度，但是它让对概率模型求值的开销成为了新的瓶颈。为了在概率模型精度与模型求值效率之间寻求新的平衡，北京大学的熊英飞团队于 2020 年定义了一种轻量级的、可用于引导空间表示方法的概率模型，名为自顶向下预测模型 (topdown prediction model) [36]。

自顶向下预测模型是概率高阶文法的一个特例。在自顶向下预测模型中，上下文函数 α 又转移函数 τ 与初始上下文 c_0 定义。对于任意展开结果中的任意位置，如果该位置是根节点，则其上下文为 c_0。否则假设其父节点的上下文为 c，父节点使用的展开式为 r，当前节点是父节点的第 k 个孩子，则当前节点的上下文为 $\tau(c, r, k)$。

例 25 *抽象语法树上的 n-gram 模型是自顶向下预测模型的一个特例。给定参数 n，n-gram 模型中一个语法树节点的上下文为它的 n 个祖先的展开式信息与位置信息。以 2-gram 模型为例，它可以通过如下方式被定义成一个自顶向下预测模型。*

$$c_0 \coloneqq ((\bot, \bot), (\bot, \bot)) \quad \tau(((r_1, p_1), (r_2, p_2)), r, p) \coloneqq ((r, p), (r_1, p_1))$$

为了在自顶向下构造变形代数空间的过程中对自顶向下预测模型求值，需要在每一个变形代数空间节点中显式地引入上下文。给定自顶向下预测模型 $\mathfrak{T} = \langle \tau, c_0, \gamma\rangle$，其中 γ 表示预测函数，初始变形代数空间 G 与输入输出样例 $e = I \mapsto O$，变形代数空间 $G[e, \mathfrak{T}]$ 中的每个节点形如 $\langle S, v, c\rangle$，其中节点 S 和输出值 v 的含义与 $G[e]$ 相同，c 表示该节点的上下文，通过如下方式构造：

- $G[e, \mathfrak{T}]$ 的初始节点的上下文为 c_0。

- 对于上下文为 c 的节点 $f(N_1, \ldots, N_k)$，N_i 的上下文为 $\tau(c, f, i)$。
- 对于上下文为 c 的节点 $\mathbf{U}(N_1, \ldots, N_k)$，$N_i$ 的上下文仍然为 c。

在构造 $F[e, \mathcal{T}]$ 时，概率函数 γ 可以使用记录在节点中的上下文进行求值：使用展开式 r 展开节点 $\langle S, v, c\rangle$ 的概率为 $\gamma(c, r)$。

在使用自顶向下预测模型时，需要权衡引入上下文带来的精度收益与额外开销。一方面，可能的上下文数量越多，预测模型的精度就会越高，引导作用也就会越明显。但是另一方面，变形代数空间 $G[e, \mathcal{T}]$ 中的节点数量也会越多。实验结果表明，对于较为困难的合成任务，使用 1-gram 模型的求解效率相比于概率上下文无关文法有着 5 – 10 倍的提升，而使用 2-gram 模型的求解效率反而比 1-gram 模型慢 2 – 3 倍。

参考文献

参考文献

[1] Amir Pnueli and Roni Rosner. "*On the Synthesis of a Reactive Module*". In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, **1989**: 179–190. https://doi.org/10.1145/75277.75293.

[2] Alonzo Church. "*Application of recursive arithmetic to the problem of circuit synthesis*". *Journal of Symbolic Logic*, **1963**, *28*(4).

[3] Nadia Polikarpova, Ivan Kuraj and Armando Solar-Lezama; ed. by Chandra Krintz and Emery D. Berger. "*Program synthesis from polymorphic refinement types*". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. ACM, **2016**: 522–538. https://doi.org/10.1145/2908080.2908093.

[4] Azadeh Farzan and Victor Nicolet; ed. by Stephen N. Freund and Eran Yahav. "*Phased synthesis of divide and conquer programs*". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. ACM, **2021**: 974–986. https://doi.org/10.1145/3453483.3454089.

[5] Ruyi Ji, Tianran Zhu, Yingfei Xiong *et al.* "*Synthesizing Efficient Dynamic Programming Algorithms*". *CoRR*, **2022**, *abs/2202.12208*. https://arxiv.org/abs/2202.12208.

[6] Ruyi Ji, Yingfei Xiong and Zhenjiang Hu. "*Black-Box Algorithm Synthesis - Divide-and-Conquer and More*". *CoRR*, **2022**, *abs/2202.12193*. https://arxiv.org/abs/2202.12193.

[7] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia *et al.*; ed. by Jeff Kramer, Judith Bishop, Premkumar T. Devanbu *et al.* "*Oracle-guided component-based program synthesis*". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, **2010**: 215–224. https://doi.org/10.1145/1806799.1806833.

[8] Tim Blazytko, Moritz Contag, Cornelius Aschermann *et al.*; ed. by Engin Kirda and Thomas Ristenpart. "*Syntia: Synthesizing the Semantics of Obfuscated Code*". In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, **2017**: 643–659. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko.

[9] Robin David, Luigi Coniglio and Mariano Ceccato. "*Qsynth-a program synthesis based approach for binary code deobfuscation*". In: *BAR 2020 Workshop*, **2020**.

[10] William R. Harris and Sumit Gulwani; ed. by Mary W. Hall and David A. Padua. "*Spreadsheet table transformations from examples*". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, **2011**: 317–328. https://doi.org/10.1145/1993498.1993536.

[11] Chenglong Wang, Alvin Cheung and Rastislav Bodk; ed. by Albert Cohen and Martin T. Vechev. "*Synthesizing highly expressive SQL queries from input-output examples*". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, **2017**: 452–466. https://doi.org/10.1145/3062341.3062365.

[12] Rajeev Alur, Rastislav Bodk, Garvit Juniwal *et al.* "*Syntax-guided synthesis*". In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, **2013**: 1–8. https://ieeexplore.ieee.org/document/6679385/.

[13] Zohar Manna and Richard J. Waldinger. "*Fundamentals of Deductive Program Synthesis*". *IEEE Trans. Software Eng.* **1992**, *18*(8): 674–704. https://doi.org/10.1109/32.153379.

[14] Sumit Gulwani; ed. by Orna Grumberg, Helmut Seidl and Maximilian Irlbeck. "*Program Synthesis*". In: *Software Systems Safety*. IOS Press, **2014**: 43–75. https://doi.org/10.3233/978-1-61499-385-8-43.

[15] Azadeh Farzan and Victor Nicolet; ed. by Kathryn S. McKinley and Kathleen Fisher. "*Modular divide-and-conquer parallelization of nested loops*". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, **2019**: 610–624. https://doi.org/10.1145/3314221.3314612.

[16] Azadeh Farzan and Victor Nicolet; ed. by Albert Cohen and Martin T. Vechev. "*Synthesis of divide and conquer parallelism for loops*". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, **2017**: 540–555. https://doi.org/10.1145/3062341.3062355.

[17] Nadia Polikarpova and Ilya Sergey. "*Structuring the synthesis of heap-manipulating programs*". *Proc. ACM Program. Lang.* **2019**, *3*(POPL): 72:1–72:30. https://doi.org/10.1145/3290385.

[18] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova *et al.*; ed. by Stephen N. Freund and Eran Yahav. "*Cyclic program synthesis*". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. ACM, **2021**: 944–959. https://doi.org/10.1145/3453483.3454087.

[19] Markus Püschel, José M. F. Moura, Jeremy R. Johnson *et al.* "*SPIRAL: Code Generation for DSP Transforms*". *Proc. IEEE*, **2005**, *93*(2): 232–275. https://doi.org/10.1109/JPROC.2004.840306.

[20] Martin T. Vechev and Eran Yahav; ed. by Rajiv Gupta and Saman P. Amarasinghe. "*Deriving linearizable fine-grained concurrent objects*". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. ACM, **2008**: 125–135. https://doi.org/10.1145/1375581.1375598.

[21] Yasunari Watanabe, Kiran Gopinathan, George Prlea *et al.* "*Certifying the synthesis of heap-manipulating programs*". *Proc. ACM Program. Lang.* **2021**, *5*(ICFP): 1–29. https://doi.org/10.1145/3473589.

[22] Nadia Polikarpova; ed. by Liron Cohen and Cezary Kaliszyk. "*Synthesis of Safe Pointer-Manipulating Programs (Invited Talk)*". In: *12th International Conference on Interactive Theorem Proving, ITP*

2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, **2021**: 2:1–2:1. https://doi.org/10.4230/LIPIcs.ITP.2021.2.

[23] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova *et al.*; ed. by Alexandra Silva and K. Rustan M. Leino. "*Deductive Synthesis of Programs with Pointers: Techniques, Challenges, Opportunities - (Invited Paper)*". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Springer, **2021**: 110–134. https://doi.org/10.1007/978-3-030-81685-8%5C_5.

[24] John C. Reynolds. "*Separation Logic: A Logic for Shared Mutable Data Structures*". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, **2002**: 55–74. https://doi.org/10.1109/LICS.2002.1029817.

[25] Susmit Jha and Sanjit A. Seshia. "*A Theory of Formal Synthesis via Inductive Learning*". *CoRR*, **2015**, *abs/1505.03953*. http://arxiv.org/abs/1505.03953.

[26] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh *et al.*; ed. by Hans-Juergen Boehm and Cormac Flanagan. "*TRANSIT: specifying protocols with concolic snippets*". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, **2013**: 287–296. https://doi.org/10.1145/2491956.2462174.

[27] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti *et al.*; ed. by Gary T. Leavens, Alessandro Garcia and Corina S. Pasareanu. "*Symbolic execution with existential second-order constraints*". In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, **2018**: 389–399. https://doi.org/10.1145/3236024.3236049.

[28] Oleksandr Polozov and Sumit Gulwani; ed. by Jonathan Aldrich and Patrick Eugster. "*FlashMeta: a framework for inductive program synthesis*". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. ACM, **2015**: 107–126. https://doi.org/10.1145/2814270.2814310.

[29] Xinyu Wang, Isil Dillig and Rishabh Singh. "*Program synthesis using abstraction refinement*". *Proc. ACM Program. Lang.* **2018**, *2*(POPL): 63:1–63:30. https://doi.org/10.1145/3158151.

[30] Xinyu Wang, Isil Dillig and Rishabh Singh. "*Synthesis of data completion scripts using finite tree automata*". *Proc. ACM Program. Lang.* **2017**, *1*(OOPSLA): 62:1–62:26. https://doi.org/10.1145/3133886.

[31] Rajeev Alur, Pavol Cerný and Arjun Radhakrishna; ed. by Daniel Kroening and Corina S. Pasareanu. "*Synthesis Through Unification*". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. Springer, **2015**: 163–179. https://doi.org/10.1007/978-3-319-21668-3%5C_10.

[32] Ruyi Ji, Jingtao Xia, Yingfei Xiong *et al.* "*Generalizable synthesis through unification*". *Proc. ACM Program. Lang.* **2021**, *5*(OOPSLA): 1–28. https://doi.org/10.1145/3485544.

[33] Rajeev Alur, Arjun Radhakrishna and Abhishek Udupa; ed. by Axel Legay and Tiziana Margaria. "*Scaling Enumerative Program Synthesis via Divide and Conquer*". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, **2017**: 319–336. https://doi.org/10.1007/978-3-662-54577-5%5C_18.

[34] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov *et al.* "*Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples*". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, **2018**. https://openreview.net/forum?id=rywDjg-RW.

[35] Woosuk Lee, Kihong Heo, Rajeev Alur *et al.*; ed. by Jeffrey S. Foster and Dan Grossman. "*Accelerating search-based program synthesis using learned probabilistic models*". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, **2018**: 436–449. https://doi.org/10.1145/3192366.3192410.

[36] Ruyi Ji, Yican Sun, Yingfei Xiong *et al.* "*Guiding dynamic programing via structural probability for accelerating programming by example*". *Proc. ACM Program. Lang.* **2020**, *4*(OOPSLA): 224:1–224:29. https://doi.org/10.1145/3428292.

[37] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodk *et al.*; ed. by John Paul Shen and Margaret Martonosi. "*Combinatorial sketching for finite programs*". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. ACM, **2006**: 404–415. https://doi.org/10.1145/1168857.1168907.

[38] Vu Le, Daniel Perelman, Oleksandr Polozov *et al.* "*Interactive Program Synthesis*". *CoRR*, **2017**, *abs/1703.03539*. http://arxiv.org/abs/1703.03539.

[39] Ruyi Ji, Jingjing Liang, Yingfei Xiong *et al.*; ed. by Alastair F. Donaldson and Emina Torlak. "*Question selection for interactive program synthesis*". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, **2020**: 1143–1158. https://doi.org/10.1145/3385412.3386025.

[40] Mikaël Mayer, Gustavo Soares, Maxim Grechkin *et al.*; ed. by Celine Latulipe, Bjoern Hartmann and Tovi Grossman. "*User Interaction Models for Disambiguation in Programming by Example*". In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*. ACM, **2015**: 291–301. https://doi.org/10.1145/2807442.2807459.

[41] Osbert Bastani, Xin Zhang and Armando Solar-Lezama. "*Synthesizing Queries via Interactive Sketching*". *CoRR*, **2019**, *abs/1912.12659*. http://arxiv.org/abs/1912.12659.

[42] Daniel Ramos, Jorge Pereira, Inês Lynce *et al.* "*UNCHARTIT: An Interactive Framework for Program Recovery from Charts*". In: *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, **2020**: 175–186. https://doi.org/10.1145/3324884.3416613.

[43] Yuepeng Wang, Xinyu Wang and Isil Dillig. "*Relational program synthesis*". *Proc. ACM Program. Lang.* **2018**, *2*(OOPSLA): 155:1–155:27. https://doi.org/10.1145/3276525.

[44] Ignasi Abo, Robert Nieuwenhuis, Albert Oliveras *et al.*; ed. by Christian Schulte. "*A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints*". In: *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*. Springer, **2013**: 80–96. https://doi.org/10.1007/978-3-642-40627-0%5C_9.

[45] Tom M. Mitchell. "*Generalization as Search*". *Artif. Intell.* **1982**, *18*(2): 203–226. https://doi.org/10.1016/0004-3702(82)90040-6.

[46] Daniel W. Barowy, Sumit Gulwani, Ted Hart *et al.*; ed. by David Grove and Stephen M. Blackburn. "*FlashRelate: extracting relational data from semi-structured spreadsheets using examples*". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, **2015**: 218–228. https://doi.org/10.1145/2737924.2737952.

[47] Xinyu Wang, Greg Anderson, Isil Dillig *et al.*; ed. by Hana Chockler and Georg Weissenbacher. "*Learning Abstractions for Program Synthesis*". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Springer, **2018**: 407–426. https://doi.org/10.1007/978-3-319-96145-3%5C_22.

[48] J. Ross Quinlan. "*Induction of Decision Trees*". *Mach. Learn.* **1986**, *1*(1): 81–106. https://doi.org/10.1023/A:1022643204877.

[49] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler *et al.* "*Occam's Razor*". *Inf. Process. Lett.* **1987**, *24*(6): 377–380. https://doi.org/10.1016/0020-0190(87)90114-1.

[50] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt *et al.* "*DeepCoder: Learning to Write Programs*". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, **2017**. https://openreview.net/forum?id=ByldLrqlx.

[51] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani *et al.* "*A Machine Learning Framework for Programming by Example*". In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. JMLR.org, **2013**: 187–195. http://proceedings.mlr.press/v28/menon13.html.

[52] Yingfei Xiong and Bo Wang. "*L2S: A framework for synthesizing the most probable program under a specification*". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **2022**, *31*(3): 1–45.

[53] Alexander M. Rush and Michael Collins. "*A Tutorial on Dual Decomposition and Lagrangian Relaxation for Inference in Natural Language Processing*". *J. Artif. Intell. Res.* **2012**, *45*: 305–362. https://doi.org/10.1613/jair.3680.

[54] Pavol Bielik, Veselin Raychev and Martin T. Vechev; ed. by Maria-Florina Balcan and Kilian Q. Weinberger. "*PHOG: Probabilistic Model for Code*". In: *Proceedings of the 33nd International*

Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. JMLR.org, **2016**: 2933–2942. http://proceedings.mlr.press/v48/bielik16.html.

[55] Richard E. Korf. "*Depth-First Iterative-Deepening: An Optimal Admissible Tree Search*". *Artif. Intell.* **1985**, *27*(1): 97–109. https://doi.org/10.1016/0004-3702(85)90084-0.

[56] Sepp Hochreiter and Jürgen Schmidhuber. "*Long short-term memory*". *Neural computation*, **1997**, *9*(8): 1735–1780.

第二章　程序演算技术

2.1　引言

程序演算 (program calculation) 技术由程序变换 (program transformation) 系统发展而来。程序变换系统以一个程序为输入，不断地应用变换规则从当前程序产生新的程序，直到得到一个满足要求的（例如更高效的）程序 [1]。一般情况下，程序变换系统中的每一条规则都保证产生的程序与原程序语义等价，从而保证了变换结果一定是正确的：程序变换的过程直接构成了对结果程序正确性的证明。经过几十年的发展，研究人员提出了大量依托于不同语言的程序变换系统，例如函数式语言 [2–4]、逻辑语言 [5–8] 和约束语言 [9–12] 等。

为了处理复杂的程序，程序变换系统一般遵循展开/折叠 (unfold/fold) 框架 [3]。简单来说，该框架首先将给定的程序展开，得到一个规模更大但是更加底层的程序。该框架在展开的程序上通过问题特定的规则进行变换与化简，并最终将程序折叠得到变换结果。例 26 展示了一个使用展开/折叠框架变换函数式程序的例子。

例 26 考虑按照如下方式定义的递归函数 $dot(x, y, n)$，它接受两个长度为 n 的向量 x, y，并计算它们的内积 $\sum_{i=1}^{n} x_i y_i$。

$$dot(x, y, n) \coloneqq \mathit{if}\ n = 0\ \mathit{then}\ 0\ \mathit{else}\ dot(x, y, n-1) + x_n y_n\ \mathit{fi}$$

假设现在需要推导一个递归函数 $f(a, b, c, d, n)$，它接受四个长度为 n 的向量，并计算前两个向量的内积与后两个向量的内积的和，即 $\sum_{i=1}^{n} a_i b_i + \sum_{i=1}^{n} c_i d_i$。下面展示了一个遵循展开折叠框架的推导过程。

$$\begin{aligned}
f(a,b,c,d,n) &= dot(a,b,n) + dot(c,d,n) \\
&= \mathit{if}\ n = 0\ \mathit{then}\ 0\ \mathit{else}\ dot(a,b,n-1) + a_n b_n\ \mathit{fi} + \\
&\quad \mathit{if}\ n = 0\ \mathit{then}\ 0\ \mathit{else}\ dot(c,d,n-1) + c_n d_n\ \mathit{fi} \\
&= \mathit{if}\ n = 0\ \mathit{then}\ 0\ \mathit{else}\ dot(a,b,n-1) + dot(c,d,n-1) + a_n b_n + c_n d_n\ \mathit{fi} \\
&= \mathit{if}\ n = 0\ \mathit{then}\ 0\ \mathit{else}\ f(a,b,c,d,n-1) + a_i b_i + c_i d_i\ \mathit{fi}
\end{aligned}$$

在第一步变换中，变换系统展开了函数 dot 的定义。接着，该系统利用 *if-then-else* 操作的性质，将两个分支合并了起来。最后，该系统根据最开始的定义，将表达式 $dot(a, b, n-1) + dot(c, d, n-1)$ 给折叠回了 $f(a, b, c, d, n-1)$，从而得到了 f 的递归定义。

尽管展开/折叠框架有着很强的表达能力，但是在实践中，其核心缺点在于无法确

定展开的层数。一方面，如果展开的层数过少，则有可能不足以让后续变换过程得到一个想要的程序。另一方面，如果展开的层数过多，则展开后的程序会变得过于复杂，从而导致后续变换的搜索空间过大。

为了解决这一问题，程序演算对不进行展开/折叠的程序变换过程进行了研究。程序演算关注一系列通用的组件（通常是高阶函数），并研究它们之间的一系列转换规则。在使用程序演算的框架进行程序变换时，用户需要使用这些给定的组件定义初始程序。接着，变换系统会使用对应的转换规则在组件的层面对程序进行变换，直到得到满足要求的程序。

例 27 考虑一个与例 26 中函数 f 类似的程序 g，它接受一个以四元组为元素的列表 x 为输入，返回所有四元组前两个元素乘积的和加上后两个元素成绩的和，即 $\sum_{i=1}^{n} x_{i,1}x_{i,2} + \sum_{i=1}^{n} x_{i,3}x_{i,4}$。在程序演算框架中，该程序可以使用如下的组件 *fold* 定义。

$$fold(\oplus, e, [x_1, \ldots, x_n]) \coloneqq (((e \oplus x_1) \oplus x_2) \cdots \oplus x_n)$$

fold 接受二元操作符 $\oplus$、初值 e 和一个列表为输入。它依次访问列表中的每一个元素，并使用当前操作符 $\oplus$ 与 e 去更新当前结果。使用这一组件，函数 g 的定义如下。

$$g = + \cdot \Big(fold\Big(\lambda w.\lambda(a,b,c,d).w + ab, 0\Big) \triangle fold\Big(\lambda w.\lambda(a,b,c,d).w + cd, 0\Big)\Big)$$

其中 $\cdot$ 表示函数复合，$\triangle$ 将两个函数作用在同一个输入上并返回其输出的元组，即 $(f \triangle g)(l) \coloneqq (f(l), g(l))$。通过元组化 *(tupling)* 规则 *[13, 14]*，该程序中的两个 *fold* 函数可以被合并成一个，并得到如下程序。

$$g = + \cdot fold\Big(\lambda(w_1, w_2).\lambda(a,b,c,d).(w_1 + ab, w_2 + cd), (0,0)\Big)$$

在实际问题中，表达式层面的等价变换往往是不足以得到高效的程序的，很多时候算法层面上的变换也是必不可少的。因此，程序演算的研究者们对各种算法展开了研究，并针对大量的常用算法提出了对应的变换规则，例如分治 [15, 16]、贪心 [17, 18]、动态规划 [19–22] 等。在对算法层面的程序变换研究过程中，一大问题在于如何处理数据结构。因为在一般的编程语言中，程序由算法与数据结构两部分组成，所以为了保证算法变换的通用性，需要保证数据结构的部分对于算法变换来说是抽象的。为了做到这一点，以 Bird 与 de Moor 为代表的研究者使用范畴论的概念构造了一个对数据结构抽象的语言模型，并对该模型上的抽象组件进行了研究 [23]。该语言模型的表达能力是显著的：只需要使用至多三个抽象组件的复合，便足以描述目前关注的所有算法问题。

本章的内容可以被分为四部分。在第 2.2 节中，本文以列表这一数据结构为例，介绍程序演算在组件层面进行程序变换的思想。接着，本文在第 2.3 节中介绍使用范畴论

概念定义的程序模型并简单展示其能力。然后在第 2.4 节中，本文对一些已有的算法变换规则进行介绍。最后，本文在第 2.5 节中简单展示一些程序演算相关的自动化工具。

2.2 列表程序上的演算

2.2.1 Bird-Meertens 形式化

Bird-Meertens 形式化 (Bird-Meertens formalism, BMF) 是由 Bird 和 Meertens 提出的用于描述列表函数的演算系统 [24]。该系统包含如下两个基本的高阶函数。

- 映射 (map) 操作 $*$ 在左侧接受一个函数，右侧接受一个列表，并将这个函数应用到列表中的每个元素。其形式化定义如下所示。

$$f * [x_1, x_2, \ldots, x_n] \coloneqq [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

- 化简 (reduce) 操作 $/$ 在左侧接受一个满足结合律的二元函数，右侧接受一个列表，并通过该函数将列表中的所有合并成为一个值。其形式化定义如下所示。

$$\oplus/[x_1, x_2, \ldots, x_n] \coloneqq x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

例 28 下面展示了几个使用 *BMF* 描述函数的例子。

- 返回列表最大元素的函数 max 可以被定义为 $\uparrow/$，其中 $\uparrow$ 返回两个元素中的较大值，即 $a \uparrow b \coloneqq \max(a, b)$。对应地，返回最小元素的 min 可以被定义为 $\downarrow/$。
- 返回列表第一个元素的函数 $head$ 可以被定义为 $\lessdot/$，其中 $\lessdot$ 返回两个元素中的第一个，即 $a \lessdot b \coloneqq a$。对应地，返回最后元素的 $last$ 可以被定义为 $\gtrdot/$。

下面展示了几个与 $*$ 和 $/$ 有关的变换规则，其中 K 为常函数，定义为 $K\ a\ b = a$，$id_\oplus$ 表示二元操作符 $\oplus$ 的单位元，$[\cdot]$ 为大小为 1 的列表的构造函数，定义为 $[\cdot]\ a = [a]$，$+\!\!+$ 为列表的合并操作，定义为 $[x_1, \ldots, x_n] +\!\!+ [y_1, \ldots, y_m] = [x_1, \ldots, x_n, y_1, \ldots, y_m]$。

空规则	$f * \cdot K\ [] = K\ []$	$\oplus/ \cdot K\ [] = K\ id_\oplus$
单点规则	$f * \cdot [\cdot] = [\cdot] \cdot f$	$\oplus/ \cdot [\cdot] = id$
合并规则	$f * \cdot +\!\!+/ = +\!\!+/ \cdot (f*)*$	$\oplus/ \cdot +\!\!+/ = \oplus/ \cdot (\oplus/)*$
分配率	$f * \cdot g* = (f \cdot g)*$	

例 29 考虑如下的程序。给定一个元素为列表的列表，该程序首先将 g 应用在每一个列表上（即 $g*$），接着把所有列表合并起来得到一个列表（即 $+\!\!+/$），然后把 f 应用在每一个元素上（即 $f*$），最后通过二元操作符 $\oplus$ 将所有值合并（即 $\oplus/$）。该遍历了所有元素

四遍，而通过之前定义的规则与以下变换，可以得到一个只遍历所有元素两遍的程序。

$$\begin{aligned}\oplus/\cdot \underline{f*\cdot +\!\!+/}\cdot g* &= \underline{\oplus/\cdot +\!\!+/}\cdot f**\cdot g* \\ &= \oplus/\cdot \underline{(\oplus/)*\cdot f**\cdot g*} \\ &= \oplus/\cdot(\oplus/\cdot f*\cdot g)*\end{aligned}$$

除了组件 $*$ 和 $/$ 以外，下面列举了一些其他的列表相关的常用组件。

- $\not\rightarrow$ 和 $\not\leftarrow$ 分别表示从左到右和从右到左的化简操作，其形式化定义如下。因为这两个组件规定了化简的顺序，所以它们接受不满足结合律的操作符 $\oplus$。

$$\begin{aligned}\oplus\not\rightarrow_e[x_1,\ldots,x_n] &= ((e\oplus x_1)\oplus x_2)\oplus\cdots\oplus x_n \\ \oplus\not\leftarrow_e[x_1,\ldots,x_n] &= x_1\oplus\cdots\oplus(x_{n-1}\oplus(x_n\oplus e))\end{aligned}$$

- $\not\Rightarrow$ 和 $\not\Leftarrow$ 分别表示从左到右和从右到左的累加操作。它们的计算过程与 $\not\rightarrow$ 和 $\not\leftarrow$ 类似，但是会返回所有的中间结果。其形式化定义如下。

$$\begin{aligned}\oplus\not\Rightarrow_e[x_1,\ldots,x_n] &= [e, e\oplus x_1, (e\oplus x_1)\oplus x_2,\ldots,((e\oplus x_1)\oplus x_2)\oplus\cdots\oplus x_n] \\ \oplus\not\Leftarrow_e[x_1,\ldots,x_n] &= [x_1\oplus\cdots\oplus(x_{n-1}\oplus(x_n\oplus e)),\ldots,x_{n-1}\oplus(x_n\oplus e), x_n\oplus e, e]\end{aligned}$$

2.2.2　列表同态定理

列表同态是一类特殊的列表上的递归函数。函数 h 是一个列表同态当且仅当存在二元操作符 $\odot$ 满足如下公式。

$$\forall l_1, l_2 \in \mathtt{List}, h\ (l_1 +\!\!+ l_2) = (h\ l_1)\odot(h\ l_2)$$

列表同态在并行计算领域十分重要，因为它可被分治的并行化算法高效计算 [15, 25–28]。给定对应操作符 $\odot$ 的列表同态 h 和列表 l，$h\ l$ 的值可以通过如下三步计算得到：(1) 将 l 拆分为 $l_1 +\!\!+ l_2$，(2) 并行地递归计算 $o_1 = h\ l_1$ 和 $o_2 = h\ l_2$，(3) 返回 $o_1\odot o_2$。

例 30 如下所示，在第 *2.2.1* 节中介绍的组件 $*, /$ 均为特殊的列表同态。

$$f*(l_1 +\!\!+ l_2) = (f*l_1) +\!\!+ (f*l_2) \qquad \oplus/(l_1 +\!\!+ l_2) = (\oplus/l_1)\oplus(\oplus/l_2)$$

而组件 $\not\rightarrow, \not\leftarrow, \not\Rightarrow$ 和 $\not\Leftarrow$ 描述的函数可能不是列表同态，因为它们要求了运算顺序。

列表同态可以通过组件 *hom* 表示。给定具有单位元 e 的二元操作符 $\odot$ 和函数 f，*hom* $(\odot)$ f 唯一地描述了如下的列表同态 h。

$$h\ [] = e \qquad h\ [a] = f\ a \qquad h\ (l_1 +\!\!+ l_2) = (h\ l_1)\odot(h\ l_2)$$

程序演算领域的研究人员提出了三个列表同态定理用于刻画列表同态的性质。

定理 1 (第一列表同态定理) 列表同态都等于一次映射和一次化简的复合.

$$hom\ (\odot)\ f = \odot/ \cdot f*$$

定理 2 (第二列表同态定理 [29]) 列表同态可以被从左到右或从右到左计算。

$$hom\ (\odot)\ f = \oplus \not\rightarrow_e \qquad a \oplus b \coloneqq a \odot (f\ b)$$
$$hom\ (\odot)\ f = \otimes \not\leftarrow_e \qquad a \otimes b \coloneqq (f\ a) \odot b$$

其中 e 是二元操作符 $\odot$ 的单位元。

定理 3 (第三列表同态定理 [30]) 任何可以同时被从左到右计算和从右到左计算的函数都是列表同态.

根据定义，前两个列表同态定理显然成立，因此接下来本文给出第三列表同态定理的证明。首先考虑如下引理。

引理 1 列表函数 h 是一个列表同态当且仅当它满足如下公式。

$$\forall l_1, l_2, l_1', l_2' \in \texttt{List}, h\ l_1 = h\ l_1' \wedge h\ l_2 = h\ l_2' \implies h\ (l_1 +\!\!+ l_2) = h\ (l_1' +\!\!+ l_2') \tag{2.1}$$

证明 2.2.1 根据列表同态定义，必要性显然。要证明充分性，令函数 g 为 h 的右逆，*i.e.*, $h \cdot g \cdot h = h$。定义二元操作符 $\odot$ 如下所示。

$$a \odot b \coloneqq h\ ((g\ a) +\!\!+ (g\ b))$$

现在证明函数 h 是一个对应操作符 $\odot$ 的列表同态。对于任意两个列表 l_1, l_2，令 $l_1' = g\ (h\ l_1), l_2' = g\ (h\ l_2)$，则根据 g 的定义，有 $h\ l_1 = h\ l_1'$ 且 $h\ l_2 = h\ l_2'$。因此根据公式 *2.1*，有 $h\ (l_1 +\!\!+ l_2) = h\ (l_1' +\!\!+ l_2')$。从而可以得到如下推导。

$$h\ (l_1 +\!\!+ l_2) = h\ (l_1' +\!\!+ l_2') = h\ ((g\ (h\ l_1)) +\!\!+ (g\ (h\ l_2))) = (h\ l_1) \odot (h\ l_2)$$

假设函数 h 等于 $\oplus \not\rightarrow_e$ 和 $\otimes \not\leftarrow_e$，现在证明函数 h 满足公式 2.1。假设 $h\ l_1 = h\ l_1' = e_1$ 且 $h\ l_2 = h\ l_2' = e_2$，则可以进行如下推导。

$$\begin{aligned} h\ (l_1 +\!\!+ l_2) &= \oplus \not\rightarrow_e (l_1 +\!\!+ l_2) = \oplus \not\rightarrow_{e_1} l_2 = \oplus \not\rightarrow_e (l_1' +\!\!+ l_2) = h\ (l_1' +\!\!+ l_2) \\ &= \otimes \not\leftarrow_e (l_1' +\!\!+ l_2) = \otimes \not\leftarrow_{e_2} l_1' = \otimes \not\leftarrow_e (l_1' +\!\!+ l_2') = h\ (l_1' +\!\!+ l_2') \end{aligned}$$

2.2.3 霍纳法则和最大子段和问题

霍纳法则指代如下的数学等式，它可以通过不断运用加法乘法的分配率得到。

$$\begin{aligned} x_1 \times x_2 \times \cdots \times x_n + x_2 \times \cdots \times x_n + \ldots + x_{n-1} \times x_n + x_n + 1 = \\ ((1 \times x_1 + 1) \times x_2 + 1) \cdots \times x_n + 1 \end{aligned}$$

在 BMF 中，该法则可以被定义为如下等式。

$$+\not\rightarrow_0 \cdot (\times \not\rightarrow_1) * \cdot tails = \odot \not\rightarrow_1$$

其中 *tails* 的定义如下，它返回列表的所有后缀，而操作符 $\odot$ 定义为 $x \odot y = xy + 1$。

$$tails\ [x_1, \ldots, x_n] \coloneqq [[x_1, \ldots, x_n], [x_2, \ldots, x_n], \ldots, [x_n], []]$$

因为霍纳法则只利用了加法乘法的分配律，所以可以被用于任意具有分配率的操作符。

定理 4 (泛化的霍纳法则 [31]) 假设 $\oplus$ 和 $\otimes$ 是满足结合律的两个二元操作符，即 $(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$，且 $\oplus$ 具有左单位元 a，则如下等式始终成立。

$$\oplus \not\rightarrow_a \cdot (\otimes \not\rightarrow_b) * \cdot tails = \odot \not\rightarrow_b$$

其中二元操作符 $\odot$ 的定义为 $x \odot y = (x \otimes y) \oplus b$

在本节的剩余部分，本文展示程序演算技术如何为一个经典的算法问题，最大子段和问题 [32]，推导出一个高效算法。给定一个包含整数（可能有负数）的列表，最大子段和问题的目标是找到该列表的一个子段并最大化这个子段中所有数的和。

为了形式化地描述最大子段和问题，本文首先引入两个函数 *inits* 和 *segs*。

- *inits* 依次返回一个列表的所有前缀，它可以被定义为 $\oplus \not\Rightarrow_{[]}$，其中操作符 $\oplus$ 的定义为 $xs \oplus x \coloneqq xs +\!\!+ [x]$。
- *segs* 返回一个列表的所有子段，它可以被定义为 $+\!\!+/ \cdot tails * \cdot inits$。

利用这函数 *segs*，最大子段和问题 *mss* 可以被定义为 $mss \coloneqq max \cdot sum * \cdot segs$。

尽管该定义十分直接，但是它导出的程序是十分低效：它首先对输入的列表算出所有的子段，对每一个子段计算所有元素的和，并最后求出其中的最大值。当输入列表的长度为 n 时，这一程序的时间复杂度为 $O(n^3)$。使用泛化后的霍纳法则以及组件之间的变换规则，可以通过如下程序变换得到一个时间复杂度 $O(n)$ 的高效程序。

$$\begin{aligned}
\underline{mss} &= max \cdot sum * \cdot \underline{segs} && mss\text{的定义} \\
&= max \cdot \underline{sum * \cdot +\!\!+/} \cdot tails * \cdot inits && segs\text{的定义} \\
&= \underline{max \cdot +\!\!+/} \cdot sum ** \cdot tails * \cdot inits && \text{合并规则} \\
&= max \cdot \underline{max * \cdot sum ** \cdot tails *} \cdot inits && max = \uparrow/,\ \text{合并规则} \\
&= max \cdot (\underline{max \cdot sum * \cdot tails}) * \cdot inits && \text{映射的分配率} \\
&= max \cdot \underline{(\otimes \not\rightarrow_0) * \cdot inits} && \text{霍纳法则，} x \otimes y \coloneqq (x + y) \uparrow 0 \\
&= \underline{max \cdot \otimes \not\Rightarrow_0} && \text{规则} (\oplus \not\rightarrow_a) * \cdot inits = \oplus \not\Rightarrow_a \\
&= fst \cdot \odot \not\rightarrow_{(0,0)} && (u, v) \odot x \coloneqq (u \uparrow (v + x) \uparrow 0, (v + x) \uparrow 0)
\end{aligned}$$

其中 fst 表示取二元组的第一个元素，即 $fst\ (a, b) \coloneqq a$。在推导的最后一步中，用到了 max 的从左到右的计算形式 $\uparrow\!\not\rightarrow_0$ 以及如下的化简-累加融合规则。

$$\oplus \not\rightarrow_a \cdot \otimes \not\twoheadrightarrow_b = fst \cdot \odot \not\rightarrow_{(a\oplus b, b)} \qquad (u, v) \odot x \coloneqq (u \oplus (v \otimes x), v \otimes x)$$

2.3 范畴论概念下的程序模型

在本节中，本文简单介绍程序演算领域中使用范畴论定义的程序模型。由于篇幅原因，本文只会在直观上介绍相关的概念。有关该模型严谨的描述可以参考 Bird 与 de Moor 出版的程序演算专著 [23]。

2.3.1 范畴、函子、递归数据结构、态射与类型函子

2.3.1.1 范畴

一个范畴 (category) **C** 包括以下三个要素。

- 一个由对象组成的类 $ob(\mathbf{C})$。
- 由物件间的态射构成的类 $hom(\mathbf{C})$。每一个态射 f 都有唯一的一个源对象 a 和目标对象 b，记作 $f : b \leftarrow a$。
- 态射类上的二元操作复合。给定任意两个态射 $f : a \leftarrow b$ 和 $g : b \leftarrow c$，它们的复合（记作 $f \cdot g$）是一个对象 c 到对象 a 的态射。

态射的复合操作需要满足如下的两个公理。

- （结合律）对于任意态射 $f : a \leftarrow b, g : b \leftarrow c, h : c \leftarrow d$, $f \cdot (g \cdot h) = (f \cdot g) \cdot h$。
- （单位元）对于任意对象 a，存在态射 $id_a : a \leftarrow a$ 使得对于任意态射 $f : a \leftarrow b$，都有 $id_a \cdot f = f = f \cdot id_b$。

在本小节中，本文只考虑所有函数范畴 **Fun**。在该范畴中，每一个对象都是一个集合，而从集合 A 到集合 B 的态射包含所有从 A 映射到 B 的函数。为了区分，本文使用大写字母 $A, B, \ldots$ 来表示集合，使用小写字母 $f, g, \ldots$ 来表示函数。在第 2.3.3 小节中，本文会介绍另一个被程序演算使用的范畴，所有关系的范畴 **Rel**。

2.3.1.2 函子与多项式函子

函子 (functor，记作 F) 是范畴上的同态。它将对象映射到对象，态射映射到态射，并在映射结果上保证单位元与态射复合的一致性，即如下公式。

$$\forall A, \mathsf{F} id_A = id_{\mathsf{F}A} \qquad \forall f, g, \mathsf{F}(f \cdot g) = \mathsf{F}f \cdot \mathsf{F}g$$

在程序演算中，*多项式函子*被用于定义数据类型，它由以下基础函子构成。

- 单位函子 I 保持对象与态射不变，即 $\mathsf{I}A \coloneqq A, \mathsf{I}f \coloneqq f$。

- 对于任意对象 B，常函子 $!B$ 将对象映射到 A，即 $(!B)A \coloneqq B, (!B)f \coloneqq id_B$。
- 二元函子 $\times$ 表示直积。在 **Fun** 中，该函子以及相关函数的定义如下。

$$A \times B \coloneqq \{(a,b) | a \in A, b \in B\} \qquad (f \times g)(a,b) \coloneqq (f\ x, g\ x)$$
$$\pi_1\ (a,b) \coloneqq a \qquad \pi_2\ (a,b) \coloneqq b \qquad (f \triangle g)\ a = (f\ a, g\ a)$$

- 二元函子 $+$ 表示直和。在 **Fun** 中，该函子以及相关函数的定义如下。

$$A + B \coloneqq \{1\} \times A \cup \{2\} \times B \qquad (f_1 + f_2)\ (i,x) \coloneqq (i, f_i\ x)$$
$$(f_1 \triangledown f_2)\ (i,x) \coloneqq f_i\ x$$

虽然上述定义中，$\times$ 和 $+$ 是二元函子，但是它们以及相关函数都可以被自然地推广到多元的情况。举例来说，在 **Fun** 中，n 个类型的直和 $\sum_{i=1}^{n} A_i$ 为 $\cup_{i=1}^{n}\{i\} \times X_i$。

2.3.1.3　递归数据结构

在范畴论中，多项式函子可以被用于定义递归数据结构 [33–36]。在这一小节中，本文使用两个例子来介绍其对应关系。首先考虑所有元素均在集合 A 内的列表形成的集合，记作 $List\ A$。下列公式给出了该集合的递归定义。

$$List\ A = Nil \mid Cons\ (A, List\ A)$$

该数据结构的递归结构可以被描述为函子 $\mathsf{F}_{LA} = !\mathbf{1} + !A \times \mathsf{I}$，其中 $\mathbf{1}$ 表示一个大小为 1 的集合，其元素为空元组 ()。直观来说，F_{LA} 是函子 $!\mathbf{1}$ 和 $!A \times \mathsf{I}$ 的直和，分别对应 Nil 和 $Cons$ 的情况。Nil 不需要任何参数，因此对应只包含空元组的函子 $!\mathbf{1}$。而 $Cons$ 接受一个由 A 集合内的元素和递归列表组成的二元组，因此对应函子 $!A \times \mathsf{I}$，其中 I 表示递归。根据该直观解释，可以得到 $List\ A$ 的构造函数 $in_{\mathsf{F}_{LA}} : List\ A \leftarrow \mathsf{F}_{LA}(List\ A)$，如下所示。

$$in_{\mathsf{F}_{LA}} = Nil \triangledown Cons$$

实际上，$List\ A$ 是满足方程 $X = in_{\mathsf{F}_{LA}}(\mathsf{F}_{LA}\ X)$ 的最小集合 X [33]。对应地，可以得到 $in_{\mathsf{F}_{LA}}$ 的逆函数，即 $List\ A$ 的展开函数 $out_{\mathsf{F}_{LA}} : \mathsf{F}_{LA}(List\ A) \leftarrow List\ A$。

$$out_{\mathsf{F}_{LA}} = \lambda xs.\text{case}\ xs\ of\ Nil \rightarrow (1, ()); Cons\ (a, as) \rightarrow (2, (a, as)).$$

二叉树同样可以通过类似的方式对应到多项式函子。考虑所有元素均在集合 A 内的二叉树形成的集合，记作 $BTree\ A$。下列公式给出了该集合的递归定义。

$$BTree\ A = Leaf\ A \mid Node\ (BTree\ A, BTree\ A)$$

该递归定义可以对应到如下的函子 F_{TA} 和构造函数 $in_{\mathsf{F}_{TA}}$。

$$\mathsf{F}_{TA} = !A + \mathsf{I} \times \mathsf{I} \qquad in_{\mathsf{F}_{TA}} = Leaf \triangledown Node$$

2.3.1.4 态射

在这一小节中，本文介绍程序演算领域关注的与递归数据结构有关的组件，它们描述了常见的递归函数。给定由多项式函子 F 定义的数据结构 T 与函数 $f : A \leftarrow \mathsf{F}A$，下态射 (catamorphism) $(\!|f|\!)_{\mathsf{F}} : A \leftarrow T$ 的定义如下 [34, 36]。

$$h = (\!|f|\!)_{\mathsf{F}} \equiv h \cdot in_{\mathsf{F}} = f \cdot \mathsf{F}h$$

为了方便，当下态射对应的函子不存在歧义时，本文隐去其下标并使用简写 $(\!|\mathsf{F}|\!)$。因为 out_{F} 是 in_{F} 的逆函数，所以有 $(\!|f|\!) = f \cdot \mathsf{F}(\!|f|\!) \cdot out_{\mathsf{F}}$。该等式说明 $(\!|f|\!)$ 是数据结构 T 上的一个递归函数：它通过 out_{F} 将数据结构展开，并通过函数 f 将展开结果合并。根据例 31 可以得到，下态射在列表上的行为类似规约操作。

例 31 给定数据结构 $T = List\ Int$ 和函数 $f = zero \triangledown plus$，其中 $zero$ 接受空元组返回 0，$plus$ 接受两个整数返回它们的和，下态射 $(\!|f|\!)$ 递归地计算一个整数列表中所有元素的和。下面展示了 $(\!|f|\!)\ [1,2]$ 的计算过程。

$$\begin{aligned}
\underline{(\!|f|\!)\ [1,2]} &= (f \cdot \mathsf{F}(\!|f|\!) \cdot \underline{out})\ [1,2] = (f \cdot \underline{\mathsf{F}(\!|f|\!)})\ (2,1,[2]) = f\ (2,1,\underline{(\!|f|\!)\ [2]}) \\
&= f\ (2,1,(f \cdot \underline{\mathsf{F}(\!|f|\!) \cdot out})\ [2]) = f\ (2,1,f\ (2,2,\underline{(\!|f|\!)\ []})) \\
&= f\ (2,1,f\ (2,2,(f \cdot \underline{\mathsf{F}(\!|f|\!) \cdot out})\ [])) = f\ (2,1,f\ (2,2,\underline{f\ (1)})) \\
&= f\ (2,1,\underline{f\ (2,2,0)}) = \underline{f\ (2,1,2)} = 3
\end{aligned}$$

在变换下态射时，融合是一个重要的变换过程：它给出了下态射和另一个函数的复合可以被合并成一个下态射的充分条件。

定理 5 (融合定理) 给定对应多项式函子 F 的数据结构，对于任意态射 $h : A \leftarrow B, f : B \leftarrow \mathsf{F}B, g : A \leftarrow \mathsf{F}A$，下列公式始终成立。

$$h \cdot f = g \cdot \mathsf{F}h \implies h \cdot (\!|f|\!) = (\!|g|\!)$$

证明 2.3.1 根据下态射的定义，只需要证明 $h \cdot (\!|f|\!) \cdot in = g \cdot \mathsf{F}(h \cdot (\!|f|\!))$。其证明如下。

$$h \cdot (\!|f|\!) \cdot in = h \cdot f \cdot \mathsf{F}(\!|f|\!) = g \cdot \mathsf{F}h \cdot \mathsf{F}(\!|f|\!) = g \cdot \mathsf{F}(h \cdot (\!|f|\!))$$

与下态射对应地，给定由多项式函子 F 定义的数据结构 T 与函数 $f : \mathsf{F}A \leftarrow A$，上态射 (anamorphism) $[\!(f)\!]_{\mathsf{F}} : T \leftarrow A$ 的定义如下 [34]。

$$h = [\!(f)\!]_{\mathsf{F}} \equiv out_{\mathsf{F}} \cdot h = \mathsf{F}h \cdot f$$

同样，在对应的函子不存在歧义时，本文将上态射简写为 $[\![f]\!]$。根据 out_{F} 与 in_{F} 的逆函数关系，可以得到 $[\![f]\!]$ 的递归形式 $[\![f]\!] = in_{\mathsf{F}} \cdot \mathsf{F}[\![f]\!] \cdot f$。该等式表明 $[\![f]\!]$ 通过函数 f 将输入值展开，并最终合并成为一个数据结构。上态射定义了数据结构的构造过程。

例 32 给定数据结构 $T = List\ Int$ 和通过如下定义的函数 $f : (!1+!Int\times!Int) \leftarrow Int$。

$$f\ 0 \coloneqq (1) \qquad f\ n \coloneqq (2, 0, n-1), n \geq 1$$

上态射 $[\![f]\!]$ 接受一个整数 n，并返回一个长度为 n 且元素全为 0 的数组。下面展示了 $[\![f]\!]\ 2$ 的计算过程。

$$\begin{aligned}
\underline{[\![f]\!]\ 2} &= (in \cdot \mathsf{F}[\![f]\!] \cdot \underline{f)\ 2} = \underline{(in \cdot \mathsf{F}[\![f]\!])\ (2,0,1)} = in\ (2, 0, \underline{[\![f]\!]\ 1)} \\
&= in\ (2, 0, \underline{(in \cdot \mathsf{F}[\![f]\!] \cdot f)\ 1}) = in\ (2, 0, in\ (2, 0, \underline{[\![f]\!]\ 0})) \\
&= in\ (2, 0, in\ (2, 0, \underline{(in \cdot \mathsf{F}[\![f]\!] \cdot f)\ 0})) = in\ (2, 0, in\ (2, 0, \underline{in\ (1)})) \\
&= in\ (2, 0, \underline{in\ (2, 0, [])}) = \underline{in\ (2, 0, [0])} = [0, 0]
\end{aligned}$$

给定由多项式函子 F 定义的数据结构 F 与函数 $\psi : \mathsf{F}B \leftarrow B$ 和 $\phi : A \leftarrow \mathsf{F}A$，木态射 (hylomorphism) $[\![\phi, \psi]\!]_{\mathsf{F}} : A \leftarrow B$ 为方程 $h = \phi \cdot \mathsf{F}h \cdot \psi$ 的最小不动点 h [37]。该定义直接给出了木态射的递归形式：它通过 ψ 将输入值展开，并通过 ϕ 将展开的结果合并。因此，可以证明 $[\![\phi, \psi]\!] = (\!|\phi|\!) \cdot [\![\psi]\!]$，即一个木态射是一个下态射和一个上态射的复合。

例 33 给定对应整数列表的函子 $\mathsf{F}_1 = !1+!Int \times \mathsf{I}$ 和如下定义的两个函数 $\phi_1 : Int \leftarrow \mathsf{F}_1 Int$ 和 $\psi_1 : \mathsf{F}_1 Int \leftarrow Int$，木态射 $[\![\phi_1, \psi_1]\!]$ 为阶乘函数。$[\![\phi_1, \psi_1]\!]\ 3$ 的一种计算过程如下所示。

$$\psi_1\ 0 \coloneqq (1) \qquad \psi_1\ n \coloneqq (2, n, n-1), n \geq 1$$

$$\phi_1\ (1) \coloneqq 1 \qquad \phi_1\ (2, a, b) \coloneqq a \times b$$

$$\underline{[\![\phi_1, \psi_1]\!]\ 3} = (\!|\phi_1|\!) \cdot \underline{[\![\psi_1]\!])\ 3} = \underline{(\!|\phi_1|\!)\ [3, 2, 1]} = 6$$

给定对应树结构的函子 $\mathsf{F}_2 = !1 + \mathsf{I} \times \mathsf{I}$ 和如下定义的两个函数 $\phi_2 : Int \leftarrow \mathsf{F}_2 Int$ 和 $\psi_2 : \mathsf{F}_2 Int \leftarrow Int$，木态射 $[\![\phi_2, \psi_2]\!]$ 为斐波那契函数。$[\![\phi_2, \psi_2]\!]\ 3$ 的一种计算过程如下所示。

$$\psi_2\ 0 \coloneqq (1) \qquad \psi_2\ 1 \coloneqq (1) \qquad \psi_2\ n \coloneqq (2, n-1, n-2), n \geq 2$$

$$\phi_2\ (1) \coloneqq 1 \qquad \phi_2\ (2, a, b) \coloneqq a + b$$

$$\underline{[\![\phi_2, \psi_2]\!]\ 3} = (\!|\phi_2|\!) \cdot \underline{[\![\psi_2]\!])\ 3} = \underline{(\!|\phi_2|\!)\ (in\ (2, in\ (2, in\ (1), in\ (1)), in\ (1)))} = 3$$

木态射的表达能力是十分强大的，它几乎可以描述实际中的所有递归函数 [38]。

2.3.1.5 类型函子

在第 2.3.1.3 节中，本文以列表 *List A* 和二叉树 *BTree A* 为例，使用多项式函子定义了递归数据结构。在这个过程中，*List* 与 *BTree* 可以被看做对象到对象的函数：*List A* 与 *BTree A* 接受集合 A 并分别返回所有元素均在集合 A 内的列表与二叉树集合。实际上，它们的定义可以被扩充为函子，即 *类型函子*。

给定具有两个参数的函子 $\mathsf{F}(A,B)$，令 $\mathsf{T}A$ 为固定参数 A 后得到的递归数据结构，即方程 $X = \mathsf{F}(A,X)$ 的最小解。对于态射 $f : A \leftarrow B$，定义 $\mathsf{T}f : \mathsf{T}A \leftarrow \mathsf{T}B$ 为下态射 $(\!|in_{\mathsf{T}B} \cdot \mathsf{F}(f, id_{\mathsf{T}B})|\!)$。要证明函子 T 的定义是合法的，需要验证 T 保持态射的单位元与复合一致。首先，下列公式验证了函子 T 保持态射的单位元一致。

$$\mathsf{T}id_A = (\!|in_{\mathsf{T}A} \cdot \mathsf{F}(id_A, id_{\mathsf{T}A})|\!) = (\!|in_{\mathsf{T}A} \cdot id_{\mathsf{F}\mathsf{T}A}|\!) = (\!|in_{\mathsf{T}A}|\!) = id_{\mathsf{T}A}$$

接着，要证明 T 保持态射的复合一致，根据下态射定义，需要证明如下等式。

$$\mathsf{T}f \cdot \mathsf{T}g = \mathsf{T}(f \cdot g) \iff (\mathsf{T}f \cdot \mathsf{T}g) \cdot in = (in \cdot \mathsf{F}(f \cdot g, id)) \cdot \mathsf{F}(id, \mathsf{T}f \cdot \mathsf{T}g)$$

其中第二个等式的证明过程如下所示。

$$\begin{aligned}\mathsf{T}f \cdot \mathsf{T}g \cdot in &= \mathsf{T}f \cdot (in \cdot \mathsf{F}(g, id)) \cdot \mathsf{F}(id, \mathsf{T}g) = in \cdot \mathsf{F}(f, id) \cdot \mathsf{F}(id, \mathsf{T}f) \cdot \mathsf{F}(g, id) \cdot \mathsf{F}(id, \mathsf{T}g) \\ &= in \cdot \mathsf{F}(f \cdot g, \mathsf{T}f \cdot \mathsf{T}g) = in \cdot \mathsf{F}(f \cdot g, id) \cdot \mathsf{F}(id, \mathsf{T}f \cdot \mathsf{T}g)\end{aligned}$$

直观上来说，类型 $\mathsf{T}A$ 表示了元素类型为 A 的一系列数据结构，而态射 $\mathsf{T}f$ 将 f 应用到了其中的每一个元素上。在列表上，$\mathsf{T}f$ 的行为等价于 BMF 中的映射操作 $f*$。

作为融合定理（定理 5）的特例，可以得到关于下态射与类型函子的融合引理。

引理 2 *给定具有两个参数的函子 F 和对应的类型函子 T，对于任意态射 $f : A \leftarrow B, g : C \leftarrow \mathsf{F}(B,C)$，下列等式始终成立。*

$$(\!|g|\!) \cdot \mathsf{T}f = (\!|g \cdot \mathsf{F}(f, id)|\!)$$

证明 2.3.2 *根据融合定理（定理 5），只需要证明如下等式。*

$$(\!|g|\!) \cdot in \cdot \mathsf{F}(f, id) = g \cdot \mathsf{F}(f, id) \cdot \mathsf{F}(id, (\!|g|\!))$$

该等式的证明过程如下所示。

$$(\!|g|\!) \cdot in \cdot \mathsf{F}(f, id) = g \cdot \mathsf{F}(id, (\!|g|\!)) \cdot \mathsf{F}(f, id) = g \cdot \mathsf{F}(f, (\!|g|\!)) = g \cdot \mathsf{F}(f, id) \cdot \mathsf{F}(id, (\!|g|\!))$$

2.3.2 范畴论模型的应用

在本节中，本文介绍几个在范畴论模型下进行程序变换的例子。

2.3.2.1　香蕉分离定理

香蕉分离定理可以将两个作用于同一个递归数据结构的下态射融合成一个下态射，从而减少对数据结构的遍历次数 [39]。该定理也被称为元组化 [40]。

定理 6 (香蕉分离定理) 给定下态射 $(\!|f|\!)_{\mathsf{F}}$ 和 $(\!|g|\!)_{\mathsf{F}}$，如下等式总是成立。

$$(\!|f|\!)_{\mathsf{F}} \triangle (\!|g|\!)_{\mathsf{F}} = (\!|(f \cdot \mathsf{F}\pi_1) \triangle (g \cdot \mathsf{F}\pi_2)|\!)_{\mathsf{F}}$$

证明 2.3.3 根据下态射的定义，只需要证明如下等式。

$$((\!|f|\!) \triangle (\!|g|\!)) \cdot in = ((f \cdot \mathsf{F}\pi_1) \triangle (g \cdot \mathsf{F}\pi_2)) \cdot \mathsf{F}((\!|f|\!) \triangle (\!|g|\!))$$

该等式的证明过程如下所示。

$$\begin{aligned}((\!|f|\!) \triangle (\!|g|\!)) \cdot in &= ((\!|f|\!) \cdot in) \triangle ((\!|g|\!) \cdot in) = (f \cdot \mathsf{F}(\!|f|\!)) \triangle (g \cdot \mathsf{F}(\!|g|\!)) \\ &= (f \cdot \mathsf{F}(\pi_1 \cdot ((\!|f|\!) \triangle (\!|g|\!)))) \triangle (g \cdot \mathsf{F}(\pi_2 \cdot ((\!|f|\!) \triangle (\!|g|\!)))) \\ &= ((f \cdot \mathsf{F}\pi_1) \triangle (g \cdot \mathsf{F}\pi_2)) \cdot \mathsf{F}((\!|f|\!) \triangle (\!|g|\!))\end{aligned}$$

下面展示一个应用香蕉分离定理的例子。考虑函数 ave_L，它计算一个列表中所有数的平均值（下取整），其定义如下：

$$ave_L \coloneqq div \cdot ((\!|zero \triangledown plus|\!) \triangle (\!|zero \triangledown inc|\!))$$

其中 div 接受二元组 (a, b)，返回 $\lfloor a/b \rfloor$，$zero$ 接受空元组返回 0，$plus$ 接受 (a, b) 返回 $a + b$，inc 接受 (a, b) 返回 $b + 1$。应用香蕉分离定理，可以得到如下等价实现。

$$ave_L = div \cdot (\!|zeros \triangledown pluss|\!)$$

其中 $zeros \coloneqq zero \triangle zero$，$pluss\ (a, (b, n)) \coloneqq (a + b, n + 1)$。上述推导可以被直接推广到其他递归数据结构上，例如二叉树。

2.3.2.2　另一种形式的霍纳法则及其推广

在第 2.2.3 节中，本文介绍了霍纳法则以及其对任何具有分配率操作的推广。在这一节中，本文考虑如下所示的另一种形式的霍纳法则，并使用范畴论概念下的程序模型，将其推广到任意递归数据结构上 [41]。

$$a_0 + a_1 \times x + a_2 \times x^2 + \cdots + a_n \times x^n = a_0 + (a_1 + (a_2 + \cdots (a_n + 0) \times x \cdots) \times x) \quad (2.2)$$

为了描述这一法则，首先定义组件 tri。给定多项式函子 $\mathsf{F}(A, B)$，对应的类型函子 $\mathsf{T}A$ 和态射 $f : A \leftarrow A$，则 $trif : \mathsf{T}A \leftarrow \mathsf{T}A$ 为下态射 $(\!|in \cdot \mathsf{F}(id, \mathsf{T}f)|\!)$。直观上来说，$tri\ f$ 将

f 作用到数据结构 $\mathsf{T}A$ 中的每一个元素上，且对深度为 d 的元素重复应用了 d 次。下列公式展示了组件 *tri* 在列表上的行为，其中 f^i 表示 i 个 f 的复合。

$$tri\ f\ [x_0, x_1, \ldots, x_n] = [x_0, f\ x_1, \ldots, f^i\ a_i, \ldots, f^n\ a_n]$$

定理 7 给定具有两个参数的多项式函子 F 和对应的类型函子 T，对于任意态射 $f : A \leftarrow A, g : A \leftarrow \mathsf{F}(A, A)$，下列公式总是成立。

$$f \cdot g = g \cdot \mathsf{F}(f, f) \implies (\!|g|\!) \cdot trif = (\!|g \cdot \mathsf{F}(id, f)|\!)$$

证明 2.3.4 根据融合定理（定理 5），只需要证明如下等式。

$$(\!|g|\!) \cdot in \cdot \mathsf{F}(id, \mathsf{T}f) = g \cdot \mathsf{F}(id, f) \cdot \mathsf{F}(id, (\!|g|\!))$$

该等式的证明过程如下所示。

$$\begin{aligned}(\!|g|\!) \cdot in \cdot \mathsf{F}(id, \mathsf{T}f) &= g \cdot \mathsf{F}(id, (\!|g|\!)) \cdot \mathsf{F}(id, \mathsf{T}f) = g \cdot \mathsf{F}(id, \underline{(\!|g|\!) \cdot \mathsf{T}f}) \\ &= g \cdot \mathsf{F}(id, \underline{(\!|g \cdot \mathsf{F}(f, id)|\!)}) = g \cdot \mathsf{F}(id, f \cdot (\!|g|\!)) \\ &= g \cdot \mathsf{F}(id, f) \cdot \mathsf{F}(id, (\!|g|\!))\end{aligned}$$

其中第一个下划线使用的规则为下态射于类型函子的融合引理（引理 2），第二个下划线使用了断言 $f \cdot (\!|g|\!) = (\!|g \cdot \mathsf{F}(f, id)|\!)$，根据融合定理和前条件，该断言的证明如下。

$$f \cdot g = g \cdot \mathsf{F}(f, f) = g \cdot \mathsf{F}(f, id) \cdot \mathsf{F}(id, f)$$

例 34 等式 2.2 是定理 7 的特例。可以验证当 $\mathsf{T}A \coloneqq List\ A, f\ a \coloneqq a \times x, g\ (1) \coloneqq 0$ 且 $g\ (2, a, b) \coloneqq a + b$ 时，定理 7 的形式与等式 2.2 完全相同。

经过推广，定理 7 可以用于优化其他数据结构上的程序。考虑由函子 $\mathsf{F}_T(A, B) = A + B \times B$ 定义的二叉树函子 T_T。下面定义的程序 *depth* 可以用于计算二叉树的深度。

$$depth = tmax \cdot tri\ succ \cdot \mathsf{T}_T\ zero$$

其中函数 $tmax \coloneqq (\!|id\triangledown \uparrow|\!)$ 求出二叉树上所有数的最大值，$succ\ a \coloneqq a + 1$，函数 *tri succ* 给二叉树上深度为 i 的数加上 i，函数 $\mathsf{T}_T\ zero$ 将二叉树中的所有数都变成 0。因此，程序 *depth* 首先使用 $tri\ succ \cdot \mathsf{T}_T\ zero$ 将树上所有的数字变成其深度，然后使用 *tmax* 求出深度的最大值。假设树的深度还有大小都是 $O(n)$ 级别的，那么直接运行程序 *depth* 的时间复杂度是 $O(n^2)$ 的。使用推广后的霍纳法则（定理 7）和变换规则，可以得到一个线性时间复杂度的算法。

容易验证 $succ \cdot (id\triangledown \uparrow) = (id\triangledown \uparrow) \cdot (succ + succ \times succ)$，所以可以进行如下的推导。

$$\begin{aligned}\underline{tmax \cdot tri\ succ} \cdot \mathsf{T}_T\ zero &= (\!|(id \triangledown \uparrow) \cdot (id + succ \times succ)|\!) \cdot \mathsf{T}_T\ zero \\ &= (\!|id \triangledown (\uparrow \cdot (succ \times succ))|\!) \cdot \mathsf{T}_T\ zero \\ &= \underline{(\!|id \triangledown (succ \cdot \uparrow)|\!) \cdot \mathsf{T}_T\ zero} = (\!|zero \triangledown (succ \cdot \uparrow)|\!)\end{aligned}$$

其中第一个下划线使用的规则是霍纳法则（定理 7），第二个下划线使用的是下态射与类型函子的融合引理（引理 2）。

2.3.2.3　酸雨定理

从前几节的例子可以看出，推导高效算法的一个重要方式是将原程序中的组件融合到一起。目前已经介绍的两个融合规则（定理 7 和引理 2）都局限在下态射层面，表达能力有限。在这一小节中，本文将简单介绍酸雨定理 (acid rain theorem) [42]，它给出了木态射层面的融合规则。该定理依赖自然变换 (natural transformation) 的概念与木态射的三元组形式。

定义 3 (自然变换) 给定函子 F 和 G，多态函数 $\phi_A : \mathsf{F}A \leftarrow \mathsf{G}A$ 是一个 G 到 F 的自然变换（记作 $\phi : \mathsf{F} \leftarrow \mathsf{G}$）当且仅当对于任意的态射 $f : A \leftarrow B$，都有 $\phi_A \cdot \mathsf{G}f = \mathsf{F}f \cdot \phi_B$。

例 35 多态函数 $assoc\ (a,(b,c)) \coloneqq ((a,b),c)$ 是一个从 $(\mathsf{I} \times \mathsf{I}) \times \mathsf{I}$ 到 $\mathsf{I} \times (\mathsf{I} \times \mathsf{I})$ 的自然变换。

直观来说，G 到 F 的自然变换保留了 G 中的信息不变，只是修改了其结构，从而在 G 处的计算可以被直接移交到 F 处进行。以下的移动引理说明了，在木态射中，一个自然变换可以自然地在上态射与下态射之间移动。

引理 3 (移动引理) 给定函子 G 与 F，对于任意的态射 $\psi : \mathsf{G}B \leftarrow B, \phi : A \leftarrow \mathsf{F}A$ 和自然变换 $\eta : \mathsf{F} \leftarrow \mathsf{G}$，都有如下等式成立。

$$[\![\phi \cdot \eta, \psi]\!]_{\mathsf{G}} = [\![\phi, \eta \cdot \psi]\!]_{\mathsf{F}}$$

证明 2.3.5 考虑任意的态射 h，利用自然变换的定义，有如下的推导。

$$h = \phi \cdot \eta \cdot \mathsf{G}h \cdot \psi \iff h = \phi \cdot \mathsf{F}h \cdot \eta \cdot \psi$$

如果将两个等式分别看成关于 h 的方程，则其解集相同。根据木态射的定义，$[\![\phi \cdot \eta, \psi]\!]$ 和 $[\![\phi, \eta \cdot \psi]\!]$ 分别是两个方程的最小不动点，因此它们也相同。

根据引理 3，可以得到木态射的三元组形式。给定函子 G, F，态射 $\psi : \mathsf{G}B \leftarrow B, \phi : A \leftarrow \mathsf{F}A$ 和自然变换 $\eta : \mathsf{F} \leftarrow \mathsf{G}$，定义 $[\![\phi, \eta, \psi]\!]_{\mathsf{F},\mathsf{G}} = [\![\phi \cdot \eta, \psi]\!]_{\mathsf{G}} = [\![\phi, \eta \cdot \psi]\!]_{\mathsf{F}}$。下面列举

了关于木态射三元组形式的一些显然的事实。

$$[\![\phi, \eta, \psi]\!]_{\mathsf{F},\mathsf{G}} = [\![\phi \cdot \eta, id, \psi]\!]_{\mathsf{G},\mathsf{G}} = [\![\phi, id, \eta \cdot \psi]\!]_{\mathsf{F},\mathsf{F}}$$

$$(\!|f|\!)_{\mathsf{F}} = [\![in_{\mathsf{F}}, id, f]\!]_{\mathsf{F},\mathsf{F}} \quad [\!(f)\!]_{\mathsf{F}} = [\![f, id, out_{\mathsf{F}}]\!]_{\mathsf{F},\mathsf{F}}$$

酸雨定理可以将一部分特殊的木态射与上态射（下态射）融合，其内容如下 [42]。

定理 8 (下态射-木态射融合定理) 给定高阶多态函数 $\tau : \forall A, (\mathsf{F}A \to A) \to \mathsf{F}A \to A$，如下等式始终成立。

$$(\!|\phi|\!)_{\mathsf{F}} \cdot [\![\tau\ in_{\mathsf{F}}, \eta, \psi]\!]_{\mathsf{F},\mathsf{G}} = [\![\tau\ \phi, \eta, \psi]\!]_{\mathsf{F},\mathsf{G}}$$

定理 9 (木态射-上态射融合定理) 给定高阶多态函数 $\sigma_A : \forall A, (A \to \mathsf{F}A) \to A \to \mathsf{F}A$，如下等式始终成立。

$$[\![\phi, \eta, \sigma\ out_{\mathsf{F}}]\!]_{\mathsf{G},\mathsf{F}} \cdot [\!(\psi)\!]_{\mathsf{F}} = [\![\phi, \eta, \sigma\ \psi]\!]_{\mathsf{G},\mathsf{F}}$$

本文略去酸雨定义的证明，因为它依赖于免费定理 (the free theorem) [43]。在本小节的最后，本文展示一个使用酸雨定理推导的例子。考虑如下定义的列表连接操作。

$$(+\!\!+\ ys) = (\!|\tau\ in|\!) \quad \tau = \lambda n \triangledown c.(\overline{(\!|n \vartriangle c|\!)\ ys}) \triangledown c$$

其中 $\lambda n \triangledown c$ 表示接受一个形如 $f_1, \triangledown f_2$ 的函数，并分别将 f_1, f_2 代入 n 和 c，$\overline{x}$ 表示一个接受空元组并返回 x 的函数。展开 $\tau\ in$，可以得到 $\overline{ys} \triangledown c$。在接受 (1) 时，$\tau\ in$ 返回 ys；在接受 $(2, x, r)$ 时，$\tau\ in$ 返回 $[x] +\!\!+ r$。因此 $(\!|\tau\ in|\!)$ 将 ys 插入到给定的列表的最后。

现在假设目标是连接后的列表的长度，即 $length \cdot (+\!\!+\ ys)$，可以得到如下推导。

$$\begin{aligned} length \cdot (+\!\!+\ ys) &= (\!|zero \triangledown succ|\!) \cdot (\!|\tau\ in|\!) = (\!|zero \triangledown succ|\!) \cdot [\![\tau\ in, id, out]\!] \\ &= (\!|\tau\ (zero \triangledown succ)|\!) = (\!|(\overline{length\ ys}) \triangledown succ|\!) \end{aligned}$$

其中倒数第二步使用了定理 8。最终得到的程序是符合直觉的：它递归地展开输入的列表，并从 ys 的长度开始每递归一层就将答案加一。相比于初始的程序 $length \cdot (+\!\!+\ ys)$，推导得到的程序不需要显示地构造出列表连接的结果，从而节约了时间与空间开销。

2.3.3 关系范畴和优化问题

一般情况下，我们可以通过定义输入到输出的单射来定义程序，即定义在每一个输入下的期望输出。此时，函数以及包含函数的范畴 **Fun** 提供了很好的抽象模型。但是在定义优化问题的时候，我们通常需要一到多的映射，因为在一个优化问题中往往有很多个合法解，甚至最优解也都不是唯一的。为了解决这一问题，Bird 和 de Moor 等人引入了关系以及包含关系的范畴 **Rel** 来建模优化问题 [23]。

2.3.3.1　关系范畴简介

给定集合 A 和集合 B，一个 B 到 A 的关系 R 可以被 $A \times B$ 的一个子集 S_R 描述，其中 $(a,b) \in S_R$ 表示 R 将 b 关联到 a，记作 aRb。本文使用编号靠后的大写字母（例如 R,S,T）来表示关系。与范畴 **Fun** 相比，**Rel** 中的对象仍然是集合，但是 B 到 A 的所有态射从 B 到 A 的函数集合扩充到了 B 到 A 的所有关系集合。下面定义了与关系有关的一些基本操作。

- 给定关系 $R : A \leftarrow B$，其逆关系 $R^\circ : B \leftarrow A$ 定义为 $bR^\circ a \iff aRb$。
- 给定关系 $R,S : A \leftarrow B$，R 是 S 的子集（记作 $R \subseteq S$）等且仅当 $aRb \Rightarrow aSb$。
- 给定关系 $R,S : A \leftarrow B$，其交 $R \cap S$ 定义为 $a(R \cap S)b \iff aRb \wedge aSb$。
- 给定关系 $R,S : A \leftarrow B$，其并 $R \cup S$ 定义为 $a(R \cup S)b \iff aRb \vee aSb$。
- 给定关系 $R : A \leftarrow B, S : B \leftarrow C$，其复合 $R \cdot S$ 定义为 $a(R \cdot S)c \iff \exists b, aRb \wedge bSc$。

与 **Fun** 类似，在 **Rel** 中同样可以通过多项式函子来定义数据结构。下面展示了函子 $\times, +$ 以及相关操作 $\triangle, \triangledown$ 在关系上的定义。

$$(a_1,a_2)(R_1 \times R_2)(b_1,b_2) \iff a_1R_1b_1 \wedge a_2R_2b_2 \quad (a_1,a_2)(R_1 \triangle S_2)b \iff a_1R_1b \wedge a_2R_2b$$

$$(i,a)(R_1 + R_2)(j,b) \iff i = j \wedge aR_ib \quad a(R_1 \triangledown R_2)(i,b) \iff aR_ib$$

范畴 **Rel** 中下态射的概念可以从 **Fun** 中的下态射延拓得到。为此，需要先引入有关幂集的一些概念。

- 对于集合 A，$\mathsf{P}A$ 表示其幂集，即 $\{A' | A' \subseteq A\}$。P 可以被定义为 **Rel** 上的一个函子，但是本文不会使用其作用在关系上的部分。
- 关系 $\in : A \leftarrow \mathsf{P}A$ 将集合关联到其元素，即 $aRA \iff a \in A$。
- 给定关系 $R : A \leftarrow B$，函数 $\Lambda R : \mathsf{P}A \leftarrow B$ 返回和一个元素关联的所有元素形成的集合，即 $\Lambda R\ b \coloneqq \{a | aRb\}$。

给定多项式函子 F 和关系 $R : A \leftarrow \mathsf{F}A$，下态射 $(\!|R|\!)_\mathsf{F}$ 的定义为 $\in \cdot (\!|\Lambda(R \cdot \mathsf{F} \in)|\!)_\mathsf{F}$。因为 $\Lambda(R \cdot \mathsf{F} \in)$ 是一个函数，所以下态射 $(\!|\Lambda(R \cdot \mathsf{F} \in)|\!)$ 已经在第 2.3.1.4 中定义了。

例 36 考虑对应列表 $List\ A$ 的函子 F 与定义如下的关系 $add : List\ A \leftarrow \mathsf{F}(List\ A)$。

$$[]\ add\ (1) \qquad xs\ add\ (2,x,xs) \qquad ([x] +\!\!+ xs)\ add\ (2,x,xs)$$

则下态射 $(\!|add|\!)$ 将一个列表关系到所有与它的子序列。其对应的函数 $f = (\!|\Lambda(add \cdot \mathsf{F} \in)|\!)$ 在列表 $[1,2]$ 上的运行过程如下所示。

$$f\ [] = (\Lambda(add \cdot \mathsf{F} \in) \cdot \mathsf{F}f \cdot out)\ [] = \Lambda(add \cdot \mathsf{F} \in)\ (1) = \{[]\}$$

$$
\begin{aligned}
f\ [2] &= (\Lambda(add \cdot \mathsf{F} \in) \cdot \mathsf{F}f \cdot out)\ [1] = (\Lambda(add \cdot \mathsf{F} \in) \cdot \mathsf{F}f)\ (2, 2, [])\\
&= \Lambda(add \cdot \mathsf{F} \in)\ (2, 2, \{[]\}) = \{a \mid a\ add\ (2, 2, [])\} = \{[], [2]\}\\
f\ [1, 2] &= (\Lambda(add \cdot \mathsf{F} \in) \cdot \mathsf{F}f \cdot out)\ [1, 2] = (\Lambda(add \cdot \mathsf{F} \in) \cdot \mathsf{F}f)\ (2, 1, [2])\\
&= \Lambda(add \cdot \mathsf{F} \in)\ (2, 1, \{[], [2]\}) = \{a | a\ add\ (2, 1, [])\} \cup \{a \mid a\ add\ (2, 1, [2])\}\\
&= \{[], [1], [2], [1, 2]\}
\end{aligned}
$$

在使用范畴 **Rel** 描述问题时，并不需要引入上态射 $[\![(R)]\!]$，因为它等价于其逆的下态射的逆，即 $(\!|R^\circ|\!)^\circ$。对应地，可以定义 **Rel** 上的木态射。给定函子 F，关系 $R : A \leftarrow \mathsf{F}A$ 和 $S : \mathsf{F}B \leftarrow B$，木态射 $[\![R, S]\!]_\mathsf{F}$ 为满足方程 $X = R \cdot \mathsf{F}X \cdot S$ 的最小解。与范畴 **Fun** 的情况相同，可以证明木态射是一个下态射和一个上态射的复合，即 $[\![R, S]\!] = (\!|R|\!) \cdot (\!|S^\circ|\!)^\circ$。

2.3.3.2 使用关系范畴描述优化问题

为了描述优化问题，需要引入操作符 *min* 来描述取最优值这一操作。给定描述大小顺序的关系 $R : A \leftarrow A$，其中 aRb 表示 a 在当前问题中不劣于 b，最优值关系 $min\ R : A \leftarrow \mathsf{P}A$ 将一个集合关联到其中的最优元素，它的定义如下。

$$a(min\ R)A \iff a \in A \wedge \forall b \in A, aRb$$

即 a 是集合 A 的一个最优元素当且仅当 a 在 A 中且 a 不劣于 A 中的任何一个元素。

一般性地，优化问题可以被看做从所有合法方案中选取最优方案的问题，因此它可以被两个关系描述。给定输入类型 A 和方案类型 B，关系 $S : B \leftarrow A$ 将输入关联到所有合法的方案上，关系 $R : B \leftarrow R$ 描述了方案之间的优劣关系。此时，优化问题可以被定义为 $min\ R \cdot \Lambda S$。

鉴于木态射的表达能力，关系 S 通常可以被定义成一个木态射。因此，程序演算中有关优化问题的工作基本只考虑形如 $min\ R \cdot \Lambda[\![S, T]\!]$ 的优化问题 [23, 44]。特别地，当木态射中的上态射是平凡的时候，即 $min\ R \cdot \Lambda(\!|S|\!)$，对应的优化问题又被称为*顺序决策过程* [19, 45, 46]。这一名字来自于 $(\!|S|\!)$ 的行为：它构造合法解的过程（即决策过程）严格按照输入的结构，并依次使用子结构的合法解构造更上一层结构的合法解。

例 37 *01 背包是一个经典的优化问题 [47]。给定背包容量和一系列的物品，该问题询问一个最优的将物品装入背包的方案，在装入背包的物品重量和不超过背包容量的情况下最大化这些物品的价值和。*

令 Item 是物品的类型，函数 $val, wt : Int \leftarrow Item$ 分别给出一个物品的价值与重量，保证价值与重量都非负。假定背包容量 w 是一个全局变量，物品列表为输入，则背包问题的所有合法方案可以被定义为下态射 $(\!|add|\!) : List\ Item \leftarrow List\ Item$，其中关系 add

的定义如下所示。

$$[]\ add\ (1) \quad xs\ add\ (2, x, xs) \quad wt\ x + (sum \cdot wt*)\ xs \leq w \Rightarrow ([x] +\!\!+ xs)\ add\ (2, x, xs)$$

其中第一条规则表示当物品列表为空时，唯一的合法方案是空列表。之后两条规则表示在考虑一个新的物品 x 时，如果从子结构的合法方案 xs 构造当前物品列表的合法方案。第二条规则对应不把新物品加入背包的情况，此时 xs 一定还是一个合法解。第三条规则对应把新物品加入背包的情况，这时需要保证当前物品的重量 $(wt\ x)$ 和已有物品重量 $((sum \cdot wt*)\ xs)$ 的和不超过背包容量 (w)，对应的合法方案为 $[x] +\!\!+ xs$。

利用关系 $(|add|)$，*01* 背包问题可以被定义为 $min\ R \cdot \Lambda(|add|)$，其中 $xs\ R\ xs'$ 当且仅当 xs 中物品的价值和比 xs' 大，即 $(sum \cdot val*)\ xs \geq (sum \cdot val*)\ xs'$。

举例来说，当背包容量 $w = 4$，物品列表为 $[x_1, x_2, x_3]$ 且它们的重量分别为 $3, 2, 1$，价值分别为 $3, 2, 2$ 时，函数 $(|\Lambda(add \cdot \mathsf{F} \in)|)$ 的输出为 $\{[], [x_1], [x_2], [x_3], [x_1, x_3], [x_2, x_3]\}$。此时，关系 $min\ R \cdot \Lambda(|add|)$ 将列表 $[x_1, x_2, x_3]$ 关联到 $[x_1, x_3]$，即背包问题的最优解。

2.4　算法层面的变换规则

在这一节中，本文介绍两条算法层面的变换规则。第一条利用范畴论概念下的程序模型将第三列表同态定理 (定理 3) 扩展到了任意多项式数据结构上 [16]。而第二条被称作稀疏定理 (thinning theorem) [19]，它可以为顺序决策过程推导高效的决策算法。

2.4.1　递归数据结构上的第三同态定理

Morihata 等人于 2009 年将第三列表同态定理 (定理 3) 推广到了任意递归数据结构上 [16]。为了方便，本文首先在如下定义的二叉树结构 *Tree* 上介绍这一推广的思想，然后再将其推广到任意的数据结构上。

$$Tree = Node\ Int\ Tree\ Tree \mid Leaf$$

2.4.1.1　二叉树上的第三同态定理

第三列表同态定理指出，列表上的一个函数如果可以同时被从左到右和从右到左地计算，那么它一定可以被高效地并行计算。若要将这一定理推广到树结构上，一个自然的形式是“一个函数如果可以同时被自顶向下地和自底向上地计算，那么它一定可以被高效地并行计算”。为此，需要先定义树上这三类计算的含义。Morihata 等人的工作使用树上的路径来定义这三类计算，自然地，自顶向下表示按照从上到下的顺序处理路径上的每一个节点，自底向上表示按照从下到上的顺序处理每一个节点，而并行计算则是从路径中间的某一个位置切开，并行地计算树的两个部分。

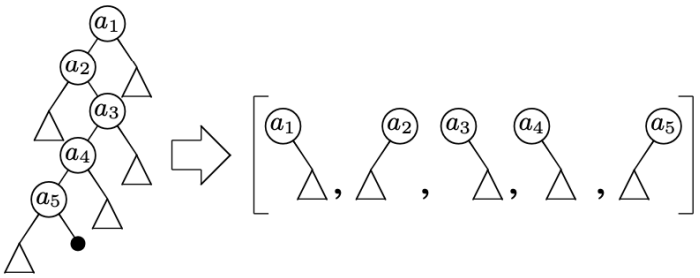

图 2.1　使用压缩子来表示二叉树结构的例子，图中的压缩子记录了从根到黑色叶子节点的路径。

这一工作使用了 *Huet* 压缩子 [48] 来表示树上的路径。对于二叉树，其压缩子 *Zipper* 是一个元素类型为 *Either* (*Int*, *Tree*) (*Int Tree*) 的列表，其中 *Either A B* 的定义如下。

$$Either\ A\ B = L\ A \mid R\ B$$

直观来说，压缩子代表了树上从根到叶子节点的一条路径。对于每一个元素，$L\ (n,t)$ 表示当前节点的值为 n，左子树为 t，而路径接着走入右子树；$R\ (n,t)$ 表示当前节点的值为 n，右子树为 t，而路径接着走入左子树。因此可以定义压缩子到二叉树的转换函数 $z2t : Tree \leftarrow Zipper$ 如下所示。

$$z2t\ [] \coloneqq Leaf \quad z2t\ ([L\ (n,l)] +\!\!+ x) \coloneqq Node\ n\ l\ (z2t\ x)$$
$$z2t\ ([R\ (n,r)] +\!\!+ x) \coloneqq Node\ n\ (z2t\ x)\ r$$

基于压缩子，可以定义路径计算的概念，即通过压缩子的形式来完成原本在树上进行的计算。对应地可以自然定义自顶向下的路径计算与自底向上的路径计算。

定义 4 (二叉树路径计算) 给定函数 $h : A \leftarrow Tree$，函数 $h' : A \leftarrow Zipper$ 是对应 h 的一个路径计算当且仅当 $h \cdot z2t = h'$。

定义 5 (二叉树自顶向下计算) 给定函数 $h : A \leftarrow Tree$ 及其路径计算 $h' : A \leftarrow Zipper$，如果存在 $\otimes : A \leftarrow (A \times (Either\ (Int, A)\ (Int, A)))$ 满足如下等式，则称 h' 是自顶向下的。

$$h'\ (x +\!\!+ [L\ (n,t)]) = h'\ x \otimes L\ (n, h\ t) \quad h'\ (x +\!\!+ [R\ (n,t)]) = h'\ x \otimes R\ (n, h\ t)$$

定义 6 (二叉树自底向上计算) 给定函数 $h : A \leftarrow Tree$ 及其路径计算 $h' : A \leftarrow Zipper$，如果存在 $\oplus : A \leftarrow ((Either\ (Int, A)\ (Int, A)) \times A)$ 满足如下等式，则称 h' 是自底向上的。

$$h'\ ([L\ (n,t)] +\!\!+ x) = L\ (n, h\ t) \oplus h'\ x \quad h'\ ([R\ (n,t)] +\!\!+ x) = R\ (n, h\ t) \oplus h'\ x$$

例 38 考虑如下函数 $sum : Int \leftarrow Tree$，它计算一棵树上所有点的权值和。

$$sum\ Leaf \coloneqq 0 \quad sum\ (Node\ n\ l\ r) = n + (sum\ l) + (sum\ r)$$

如下的函数 $sum' : Int \leftarrow Zipper$ 展示了一个对应的路径计算。

$$sum'\ [] \coloneqq 0 \quad sum'\ ([L\ (n,l)] +\!\!+ x) \coloneqq n + (sum\ l) + (sum'\ x)$$
$$sum'\ ([R\ (n,r)] +\!\!+ x) \coloneqq n + (sum\ r) + (sum'\ x)$$

函数 sum' 既是自顶向下的也是自底向上的。对应的操作符 $\otimes$ 和 $\oplus$ 如下所示。

$$sum_1 \otimes L\ (n, sum_2) = sum_1 + n + sum_2 \qquad sum_1 \otimes R\ (n, sum_2) = sum_1 + n + sum_2$$
$$L\ (n, sum_1) \oplus sum_2 = n + sum_1 + sum_2 \qquad R\ (n, sum_1) \oplus sum_2 = n + sum_1 + sum_2$$

自顶向下（自底向上）路径计算的定义在形式上与压缩子数组上的从左到右（从右到左）计算高度相似。对应地，压缩子上的分治计算的定式也与列表同态类似，它要求在任何位置切开压缩子的情况下都能够完成计算。

定义 7 (二叉树分解计算) 函数 $h : A \leftarrow Tree$ 可以被分解计算当且仅当存在函数 $\phi : A \leftarrow Either\ (Int, A)\ (Int, A)$ 和操作符 $\odot : A \leftarrow (A \times A)$，满足如下等式。

$$h'\ [] = id_{\odot} \quad h'\ [L\ (n,t)] = \phi\ (L\ (n, h\ t))$$
$$h'\ [R\ (n,t)] = \phi\ (R\ (n, h\ t)) \quad h'(x +\!\!+ y) = h'\ x \odot h'\ y$$

其中函数 h' 等于 $h \cdot z2t$，值 $id_{\odot}$ 表示操作符 $\odot$ 的单位元。

例 39 例 *38* 中的函数 *sum* 也是一个分解计算，其对应的函数 ϕ 和操作符 $\odot$ 如下所示。

$$\phi\ (L\ (n, sum_l)) \coloneqq n + sum_l \quad \phi\ (R\ (n, sum_r)) \coloneqq n + sum_r \quad sum_1 \odot sum_2 = sum_1 + sum_2$$

分解计算可以通过传统的树分治算法来高效地并行化 [49]。以边分治为例，在每一层计算中，边分治算法选择树的重心到其最大的子树的边的将树切开。在每一个点的度数都不超过一个常数时，这样可以保证每一个子问题中树的大小都被缩小了常数倍，从而保证递归深度是对数级别的。对于一个分解计算，边分治算法可以通过两步来实现：（1）选择一条包含需要切开的边的路径作为压缩子，然后（2）在对应的位置将压缩子切开并并行地递归到两边。Morihata 等人证明了在给定 p 个处理器的情况下，如果 ϕ 和 $\odot$ 都是常数时间的，则分解计算的结果可以在 $O(n/p + \log p)$ 的并行时间内求得，其中 n 为树的大小。

基于以上的定义，可以自然地得到二叉树上的第三同态定理。

定理 10 (二叉树上的第三同态定理) 给定函数 $h : A \leftarrow BTree$，如果它存在自顶向下且自底向上的路径计算 $h' : A \leftarrow Zipper$，则 h 一定是被如下二元组 $(\phi, \odot)$ 描述的分解计算，其中函数 g 是 h' 的右逆，即 $h' \cdot g \cdot h' = h'$。

$$\phi\ a = h'\ [a] \quad a \odot b = h'\ (g\ a +\!\!+ g\ b)$$

该定理的证明与列表上的第三同态定理（定理 3）几乎完全相同，因此本文将其略去。

2.4.1.2 到任意递归数据结构的推广

因为任意递归数据结构都可以被看作是一个树结构，所以上一小节中在二叉树上的讨论都可以自然地推广到任意递归数据结构上。在这一小节中，本文简单地对应的概念进行罗列。首先，McBride 在 2001 年通过引入导数函子的概念将压缩子推广到了任意的递归数据结构上 [50]。给定多项式函子 F，其导数函子 $\partial\mathsf{F}$ 的定义如下。

$$\partial(!A) :=!\emptyset \quad \partial\mathsf{I} :=!\{\bullet\} \quad \partial(\mathsf{F}\times\mathsf{G}) := \mathsf{F}\times\partial\mathsf{G}+\partial\mathsf{F}\times\mathsf{G} \quad \partial(\mathsf{F}+\mathsf{G}) = \partial\mathsf{F}+\partial\mathsf{G}$$

其中 $\bullet$ 是一个特殊符号。直观来说，$\partial\mathsf{F}$ 表示在 F 对应的数据结构（记作 $\mu\mathsf{F}$）上向下走一步的方案，而 $\bullet$ 表示走到的位置。下面是对上面四条定义的直观解释。

- 因为路径不能走到常函子，所以常函子的导数为空。
- 因为路径中的每一步都是走到 F 中的一个递归处（即单位函子 I），所以单位函子的导数就是表示目标位置的常函子 $!\{\bullet\}$。
- 因为函子的直积代表了两个并列的孩子节点，所以其导数对应着两种情况下路径，分别是走到函子 G 对应的子树（即 $\mathsf{F}\times\partial\mathsf{G}$）和走到 F 对应的子树（即 $\partial\mathsf{F}\times\mathsf{G}$）。
- 因为函子的直和代表了两种独立的结构，所以其导数也对应着两种路径的直和。

根据上述定义，可以得到函子 F 的压缩子（记作 Z_F）的定义为 $[\partial\mathsf{F}(\mu\mathsf{F})]$。其对应的转换函数 $\lhd_\mathsf{F}$: $\mu\mathsf{F} \leftarrow (Z_\mathsf{F}\times\mu\mathsf{F})$ 的定义如下所示，其中 $z \lhd_\mathsf{F} t$ 返回给定压缩子 z 且压缩子 z 对应的路径终点的数据结构为 t 时的完整的数据结构。

$$z \lhd_\mathsf{F} t := (\lessdot_\mathsf{F}) \leftthreetimes_t z \quad \lessdot_\mathsf{F} : \mu\mathsf{F} \leftarrow (Z_\mathsf{F}\times\mu\mathsf{F})$$

$$a \lessdot_{!A} t := a \quad \bullet \lessdot_\mathsf{I} t := t$$

$$(1,(a,b)) \lessdot_{\mathsf{F}\times\mathsf{G}} t := (a, b \lessdot_\mathsf{G} t) \quad (2,(a,b)) \lessdot_{\mathsf{F}\times\mathsf{G}} t := (a \lessdot_\mathsf{F} t, b)$$

$$(1,a) \lessdot_{\mathsf{F}+\mathsf{G}} t := (1, a \lessdot_\mathsf{F} t) \quad (2,b) \lessdot_{\mathsf{F}+\mathsf{G}} t := (2, b \lessdot_\mathsf{G} t)$$

例 40 *考虑对应第 2.4.1.1 节中的二叉树结构对应的函子 $\mathsf{F}_T = !Int\times(\mathsf{I}\times\mathsf{I})+!1$。根据定义，其对应的导数函子 $\partial\mathsf{F}_T$ 为如下所示。*

$$\partial\mathsf{F}_T = (!\emptyset\times(\mathsf{I}\times\mathsf{I})+!Int\times(\mathsf{I}\times!\{\bullet\}+!\{\bullet\}\times\mathsf{I}))+!\emptyset$$

因为 $!\emptyset$ 以及包含它的直积均对应空集，所以上述函子可以被化简为 $\partial\mathsf{F}'_T = !Int\times(\mathsf{I}\times!\{\bullet\}+\{\bullet\}\times\mathsf{I})$，而对应的压缩子为 $[\partial\mathsf{F}'_T(Tree)] = [!Int\times(Tree\times!\{\bullet\}+\{\bullet\}\times Tree)]$。不难发现这一定义与第 2.4.1.1 中的压缩子是等价的。

根据上属于定义，可以自然地推广路径计算、自顶向下计算、自底向上计算、分解计算以及第三同态定理的内容。

定义 8 (路径计算) 给定函数 $h : A \leftarrow \mu\mathsf{F}$，函数 $h' : A \leftarrow Z_{\mathsf{F}}$ 是一个路径计算当且仅当存在操作符 $\ominus : A \leftarrow (A \times \mu\mathsf{F})$ 满足如下等式。

$$h\ (z \lhd_{\mathsf{F}} t) = h'\ z \ominus t$$

定义 9 (向下计算) 给定函数 $h : A \leftarrow \mu\mathsf{F}$ 及其路径计算 $h' : A \leftarrow Z_{\mathsf{F}}$，如果存在操作符 $\otimes : A \leftarrow (A \times \partial\mathsf{F}A)$ 和值 $e : A$ 满足如下等式，则称 h' 是自顶向下的。

$$h' = \otimes \not\rightarrow_e \cdot (\partial\mathsf{F}h)*$$

定义 10 (向上计算) 给定函数 $h : A \leftarrow \mu\mathsf{F}$ 及其路径计算 $h' : A \leftarrow Z_{\mathsf{F}}$，如果存在操作符 $\oplus : A \leftarrow (\partial\mathsf{F}A \times A)$ 和值 $e : A$ 满足如下等式，则称 h' 是自底向上的。

$$h' = \oplus \not\leftarrow_e \cdot (\partial\mathsf{F}h)*$$

定义 11 (分解计算) 函数 $h : A \leftarrow Tree$ 可以被分解计算当且仅当它可以通过压缩子上的一个列表同态计算得到，即存在三元组函数 $\phi : A \leftarrow \partial\mathsf{F}A$ 和操作符 $\odot : A \leftarrow (A \times A), \ominus : A \leftarrow (A \times \mu\mathsf{F})$ 满足如下等式。

$$h\ (z \lhd t) = (hom\ (\odot)\ (\phi \cdot \partial\mathsf{F}h)\ z) \ominus t$$

定理 11 (递归数据结构上的列表同态定理) 给定多项式函子 F 和函数 $h : A \leftarrow \mu\mathsf{F}$，如果它存在自顶向下且自底向上的对应操作符 $\ominus$ 的路径计算 $h' : A \leftarrow Z_{\mathsf{F}}$，则 h 一定是被三元组 $(\phi, \odot, \ominus)$ 描述的分解计算，其中函数 $\phi, \odot$ 的定义如下，函数 g 是 h' 的右逆，即 $h' \cdot g \cdot h' = h'$。

$$\phi\ (\partial\mathsf{F}h\ a) = h'\ [a] \quad a \odot b = h'\ (g\ a +\!\!+ g\ b)$$

2.4.2　稀疏定理

de Moor 于 2005 年提出了稀疏定理，用于为顺序决策过程推导高效的决策算法 [19]。回顾第 2.3.3.2 节中的定义，一个顺序决策过程是形式为 $min\ R \cdot (\!|S|\!)$ 的优化问题。通过下态射 $(\!|S|\!)$，它严格按照输入的结构顺序地构造合法解，并通过关系 S 与子结构的合法解来构造更上一层结构的合法解。稀疏定理来源于一个自然的剪枝思路。在通过子结构的合法解的过程构造上一层结构的合法解时，并不是所有合法解都是有用的。对于一些显然不优的合法解，可以在子结构层面就将其删去，从而节约后续的构造时间。

为了描述删去显然不优的合法解这一过程，de Moor 引入了稀疏化算子 *thin*。给定关系 $Q : A \leftarrow A$，稀疏化关系 $thin\ Q : \mathsf{P}A \leftarrow \mathsf{P}A$ 的定义如下所示。

$$x(thin\ Q)\,y \iff x \subseteq y \wedge \forall b \in y, \exists a \in x, aQb$$

直观来说，当 A 表示优化问题中解的类型且关系 Q 描述了解之间的优劣关系（即 aQb 表示解 a 的效果覆盖了 b）时，关系 $thin\ Q$ 将一个解的集合关系到它的一些效果上等价的子集。具体来说，令 y 是一个解的集合，x 是它的一个满足 $x(thin\ Q)\,y$ 的子集，那么条件 $\forall b \in y, \exists a \in x, aQb$ 要求 y 中任何一个解的效果都被 x 中的某一个解覆盖了。因此，在构造合法解的过程中，可以直接使用解集 x 取代解集 y。

基于稀疏化算子 $thin\ Q$ 的定义，下面展示了稀疏定理的内容。

定理 12 (稀疏定理) 如果 $Q \subseteq R$ 且 $S \cdot \mathsf{F}Q^\circ \subseteq Q^\circ \cdot S$，则如下公式成立。

$$min\ R \cdot (\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F} \in)|\!) \subseteq min\ R \cdot \Lambda(\!|S|\!) \tag{2.3}$$

该定理的证明依依赖于一些范畴 **Rel** 上的变换规则，因此本文将其略去，只介绍其直观意义。首先，公式 2.3 的右侧是顺序决策过程的定义，左侧给出了一个更小的关系，它表示在只关心其中一个最优解的情况下，可以使用左侧的关系来代替右侧的定义。

接着，考虑左侧关系的含义。$(\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F} \in)|\!)$ 是一个从输入到方案集合的关系。与 $(\!|S|\!)$ 类似，该关系严格按照输入的结构顺序构造解的集合。每一步的决策过程 $thin\ Q \cdot \Lambda(S \cdot \mathsf{F} \in)$ 将子结构上的解集合（类型为 $\mathsf{FP}A$）关联到上一层结构上的解集合（类型为 $\mathsf{P}A$），其中 A 为解的类型。这一决策过程可以被划分为两个阶段。在第一阶段中，函数 $\Lambda(S \cdot \mathsf{F} \in)$ 输出一个包含所有可以从子结构上的解构造得到的解的集合。接着，在第二阶段中，关系 $thin\ Q$ 将这个解的集合关联到所有与之等效的子集，表示删去那些在关系 Q 下不优的解。

最后，考虑定理的两个条件。首先，$Q \subseteq R$ 蕴含 $min\ R = min\ R \cdot thin\ Q$，即对于任意解的集合，$thin\ Q$ 都不会将其中（在 R 意义下的）最优解删去，从而保证左侧程序在通过 $thin\ Q$ 剪枝的过程中，永远不会将最优解删去。

接着，考虑条件 $S \cdot \mathsf{F}Q^\circ \subseteq Q^\circ \cdot S$。该公式的两侧均为 $\mathsf{F}A$ 到 A 的关系，即将子结构上的解关联到上层结构的解。下面展示了这两个关系的含义。

$$a(S \cdot \mathsf{F}Q^\circ)x \iff \exists y, (x\mathsf{F}Qy \wedge aSy)$$
$$a(Q^\circ \cdot S)x \iff \exists b, (bSx \wedge bQa)$$

因此，第二个条件等价于如下的公式。

$$\forall x, y \in \mathsf{F}A, \forall a \in A, (x\mathsf{F}Qy \wedge aSy \to \exists b \in A, bSx \wedge bQa) \tag{2.4}$$

它要求对于任意两个子结构的解 x, y，如果 x 在 Q 中优于 y (即 $x\mathsf{F}Qy$)，则对于任意 y 能产生的上一层次解 a，都存在 x 能产生的上一层次解 b，使得 b 在 Q 中优于 a（即 bQa）。根据这一解释，第二个条件实际上要求了 Q 定义的优劣关系关于顺序的构造过程是一致的，从而保证了每一步构造后都使用 $thin\ Q$ 剪枝这一优化的正确性。

例 41 考虑使用稀疏定理（定理 *12*）优化例 *37* 中定义的 *01* 背包问题。定义一个物品序列优于另一个当且仅当前者使用的重量总和更小且获得的价值和更大，即定义物品序列上的关系 Q 如下所示。

$$xsQxs' \iff \Big((sum \cdot wt*)\ xs \leq (sum \cdot wt*)\ xs'\Big) \wedge \Big((sum \cdot val*)\ xs \geq (sum \cdot val*)\ xs'\Big)$$

根据定义，显然有 $Q \subseteq R$。而对于第二个条件 $S \cdot \mathsf{F}Q^{\circ} \subseteq Q^{\circ} \cdot S$，考虑其等价形式公式 *2.4*。对于任意物品 x 以及任意两个物品序列 xs 和 xs' 满足 $xsQxs'$，分三种情况讨论。

- $(sum \cdot wt*)\ xs + (wt\ x) > w$，即 x 无法被加入方案 xs 中。因为 xs' 中物品的重量和一定小于等于 xs'，所以 x 同样无法被加入方案 xs' 中。此时，xs 和 xs' 能够产生的方案集合分别为 $\{xs\}$ 和 $\{xs'\}$。因为 $xsQxs'$，所以此时公式 *2.4* 成立。
- x 可以被加入方案 xs 中但是不能被加入 xs' 中。此时 xs 和 xs' 可以产生的方案集合分别为 $\{[x] +\!\!+ xs, xs\}$ 和 $\{xs'\}$。同样因为 $xsQxs'$，所以公式 *2.4* 成立。
- x 即可以被加入方案 xs 中也可以被加入方案 xs' 中，此时 xs 和 xs' 可以产生的方案集合分别为 $\{[x] +\!\!+ xs, xs\}$ 和 $\{[x] +\!\!+ xs', xs'\}$。不难验证 $([x] +\!\!+ xs)Q([x] +\!\!+ xs')$ 且 $xsQxs'$，所以公式 *2.4* 成立。

综上，Q 符合稀疏定理的使用条件。基于该关系使用稀疏定理后，可以得到如下关系。

$$min\ R \cdot (\!|thin\ Q \cdot \Lambda(add \cdot \mathsf{F} \in)|\!) \tag{2.5}$$

考虑另一个物品序列上的关系 $Q' = R$，即 $xsQxs'$ 当且仅当 xs 中物品的价值而和大于等于 xs'。尽管 Q' 满足稀疏定理的第一个条件，即 $Q' \subseteq R$，但是它并不满足第二个条件，即公式 *2.4*。

考虑物品序列 $xs = [x_1], xs' = []$，其中 x_1 的重量和价值均为 1。根据 Q' 的定义，显然有 $xsQ'xs'$。现在，假设背包容量是 2 且新的物品 x 的重量和价值均为 2。此时，xs 只能产生方案 $[x_1]$，但是 xs' 可以产生方案 $[]$ 和 $[x]$。因为在关系 Q 中，方案 $[x_1]$ 并不比 $[x]$ 优，所以 Q' 不满足公式 *2.4*，从而无法基于 Q' 对 *01* 背包问题使用稀疏定理。

因为稀疏定理的结果是一个关系，所以要在实践中使用稀疏定理得到的结果，就需要把这一关系实现成一个函数，即找到函数 $f \in min\ R \cdot (\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F} \in)|\!)$。此时，只需要找到两个函数 $m_R \in min\ R$ 和 $t_Q \in thin\ Q$，则函数 $m_R \cdot (\!|t_Q \cdot \Lambda(S \cdot \mathsf{F} \in)|\!)$ 就是一个满足条件的程序。

因为在优化问题中，R 一般是一个全序，所以 m_R 可以直接通过遍历整个解集合来实现。而对 t_Q 的实现并没有标准做法，通常需要根据问题在 t_Q 的时间复杂度与返回集合的大小之间进行取舍。在极端情况下，t_Q 可以不进行任何剪枝，即 $t_Q = id$，此时 t_Q 的时间复杂度与输入规模一致，但是它返回的集合大小与输入相同，起不到优化的

效果。在另一种极端情况下，t_Q 可以返回一个极小的集合，但是此时它的时间复杂度通常会达到输入规模的平方量级。

例 42 对于例 *41* 中得到的优化结果（公式 *2.5*），函数 t_Q 可以通过如下方式实现。

- 对每一个解计算其中所有物品的重量和以及价值和。
- 对于每一个种可能的重量和，都只保留价值和最大的一个方案并删除其他方案。

假设背包容量为 w，物品数量为 n，则 t_Q 在给定 k 个解的情况时的时间复杂度为 $O(nk)$ 并且返回的解的数量为 $O(w)$。根据该实现，可以得到背包问题的一个 $O(n^2w)$ 时间复杂度的高效程序。在假设 w 和 n 同一级别的情况下，该程序相比于背包问题的穷举算法在时间复杂度上有着指数级的提升。

在为优化问题推导高效算法时，稀疏定理存在两个不足之处。一方面，它的使用范围有限，只能支持顺序决策过程。另一方面，仅使用稀疏定理获得的程序通常还有继续优化的空间。以背包问题为例，虽然通过稀疏定理可以得到一个时间复杂度为 $O(n^2w)$ 的高效算法，但是距离该问题上经典的 $O(nw)$ 时间的动态规划算法还存在一定的差距。针对这两个不足之处，后续的研究对稀疏定理提出了一系列的改进，包括将该定理推广到更一般的优化问题 [51]，将该定理与其他推导技术结合从而进一步提升结果程序的时间复杂度 [21] 等。

2.5 程序演算的自动化系统

在之前的几个节中，本文介绍了程序演算领域中提出的一些具有代表性的程序变换规则。鉴于这些规则抽象性与复杂性，若要在实践中使用这些规则来优化程序，那么这些规则的应用过程都应当被集成进一个自动的系统中。在这一小节中，本文对程序演算领域中这一方面的工作进行简单的介绍。

2.5.1 高阶模式匹配问题

程序变换规则通常使用模式串来定义它所适用的程序以及所需要的条件，并使用一个与模式串中的变量相关的程序来指明程序变换结果。因此，应用程序变换规则的过程通常可以被转化为一个（或者多个）模式匹配问题。下面展示了这样的一个例子。

例 43 在这个例子中，本文以融合定理（定理 *5*）在列表上的特例为例，展示程序变换规则的应用于模式匹配问题的关系。为了更贴合实际，在这个例子中，本文采用类似 *Haskell* 语言的语法来描述程序与规则，而不是在前文中使用的形式化符号。此时，该规则的内容如下所示。

$$\frac{a \otimes f\ r = f\ (a \oplus r)}{f\ .foldr\ (\oplus)\ e = foldr\ (\otimes)\ (f\ e)}$$

现在考虑将这一规则应用到如下的计算列表中所有数的平方和的程序中。

$$\begin{aligned} sumsq &= sum\ .foldr\ (\lambda a.\lambda r.(a \times a) : r)\ [] \\ where \quad sum\ [] &= 0 \\ sum\ (a : x) &= a + sum\ x \end{aligned}$$

- 通过使用模式串 $f\ .foldr\ (\oplus)\ e$ 匹配这一程序，可以得到匹配结果 $f = sum, \oplus = \lambda a.\lambda r.(a \times a) : r$ 且 $e = []$。
- 根据应用条件，需要得到一个满足 $a \otimes f\ r = f\ (a \oplus r)$ 的操作符 $\oplus$。代入第一次匹配的结果，可以得到一个模式匹配问题，其中模式串为 $\lambda a.\lambda r.a \otimes sum\ r$，程序为 $\lambda a.\lambda r.sum\ ((a \times a) : r)$。将 sum 的定义展开一层，可以得到程序 $a \times a + sum\ r$，从而得到匹配结果 $\otimes = \lambda a.\lambda b.a \times a + b$。
- 将通过模式匹配得到的 $\otimes, f, e$ 代入结果串 $foldr\ (\otimes)\ (f\ e)$，即可得到变换结果。

在本节的剩余部分，本文在 λ-演算这一模型下，介绍高阶模式匹配问题及其特例。在该模型中，表达式由如下文法定义，其中四条规则分别表示常量、变量、函数应用和 λ 抽象（即函数的引入）。

$$T ::= c \mid v \mid T\ T \mid \lambda x.T$$

给定表示模板串的表达式 P 和表示待匹配语句的封闭表达式 T（即 T 不包含自由变量），高阶模板匹配问题的目标是找到一个对 P 中自由变量的一个替换 ϕ，使得将 ϕ 作用到 P 后得到的表达式与 T 等价，即满足如下等式。

$$\phi\ P =_{\alpha\beta\eta} T$$

其中 $=_{\alpha\beta\eta}$ 表示语法上的等价关系对 α, β, η 三种变换取模，即等号两侧的表达式可以在有限次 α, β, η 变换后变得完全一致。在三种变换的定义如下所示。

- α 变换对 λ 抽象的变量名产生修改，即 $\lambda x.M[x] \to_\alpha \lambda y.M[y]$。
- β 变换对函数应用进行展开，即 $(\lambda x.M[x])\ E \to_\beta M[E]$。
- η 变换去除最外层的 λ 抽象，即 $\lambda x.M\ x \leftrightarrow_\eta M$，要求 x 不是 M 中的自由变量。从效果上来说，η 变换考虑函数的外延。在引入该变换后，如果两个函数在任何相同参数下的结果都相同，则它们会被视为等价。

例 44 考虑高阶模式匹配问题 $\phi\ (f\ x) =_{\alpha\beta\eta} 0$。下面展示了该问题的一些解。

- $f ::= (\lambda x.x), x ::= 0$ 是该问题的一个解，因为 $(\lambda x.x)\ 0 \to_\beta 0$。

- $f := (\lambda g.g\ 0), x := (\lambda a.a)$ 是该问题的一个解，对应的推理过程如下所示。

$$(\lambda g.g\ 0)\ (\lambda a.a) \rightarrow_\beta (\lambda a.a)\ 0 \rightarrow_\beta 0$$

- 定义 $t_1 = 0, t_i = g\ t_{i-1}$，则通过简单的归纳法可以证明，对于任意的 $n \geq 1$，$f := (\lambda g.t_n), x := (\lambda a.a)$ 都是该问题的一个解。

从例 44 中可以看出，高阶模式匹配问题是困难的。即使对于很简单的模板串与待匹配语句，都可能有无穷多种匹配方案，且合法方案的规模可以任意地复杂。实际上，在只允许 α, β 变换时，高阶模式匹配问题是不可判定的 [52]。尽管在引入 η 变换后该问题的不可判定性还没有被得到证明，但是它通常被相信是困难的。

为了在实际中进行高阶模式匹配，Heut 和 Lang 引入了阶的概念来将匹配结果限制在一个更贴合实际程序的子集中 [53]。阶的限制定义在简单类型 λ 演算基础上 [54]。给定基础类型集合 T_0，简单类型 λ 演算的类型集合 T 如下所示。

$$\alpha \in T_0 \Rightarrow \alpha \in T \quad \alpha, \beta \in T \Rightarrow \alpha \rightarrow \beta \in T$$

其中基础类型 $\alpha \in T_0$ 表示了某种类型的值（例如整数），而类型 $\alpha \rightarrow \beta$ 则表示了一个以 α 为输入，β 为输出的函数。类型的阶 ord 是一个类型到整数的函数，它的定义如下所示。

$$\forall \alpha \in T_0, ord(\alpha) = 1 \quad ord(\alpha \rightarrow \beta) = \max(ord(\alpha) + 1, ord(\beta))$$

直观来说，在函数柯里化的概念下，函数的阶数反映了函数在其输入中嵌套的层数：一个具有 n 阶类型的函数只能以 $n-1$ 阶类型的函数为输入，且至少使用了一个 $n-1$ 阶类型的函数。而一个具有二阶类型的函数只能以常量而不能以函数为输入。

在 n 阶的模式匹配问题中，替换 ϕ 只被允许将模板串 P 中的自由变量替换为简单类型 λ 演算中至多 n 阶的表达式。在加入阶的限制后，高阶模式匹配问题的难度有了显著的下降。目前已知二阶模式匹配问题是 NP-完全的 [55] 且合法替换数量一定有限 [53]，三阶和四阶的模式匹配问题都是可判定的 [56, 57]。

2.5.2 高阶模式匹配算法

尽管对于一般性的二到四阶的模式匹配问题，理论上的算法已经成熟，但是在程序变换系统这一实际应用中，根据使用场景的不同通常存在不同方面的取舍问题，包括匹配算法的完备性、匹配算法的效率、匹配结果的确定性以及匹配系统的表达能力等。根据不同的取舍，产生了一系列细分的模式匹配问题以及对应的算法。

因为三阶模式匹配问题可能存在无穷多的合法解，所以为了保证变换系统的完备性，Heut 和 Lang 在 1978 年提出在程序变换时只考虑二阶的模式匹配，并给出了一个

求出所有二阶解的高效算法 [53]。但是随着涉及三阶函数的程序变换规则的提出，例如第 2.3.2.3 中介绍的酸雨定理（定理 8 和 9），只考虑二阶模式匹配的局限性渐渐被暴露了出来。于是，de Moor 和 Sittampalam 于 1999 年提出了一个支持更高阶模式匹配的匹配算法 [58]。该算法保证可以找到所有不超过二阶的合法替换，同时可能可以找到一部分更高阶的合法替换，但是并不保证其完备性。这一算法也被集成进了由牛津大学发布的程序变换系统 MAG 中 [59]。下面这个例子展示了该高阶模式匹配算法在表达能力方面对程序变换系统的提升。

例 45 *考虑按照如下方式定义的程序 rev，它可以将给定的列表翻转。*

$$rev = foldr\ (\lambda x.\lambda xs \mathbin{+\!\!+} [x])\ []$$

因为 $xs \mathbin{+\!\!+} [x]$ 的时间复杂度是线性的，所以 rev 的时间复杂度是关于输入长度平方的。现在考虑使用例 43 中的变换规则优化程序 $rev'\ xs\ ys = rev\ xs \mathbin{+\!\!+} ys$。首先，使用模板 $f\ .foldr\ (\oplus)\ e$ 匹配该程序，可以得到如下的二阶替换。

$$f = \lambda xs.\lambda ys.xs \mathbin{+\!\!+} ys \quad \oplus = \lambda x.\lambda xs.xs \mathbin{+\!\!+} [x] \quad e = []$$

接着，为了得到操作符 $\otimes$，需要求解如下的化简后的匹配问题。

$$P = \lambda a.\lambda r.a \otimes (\lambda ys.r \mathbin{+\!\!+} ys) \quad T = \lambda a.\lambda r.\lambda ys.r \mathbin{+\!\!+} [a] \mathbin{+\!\!+} ys$$

此时，de Moor 和 Sittampalam 的算法可以得到三阶替换 $\otimes = \lambda x.\lambda g.\lambda ys.g\ ([x] \mathbin{+\!\!+} ys)$，从而得到 rev'（即 rev）的一个线性时间复杂度的实现。

在另一个方面，与一阶模式匹配问题相比，通用的二阶模式匹配一方面需要更多的时间开销，另一方面存在多个可行解，从而为程序变换系统引入了不确定性。为了改进这一点，Tokoyama 等人于 2003 年提出了确定性二阶模板的概念 [60]。确定性二阶模板是一类具有特殊形式的模板，它们保证了在对应的二阶模板匹配问题中，至多只有一个合法解。具体来说，确定性二阶模板在要求每一个自由变量都至多二阶的基础上，还对使用每一个二阶自由变量时的参数作出了两点要求：(1) 每一个参数都不是封闭的，且参数中不包含模板层面的自由变量，(2) 不同参数之间不存在子表达式关系。下面展示了两个确定性二阶模板的例子。

例 46 *被例 43 中的变换规则包含的模板 $\lambda a.\lambda r.a \otimes f\ r$ 是一个确定性二阶模板，因为 f 的参数 r 本身不是封闭的且 r 在不是模板的自由变量。*

更复杂地，模板 $\lambda w.\lambda x.if\ p\ x\ then\ q\ x\ else\ r\ (Car\ x)\ (w\ (Cdr\ x))$ 也是一个确定性二阶模板，容易验证自由变量 p 的参数 x，q 的参数 x，还有 r 的参数 $\{(Car\ x), (w\ (Cdr\ x))\}$ 都满足确定性二阶模板定义中的条件。

Tokoyama 等人为确定性二阶模板匹配问题提出了一个效率接近一阶模板匹配算法的高效匹配算法。在假定每一个二阶变量的参数数量都是常数的时候，该算法的时间复杂度关于输入规模是线性的。这一算法也被集成进了东京大学发布的程序变换系统 Yicho 中 [61]。

除了以上两个通用系统以外，程序演算领域的研究人员还提出了针对特定规则的变换系统，例如针对酸雨定理的变换系统 Hylo [62] 和针对稀疏定理的一个特例的变换系统 [63]。因为在这些系统中程序变换规则是已知的，所以设计针对性的模式匹配算法的难度就大大降低了。

同时，因为高阶模式匹配问题只在语法层面求解程序变换规则所需的函数，所以它并不能应对一些依赖语义信息的复杂场景①。对于一些复杂的程序变换规则，尤其是算法层面的程序变换规则，已经有工作尝试使用程序合成技术来直接合成程序变换规则所需的函数 [51, 64, 65]。由于篇幅原因，本文便不再对这一方面继续展开描述了。

① 尽管总是可以将语义信息编码进 λ 演算，但是这样会极大地提高问题的规模与解的阶数。

参考文献

参考文献

[1] Martin Ward. *Proving program refinements and transformations* [phdthesis], **1989**. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.306712.

[2] Rod M. Burstall and John Darlington. "*A Transformation System for Developing Recursive Programs*". *J. ACM*, **1977**, *24*(1): 44–67. https://doi.org/10.1145/321992.321996.

[3] Laurent Kott. *Unfold/fold program transformations* [phdthesis], **1982**.

[4] David Sands. "*Total Correctness by Local Improvement in the Transformation of Functional Programs*". *ACM Trans. Program. Lang. Syst.* **1996**, *18*(2): 175–234. https://doi.org/10.1145/227699.227716.

[5] Philippa Gardner and John C. Shepherdson; ed. by Jean-Louis Lassez and Gordon D. Plotkin. "*Unfold/Fold Transformations of Logic Programs*". In: *Computational Logic - Essays in Honor of Alan Robinson*. The MIT Press, **1991**: 565–583.

[6] Taisuke Sato. "*Equivalence-Preserving First-Order Unfold/Fold Transformation Systems*". *Theor. Comput. Sci.* **1992**, *105*(1): 57–84. https://doi.org/10.1016/0304-3975(92)90287-P.

[7] Alberto Pettorossi, Maurizio Proietti and Sophie Renault; ed. by Peter Lee, Fritz Henglein and Neil D. Jones. "*Reducing Nondeterminism while Specializing Logic Programs*". In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*. ACM Press, **1997**: 414–427. https://doi.org/10.1145/263699.263759.

[8] Abhik Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan *et al.* "*An unfold/fold transformation framework for definite logic programs*". *ACM Trans. Program. Lang. Syst.* **2004**, *26*(3): 464–509. https://doi.org/10.1145/982158.982160.

[9] Michael J. Maher; ed. by C. E. Veni Madhavan. "*A Transformation System for Deductive Database Modules with Perfect Model Semantics*". In: *Foundations of Software Technology and Theoretical Computer Science, Ninth Conference, Bangalore, India, December 19-21, 1989, Proceedings*. Springer, **1989**: 89–98. https://doi.org/10.1007/3-540-52048-1%5C_35.

[10] Sandro Etalle and Maurizio Gabbrielli. "*Transformations of CLP Modules*". *Theor. Comput. Sci.* **1996**, *166*(1&2): 101–146. https://doi.org/10.1016/0304-3975(95)00148-4.

[11] Nacéra Bensaou and Irène Guessarian; ed. by Patrice Enjalbert, Ernst W. Mayr and Klaus W. Wagner. "*Transforming Constraint Logic Programs*". In: *STACS 94, 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen, France, February 24-26, 1994, Proceedings*. Springer, **1994**: 33–46. https://doi.org/10.1007/3-540-57785-8%5C_129.

[12] Sandro Etalle, Maurizio Gabbrielli and Maria Chiara Meo. "*Transformations of CCP programs*". *ACM Trans. Program. Lang. Syst.* **2001**, *23*(3): 304–395. https://doi.org/10.1145/503502.503504.

[13] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi *et al.*; ed. by Simon L. Peyton Jones, Mads Tofte and A. Michael Berman. "*Tupling Calculation Eliminates Multiple Data Traversals*". In: *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*. ACM, **1997**: 164–175. https://doi.org/10.1145/258948.258964.

[14] Maarten M Fokkinga. "*Tupling and mutumorphisms*". *Squiggolist*, **1989**, *1*(4).

[15] Murray Cole. "*Parallel Programming with List Homomorphisms*". *Parallel Process. Lett.* **1995**, *5*: 191–203. https://doi.org/10.1142/S0129626495000175.

[16] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu *et al.*; ed. by Zhong Shao and Benjamin C. Pierce. "*The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer*". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. ACM, **2009**: 177–185. https://doi.org/10.1145/1480881.1480905.

[17] Richard S. Bird and Oege de Moor; ed. by Bernhard Möller, Helmuth Partsch and Stephen A. Schuman. "*From Dynamic Programming to Greedy Algorithms*". In: *Formal Program Development - IFIP TC2/WG 2.1 State-of-the-Art Report*. Springer, **1993**: 43–61. https://doi.org/10.1007/3-540-57499-9%5C_16.

[18] Paul Helman. *A theory of greedy structures based on k-ary dominance relations*. Department of Computer Science, College of Engineering, University of New Mexico, **1989**.

[19] Oege de Moor; ed. by Manuel V. Hermenegildo and S. Doaitse Swierstra. "*A Generic Program for Sequential Decision Processes*". In: *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings*. Springer, **1995**: 1–23. https://doi.org/10.1007/BFb0026809.

[20] Akimasa Morihata. "*A Short Cut to Optimal Sequences*". *New Gener. Comput.* **2011**, *29*(1): 31–59. https://doi.org/10.1007/s00354-010-0098-4.

[21] Akimasa Morihata, Masato Koishi and Atsushi Ohori; ed. by Michael Codish and Eijiro Sumii. "*Dynamic Programming via Thinning and Incrementalization*". In: *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*. Springer, **2014**: 186–202. https://doi.org/10.1007/978-3-319-07151-0%5C_12.

[22] Shin-Cheng Mu; ed. by Robert Glück and Oege de Moor. "*Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths*". In: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*. ACM, **2008**: 31–39. https://doi.org/10.1145/1328408.1328414.

[23] Richard S. Bird and Oege de Moor; ed. by Manfred Broy. "*The algebra of programming*". In: *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, **1996**: 167–203.

[24] Richard S Bird and LGLT Meertens. "*Two exercises found in a book on algorithmics*". *Program Specification and Transformation*, **1987**: 451–458.

[25] Jeffrey Dean and Sanjay Ghemawat; ed. by Eric A. Brewer and Peter Chen. "*MapReduce: Simplified Data Processing on Large Clusters*". In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. USENIX Association, **2004**: 137–150. http://www.usenix.org/events/osdi04/tech/dean.html.

[26] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto *et al.*; ed. by Xiaohua Jia. "*A library of constructive skeletons for sequential style of parallel programming*". In: *Proceedings of the 1st International Conference on Scalable Information Systems, Infoscale 2006, Hong Kong, May 30-June 1, 2006*. ACM, **2006**: 13. https://doi.org/10.1145/1146847.1146860.

[27] Ed. by Fethi A. Rabhi and Sergei Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, **2003**. https://doi.org/10.1007/978-1-4471-0097-3.

[28] Guy L. Steele Jr.; ed. by Masami Hagiya and Philip Wadler. "*Parallel Programming and Parallel Abstractions in Fortress*". In: *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*. Springer, **2006**: 1. https://doi.org/10.1007/11737414%5C_1.

[29] Richard S Bird. "*An introduction to the theory of lists*". In: *Logic of programming and calculi of discrete design*. Springer, **1987**: 5–42.

[30] Jeremy Gibbons. "*Functional pearls: The third homomorphism theorem*". *Journal of Functional Programming*, **1996**, *6*(4): 657–665.

[31] Richard S. Bird. "*Algebraic Identities for Program Calculation*". *Comput. J.* **1989**, *32*(2): 122–126. https://doi.org/10.1093/comjnl/32.2.122.

[32] Jon Louis Bentley. *Programming pearls*. Addison-Wesley, **1986**.

[33] Tatsuya Hagino. *Category theoretic approach to data types* [phdthesis], **1987**.

[34] Erik Meijer, Maarten M. Fokkinga and Ross Paterson; ed. by John Hughes. "*Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*". In: *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*. Springer, **1991**: 124–144. https://doi.org/10.1007/3540543961%5C_7.

[35] Maarten M. Fokkinga. *Law and order in algorithmics*. Univ. Twente, **1992**.

[36] Tim Sheard and Leonidas Fegaras; ed. by John Williams. "*A Fold for All Seasons*". In: *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*. ACM, **1993**: 233–242. https://doi.org/10.1145/165180.165216.

[37] Akihiko Takano and Erik Meijer. "*Shortcut deforestation in calculational form*". In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, **1995**: 306–313.

[38] Zhenjiang Hu, Hideya Iwasaki and Masato Takeichi. "*Deriving structural hylomorphisms from recursive definitions*". *ACM Sigplan Notices*, **1996**, *31*(6): 73–82.

[39] Maarten M. Fokkinga. "*Calculate Categorically!*" *Formal Aspects Comput.* **1992**, *4*(6A): 673–692.

[40] Alberto Pettorossi. *Methodologies for transformations and memoing in applicative languages* [phdthesis], **1984**. http://hdl.handle.net/1842/15643.

[41] Geraint Jones and Mary Sheeran. "*Designing Arithmetic Circuits by Refinement in Ruby*". *Sci. Comput. Program.* **1994**, *22*(1-2): 107–135. https://doi.org/10.1016/0167-6423(94)90009-4.

[42] Akihiko Takano and Erik Meijer; ed. by John Williams. "*Shortcut Deforestation in Calculational Form*". In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*. ACM, **1995**: 306–313. https://doi.org/10.1145/224164.224221.

[43] Philip Wadler; ed. by Joseph E. Stoy. "*Theorems for Free!*" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. ACM, **1989**: 347–359. https://doi.org/10.1145/99370.99404.

[44] Oege de Moor. "*Categories, Relations and Dynamic Programming*". *Math. Struct. Comput. Sci.* **1994**, *4*(1): 33–69. https://doi.org/10.1017/S0960129500000360.

[45] Richard Bellman. "*Some Applications of the Theory of Dynamic Programming - A Review*". *Oper. Res.* **1954**, *2*(3): 275–288. https://doi.org/10.1287/opre.2.3.275.

[46] Richard M Karp and Michael Held. "*Finite-state processes and dynamic programming*". *SIAM Journal on Applied Mathematics*, **1967**, *15*(3): 693–718.

[47] David Pisinger and Paolo Toth. "*Knapsack problems*". In: *Handbook of combinatorial optimization*. Springer, **1998**: 299–428.

[48] Gérard Huet. "*The zipper*". *Journal of functional programming*, **1997**, *7*(5): 549–554.

[49] Karl R. Abrahamson, Norm Dadoun, David G. Kirkpatrick *et al.* "*A Simple Parallel Tree Contraction Algorithm*". *J. Algorithms*, **1989**, *10*(2): 287–302. https://doi.org/10.1016/0196-6774(89)90017-5.

[50] Conor McBride. "*The derivative of a regular type is its type of one-hole contexts*". *Unpublished manuscript*, **2001**: 74–88.

[51] Ruyi Ji, Tianran Zhu, Yingfei Xiong *et al.* "*Synthesizing Efficient Dynamic Programming Algorithms*". *CoRR*, **2022**, *abs/2202.12208*. https://arxiv.org/abs/2202.12208.

[52] Ralph Loader. "*Higher order β matching is undecidable*". *Logic Journal of the IGPL*, **2003**, *11*(1): 51–68.

[53] Gérard Huet and Bernard Lang. "*Proving and applying program transformations expressed with second-order patterns*". *Acta informatica*, **1978**, *11*(1): 31–55.

[54] Alonzo Church. "*A Formulation of the Simple Theory of Types*". *J. Symb. Log.* **1940**, *5*(2): 56–68. https://doi.org/10.2307/2266170.

[55] L. D. Baxter. "*The complexity of unification*". **1977**.

[56] Gilles Dowek. "*Third Order Matching is Decidable*". In: *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992*. IEEE Computer Society, **1992**: 2–10. https://doi.org/10.1109/LICS.1992.185514.

[57] Vincent Padovani; ed. by Stefano Berardi and Mario Coppo. "*Decidability of All Minimal Models*". In: *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*. Springer, **1995**: 201–215. https://doi.org/10.1007/3-540-61780-9%5C_71.

[58] Oege de Moor and Ganesh Sittampalam. "*Higher-order matching for program transformation*". *Theor. Comput. Sci.* **2001**, *269*(1-2): 135–162. https://doi.org/10.1016/S0304-3975(00)00402-3.

[59] Oege de Moor and Ganesh Sittampalam; ed. by S. Doaitse Swierstra, Pedro Rangel Henriques and José Nuno Oliveira. "*Generic Program Transformation*". In: *Advanced Functional Programming, Third International School, Braga, Portugal, September 12-19, 1998, Revised Lectures*. Springer, **1998**: 116–149. https://doi.org/10.1007/10704973%5C_3.

[60] T. Yokoyama, Z. Hu and M. Takeichi. "*Deterministic Second-order Patterns in Program Transformation*". In: *Conference Japan Society for Software Science & Technology*, **2003**.

[61] Tetsuo Yokoyama, Zhenjiang Hu and Masato Takeichi. "*Yicho - A System for Programming Program Calculations*". In: *The Third Asian Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29 - December 1, 2002, Proceedings*, **2002**: 366–382.

[62] Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki *et al.*; ed. by Richard S. Bird and Lambert G. L. T. Meertens. "*A calculational fusion system HYLO*". In: *Algorithmic Languages and Calculi, IFIP TC2 WG2.1 International Workshop on Algorithmic Languages and Calculi, 17-22 February 1997, Alsace, France*. Chapman & Hall, **1997**: 76–106.

[63] Isao Sasano, Zhenjiang Hu, Masato Takeichi *et al.*; ed. by Martin Odersky and Philip Wadler. "*Make it practical: a generic linear-time algorithm for solving maximum-weightsum problems*". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*. ACM, **2000**: 137–149. https://doi.org/10.1145/351240.351254.

[64] Ruyi Ji, Yingfei Xiong and Zhenjiang Hu. "*Black-Box Algorithm Synthesis - Divide-and-Conquer and More*". *CoRR*, **2022**, *abs/2202.12193*. https://arxiv.org/abs/2202.12193.

[65] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki *et al.*; ed. by Jeanne Ferrante and Kathryn S. McKinley. "*Automatic inversion generates divide-and-conquer parallel programs*". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, **2007**: 146–155. https://doi.org/10.1145/1250734.1250752.

第三章　概率编程语言

3.1　引言

在统计学中，概率推断任务是指根据样本数据去推测样本概率分布的过程。在贝叶斯学派的观点下，给定模型参数 X 的先验分布 $p(X)$，参数 X 与样本值 Y 之间的条件分布 $p(Y|X)$，概率推断任务就是在给定样本值 Y 情况下，计算模型参数 X 的后验分布 $p(X|Y)$。这几个分布之间的关系由如下公式，即贝叶斯公式，所刻画。

$$p(X|Y) = \frac{p(Y|X) \times p(Y)}{p(Y)}$$

在传统情况下，对于实际问题进行概率推断的过程可以被分为三个步骤。

1. 针对问题的特点设计概率模型，即定义先验分布 $p(X)$ 与条件分布 $p(Y|X)$。
2. 针对模型特点设计合适的推断算法。
3. 讲推断算法编程实现，并根据已有数据推断后验分布 $p(X|Y)$。

在实践过程中，上述步骤通常是迭代进行的。如果后验分布 $p(X|Y)$ 在实践中的拟合效果并不理想，则研究者会回到第一步重新尝试其他的模型。对于计算机科学领域外的研究者来说，上述步骤的第二步与第三步都是困难的，因为它们分别依赖机器学习相关的知识与编程相关的知识。这些研究者往往会花大量的时间在推导与编程上，甚至有时为了降低这两方面的难度，只能被迫简化自己的模型。

概率编程语言的目标就是自动化概率模型的推断过程，从而使得用户的精力可以集中于概率模型的设计与调整上 [1, 2]。在语法上，概率编程语言与传统编程语言类似，从而使得用户可以便捷设计、迭代概率模型。而在语义上，概率编程语言可以使用通用的推断算法来自动地完成概率推断任务。经过几十年的发展，研究者们已经设计、发布了大量可供使用的概率编程语言 [3–9]，这些语言也已经被广泛地应用在了信息提取、语音识别、计算机视觉、编码理论、生物学领域等。

在本章中，本文将首先在第 3.2 节中对已有的概率编程语言进行简单的介绍。接着，在第 3.3 中，本文以一个基本的概率编程语言为例，介绍概率编程语言的操作语义与指称语义。然后在第 3.4，本文将对概率编程语言中的两类基本推断算法进行介绍，分别是静态的基于图模型的推断算法和动态的基于采样的推断算法。最后，在第 3.5，本文将与第一章中讨论的程序合成技术将结合，介绍概率程序合成的近期进展。

3.2　概率编程语言简介

在这一节中，本文先通过几个基础的例子来展示概率编程语言的表达能力与运行效果。接着，本文对已有的概率编程语言作简单的归类与介绍。

3.2.1　概率程序示例

概率编程语言在传统的命令式或者函数式语言之上，引入两个与概率相关的组件：

- 语句 $x = dist(\theta)$ 赋予了概率程序获取随机变量的能力。它表示变量 x 的取值服从参数为 θ 的概率分布 $dist$。举例来说，x = Bernoulli(0.5) 表示布尔变量 x 的取值有一半的概率为真，一半的概率为假。
- 语句 $condition(p)$ 赋予了概率程序观察环境、推断后验分布的能力。它表示谓词 p 为真，从而影响了相关随机变量的后验分布。

```
bool c1, c2;
c1 = Bernoulli(0.5);
c2 = Bernoulli(0.5);
condition(c1 || c2);
return (c1, c2);
```

图 3.1　一个基本的概率程序。

图 3.1 展示了一个基本的概率程序。在该程序中，c_2 和 c_3 是服从分布 Bernoulli(0.5) 的随机变量，它们可以被视为两次抛一枚均匀硬币得到的结果。而第四行的 *condition* 语句表示观测到两次抛硬币的结果中，至少有一次的结果的正面。在该观测结果的影响下，(c_1, c_2) 的后验分布为集合 $\{(true, true), (true, false), (false, true)\}$ 上的均匀分布。

```
bool b, c;
b = 1;
c = Bernoulli(0.5);
while (c) {
    b = !b;
    c = Bernoulli(0.5);
}
return b;
```

图 3.2　一个基本的概率程序。

图 3.2 展示了一个包含循环的概率程序。在该程序中，c 可以被视为抛一枚均匀硬币得到的结果，而 b 表示当前抛硬币次数的奇偶性。图 3.2 中的程序描述的过程为：不

断抛一枚均匀硬币直到得到正面的结果，并输出此时抛硬币数量的奇偶性。根据该描述，可以得到程序输出 b 为真的概率是 $0.5 + 0.5^3 + 0.5^5 + \cdots = 2/3$。

除了语句 $x = dist(\theta)$ 和 $condition(p)$ 以外，还有一些其他的在概率编程语言中常见的概率相关语句。在本小节的剩余部分中，本文将对这些语句进行简单的介绍。

一些理论工作会使用语句 $x = coin()$ 和 $x = rand()$ 来代替通用的 $x = dist(\theta)$，其中 $x = coin()$ 表示整数变量 x 取 0 和 1 的概率均为 0.5, $x = rand()$ 表示实数变量 x 服从区间 $[0, 1]$ 上的均匀分布。一方面，这样的简化使得在理论分析的时候，只需要考虑两个形式相对简单的概率分布，从而降低了分析的难度。另一方面，这样的简化并不会影响语言的表达能力，因为常用的概率分布往往可以被这两个基础分布模拟出来。

例 47 下面是两个使用 $coin()$ 与 $rand()$ 模拟常见分布的例子。

- 语句 $x = Bernoulli(p)$ 等价于 $x = (rand() \le p)$。
- 根据 *Box-Muller* 变换 *[10]*，可以得到 $x = Gaussian(0, 1)$ 等价于如下表达式。

$$x = \sqrt{-2 \ln rand()} \cos(2\pi rand())$$

在概率编程语言 WebPPL [11] 中，除了语句 $condition(p)$ 以外，还使用了观测语句 $observe(x, y)$ 和 $factor(w)$。

- 语句 $observe(x, y)$ 是语句 *condition* 的特例，它表示 x 的观测结果等于 y，语义上等价于 $condition(x = y)$。
- 语句 $factor(w)$ 提供了一种定性的方式来增大或减小某种情况下的概率。在运行到该语句时，它会把这一次运行的权重（概率分布函数或概率密度函数）乘上 $\exp(w)$。因为概率会进行归一化，所以 $factor(w)$ 并不能被用于准确地操作概率分布，只能代表对相对概率大小的定性修改。

相比于语句 *condition*，语句 *observe* 和 *factor* 通常可以增加后验概率的连续性，从而降低概率推断的难度。

例 48 假设 x 是一个服从标准正态分布 $Gaussian(0, 1)$ 的连续随机变量，y 是一个服从正态分布 $Gaussian(x, 1)$ 的连续随机变量，并假设对 y 的观测值为 0。下面考虑两种对该观测行为的实现方式。

- $condition(y = 0)$。此时，因为 $y = 0$ 的概率为无穷小，所以该概率程序几乎所有的运行都被判断为了非法。因此很难推断该条件被满足时的后验分布。
- $observe(y, 0)$。此时，推断算法会使用 y 的概率密度函数来进行推断。对于每一个 x 的取值，它的权重会被赋为分布 $Gaussian(x, 1)$ 在 0 处的概率密度函数，从而对 x 的所有采样都可以被用于推断其后验分布。

```
1  theta = Beta(a=1.0, b=1.0)
2  x = Bernoulli(p=tf.ones(50) * theta)
```

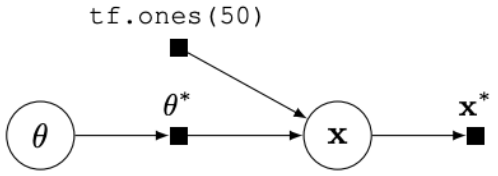

图 3.3　使用 Edward 语言编写的概率程序示例，它生成一个由独立的伯努利随机变量组成的向量。

3.2.2　已有的概率编程语言简介

在这一节中，本文按照概率编程语言的形式、应用领域，来对已有的概率编程语言进行归类与介绍。

3.2.2.1　统计学研究中的概率编程语言

如第 3.1 节中讨论的，研究概率编程语言的一个动机便是为了帮助统计学研究者提高设计模型与推断算法实现的效率。早期的概率编程语言多是以此为目标设计的。

Gilks 等人于 1994 年提出了基于 Pascal 语言的概率编程语言 BUGS [4]。一方面，该语言使用了声明式的语法来定义、描述概率模型。另一方面，该语言使用了一个通用的推断算法来进行概率推断：它首先将概率模型翻译到贝叶斯网络，然后应用吉布斯采样 (Gibss Sampling) [12] 来进行自动推断。

概率编程语言 Stan 由 Gelman 等人于 2015 年提出 [13]，是目前被统计学研究者广泛使用的概率编程语言之一。与 BUGS 相比，Stan 的推断算法并不依赖与贝叶斯网络，从而在表达能力上有所提升。Stan 主要是用马尔科夫链蒙特卡洛采样 (MCMC Sampling) [14] 和变分推断 (Variational Inference) [15] 来近似地推断后验分布。

3.2.2.2　机器学习领域中的概率编程语言

随着深度学习与神经网络相关技术的发展，设计与迭代神经网络模型渐渐成为了一项重要的任务。因此，在近几年，基于深度学习平台的概率编程语言逐渐被提出，其中具有代表性的包括基于深度学习平台 Tensorflow [16] 的 Edward [15] 和基于深度学习平台 Pytorch [17] 的 Pyro [18] 和 ProbTorch [19]。

相比于面向统计学研究的概率编程语言，这些语言依托深度学习平台搭建，因此在推断时可以使用深度学习平台提供的各种功能，例如 GPU 支持、向量化、可视化以及分布式训练。图 3.3 展示了使用 Edward 语言编写的一个简单程序。利用 Tensorflow 中对向量的支持，该程序可以方便地生成一个随机向量，同时该生成过程可以直接被并行化或者由 GPU 计算完成。

3.2.2.3 通用性的概率编程语言

在编程语言领域中，许多已有研究关注与将概率相关的组件集成进入通用的命令式或者函数式语言中，从而得到具有一定表达能力与通用性的概率编程语言。

IBAL [9]、Church [5] 等语言以函数式语言模型 lambda 演算 [20] 为基础，尝试用函数式语言来描述高阶概率模型，并使用了采样的方法来进行概率推断。

PSI [21] 是一个命令式语言，它使用符号来准确表示每个变量对应的概率分布，并使用概率分布与运算之间的关系来实现自动推断。

例 49 下面是两个在概率编程语言 *PSI* 中用符号表示概率分布的例子。

- 对于语句 $x = Uniform(a, b)$，*PSI* 会输出表达式 $[x \geq a][x \leq b]/(b - a)$，它表示了连续变量 x 的概率密度函数，其中 $[p]$ 在 p 为真时取 1，否则取 0。
- 对于语句 $x = Uniform(0, 1), y = Bernoulli(x)$，*PSI* 会输出如下表达式，它描述了连续随机变量 x 与离散随机变量 y 的联合分布。

$$[x \geq 0][x \leq 1](x \times \delta(1 - y) + (1 - x) \times \delta(y))$$

其中 $\delta(e)$ 为狄拉克 *(Dirac deltas)* 函数，它为表达式 e 的零点赋予了单位密度。

与基于采样的方法相比，PSI 使用的符号推断算法能实现对分布的精确推断。但是相对应的，它牺牲了语言的表达能力。受到变量依赖的影响，随着需要考虑的变量数量的增多，符号化表示的概率分布的规模也会指数级地上升。因此 PSI 只能处理规模相对较小的概率模型。在后续工作中，PSI 的作者 Gehr 等人又提出了概率编程语言 λPSI，将符号推断推广到了高阶函数上 [22]。

Hakaru [23] 同时支持精确的与近似的推断算法，从而给予用户在效率与准确度之间取舍的空间。Hakaru 直接使用测度论来描述概率模型，并使用名为“分解”(disintegration) 的程序变换将程序描述的概率模型分解成条件分布。Hakaru 的推断算法基于计算机代数系统 Maple [24]，并同时支持符号的精确推断与数值的近似推断。

Venture [25] 支持动态的概率模型与高阶函数。它将概率程序的执行形式化成概率运行轨迹 (probabilistic execution trace, PET)，并基于采样实现近似的概率推断。

3.2.2.4 概率逻辑语言

逻辑语言使用对象与对象之间的关系来描述一个系统，并使用自动的推理算法来进行逻辑推理。在逻辑语言的基础上，概率逻辑语言在对象之间的关系上引入了不确定性，将推理系统建模成了一个概率模型，并对推理结果的概率分布进行自动推断。已有的概率逻辑语言包括 ProbLog [26, 27], BLOG [28], PRISM [29], DeepProbLog [30] 等，而在本节中，本文以 ProbLog 为例对概率逻辑语言作简单的介绍。

```
Path(v1, v2) :- Edge(v1, v2)
Path(v1, v3) :- Path(v1, v2), Edge(v2, v3)
```

图 3.4　一个简单的 DataLog 程序。

```
Path(v1, v2) :- Edge(v1, v2)
0.5: Path(v1, v3) :- Path(v1, v2), Edge(v2, v3)
```

图 3.5　一个简单的 DataLog 程序。

ProbLog 由逻辑语言 DataLog [31] 扩展而来。图 3.4 展示了一个描述了边与路径关系的 DataLog 程序。

- 第一条规则表示任意边本身都是一条路径，它等价于如下公式。

$$\forall v_1, v_2, \text{Edge}(v_1, v_2) \rightarrow \text{Path}(v_1, v_2)$$

- 第二条规则表示任何路径都可以被一条相邻的边扩展，它等价于如下公式。

$$\forall v_1, v_2, v_3, \text{Path}(v_1, v_2) \wedge \text{Edge}(v_2, v_3) \rightarrow \text{Path}(v_1, v_3)$$

给定 $\text{Edge}(v_1, v_2)$ 和 $\text{Edge}(v_2, v_3)$，则该程序可以推出 $\text{Edge}(v_1, v_2), \text{Edge}(v_1, v_3), \text{Edge}(v_2, v_3)$。

图 3.5 展示了一个与图 3.4 相对应的 ProbLog 程序。在该程序中，关于路径扩展的规则被赋予了 0.5 的概率，它表示这一条规则的任何实例都以独立的 0.5 的概率成立。举例来说，下面两个公式是路径规则的两个不同的实例，其中 u, v, w 是三个具体节点。

$$\text{Path}(u, v) \wedge \text{Edge}(v, w) \rightarrow \text{Path}(u, w) \quad \text{Path}(u, v) \wedge \text{Edge}(v, u) \rightarrow \text{Path}(u, u)$$

根据图 3.5 中的第二条规则，在逻辑推断时，这两个实例（以及其他可能的实例）都以独立的 0.5 概率为真。此时，给定输入 $\text{Edge}(v_1, v_2)$ 和 $\text{Edge}(v_2, v_3)$，该 Problog 可以推出 $\text{Path}(v_1, v_3)$ 成立的概率为 0.5。

3.2.2.5　概率数据库编程语言

在使用关系数据库时，对象关系阻抗失衡 (object-relational impedance mismatch) 是一个重要的技术难题，它特指程序运行时使用的对象与关系数据库中以关系形式存储的数据在类型上的不匹配现象 [32]。在概率编程领域，Gordon 等人提出了概率编程语言 Tabular [33]。该语言使用关系来定义概率模型，从而在模型定义的层面上减轻了阻抗失衡的问题。

例 50 *图 3.6 展示了一个用 Tabular 语言编写的概率模型。该模型包含两个表格。*

Players			
Name	**string**	**input**	
Skill	**real**	**latent**	Gaussian(25.0,0.01)
Matches			
Player1	**link**(Players)	**input**	
Player2	**link**(Players)	**input**	
Perf1	**real**	**latent**	Gaussian(Player1.Skill,1.0)
Perf2	**real**	**latent**	Gaussian(Player2.Skill,1.0)
Win1	**bool**	**output**	Perf1 > Perf2

图 3.6　使用 Tabular 编写的概率模型示例。

$$
\begin{aligned}
d &::= a \\
&\mid x \\
&\mid d \ \mathtt{op}\ d \\
t &::= d \\
&\mid \mathtt{coin()} \mid \mathtt{rand()} \\
&\mid t \ \mathtt{op}\ t
\end{aligned}
\qquad
\begin{aligned}
b &::= \mathtt{true} \mid \mathtt{false} \\
&\mid d == d \mid d < d \mid d > d \\
&\mid b \ \&\& \ b \mid b \mid\mid b \mid !b \\
e &::= \mathtt{skip} \\
&\mid x := t \\
&\mid e \ ; \ e \\
&\mid \mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e \\
&\mid \mathtt{while}\ b\ \mathtt{do}\ e
\end{aligned}
$$

图 3.7　本节中用于定义操作语义的概率编程语言, 其中非终结符 x 表示变量，a 表示常量，d 表示确定性的语句，t 表示可能存在随机性的语句，b 表示布尔表达式，e 表示程序，而标记为红色的 *coin*() 和 *rand()* 分别表示获取一个服从分布 Bernoulli(0.5) 和分布 Uniform(0, 1) 的随机变量。

- 表格 *Players* 中的每一行描述了一名选手，包括姓名与能力值。其中每一名选手能力值的先验分布都是均值为 25.0、方差为 0.01 的正态分布。
- 表格 *Matches* 中的每一行描述了一场比赛，包括参赛双方、双方的表现与胜者。其中每一名选手的表现都是均值为其能力值、方差为 1.0 的正态分布。

给定概率模型与表格中一些位置的观测值，*Tablur* 可以自动推断缺省值的后验分布。

Tabluar 并没有自己的推断算法。它通过将用户定义的关系概率模型翻译到已有的命令式概率编程语言，并直接调用对应语言的推断算法来完成概率推断。

3.3　概率编程语言语义

在本节中，本文分别从操作语义和指称语义的视角来定义概率编程语言的语义。

3.3.1　概率编程语言的操作语义

在本节中，本文介绍由 Dahlqvist 提出的操作语义 [34]。图 3.7 展示了该操作语义基于的概率编程语言。操作语义从程序的具体执行来刻画语义。在执行概率编程语言时，程序会和环境产生三种类型的交互：

- 通过 x 获取一个变量的值或者通过 $x := t$ 写入一个变量。
- 通过 *coin*() 获取一个服从分布 Bernoulli(0.5) 的整数随机变量。
- 通过 *rand*() 获取一个服从分布 Uniform(0, 1) 的实数随机变量。

因此，概率编程语言的运行状态可以用一个三元组 (s, m, p) 表示，其中 s 存储了每一个变量的值，m 是一个从分布 Bernoulli(0.5) 产生独立随机变量的无穷序列，q 是一个从分布 Uniform(0, 1) 产生独立随机变量的无穷序列。

定义语句的求值函数 $[\![t]\!]$ 如下。它接受一个语句 t 与程序状态 (s, m, p)，并返回三元组 (a, m', p')，其中 a 表示语句 t 的值，m' 和 p' 表示求值后剩下的随机序列。

$$
\begin{aligned}
[\![a]\!] &: (s, m, p) \mapsto (a, m, p) \\
[\![x]\!] &: (s, m, p) \mapsto (s(x), m, p) \\
[\![coin()]\!] &: (s, m, p) \mapsto (head\ m, tail\ m, p) \\
[\![rand()]\!] &: (s, m, p) \mapsto (head\ p, m, tail\ p) \\
[\![t_1\ op\ t_2]\!] &: (s, m, p) \mapsto \text{let } (a_1, m', p') = [\![t_1]\!](s, m, p) \text{ in} \\
&\qquad\qquad \text{let } (a_2, m'', p'') = [\![t_2]\!](s, m', p') \text{ in} \\
&\qquad\qquad (a_1\ op\ a_2, m'', p'')
\end{aligned}
$$

给定语句的求值函数，概率程序的求值规则如下所示。

$$
\frac{[\![t]\!](s, m, p) = (a, m', p')}{(x := t, s, m, p) \to (\text{skip}, s[x_i \mapsto a], m', p')} \qquad \frac{(e_1, s, m, p) \to (e_1', s', m', p')}{(e_1; e_2, s, m, p) \to (e_1'; e_2, s', m', p')}
$$

$$
\frac{}{(\text{skip}; e, s, m, p) \to (e, s, m, p)}
$$

$$
\frac{[\![b]\!](s, m, p) = (\text{true}, m', p')}{(\text{if } b \text{ then } e_1 \text{ else } e_2, s, m, p) \to (e_1, s, m', p')}
$$

$$
\frac{[\![b]\!](s, m, p) = (\text{false}, m', p')}{(\text{if } b \text{ then } e_1 \text{ else } e_2, s, m, p) \to (e_2, s, m', p')}
$$

$$
\frac{}{(\text{while } b \text{ do } e, s, m, p) \to (\text{if } b \text{ then } (e; \text{while } b \text{ do } e) \text{ else skip}, s, m, p)}
$$

$$
\begin{array}{lll}
\mathcal{E} & ::= & \mid x \\
& & \mid c \\
& & \mid \mathcal{E}_1 \ \mathbf{bop} \ \mathcal{E}_2 \\
& & \mid \mathbf{uop} \ \mathcal{E}
\end{array}
\qquad
\begin{array}{lll}
\mathcal{S} & ::= & \\
& & \mid x = \mathcal{E} \\
& & \mid x \sim \texttt{Dist}(\bar{\theta}) \\
& & \mid \texttt{skip} \\
& & \mid \texttt{observe}\ (\varphi)
\end{array}
\qquad
\begin{array}{l}
\\
\mid \mathcal{S}_1; \mathcal{S}_2 \\
\mid \texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2 \\
\mid \texttt{while}\ \mathcal{E}\ \texttt{do}\ \mathcal{S}
\end{array}
$$

$$
\mathcal{P} \quad ::= \quad \mathcal{S}\ \texttt{return}\ (\mathcal{E})
$$

图 3.8　本节中用于定义指称语义的概率编程语言, 其中非终结符 x 表示变量，c 表示常量，$\mathcal{E}$ 表示语句，$\mathcal{S}$ 表示程序片段，$\mathcal{P}$ 表示完整的程序，而符号 bop 表示二元操作符，uop 表示一元操作符，Dist($\bar{\theta}$) 表示一个参数为 $\bar{\theta}$ 概率分布。

3.3.2　概率编程语言的指称语义

在指称语义的视角下，概率程序的状态是涉及的变量在数学上的联合分布，而概率程序的运行是将初始分布变换到最终分布的过程。因为严谨的指称语义依赖大量测度论相关的定义，所以为了简单起见，本文只介绍 Gordon 等人给出的一个简化版的语义 [33]。图 3.8 展示了该指称语义基于的概率编程语言。

Gordon 等人给出的指称语义只考虑连续变量。该语义使用求值函数 $\sigma : \text{Expr} \to \text{Val}$ 来描述程序的状态：给定语句 $\mathcal{E}$，$\sigma(\mathcal{E})$ 为该语句在当前状态下的取值。同时定义对求值函数的修改操作 $\sigma[x \leftarrow a]$，表示把变量 x 的值修改为 a，并对应地更新相关的值。

在本节考虑的语义中，一个概率程序的含义是返回结果的期望值。定义程序的返回值为函数 f，一个从具体状态到具体值的函数。定义语义函数 $[\![\mathcal{S}]\!](f)(\sigma)$ 如下所示，它表示当返回值为 f，初始状态为 σ，且执行完程序 $\mathcal{S}$ 后的返回值的期望。

$$
\begin{aligned}
[\![\text{skip}]\!](f)(\sigma) &\coloneqq f(\sigma) \\
[\![x = \mathcal{E}]\!](f)(\sigma) &\coloneqq f(\sigma[x \leftarrow \sigma(\mathcal{E})]) \\
[\![x \sim \text{Dist}(\bar{\theta})]\!](f)(\sigma) &\coloneqq \int_v \text{Dist}(\sigma(\bar{\theta}))(v) \times f(\sigma[x \leftarrow v])\mathrm{d}v \\
[\![\text{observe}(\varphi)]\!](f)(\sigma) &\coloneqq \sigma(\varphi)?f(\sigma) : 0 \\
[\![\mathcal{S}_1; \mathcal{S}_2]\!](f, \sigma) &\coloneqq [\![\mathcal{S}_1]\!]([\![\mathcal{S}_2]\!](f)))(\sigma) \\
[\![\text{if } \mathcal{E} \text{ then } \mathcal{S}_1 \text{ else } \mathcal{S}_2]\!](f)(\sigma) &\coloneqq \sigma(\mathcal{E})?[\![\mathcal{S}_1]\!](f)(\sigma) : [\![\mathcal{S}_2]\!](f)(\sigma) \\
[\![\text{while } \mathcal{E} \text{ do } \mathcal{S}]\!](f)(\sigma) &\coloneqq \sup_{n \geq 0} [\![\text{while } \mathcal{E} \text{ do}_n \ \mathcal{S}]\!](f)(\sigma) \\
&\quad \text{where} \quad \text{while } \mathcal{E} \text{ do}_0 \ \mathcal{S} = \text{observe(false)} \\
&\qquad\quad\ \ \text{while } \mathcal{E} \text{ do}_{n+1} \ \mathcal{S} = \text{if } \mathcal{S} \text{ then } (\mathcal{S}; \text{while } \mathcal{E} \text{ do}_n \ \mathcal{S}) \text{ else skip}
\end{aligned}
$$

在上述语义中，并没有考虑不符合 observe 以及会导致死循环的可能性，因此其输出的期望值是未经标准化的，即考虑的所有情况的概率和并不为 1。因此在最终输出的时候，需要通过如下公式将期望标准化。

$$[\![\mathcal{S} \text{ return } \mathcal{E}]\!] \coloneqq \frac{[\![\mathcal{S}]\!](\lambda\sigma, \sigma(\mathcal{E}))(\bot)}{[\![\mathcal{S}]\!](\lambda\sigma, 1)(\bot)}$$

其中 $\bot$ 表示空状态，其中所有变量的取值均为默认值。

3.4　概率推断方法

已有的自动推断方法大致可以分为静态和动态两类。

- 静态方法一般根据程序的文法将程序翻译到静态的概率模型（例如图模型），然后直接使用对应模型的推断算法。
- 动态方法一般通过对概率程序运行轨迹的采样来得到近似的推断结果。

在这一节中，本文将分别对这两类推断方法中的典型技术进行介绍。

3.4.1　静态的自动推断算法

静态的推断算法将概率程序翻译到概率图模型，然后再使用已有的图模型上的推断算法来进行概率推断。

3.4.1.1　概率程序到图模型的自动翻译

贝叶斯网 (Bayesian network) 是一种经典的概率图模型，它将变量之间的条件依赖建模成一张有向无环图 (directed acyclic graph, DAG)。其中图上每一个节点代表着一个随机变量，而一条 u 到 v 的有向边表示 v 的取值对 u 的取值有条件依赖。

对于随机变量 x_i，假设它依赖的随机变量的编号为 $a^i_1, \ldots, a^i_{t_i}$，根据有向无环图的结构，由贝叶斯网表示的 n 个变量 $x_1, \ldots, x_n$ 的联合分布可以被拆分成关于每一个变量条件分布的乘积，如下所示。

$$\Pr[x_1 = v_1, \ldots, x_n = v_n] \coloneqq \prod_{i=1}^{n} \Pr\left[x_i = v_i \middle| \bigwedge_{j=1}^{t_i} x_{a^i_j} = v_{a^i_j}\right]$$

例 51 图 *3.9* 展示了一个设计三个随机变量的贝叶斯网。它所描述的联合概率分布可以通过如下方式拆分成为条件分布。

$$\Pr[a = v_1, b = v_2, c = v_3] = \Pr[a = v_1] \times \Pr[b = v_2 | a = v_1] \times \Pr[c = v_3 | a = v_1, b = v_2]$$

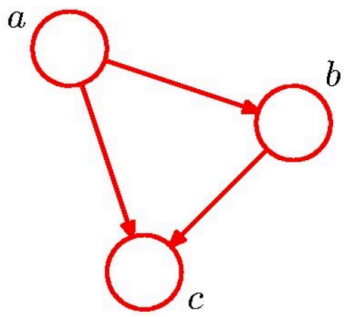

图 3.9　一个关于三个随机变量 $\{a, b, c\}$ 的贝叶斯网。

将概率程序翻译成贝叶斯网的过程可以分为两步。首先，因为在贝叶斯网中，每一个随机变量的取值是不可更改的，所以需要将概率程序转成静态单赋值 (static single assignment, SSA) 的形式。然后再将静态单赋值的程序翻译到贝叶斯网络。

例 52 考虑如下的概率程序。该程序涉及三个变量 x, y, z，其中 y 的分布取决于 x 的布尔值，z 的期望依赖于 y 的取值，而 z 又被要求满足大于 10 这一条件。该程序对

```
x = bernoulli(0.2)
if (x) y = uniform(0, 2)
else y = gaussian(0, 5)
z = gaussian(y, 1)
condition(z > 10)
```

应的静态单赋值形式如下所示，其中 y 被拆分成了三个变量 y_T, y_F 和 ϕ，分别表示 y 在两个分支运行时的取值以及其真实值，$\phi(b, w_F, w_F)$ 是一个合并算子，其语义等同于 $b?w_T : w_F$。

```
x = bernoulli(0.2)
y_T = uniform(0, 2)
y_F = gaussian(0, 5)
y = φ(x, y_T, y_F)
z = gaussian(y, 1)
condition(z > 10)
```

对应该静态单赋值的贝叶斯网络如图 3.10 所示。直观来说，它对每一次赋值构建了一个节点，并根据对应的依赖的关系连边。

3.4.1.2　图模型上的自动推断

当所有随机变量均为值域有限的离散随机变量时，变量消除 (variable elimination) [35] 和信念传播 (belief propagation) [36] 均可以在贝叶斯网上作精确的概率推断。

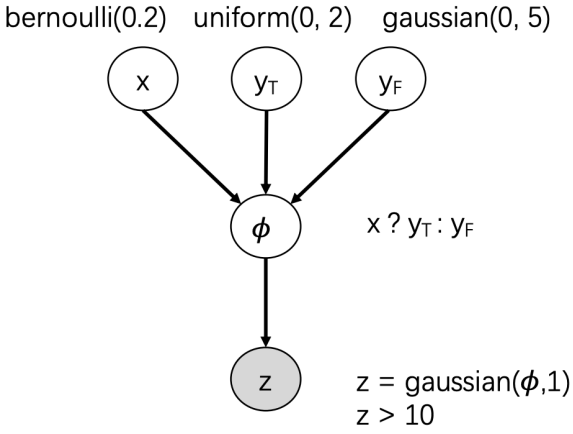

图 3.10　由例 52 中的程序翻译得到的贝叶斯网络。

变量消除算法每次消去贝叶斯网络中的一个随机变量，直到得到推断目标的概率分布。假设当前要消去的变量是 x_i，它依赖随机变量 $u_1, \ldots, u_n$ 且被随机变量 $v_1, \ldots, v_m$ 依赖。变量消除算法在删去变量 x_i 的同时，会在随机变量 (u_i, v_j) 之间以及 (v_i, v_j) 之间按照拓扑序添加依赖关系，并对应地修改每个节点条件分布。

当贝叶斯网的结构是树形的时候，信仰传播算法可以作精确的概率推断。简单来说，信仰传播算法对每一棵子树都计算出这棵子树对应的概率模型中根节点所对应的随机变量的边缘分布，并把改边缘分布传播到整棵树的根节点。而对于一般性的图模型，信仰传播算法只能作近似的概率推断。它不断地根据图的结构迭代每一个点的边缘分布，直到目标分布收敛为止。

理论上，当所有随机变量均为值域有限的离散随机变量时，对贝叶斯网络的精确推断是一个 NP-hard 的问题。而当随机变量的值域无界时，该精确推断问题甚至是不可判定的。因此，为了对涉及大量随机变量的概率程序进行概率推断，发展并使用近似的推断算法是非常必要的。

变分推断 (variational inference) [37] 尝试使用简单的概率分布来近似地逼近后验分布。假设我们要用模型 $q(z)$ 来近似后验分布 $p(z|x)$，则变分推断基于如下的推导。

$$\ln p(x) = \ln p(x, z) - \ln p(z|x) = \ln p(x, z) - \ln p(z) + \ln \frac{q(z)}{p(z|x)}$$

在等式两遍同对分布 $q(z)$ 作期望，可以得到如下推导。

$$\begin{aligned}\ln p(x) &= \left(\int q(z) \log p(x, z) \mathrm{d}z - \int q(z) \log q(z) \mathrm{d}z \right) + \int q(z) \ln \frac{q(z)}{p(z|x)} \mathrm{d}z \\ &= \mathcal{L}\Big(q(z)\Big) + \mathrm{KL}\Big(q(z) \Big\| p(z|x)\Big)\end{aligned}$$

其中 $\mathcal{L}(q)$ 是证据下界 (evidence lower bound, ELOB)，$\mathrm{KL}(p||q)$ 是两个分布的 KL 散度。

因为 KL 散度非负且其值越小代表两个分布越接近，所以要用 $q(z)$ 去近似 $p(z|x)$，只需要让证据下界 $\mathcal{L}(q(z))$ 尽可能大即可。

蒙特卡洛 (Monte Carlo) [38] 方法使用采样的方式来作近似的概率推断。假设目前需要计算函数 $f(x)$ 的期望，其中 x 是服从概率分布 p 的一个随机变量，则蒙特卡洛方法通过如下方式来近似该期望。

$$\mathop{\mathbb{E}}_{x\sim p}[f(x)] \approx \sum_{i=1}^{n} f(x_i) \Big/ n$$

其中 $x_1, \ldots, x_n$ 是一系列从分布 p 上采得的样本。

在实践中，因为概率分布 p 通常较为复杂，例如是一个复杂概率模型的后验分布，所以直接得到样本 $x_1, \ldots, x_n$ 通常是复杂的。此时需要引入一些相对复杂的采样算法。

拒绝采样 (reject sampling) [39] 和重要性采样 (importance sampling) [40] 都尝试将采样任务转移到另一个相对简单的分布 q 上。拒绝采样对概率分布 p 与 q 之间的相似度有一定要求：它假设存在一个常数 K 使得 $kq(x) \geq p(x)$。在这一假设下，拒绝采样通过如下方式来获得分布 p 上的样本。

- 按照概率分布 q 采样得到样本 x，按照 $[0, kq(x)]$ 上的均匀分布采样得到样本 y。
- 如果 $y \leq p(x)$，则返回 x 作为结果。否则放弃样本 x，回到第一步重新采样。

分析上述过程可以得到，拒绝采样结果服从概率分布 p 且平均采样次数为 k。

相比之下，重要性采样通过如下推导将分布 p 上的期望转化成分布 q 上的期望。

$$\mathop{\mathbb{E}}_{x\sim p}[f(x)] = \int p(x)f(x)\mathrm{d}x = \int q(x)\frac{p(x)f(x)}{q(x)}\mathrm{d}x = \mathop{\mathbb{E}}_{x\sim q}\left[\frac{p(x)f(x)}{q(x)}\right]$$

p 与 q 之间的相关性并不依赖重要性采样的正确性，但是它会影响重要性采样的收敛速度。q 越接近 p，则重要性采样收敛的速度就会越快。

马尔科夫链蒙特卡洛方法 (Markov chain Monte Carlo, MCMC) 是另一种常用的采样方法 [41]。简单来说，该方法的思想是构造一个在随机变量取值上的马尔科夫链满足其平稳分布等于采样的目标分布，并通过在马尔科夫链上随机游走来得到目标分布上的样本。使用该方法的关键在于构造一个具有目标稳态分布的马尔科夫链。

Metropolis 等人提出了一个通用的构造符合条件的马尔科夫链的算法，即 Metropolis 算法 [42]。给定目标分布 q 以及任意具有对称转移函数 q (即 $q(x|y) = q(y|x)$) 的马尔科夫链，该方法通过如下方式（隐形地）构造了一个新的新的转移函数 $q^*(x|y)$：对于每一次从赋值 x 到赋值 y 的转移，都只以 $\alpha(x, y) = \min(1, p(y)/p(x))$ 的概率接受，

否则保持不变。通过如下等式可以验证 p 是构造得到的马尔科夫链上的一个平稳分布。

$$p(y)q^*(x|y) = p(y)q(x|y)\min\left(1, \frac{p(x)}{p(y)}\right) = q(x|y)\min(p(x), p(y))$$
$$p(x)q^*(y|x) = p(x)q(y|x)\min\left(1, \frac{p(y)}{p(x)}\right) = q(x|y)\min(p(x), p(y))$$

Hastings 将该算法拓展到了非对称的马尔科夫链上，即 Metropolis-Hasting 算法 [43]。在拓展的算法中，接受概率 $\alpha(x, y)$ 被修改为了 $\min(1, (p(y)q(x|y))/(p(x)q(y|x)))$。此时，下列公式说明了该方法的正确性。

$$p(y)q^*(x|y) = p(y)q(x|y)\min\left(1, \frac{p(x)q(y|x)}{p(y)q(x|y)}\right) = \min(p(x)q(y|x), p(y)q(x|y))$$
$$p(x)q^*(y|x) = p(x)q(y|x)\min\left(1, \frac{p(y)q(x|y)}{p(x)q(y|x)}\right) = \min(p(x)q(y|x), p(y)q(x|y))$$

3.4.2　动态的自动推断算法

因为概率程序到静态图模型的翻译需要在编译时就确定运行时产生的随机变量以及其依赖关系，所以静态的推断算法难以处理递归的控制结构以及高阶函数。以图 3.2 中的抛硬币程序为例：该程序不断地产生服从伯努利分布的随机变量直到得到真为止。该程序使用的随机变量的数量只有在运行时才能决定，且其具体数值可以是任意大的。因此，这一程序无法被翻译到静态图模型，从而无法使用基于静态图模型的静态推断算法来进行概率推断。

为了处理这些动态产生的随机变量，动态的推断算法直接对程序的运行轨迹采样，从而近似地进行概率推断。这些推断算法主要基于第 3.4.1.2 中介绍的蒙特卡洛方法。根据程序运行轨迹的特点，研究人员提出了大量适用于采用运行轨迹的高效算法。本文选取其中的两个典型算法，*单点更新 MH 算法*与*序列蒙特卡洛算法*，进行介绍。

单点更新 MH 算法 (single-site Metropolis-Hasting) 在 Metropolis-Hasting 算法的基础上，使用了一个适用于采样运行轨迹的初始转移函数 $q(x|y)$。简单来说，在每次转移的时候，单点更新 MH 算法从当前轨迹使用的随机变量中等概率选取一个，并对该变量以及后续运行轨迹进行重新求值。假设原轨迹和新轨迹中变量的赋值分别为 X, X'，观测集合为 Y，原轨迹和新轨迹中变量的概率密度函数以及观测发生的概率为 P 和 P'，且转移时选取的随机变量为 x_0。根据 Metropolis-Hasting 算法，使用接受函数 $\min(1, \alpha)$ 即可得到一个稳态分布为轨迹上的后验分布 $p(X|Y)$ 的马尔科夫链，α 的定义如下。

$$\begin{aligned}\alpha &= \frac{p(X'|Y)q(X|X')}{p(X|Y)q(X'|X)} = \frac{p(X',Y)q(X|X')}{p(X,Y)q(X'|X)} \\ &= \frac{p(Y,X')}{q(X'|X,x_0)} \times \frac{q(X|X',x_0)}{p(Y,X)} \times \frac{q(x_0|X')}{q(x_0|X)} \\ &= \frac{|\mathrm{dom}(X)|}{|\mathrm{dom}(X')|} \times \frac{\prod_{y\in Y} P'(y) \prod_{x\in(X\cap X')} P'(x)}{\prod_{y\in Y} P(y) \prod_{x\in(X\cap X')} P(x)}\end{aligned}$$

其中 $\mathrm{dom}(X)$ 为运行轨迹 X 中使用的随机变量数量。因为该接受函数只涉及随机变量数量、每一个随机变量的概率（或概率密度函数）以及每一次观测成立的概率（或概率密度函数），所以它可以被高效地计算。

序列蒙特卡洛算法 (sequential Monte-Carlo, SMC) [44] 将概率程序的运行看成一个序列决策过程：每一步运行或者引入一个新的随机变量，或者通过一次观察改变后验分布。从空程序以及空程序运行轨迹的一些样本（即空轨迹）出发，序列蒙特卡洛算法依次考虑概率程序的每一步运行过程，并根据前一步的样本产生一系列对应当前程序后验分布的运行轨迹样本。

- 如果新一步运行产生了一个新的随机变量，则对每一个轨迹样本计算该随机变量的概率分布，并从对应的概率分布中采样得到该轨迹中新随机变量的取值。
- 如果新一步运行产生了一个新的观测，则计算每一个轨迹样本满足该观测的概率（或概率密度函数）调整每一个轨迹样本的权值。

当观测数量过多的时候，上述采样过程可能会导致每一个轨迹样本的权重都非常小，从而无法得到对目标分布的一个高效估计。因此在实践中，序列蒙特卡洛算法通常会使用重采样过滤器 (Bootstrap filter) 使得采样过程专注于高可能的运行分支。

在一次新的观测结束后，假设当前的样本为 $x_1, \ldots, x_n$ 且权重为 $w_1, \ldots, w_n$。重采样过滤器会按照正比于 w_i 的概率在集合 $\{x_1, \ldots, x_n\}$ 中重新采样 n 次，得到新的样本集合 $x'_1, \ldots, x'_n$。因为这次采样的概率已经正比于原先的权重 w_i 了，所以每一个新样本的权重都被重置为了 1。从效果上来说，权重越高的原先样本在新样本中的使用次数会越多，因此后续的采样过程会更加关注于具有高概率的分支。

3.5 概率程序的自动合成

随着概率变成语言的发展，与概率程序相关的问题与领域也渐渐变得活跃，包括概率程序的验证 [45–48]、概率程序上的逻辑系统 [49–52]、概率程序上的抽象解释 [53–55] 以及概率程序的自动合成 [56]。与本文第一章相呼应，在这一节中，本文简单介绍概率程序上的自动合成技术的进展。

概率程序的自动合成类似于机器学习领域的自动学习 (automated machine learning,

AutoML) 问题。因为一个概率程序描述了一个概率模型，所以概率程序的自动合成本质上是自动地寻找模型来建模实际问题的过程。而相比于传统的程序合成，概率程序合成的难点在于概率程序的输出是一个（近似的）分布，从而传统程序合成依赖的基于例子的方法以及验证方法都变得难以使用了。

Saad 等人于 2019 年形式化了概率程序的自动合成问题，并提出了一个使用概率推断算法本身来进行概率程序合成的框架 [56]。给定一个包含概率程序的领域特定语言 $\mathcal{L}$，该语言上的先验分布 $\texttt{Prior}$，以及一系列的观测值 X，概率程序合成的目标就是从后验分布 $\texttt{Post}[X]$ 中进行采样，其定义为 $\texttt{Post}[X](p) \propto (\texttt{Lik}[p](X)\texttt{Prior}(p))$，其中 $\texttt{Lik}[p](X)$ 表示在使用概率语言 p 时，观测值 X 的似然度。

Saad 等人提出的合成算法的核心是将领域特定语言 $\mathcal{L}$ 中的一个随机程序的语义给编写成一个概率程序。该方法使用参数化的概率上下文无关文法来描述领域特定语言 $\mathcal{L}$ 上的分布。该方法使用类似于第 1.4.3 节中介绍的基于语法树的语义编码方法，构造了一个元概率程序使得其语义等价于服从分布 $\mathcal{P}$ 的随机程序的语义。此时，根据观测值 X 对元程序进行推断，即可近似地得到领域特定语言 $\mathcal{L}$ 中程序的后验分布。

作为概率程序合成的早期工作，该方法的合成效率受限于元概率程序所使用的概率编程语言中推断算法的效率，因此还无法被用于生成复杂的概率程序。如何为概率程序合成定制高效的推断算法，以及如何将其他类别的程序合成技术应用到概率程序的合成中，都是该领域中值得研究的后续问题。

参考文献

参考文献

[1] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang *et al.* "*An Introduction to Probabilistic Programming*". *CoRR*, **2018**, *abs/1809.10756*. http://arxiv.org/abs/1809.10756.

[2] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori *et al.*; ed. by James D. Herbsleb and Matthew B. Dwyer. "*Probabilistic programming*". In: *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*. ACM, **2014**: 167–181. https://doi.org/10.1145/2593882.2593900.

[3] Johannes Borgström, Andrew D. Gordon, Michael Greenberg *et al.*; ed. by Gilles Barthe. "*Measure Transformer Semantics for Bayesian Machine Learning*". In: *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Springer, **2011**: 77–96. https://doi.org/10.1007/978-3-642-19718-5%5C_5.

[4] Wally R Gilks, Andrew Thomas and David J Spiegelhalter. "*A language and program for complex Bayesian modelling*". *Journal of the Royal Statistical Society: Series D (The Statistician)*, **1994**, *43*(1): 169–177.

[5] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy *et al.*; ed. by David A. McAllester and Petri Myllymäki. "*Church: a language for generative models*". In: *UAI 2008, Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence, Helsinki, Finland, July 9-12, 2008*. AUAI Press, **2008**: 220–229. https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1%5C&smnu=2%5C&article%5C_id=1346%5C&proceeding%5C_id=24.

[6] Andrew D. Gordon, Mihhail Aizatulin, Johannes Borgström *et al.*; ed. by Roberto Giacobazzi and Radhia Cousot. "*A model-learner pattern for bayesian reasoning*". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM, **2013**: 403–416. https://doi.org/10.1145/2429069.2429119.

[7] SKPSM Richardson, Pedro Domingos and Marc Sumner Hoifung Poon. *The alchemy system for statistical relational ai: User manual*. Citeseer, **2007**.

[8] Daphne Koller, David A. McAllester and Avi Pfeffer; ed. by Benjamin Kuipers and Bonnie L. Webber. "*Effective Bayesian Inference for Stochastic Programs*". In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*. AAAI Press / The MIT Press, **1997**: 740–747. http://www.aaai.org/Library/AAAI/1997/aaai97-115.php.

[9] Avi Pfeffer. "*The design and implementation of IBAL: A generalpurpose probabilistic programming language*". In: *Harvard Univesity*, **2005**.

[10] George EP Box. "*A note on the generation of random normal deviates*". *Ann. Math. Statist.* **1958**, *29*: 610–611.

[11] David Tolpin, Jan-Willem van de Meent, Hongseok Yang *et al.* "*Design and implementation of probabilistic programming language anglican*". In: *Proceedings of the 28th Symposium on the Implementation and Application of Functional programming Languages*, **2016**: 1–12.

[12] Stuart Geman and Donald Geman. "*Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images*". *IEEE Trans. Pattern Anal. Mach. Intell.* **1984**, *6*(6): 721–741. https://doi.org/10.1109/TPAMI.1984.4767596.

[13] Andrew Gelman, Daniel Lee and Jiqiang Guo. "*Stan: A probabilistic programming language for Bayesian inference and optimization*". *Journal of Educational and Behavioral Statistics*, **2015**, *40*(5): 530–543.

[14] Christophe Andrieu, Nando de Freitas, Arnaud Doucet *et al.* "*An Introduction to MCMC for Machine Learning*". *Mach. Learn.* **2003**, *50*(1-2): 5–43. https://doi.org/10.1023/A:1020281327116.

[15] David M. Blei, Alp Kucukelbir and Jon D. McAuliffe. "*Variational Inference: A Review for Statisticians*". *CoRR*, **2016**, *abs/1601.00670*. http://arxiv.org/abs/1601.00670.

[16] Martn Abadi, Ashish Agarwal, Paul Barham *et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, **2015**. https://www.tensorflow.org/.

[17] Adam Paszke, Sam Gross, Francisco Massa *et al.* "*PyTorch: An Imperative Style, High-Performance Deep Learning Library*". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., **2019**: 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[18] Eli Bingham, Jonathan P. Chen, Martin Jankowiak *et al.* "*Pyro: Deep Universal Probabilistic Programming*". *J. Mach. Learn. Res.* **2019**, *20*: 28:1–28:6. http://jmlr.org/papers/v20/18-403.html.

[19] N. Siddharth, Brooks Paige, Jan-Willem van de Meent *et al.*; ed. by I. Guyon, U. V. Luxburg, S. Bengio *et al.* "*Learning Disentangled Representations with Semi-Supervised Deep Generative Models*". In: *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., **2017**: 5927–5937. http://papers.nips.cc/paper/7174-learning-disentangled-representations-with-semi-supervised-deep-generative-models.pdf.

[20] Alonzo Church. "*A set of postulates for the foundation of logic*". *Annals of mathematics*, **1932**: 346–366.

[21] Timon Gehr, Sasa Misailovic and Martin T. Vechev; ed. by Swarat Chaudhuri and Azadeh Farzan. "*PSI: Exact Symbolic Inference for Probabilistic Programs*". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Springer, **2016**: 62–83. https://doi.org/10.1007/978-3-319-41528-4%5C_4.

[22] Timon Gehr, Samuel Steffen and Martin T. Vechev; ed. by Alastair F. Donaldson and Emina Torlak. "*λPSI: exact inference for higher-order probabilistic programs*". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, **2020**: 883–897. https://doi.org/10.1145/3385412.3386006.

[23] Praveen Narayanan, Jacques Carette, Wren Romano *et al.* "*Probabilistic inference by program transformation in Hakaru (system description)*". In: *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, **2016**: 62–79. http://dx.doi.org/10.1007/978-3-319-29604-3_5.

[24] Maplesoft, a division of Waterloo Maple Inc.. *Maple*. Waterloo, Ontario, **2019**. https://hadoop.apache.org.

[25] Vikash K. Mansinghka, Daniel Selsam and Yura N. Perov. "*Venture: a higher-order probabilistic programming platform with programmable inference*". *CoRR*, **2014**, *abs/1404.0099*. http://arxiv.org/abs/1404.0099.

[26] Luc De Raedt, Angelika Kimmig and Hannu Toivonen; ed. by Manuela M. Veloso. "*ProbLog: A Probabilistic Prolog and Its Application in Link Discovery*". In: *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, **2007**: 2462–2467. http://ijcai.org/Proceedings/07/Papers/396.pdf.

[27] Anton Dries, Angelika Kimmig, Wannes Meert *et al.*; ed. by Albert Bifet, Michael May, Bianca Zadrozny *et al.* "*ProbLog2: Probabilistic Logic Programming*". In: *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part III*. Springer, **2015**: 312–315. https://doi.org/10.1007/978-3-319-23461-8%5C_37.

[28] Brian Milch, Bhaskara Marthi, Stuart J. Russell *et al.*; ed. by Leslie Pack Kaelbling and Alessandro Saffiotti. "*BLOG: Probabilistic Models with Unknown Objects*". In: *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*. Professional Book Center, **2005**: 1352–1359. http://ijcai.org/Proceedings/05/Papers/1546.pdf.

[29] Taisuke Sato and Yoshitaka Kameya. "*PRISM: A Language for Symbolic-Statistical Modeling*". In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*. Morgan Kaufmann, **1997**: 1330–1339. http://ijcai.org/Proceedings/97-2/Papers/078.pdf.

[30] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig *et al.* "*Neural probabilistic logic programming in DeepProbLog*". *Artif. Intell.* **2021**, *298*: 103504. https://doi.org/10.1016/j.artint.2021.103504.

[31] Serge Abiteboul, Richard Hull and Victor Vianu. *Foundations of Databases*. Addison-Wesley, **1995**. http://webdam.inria.fr/Alice/.

[32] Christopher Ireland, David Bowers, Michael Newton *et al.*; ed. by Qiming Chen, Alfredo Cuzzocrea, Takahiro Hara *et al.* "*A Classification of Object-Relational Impedance Mismatch*". In: *The First International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDS 2009, Gosier, Guadeloupe, France, 1-6 March 2009*. IEEE Computer Society, **2009**: 36–43. https://doi.org/10.1109/DBKDA.2009.11.

[33] Andrew D. Gordon, Thore Graepel, Nicolas Rolland *et al.*; ed. by Suresh Jagannathan and Peter Sewell. "*Tabular: a schema-driven probabilistic programming language*". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego,*

CA, USA, January 20-21, 2014. ACM, **2014**: 321–334. https://doi.org/10.1145/2535838.2535850.

[34] Fredrik Dahlqvist, D Kozen and A Silva. "*Semantics of probabilistic programming: A gentle introduction*". *Foundations of Probabilistic Programming*, **2020**: 1–42.

[35] Nevin L Zhang and David Poole. "*A simple approach to Bayesian network computations*". In: *Proc. of the Tenth Canadian Conference on Artificial Intelligence*, **1994**.

[36] Judea Pearl. "*Reverend Bayes on inference engines: A distributed hierarchical approach*". In: *Probabilistic and Causal Inference: The Works of Judea Pearl*, **2022**: 129–138.

[37] Charles W. Fox and Stephen J. Roberts. "*A tutorial on variational Bayesian inference*". *Artif. Intell. Rev.* **2012**, *38*(2): 85–95. https://doi.org/10.1007/s10462-011-9236-8.

[38] Dirk P Kroese, Tim Brereton, Thomas Taimre *et al.* "*Why the Monte Carlo method is so important today*". *Wiley Interdisciplinary Reviews: Computational Statistics*, **2014**, *6*(6): 386–392.

[39] Martin T Wells, George Casella, Christian P Robert *et al.* "*Generalized accept-reject sampling schemes*". In: *A festschrift for herman rubin*. Institute of Mathematical Statistics, **2004**: 342–348.

[40] Surya T Tokdar and Robert E Kass. "*Importance sampling: a review*". *Wiley Interdisciplinary Reviews: Computational Statistics*, **2010**, *2*(1): 54–60.

[41] Steve Brooks, Andrew Gelman, Galin Jones *et al. Handbook of markov chain monte carlo*. CRC press, **2011**.

[42] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth *et al.* "*Equation of state calculations by fast computing machines*". *The journal of chemical physics*, **1953**, *21*(6): 1087–1092.

[43] W Keith Hastings. "*Monte Carlo sampling methods using Markov chains and their applications*". **1970**.

[44] Arnaud Doucet, Nando de Freitas and Neil J. Gordon; ed. by Arnaud Doucet, Nando de Freitas and Neil J. Gordon. "*An Introduction to Sequential Monte Carlo Methods*". In: *Sequential Monte Carlo Methods in Practice*. Springer, **2001**: 3–14. https://doi.org/10.1007/978-1-4757-3437-9%5C_1.

[45] Jinyi Wang, Yican Sun, Hongfei Fu *et al.*; ed. by Stephen N. Freund and Eran Yahav. "*Quantitative analysis of assertion violations in probabilistic programs*". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. ACM, **2021**: 1171–1186. https://doi.org/10.1145/3453483.3454102.

[46] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady *et al.*; ed. by Kathryn S. McKinley and Kathleen Fisher. "*Cost analysis of nondeterministic probabilistic programs*". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, **2019**: 204–220. https://doi.org/10.1145/3314221.3314581.

[47] Di Wang, Jan Hoffmann and Thomas W. Reps; ed. by Stephen N. Freund and Eran Yahav. "*Central moment analysis for cost accumulators in probabilistic programs*". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual*

Event, Canada, June 20-25, 2021. ACM, **2021**: 559–573. https://doi.org/10.1145/3453483.3454062.

[48] Di Wang, Jan Hoffmann and Thomas W. Reps; ed. by Stephen N. Freund and Eran Yahav. "*Sound probabilistic inference via guide types*". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. ACM, **2021**: 788–803. https://doi.org/10.1145/3453483.3454077.

[49] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen *et al.* "*Relatively complete verification of probabilistic programs: an expressive language for expectation-based reasoning*". *Proc. ACM Program. Lang.* **2021**, *5*(POPL): 1–30. https://doi.org/10.1145/3434320.

[50] Gilles Barthe, Benjamin Grégoire and Santiago Zanella Béguelin; ed. by Zhong Shao and Benjamin C. Pierce. "*Formal certification of code-based cryptographic proofs*". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. ACM, **2009**: 90–101. https://doi.org/10.1145/1480881.1480894.

[51] Gilles Barthe, Boris Köpf, Federico Olmedo *et al.* "*Probabilistic Relational Reasoning for Differential Privacy*". *ACM Trans. Program. Lang. Syst.* **2013**, *35*(3): 9:1–9:49. https://doi.org/10.1145/2492061.

[52] Tetsuya Sato, Alejandro Aguirre, Gilles Barthe *et al.* "*Formal verification of higher-order probabilistic programs: reasoning about approximation, convergence, Bayesian inference, and optimization*". *Proc. ACM Program. Lang.* **2019**, *3*(POPL): 38:1–38:30. https://doi.org/10.1145/3290351.

[53] David Monniaux; ed. by Jens Palsberg. "*Abstract Interpretation of Probabilistic Semantics*". In: *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings*. Springer, **2000**: 322–339. https://doi.org/10.1007/978-3-540-45099-3%5C_17.

[54] David Monniaux. "*Abstract interpretation of programs as Markov decision processes*". *Sci. Comput. Program.* **2005**, *58*(1-2): 179–205. https://doi.org/10.1016/j.scico.2005.02.008.

[55] David Monniaux; ed. by David Sands. "*Backwards Abstract Interpretation of Probabilistic Programs*". In: *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. Springer, **2001**: 367–382. https://doi.org/10.1007/3-540-45309-1%5C_24.

[56] Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle *et al.* "*Bayesian synthesis of probabilistic programs for automatic data modeling*". *Proc. ACM Program. Lang.* **2019**, *3*(POPL): 37:1–37:32. https://doi.org/10.1145/3290350.

第四章　博士论文研究内容设想

本文对程序语言领域的三个不同的研究方向，程序合成、程序演算和概率编程，进行了系统性的调研。虽然这三个方向的关注点个各不相同，分别在于程序的自动生成、程序之间的变换和程序对概率模型的支持，但是它们之间的结合是有着显著的意义的。

传统的合成问题只关系合成结果的正确性。但是在实践中，程序的效率同样是十分重要的。在保证正确性的情况下，一个自然的目标便是保证合成结果的效率（例如时间开销、空间开销）在一个可以接受的范围内。然而，在程序合成问题中引入效率相关的限制通常会极大地增加合成问题的难度。一个高效的程序往往需要使用算法层面上的优化，例如动态规划、分治并行化等，这些优化往往会增大目标程序的规模与复杂程度。受限于合成效率，目前几乎所有的程序合成技术仅能支持表达式层面的程序，从而很难将已有的技术用于高效算法的合成。

本文作者发现，通过使用程序演算领域中算法相关的变换规则，可以将特定算法的合成问题重新简化到表达式的层面。以分治并行化为例，显而易见，列表上的分治并行化算法是复杂的。一方面，它引入了递归、并行等复杂的控制流。另一方面，它需要大量的语法组件用于递归调用以及边界情况的处理，其规模早已超出了表达式的级别。然而，在已知第三列表同态定理（定理 3）以及并行化列表同态的通用算法的情况下，合成分治并行化算法这一复杂问题可以被转化为列表同态的合成问题。此时，程序合成器只需要找到操作符 $\odot$ 和函数 f 使得列表同态 $hom\,(\odot)\ f$ 是一个合法程序即可。注意到 $\odot$ 和 f 往往远比完整的分置并行化程序简单，因为它们既不设计任何循环、递归、并行的复杂控制流，也不需要处理分治时的变价情况。因此，通过使用程序演算领域中的结果，程序合成问题的难度被极大地降低了。

而在这个生成过程中，概率编程相关的技术可以被用于解决验证方面的问题。在语法制导的程序合成框架中，SMT 求解器被用于判定一个候选程序的正确性并提供对应的反例。然而，因为算法相关的规约往往涉及不被已有的 SMT 求解器支持的语法组件，例如高阶函数、递归数据结构、递归函数等，所以这一方法并不能被应用到算法合成的场景中。避开这一问题的一个方法是使用随机测试替代形式化的正确性验证。根据概率近似正确性的框架，如果一个程序在足够多的独立同分布的随机例子下都满足规约，则该程序在相同分布的随机例子下违反规约的概率以高概率不超过一个足够小的阈值。因此，如果能对实际中例子的分布进行准确的建模，那么就可以通过在该分布下对合成结果进行随机测试来获得一个近似的正确性保证。而对分布建模的这一过程可以使用概率编程的技术完成。除此以外，目前也有工作使用概率编程中的分析工

具来将复杂的规约分解成简单的子问题。因为算法合成问题中的规约通常具备一定的复杂性，所以这一方向上将概率编程与程序合成技术结合的尝试也是值得关注的。

基于以上的分析，本文作者计划对程序合成技术、程序演算规则与概率编程技术在算法合成这一问题上的结合展开深入研究，目标是对常见的算法，例如动态规划、贪心、分治等，提出高效的自动程序合成器。其中可以预见的难点在于，通过程序演算规则转化得到的程序合成问题通常是关系的，即通常涉及多个目标程序且不存在输入输出样例。根据第一章中的总结，目前已有的程序合成技术在求解关系的程序合成问题是的效率并不理想，因此已有的合成技术可能并不足以求解化简后的程序合成问题。在这项研究中，可能需要根据化简后问题的具体形式，来设计针对性的程序合成方法。

参与的研究工作情况

会议论文

1. Ruyi Ji, Jingtao Xia, Yingfei Xiong, Zhenjiang Hu. *Generalizable Synthesis Through Unification*. OOPSLA'21: Object Oriented Programming Languages, Systems and Applications, October 2021.
2. Jingjing Liang, Ruyi Ji, Jiajun Jiang, Shurui Zhou, Yiling Lou, Yingfei Xiong, Gang Huang. *Interactive Patch Filtering as Debugging Aid*. ICSME'21: 37th International Conference on Software Maintenance and Evolution, September 2021. *IEEE TCSE Distinguished Paper Award*.
3. Ruyi Ji, Yican Sun, Yingfei Xiong, Zhenjiang Hu. *Guiding Dynamic Programming via Structural Probability for Accelerating Programming by Example*. OOPSLA'20: Object-Oriented Programming, Systems, Languages, and Applications 2020, November 2020.
4. Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, Zhenjiang Hu. *Question Selection for Interactive Program Synthesis*. PLDI'20: 41st ACM-SIGPLAN Symposium on Programming Language Design and Implementation, June 2020.

期刊论文

1. Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, Abhik Roychoudhury. *Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction*. TOSEM: ACM Transactions on Software Engineering and Methodology, August 2020.