自动化分析与

动态规划算法的改进

及其在

自然语言处理中的应用

by

Tim Vieira

提交给约翰斯·霍普金斯大学的博士论文，符合哲学博士学位的要求。

巴尔的摩，马里兰州

2023年7月

© 2023年 Tim Vieira 保留所有权利

摘要

本论文开发了一套系统，用于自动分析和改进动态程序，例如数十年来推动自然语言处理以及更广泛的计算机科学领域取得进步的程序。找到一个正确且具有最优渐近运行时间的程序，可能并非直觉所能及，而且耗时费力、容易出错。本论文旨在将这一繁琐过程自动化。为此，我们开发了一种基于

1. 一种名为 Dyna 的高级领域专用语言，用于简洁地描述动态程序

2. 一个用于高效执行这些程序的通用求解器

3. 一个静态分析系统，可进行类型分析和最坏情况时间/空间复杂度分析

4. 一组丰富的程序保义变换，将众多作者在文献中加速算法时反复获得的见解系统化

5. 一个 用于识别一系列优良变换的架构算法 t
se

在给定初始正确程序的情况下，降低运行时复杂度。我们表明，在实践中，自动化搜索——类似于人类程序员进行的思维搜索——能够显著改进初始程序。从实证结果来看，自然语言处理文献中描述的许多加速都可以由我们的系统自动发现。我们提供了一个可免费使用的原型系统，地址为
https://github.com/timvieira/dyna-pi。

主要审阅人兼导师： Jason Eisner(约翰斯·霍普金斯大学)**次要审阅人：** David Chiang (圣母大学)Scott Smith(约翰斯·霍普金斯大学)

致谢

学术

我由衷感谢我的导师 Jason Eisner，感谢他接纳我成为他的学生，并多年来耐心地指导我。我学到了很多关于如何教学、演讲、写作、思考、把细节处理妥当——以及如何乐在其中！多亏了 Jason，我的博士学习经历真正带来了深刻的转变。

我要感谢 Ryan Cotterell，他在我最后两年里非正式地担任了我的第二导师。若没有 Ryan 的支撑集、知识、耐心、鼓励和真知灼见，我就不可能完成这篇博士论文。

我要感谢 Wes Filardo，他教会了我这篇论文背后大量的基础知识。能坐在这样一位学识渊博、慷慨待人的人身旁，让我的博士学习经历更有价值，也更加愉快。

我要感谢我的委员会成员——Jason Eisner、David Chiang 和 Scott Smith，他们耐心地审阅了这份冗长的手稿，并提供了有用且富有见地的反馈。他们的反馈极大地完善了这份手稿。所有错误均由我本人负责。

我要感谢 Andrew McCallum 和 Dan Roth 聘用我担任研究程序员，这让我走上了攻读博士学位的道路。Andrew 甚至还为我引荐了 Jason Eisner，这一引荐至关重要！

我想感谢 Gerald DeJong、Jeff Erickson、Viraj Kumar 和 Cinda Heeren 在我本科期间给予我的支持。正是 Viraj 让我第一次有信心去读研究生院——我对此非常感激。

如果没有我有幸加入的这个卓越的学术共同体，我的博士学习经历就不会是现在这个样子。

- **合作者：** Afra Amini、Alexandra Butoi、Andreas Opedal、AnejSvete、Benjamin Dayan、Brian DuSell、Clara Meister、Clemente Pasti、He He、Jiarong Jiang、Jiawei Huang、Matthew Francis-Landau、Qi He、Ran Zmi- grod、Tiago Pimentel 和 Wes Filardo。

- **实验室同事（约翰斯·霍普金斯大学）：** Adam Teichert、Brian Lu、Chu-Cheng Lin、Darcey Riley、Dingquan Wang、Hongyuan Mei、Juneki Hong、Leo Du、Matthew Francis-Landau、Matt Gormley、Michael Paul、Nick Andrews、Ryan Cot- terell、Sabrina Mielke 和 WesFilardo。

- **实验室同事（马萨诸塞大学）：** Greg Druck、SameerSingh、Sebastian Riedel、Michael Wick、Kedar Bellare、Adam Saunders、Rachel Shorey、David Mimno、Pal- lika Kanani、Anton Bakalov、Karl Schultz、Limin Yao、Meagan Day、Laura Dietz、Hanna Wallach 和 Jason Naradowsky。

- **实验室同事（伊利诺伊大学厄巴纳-香槟分校）：** Alex Klementiev、Dan Goldwasser、Jeff Pasternack、Kevin Small、Lev Ratinov、Mark Sammons、Michael Connor、Ming-Wei

Chang、Nick Rizzolo、Quang Do、Rajhans Samdani、Subhro Roy、V.G.Vinod Vydiswaran、Vivek Srikumar 和 YuanchengTu。

- **更广泛的约翰斯·霍普金斯大学：**Ben Van Durme、Colin Lea、Elizabeth Salesky、Frank Ferraro、Hainan Xu、Huda Khayrallah、Juri Ganitkevitch、Kim Franklin、Keith Levin、Sanjeev Khudanpur、Tongfei Chen、Mark Dredze、Naomi Saphra、OliviaBuzek、Pushpendre Rastogi、Rachel Rudinger、RuthScally、Sheng Zhang、Travis Wolfe 和 Yanif Ahmad。

- **更广泛的社区：**AaronSchein、David Bellanger、David Smith、Hal Daumé、Kianté Brantley、Mathieu Blondel、Nikos Karampatziakis、Phil Thomas、Pierre-Luc Bacon、Sasha Rush 和 Vlad Niculae。

非学术

我想感谢我的父母 Marcia 和 Terry Vieira，感谢他们无条件的爱与支持，感谢他们始终把我的教育放在首位，也感谢他们为送我上私立学校和外州大学所做的牺牲。我无法奢求更好的父母。谢谢你们，妈妈和爸爸！

我要感谢我了不起的朋友和家人，感谢他们给予我的爱、支撑和欢乐：Hanna Wallach、Maia Papaya、Marcia Vieira、Terry Vieira、Tamara Vieira、Roseli Fabbri、Rachael Wallach von Portheim、Luna Petunia、Robin Wallach、Rob Wallach、Parth Naik、Ryan Cotterell、Boris U manov、Alex O'Neill、Keanu、Dan Muriello、Jason Naradowsky、Ian Baran、Dan Schreiber,

Abbie Jacobs、Aaron Schein、Clint Adams、Hal Daumé 和 Amber Boydstun。

最重要、最衷心的感谢要献给我的伴侣 Hanna Wallach。她陪伴我走过了整个博士学位历程，甚至还帮我写申请文书！若没有她的爱、鼓励、慷慨、耐心和支撑集，我不可能走到毕业的终点。谢谢你，Hanna。

先前发表作品

本论文的部分内容已发表于以下文献：

• Tim Vieira、Matthew Francis-Landau 等（2017年）。《Dyna：面向机器学习应用的自优化声明式语言》。发表于：*ACM SIGPLAN* 机器学习与编程语言研讨会论文集），第 8–17 页。ISBN：978-1-4503-5071-6。统一资源定位符：http://cs.jhu.edu/~jason/papers/#vieira-et-al-2017

• Tim Vieira、Ryan Cotterell 等人（2021年）。《搜索更高效的动态程序》。发表于：计算语言学协会论文集。统一资源定位符：https://arxiv.org/abs/2109.06966

目录

摘要 ii

致谢 v

先前发表作品 ix

表目录 xvi

图目录 xvii

算法列表 xviii

1 引言 1

2 面向动态规划的领域专用语言 9

3 符号表示、背景与实用工具 17

3.1 程序的内部表示.18

3.1.1 记号与表示 . 18

3.1.2 查找与更新表示. 22

3.1.2.1 隐式结果流数据结构 23

3.1.2.2 查找 (p[x]).243.1.2.3 更新 (p[x] ⊕= ...).29

3.2 一般数学.31

3.2.1 序关系.323.2.2 函数.343.2.3 图.353.2.4 抽象代数.38

3.3 半环：动态规划的强大抽象 . . 42

4 Dyna 程序语义 54

4.1 所有推导的总值.554.2 不动点近似.604.3 不动点迭代.634.4 收敛理论.65

5 推理算法 71

5.1 不动点迭代算法.73

5.1.1 朴素不动点迭代.735.1.2 半朴素不动点迭代.755.1.3 优先级不动点迭代.805.1.4 优先级函数.83

5.2 牛顿法.875.3 专用右线性求解器.925.4 混合算法.96

6 程序分析 **100**

6.1 类型分析.1026.1.1 形式化.1026.1.2 抽象前向链接.1066.1.2.1 类型推断算法.1076.1.2.2 松弛.1086.1.2.3 约束传播.1116.1.2.4 移除冗余简单类型 1146.1.2.5 终止.1156.2 基于类型的程序特化.120

6.2.1 为有用项推断类型.1226.2.2 特化变换.1236.2.3 缩写变换.125

6.3 自动空间与时间分析.129

6.3.1 空间分析.1296.3.2 时间分析.133

6.4 应用.137

7 程序变换 142

7.1 非正式介绍.1447.2 保语义变换.1527.3 定义引入变换.1567.4 折叠与展开变换.160

7.4.1 基本折叠与展开.161

7.4.2 广义折叠与展开 . 163

7.4.2.1 形式化定义.1667.4.2.2 挑战.1687.4.2.3 是什么导致部分正确性? . . . . . 1697.4.2.4 正确性的实用充分条件 . . 175

7.5 提升变换.186

7.5.1 推测变换.1867.5.2 左角变换.1937.5.3 示例.196

7.6 讨论 . 201

8 搜索更快的动态程序 206

8.1 图搜索问题.207

8.1.1 优先搜索算法.2088.1.2 搜索空间细节.210

8.2 代价函数.214

8.2.1 细粒度运行时间界.2178.2.2 粗粒度运行时间界.219

第8卷第3期 实验.222

9 相关工作与未来工作 227

9.1 相关工作.2279.2 未来工作.232

10 结论 **236**

A 推理算法 **238**

B 程序变换 **241**

B.1 强等价.242B.2 折叠与展开变换的正确性.243B.3 推测.264

C 模式匹配算法 266

C.1 合一.266C.2 子乘积枚举.268C.2.1 交换子乘积.268C.2.2 不可交换子乘积.269

D Dyna 2 **270**

E 实验所用程序 274

E.1 链（chain-05、chain-10、bad-chain-05、bad-chain-10）. . . . . 275E.2 投射依存句法分析.276E.2.1 双词汇无标注（bilexical-unlabeled）.276E.2.2 双词汇有标注（双词汇标注）.277E.2.3 弧先行（弧优先）.277E.2.4 分裂式双词汇依存句法分析.278E.2.4.1 Johnson（2007）版本（split-head-J）. 278

E.2.4.2 Eisner 和 Blatz（2007）的版本（split-head-EB）278

E.3 成分句法分析.279E.3.1 CKY（cky3）.279E.3.2 带四元产生式的 CKY（cky4）.279E.3.3 带固定文法的 CKY（cky+语法）. . . . . 280E.3.4 张量分解分析器（tensor-decomp-parser）. . . 281E.4 反转传导语法（反向转写文法）.281E.5 编辑距离（编辑）.282E.6 Bar-Hillel 构造（bar-hillel）.282E.7 线性链条件随机场下的期望（链期望）.283E.8 偶数和奇数 Peano 数的交集（even-odd-peano）. 284E.9 隐马尔可夫模型（隐马尔可夫模型）.285E.10 半马尔可夫模型（半马尔可夫）.285E.11 图中的最短路径.286E.11.1 显式路径（path-list）.286E.11.2 起始路径（路径起点）.286E.12 带显式栈的下推自动机（explicit-pda）. . . . 287

引用 **290**

索引 **323**

表目录

8.1 § 中所述基准测试上的搜索式程序优化器的实验演示E.
224

图目录

3.1 序关系分类.34

5.1 推导的维度示意图（定义 29）。..... 89

8.1 程序优化图搜索问题的图示。本图所用程序为 CKY（示例 6）。节点是 Dyna 程序（§2）。“起始”所指向的节点表示用户的程序。边是程序变换（§7）。代价由程序分析得出（§6）。搜索图中仅显示了节点和边的一个极小子集。虚线无标签出边表示我们为减少图示杂乱而未在图示中详述的其他变换. 20

8

算法列表

1 unify_implicitly：隐式合一算法.252 $\mathfrak{p}[x]$：求交查询 x 和程序 $\mathfrak{p}$ 的规则。返回一个隐式结果流（即一个迭代器，它会改变 x 的变量绑定并发出规则体）. 26 3 内置关系分派.274 $\beta_1 \circ \beta_2$：连接结果流 β_1 和 β_2 的算法。. . . . . 28 5 sccs(incoming, targets)：按拓扑序枚举有向图的强连通分量（强连通分量；定义12）。37 6 naive：通过（朴素）不动点迭代 solve 程序 $\mathfrak{p}$ 。. . . . . 74 7 seminaive：通过半朴素不动点迭代 solve 程序 $\mathfrak{p}$ 。. . . . 76 8 agenda：通过基于议程的不动点迭代 solve 程序 $\mathfrak{p}$ 。. . . . 81 9 support：估计 $\mathfrak{p}$的支撑集上界。. . . . . 84 10 incoming and outgoing：在给定其估计支持度 $\mathfrak{m}$的情况下，惰性枚举程序 $\mathfrak{p}$ 的依赖图中的入弧和出弧。. 85 11 newton：通过牛顿法 solve 程序 $\mathfrak{p}$ .90

12 solve_linear：通过归约到矩阵星运算（定义 21）来求解右线性程序 $\mathfrak{p}$ .95 13 solve：使用混合算法求解程序 $\mathfrak{p}$ .98 14 abstract_fc：针对类型程序 $\mathfrak{p}$的抽象前向链接算法。此过程假定输入类型已包含在 $\mathfrak{p}$中。108 15 简单类型的子类型、交集和合并算法。 . . . 114 16 useful：对于给定的程序 $\mathfrak{p}$、输入类型 $\mathfrak{D}$和输出类型 $\mathfrak{D}$，定义新的项集合$\widehat{\cdot}$，其中$\widehat{x}$为真表示 x可能有用。 . . 122 17 特化：将程序 $\mathfrak{p}$ 特化为类型 $\widehat{\mathfrak{m}}$.124 18 缩写：根据类型 $\widehat{\mathfrak{m}}$缩写程序 $\mathfrak{p}$ 。 . . . . . 126 19 运行时间：给定导出类型 $\mathfrak{m}$，推断 $\mathfrak{p}$ 的运行时界。 . . . . 136 20 展开：应用基本展开变换（定义 33）。 . . . . 162 21 折叠：应用基本折叠变换（定义 34）。 . . . . . 163 22 展开：应用广义展开变换（定义 35）。 166 23 折叠：应用广义折叠变换（定义 36）。 . . . 167 24 $\mathfrak{p}^{(s)}$ unfold $\xrightarrow[r\to\mathcal{S}]{d\ j}\ \mathfrak{p}^{(t)}$的推导映射.173 25 $\mathfrak{p}^{(s)}$ fold $\xrightarrow[\mathcal{S}\to r]{d\ j}\ \mathfrak{p}^{(t)}$的推导映射.174 26 折叠–展开变换序列安全性测试。假设输入为一个初始参照程序 $\mathfrak{p}^{(0)}$，其规则度量为 $\psi_0(\cdot)$。 . 185 27 推测：应用推测变换（定义 41）。 191 28 not-matches：不匹配关系的实现。 . . . . . 192 29 leftcorner：应用左角变换（定义 42）。 . . 195

30 搜索：给定图搜索问题的算法 $\langle \mathcal{S},\ \mathcal{A},\ s_0,\ \mathcal{T},$ 转换，余弦 t$\rangle$.

需要指定搜索策略 π，以选择要展开的候选状态。我们还要求指定搜索预算，例如搜索所花总执行时间的上限。 . . . . . 209 31 $\mathfrak{p}^{(s)}$ 展开 $\xrightarrow[r\to\mathcal{S}]{d\ j}\mathfrak{p}^{(t)}$的逆推导映射.251 32 $\mathfrak{p}^{(s)}$ 折叠 $\xrightarrow[\mathcal{S}\to r]{d\ j}\mathfrak{p}^{(t)}$的逆推导映射.251 33 $\mathfrak{p}^{(s)}$ 推测 $\xrightarrow{z\ \sigma}\mathfrak{p}^{(t)}$的推导映射.265 34 合一：合一算法。对于给定的可能含非基项 x 和 y，求解任何形如 $\theta(x) = \theta(y)$ 的方程，其中 θ .267 35 覆盖：单向合一算法。对于给定的可能含非基项 x 和 y，求解任何形如 $\theta(x) = y$ 的方程，其中 θ 。 . . . . 267 36 覆盖 _ 乘积：从子目标集合 $x_1 \circ \cdots \circ x_N$ 到另一个子目标集合 $y_1 \circ \cdots \circ y_M$，惰性枚举交换匹配。每次迭代都是一个替换映射与索引到索引对齐组成的有序对.269

37 覆盖_乘积：从子目标集合中惰性枚举不可交换匹配 $x_1 \circ \cdots \circ x_N$ 到另一个子目标集合$y_1 \circ \cdots \circ y_M$。每次迭代都是一个替换映射与匹配起始位置组成的有序对.269

第1章

引言

自然语言处理（NLP）中的算法研究，很大程度上一直专注于针对该领域中出现的组合问题开发动态规划解。这些算法多年来已被引入众多语言形式体系，例如

- 有限状态转换（例如，Mohri（1997）、Eisner（2002））,
- 上下文无关句法分析（例如，Stolcke（1995）、Goodman（1999））,
- 依存句法分析（例如，Eisner（1996）、Koo 和 Collins（2010））
- 弱上下文相关句法分析（例如，Vijay-Shanker 和 Weir（1989年）、Vijay-Shanker 和 Weir（1990）、Kuhlmann、Satta 和 Jonsson（2018））。

近年来，深度结构化预测通常使用相同的算法，并采用在结构上进行分解的神经评分函数。[1]

[1]例如，Durrett 和 Klein（2015）、Rastogi 等人（2016）、Lee 等人（2016）、Dozat 和 Manning（2017年）、Stern 等人（2017年）、Kim 等人（2017年）、Hong 和 Huang（2018）、吴、Shapiro 等人（2018）、吴和 Cotterell（2019年）、Qi 等人（2020年）以及 Rush（2020年）。

设计高效算法颇具挑战性。设计并实现尽可能高效的动态程序颇具挑战性，也容易出错。当一种新问题的动态规划算法首次在文献中提出时，其运行时间有时并非最优——随着时间推移，通常会发表更快的版本（见例 1）。事实上，在计算机科学领域，先引入第一个算法，随后再不断寻找改进方案，是一种普遍现象。我们希望通过自动化来促进这一过程。

示例1。请看以下已发表动态程序的实例[2]，其运行时间界后来得到了改进。在下面的示例中，n 表示输入句子的长度。

- 投射依存句法分析：*Collins*（*1996*）提出了一种 $\mathcal{O}(n^5)$ 算法，后来由 *Eisner* 和 *Satta*（*1999a*）将其提速至 $\mathcal{O}(n^4)$。

- 分裂头部因子化依存句法分析：朴素实现的运行时间为 $\mathcal{O}(n^5)$；经过一些努力，可以推导出一种 $\mathcal{O}(n^3)$算法（*Eisner*，*1996*；*Johnson*，*2007*；*Eisner and Blatz*，*2007*）。

- 线性索引文法句法分析：$\mathcal{O}(n^7)$ 由 *Vijay-Shanker* 和 *Weir*（*1989*）提出，后由 $\mathcal{O}(n^6)$*Vijay-Shanker* 和 *Weir*（*1993*）加速。

- 词汇化树连接文法分析：$\mathcal{O}(n^8)$ *Vijay-Shankar* 和 *Joshi*（*1985年*）提出的方法，后来由 *Eisner* 和 *Satta*（*2000*）将其提速至 $\mathcal{O}(n^7)$。

[2]我们在 Eisner and Blatz（2007）和 Gildea（2011年的研究中了解到许多此类示例；其中还可找到进一步的讨论。

• 反向转换语法： $\mathcal{O}(n^7)$ 由吴（*1996*）提出，后加速至 $\mathcal{O}(n^6)$ *Huang*、*Zhang* 等人（*2005*年）。

• *CKY*句法分析（*Cocke* 和 *Schwartz*，*1970*；*Younger*，*1967*；*Kasami*，*1965*）通常以次优 $\mathcal{O}(k^3n^3)$ 形式呈现，但可以加速至$\mathcal{O}(k^2n^3 + k^3n^2)$ （*Lange* 和 *Leiß*，*2009*；*Eisner and Blatz*，*2007*），其中 k 是语法中非终结符的数量。

• *Tomita* 的上下文无关句法分析算法（*1985*年）运行时间为 $\mathcal{O}(n^{\rho+1})$ 其中 ρ 是语法中上下文无关产生式右侧的最长长度（*Johnson*，*1989*年）。不过，通过将产生式规则二元化，可以使其运行时间达到 $\mathcal{O}(n^3)$ 。

> **核心问题**。本博士论文探讨以下问题：
>
> 我们能否自动发现这些更快的算法?

动态规划的统一符号表示。我们的方法始于 Goodman（1999）提出的一项观察：动态规划算法可以视为在半环加权演绎系统中执行推理。他们的工作建立在（非加权）演绎系统的悠久研究历史之上，我们将在 §9.1中回顾这一历史。Eisner、Goldlust 等人（2005年）提供了一种名为 Dyna 的程序设计语言，用于表达此类演绎系统，并配有一个能够生成快速推理代码的编译器。本论文将使用 Dyna 作为算法描述语言。我们将在 §2 中介绍该语言，并在 §4中给出其形式语义。

下面是一个 Dyna 程序示例，用于计算加权有向图中节点对之间所有路径的总权重。

```
1 路径(I,I) += 1.2 路径(I,K) += 边(I,J) *
路径(J,K).
```

这里，边的权重来自关系 `edge(I,J)`。例如，要声明从标记为`"a"` 的节点到标记为`"b"` 的节点的边权重为 13，可以写成 `edge("a","b") += 13`。由此，`path(I,K)` 项编码了图中所有节点之间的带权关系。具体而言，对于所有节点有序对 `I`,`K`：

$$\mathsf{path}(\mathtt{I},\mathtt{K}) = \underbrace{1[\mathtt{I}{=}\mathtt{K}]}_{\#1} + \sum_{\mathtt{J}} \underbrace{\mathsf{edge}(\mathtt{I},\mathtt{J}) \cdot \mathsf{path}(\mathtt{J},\mathtt{K})}_{\#2} \tag{1.1}$$

这里，变量 `J` 遍历节点集合。之所以在（`J`中求和（1.1），而 `I` 和 `K` 不参与求和，是因为它是 #2右侧的局部变量。读者可能会认出（1.1）是图中节点对之间所有路径总权重的数学递推式。要将这一数学规范转化为高效的计算过程，还需要明确若干实现细节。在此例中，我们需要选择用于存储和索引边关系与路径关系的数据结构——但更重要的是，需要 *solve* 递归方程以推导总权重。在 §5中，我们将介绍通用过程，用于将（以 Dyna 编写的）数学递推式转化为高效算法。

自动化分析。 我们系统的一个关键组成部分是静态分析：无需针对特定输入运行程序，就能对其进行推理。

此类分析包括类型分析、运行时复杂度分析和空间复杂度分析（Cousot 和 Cousot，1977年；McAllester，2002年）。这些分析对于理解未知输入数据下可能出现的程序行为至关重要。[3] 自动化程序分析有助于程序员，因为手动程序分析可能既繁琐又容易出错。在自然语言处理领域，自动分析动态程序（例如 Dyna 中指定的程序）有望帮助自然语言处理算法研究人员正确、快速地分析他们创建的程序。事实上，自然语言处理文献中就有一些例子，说明此类自动化分析本可以发挥作用。

示例2。考虑以下来自自然语言处理文献的示例 *：*

• *Kuhlmann*、*Gómez-Rodríguez*等（*2011*年，*§5.4*）发现了其表格依存句法分析算法的一个令人惊讶的性质。他们最初的朴素分析给出的时间复杂度为 $\mathcal{O}(n^5)$，空间复杂度为 $\mathcal{O}(n^3)$。尽管如此，他们仍能将这一分析收紧为时间复杂度 $\mathcal{O}(n^3)$ 和空间复杂度 $\mathcal{O}(n^2)$，这是在注意到其转移系统所构建的对象总是具有两个相等的变量（这一点可以归纳证明）之后实现的。随后，他们修订了程序表述，以消除该冗余变量。

• *Huang*和*Sagae*（*2010*年）推导出一种射影依存句法分析算法，其所述复杂度为 $\mathcal{O}(n^7)$，后来被发现实际上是$\mathcal{O}(n^6)$，根据*Shi*等人（*2017*年）。[4]

[3]虽然我们在本论文中并未探讨这一点，但我们也指出，静态程序分析对于编译高效代码至关重要。例如，类型推断使编译器能够合成用于表示内部对象的高效数据结构，以及用于对它们进行模式匹配的高效代码。[4]这一点在Shi等人（2017年）中有所陈述，但没有证明。我们所能找到的唯一解释在

我们的自动分析系统给出运行时间与空间复杂度的大$\mathcal{O}$ 界（§6.3）。这些界是通过为给定规则中的每个变量推断类型，并利用每种类型来界定该规则在执行期间触发方式的数量而得出的（§6.3）。从高层来看，要将此技术应用于路径程序，我们首先看到变量I、J和K是节点。假设节点集合的大小为 n。规则path(I,K) += edge(I,J) * path(J,K)在推理期间触发的方式数为 $\mathcal{O}(n^3)$。该技术的基础由McAllester（2002）的元定理给出。许多作者[5] 手动应用该元定理来分析其递归算法的运行时间。我们的运行时分析器（§6.3）自动应用该元定理，以推导执行给定程序时最坏情况下空间与运行时复杂度的紧界（使用§ 5中的执行方法）。

系统化算法加速。示例中提到的所有运行时改进1都可以视为针对 Dyna 程序的源到源程序转换（Eisner and Blatz，2007）；我们将在 §7 中介绍这些变换。[6] 在动态规划中，可以利用程序变换来

文献见 Hong和Huang（2018）。在该研究中，作者指出，某个归约规则（来自一个密切相关的模型）中的一个变量似乎可以取 $\mathcal{O}(n)$个值，但它函数依赖于其他变量。这意味着给定作用域内其他变量的值后，该变量至多只能取一个值；因此， n 的运行时间中有一个因子变成了常量。这一观察似乎也适用于Huang和Sagae（2010）的模型。[5]明确采用这种分析技术的自然语言处理论文包括以下文献：Gildea（2011）、Nederhof和Satta（2011）、Gilroy等（2017）、Melamed（2003）、Kuhlmann（2013）、Nederhof和Sánchez-Sáez（2011）、Büchse等（2011）、Lopez（2009）、Eisner 和 Blatz（2007）、Huang和Sagae（2010），以及Kuhlmann、Gómez-Rodríguez等（2011）。[6]在一篇开创性论文中，Burstall和Darlington（1977）提出了通过源到源变换来合成更快、更好的（函数式）程序这一思想：给定一个简单、清晰且最重要的是正确的初始程序，我们可以通过改变其递归结构反复对其进行变换，最终得到改进后的程序。后来，Tamaki 和 Sato（1984）将这一技术扩展到了逻辑编程。

推导出运行时间更快的算法。这些变换将一个程序映射为另一个含义相同的程序（也就是说，在输入相同的情况下，会产生相同的输出），但运行时间可能不同。下面给出了一个这样的加速示例。

假设我们只想计算路径的总权重，这些路径从一个节点集合 起始(I) 到另一个节点集合 停止(K)。下面的程序先计算所有有序对的总权重，然后对一个子集求和：

```
3 目标  +=起始(I) * 路径(I,K) * 停止(K). 4 路径
(I,I)  +=  1. 5 路径(I,K)  +=边(I,J) * 路径(J,K).
```

然而，该程序的效率并非尽可能高。通过利用加法和乘法的分配性质，我们可以在递推中更早地对可能的停止节点求和：

```
6 目标  +=起始(I) * pathstop(I).7 pathstop(I) +=
停止(I).8 pathstop(I) += 边(I,J) *pathstop(J).
```

这里 pathstop(I) 计算从节点 I 到停止中某个节点的所有路径的总权重。该程序的运行时间为 $\mathcal{O}(n^2)$。在§7中，我们提供了一个能够恢复这一加速的程序到程序变换集合。

通过自动化变换搜索改进程序。这项工作展示了如何搜索程序变换序列，以自动发现更快的算法，从而自动完成自然语言处理算法研究者的工作。我们将程序优化视为在初始程序的变换版本中进行搜索。我们介绍如何利用搜索快速重新发现示例1中列出的许多已知优化。程序优化器将论文的多个方面联系

起来：可能程序空间（§2）、作为程序运行时间代理指标的代价函数（§6），以及连接语义等价程序的有向边集合（§7）。我们的搜索从初始程序开始，并通过遍历有向边，寻找一个代价较低的等价程序。在§8中，我们回顾了将用于此目的的搜索技术。我们在§第8卷第3期进行了实验演示。

总结。 我们通过自然语言处理领域的许多示例阐明了我们的方法，这些示例表明，设计和分析高效的动态程序颇具挑战性。我们将构建一个框架，把系统地分析和改进动态程序这一问题分解为以下几个组成部分：

1. 一种领域专用语言，用于以逻辑方式指定算法（§2 和 §4），以及用于执行这些算法的通用机制（§3.1 和 §5）

2. 一个系统，用于自动分析动态程序的空间/运行时间复杂度，以及进行类型分析（§6）

3. 一组丰富的保义程序变换（§7）

4. 一种搜索算法，用于找出能够降低运行时间复杂度的一系列变换（§8）。

我们提供了一个强大的系统，能够实现示例1中展示的加速效果。[7] 我们在第8卷第3期中对该系统进行了实验演示。原型实现可从以下地址免费获取：
`https://github.com/timvieira/dyna-pi`。

[7]我们的实验涵盖了示例1 中的所有加速情形， 但不包括线性索引文法和词汇化树邻接文法解析器。实验中还包括许多其他有趣的加速情形，这里没有提及。

第2章

一种用于动态规划的领域专用语言

Dyna 是一种高级领域专用语言，可用于简洁地将动态程序表示为递推关系。Eisner、Goldlust 等人最早于（2005年）介绍了这种语言，并展示了它作为一种符号表示和软件库的用途，可用于快速开发多种流行语言结构模型（如标注、句法分析和翻译）的高效算法，也可用于开发许多可通过动态规划求解的组合优化问题（如最短路径、背包问题和序列比对）的高效算法。

我们对Dyna语言的阐述将以一系列简单且希望读者熟悉的计算展开——在必要或方便时引入概念与术语。

Exam例 3. 加权有向图中长度为 *4* 的路径的总权重 *:*

```
9 总计 += w(Y1,Y2) *w(Y2,Y3) *w(Y3,Y4) * w(Y4,Y5)。
```

这个程序用输入项 w(...) 来定义派生项 total 的值。该程序假设我们已经通过为 w(...) 项声明值，对给定图中的边集合进行了编码。例如，要声明从标签为 "a" 的节点到标签为 "b" 的节点存在一条权重为 13 的边，我们写作 w("a","b") += 13。total 的值为 $\sum_{Y_1} \cdots \sum_{Y_5} w(Y_1,Y_2) \cdot w(Y_2,Y_3) \cdot w(Y_3,Y_4) \cdot w(Y_4,Y_5)$。

Dyna规则可以是递归的：

示例 4. 图中的最小成本路径。

```
10 目标 最小= 起始(I) + β(I)。
11 β(J) 最小=停止(J)。12 β(I) 最
小=w(I,J) + β(J)。
```

该程序定义 β(I) 为从节点 I 到任意终端节点J的图中最小成本路径的代价。这里必须先将 stop(J) 定义为在J 为终端节点时于该节点停止的代价（否则应保持未定义）。此外，w(I,J) 必须先定义为从I到J 的有向边的代价（如果不存在这样的边，则应保持未定义）。

逻辑与控制的分离。 示例 4 描述了在加权有向图中寻找单目标最短路径的问题。然而，具体的实现细节被抽象掉了。因此，许多著名的单目标最短路径算法，例如 Bellman–Ford–Moore 算法（Bellman，1958年；Ford，1956；Moore，1959年）和 Dijkstra 算法（Dijkstra，1959年），都表示为同一个 Dyna 程序（示例 4），因为它们基于相同的路径递归分解。它们之间的差异

比基本 Dyna 表示法所能表达的层次更低，因为它们本质上只是求值顺序上的差异（§5.1.3）。Dyna 表示法的优点之一，是将算法的数学规范与该算法的执行细节（例如数据结构、控制流、执行顺序）分离开来。这一思想通常称为逻辑与控制的分离（Kowalski，1979年）。

注意，如果我们将方程从 ⟨min +⟩ 改为 ⟨max，*⟩，就得到了用于寻找图中从起始集合中的节点到停止集合中的节点之间最可能路径的维特比算法（Viterbi，1967年）。路径的概率由边概率（或权重）的乘积给出，w(...)：

示例 5。 寻找图中最可能的路径。

```
13 目标max=起始(J)*  β(J)。14  β
(J)max=停止(J)。15  β(J)max= β
(I)* w(I,J)。
```

示例 4 与示例5 之间的连接之处很有意思：我们保留了相同的递推，但改变了要计算的聚合量。两个程序定义了相同的推导集合，我们可以将其解释为图中的路径。路径的权重及其解释方式有所不同。在示例 4中，路径具有代价；代价通过加法进行扩展，并通过 min 进行聚合。在示例 5中，路径具有概率；概率通过乘法进行扩展，并通过加法进行聚合。稍后我们将回到扩展运算和聚合运算。

接下来，我们给出一个自然语言处理的基本算法。

例6. 加权上下文无关句法分析和*CKY*（*Cocke* 和 *Schwartz*，*1970*；*Younger*，*1967年*；*Kasami*，*1965年*），或者更准确地说，内向算法（*Baker*，*1979年*；*Jelinek*，*1985年*）：[8]

```
16 β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K).17 β
(X,I,K) += γ(X,Y) * β(Y,I,K).18 β(X,I,K) += γ(X,Y)
* 单词(Y,I,K). 19 目标 += β(s,0,N) * 长度(N).
```

这些 γ 项的值应定义为相应上下文无关文法规则的权重：例如，项 $\gamma(\texttt{s},\texttt{np},\texttt{vp}) = 0.7$ 编码产生式 $\texttt{s} \xrightarrow{0.7} \texttt{np vp}$。此外，项 单词(X,I,K) 应为 1，如果输入单词X出现在位置 K（输入句子中），并且 $\texttt{I} = \texttt{K} - 1$，否则应保持未定义。然后，对于任意非终结符X 以及跨越位置（I,K] 的输入句子中的任意子串，项 β(X,I,K) 表示该子串从 X 出发的所有文法推导的总权重。

语法与术语。Dyna 程序 p 用于为项定义值。因此，该程序定义了一个从项到值的函数；我们将项 x 的值记为 $[\![x]\!]$。我们将 $[\![\cdot]\!]$ 称为赋值函数；其形式定义留待定义 22。每个项由 Herbrand 宇宙 $\mathbb{H}$中的一个元素命名——即一个（可能嵌套的）带标签元组，例如 f(g(z,h(3)))，它是一个标签为 f 的一元组，其唯一元素是一个标签为 g 的二元组。$\mathbb{H}$ 的元素称为基项，因为它们不含变量。程序p 本身是形式为 $x \oplus\!= y_1 \circ \cdots \circ y_K$的规则[9] 的无序集合。我们将$x$ 称为头部，将 y_1、……、 y_K 称为规则体。规则体中的每个 y_k 称为一个

[8]如果读者不熟悉上下文无关句法分析，我们推荐 Jurafsky 和 Martin（2020年，第12–13章）。[9] 请注意，该集合可能包含重复项（§3.1.1）。

子目标.

$\oplus$ 和[10] 这两种运算对来自集合 $\mathbb{V}$ 的值进行操作，并且可以是构成半环（§3.3）的任意一对运算。令 0 和 1 分别表示该半环的加法单位元和乘法单位元。如果某个项的值未被定义或不含 $\oplus$加项，则这等价于说它的值为 0，这意味着它不能影响任何其他项的值；实际上，它并不存在。常见的半环包括总权重 $\langle \mathbb{R}_{\geq 0}, +, * \rangle$，布尔$\langle \{\text{false}, \text{true}\}, \vee, \wedge \rangle$，最小成本 $\langle \mathbb{R} \cup \{\infty\}, \min, + \rangle$，以及维特比 $\langle \mathbb{R}_{\geq 0}, \max, * \rangle$。[11]半环的分配律引出一种重要的解释：给定项的值是在程序规则下对其所有推导（即证明树）的某种求和。[12] 该推导的值是其公理的值的乘积。[13] 在本文中，我们假设程序中的所有规则使用同一个半环。[14]

规则无需指定具体的项。规则中的项可以包含**变量**——以大写字母表示，例如 X——并在 $\mathbb{H}$ 上作全称量化。这使它们能够与许多项进行模式匹配。令 $X_1, \ldots, X_N$ 表示规则 **R** 中不同的变量。**R** 的一个**基化**是一个

[10]用于表示半环运算的符号表示因作者而异。许多作者（Lehmann，1977年；Kuich，1997年；Green、Karvounarakis 等人，2007；Esparza等人，2007）使用 $\langle +, \cdot, 0, 1 \rangle$，这会造成混淆，因为这些运算在同一文本中常常也被用来表示其通常含义。在自然语言处理文献中，最常见的做法是$\langle \oplus, \otimes, 0, 1 \rangle$，以避免对 $\langle +, \cdot, 0, 1 \rangle$ 的过度重载。我们选择使用 $\langle \oplus, \otimes, 0, 1 \rangle$，因为它们在视觉上更容易区分。[11]半环为计算自然语言处理模型中的许多常见量提供了一种优雅的抽象，例如期望值、协方差、梯度、熵以及 K-best（Goodman，1999；Huang，2008年；Eisner、Goldlust 等人，2005年；Huang 和 Chiang，2005年；Li 和 Eisner，2009年）。[12]精确的语义，包括推导集合的定义，在§4中给出。[13]我们在§4中形式化对推导求和的解释。[14]Dyna 2（Eisner 和 Filardo，2011）relax了这一限制。我们在§D中讨论关于 Dyna 2 的未来工作。

无变量规则 $\{\texttt{X}_1 \mapsto x_1, \ldots, \texttt{X}_N \mapsto x_N\}(\mathbf{R})$ (即，将x_i 代入 $\texttt{X}_i$ 在 $\mathbf{R}$中的所有副本，对每个 i，其中 $x_1, \ldots, x_N \in \mathbb{H}$。 $\Gamma(\mathbf{R})$ 表示**所有基例的集合**，即 $\mathbf{R}$ 的所有基例；在 $\mathfrak{p}$ 中加入一条非基规则，等价于加入它的所有基例。

Dyna语言允许逻辑**边条件**作用于规则，例如目标 `+= f(X) for X < 9`。这是语法糖，等价于 目标 `+= f(X) * lessthan(X,9)`，其中`lessthan` 约束的值由

$$[\![\texttt{lessthan(X,Y)}]\!] \stackrel{\text{def}}{=} 1 \textbf{ if } \texttt{X} < \texttt{Y} \textbf{ else } 0$$

约束`X < 9` 使用了**内置关系** `<`；Dyna 标准库支持许多其他常见的算术关系，例如

- $[\![\texttt{X}\text{是 }\texttt{Y+Z}]\!] \stackrel{\text{def}}{=} 1$ **如果** $\texttt{X} = \texttt{Y} + \texttt{Z}$ **否则** 0
- $[\![\texttt{X}\text{是 }\texttt{Y}^K]\!] \stackrel{\text{def}}{=} 1$ **如果** $\texttt{X} = \texttt{Y}^K$ **否则** 0

与由程序规则定义的项不同，内置关系是不透明的——它们并非由规则定义，而是由可执行的（非 Dyna）代码定义。

输入数据被表示为程序 $\mathfrak{D}$，由**公理**组成——即形如 $(\boldsymbol{x} \oplus\!= v)$ 其中 $v \in \mathbb{V}$ 的规则。例如，将最短路径程序视为 $\mathfrak{p}$，并将要在其上运行的特定图视为 $\mathfrak{D}$。

$\mathfrak{p} =$

```
% definition of shortest
% path from S to stop
β(S) min= w(S,S') + β(S').
β(S) min= stop(S).
```

$\mathfrak{D} =$

```
% define graph (w, stop)
w(bal,phl) min= 3.
w(phl,nyc) min= 1.
...
stop(nyc) min= 0.
```

添加 th e $\mathfrak{D}$ 规则到 $\mathfrak{p}$ 结果在.....里面一个新程序 $\mathfrak{p}'$ 随后可以b e

被查询，正如我们将在……下面展示的那样。

查询程序。 用户与 Dyna 程序交互的方式是通过查询。一个**查询** 针对程序 `p` 指定为一个（可能非基的）项 x，而查询的**结果** 是一个**专用程序**，它对赋值函数的限制进行编码 $[\![\cdot]\!]$（定义 22），并将其限制在 x的可能基化范围内。下面从左到右展示了一个程序、一个查询及其查询结果的示例。

```
% 最短路径的定义
% 从 S 到 stop 的路径
β（S）最小值= w(S, S’) + β(S’)。
β（S）最小值= 停止（S）。
% 定义图 (w, stop)
w(bal,phl)最小值= 3。
w(phl,nyc)最小值= 1。
...
停止(nyc)最小值= 0。
```

查询 `β(S)`

```
β(bal) 最小值= 4。
β(phl) 最小= 1.
β(nyc) min= 0.
...
```

查询 `β(S)` 返回一个映射：形式为 `β(S)` 的每一项将 `S` 映射到其到任意停止 节点的最短路径的代价。如上面的示例输出所示，Dyna 将查询的结果表示为特化程序，即仅包含特定查询答案的程序。作为查询答案的特化程序被限制为具有如下性质：其所有右部均为常量。[15]

请注意，查询的结果可能是无限大的赋值。为了简洁地表示某些无限赋值，我们允许结果中的规则头部包含自由变量。例如，

[15]可以 relax 此条件，以允许规则的右侧包含内置关系。例如，`f(X) += 3 for (1 is X mod 2)` 表示 `f(X)` 项的贡献为 3 如果`X` 是奇数。

```
42 f(X) += 7. % 自由变量 43 f(3) += 11. ⟦f(X)⟧ = { 18 如果 X 7
                                                  = 3
                                                  否则
```

为了提高效率，我们尽可能对每个头部的**常量折叠**进行简化。例如，以下结果将替换为#42–43。

```
44 f(X) += 6.
45 f(Y) += 1.
46 f(3) += 9.
47 f(3) += 2.
```

总结

本章对 Dyna 语法进行了高层概述。我们介绍了 Dyna 用户应当熟悉的主要概念和术语。与程序的内部表示相关的概念将在§3中加以描述，推导和赋值映射的确切含义将在§ 4中说明，程序的执行方法将在§5中介绍。

第 3 章

符号表示、背景与实用工具

本章包含我们将在整篇论文中使用的背景材料。这些背景材料包括：

3.1 程序的内部表示.183.1.1 记号与表示.183.1.2 查找与更新表示.223.2 一般数学.313.2.1 序关系.323.2.2 函数.343.2.3 图.353.2.4 抽象代数.383.3 半环：动态规划的强大抽象 . . 42

3.1 程序的内部表示

本节介绍重要的符号表示和程序的内部表示。我们将介绍一种高效的数据结构，用于将查询结果表示为 结果流 (§3.1.2.1)。我们将开发一种表示抽象，使我们在操作规则时能够将其视为不含变量的形式（例如，在推理过程中（§5））。

3.1.1 记号与表示

现在我们将说明程序在内部是如何表示的。当用户编写如下程序时

```
48  f(X) += g(X,Y) * h(Y).
49  f(X) += 1.
```

它被表示为一个规则多重集。每条规则 R 的形式为 $x \oplus\!= y_1 \circ \cdots \circ y_K$。它在内部被表示为一个头部与一个体构成的有序对。体是一个子目标列表。令 head(R) 和 body(R) 分别表示规则 R 的头部和规则体。头部 x 以及每个子目标 y_k 均被表示为非基项。一个非基项[16] 是一个带标签元组，其中可能包含命名变量（用大写字母表示，例如示例中的 X 和 Y）。令 vars(·) 为一个函数，返回非基项中所含自由变量的集合，例如 vars(f(g(X),4,X)) $\mapsto$ {X}。变量可以在非基项之间共享，例如，X 在 # 48 的头部 f(X) 与子目标 g(X,Y) 之间共享，而 Y 在其两个子目标之间共享。

[16]当我们写“非基项”时，指的是“可能非基”的项，即允许为非基、但不一定包含自由变量的项。

替换 是一个函数 $\theta(\cdot)$ 从非基项到非基项；它由一个 **替换映射** θ 参数化，该映射将变量映射到非基项，例如 $\theta = \{\texttt{Y} \mapsto \texttt{g(X)}, \texttt{Z} \mapsto 4\}$。当我们应用替换映射 $\theta(x)$ 时，它会将箭头左侧变量的所有实例（$\mapsto$）替换为箭头右侧的内容。

合一。对非基项执行的主要操作是合一；我们的求解器（§ 5）会在内部执行这一操作以绑定变量。合一用于求解形如 $\theta(x) = \theta(x')$ 的方程，其中 x 和 x' 是非基项。此类约束可通过合一算法高效求解：$\theta \leftarrow \text{unify}(x, x')$（例如 Robinson（1965年）以及 Martelli 和 Montanari（1982年））。[17] 这里，unify 会返回替换映射 θ；如果不存在解，则返回 FAIL。例如，$\theta \leftarrow$ unify(f(Y,4), f(g(X),Z)) 会返回解 $\theta = \{$Y $\mapsto$ g(X), Z $\mapsto$ 4$\}$。[18] 将此替换映射应用于等式两边后，项便相等：θ(f(Y,4)) $=\theta$ (f(g(X),Z))。另一方面，θ(f(X,g(X))) $=\theta$(f(3,g(4))) 无解；因此，$\theta =$ FAIL。我们还提及形如 $\theta(x) = x'$ 的方程。这类约束出现在程序变换 §7 中，可以采用类似的方法，通过 $\theta \leftarrow$cover(x, x') 求解；伪代码见算法 35。

由于变量名称是全局的，我们经常需要重命名它们（X 在.....里面#48 应当不同于X 在.....里面#49。为此，我们使用函数 fresh，它将项、规则或整个程序中的所有变量重命名为新的名称，以避免不必要的命名冲突。例如，

[17]我们在§C.1.中给出了我们的合一算法的伪代码和实现细节。[18]我们允许 合一 将替换映射作为可选的第三个参数，从而使算法能够将多次合一调用的结果串联起来。

fresh(f(g(X),X))=f(g(X*),X*) 其中 X* 是一个新的变量名称。

解构。我们将使用解构，例如 $(x \oplus= y_1 \circ \cdots \circ y_K) \leftarrow$ R，将规则 R 拆分为其头部和可变长度体。它等价于 $x \leftarrow$ head(R), $K \leftarrow |$ body(R) $|$, $\langle y_1, \ldots, y_K \rangle \leftarrow$ body(R)。如果 $K = 0$，我们将体视为 1。我们可以写作 $f(x_1, ..., x_N) \leftarrow x$，将项解构为其参数，并将 f 绑定到其标签。例如， $f(x_1, ..., x_N) \leftarrow$ p(q(X),r(X)) 集合 $N \leftarrow 2$, $f \leftarrow$ p, $x_1 \leftarrow$ q(X), $x_2 \leftarrow$ r(X)。在解构表达式中，例如 $(_ \oplus= y_1 \circ \cdots \circ y_K) \leftarrow$ R，我们可以使用下划线 (_) 来匹配我们不希望命名的子表达式。

包。我们将使用包（也称为多重集）来表示规则（程序）的集合。字面包用成对的分隔符 $\wr \cdot \int$ 表示。设 α 和 β 为包。我们用 $\alpha \cup \beta$ 表示包并集，用 $\smallsetminus$ 表示包差。作为条件使用时，当且仅当 x 在 β 中至少出现一次， $x \in \beta$ 才为真。然而，在循环中使用时， $x \in \beta$ 会重复处理 x 的重复实例。我们将通过 i 这个“位置”来访问 β_i 中 β 的元素——尽管位置是任意的——这些位置只是对（可能重复的）元素所建立的一种规范线性序。

程序。 符号表示 $\mathfrak{p}[x] \oplus= y_1 \circ \cdots \circ y_K$ 将规则 $x \oplus= y_1 \circ \cdots \circ y_K$ 添加到程序 $\mathfrak{p}$ 中。我们使用符号表示 $\mathfrak{p}[x]$ 来获取集合，其中包含 $\mathfrak{p}$ 中（特化后的）规则体，这些规则的头部与子目标 x 统一。[19] 要计算两个子目标 x 和 x' 的乘积，我们可以写作 $\mathfrak{p}[x \circ x']$ 或 $\mathfrak{p}[x] \circ \mathfrak{p}[x']$。该符号表示可用于读取（规则体）和写入（新规则）

[19] We 可以选择将规则体与头部统一的替换映射配对 g.

通过类似 $\mathfrak{p}'[x] \oplus\!= \mathfrak{p}[y_1] \circ \cdots \circ \mathfrak{p}[y_K]$这样的表达式，在程序之间 $\mathfrak{p} \oplus \mathfrak{p}'$。我们用 $\mathfrak{p}$ 连接两个程序的规则： $\mathfrak{p}'$和 $\mathfrak{p}$。要创建一个空程序 $\mathfrak{p} \leftarrow \wr\!\int$，我们写作 。我们将在第 3.1.2 节更详细地解释所有这些符号表示。

定义整数区间的简写方式。 我们发现，以下符号表示在定义连续整数列表时很有用， i， $j \in \mathbb{Z}$:

$$
\begin{aligned}
[i{:}j) &\stackrel{\text{def}}{=} [\, i, i{+}1, \ldots, j{-}1 \quad] \\
[i{:}j] &\stackrel{\text{def}}{=} [\, i, i{+}1, \ldots, j{-}1, j \,] \\
(i{:}j) &\stackrel{\text{def}}{=} [\quad i{+}1, \ldots, j{-}1 \quad] \\
(i{:}j] &\stackrel{\text{def}}{=} [\quad i{+}1, \ldots, j{-}1, j \,]
\end{aligned}
$$

当未指定端点时，我们将其视为极值，例如，$[-3{:}) = [-3, -2, \ldots]$ 和 $[{:}10) = [\ldots, 9, 10]$.[20] 我们主要使用这种符号表示来索引规则体。

规则体。 我们使用以下符号表示简写来操作规则体。设 $\boldsymbol{\mu} = y_1 \circ \cdots \circ y_K$ 为子目标列表。设 $\boldsymbol{\beta}$ 为索引列表，则定义 $\boldsymbol{\mu}_{\boldsymbol{\beta}} \stackrel{\text{定义}}{=} [\, y_i \mid i \in \boldsymbol{\beta} \cap [1{:}\ K] \,]$。我们经常使用整数范围语法来表示 $\boldsymbol{\beta}$，例如，$\boldsymbol{\mu}_{[i:j)}$ 表示[i: j中的连续子目标范围。我们也可以写作 $\boldsymbol{y}_{[i:j]} \circ \boldsymbol{y}_{(j:k]} = \boldsymbol{y}_{[i:k]}$。类似地，$\boldsymbol{\mu}_{[i:j)} \circ \boldsymbol{\mu}_j \circ \boldsymbol{\mu}_{(j:k]} = \boldsymbol{\mu}_{[i:k]}$，以及$\boldsymbol{\mu}_{[:j)} \circ \boldsymbol{\mu}_j \circ \boldsymbol{\mu}_{(j:]} = \boldsymbol{\mu}$。

[20]在这些极端情况下，开（“(”或“)”）端点与闭（“[”或“]”）端点之间的区别无关紧要。

3.1.2 查找与更新表示法

在本节中，我们将介绍一种符号表示，大幅减少因显式传递替换映射以及迭代查询结果而产生的冗杂内容。左侧是我们将要介绍的符号表示，右侧是它所代表的代码。

$$\mathfrak{p}[\mathtt{a}(\mathtt{I},\mathtt{L})] \oplus= \mathfrak{m}[\mathtt{b}(\mathtt{I},\mathtt{J})] \circ \mathfrak{m}[\mathtt{c}(\mathtt{J},\mathtt{K})] \circ \mathfrak{m}[\mathtt{d}(\mathtt{K},\mathtt{L})] \quad \overset{\text{翻译}}{\Longrightarrow}$$

$$\begin{aligned}
&\textbf{for } \langle v_1, \boldsymbol{\theta}_1 \rangle \in \mathfrak{m}[\mathtt{b}(\mathtt{I},\mathtt{J})]: \\
&\quad \textbf{for } \langle v_2, \boldsymbol{\theta}_2 \rangle \in \mathfrak{m}[\boldsymbol{\theta}_1(\mathtt{c}(\mathtt{J},\mathtt{K}))]: \\
&\qquad v' \leftarrow v_1 \circ v_2 \\
&\qquad \textbf{for } \langle v_3, \boldsymbol{\theta}_3 \rangle \in \mathfrak{m}[\boldsymbol{\theta}_2(\mathtt{d}(\mathtt{K},\mathtt{L}))]: \\
&\qquad\quad \mathfrak{p} \cup= \boldsymbol{\theta}_3((\mathtt{a}(\mathtt{I},\mathtt{L}) \oplus= v' \circ v_3))
\end{aligned}$$

在这个例子中，有两个程序： $\mathfrak{p}$ 和 $\mathfrak{m}$。我们在 $\mathfrak{m}$中查找值，并更新 $\mathfrak{p}$。括号表示法 $\mathfrak{m}[\boldsymbol{x}]$ 会返回一个流，其中包含值的有序对，以及对 $\boldsymbol{x}$中变量的绑定。[21] 符号表示 $\mathfrak{p}[\boldsymbol{x}] \oplus= \beta$会向 $\mathfrak{p}$ 添加由展开 β中的查询而得到的规则。[22] 我们很快就会对查找与更新记法进行形式化定义。

一个**显式结果流**针对该查询 $\boldsymbol{x}$ 将针对给定程序的查询结果表示为一个流：$[\boldsymbol{\theta}^{(t)}((\boldsymbol{x} \oplus= v^{(t)}))]_{t=1\ldots}$ 其中每个 $\boldsymbol{\theta}^{(t)}$ 都是一个替换映射（在 $\boldsymbol{x}$的变量上），而 $v^{(t)} \in \mathbb{V}$ 则是对项 $\boldsymbol{\theta}^{(t)}(\boldsymbol{x})$的值的贡献。[23] 我们还会使用另一个技巧：随着流中的索引不断推进，我们会隐式 $\boldsymbol{x}$ 中变量的绑定。我们通过将变量表示为指针，并用每个 t 所指向的内容来表示替换映射。

[21]符号表示 $\mathfrak{m}[\boldsymbol{x}]$ 不应与 $\mathfrak{m}[\![\boldsymbol{x}]\!]$混淆，后者会计算与 $\boldsymbol{x}$匹配的项所对应的赋值映射。[22]请注意，展开这些查询的结果可能是任意规则体。[23]结果流还有一个额外优点：惰性求值支持无限长的结果。如果流的使用者不会完全消费该流（也就是说，无限流会被短路），这种灵活性就很有用。

变量（指针）所指向的内容。我们称之为**隐式结果流**。这样做的好处是，如果 x 中的变量出现在 x之外，其绑定就能无缝且高效地传播。[24] 例如，如果我们为f(x)创建一个结果流，那么绑定到x 的内容就会传播到 g(x) 中，因为它们共享变量x。这种实现策略的好处是，在大多数算法中，都可以将 $\theta^{(t)}$ 及其产生的替换抽象掉，从而编写简洁的伪代码，无需直接操作或管理 $\theta^{(t)}$。

3.1.2.1 隐式ResultStream数据结构

我们的隐式结果系统通过原地合一（算法 x ）直接绑定1中的变量，从而实现替换。此系统中的每个变量都表示为一个指针（或引用）。每当不同的变量x 和 y 合一时，我们只需将 x 指针绑定到 y。之后，若要测试两个变量是否相等，只需沿着各自的指针找到对应的根，并比较它们是否字面相等。[25] 我们扩展 vars(·)，使其返回由给定结果流控制的变量集合。

由于我们的迭代器是原地操作的，要继续进行迭代 $t+1$，迭代器需要撤销迭代 t 中的合一，之后才能安全地应用任何新的合一。原地实现避免了不必要的复制操作（例如调用fresh、复制 θ）。为了便于撤销合一，我们

[24]类似的数据结构也构成了 Warren 抽象机的基础，而 Warren 抽象机又是大多数 Prolog 实现的基础（Warren，1983；Aït-Kaci，1991）。[25]我们的数据结构受到广为人知的并查集数据结构（Galler 和 Fischer，1964）的启发，该结构用于表示等价关系。Mannila 和 Ukkonen 曾探讨将并查集用于合一（1986）。

实现了合一_隐式地作为 Python风格迭代器，其中包含局部的执行和撤销代码。如果合一失败，该迭代器循环零次。如果合一成功，该迭代器会将变量绑定作为副作用进行修改，并 yield 一个值，然后挂起，等待处理该结果。该迭代器恢复后，会撤销其绑定——将其参数恢复到调用合一_隐式地之前的状态。

隐式合一的伪代码见算法 1；它使用基于 Python生成器 创建迭代器的语法。[26] 该过程中的逻辑与 合一 （算法 34）相同。伪代码使用了以下辅助函数： isbound(·)，当且仅当其参数不是空指针（即不是自由变量）时返回 true；而deref(·)则沿着指针进行解引用。

最后，我们将 $[x = y]$ 简写为 隐式合一_ $(x,\ y)$，[27] 并将 $v[x = y]$ 简写为 $v \circ [x = y]$ 其中 $v \in \mathbb{V}$ 是常量值。

3.1.2.2 查找（$\mathfrak{p}[x]$）

在本节中，我们将展示如何通过将查询与程序规则取交集，并连接结果流使其对应于积查询，从而创建（隐式的）结果流。查询与程序 x进行 $\mathfrak{p}$ 交集运算的伪代码见算法2。在此过程中，查询 x 会

[26]Python生成器的语法（https://peps.python.org/pep-0255/）包含**yield** 关键字，与常规函数中的 **return** 类似。不过，两者的主要区别在于，**return**会提供一个单一 值，而**yield** 则允许生成由多个值组成的序列 。此外，生成器可以暂停并恢复：**yield** 语句会输出一个值，并暂时挂起执行。下次调用生成器时，它会从上次中断的位置继续执行，直到遇到另一个**yield** 或到达函数末尾。此外，表达式“**yield from** xs”是“**for**x ∈ xs: **yield** x”的简写。[27]这种符号表示受到艾佛森括号记法的启发。

算法 1 合一_隐式地：隐式合一算法。

```
1.  定义 合一_隐式地(x, y):
2.      如果 x = y:  yield 1 _                                  ▷成功：挂起
3.      否则如果 isvar(x):
4.          如果 isbound(x): yield from 合一_隐式地(解引用(x)  y)
5.          否则，如果 是变量(y) 并且 isbound(y)：从……中 yield 合一_隐式地(x,  解引用(y))
6.          否则，如果 x 不在 y 中:
7.              try:
8.                  绑定 x 到 y                                 ▷ x 是自由的；在此将其绑定到 y
9.                  yield 1                                     ▷成功：挂起
10.             finally:
11.                 解除绑定 x                                  ▷ 撤销绑定
12.     否则如果 isvar(y)：  从……里面 yield 合一_隐式地(y, x)      ▷ 由对称性处理
13.     否则如果 isterm(x) 且 isterm(y) 且它们的标签相同:
14.         yield from 合一_隐式地(x.参数  y.参数)
15.     否则如果 isseq(x) 并且 isseq(y) 并且 |x| = |y|:          ▷ 合一参数序列
16.         如果 |x| = 0:  yield 1 _                            ▷成功：挂起
17.         否则:
18.             for _ ∈ 合一_隐式(x_1, y_1):
19.                 yield 自 合一_隐式(x_(1:], y_(1:])
```

以非确定性方式与不同情况进行匹配，包括值、内置项和程序规则。对于程序规则，如果查询 x 与规则的头部 x' 合一 $x' \oplus\!= y_1 \circ \cdots \circ y_K$，分派过程就会挂起，从而允许方法的调用者查看 x 中变量的状态以及$y_1 \circ \cdots \circ y_K$ 由**yield**发出的值。当调用者准备好继续寻找下一个匹配项时，处理过程会继续检查 p中的后续规则。请注意，此过程不会计算 $\mathrm{p}[\![x]\!]$；它只对查询 x 进行一步扩展。

内置关系。 在第 2.6 行，我们指出内置关系应将其逻辑插入查找过程。为说明这种逻辑

[28]在实践中，我们通过对第 2.8 行规则进行索引来加速循环 p ，方法是使用“哈希码”，确保对于任意 q，合一(头部(s) q) ≠ 失败 ⇒ 哈希(头部(s)) = 哈希(q)。此外，通过为每条规则保留一组 fresh 副本，并在没有活动引用时从池中重复使用副本，可以在一定程度上摊销 fresh 的代价。

算法 2 $\mathfrak{p}[x]$：求交查询 x 与程序 $\mathfrak{p}$的规则。返回一个隐式结果流（即一个会修改 x 的变量绑定并发出规则体的迭代器）。

```
def 𝔭[x]:
    ▷ 隐式地与 x 中的规则头部合一；返回其规则体的流，并满足隐式合一。如果 x 是一个值，则返回该值。如果 x 是一个内置关系，分派到内置结果流方法。
    如果 x ∈ 𝕍: yield x
    否则，如果 x 是内置项:
        ▷ 分派到特定于内置项的代码；示例代码见算法3。
        (...)
    else
        对于R ∈ 𝔭:                                          ▷ 在程序中使用规则[28]
            在.....下面，我们通过 fresh 化来避免递归中的变量名冲突
            (x' ⊕= y_1 ∘ ··· ∘ y_K) ← fresh(R)              ▷ 通常，规则的体只是一个常量。
            对于 _ ∈ [x' = x]:
                yield y_1 ∘ ··· ∘ y_K
```

可能是什么样子，考虑查询 x 匹配内置约束A 是 B + C。分派过程的伪代码如算法 3 所示。

请注意，伪代码中有几种特殊情况，取决于 A、B和 C是否已绑定为实际数值。在最后的“else”情况中，我们看到一个关于内置关系的问题：它们必须充分实例化后才能进行求值。例如，加法内置谓词A 是 B+1 实例化不足：它拒绝枚举满足该关系的无限多个有序对 ⟨A、B⟩ 。判断一个关系何时算充分实例化，由内置库设计者自行决定。因此，通过修改算法3，该库可以（惰性地）枚举无限多个浮点数有序对，以此作为不可数实数解集合的替代表示。通常，当解的数量为无限时，内置关系会抛出实例化错误异常，因为用户更可能是犯了错误，而不是想要一个

算法 3 内置关系分派。

```
匹配 x:
    ⋮
  情况A 是 B + C:
    如果 isnum(B) 且 B = 0:                          ▷ 单位元（即 0）的特殊情况
      yield from[A=C]
    否则如果 isnum(C) 并且 C = 0:
      yield 从[A=B]
    否则如果 isnum(B) 并且 isnum(C):                  ▷ 给定 B 和 C 求解 A
      a ← B +C                                  ▷ 该 + 运算在此处是宿主语言的一部分。
      yield from[一个=a]
    否则如果 isnum(A) 并且 isnum(C):                  ▷ 已知 A 和 C，求解 B
      b ← A −C
      yield from [B=b]
    否则如果 isnum(A) 并且 isnum(B):                  ▷ 已知 A 和 B，求 solve
      c ← A −B
      yield from [C=c]
    else
      如果 可选:                                      ▷ 如果允许延迟
        yield x                                  ▷ 返回一个“延迟约束”
      else
        引发 InstantiationFault()
    ⋮
  情况A 是 B * C:
      ⋮
  情况A 是 B / C:
      ⋮
```

无限解流。另一种策略（在代码中标记为“可选”）是延迟处理该约束，希望它之后能得到更多实例化。我们注意到，某些内置项的实现可能会抛出运行时异常，而其他实现可能会将错误编码为特殊值（类似于浮点数中的NaN ）。具体采用哪种方式，由特定内置关系的设计者决定。

连接。接下来，我们将介绍如何提升该运算符，以连接值流及其关联的变量绑定。假设 $\beta_1 \leftarrow \mathfrak{p}[y_1]$ 和 $\beta_2 \leftarrow \mathfrak{p}[y_2]$ 对于

查询 y_1 和 y_2。正如符号表示的选择 $\beta_1 \circ \beta_2$所暗示的，连接结果流的含义与规则右侧 $y_1 \circ y_2$的含义相同。算法 4 给出了如何为 $\beta_1 \circ \beta_2$ 创建结果流的算法4。这个算法简单得令人惊讶：它只是一对嵌套循环。它之所以如此简单，是因为采用了隐式合一策略：变量会在子目标之间无缝且高效地传播。[29] 特别是，无需传递或返回显式的替换映射（$\boldsymbol{\theta}$）。这些嵌套循环对应一个连接操作，因为外层循环在 β_1上的绑定会传递给内层循环，使其在 β_2上迭代。每次外层循环迭代时，内层循环迭代器都会重新初始化： β_2 在给定初始绑定的情况下，会从头开始，而这些绑定通常与外层循环上一次迭代的绑定不同。换句话说，为确保正确的连接行为，只有当 $\mathfrak{p}[y_1]$ 和 $\mathfrak{p}[y_2]$ 被置于迭代上下文中时（例如，第 *4.3* 行或第 4.3 行或第 4.4 行），才应初始化它们的迭代器。这确保内层循环 $\mathfrak{p}[y_2]$在外层循环的每一轮中都对应一个新的迭代器。

算法 4 $\beta_1 \circ \beta_2$：连接结果流的算法 β_1 与 β_2 .

```
1. def β1 ∘ β2:
2.     ▷ 外层迭代器的每次迭代都会通过隐式统一 β1 绑定变量，
        这些绑定会传播到内层迭代器 β2。
3.   对于 v1 ∈ β1:
4.     对于 v2 ∈ β2:
5.       yield v1 ∘ v2
```

连接操作的结果是另一个隐式结果流。我们将$\mathfrak{p}[y_1 \circ \cdots \circ y_K]$简写为 $\mathfrak{p}[y_1] \circ \cdots \circ \mathfrak{p}[y_K]$。这种简写可以形式化为

[29]这种传播在逻辑程序设计文献中通常被称为横向信息传递策略 。

在算法2 中添加一个分支，使其调用算法4。[30] 请注意，算法中的 for循环可以按任意顺序执行；不过，顺序的选择可能会影响运行时间。[31] 由于连接操作具有结合性，多路连接无需加括号；它被明确定义为多个结果流的 -乘积。在 §6.3中，我们将介绍一种自动化方法，用于确定多路积采用哪种序最为高效。

。

3.1.2.3 更新（$\mathfrak{p}[x] \oplus\!\!= \ldots$）

本节讨论如何将结果流聚合为对程序的更新。设 $\mathfrak{p}$ 是一个程序。由于 $\mathfrak{p}$ 只是赋值映射的规约，因此它实际上是一种经过扩展的映射数据结构（类似于 Python 字典或 Java哈希映射）。因此，用户或算法应当能够使用特定值来更新给定项的值。为此，我们允许 $\mathfrak{p}$ 使用与更新映射相同的符号表示进行更新：[32]

$$\mathfrak{p}[\mathtt{f(1)}] \oplus\!\!= 3 \quad \overset{\text{translates to}}{\Longrightarrow} \quad \mathfrak{p} \leftarrow \mathfrak{p} \cup \{\mathtt{f(1)} \oplus\!\!= 3\}$$

同样，

$$\mathfrak{p}[\mathtt{f(X)}] \oplus\!\!= 3 \quad \overset{\text{translates to}}{\Longrightarrow} \quad \mathfrak{p} \leftarrow \mathfrak{p} \cup \{\mathtt{f(X)} \oplus\!\!= 3\}$$

我们还可以添加一条包含子目标的新规则

$$\mathfrak{p}[\mathtt{f(X)}] \oplus\!\!= \mathtt{g(X,Y)} \circ \mathtt{h(Y)} \quad \overset{\text{translates to}}{\Longrightarrow} \quad \mathfrak{p} \leftarrow \mathfrak{p} \cup \{\mathtt{f(X)} \oplus\!\!= \mathtt{g(X,Y)} \circ \mathtt{h(Y)}\}$$

[30]另一种选择是将其设为内置项。[31]需要特别注意，非交换的 运算的被乘数必须按正确顺序出现。[32]请注意，这里的更新是加法式的（正如 $\oplus\!\!=$所建议的那样）；它们不是替换或覆盖。

那么，包含作为结果流的子目标（可能来自其他程序）的新规则会发生什么?

$$\mathfrak{p}[\mathtt{f(X)}] \oplus\!\!= \mathtt{g(X,Y)} \text{。} \mathfrak{m}[\mathtt{h(Y)}] \text{ 可转写为} \Longrightarrow \textbf{对于} v \in \mathfrak{m}[\mathtt{h(X)}]\text{：}\quad \mathfrak{p}[\mathtt{f(X)}] \oplus\!\!= \mathtt{g(X,Y)} \quad v \quad \text{。}$$

请注意，流中的元素不一定必须是 $\mathbb{V}$中的值；它们可以包含延迟子目标，例如，如果 $\mathfrak{m}$ 具有以下规则$\mathtt{h(X)} \oplus\!\!= \mathtt{p(X,Z)}$，如上例所示。使用这种符号表示时需要谨慎，以确保$\mathtt{p(X,Z)}$ 项在 $\mathfrak{m}$ 和 $\mathfrak{p}$中的含义一致。安全的设置是指能够保证延迟子目标在不同程序中的含义相同；内置项就是这类关系的典型例子，它们在不同程序中的含义相同。这种灵活性使我们能够将$\mathtt{h(X)}$ += 3 作为(1 是 X 取模 2)这一规则纳入 $\mathfrak{m}$，从而扩大我们能够紧凑表示的赋值集合。

最后，我们提及一种在此方法中隐式处理的重要特殊情况。用一组值（或规则体）替换子目标，可以“丢失”一个变量。请看以下情况，

示例 7. 假设我们有一个空程序 $\mathfrak{p}$ 和

$$\mathfrak{m} = \left\{ {}_{50} \quad \mathtt{f(X)} \mathtt{+=} \mathtt{4}. \right.$$

我们希望应用规则目标 += f(X) 到值在.....里面 $\mathfrak{m}$ 使用以下内容:

$$\mathfrak{p}[\text{目标}] \oplus\!\!= \mathfrak{m}[\mathtt{f(X)}]$$

如果我们 不 小心，那么添加到 $\mathfrak{p}$ 的目标 += 4规则会是，但这是不正确的，因为值4需要被无限次求和。我们通过在更新中加入一个带有无限4 和多重性因子的

实数半环 $\infty \cdot 4 = \infty$的情况。因此，正确的更新是目标 += ∞。[33]

丢弃变量检查封装在添加规则的过程中。在我们的实现中，执行如下操作。设 $\mathfrak{p}[\boldsymbol{x}] \oplus\!= \beta$为一个更新操作，其中 β 是一个流。设 v 为 β的流中的当前值。我们使用以下**重数校正**：

$$m = \begin{cases} 1 & \text{if } \mathsf{vars}(\boldsymbol{x} \oplus\!= \beta) \subseteq \mathsf{vars}(\boldsymbol{x} \oplus\!= v) \\ \infty & \text{otherwise} \end{cases} \tag{3.1}$$

其中$\mathsf{vars}(\cdot)$ 返回其参数中的自由变量集合。该条件表示：如果将规则 $\boldsymbol{x} \oplus\!= \beta$替换为 $\boldsymbol{x} \oplus\!= v$会丢失一个变量，则需要对 v求和无穷多个副本。对于流 β的每次迭代，我们使用 $v \cdot m$，其中 m 是将（3.1）应用于该次迭代（处于其当前状态）所得的结果。每当我们使用更新记法（$\mathfrak{p}[\boldsymbol{x}] \oplus\!= \beta$）添加规则时，都会应用此校正。

3.2 一般数学

为完整起见，本节给出全文所用关键数学概念的精确定义。

定义 1。 设二元关系 $\mathcal{R}$ 在集合 $\mathcal{A}$ 上是笛卡尔积的一个子集：$\mathcal{R} \subseteq \mathcal{A} \times \mathcal{A}$。我们通常使用中缀运算符表示包含性测试，例如 $a \mathcal{R} b \iff \langle a,\ b \rangle \in \mathcal{R}$。

用于定义二元关系的形容词包括：

[33]对于幂等半环（例如布尔半环、最小加法），多重性 ≥ 1 都相同。这就是为什么在普通逻辑编程场景中，不会出现检查更新多重性所带来的额外复杂性：布尔半环是幂等的。

- **自反**：$\forall a \in \mathcal{A}$：$a \mathcal{R} a$
- **对称**：$\forall a, b \in \mathcal{A}$：$a \mathcal{R} b \Longrightarrow b \mathcal{R} a$。
- **非对称**：$\forall a, b \in \mathcal{A}$：$a \mathcal{R} b \Longrightarrow \neg (b \mathcal{R} a)$
- **反对称**：$\forall a, b \in \mathcal{A}$：$a \mathcal{R} b, b \mathcal{R} a \Longrightarrow a = b$
- **传递**：$\forall a, b, c \in \mathcal{A}$：$a \mathcal{R} b, b \mathcal{R} c \Longrightarrow a \mathcal{R} c$

定义 2. 一个 划分 σ 集合 $\mathcal{A}$ 是由（非空）子集组成的集合 $\{\sigma_1, \dots, \sigma_K\}$属于 $\mathcal{A}$，且每个元素 $a \in \mathcal{A}$ 恰好属于其中一个子集。换言之，σ 必须是

(i) 互斥：$\forall i \neq j$：$\sigma_i \cap \sigma_j = \emptyset$*(ii)* 完备：$\mathcal{A} = \bigcup_{k=1}^{K} \sigma_k$

3.2.1 序关系

定义 3. 集合 上的二元关系 $\equiv$ 在集合 $\mathcal{A}$的元素上定义了一个等价关系，$\mathcal{A}$ 当且仅当对所有a、b和$c \in \mathcal{A}$，

(i) 自反：$(a \equiv a)$。*(ii)* 对称：$(a \equiv b) \Longrightarrow (b \equiv a)$。*(iii)* 传递：$(a \equiv b), (b \equiv c) \Longrightarrow (a \equiv c)$。

对于给定元素 $a \in \mathcal{A}$，其 等价类 $[a]^{(\equiv)} \overset{\text{定义}}{=} \{a' \mid a \equiv a', a' \in \mathcal{A}\}$。所有等价类组成的集合 $\left\{ [a]^{(\equiv)} \,\middle|\, a \in \mathcal{A} \right\}$ 构成 划分，其划分对象为 $\mathcal{A}$。

定义 4。 集合 上的二元关系 $\preccurlyeq$ 定义集合 $\mathcal{A}$上的 偏序，当且仅当 $\mathcal{A}$ 的所有元素均满足以下条件：a、b、$c \in \mathcal{A}$：

(i) 自反：$a \preccurlyeq a$ *(ii)* 反对称：$a \preccurlyeq b$, $b \preccurlyeq a \implies a = b$ *(iii)* 传递：$a \preccurlyeq b$, $b \preccurlyeq c \implies a \preccurlyeq c$

定义 5. 一个二元关系 $\prec$ 在集合 $\mathcal{A}$ 上定义了严格偏序在 $\mathcal{A}$ 的元素上，当且仅当以下条件对所有a，b，$c \in \mathcal{A}$均成立：

(i) 非对称：$a \prec b \implies \neg(b \prec a)$
(ii) 传递：$a \prec b$, $b \prec c \implies a \preccurlyeq c$

定义 6. 一个二元关系 $\preccurlyeq$ 在集合 $\mathcal{A}$ 上定义了全序当且仅当它既是偏序，又是全序：$a \preccurlyeq b$ 或者 $b \preccurlyeq a$ 对于所有 a，b，$c \in \mathcal{A}$。

定义 7. 一个二元关系 $\lesssim$ 在集合 $\mathcal{A}$ 上定义了预序在 $\mathcal{A}$ 的元素上，当且仅当以下条件对所有 a，b，$c \in \mathcal{A}$均成立：

(i) 自反：$a \lesssim a$ *(ii)* 传递：$a \lesssim b$、$b \lesssim c \implies a \lesssim c$

定义 8。 集合 $\prec$ 上的 二元关系 $\mathcal{A}$ 是 良基序，当且仅当它不包含任何 无限下降链：不存在无限序列 a_0、a_1、a_2、$\ldots \in \mathcal{A}$ 使得对所有 $a_{n+1} \prec a_n$ 都有 $n \in \mathbb{N}$。

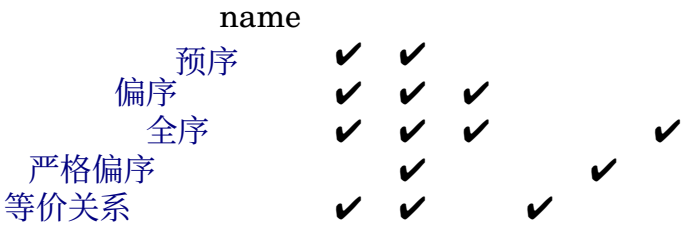


图 3.1： 序关系分类。

3.2.2 函数

定义 9。 从集合 到集合 ϕ 的函数 $\mathcal{A}$ 是一个二元关系，它为 $\mathcal{B}$ 中的每个元素 $\mathcal{A}$ 恰好分配一个元素。为简洁起见，我们写作 $\phi:\ \mathcal{A} \to \mathcal{B}$ 。我们通常将 $\mathcal{A} \to \mathcal{B}$ 称为 ϕ的类型。

定义 10. 映射 $\phi : \mathcal{A} \to \mathcal{B}$ 是一个 双射 如果对所有 $a \in \mathcal{A}$:

- 满射: $\forall\, b \in \mathcal{B}:\ \exists\, a \in \mathcal{A}:\ \phi(a) = b$.
- 单射: $\forall\, a, a' \in \mathcal{A} : (\phi(a) = \phi(a')) \implies (a = a')$.

一个等价定义是， ϕ 是双射，从 $\mathcal{A} \to \mathcal{B}$ 出发，当且仅当以下两个条件成立:

- $\forall\, a \in \mathcal{A}:\ \exists!\ b:\ \phi(a) = b$
- $\forall\, b \in \mathcal{B}:\ \exists!\ a:\ \phi(a) = b$

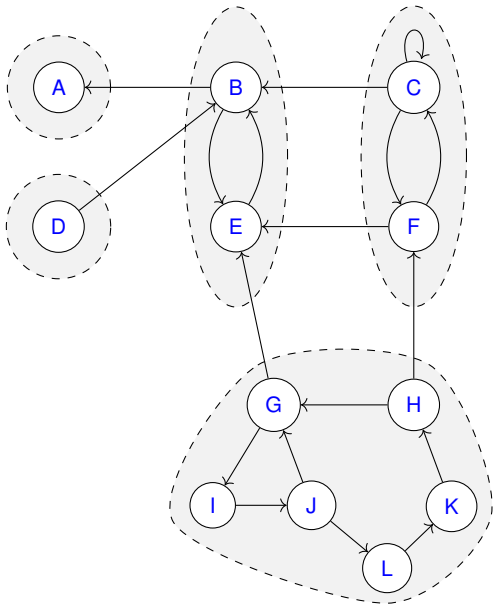
此外，我们称函数 ϕ^{-1} 是**的逆** ϕ 当且仅当$\forall\, b:\ \phi^{-1}(b) = a$ 其中 a 是 $\phi(a) = b$的唯一解。

3.2.3 图

定义 11。 如果有向子图中每一对节点 i 和 j 之间都存在从 $i \rightsquigarrow j$ 到 $j \rightsquigarrow i$的路径，且路径位于该子图内，则该有向子图是强连通的。

定义12. 图 G 的强连通分量 （强连通分量）将其划分为强连通子图。这些强连通分量是由共可达性等价关系$(i \nleftrightsquigarrow j) \iff (i \rightsquigarrow j)$和$(j \rightsquigarrow i)$给出的等价类。将各强连通分量关联起来的图称为图 G 的凝聚图 。

示例8. 在.....下面是一个图示[34]，该图示为一个有向图，其强连通分量已用虚线椭圆标注。



定义 13. 有向图的拓扑排序*(*拓扑排序，简称 *toposort*）是其节点的一种全序（定义 *6*），满足对于每条边 $i \to j$， i 在序中位于 j 之前。当且仅当图无环时，拓扑排序才存在。

[34]此图示基于
https://tex.stackexchange.com/questions/592539/how-to-create-figures-with-strongly-connected-components。

我们经常讨论有向循环图的凝聚图的拓扑排序；凝聚图总是无环的。为求完备起见，我们在算法 1972中提供了 Tarjan（1972）的算法伪代码，用于求出图的强连通分量的拓扑排序列表。5。

算法5 强连通分量(传入，目标)：枚举强连通分量——（强连通分量；定义 12）在有向图中的拓扑序。

```
▷ 这里，图 ⟨V， E⟩ 以隐式方式表示，使用函数 传入(v) 来枚举节点 u，使得(u → v) ∈ E；并以目标节点的子集目标 ⊆ V 作为遍历的起点；如果目标节点目标 中没有任何节点（传递地）依赖于给定的强连通分量，它将从返回的强连通分量列表中省略。该算法由 Tarjan 提出（1972）。它的运行时间为在 O(|V| + |E|) 时间内运行，并使用 O(|V|) 额外空间。 注意，强连通分量(出边 V)，其中出边(u) 会枚举节点 v，使得(u → v) ∈ E，并返回相同的强连通分量集合强连通分量(入边 V)，但采用逆向拓扑序。
定义 强连通分量(入边，目标):
    栈 ← []                                          ▷ 我们维护一个由不同项组成的栈
    位置 ← 映射()                                    ▷ 我们记录栈中每个节点的位置
    已访问 ← {}                                      ▷ 我们维护一个已访问节点的集合
    sccs_列表 ← []                                   ▷ 将作为返回值
    定义 遍历(v):
        low ← |栈 |                                  ▷ 将作为返回值
        如果 v ∉ 已访问:                              ▷ 首次访问
            已访问。添加(v)
            位置[v] ← |栈 |
            栈。推入(v)                               ▷ 我们可以撤销此操作
            对于 u ∈ 传入(v):
                如果 u ∈ 位置:                        ▷ 检测到环
                    low 最小值=位置[u]
                else
                    low 最小值=遍历(u)
            如果 low = 位置[v]:                       ▷ 如果 low 未改变；其下的所有内容 v 都在它的强连通分量中
                σ ← {}
                当 |栈 | > low:
                    u ←栈.弹出()
                    σ.添加(u)
                    位置.移除(u)
                强连通分量_列表.追加(σ)               ▷ v 的强连通分量已就绪
        返回 low
    对于 v ∈ 目标：遍历(v)
    返回 强连通分量_list
```

3.2.4 抽象代数

定义 14. 一个 二元运算 在集合 $\mathbb{V}$ 上是任意函数 $\odot : \mathbb{V} \times \mathbb{V} \to \mathbb{V}$。二元运算符通常使用中缀表示法 $a \odot b$ 而不是 $\odot(a, b)$，后者是函数应用的典型写法。

定义 15. 一个 幺半群 $\langle \mathbb{V}, \odot, i \rangle$ 是一个集合 $\mathbb{V}$ 配备了 二元运算 $\odot$和一个单位元 i，并满足以下性质：

(i) $\odot$ 是结合的： $\forall a, b, c \in \mathbb{V} : (a \odot b) \odot c = a \odot (b \odot c)$*(ii)* i 是左、右单位元： $\forall a \in \mathbb{V} : i \odot a = a \odot i = a$

用于描述幺半群的一些形容词包括：

- **交换的** $\iff \forall a, b \in \mathbb{V} : a \odot b = b \odot a$
- **幂等的** $\iff \forall a \in \mathbb{V} : a \odot a = a$

定义16。 一个群是一个幺半群 $\langle \mathbb{V}, \odot, i \rangle$ 其中，此外每个元素都有一个逆元： $\forall a \in \mathbb{V}, \exists b \in \mathbb{V} : b \odot a = a \odot b = i$。元素 b 可记作 a^{-1}，如果该群被视为乘法型；或 $-a$，如果该群被视为加法型。

定义17。 一个半环 $\langle \mathbb{V}, \oplus, \cdot, 0, 1 \rangle$是一个集合 $\mathbb{V}$，配备两个幺半群运算 $\langle \mathbb{V}, \oplus, 0 \rangle$ 以及 $\langle \mathbb{V}, \cdot, 1 \rangle$，使得以下性质成立：

(i) $\langle \mathbb{V}, \oplus, 0 \rangle$ 是一个交换的幺半群*(ii)* $\langle \mathbb{V}, \cdot, 1 \rangle$ 是一个幺半群

(iii) 分配律： $\forall a,\ b,\ c \in \mathbb{V}$:

$$a \circ (b \oplus c) = (a \circ b) \oplus (a \circ c) \quad \textit{and} \quad (b \circ a) \oplus (c \circ a) = (b \oplus c) \circ a$$

(iv) 乘法消去律 由 0： $\forall a \in \mathbb{V}$:

$$\underline{0} \circ a = a \circ \underline{0} = \underline{0}$$

此外，我们采用以下符号表示 n**重加**

$$b \cdot n \overset{\text{def}}{=} n \cdot b \overset{\text{def}}{=} \bigoplus_{i=1}^{n} b \quad \text{for } n \in \mathbb{N} \cup \{\infty\}, b \in \mathbb{V} \tag{3.2}$$

和 n**重乘**

$$b^n \overset{\text{def}}{=} \bigotimes_{i=1}^{n} b \quad \text{for } n \in \mathbb{N} \cup \{\infty\}, b \in \mathbb{V} \tag{3.3}$$

用于描述半环的一些形容词包括：

- **交换的** $\iff$ 是交换的
- **幂等的** $\iff$ $\oplus$ 是幂等的

定义 18. 一个 闭半环 具有一个额外的运算符，称为（*Kleene*）星运算符 $(\cdot)^\star$。该运算必须满足以下公理，对所有 $a \in \mathbb{V}$:

$$a^\star = \underline{1} \oplus a \circ a^\star \quad \textit{and} \quad a^\star = \underline{1} \oplus a^\star \circ a$$

我们常常把 $a^\star$ 解释为由展开其任一递归恒等式而产生的无穷级数的解：

$$a^\star = \bigoplus_{n=0}^{\infty} a^n \quad \text{for } a \in \mathbb{V}$$

星运算对于求解某些线性方程很有用。例如，$x = b \oplus a \circ x \oplus c \circ y$ 可以被重新排列，从而“solve”出 x，而不使用减法或除法：$x = a^\star \ (b \oplus c \circ y)$。

。

总是具有星运算的一类特殊半环是 K-闭半环。

定义 19. 对于 $K \in \mathbb{N}$，半环 $\langle \mathbb{V}, \oplus, , 0, 1 \rangle$是 K-闭的 当且仅当[35]

$$\forall\, a \in \mathbb{V}, \forall\, K' > K : \bigoplus_{k=0}^{K'} a^k = \bigoplus_{k=0}^{K} a^k.$$

直观而言，K-闭半环意味着，我们只需至多沿循环迭代 K 次，就能对循环的贡献求和；进一步迭代不会影响求和的值。当 $K = 0$时，该条件可简化为$\forall\, a \in \mathbb{V}$: $1 \oplus a = 1$。下面简要介绍几个 K-闭半环的例子；这些例子定义于 §3.3。0-闭半环的例子包括布尔、最小加法和维特比半环。一般的 K-闭半环的例子包括$(K+1)$-最佳半环（Mohri，2002年，§6）以及 K-折叠自然数。不属于 K-闭半环的例子包括算术半环。 –

每个 K闭半环都有用于计算星号运算的简单表达式，因为它们不再是无穷求和。

[35]可以用 $\forall K'$ 和 $K' = K + 1$替换——这些条件是等价的（Mohri，2002年，引理 4）。

$$\begin{cases} 0\text{-closed} : a^* = \underline{1} \\ 1\text{-closed} : a^* = \underline{1} \oplus a \\ 2\text{-closed} : a^* = \underline{1} \oplus a \oplus a^2 \\ \quad \vdots \\ K\text{-closed} : a^* = \bigoplus_{k=0}^{K} a^k \end{cases} \tag{3.4}$$

定义 20。 如果一个半环在 偏序关系 下是单调的 $\preccurlyeq$，则

$$\forall\, a \preccurlyeq b, \forall\, c \in \mathbb{V} : \begin{cases} a \oplus c \preccurlyeq b \oplus c \\ a \circ c \preccurlyeq b \circ c \\ c \circ a \preccurlyeq c \circ b \end{cases}$$

接下来呢? 在第 §3.3节中，我们将介绍若干半环示例。在第 §4.4节中，我们将介绍一些特殊的半环族，它们有助于为 Dyna 程序提供收敛保证。

3.3 半环：动态规划的强大抽象

用于动态规划

半环是动态规划的一种强大抽象。它们提供了一种有用而优雅的抽象层次，可用于

- 自然语言处理算法（例如，Teitelbaum(1973年)、Mohri（1997年）、Goodman（1999）和 Huang（2008年）
- 概率推断（例如，Aji 和 McEliece（2006年））
- 数据库溯源（例如，Green、Karvounarakis 等人（2007）、Green 和 Tannen（2017年）以及 Cheney 等人（2009年））
- 网络分析（例如，Gondran 和 Minoux（2008年）以及 Baras 和 Theodorakopoulos（2010年））
- 程序分析（例如，Cousot 和 Cousot（1977年）、Esparza 等人（2010年）和 Dolan（2013））

半环使以下关注点分离成为可能：

（1）设计一种紧凑的编码来表示推导集合（例如，图中的路径、句子的句法分析结果）

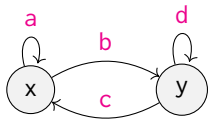
(2) 如何评估并汇总这些推导（例如，最高权重推导、所有推导的总权重）

在 Dyna 中，第 1 部分由程序规则指定，第 2 部分则通过选择半环来确定。按半环进行模块化，可以让同一算法在时间/空间复杂度相同的情况下呈现不同的语义（假设半环运算的时间复杂度为常数），而且我们甚至可以重复利用该实现。[36]

半环的关键性质是分配律，因为它允许我们提取公共子表达式：我们可以安全地 rewrite $(a \circ b) \oplus (a \circ c)$（三次运算）为一个等价的更小表达式 $a \circ (b \oplus c)$（两次运算）。提取后的表达式可以显著更小。

示例 9. 下面的示例可视为对带有两个状态x 和 y 的加权有向图进行编码。图中的权重被抽象地保留为符号a、b、c和 d。

```
x += a * x.
x += b * y.
y += c * x.
y += d * y.
y += 1.  % final state
```



我们可以将 *Dyna* 程序转换为方程组：

$$\begin{aligned} \mathrm{x} &= \mathrm{a} \circ \mathrm{x} \oplus \mathrm{b} \circ \mathrm{y} \\ \mathrm{y} &= \mathrm{c} \circ \mathrm{x} \oplus \mathrm{d} \circ \mathrm{y} \oplus \underline{1} \end{aligned}$$

方程组的解是一个映射，其中与x 关联的值是所有从状态x 出发并在状态y结束的路径的集合，而与y 关联的值是所有从状态y 出发并在状态y结束的路径的集合。在这张图中，路径有无限多条，但我们仍然可以用一小组方程来表示它们。下面我们考虑将权重（a、b、c、d）替换为

[36]尽管某些半环可能允许其他半环所不具备的额外效率捷径，但某些实现可能会假定超出基本半环公理的性质。

各种半环中的值。在每种情况下，我们都会描述与x 和y关联的半环值的含义。

- $\langle \{\text{假}, \text{真}\}, \vee, \wedge, \text{假}, \text{真} \rangle$ 其中 真 被置于每条边上；$\Longrightarrow$ y 可从x到达，且y 可从y到达。

- $\langle \mathbb{R} \cup \{\infty\}, +, \cdot, \infty, 0 \rangle \Longrightarrow$ 从x 到y 以及从y 到y的所有路径的总权重。

- $\langle \mathbb{R}, \text{最小值}\ +, 0, 1 \rangle \Longrightarrow$ 从x 到y 以及从y 到y的最小代价路径。

- $\left\langle 2^{\Sigma^*}, \cup, \cdot, \emptyset, \varepsilon \right\rangle$ 其中 Σ 是字母表 $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\} \Longrightarrow$，它是描述从x 到y 以及从y 到y的路径集合的（正则）形式语言。

*T*这只是一个小例子，说明我们可以通过替换半环来实现什么。

*k*同时基本保持 *Dyna* 程序不变。

半环大全

下面列出了几种常用的半环，以及它们出现的一些应用。Dyna 库中提供了其中许多半环 https://github.com/timvieira/dyna-pi。我们并没有为这个列表设定特定的组织原则，但希望它能为一些读者提供有用的引用，并展示半环也能发挥一些出人意料的作用。

逻辑

布尔半环$\langle \{\text{假}, \text{真}\}, \vee, \wedge, \text{假}, \text{真} \rangle$ 用于测试在图里面的可达性，以及更一般地，在规则下方各项的可证明性，这些规则属于

程序。其星运算为$(\text{假})^* = (\text{真})^* = \text{真}$

多值逻辑。

- **三值逻辑** $\langle \{\text{真}, \text{假}, \text{未知}\}, \text{最大值}, \text{最小值}, \text{假}, \text{真} \rangle$ 其中 假 <未知 <真。[37]

- **F模糊集** $\langle [0,1], \text{最大值}, \text{最小值}\, 0, \ 1 \rangle$ 与模糊集相关（Zadeh，1965年）。[38]

- **卢卡西维茨半环** $\langle [0,1], \text{最大值}, \max(0, \ x+y-1)\, 0, \ 1 \rangle$ (卢卡西维茨，1920)

这些半环各自都是 0-闭的；因此，可以使用 K-闭捷径（3.4）。

算术半环

- **计数半环**：$\langle \mathbb{N}, \ +, \ \cdot, \ 0, \ 1 \rangle$。顾名思义，这种半环用于计数，例如计算图中两个节点之间的路径数量。如果 $\mathbb{N}$中加入 ∞，那么 $a^* = 1$ **如果** $a = 0$ **否则** ∞。

- **整数半环**：$\langle \mathbb{Z}, \ +, \ \cdot, \ 0, \ 1 \rangle$。其星运算为 $0^* = 1, \ a^* = \infty$ **如果** $a >$为 0；当 $a < 0$时未定义。

- **实数半环**：$\langle \mathbb{R}, \ +, \ \cdot, \ 0, \ 1 \rangle$。如果我们用 ∞ 扩展 $\mathbb{R}$，那么当 $|a| < 1$时，$a^* = \frac{1}{1-a}$；当 $a \geq 1$时，∞；而当 $a \leq -1$时，它没有定义。所以遗憾的是，这个半环并不完备，无法求解该无穷级数。

[37]通常，$\langle$假，未知，真$\rangle$ 会编码为 $\langle -1, \ 0, \ +1 \rangle$。[38]模糊集扩展了普通集合，支持分级隶属度，用于表示给定元素属于给定集合的程度（0 表示确定不在集合中，而 1表示确定在集合中）。

• **非负实数半环**：$\langle \mathbb{R}_{\geq 0} \cup \{\infty\},\ +,\ \cdot,\ 0,\ 1\rangle$。该半环用于计算概率模型的归一化常数（Goodman，1999）。它的星运算是 $a^* = \frac{1}{1-a}$ 当 $a < 1$ 时，以及 ∞ 其他情况。因此，为了使其闭合，我们要求将 ∞ 加入集合 $\mathbb{R}$。

—

• 对数非负实数半环：$\left\langle \mathbb{R} \cup \{-\infty, \infty\}, +\log +, -\infty, 0 \right\rangle$ 其中$x + y \stackrel{\text{def}}{=} \log(\exp(x)+\exp(y))$。[39] 这个半环与对数非负实数半环同构；不过，它通常更受青睐，因为它可以避免某些数值问题。[40] 它的星运算为 $a^* = -\log(1-\exp(a))$（如果满足 $a \leq 0$），否则为 ∞。

• **一阶期望半环** (Eisner，2001年) 可用于计算一阶统计量，例如期望错误率、熵或期望特征计数。

• **二阶期望半环**(Li 和 Eisner，2009年) 可用于计算二阶统计量，例如错误率的方差、熵的梯度以及特征计数的协方差矩阵。

• K**-折叠自然数** $\langle \{0, \cdots, K\},\ \min(K,\ x+y),\ \min(K,\ x \cdot y)\, 0,\ 1\rangle$可用作护栏，防止因不断计数直至无穷而导致运行时间无限。 当 $K = 2$ 时，它与布尔半环同构。其星运算为 $a^* = 1$**若** $a = 0$**则** K。

[39]为了提高速度和数值精度，对数空间加法运算不应按所写的方式计算，而应使用库函数计算（Mächler，2012 年）。Python 的 NumPy 库提供了用于此目的的函数`numpy.logaddexp`。[40]我们还提及对数实数半环（Li 和 Eisner，2009年，表 3），它在对数非负实数半环的基础上增加了一个符号位，因此可以表示负数。

优化

在某个值集合上求最小值/最大值时，常会用到这些半环，例如图中路径的加权集合。

- 热带半环（又称最小加半环）$\langle \mathbb{R} \cup \{\infty\}, \min, +, \infty, 0 \rangle$用于寻找最短路径或代价最低的推导。它的星运算为 $a^* = 0$

- 北极半环（又称最大加半环）$\langle \mathbb{R} \cup \{-\infty\}, \max, +, -\infty, 0 \rangle$用于寻找最长路径或最高权重推导。它的星运算为 $a^* = 1 \oplus a$ 若 $a \leq 0$ 则 ∞

- 维特比半环（又称最大乘半环）$\langle [0,1], \max, \cdot, 0, 1 \rangle$ 用于在概率模型中寻找最可能解释（MPE）。当模型没有潜变量时，MPE 对应于最大后验（MAP）解码。它的星运算为 $a^* = 1$。如果用数字 > 1扩展载体集合，我们必须将定义扩展为 $a^* = \infty$对于 $a > 1$。

- 最小乘半环 $\langle \mathbb{R}_{\geq 0} \cup \{\infty\}, \min, \cdot, \infty, 1 \rangle$ 可用于找出概率最低的推导。它的星运算为 $a^* = 1 \oplus a$

这些半环可以通过附加的记录信息加以扩充，以便在返回最小值/最大值之外，再返回一个最小值参数/最大值参数（甚至是最小值参数/最大值参数的集合）。还可以使这些优化中被比较的元素

半环元组，并在字典序下对它们进行比较。这是跟踪回溯指针并添加平局决胜操作的一种方法。

该**最大容量半环** $\langle \mathbb{R} \cup \{-\infty, \infty\}$、最大值、最小值 $-\infty$， $\infty \rangle$ 可用于查找图中的最大容量路径。它是0-闭的，因此 $a^* = 1$。

该 K**-最佳半环**（顶层K 半环或底层K 半环）扩展了优化半环，以提供 K-最佳替代方案，其中 $K \geq 0$ 是一个固定常量。[41] 这些 K-最佳半环适用于任何单调乘积运算，包括（乘以非负数的）乘法、笛卡尔积和加法。设 $\langle \mathcal{A}$、 、 $\varepsilon \rangle$ 构成一个幺半群。假设它在 $\preccurlyeq$下具有偏序，且该运算是单调的。

。

。

- $\mathbb{V} \overset{\text{定义}}{=} \left\{ \text{排序}_{\preccurlyeq}^{K}(a_1 \cdots a_K) \mid a_1 \ldots a_K \in \mathcal{A}^K \right\}$：该集合中的每个元素都是一个长度为$K$的有序列表，列表元素来自 $\mathcal{A}$。这里，函数$\text{排序}_{\preccurlyeq}^{K}$ 接收一个序列，根据 $\preccurlyeq$对其进行排序，并截断列表，仅保留 K 个最小元素。

- $A \oplus B \overset{\text{定义}}{=} \text{排序}_{\preccurlyeq}^{K}(A \cup B)$

- $A \circ B \overset{\text{定义}}{=} \text{排序}_{\preccurlyeq}^{K}([a \circ b \mid a \in A, b \in B])$

- $0 \overset{\text{定义}}{=} [\,]$

- $1 \overset{\text{def}}{=} [\varepsilon]$

[41] Goodman（1998）给出了 K-最佳半环，它会急切地合并某个固定K 的最佳值 K。[42] 后来，这种方法被推广（Huang 和 Chiang，2005年），从而允许 K-最佳列表以惰性方式计算（且无需预先知道 K），这通常能显著加快速度，因为许多此类 K-最佳列表根本不会用到（例如，某个不出现在任何完备句法分析中的跨度的 K-最佳列表）。关于如何高效地惰性合并流的实现细节见 Huang 和 Chiang（2005年）。

由于该半环是$(K-1)$-闭的。我们可以使用截断和来实现星运算 $a^* = \sum_{k=0}^{K-1} a^k$。

矩阵

矩阵半环（Lehmann，1977年）非常重要，因为我们将在 §5.3 中用它来求解某些无限和的线性族。

定义 21 （半环上的矩阵）。我们可以将任意基半环"提升"为方阵上的半环$\langle \mathbb{V},\ \oplus,\ 0,\ 1\rangle$ 其中 $\left\langle \mathbb{V}^{D\times D},\ +,\ \mathbf{0},\ \mathbf{1}\right\rangle$，其中 D 是一个集合。[43] 矩阵加法和矩阵乘法运算（如下所示）是对 $\mathbb{R}$上常见矩阵乘法的直接推广。令 $\square$

$\mathbf{A},\ \mathbf{B} \in \mathbb{V}^{D\times D}$。

- $[\mathbf{A}+\mathbf{B}]_{ij} \stackrel{\text{def}}{=} \mathbf{A}_{ij} \oplus \mathbf{B}_{ij}$
- $[\mathbf{A}\,\mathbf{B}]_{ik} \stackrel{\text{def}}{=} \bigoplus_{j\in D} \mathbf{A}_{ij} \circ \mathbf{B}_{jk}$
- $[\mathbf{0}]_{ij} \stackrel{\text{def}}{=} \underline{0}$
- $[\mathbf{1}]_{ij} \stackrel{\text{def}}{=} \underline{1}\ \textbf{\textit{if}}\ i=j\ \textbf{\textit{else}}\ \underline{0}$

星运算 $\mathbf{A}^*$ 可按如下方式计算。如果 $|D|=1$，则 $\mathbf{A}^* = [a]^* = [a^*]$。否则，$|D|>1$，并且我们可以将 $\mathbf{A}$ 分解为非平凡分块 $\langle \mathbf{B}、\mathbf{C}、\mathbf{D}、\mathbf{E}\rangle$，然后应用以下分块递归。

$$\mathbf{A}^* = \begin{bmatrix} \mathbf{B} & \mathbf{C} \\ \mathbf{D} & \mathbf{E} \end{bmatrix}^\star = \begin{bmatrix} \mathbf{B}^* + \mathbf{B}^*\,\mathbf{C}\,\boldsymbol{\Delta}^*\,\mathbf{D}\,\mathbf{B}^* & \mathbf{B}^*\,\mathbf{C}\,\boldsymbol{\Delta}^* \\ \boldsymbol{\Delta}^*\,\mathbf{D}\,\mathbf{B}^* & \boldsymbol{\Delta}^* \end{bmatrix} \tag{3.5}$$

[43]我们有 e 将并置操作（例如 $\mathbf{AB}$）表示为 , 这就是所谓的"可见空格"。

其中 $\mathbf{\Delta} = \mathbf{E}+\mathbf{DB}^*\mathbf{C}$。很容易验证，每次对 $(\cdot)^*$的递归调用，其规模都会减小；因此，该递归定义是良基的。此外，若各个块的大小相等，则运行时间为 $\mathcal{O}(|\,D\,|^3)$。关于正确性证明及更多讨论，请参阅 *Lehmann*（*1977年*）。

溯源半环

溯源半环 (Green、Karvounarakis 等人，2007；Green 和 Tannen，2017年；Cheney 等人，2009年；Schlund，2016)用于数据库系统中验证或解释查询结果。类似地，“推导半环”（Goodman，1998）可用于编码给定项的“推导”集合。[44] Green、Karvounarakis 等人（2007）给出的一些溯源半环示例如下：

- 该**集合半环** $\langle 2^{\Sigma},\ \cup,\ \cap,\ \emptyset,\ \Sigma\rangle$ 其中 Σ 是一个基集，可用于追踪系统中事件发生的可能性。

- 该**why 半环** $\left\langle 2^{2^{\Sigma}},\ \cup,\ \{x\cup y\}_{x\in X,\ y\in Y},\ \emptyset,\ \{\emptyset\}\right\rangle$ 其中 Σ 是一个基集。基集 Σ 是一个注释集合；使用该半环计算得到的值，是给定元组出现在查询结果中所需注释的组合。

- **谱系半环** $\langle 2^{\Sigma}\cup\{\bot\},\ \cup,\ \cup,\ \bot,\ \emptyset\rangle$ 其中 Σ 是一个基集，而 $\bot$ 是

[44]注：Goodman 对“推导”的符号表示与我们的概念不同（我们的概念将在 §4 中形式化）。具体而言，他的概念不如我们的概念一般，因为我们的概念并非由半环运算构建：我们的推导概念由嵌套规则给出，而这些规则不满足乘法结合律。

满足 0 的半环性质的特殊元素。在图的语境中，谱系半环决定了可达弧标签的子集。

形式语言

形式语言： $\left\langle 2^{\Sigma^\star},\ \cup,\ \cdot,\ \emptyset,\ \varepsilon \right\rangle$，其中 Σ 是符号的集合，$\Sigma^\star$ 表示由这些符号组成的字符串集合，ε 是空字符串，而 $\cdot$ 表示字符串连接。形式语言的半环可用于枚举上下文无关文法、有限状态自动机以及许多其他形式系统所接受的字符串集合。遗憾的是，计算星运算并无捷径。有限状态自动机能够紧凑地表示某些无限字符串集合；它们在并集、连接和星运算下是闭的。[45]

加权形式语言：形式语言的半环可以扩展为半环加权语言，其中为每个字符串赋予一个半环权重。令 $\langle \mathbb{V}, \oplus, , 0, 1 \rangle$ 为一个基半环。令 Σ 为一个字母表，如形式语言的半环中所述。加权形式语言的半环 $\left\langle \widehat{\mathbb{V}}, \widehat{\oplus}, \widehat{\ }, \widehat{0}, \widehat{1} \right\rangle$ 定义如下。令 $\widehat{\mathbb{V}} = \mathbb{V}^{\Sigma^*} = \Sigma^* \to \mathbb{V}$ 为所有从 Σ^* 中的字符串映射到基半环中各值的函数集合。令 $a, b \in \mathbb{V}$ 和 $x \in \Sigma^*$:

。

- $[a \mathbin{\widehat{\oplus}} b](x) \stackrel{\text{def}}{=} a(x) \oplus b(x)$

[45]尽管上下文无关语言（CFL）在半环运算下也是闭的，但 CFL 的表示（即上下文无关文法和下推自动机）并没有可用于检验其所表示的上下文无关语言是否相等的（完备）算法，这严重限制了它们作为计算表示的实用性。

- $[a \mathbin{\widehat{\circ}} b](x) \overset{\text{定义}}{=} \bigoplus_{(z)x:\ x=y\cdot z} a(y) \otimes b$

- $[\widehat{0}](x) \overset{\text{定义}}{=} \underline{0}$

- $[\widehat{1}](x) \overset{\text{定义}}{=} \underline{1}$ **如果** $x = \varepsilon$ **否则** $\underline{0}$

遗憾的是，计算星运算并无捷径。

向量集合

- 半线性集半环 是 $\langle 2^{\mathbb{N}^\Sigma},\ \cup,\ +,\ \emptyset,\ \{\mathbf{0}\}\rangle$ 其中 $X + Y \overset{\text{定义}}{=} \{x + y\}_{x\in X,\ y\in Y}$。该半环可用于计算字母表为 Σ的形式语言的 Parikh像。[46] 关于进一步讨论和高效实现，请参阅 Schlund（2016，第5章）。

- 该**帕累托前沿半环**是 ⟨帕累托有效向量集、pareto$(X \cup Y)$、pareto$(X + Y)$ $\emptyset$，$\{\mathbf{0}\}\rangle$。函数pareto 会将向量集合筛选为帕累托有效向量。一个向量集合是**帕累托有效的**，当且仅当其中的每个向量都是非支配的。如果一个向量在所有维度上 $\leq$ 所有其他向量，并且 $<$ 至少一个维度上，则称该向量支配另一个向量。帕累托有效向量集合表示不同目标或准则之间的最优权衡（即，改善一个目标会以恶化另一个目标为代价）。在 $\mathbb{R}^D_{\geq 0}$ 上的帕累托前沿半环是0-闭的，因此其星运算是 $X^* = \underline{1}$。Gondran 和 Minoux（2008年）进一步讨论了该半环。

[46]形式语言的 **Parikh像** L 定义为Parikh$(L) \overset{\text{定义}}{=} \{\text{bag}(w) \mid w \in L\}$ 其中 bag$(w) \overset{\text{定义}}{=} \{\!\{a \mid a \in w\}\!\}$。将每个单词中的符号放入一个袋子，就会去除它们的顺序。

- 向量空间中的**凸集半环**是 ⟨向量空间上的凸集、convexhull$(X \cup Y)$、convexhull$(X+Y)$ $\emptyset$, $\{\mathbf{0}\}$⟩。对于有限维向量空间，我们可以将凸集限制为多胞体，同时仍构成一个半环——其优势在于，每个多胞体都可以表示为有限的顶点集合，从而实现多项式时间运算。[47] Dyer（2013）展示了如何使用该半环执行高效的最小错误率训练（Och，2003年），这是机器翻译中常用的一种技术。关于该半环的更早引文见 Iwano 和 Steiglitz（1990）。

其他

- 半环上的多项式和形式幂级数（Kuich，1997年；Green、Karvounarakis等，2007）在每种情况下都构成一个半环。

- Baras 和 Theodorakopoulos（2010）以及 Gondran 和 Minoux（2008年）给出了许多有趣的网络分析半环，包括用于枚举图割和桥的半环。

- 许多研究使用半环在程序上进行数据流分析（Dolan，2013；Esparza等人，2010年；Cousot和Cousot，1977年）。

[47]凸集半环与帕累托前沿半环类似。区别在于，凸集根据基础目标的线性组合来寻找最优推导集合。

第4章

Dyna 程序语义

在§2中，我们通过若干示例介绍了Dyna语言。在本章中，我们将把Dyna 程序的含义——即它实际计算的内容——形式化为对可能无穷多个半环加权推导的某种求和（§4.1）。[48] 我们用若干简单示例来说明这些挑战。相当令人惊讶的是，我们常常能够计算——或至少准确近似——这些无穷和。我们将不动点迭代描述为用于计算这些语义的数学算法框架。稍后（§5），我们会在可执行算法中具体实现该框架。

[48]先前对 Dyna 的形式化描述（Eisner、Goldlust 等人，2005年）并未将 Dyna 的语义描述为对加权推导求和，因此与半环加权演绎领域的其他研究（如 Goodman（1999））并不一致。然而，这种不一致使得将 Dyna 1 推广到 Dyna 2 语言成为可能（Eisner 和 Filardo，2011年）。

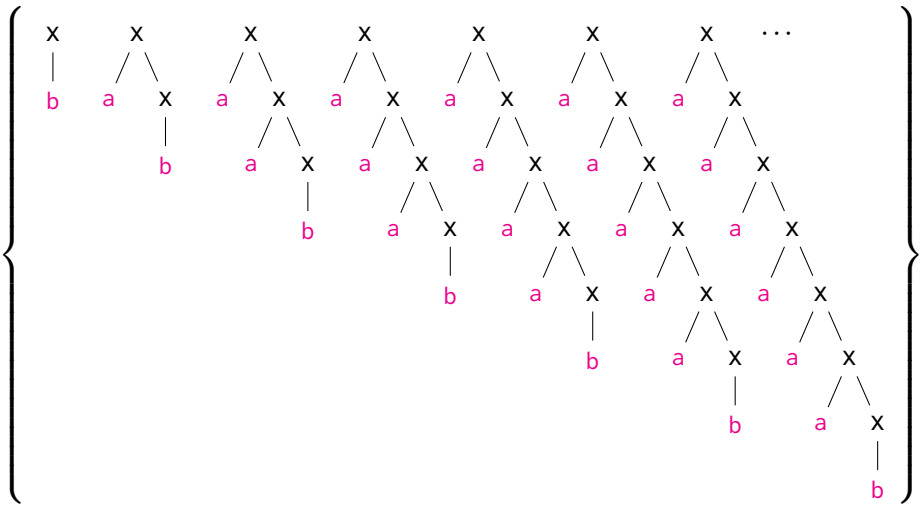
4.1 所有推导的总值

例10. 考虑以下编码几何级数的程序：

```
p = {
56  x += b.
57  x += a * x.
```

该程序中项 x 的含义是一个半环值，我们将其记为 $\mathfrak{p}[\![\mathtt{x}]\!]$。$x$ 关于 a 和 b 的值为 $\mathfrak{p}[\![\mathtt{x}]\!] = \mathtt{b} + \mathtt{a}*\mathtt{b} + \mathtt{a}*\mathtt{a}*\mathtt{b} + \mathtt{a}*\mathtt{a}*\mathtt{a}*\mathtt{b} + \mathtt{a}*\mathtt{a}*\mathtt{a}*\mathtt{a}*\mathtt{b} + \mathtt{a}*\mathtt{a}*\mathtt{a}*\mathtt{a}*\mathtt{a}*\mathtt{b} + \cdots$。我们通过枚举规则 $\mathfrak{p}$ 下 x 的推导集合，得到这个（无限）序列。一个项的推导构成一个集合，记作 $\mathfrak{p}\{\mathtt{x}\}$。下面，我们展示项 x 的 7 个（无限多个中的）推导。可以看到，每个推导叶子处的值都对应于上面求和中的各项。

$\mathfrak{p}\{\mathtt{x}\} =$



从较高层次来看，我们通过反复将匹配子目标的规则进行替换，来生成这些推导。[49] 下面，我们通过将规则嵌套在其他规则中，展示这些规则替换。每条嵌套规则都编码了一棵推导树。

[49]我们将在定义 23。

58 x += b. 59 x += a * (x += b). 60 x += a * (x += a * (x += b)). 61 x += a * (x += a * (x += a * (x += b))). 62 x += a * (x += a * (x += a * (x += a * (x += b))))). 63 x += a * (x += a * (x += a * (x += a * (x += a * (x +=b)))))). 64 x += a * (x += a * (x += a * (x += a * (x += a * (x +=a*(x +=b)))))).[65] ⋯

接下来，我们可以用值概括每个推导——也就是说，将其叶子的 e 值相乘所得的乘积。[50] 这样做，我们便能还原无穷级数的各项：

66 x += b。67 x += a * b。68 x += a * a* b。69 x += a * a* a * b。70 x += a * a* a * a * b。71 x += a * a* a * a * a * b。72 x += a * a* a * a * a * b。[73] ⋯

请注意，我们能够将x 展开为一个推导集合，其中各推导的值可以求和要求满足分配律。

现在来看一个包含变量的示例程序。

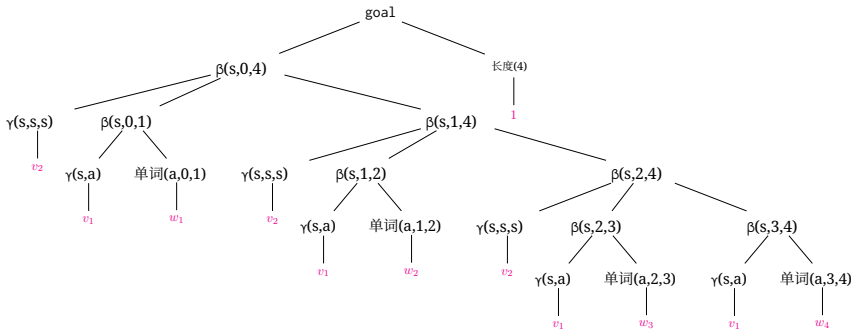
示例11. 考虑 *CKY*（示例*6*；在……下面重复出现），其中我们对原始程序进行了扩展，加入了一些输入数据：

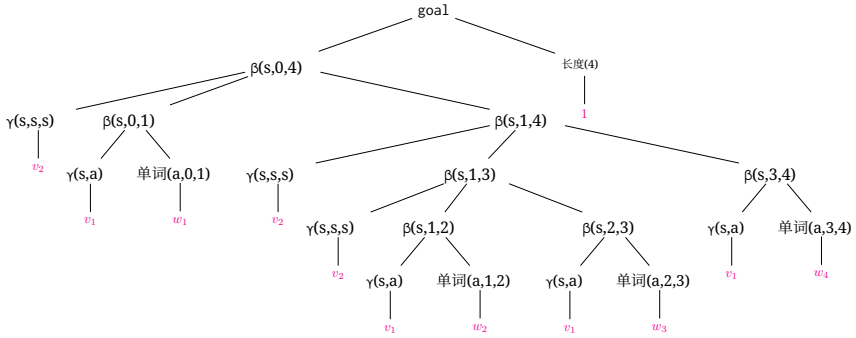
[74] β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K). 78 % 上下文无关文法规则[75] β(X,I,K) += γ(X,Y) * β(Y,I,K). [79] γ(s,a) += v_1.[76] β(X,I,K) += γ(X,Y) * word(Y,I,K). [80] γ(s,s,s) += v_2. 77 目标 += β(s,0,N) * len(N).

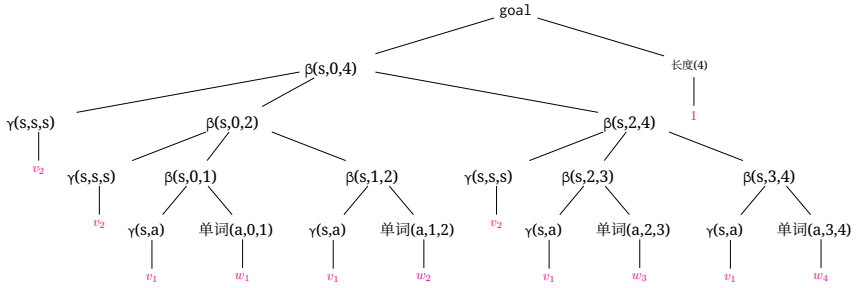
[81] % 对输入句子“a a a a”的编码[82] word(a,0,1) += w_1. word(a,1,2) += w_2. word(a,2,3) += w_3. word(a,3,4) += w_4.[83] len(4) += 1.

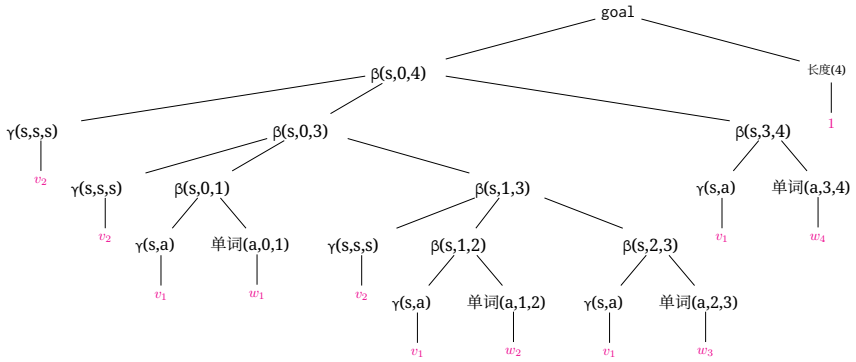
其中 $v_1, v_2, w_1, w_2, w_3, w_4 \in \mathbb{V}$。现在，考虑目标 项。在 *CKY* 程序规则下，它有五种推导：

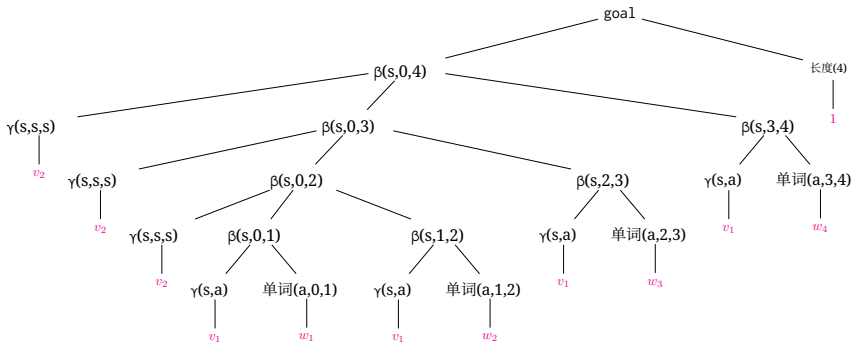
[50]该乘积按照嵌套项所表示的树进行中序遍历，以保留可能存在的非交换乘积运算的顺序。


goal
β(s,0,4)
长度(4)
1
γ(s,s,s)
v2
β(s,0,1)
β(s,1,4)
γ(s,a)
v1
单词(a,0,1)
w1
γ(s,s,s)
v2
β(s,1,2)
β(s,2,4)
γ(s,a)
v1
单词(a,1,2)
w2
γ(s,s,s)
v2
β(s,2,3)
β(s,3,4)
γ(s,a)
v1
单词(a,2,3)
w3
γ(s,a)
v1
单词(a,3,4)
w4


goal
β(s,0,4)
长度(4)
1
γ(s,s,s)
v2
β(s,0,1)
β(s,1,4)
γ(s,a)
v1
单词(a,0,1)
w1
γ(s,s,s)
v2
β(s,1,3)
β(s,3,4)
γ(s,s,s)
v2
β(s,1,2)
β(s,2,3)
γ(s,a)
v1
单词(a,3,4)
w4
γ(s,a)
v1
单词(a,1,2)
w2
γ(s,a)
v1
单词(a,2,3)
w3


goal
β(s,0,4)
长度(4)
1
γ(s,s,s)
v2
β(s,0,2)
β(s,2,4)
γ(s,s,s)
v2
β(s,0,1)
β(s,1,2)
γ(s,s,s)
v2
β(s,2,3)
β(s,3,4)
γ(s,a)
v1
单词(a,0,1)
w1
γ(s,a)
v1
单词(a,1,2)
w2
γ(s,a)
v1
单词(a,2,3)
w3
γ(s,a)
v1
单词(a,3,4)
w4


goal
β(s,0,4)
长度(4)
1
γ(s,s,s)
v2
β(s,0,3)
β(s,3,4)
γ(s,s,s)
v2
β(s,0,1)
β(s,1,3)
γ(s,a)
v1
单词(a,3,4)
w4
γ(s,a)
v1
单词(a,0,1)
w1
γ(s,s,s)
v2
β(s,1,2)
β(s,2,3)
γ(s,a)
v1
单词(a,1,2)
w2
γ(s,a)
v1
单词(a,2,3)
w3


goal
β(s,0,4)
长度(4)
1
γ(s,s,s)
v2
β(s,0,3)
β(s,3,4)
γ(s,s,s)
v2
β(s,0,2)
β(s,2,3)
γ(s,a)
v1
单词(a,3,4)
w4
γ(s,s,s)
v2
β(s,0,1)
β(s,1,2)
γ(s,a)
v1
单词(a,2,3)
w3
γ(s,a)
v1
单词(a,0,1)
w1
γ(s,a)
v1
单词(a,1,2)
w2

再一次，我们看到，每个推导的内部节点都标记为项，其叶子是半环值。每个推导的值是其叶子的乘积。要确定目标 的值，我们将其每个推导的值相加：

$$\begin{aligned}[\![\texttt{goal}]\!] = \;(&\texttt{v}_2 * \texttt{v}_1 * \texttt{w}_1 * \texttt{v}_2 * \texttt{v}_1 * \texttt{w}_2 * \texttt{v}_2 * \texttt{v}_1 * \texttt{w}_3 * \texttt{v}_1 * \texttt{w}_4 * \underline{1} \\ &+ \texttt{v}_2 * \texttt{v}_1 * \texttt{w}_1 * \texttt{v}_2 * \texttt{v}_2 * \texttt{v}_1 * \texttt{w}_2 * \texttt{v}_1 * \texttt{w}_3 * \texttt{v}_1 * \texttt{w}_4 * \underline{1} \\ &+ \texttt{v}_2 * \texttt{v}_2 * \texttt{v}_1 * \texttt{w}_1 * \texttt{v}_1 * \texttt{w}_2 * \texttt{v}_2 * \texttt{v}_1 * \texttt{w}_3 * \texttt{v}_1 * \texttt{w}_4 * \underline{1} \\ &+ \texttt{v}_2 * \texttt{v}_2 * \texttt{v}_1 * \texttt{w}_1 * \texttt{v}_2 * \texttt{v}_1 * \texttt{w}_2 * \texttt{v}_1 * \texttt{w}_3 * \texttt{v}_1 * \texttt{w}_4 * \underline{1} \\ &+ \texttt{v}_2 * \texttt{v}_2 * \texttt{v}_2 * \texttt{v}_1 * \texttt{w}_1 * \texttt{v}_1 * \texttt{w}_2 * \texttt{v}_1 * \texttt{w}_3 * \texttt{v}_1 * \texttt{w}_4 * \underline{1})\end{aligned}$$

程序中的其他项也有推导，以及由这些推导得出的值，但为使示例简洁，我们不展示它们。

接下来，我们将形式化上面示例中引入的概念。

定义 22. *Dyna* 程序的含义[51] $\mathfrak{p}$ 是一个赋值 $\mathfrak{p}[\![\cdot]\!]$ ： $\mathbb{H} \to \mathbb{V}$，[52] 它计算推导值之和。[53]

$$\mathfrak{p}[\![x]\!] \stackrel{\text{def}}{=} \begin{cases} x & \text{if } x \in \mathbb{V} \\ \mathfrak{B}[\![x]\!] & \text{if } x \in \mathfrak{B} \\ \bigoplus_{\delta \in \mathfrak{p}\{x\}} \text{yield}(\delta) & \text{otherwise} \end{cases} \tag{4.1}$$

其中 $\mathfrak{B}[\![x]\!]$ 是内置关系的赋值（位于 $\mathfrak{B}$中的项），[54] 而 $\mathfrak{p}\{x\}$ （定义 *23*，见下文）是项 x的所有推导组成的（可能无限的）集合，且 $\text{yield}(\delta) \in \mathbb{V}$ 是推导 δ 的叶子上的值的乘积（按从左到右的顺序）。注意，当 $\mathfrak{p}\{x\}$ 为无限时，(4.1) 可能未定义

[51]请注意，Dyna2 语义_（在 §D 中讨论）并非基于对推导求和。[52]有时，我们将赋值的集合记作 $\mathbb{V}^{\mathbb{H}}$ 而不是 $\mathbb{H} \to \mathbb{V}$。[53]通常， $\mathfrak{p}[\![\cdot]\!]$ 只对0赋予非零值，而这些项仅构成 $\mathbb{H}$ （不包括内置项）的有限子集。[54]与程序项不同，内置关系由其赋值直接定义。换言之，内置项的赋值是不透明的：我们没有一组规则来定义内置项——它们本质上只是外部过程调用。为简单起见，我们假设内置项的集合（即 $x \in \mathfrak{B}$）与程序中定义的项互不相交。

（例如，见 *12*）。当程序 $\mathfrak{p}$ 的含义可从上下文中明确时，我们可以写作$[\![\cdot]\!]$，作为$\mathfrak{p}[\![\cdot]\!]$的简写。

定义 23。 我们将程序中的推导 δ $\mathfrak{p}$ 定义为满足以下条件之一的树结构：

(i) $\delta \in \mathbb{V}$*(ii)* δ 具有以下形式：$x \oplus\!= v$，其中 x 是一个内置项，且 $v = \mathfrak{B}[\![x]\!]$ *(iii)* δ 是推导的乘积 $\delta_1 \circ \cdots \circ \delta_K$*(iv)* δ 具有以下形式：$(x \oplus\!= \delta_1 \circ \cdots \circ \delta_K)$，其中 *(a)* δ_k 是每个 k的有效推导 *(b)* $x \oplus\!= head(\delta_1) \cdots head(\delta_K)$ 是 $\mathfrak{p}$ 中某条规则的基化，其中 $head(\delta_k)$ 为每个 k给出 δ_k的根标签。[55]

。 。

这些情形定义了一棵带标签的树，其内部节点以规则（或内置关系）为标签，其叶子以值为标签。请注意，有效的推导可以包含变量——这意味着对其变量的任何基化也是有效的推导。该值对于推导 δ 为$\mathsf{yield}(\delta)$；我们有时写 $[\![\delta]\!]$而不写$\mathsf{yield}(\delta)$。*finally*，令 $\mathfrak{p}\{x\}$ 表示所有推导的集合，对应于项x。从技术上讲，我们想要最小的对象集合，使其满足推导的描述，因为这排除了高度为无穷的推导。[56]

[55]请注意，我们还使用$\mathsf{头部}(\cdot)$ 来访问规则的头部；重复使用这一符号并非偶然，因为每个推导都是由嵌套规则（带有基变量）构建而成的。[56]推导集合通常可以使用一种打包森林数据结构来紧凑地表示，这种结构也称为B-超图(Gallo 等人，1993）。类比而言，推导集合之于打包森林，犹如字符串集合之于有限状态自动机。在这两种情况下，我们都可以利用结构共享，高效地表示相应结构中某些可能无限的集合。打包森林通常用于自然语言处理中，以表示句法分析森林和可能的翻译输出（例如，Klein 和 Manning（2001年）、Li 和 Eisner（2009年）、Huang（2008年）以及 Eisner、Kornbluh 等人（2006年）。

定义 24. 推导的高度定义如下

$$\mathsf{height}(\boldsymbol{\delta}) \stackrel{\text{def}}{=} \begin{cases} 0 & \textit{if } \boldsymbol{\delta} \textit{ is a constant or builtin} \\ \max(\mathsf{height}(\boldsymbol{\delta}_1), \ldots, \mathsf{height}(\boldsymbol{\delta}_K)) & \textit{if } \boldsymbol{\delta} \textit{ is a product } \boldsymbol{\delta}_1 \circ \cdots \circ \boldsymbol{\delta}_K \\ 1 + \mathsf{height}(\mathsf{body}(\boldsymbol{\delta})) & \textit{if } \boldsymbol{\delta} \textit{ is a derivation by a rule} \end{cases}$$

并非所有无穷和都收敛到一个定义良好的值。如果程序中的某个给定项对推导没有定义良好的和，那么其语义也没有良好定义。在§4.4，我们描述了 ω-连续半环（Kuich，1997年）族，它们保证具有定义良好的值。然而，我们并不要求用户使用此类半环。

4.2 不动点近似

定义 25. 我们可以将 (4.1)重新整理为一个方程组，称为不动点方程[57] 对于给定的程序 $\mathfrak{p}$：[58]

$$[\![x]\!] = \bigoplus_{\substack{\mathbf{R} \in \mathfrak{p}, \\ (x \oplus\!= y_1 \circ \cdots \circ y_K) \in \Gamma(\mathbf{R})}} [\![y_1]\!] \circ \cdots \circ [\![y_K]\!] \quad \textit{for } x \in \mathbb{H} \tag{4.2}$$

*Re*称 $\Gamma(\mathbf{R})$ 为规则 $\mathbf{R}$ （定义见 §2）的所有基例的集合 .

不动点方程给出了一个递归算法，用于计算 $[\![\cdot]\!]$。该算法会通过递归计算所有 $[\![x]\!]$ 的值来求出 $[\![y_k]\!]$项；这些项 $[\![x]\!]$ 依赖于该值，并将它们代入（4.2）。然而，只要遇到循环，该算法就会陷入无限递归。

[57]一个 **不动点方程** 的形式为 $x = f(x)$ 其中 f 是一个向量值函数。[58]Dyna 2 不具有按推导求和的语义——其语义是广义版本（4.2）的任意不动点。有关进一步讨论，请参阅 §D。

关于有环不动点方程，另一个重要之处在于，它们可能存在多个解，其中包括不符合定义22的解。[59] 因此，这些不动点方程规定不足：它们是解等于定义22中潜在无限和的必要条件，但并非充分条件。请看以下示例。

示例

例 12. 以下 *Dyna* 程序表示几何级数：$[\![\mathtt{x}]\!] = 1 + a + a^2 + a^3 + \cdots$ 其中 $a = [\![\mathtt{a}]\!]$。

```
x += 1.
x += a * x.
```

几何级数中的每一项都对应于 x 的一个推导值。根据该程序，x 有无限多个推导。如果将该程序代入不动点方程(4.2)，我们会看到 x 依赖于自身：

$$[\![\mathtt{x}]\!] = [\![1]\!] + [\![\mathtt{a}]\!] \cdot [\![\mathtt{x}]\!] = 1 + a \cdot [\![\mathtt{x}]\!]$$

由于我们处于 实数半环中，可以使用减法和除法作为快捷方式来求解此方程：

$$[\![\mathtt{x}]\!] = \frac{1}{1-a}$$

然而请注意，只有当 $|a| < 1$ 时，该解才对应于推导求和（即几何级数的值）。当 $a \geq 1$ 时，$[\![\mathtt{x}]\!]$ 的推导求和语义应为 ∞。令人欣慰的是，当 $a > 0$ 时，∞ 满足不动点方程给出的必要条件：$\infty = 1 + a \cdot \infty$。请注意，

[59] 然而，在无环情形下，不动点方程为每个项定义了唯一的值 x，该值等于（有限多个）推导之和 $[\![x]\!]$（定义 22）。

此外， ∞ 的预期语义并不对应于最小不动点，因为 $\frac{1}{1-a} < \infty$。当 $a < -1$时，该级数在 $\pm\infty$之间振荡，因此没有定义良好的解；不过，不动点方程有解 $\frac{1}{1-a}$。

下面，我们再举一个欠规范的例子，它源于二次系统存在多个解：

示例13。 以下程序编码了卡特兰级数： $[\![\mathtt{x}]\!] = c + 2\,c^2 + 5\,c^3 + 14\,c^4 + 42\,c^5 + 132\,c^6 + \cdots$ 其中 $c \overset{\text{定义}}{=} [\![\mathtt{c}]\!]$。

```
86  x += c.
87  x += x * x.
```

现在，我们考虑该程序值的不动点方程。我们看到，不动点方程有两个 解：

$$
\begin{aligned}
[\![\mathtt{x}]\!] &= [\![\mathtt{c}]\!] + [\![\mathtt{x}]\!]^2 && \textit{[def. of fixpoint]} \\
[\![\mathtt{x}]\!]^2 - [\![\mathtt{x}]\!] + c &= 0 \\
[\![\mathtt{x}]\!] &\in \frac{1 \pm \sqrt{1 - 4\,c}}{2} && \textit{[by quadratic formula}^{60}\textit{]}
\end{aligned}
$$

假设 $0 \leq c \leq \frac{1}{4}$，较小的解 $\frac{1-\sqrt{1-4\,c}}{2}$对应于对推导求和。较大的解 $\frac{1+\sqrt{1-4\,c}}{2}$ 是不动点，但它不等于对推导求和。

[60]求根公式： $a\,x^2 + b\,x + c = 0 \wedge a \neq 0 \iff x = \frac{-b \pm \sqrt{b^2 - 4\,a\,c}}{2a}$。

4.3 不动点迭代

不动点迭代[61] 是一种用于近似 $[\![\cdot]\!]$ 的通用方案。该方案定义了一个近似赋值序列 $\nu^{(0)}$，$\nu^{(1)}$，……，$\nu^{(T)}$，它们逐渐趋近于$[\![\cdot]\!]$，当 $T \to \infty$。该序列定义如下

$$\begin{cases} \nu^{(0)} = \bot \\ \nu^{(t+1)} = \mathbf{T}_{\mathfrak{p}}\, \nu^{(t)} \end{cases} \tag{4.3}$$

其中 $\bot$ 是无规则程序的赋值，[62] 而 $\mathbf{T}_{\mathfrak{p}}$ 是程序 $\mathfrak{p}$**的步算子**：

$$(\mathbf{T}_{\mathfrak{p}}\, \nu)[\boldsymbol{x}] \stackrel{\text{def}}{=} \bigoplus_{\substack{\mathbf{R} \in \mathfrak{p}, \\ (\boldsymbol{x} \oplus\!\!= y_1 \circ \cdots \circ y_K) \in \Gamma(\mathbf{R})}} \nu[y_1] \circ \cdots \circ \nu[y_K] \qquad \text{for } \boldsymbol{x} \in \mathbb{H} \tag{4.4}$$

方程（4.5）只是将定义（4.2）中右侧从定义 25 抽象为赋值的函数。我们可以将不动点方程（定义 25）改写为如下的一个解，使得

$$\nu^* = \mathbf{T}_{\mathfrak{p}}\, \nu^* \tag{4.5}$$

因此，与（4.2）类似，（4.5）可能有多个不动点，也可能一个不动点都没有。实践中，通常通过某种不动点迭代来计算合适的不动点 ν^*：该迭代将 ν 初始化为 $\bot$，并反复将 ν 更新为 $\mathbf{T}_{\mathfrak{p}}\, \nu$，直到收敛到某个 $\nu^* = \mathbf{T}_{\mathfrak{p}} \mathbf{T}_{\mathfrak{p}} \cdots \mathbf{T}_{\mathfrak{p}} \bot$。[63]

[61] 在逻辑编程中，不动点迭代称为前向链或自底向上求值（Ullman，1988年；Goodman，1999；McAllester，2002年；Eisner、Goldlust 等人，2005）。[62] 注：根据定义 22，无规则程序的赋值 $\wr\!\wr[\![\cdot]\!]$ 并非完全空的赋值，$\wr\!\wr$ 完全空的赋值会将所有参数映射为值 0；相反，在 0 下，常量和内置关系可能具有非零值 $\bot = \wr\!\wr[\![\cdot]\!]$。[63] 对于某些程序，这个过程可能永远不会收敛，或者需要无限长的时间才能收敛。

命题 1。 由不动点迭代 t 在时刻 $\nu^{(t)} = \mathbf{T}_{\mathrm{p}}^{t} \bot$ 给出的赋值估计，等于高度为 $\leq t$的所有推导的总值。

证明（略）。 我们从 $\nu^{(0)} = \bot$ 开始。然后，通过扩展高度为 ≤ 0 的推导（即程序规则右部出现的常量和内置项），构造高度为 ≤ 1 的推导。从这里开始，每次步操作都会将高度为 $\leq t$ 的推导的赋值结合起来，方法是应用程序规则来推导所有高度为 $\leq t{+}1$ 的推导的值。
64
■

不动点迭代是我们在 §5中提出的算法的数学基础。假设赋值具有收敛和，不动点迭代便会逐步计算该和（直到各个高度），直至覆盖所有高度；因此，在极限情况下，它会收敛到 $[\![\cdot]\!]$的值。在后续章节中，我们将讨论并解决计算效率低下的问题，并提供实现细节。我们还将介绍一种基于牛顿法求解方程的加速不动点迭代变体。

然而，在布尔半环中，总是存在唯一的最小不动点 ν^{*}，并且对于每个项， $\boldsymbol{x}\ \nu[\boldsymbol{x}]$ 会在有限时间内到达 $\nu^{*}[\boldsymbol{x}]$。[64]这是一种动态规划： $\nu^{(t)}$ 是某个计算版本的备忘录表，其中将高度 t 作为打破循环的参数。

4.4 收敛理论

本节简要概述文献中关于半环加权系统的基本结果。本节的主要目标是回顾已有成果；本节不提出新的结果。

收敛得以保证的一个简单情形是：不动点方程是无环的（即推导集合是有限的）；此时，用不动点迭代——或者更好的是用动态规划——来计算其值并不存在问题。棘手的情形是循环方程，因为它们表示对无穷多个推导求和，而这可能并非良定义的。

我们已经看到并非所有不动点（定义25）都是（4.1）的解。然而，不动点迭代（§4.3）是一个按其高度对推导的总值求和的过程（命题1）；因此，不动点迭代趋近$[\![\cdot]\!]$，前提是它有良好定义。为了形式化所谓趋近一个值的含义，我们需要拓扑的概念；我们将在本节描述一种常用的ω-连续拓扑。

Kuich（1997年）给出了充分条件，说明不动点迭代——从一个将所有项的值赋为0的赋值开始——何时会在极限处收敛到$[\![\cdot]\!]$（定义22）。[65] 该理论假设半环是ω-连续的（定义28），幸运的是，这涵盖了自然语言处理中大多数感兴趣的半环。-连续半环类具有有用的闭包性质，可确保推导上的无限和的解等于部分和的上确

[65]我们了解到这项工作，要感谢Goodman（1998，§2.3.1–2.3.2）。

界（在适当的序关系下）。我们注意到，有些半环——包括 $\langle \mathbb{R}, +, \cdot, 0, 1 \rangle$（除非将其限制为 $\mathbb{R}_{\geq 0}$）——不是ω-连续的——这主要是因为它们包含负数，因此可能导致未定义的极限。

在定义 ω-连续半环之前，我们先在半环上定义一种序关系，因为它为半环提供了一个拓扑，使我们能够使用与收敛相关的常用数学概念，例如上确界。

定义26。 半环元素上的 自然序 （记作 $\sqsubseteq$）定义在半环$\langle \mathbb{V}, \oplus, , 0, 1 \rangle$ 上如下定义： $\forall a, b \in \mathbb{V}$： $(a \sqsubseteq b) \iff (\exists c \in \mathbb{V}: a \oplus c = b)$。自然序是预序 在 $\mathbb{V}$ 上。然而，当 $\sqsubseteq$ 定义了一个 偏序时，我们称该半环是自然有序的。

示例14 （自然有序半环的示例与非示例）。

- 在实数 （以及 整数）半环上的自然序并不定义偏序： $\forall a, b$ 总是存在 $c = (b - a)$ 满足 $a \oplus c = b$。因此， $\sqsubseteq$不满足 反对称性这一 偏序要求。

- 计数半环上的自然序（或非负实数半）与 $\leq$: $\forall a, b$ 存在 c 使得 $a \oplus c = b$ 当且仅当$b - a \in \mathbb{N}$。换句话说，如果 $a \sqsubseteq b \iff a \leq b$。

- 形式语言上的自然序与集合上的 $\subseteq$ 关系一致：对于 $L_1, L_2 \in 2^{\Sigma^\star}$，$L_1 \sqsubseteq L_2 \iff L_1 \subseteq L_2$。

命题 2。 （*Kuich，1997年，定理2.1*）：每个自然序半环都是单调的（在 $\sqsubseteq$下），且0 是其最小元（$\forall\, a \in \mathbb{V} :\ 0 \sqsubseteq a$）。

因此，对于自然有序半环，每个半环运算在 $\sqsubseteq$-格 $\mathbb{V}$上都是非递减的。自然序可以扩展到赋值（$\mathbb{V}^{\mathbb{H}}$）：令 $\nu, \nu' \in \mathbb{V}^{\mathbb{H}} :\ \nu \sqsubseteq \nu' \iff \forall \boldsymbol{x} \in \mathbb{H} :\ \nu[\boldsymbol{x}] \sqsubseteq \nu'[\boldsymbol{x}]$。接下来，我们观察到，给定程序 的步算子 $\mathbf{T}_{\mathfrak{p}}$ 是单调的：$\mathfrak{p}\ \mathbf{T}\ \nu \sqsubseteq \nu' \Longrightarrow \mathbf{T}_{\mathfrak{p}}\,\nu \sqsubseteq$。**因此，不动点迭代会在每次迭代中递增** $_{\mathfrak{p}}\,\nu'$，直到最终达到饱和；不会发生导致总和定义不明确的振荡。不过，要将这一思路严格地表述为定理，还需要补充一些技术细节；具体而言，只需半环是 $\sqsubseteq$ -连续的（定义见下文）。

定义 27. *(Kuich，1997年，第 611–612 页)*：如果半环具有满足以下性质的无穷和运算（$\bigoplus$），则称其为完备的；这些性质适用于所有可数无穷集 I：

(i) 基例：

$$\textstyle\bigoplus_{i\in\emptyset} a_i = 0,\ \bigoplus_{i\in\{j\}} a_i = a_j,\ \textit{and}\ \bigoplus_{i\in\{j,k\}} a_i = a_j \oplus a_k\ \textit{for}\ j \neq k$$

(ii) 无穷交换性和结合性：

$$\textstyle\bigoplus_{J\in\mathcal{J}} \bigoplus_{j\in J} a_j = \bigoplus_{i\in I} a_i\ \textit{where}\ \mathcal{J}\ \textit{is any partitioning of}\ I$$

(iii) 无穷分配性：

$$\textstyle c \circ \left(\bigoplus_{i\in I} a_i\right) = \bigoplus_{i\in I} (c \circ a_i)\ \textit{and}\ \left(\bigoplus_{i\in I} a_i\right) \circ c = \bigoplus_{i\in I} (a_i \circ c)\ \textit{for all}\ c \in \mathbb{V}$$

定义 28。 设 $\langle \mathbb{V}, \oplus, , 0, 1 \rangle$ 是一个 自然有序 半环，且其无穷和算子满足

$$\bigoplus_{i\in\mathbb{N}} a_i = \text{上确界}\{a_1 \oplus \cdots \oplus a_i \mid i \in \mathbb{N}\} \quad \text{对于所有无限序列} \quad \langle a_i \rangle_{i\in\mathbb{N}} \tag{4.6}$$

其中 sup[66] 是相对于序 $\sqsubseteq$取的。如果这个半环是 完备，那么我们称该半环为 ω-连续。

关于这些假设研究的权威引用是 Kuich（1997年）。这些是半环研究（例如，Goodman（1998）、Goodman（1999）、Esparza 等人（2007）、Green、Karvounarakis 等人（2007）以及 Mohri（2009年））中探讨的主要理论假设。

收敛速率分析。 当压缩映射时，可以研究不动点迭代的 $\mathfrak{p}$ 的收敛速率。上面的极限分析所作的假设较弱（即， ω-连续性），这些假设足以确保极限收敛，但无法刻画其收敛速率。

要在我们的设定中使用压缩映射分析，我们要求赋值映射集合 $\mathbb{V}^{\mathbb{H}}$ 是一个完备[67] 度量空间——也就是说，它必须配备一个距离度量 ρ: $\mathbb{V}^{\mathbb{H}} \times \mathbb{V}^{\mathbb{H}} \to \mathbb{R}_{\geq 0\infty}$ 例如，在实数半环的情况下，我们可以使用切比雪夫距离$\rho(\nu, \nu') = \max_{\boldsymbol{x}\in\mathbb{H}} |\nu[\boldsymbol{x}] - \nu'[\boldsymbol{x}]|$.[68] 完备度量空间要求每个柯西序列 $\nu^{(0)}, \nu^{(1)}, \ldots$ 都有一个同样属于 $\mathbb{V}^{\mathbb{H}}$ 的极限。若对于任意小的距离 $r > 0, \exists N > 0$ ，均满足以下条件，我们就称 $\nu^{(0)}, \nu^{(1)}, \ldots$ 为柯西序列：

[66]偏序集**中子集** sup S的 $S \subseteq P$上确界 $\langle P \sqsubseteq \rangle$是 P中大于或等于 S 中每个元素的最小元素（如果存在这样的元素）。上确界有时也称为**最小上界**。[67]请注意，这里的完备与完备半环（定义 27）中的含义不同。[68]实数半环还可以采用许多其他距离度量，例如欧几里得距离和曼哈顿距离。

$\forall m,\ n > N,\ \rho\big(\nu^{(n)},\ \nu^{(m)}\big) < r$。我们称针对给定程序 $\mathbf{T}_{\mathfrak{p}}$ 的步操作 $\mathfrak{p}$

是一个**压缩映射**当且仅当

$$\forall \nu, \nu' \in \mathbb{V}^{\mathbb{H}} \colon \rho(\mathbf{T}_{\mathfrak{p}}\, \nu, \mathbf{T}_{\mathfrak{p}}\, \nu') \leq \lambda \cdot \rho(\nu, \nu')$$

具有**压缩率** $\lambda < 1$。[69] 直观地说，压缩映射是一个函数：对于任意一对点 $\langle \nu,\ \nu' \rangle$，应用该函数后，它们之间的距离至少缩小 λ倍。请注意，并不能保证$\mathbf{T}_{\mathfrak{p}}$ 对所有程序都是压缩映射 $\mathfrak{p}$。

如果$\mathbf{T}_{\mathfrak{p}}$ 是压缩映射，那么**巴拿赫不动点定理**保证不动点迭代序列 $\nu^{(0)},\ \nu^{(1)}$..... 是柯西序列，并以线性速率收敛（作为 n的函数）到唯一的不动点 ν^*:

$$\rho\big(\nu^{(n)}, \nu^*\big) \leq \lambda^n \cdot \rho\big(\nu^{(0)}, \nu^*\big) \tag{4.7}$$

请注意，量 $\rho\big(\nu^{(0)},\ \nu^*\big)$ 在（4.7）的右侧无法计算；幸运的是，我们可以放宽这个界，使其变得可计算：用一个可以计算的量替换该量，即第一次更新的大小 $R = \rho\big(\nu^{(0)},\ \nu^{(1)}\big)$。这就得到了所谓的**先验界**:

$$\rho\big(\nu^{(n)}, \nu^*\big) \leq \frac{\lambda^n}{1-\lambda} \cdot R \tag{4.8}$$

重新整理该界，可以得到确保与 ν^* 之间的距离为 $\leq \epsilon$所需的迭代次数上界（n）：

[69]压缩映射是利普希茨连续函数的一种特殊情况，即函数将自身映射到自身（也就是说，函数的定义域和值域相同）。因此，常量 λ 有时也称为利普希茨常数。

$$n \geq \frac{\log\left(\frac{\epsilon\,(1-\lambda)}{R}\right)}{\log \lambda} \quad \Longrightarrow \quad \rho\left(\nu^{(n)}, \nu^{*}\right) \leq \epsilon \tag{4.9}$$

另一个有用的界是**后验界**:

$$\rho\left(\nu^{(n)}, \nu^{*}\right) \leq \frac{\lambda}{1-\lambda} \cdot \rho\left(\nu^{(n)}, \nu^{(n-1)}\right) \tag{4.10}$$

后验界将第 n 次迭代时更新的幅度与到 ν^{*}的剩余距离联系起来。后验界表明，如果最近一次更新（在第 n次迭代时）幅度很小，我们就接近解了。因此，后验界给出了一个近似终止条件。

$$\frac{\lambda}{1-\lambda} \cdot \rho\left(\nu^{(n)}, \nu^{(n-1)}\right) \leq \epsilon \quad \Longrightarrow \quad \rho\left(\nu^{(n)}, \nu^{*}\right) \leq \epsilon \tag{4.11}$$

但在实际 中，我们不知道 λ，所以阈值只针对 $\rho\left(\nu^{(n)}, \nu^{(n-1)}\right)$。

总结

本节将 Dyna 程序的语义形式化为半环加权推导之和。我们介绍了不动点迭代，这是我们将在 §5 中开发的执行算法的数学基础。我们概述了不动点迭代收敛到 $[\![\cdot]\!]$ 的充分条件。

第5章

推理算法

本节将介绍在 Dyna 程序中执行推理（即计算 $[\![\cdot]\!]$）的一般算法。正因有这些通用算法，Dyna 程序才不需要规约实现细节：程序员提供 逻辑 的规约，而 Dyna 系统则负责 控制流。我们将介绍几种算法：

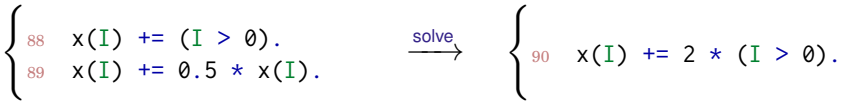
- 在 §5.1中，我们介绍三种复杂程度递增的不动点迭代实现：朴素方法（§5.1.1）、半朴素方法（§5.1.2）和优先化方法（§5.1.3）。

- 在 §5.2中，我们接着介绍一种加速的不动点算法，它基于牛顿法的一个版本，用于求解半环上的不动点方程（Esparza 等，2007）。

- 在 §5.3中，我们介绍一种用于某些线性递归程序的专用求解器。加速算法将此专用求解器用作子例程。

• 在 §5.4中，我们介绍一种统一的混合算法，它通过将大型程序拆分为较小的子程序，并使用最适用的方法分别求解这些子程序，从而求解大型程序。

解的表示。 回顾 §2 ，我们将查询的答案表示为一种受限形式的程序，其中每条规则都是简单规则。如果一条规则的体只包含常量值和内置项，则称其为简单规则。我们将仅由简单规则组成的程序称为 **简单程序**。在本章中，我们使用简单程序来表示赋值函数 $[\![\cdot]\!]$。这样，我们就能简洁地表示某些具有无限赋值的程序的解，例如[70]



这是对 Eisner、Goldlust 等人（2005年）先前工作的推广。该工作要求将解表示为一个哈希表，将基项映射到其值。用我们的符号表示法来说，这相当于要求解中的每条规则都具有基头，且其体为一个常量值。该假设的结果是，他们只能表示具有有限个非 0 项的程序的解。

我们将在算法中使用简单程序来表示中间赋值函数。我们将在本节的算法中使用结果流操作（见 §3.1），因为这些操作使我们能够无缝地操作这些程序表示。

[70]我们在 §A 中提供了更多示例。

5.1 不动点迭代算法

在 §4.3 中，我们讨论了不动点迭代的数学基础。这里，我们将以过程化的方式描述该算法，并讨论实现细节。在逻辑编程文献中，不动点迭代也称为 自底向上求值 或 前向链。[71,72] 我们将介绍三种复杂程度递增的算法：朴素算法（§5.1.1）、半朴素算法（§5.1.2）和优先级算法（§5.1.3）。我们不建议使用朴素算法（§5.1.1）和半朴素算法（§5.1.2）；将它们纳入本书是为了通过教学阐释引出优先级算法（§5.1.3）。我们的优先级不动点迭代版本将 Eisner、Goldlust 等人（2005 年）的前向链接算法推广到了支持非范围受限规则、内置关系和延迟约束。

5.1.1 朴素不动点迭代

本节介绍我们最简单的不动点迭代实现。在算法6提供的伪代码中可以看到，得益于我们用于操作流的抽象，该算法对数学定义（4.4）的翻译极其直接（§3.1）。[73]

在代码中，我们将**图表** $\mathtt{m}$ 初始化为空程序 $\wr\wr$（即不包含任何规则的程序），因此其赋值等于 $\bot$，如（4.3）所示。然后我们重复

[71]例如，Ullman（1988）、Goodman（1999）、McAllester（2002）以及 Eisner、Goldlust 等人（2005）。[72]我们的版本最接近 Eisner、Goldlust 等人（2005）的工作，因为它已推广到支持半环值。[73]这些抽象让我们能够指定每种推理算法，仿佛程序规则中不含变量一样。

算法 6 朴素：通过（朴素）不动点迭代 solve 程序 $\mathfrak{p}$

```
1.  定义朴素(p):
2.    ▷ 此过程在函数 (4.3) step 上实现不动点迭代p。
3.    m ← ⨌                          ▷ ⨌ 是一个没有规则的程序，其赋值等于 ⊥ 如 (4.3)中所示。
4.    当 真时:
5.      m′ ←步骤p(m)                                          ▷ 执行步操作
6.      如果 m = m′: 返回 m
7.      m ← m′

8.  过程 步p(m) 实现 Tp(ν) 来自 (4.4)；此处，赋值 ν 表示为
      一个程序 m。
9.  定义步p(m):
10.   m′ ← ⨌
11.   对于 (x ⊕= y_1 ∘ ⋯ ∘ y_K) ∈ p:
12.     m′[x] ⊕= m[y_1] ∘ ⋯ ∘ m[y_K]
13.   返回 m′
```

应用步函数step$_{\mathfrak{p}}$，该函数实现了$\mathbf{T}_{\mathfrak{p}}$（见（4.4）），直到达到不动点（即满足第6.6 行中的条件）。

我们看到，step$_{\mathfrak{p}}(\mathfrak{m})$ 实现了$\mathbf{T}_{\mathfrak{p}}(\mathfrak{m}[\![\cdot]\!])$，它通过实例化程序 $(x \oplus= y_1 \circ \cdots \circ y_K) \in \mathfrak{p}$ 的体并使其匹配当前图表 $\mathfrak{m}$，借助结果流抽象来汇总每条规则的贡献：$\mathfrak{m}[y_1] \circ \cdots \circ \mathfrak{m}[y_K]$。第 6.12 行使用 $\mathfrak{m}'$ 将当前规则的贡献添加到下一次迭代 $\mathfrak{m}'[x] \oplus= \mathfrak{m}[y_1] \circ \cdots \circ \mathfrak{m}[y_K]$。不难验证，$\mathfrak{m}$ 和 $\mathfrak{m}'$ 始终是简单程序。

我们注意到，算法6的运行时间并未达到理想水平。在接下来的章节中，我们将从多个方面对其进行改进。

终止条件。我们算法的终止条件（第 6.6 行）为 $\mathfrak{m} = \mathfrak{m}'$。在以下情况下可以保证终止：

- 有限无环程序（无论半环为何）。

- 在某个半环上的有限程序，该半环保证在有限次迭代后达到饱和。-闭半环（定义 19）这一类满足这一条件；其中包括许多重要的半环，例如布尔半环和优化半环（§3.3）。然而，它不包括许多重要的半环，例如实数半环。

近似终止条件。 定义近似终止条件以替换第 6.6 行 取决于具体的半环。[74] 对于实数半环，一个自然的选择是 $\forall x\colon \quad |\,\mathfrak{m}[x] - \mathfrak{m}'[x]\,| \leq \epsilon$ 对于某个较小的阈值$\epsilon \geq 0$。对于其他半环，$|\,v - v'\,|$ 应替换为适当的距离度量 $\rho(v,\ v')$。这一选择受到巴拿赫不动点定理中的后验界（4.10）的启发（该定理在第 §4.4 节讨论）。达到阈值 ϵ 所需的不动点迭代次数可能难以界定，除非步函数是压缩映射（§4.4）。

5.1.2 半朴素不动点迭代

该半朴素算法是算法6的一种更高效的实现。这一改进基于如下观察：在迭代 $h+1$上，对于在 $\mathfrak{m}^{(h)}$ 里面的项，其许多值保持不变；然而，步算子会重新推导它们用于 $\mathfrak{m}^{(h+1)}$。该半朴素算法细化了高度归纳，即精确地推导高度为 $=h$ 而非 $\leq h$ 的推导集合，在

[74] 我们的 实现提供了用于为给定半环定义近似相等性的钩子 .

每一步。这样做可以提高效率，因为每一步都无需重新推导更低高度的推导。

伪代码见算法 7。该算法的结构类似于算法 6，但除了 $\mathfrak{m}$ （以及 $\mathfrak{m}'$）之外，我们还维护新高度 $\mathfrak{d}$ （以及 $\mathfrak{d}'$）的值。可以直接验证以下不变量： $\mathfrak{m}$、 $\mathfrak{d}$、 $\mathfrak{m}'$和 $\mathfrak{d}'$ 都是简单程序。我们将在命题 3中论证该算法的正确性。

算法 7 半朴素：通过半朴素不动点迭代求解程序 $\mathfrak{p}$

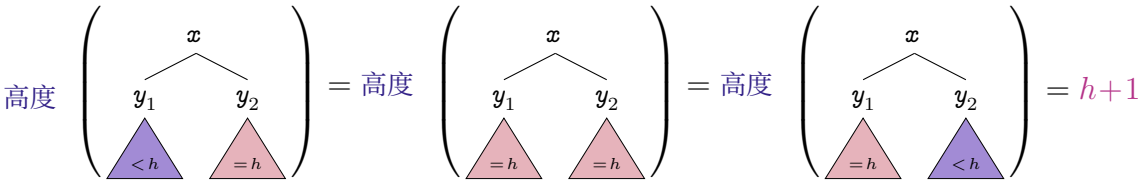
```
1.  定义半朴素(𝔭):
2.      𝔪 ← ⦇⦈
3.      𝔡 ← 步骤𝔭 (𝔪)                                  ▷ 使用算法 6 中的步骤进行初始化
4.      当 真:
5.          ⟨𝔪′, 𝔡′⟩ ← 步′𝔭(𝔪, 𝔡)
6.          如果 𝔪 = 𝔪′: 返回 𝔪
7.          ⟨𝔪, 𝔡⟩ ← ⟨𝔪′, 𝔡′⟩

8.  ⦇增量步函数；将更改 𝔡 应用于图表 𝔪 并推导出新的更改 𝔡′。
9.  定义 步′𝔭(𝔪, 𝔡):
10.     𝔡′ ← ⦇⦈                  应省略常量和内置项 ;
                                 这些由 第 7.3 行。
11.     𝔪′ ← 𝔪 ⊕ 𝔡                                    ▷ 累积更改
12.     针对 (x ⊕= y_1 ∘ ⋯ ∘ y_K) ∈ 𝔭:
13.         针对 k ∈ [1: K]: [75]
14.             𝔡′[x] ⊕= 𝔪′[y_[:k)] ∘ 𝔡[y_k] ∘ 𝔪[y_(k:]]
15.     返回 ⟨𝔪′, 𝔡′⟩
```

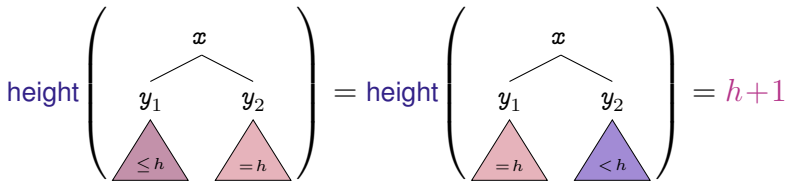
命题 3。 在……里面第 h 次迭代中，算法 7 *(第 7.7 行)*， $\mathfrak{m}$ 的值是高度 $<h$上的赋值函数，而 $\mathfrak{d}$ 的值是限制在高度 $=h$上的赋值函数。

证明（略）。根据定义 24，我们得到一个 $=h+1$ 推导：将一条规则应用于一个集合中的至少一个 $=h$ 推导以及任意数量的[75]为了提高效率，连接算法应将 $\mathfrak{d}[y_k]$ 用作最外层循环。

$<h$ 推导。举例来说，考虑一个有2 个子节点的推导。划分产生高度 $=h+1$ 推导的事件，一种相当直接的方法是分为以下三种情况，它们找出了 $=h$ 子推导可能出现的所有位置：

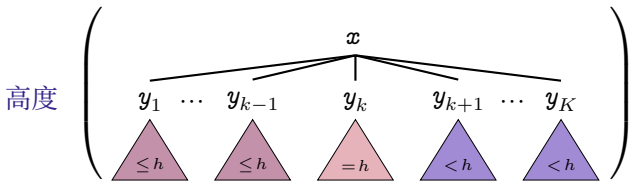


然而，对上面事件进行更高效——尽管不太直观——的划分方式，是将前两种情况合并为一种情况（$\leq h$），从而得到两种情况而不是三种：



我们在这里所做的是，将事件与其最右侧的 $=h$ 子推导

联系起来。[76] 我们在下面描述一般情况（对于 $k = 1 \dots K$）：



(5.1)

我们将这种对事件进行分组的策略称为 **游标技巧**。我们将最右侧 $=h$ 子树的位置称为游标，它在给定规则体中从左向右移动，以控制更新中使用每个子目标值的哪个版本（$\mathfrak{m}$、 $\mathfrak{d}$或 $\mathfrak{m}'$）。

[76]当我们考虑子节点多于两个的推导时，后一种分解的优势会更加明显。后一种分解需要考虑 K 种情况，而前一种则有 $\mathcal{O}(2^K)$ 种，因为它会考虑最大高度可能出现的每个位置子集。

算法 7将这一构造具体化如下。函数 step′通过对程序中的每条规则应用一次，推导出下一高度（$=h+1$）对应的推导集合的值：对于给定规则 $x \oplus\!= y_1 \circ \cdots \circ y_K \in \mathfrak{p}$，我们考虑将 $\mathfrak{d}$ 中的每一项依次填入最右侧 $=h$ 子节点。为此，我们遍历游标 k 的值，从 1 到 K (第 7.13 行)，并执行第 7.14 行的更新（在下方重复列出）：

$$\mathfrak{d}'[x] \oplus\!= \mathfrak{m}'[y_1] \circ \cdots \circ \mathfrak{m}'[y_{k-1}] \circ \mathfrak{d}[y_k] \circ \mathfrak{m}[y_{k+1}] \circ \cdots \circ \mathfrak{m}[y_K]$$

我们看到该更新与（5.1）具有相同的模式，这就是为什么 $\mathfrak{d}'$ 对应于由高度为 $=h+1$ 的推导集合所贡献的值。

■

例 15. 我们给出一个游标技巧实际运用的简单示例。

```
91 x  += y * y*y.92
y += 2.93   y +=z.94
z += 1.
```

我们是gin 通过初始化 $\mathfrak{m} = \wr\int$ 和 $\mathfrak{d} =$步骤$\mathfrak{p}$（$\mathfrak{m}$）。然后，在第一次迭代上：

$$\mathfrak{m} = \wr\int \quad \mathfrak{d} = \begin{cases} \texttt{y += 2.} \\ \texttt{z += 1.} \end{cases} \xRightarrow{\text{step}'} \mathfrak{m}' = \begin{cases} \texttt{y += 2.} \\ \texttt{z += 1.} \end{cases} \qquad \mathfrak{d}' = \begin{cases} \texttt{x += 2 * 2 * 2.} \\ \texttt{y += 1.} \end{cases}$$

接下来，我们看到游标技巧在 $\mathfrak{d}'$中付诸应用：

$$\mathfrak{m} = \begin{cases} \texttt{y += 2.} \\ \texttt{z += 1.} \end{cases} \qquad \mathfrak{d} = \begin{cases} \texttt{x += 8.} \\ \texttt{y += 1.} \end{cases} \xRightarrow{\text{step}'} \mathfrak{m}' = \begin{cases} \texttt{x += 8.} \\ \texttt{y += 3.} \\ \texttt{z += 1.} \end{cases} \qquad \mathfrak{d}' = \begin{cases} \texttt{x += 1 * 2 * 2.} \\ \texttt{x += 3 * 1 * 2.} \\ \texttt{x += 3 * 3 * 1.} \end{cases}$$

接下来，我们应用更新，并看到x 具有正确的值 27 在.....里面 $\mathfrak{m}'$：

$$\mathfrak{m} = \begin{cases} \texttt{x += 8.} \\ \texttt{y += 3.} \\ \texttt{z += 1.} \end{cases} \qquad \mathfrak{d} = \begin{cases} \texttt{x += 19.} \end{cases} \xRightarrow{\text{step}'} \mathfrak{m}' = \begin{cases} \texttt{x += 27.} \\ \texttt{y += 3.} \\ \texttt{z += 1.} \end{cases} \qquad \mathfrak{d}' = \wr\int$$

半朴素半朴素 在下一次迭代时停止，得到最终图表：

$$\mathfrak{m} = \begin{cases} \texttt{x += 27.} \\ \texttt{y += 3.} \\ \texttt{z += 1.} \end{cases}$$

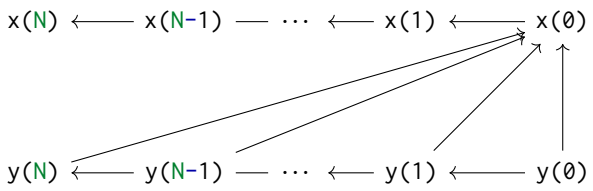
效率。 在逻辑编程中，值始终是布尔值；因此，一旦找到给定项的单个推导，我们就不再需要考虑扩展。换句话说，一旦某个项有了单个推导，它的值就已收敛——也就是说，它已达到不动点。因此，半朴素策略在布尔半环中特别高效，因为它只执行与变化量成正比的工作 $\mathfrak{m}$，并且不会执行任何不必要的规则应用。[77]然而，考虑一般的半环值时，我们会发现半朴素算法可能效率低下。特别是，半朴素算法会在项的值尚未收敛之前就将其合并，并且每次发现某个项有更高的推导时，都会重复进行合并。

示例 16。 下面给出一族无环程序的示例，其规模关于参数 n 呈线性增长。对于这族程序，由于不必要的重新传播，半朴素算法需要 $O(n^2)$ 时间。[78]

```
121  x(I+1) += x(I), I < n.
122  x(1) += y(I).
123  y(I+1) += y(I), I < n.
124  y(1) += 1.
```

我们可以将各项之间的直接依赖关系可视化为图

[77]这要求修改第7.12行中遍历规则的循环，以便只枚举基化非0的规则。[78]我们使用一种简写，允许 _ 某些内置关系（例如I+I）就地书写：x(I+1) += x(I), I < n译为x(I') += x(I)，I'是I+1，I < n。



从较高层面来看，之所以半朴素以二次时间运行，是因为x(0)的值将更新 $\mathcal{O}(n)$ *times*，因此依赖于x(0) 的项将更新$\mathcal{O}(n)$ *times*。经过预热期后， ∂ 将有 $\mathcal{O}(n)$ 个值在每次迭代中被传播。这些大幅变化会发生 $\mathcal{O}(n)$ *times*后才收敛，从而得到二次时间。

在下一节中，我们将提出另一种算法，使我们能够按照拓扑排序顺序计算该程序中每一项的值，从而在此示例中将运行时间大幅缩短至 $\mathcal{O}(n)$ 。

5.1.3 优先级固定点迭代

在本节中，我们将介绍一种更灵活的半朴素求值策略，该策略使用议程（工作列表），以安全地消除将高度用作同步屏障的做法。这样做的好处是可以避免重新传播：我们不必让所有项从高度 $\leq h$锁步前进到高度 $\leq h+1$，而是可以让某些项远远领先于其他项。理想情况下，我们会延迟传播，直到每个项都达到其不动点值。一组项收敛后，我们就不必继续将它们与其他收敛项组合（只需将一组收敛项组合一次）。在有环图中，不可能阻止更新，直到达到不动点（否则会发生死锁），但我们可以专注于更新，优先传播

算法 8 议程：通过基于议程的不动点迭代 solve 程序 $\mathfrak{p}$

```
1.  定义议程(p):
2.    m ← ∅;  m′ ← ∅
3.    ð ← 步骤ᵖ(m)                                  ▷ 使用以下内容初始化步骤 来自算法 6
4.    当 ð ≠ ∅:                                     可能包含延迟约束
5.      (y′ ⊕= v) ← ð.弹出()                        ▷ 从 ð中选取任意更新以应用
6.      m′[y′] ⊕= v
7.      如果 m′[y′] = m[y′]: 继续                   ▷ 更新没有效果；无需传播[80,81]
8.      对于 (x ⊕= y_1 ∘ ⋯ ∘ y_K) ∈ p:
9.        对于 k ∈ [1: K]:
10.         ð[x] ⊕= m′[y_[:k)] ∘ v[y_k = y′] ∘ m[y_(k:]]
11.     m[y′] ⊕= v                                  ▷ 现在， m = m′
12.   返回 m
```

有向循环子图内的内容，然后再继续。换言之，我们可以优先处理强连通分量（定义 12）内的更新，使其达到不动点后，再推进到下一个强连通分量。[79]

使这种推广成为可能的关键，是将 $\eth$ 从算法7推广为在提交更改到 $\mathfrak{m}$之前将其缓冲。这样一来，一些项可以将其赋值推进到更高的高度，而其他项则被延后。在 $\eth$ 中累积的值，对应于尚未累积到 $\mathfrak{m}$中的各个高度之和——它不再局限于单个下一个高度。假设没有更新会被无限期延后，议程策略的赋值估计应当收敛到 $[\![\cdot]\!]$（在与半朴素收敛的相同条件下）。

用于确定从**议程**中弹出哪个**更新** $\eth$ (第 8.5 行) 的策略称为**优先级策略**。它通常通过优先队列实现。我们将在 §5.1.4 中介绍优先级函数的设计指南。

[79] Goodman（1998）将强连通分量称为“循环桶”。

我们算法的伪代码见算法 8。[82] 需要指出的是，算法 8 与 Eisner、Goldlust 等人（2005）提出的基于待办队列的算法非常接近。为了支持非地面更新，我们对他们的算法做了细微但必要的修改。[83] 与算法 7 类似，基于待办队列的算法使用游标技巧，[84] 该技巧——至少从概念上来说——需要为从议程中弹出的驱动项的值保留多个版本 y' （$\mathfrak{d}$）：新版本 $\mathfrak{m}'[y']$、变更 $\mathfrak{d}[y']$ 和旧版本 $\mathfrak{m}[y']$。我们将第 8.10 行上的查询 $\mathfrak{m}'[y_{[:k)}]$ 和 $\mathfrak{m}[y_{(k:]}]$ 称为乘客查询。如果y' 不匹配多个子目标，那么乘客查询将看到变更缓冲区中尚未待处理的、最新的值。更新 $y' \oplus\!= v$ 只有在通过程序中的规则仔细传递之后（使用游标技巧避免重复计数该更新），才会在第 8.11 行提交到 $\mathfrak{m}$；此时 $\mathfrak{m} = \mathfrak{m}'$。Eisner、Goldlust 等人（2005）采用的策略基本相同。不过，他们的策略只适用于基更新，无法推广到非地面更新的情况

[80]请注意，对于第 8.7 行的这项相等性测试，允许出现假阴性是安全的，但不能出现假阳性。不过，假阴性可能会导致需要更多次迭代，$\mathfrak{d}$ 才会变为空（第 8.4 行）。[81]第 8.7 行的相等性测试比较的是 y' 在 $\mathfrak{m}'$ 和$\mathfrak{m}$各自指定的查询结果。当 y' 不是基项时，由于我们比较的是结果流是否相等，这项测试可能不再能在常数时间内运行。[82]与算法 7 类似，可以直接验证以下不变量：$\mathfrak{m}$、$\mathfrak{d}$、$\mathfrak{m}'$和 $\mathfrak{d}'$ 都是简单程序。[83]关键区别在于，我们将图表和议程表示为程序，而 Eisner、Goldlust 等人（2005）要求将它们分别表示为哈希表，将基项映射到半环值。这种表示上的差异使我们能够高效地表示某些无限赋值函数。一个更表面的区别是，我们维护图表的两个版本 $\mathfrak{m}$ 和 $\mathfrak{m}'$，而 Eisner、Goldlust 等人（2005）则用一个局部变量表示驱动程序的旧值（即 $\mathfrak{m}$中的值）。非基项更新会导致 Eisner 、 Goldlust 等人（ 2005 ）的伪代码遍历与该更新匹配的无限 Herbrand 宇宙；但他们的实现则会在这种情况下抛出异常。[84]Eisner、Goldlust 等人（2005）中也出现了游标技巧的一个版本。

可能包含延迟约束的更新。[85]

5.1.4 优先级函数

我们在算法 8 (第 8.5 行) 的每次迭代中使用优先级策略，来确定接下来要传播哪项更新。原则上，任何不会让某个项始终无法出队的优先级策略，最终都会收敛到一个不动点（假设不动点迭代本身能够收敛）。满足这一性质的一个简单默认策略是先进先出（FIFO）；然而，FIFO 本质上就是半朴素算法，而我们在示例 16 中看到，该算法的收敛速度可能不必要地慢。

本节介绍一种称为优先级函数的特定优先级策略。其工作方式是：在 $\mathfrak{d}$ 中，为每项更新分配一个数值优先级，分配时机是更新被推入（即添加到 $\mathfrak{d}$，无论是在第 8.3 行还是第 8.10 行）。[86] 调用 $\mathfrak{d}$.pop() 时（第 8.5 行），会返回优先级最高的更新（若优先级相同，则任意选择），并将其从 $\mathfrak{d}$中移除。每项更新的优先级值可以存储在优先队列数据结构中，该结构会为 $\mathfrak{d}$ 中的更新建立索引，从而高效地执行推入和弹出操作。

Goodman（1998）提出的一种优先级排序方法，首先在布尔半环中运行程序，以估算具有非零 $\mathfrak{m}$ 的项集合。

[85]示例请参阅 §A，其中涉及非地面更新和延迟约束。[86]也可以为项分配非数值分数；唯一的要求是优先级值必须偏序。

算法 9 支撑集：估计 $\mathfrak{p}$ 的支撑集上界。

```
1.  定义 支撑集(𝔭):
2.      𝔭̄ ← 布尔化(𝔭)                ▷ 通过将 𝔭 转换为布尔半环中的程序 𝔭̄
                                       将每个非 0 常量替换为 𝔭 和真。[88]
3.      𝔪 ← solve(𝔭̄)                 ▷ 警告：当 𝔭 有无限多个真项（参见正文）。
4.      返回 析取化(𝔪)
5.  定义 析取化(𝔪):
6.      ▷ 将 𝔪 转换为一个程序，使其规则表示互不重叠的集合
7.      ▷ 通过将 𝔪 划分为哈希桶，可以加快此过程；同一哈希桶中的项
          可能重叠，而不同哈希桶中的项则不会重叠。
8.      当为真时:
9.          查找 r, s ∈ 𝔪 使得 r ∩ s 并且 r ≠ s；如果不存在这样的有序对，则跳出循环
10.         ▷ 将重叠的项替换为它们的最小上界（合并）。[89,90]
11.         𝔪 ← (𝔪 ∖ {r, s}) ∪ {合并(r, s)}
12.     返回 𝔪
```

值。[87] 算法9 给出了该方法的伪代码；该方法已扩展至 Goodman（1998）所设定的范围之外，以允许非基项。

此处任一求解器（solve）都可用于求解 $\mathfrak{p}$，例如 半朴素，或 议程（采用任意非饥饿优先级函数）。然而，要使此方法有效，我们需要 solve终止。[91] 幸运的是，我们只需要一个 上界 在支撑集上即可继续推进。在§6.1中，我们描述一种“松弛的”前向链方法，可用于估计一个上界 $\mathfrak{m}$ 在有限时间内。

[87]在 §6.1 中，我们会进一步讨论如何使用“布尔化”来估计非零项的集合。[88]在这里，布尔化保证 $\forall \boldsymbol{x} \in \mathbb{H}$: $\mathfrak{p}[\![\boldsymbol{x}]\!] \neq 0 \Longrightarrow \mathfrak{p}[\![\boldsymbol{x}]\!] = \text{true}$。[89] 这里的合并是反统一算法（Reynolds，1970；Plotkin，1970）。例如，有序对 $\boldsymbol{x} = \text{f(3,Y)}$、 $\boldsymbol{y} = \text{f(X,4)}$会被向上取整为 $\boldsymbol{z} = \text{f(X,Y)}$。[90]使用合并会损失精度，因为它会为原始 $\mathfrak{m}$ 中互不相交的项找到一个上界。原则上，我们可以完全消除合并引入的不精确性，只需将第 9.11 行替换为 $\mathfrak{m} \leftarrow (\mathfrak{m} \setminus \{r, s\}) \cup \{r \cap s, r \setminus s, s \setminus r\}$。不过，这种做法在高效表示非基项集合之差方面存在计算上的挑战（Filardo，2017，第4章）；实践中我们使用合并。[91]在某些情况下，布尔化可能会严重高估前向链需要关注的项集合。例如，对于如下程序：目标 += p(N). p(z). p(s(N)) += 0.5 * p(N)。它有无限多个非零项，但 p(s^k(z)) 的值作为 k的函数非常小： $[\![\text{p(s}^k\text{(z)})]\!] = 0.5^k$ 因此，这些值会迅速趋近于 0。所以，只需经过几次不动点迭代，我们就能很好地估计目标。不过，布尔化会枚举无限多个项，因为对于所有 $k \in \mathbb{N}$， $\mathfrak{p}[\![\text{p(s}^k\text{(N))}]\!] = \text{true}$。

算法 10 入边 和出边：惰性枚举入边与出边

弧，在程序 $\mathfrak{p}$ 的依赖图中，根据其估计支持度 $\mathfrak{m}$。

```
# 惰性枚举头部到子目标的弧
定义 入边(x):
  对 (y_1 ∘⋯∘ y_K) ∈ p[x]:
    针对 _ ∈ m̄[y_1 ∘⋯∘ y_K]:
      适用于 k ∈ [1:K]:
        yield y_k
```

```
# 惰性枚举子目标到头部的弧
定义 出边(y'):
  针对 (x ⊕= y_1 ∘⋯∘ y_K) ∈ p:
    对于 k ∈ [1: K]:
      对于 _ ∈ m̄[y_{:k} ∘ (y_k=y') ∘ y_{k:}]:
        yield x
```

根据估计得到的非0值项集合 $\mathfrak{m}$，我们可以找到项间依赖图的强连通分量的拓扑有序列表，$G=\langle V,\ E\rangle$ 其中 V 是在 $\mathfrak{m}$ 里面的项集合，并且 E 是由实例化程序中头部到子目标关系所给出的边集合。算法10描述了如何惰性枚举入弧（以及出弧）。[92,93]

优先级函数则是从每个项到其强连通分量在逆向拓扑排序中的（整数）位置的映射。确定已拓扑排序的强连通分量所用的算法5 相对于实例化程序的规模以线性时间运行 $\mathcal{O}(|V|+|E|)$ 并使用相对于项数量的线性空间 $\mathcal{O}(V)$。[94] 同一强连通分量分析也将用于我们的混合算法（§5.4）。

[92]为求完整起见，我们还纳入了出边。[93]请注意，伪代码会隐式放宽来自 $\mathfrak{m}$ 的任何延迟约束，因为它在“**对于** $_\in\mathfrak{m}[\ldots]$”循环中忽略了这些约束。这可能会高估 G 中的弧集合，因此强连通分量分析会比较粗略。我们还假设内置项会延迟，而不是抛出实例化错误（§3.1.2.2）。[94]Goodman（1998）最初提出这一方案时，忽略了一个巧妙之处：运行算法 5并不需要将图边具体化。具体来说，我们可以使用 入边（或 出边）惰性地枚举某个节点的依赖项。由于使用我们的结果流实现（§3.1.2.1），其中执行的查询和合一操作都只需要常量空间，因此这个过程只需要常量空间。与本博士论文同期，Eisner（2023年）发表了一篇详细综述，介绍了加权演绎系统中高效推理的算法。我们和 Eisner 的方法都利用了上述观察，从而使空间复杂度在 $|V|$ 中保持线性。他们的“加权循环前向链算法”与我们采用基于强连通分量的优先级排序的 议程密切相关。区别在于，我们的 议程算法通过议程协调所有控制流；而他们针对每个强连通分量的递归求解采用递归算法（类似于我们的混合算法（§5.4））。他们的方法考虑的加权演绎系统并不限于半环权重。

我们注意到，如果只需要项的一个子集目标 $\subseteq \mathbb{H}$ 的值，那么我们可能希望使用sccs(incoming,目标)的已拓扑排序强连通分量，因为它只会枚举目标所依赖的强连通分量。跳过目标节点不需要的强连通分量，可以在被省略的强连通分量收敛缓慢或半环运算开销很大的情况下带来大幅节省。

为避免增加运行时间的额外开销，我们可以使用桶队列（Dial，1969）高效地实现此处的弹出操作。采用这种优先级方案，我们可以在线性时间内使用算法16 求解示例8，其性能显著优于算法7的二次运行时间。请注意，如果 G包含环，则每个环都会在处理下一个环之前得到“求解”。如果无法保证收敛（例如，在实数半环中），这可能会阻碍其他程序分量取得进展（即导致更新饥饿）。

讨论。以下是指定优先级函数的几种替代技术：

- 用户可自定义的优先级函数（Eisner 和 Filardo，2011年）。

- 使用静态分析和理论模可满足性合成优先级函数（Colón 和 Sipma，2001年；Podelski 和 Rybalchenko，2004年；Bagnara 等，2010年）。

- 最后，我们提到，在强连通分量里面采用精心设计的优先级策略可能会带来益处。在其他相关场景中，已有研究表明，优先级策略能够提高收敛速度。[95] 未来工作可以考虑研究[95]例如

p马尔可夫决策求解器 p过程（参见 Win gate 和 Se pp[i] (2005年) 针对一项调查 y),

基于最小反馈弧集的方法，在存在环时可将其视为近似拓扑排序（例如，Baharev 等人（2016）），以及优先级启发式方法的机器学习（例如，Jiang 等人（2012 年）和 Vieira、Francis-Landau 等人（2017年））。

5.2 牛顿法

我们的 牛顿法 是一种用于求解不动点问题的加速求解器，其基础是 Esparza 等人（2007）对牛顿算法的推广，该算法用于求解半环上的不动点方程。与不动点迭代方法相比，牛顿法 的有趣之处在于，它能在一次迭代中对无穷多个推导求和。更具体地说，每次迭代都按维度 （定义见下文）递增的顺序对推导集合求和，而不是像不动点迭代那样按高度递增（命题 1）。[96]

Etessami 和 Yannakakis（2009年）指出，不动点迭代的收敛速度可能比牛顿法慢几个数量级。

循环信念传播（代表性方法见 Elidan 等人（2006））、非线性方程组（综述见 Baharev 等人（2016）），以及优越半环中的推理（例如 Felzenszwalb 和 McAllester（2007））。[96]在我们的设定中，我们使用牛顿法对半环加权推导求和；因此，我们的介绍将不同于 Esparza 等人（2007）的介绍，后者以牛顿法为起点。用于求解实数域上不动点方程 $\mathbf{x} = \mathbf{f}(\mathbf{x})$（其中 $\mathbf{x} \in \mathbb{R}^D$ 且 $\mathbf{f}: \mathbb{R}^D \to \mathbb{R}^D$）的牛顿法是一种迭代方案，其更新规则为：$\mathbf{x}^{(t+1)} \leftarrow \mathbf{x}^{(t)} + (\mathbf{I} - \mathbf{J}^{(t)})^{-1}(\mathbf{f}(\mathbf{x}^{(t)}) - \mathbf{x}^{(t)})$，其中 $\mathbf{J}^{(t)}$ 是 $\mathbf{f}$ 在 $\mathbf{x}^{(t)}$处求值所得偏导数构成的雅可比矩阵。推广到半环设定，需要改造牛顿法，以适用于不具备减法、除法或求导运算的半环。对我们的目的而言，这种改造会令人分心；有关这一连接，请参阅以下任一文献：Esparza 等人（2007）、Reps 等人（2017）和 Luttenberger（2009）。我们认为 Schlund（2016）中的介绍最为清晰。

示例17。 考虑以下基于 *Etessami* 和 *Yannakakis*（*2009年*，定理 *3.2*）的程序。该程序有一个不动点$[\![\mathsf{x}]\!] = 1$。[97] `125 x += 0。 5。 126 x += 0。 5 *x* x。`

Etessami 和 *Yannakakis*（*2009年*）表明，在这个例子中，不动点迭代比牛顿法慢得多，慢了一个指数级。具体而言，不动点迭代大约需要 2^k 次迭代才能获得 k 位精度，而牛顿法每次迭代都能提升一位精度。

定义 29. 推导 维度 $\dim(\boldsymbol{\delta})$ 的定义 $\boldsymbol{\delta}$ 如下，采用归纳方式给出：

(i) 基本情况：$\dim(\boldsymbol{\delta}) \overset{\text{定义为}}{=} 0$，若 $\boldsymbol{\delta}$ 是一个值或内置项

(ii) 若 $\boldsymbol{\delta}$ 的形式为 $\boldsymbol{x} \oplus\!= \boldsymbol{\delta}_1 \circ \cdots \circ \boldsymbol{\delta}_K$，且其中没有 $\boldsymbol{\delta}_k$ 是乘积

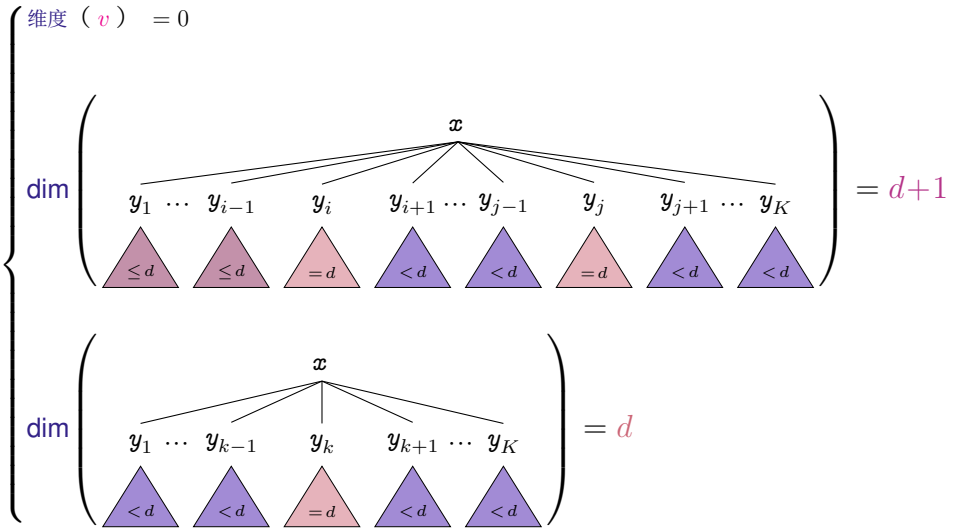
- 若 $K = 0$：$\dim(\boldsymbol{\delta}) = 0$。
- 若 $K \geq 1$：令 $d = \max(\dim(\boldsymbol{\delta}_1) \ldots, \dim(\boldsymbol{\delta}_K))$。若最大值参数唯一（即最大值不存在并列），$\dim(\boldsymbol{\delta}) \overset{\text{定义为}}{=} d$。否则，$\dim(\boldsymbol{\delta}) \overset{\text{定义为}}{=} d + 1$。

(iii) 如果 $\boldsymbol{\delta}$ 一个形如 $\boldsymbol{\delta}_1 \circ \cdots \circ \boldsymbol{\delta}_K$：维度$(\boldsymbol{\delta})$ 的乘积，则由于平局检查的定义不具结合性，乘积没有定义。

请参阅图*5.1* 了解示例。

推导的维度总是 $\leq$ 其高度，但通常是 $\ll$。维度为 d 的推导集合可能是无限的。相比之下，高度固定的推导集合是有限的。[98] 例如，几何级数中的每个推导

[97] $\frac{1}{2}x^2 - x + \frac{1}{2} = 0$根据求根公式（$a\,x^2 + b\,x + c = 0 \wedge a \neq 0 \iff x = \frac{-b \pm \sqrt{b^2 - 4\,a\,c}}{2a}$，其中$\{a \mapsto \frac{1}{2}, b \mapsto -1, c \mapsto \frac{1}{2}\}$），其解为 $\{1\}$。[98]关于固定高度推导的陈述假定……忽略了这样一种可能性：内置关系会产生无限多个高度为 $= h+1$ 的推导。



Fig图 5.1： 推导的维度示意图（定义 29）。

程序（示例 10）的维度为 ≤ 1。维度为 $d+1$ 的推导集合有一点很特别，那就是存在一种高效的线性结构，将它与维度为 $\leq d$、 $=d$和 $<d$的推导集合联系起来。这使我们能够借助专用线性求解器（§5.3），更快地扩展我们求和所涵盖的推导集合。

我们在算法 牛顿法 中给出了牛顿法的伪代码，见算法 11。[99] 牛顿法 的高层思路与半朴素 前向链非常相似：牛顿法 会跟踪其对应的 $\mathfrak{m}$和 $\mathfrak{d}$，不过在这里， $\mathfrak{m}$ 是维度为 $<d$ 的推导集合，而 $\mathfrak{d}$ 是维度为 $=d$的推导集合。为了高效地确定所有 $=d+1$推导的值，牛顿法 求解一个线性化的 程序。

我们将讨论如何求解 §5.3 中的线性子问题。不幸的是，并非所有子问题 $\mathfrak{d}'$ 都能精确求解，且并非所有半环上的求解都高效；重要

[99]我们的伪代码将 Schlund（2016，定义 3.9）针对上下文无关文法这一特殊情形的做法推广到了程序。得益于 §3.1 中的构建模块，这种推广相当顺畅。

算法 11 牛顿法：solve 程序 $\mathfrak{p}$，使用牛顿法求解

```
1.  定义牛顿法(𝔭):
2.      ⟨𝔪,𝔡⟩ ← 牛顿法_初始化(𝔭)
3.      当 真时:
4.          ⟨𝔪′,𝔡′⟩ ← 牛顿法_步进(𝔭, 𝔪, 𝔡)
5.          如果 𝔪′ = 𝔪: 跳出
6.          ⟨𝔪, 𝔡⟩ ← ⟨𝔪′, 𝔡′⟩
7.      返回 𝔪

8.  定义 牛顿法_步(𝔭, 𝔪, 𝔡):
9.      𝔪′ ← 𝔪 ⊕ 𝔡
10.     𝔡′ ← ∫∫   ~~~ 应当 省略常量和内置项；这些由 第 11.2 行 添加。
11.     对于 (x ⊕= y_1 ∘ ⋯ ∘ y_K) ∈ 𝔭:
12.         ▷ 基本情况：将至少两个维度为 =d 的推导组合起来
13.         对于 1 ≤ i < j ≤ K:
14.             𝔡′[x] ⊕= 𝔪′[y_[:i)] ∘ 𝔡[y_i] ∘ 𝔪[y_(i:j)] ∘ 𝔡[y_j] ∘ 𝔪[y_(j:]]
15.         ▷ 递归情形：推导出 =d+1 通过组合一个 =d+1 推导
              和 ≤d 推导
16.         对于 k ∈ [1: K] 其中 y_k ∉ 𝕍 ∪ 𝔅:
17.             𝔡′[x] ⊕= 𝔪′[y_[:k)] ∘ y_k ∘ 𝔪′[y_(k:]]
18.     返回 ⟨𝔪′, solve_线性(𝔡′)⟩          ▷ 使用特化线性求解器进行求解（§5.3）

19. 定义牛顿法初始化_ (𝔭):
20. ▷ 新推导的初始集合 21. 𝔪 ← ∫∫;  𝔡′ ← ∫∫ 22. 对于 (x ⊕= y_1 ∘ ⋯ ∘ y_K) ∈ 𝔭 其中至多 1 个 y_1, …,
    y_K ∉ 𝕍 ∪ 𝔅: 23. 𝔡′[x] ⊕= y_1 ∘ ⋯ ∘ y_K 24. 返回 ⟨𝔪, solve_线性(𝔡′)⟩   ▷ 使用特化
    的线性求解器求解（§5.3）
```

易于处理的特殊情况包括交换半环，以及子问题为左线性或右线性的情况。其他情况已在文献中讨论（例如，Esparza 等人（2007）、Schlund（2016）和 Reps 等人（2017年））。我们注意到，不动点迭代方法始终可用于（近似）求解 $\mathfrak{d}'$；然而，不动点迭代可能无法快速收敛（甚至完全不收敛）。

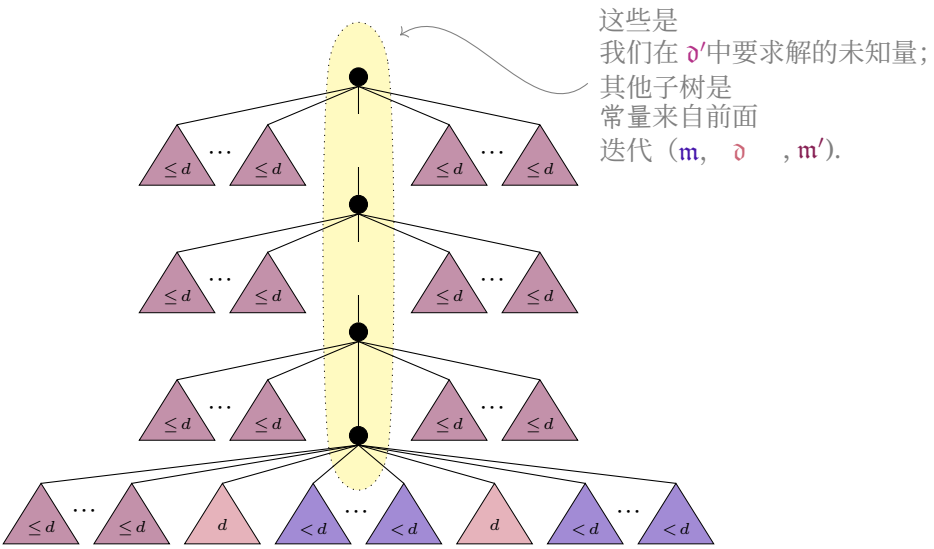
正如其他关于牛顿法的研究所指出的（例如 Schlund（2016）），将程序分解为强连通分量——正如我们在优先级排序中所做的那样

（§5.1.4）——可以显著加快算法的速度。我们的混合算法（§5.4）通过这种方式加快了牛顿法。

我们在以下命题中论证牛顿法 的正确性。

命题 4. 在迭代 d 算法 *11 (第 11.6 行)* 中， $\mathfrak{m}$ 的值是限制在维度 $<d$上的赋值函数，而 $\mathfrak{d}$ 的值是限制在维度 $=d$上的赋值函数。

证明（略）。我们首先论证，调用 $\mathfrak{d}'$ 所返回的值 牛顿法_步骤在 第 11.4 行 恰好是维度 $=d+1$ 推导的总和。下面，我们说明如何从维度 $=d$推导集合、维度 $\leq d$推导集合，以及$<d$ 推导集合归纳地构造维度 $=d+1$ 推导。



我们认为，我们用于创建新推导的方案会生成维度为 $=d+1$ 的推导。根据定义 29，要创建一个维度为 $\geq d+1$ 的推导，我们需要组合至少两个 $=d$ 推导。这就是为什么 牛顿法_ 步骤的基本情况 (第 11.13–11.14 行) 会组合至少两个 $=d$ 推导；

这表示为一条连接至少两个相邻 $=d$ 推导的规则；此步骤会创建一个维度等于 $=d+1$ 的子推导。现在，由于新推导的脊柱（即通向推导根节点的路径）上所有可以组合的子推导，其维度都为 $\leq d$，因此新推导的维度为 $=d+1$；我们可以在 牛顿法_ 步骤的递归情形 (第 11.15–11.17 行) 中看到这一点。因此，新推导的维度恰好是维度 $=d+1$。为确保对 $=d+1$ 的枚举是完备的，程序 $\mathfrak{d}'$ 包含所有能够组合这些 $\leq d$ 子推导的规则，以及至少两个 $=d$ 子推导的基本情况。程序 $\mathfrak{d}'$ 是对这个可能无限的推导集合之值的一种紧凑编码。通过调用适用于线性规则程序的专用求解器（见 §5.3）即可找到其解。

在第 11.9 行，我们看到 $\mathfrak{m}'$ 是 $\mathfrak{d}$ 和 ∂的总和。在第 11.6 行， $\mathfrak{m}'$ 覆盖$\mathfrak{m}$。因此，在第 d次 $\mathfrak{m}$ 迭代中，所有满足 $<d$的推导的总值。 ■

5.3 专用右线性求解器

本节给出一种算法，用于求解右线性程序这一特殊族，例如由牛顿法生成的那些程序（§5.2）。[100] 与不动点迭代和牛顿算法不同，该算法是精确的：它能够以闭式对右线性推导的无限集合求和。迄今为止，我们仅有的精确算法都是针对无环程序的（即那些只定义有限多个推导的程序）。我们可以高效地进行推理，因为其解可以

[100]我们注意到，本节针对右线性情形推导出的所有算法，只需稍作修改，也同样适用于左线性情形。

用有限多个正则运算来表示。右线性程序的一个简单例子是

```
x += 1.
x += 0.5 * x.
a += 0.5.
```

尽管这个程序有无穷多个x的推导，我们仍可以用初等代数求出 x 的总值（参见前文示例10）。右线性可以通过语法形式检查：每条规则只有两种可能的形式：（1）体一侧完全是常量，例如 #127；或（2）体中包含一个单独的子目标，且它是最右侧的子目标，例如 #128 （该子目标不能是内置项）。通常可以通过简单的程序变换将程序改写为右线性形式。下面的示例并非（直接）右线性。

```
x += 1. x
+=x *a. a
+= 0. 5.
```

然而，如果我们展开a 在 #131 中的子目标，并利用乘法交换性（如果程序所用的半环允许）对右侧重新排序，那么就会得到前面那个右线性程序。遗憾的是，并非所有半环都是交换的；因此，不允许对右侧进行置换。下面这个使用形式语言半环（§3.3）的示例是线性的，但不是右线性的。[101]

```
x += 1. x +=
a * x*b.
```

与前一个示例不同，我们无法将此程序转换为右线性程序，因为它的半环不是交换的。遗憾的是，

[101]该程序的解是 $\mathtt{x} = \{a^n b^n \mid n \in \mathbb{N}\}$，这不是一种正则语言。因此，它无法表示为涉及运算 $\cdot$，$\oplus$，以及$(\cdot)^*$。

右线性算法无法高效地 solve 此程序；我们需要一种通用的线性算法。

算法12采用一种通用算法来求解右线性递推，即形如$\mathbf{x} = \mathbf{b} \oplus \mathbf{A}\,\mathbf{x}$的方程，其中 $\mathbf{A} \in \mathbb{V}^{N\times N}$、$\mathbf{b} \in \mathbb{V}^{N}$。通过矩阵星运算求得解 $\mathbf{x} = \mathbf{A}^{*}\mathbf{b}$，其定义见定义 21。这种算法针对特定应用的特殊情况早已广为人知，例如：

- 矩阵求逆（高斯-若尔当消元）[102]
- 全点对最短路径（Floyd，1962；Warshall，1962；Roy，1959年）
- 有限状态自动机到正则表达式的转换（Kleene，1956）
- 传递闭包（Warshall，1962）

这些问题被统一为通常所称的**代数路径问题**（Lehmann，1977年；Tarjan，1981年）。统一这些问题的，正是半环抽象（§3.3）再次发挥了作用！选择一种半环，并使用同一个算法对图中的所有路径求和，就能解决上述所有问题以及许多其他问题。

从有用的角度来看，矩阵星运算可以计算加权有向图中路径的总权重。路径可以看作一类受限的右递归推导（定义 23）。[103] 假设

[102]Althoen 和 McLaughlin（1987 年）对高斯–约当消元算法提供了有趣的历史概述。[103]这就是算法12中采用的解释。同样，算法 11 使用这种解释对线性递归推导求和。不过请注意，算法11 要求解是线性递归的。要使用高效的右递归求解器，子目标必须可以交换（例如，采用交换半环）。

算法 12 solve_线性：通过化简为以下内容来 solve 右线性程序 $\mathfrak{p}$

矩阵星运算（定义21）。

```
定义solve_线性(𝔭):
  𝔪̄ ← 支撑集(𝔭)                                  ▷ 估计 𝔭的支撑集（算法 9）
  N ← {x | 𝔪̄[x] =真}
  一个 ← 0^{N×N};  b ← 0^N
  ▷ 以下两个循环将 𝔭的规则应用于估计支撑集。在
    在循环体中，我们组装矩阵A 和向量b，并将它们传递给
    右线性求解器。
  对于 (x ⊕= y_1 ∘ ⋯ ∘ y_K) ∈ 𝔭:
    对于 _ ∈ 𝔪̄[y_1 ∘ ⋯ ∘ y_K]:
      如果 x ⊕= y_1 ∘ ⋯ ∘ y_K 不是基项:
        抛出错误("不支持非基推理")
      否则如果 y_1 ∘ ⋯ ∘ y_K = v 其中 v ∈ 𝕍:
        b_x ⊕= v
      否则如果 y_1 ∘ ⋯ ∘ y_K = v ∘ y 其中 v ∈ 𝕍,  y ∉ 𝔅[104]:
        A_{x,y} ⊕= v
      else
        引发Error("程序不是右线性的")
  ▷ 使用矩阵星运算求解右线性系统（定义 21）。
  s ← A*b
  ▷ 以显而易见的方式将解向量s 转换为程序
  𝔪 ← ∅
  对于 x ∈ N:  𝔪[x] ⊕=s_x
  返回 𝔪
```

矩阵 $\mathbf{A}$ 是图的加权邻接矩阵。那么，$\mathbf{A}_{ik}$ 是所有长度为 $=1$ 的路径的总权重。类似地，i和 k之间所有长度为 $=2$ 的路径的总权重是（$\mathbf{A}\,\mathbf{A}$）$_{ik}$。因此，矩阵 $\mathbf{A}^{*}{}_{\text{定义}} = \sum_{i=0}^{\infty} \mathbf{A}^{i}$上的星运算会计算长度为 0、1、.....的路径的总权重，并将它们相加。

定义 21 提供了一种用于计算 $\mathbf{A}^{*}$ 的简单递归方法。其运行时间为 $\mathcal{O}(|N|^{3})$，并且可将空间复杂度优化至 $\mathcal{O}(|N|^{2})$。[105] 通过分解为强连通分量（定义 12），可以缩短运行时间。[106] 我们的混合算法（§5.4）执行了这一优化，该算法

[104]我们可以取 $v=1$以匹配该模式。[105]当 $\mathbb{V}$ 是一个环时，存在次立方算法，例如 Strassen（1969）。[106]传递闭包的基于强连通分量的算法可追溯至 Purdom（1970）。

不在算法 12中。请注意，使用右线性求解器时存在速度—精度权衡。通常，不动点迭代比精确算法更快达到可接受的准确率。[107] 最后，我们提到 solve_linear要求程序规则达到基（见 第 12.8 行）。[108]

5.4 混合算法

本章介绍了几种算法，它们各自在不同条件下表现最佳。本节将介绍一种混合算法，尝试运用现有的最佳算法来求解给定的程序 $\mathfrak{p}$。我们看到，如果 $\mathfrak{p}$ 是无环的，就可以通过议程配合拓扑排序优先级（§5.1.4）高效求解。然而，如果 $\mathfrak{p}$ 有环，那么最好将 $\mathfrak{p}$ 拆分为有环的子程序 $\mathfrak{p}^{(1)}$，…，$\mathfrak{p}^{(M)}$ 并通过强连通分量（SCC）分解来处理。[109]这种分解使我们能够分别求解每个子程序，前提是事先已计算出其所依赖的子程序的解；按拓扑序求解这些子程序即可做到这一点。通过分析 $\mathfrak{p}^{(m)}$的结构，我们可以判断哪种方法

[107]不动点迭代的收敛速度取决于 A 矩阵的性质。对于实数半环，我们可以根据 A 的谱半径来界定达到给定误差容限所需的迭代次数。[108]通常可以绕过这一限制。例如，缩写变换（§6.2）可以移除某些导致该断言失败的非基规则。例如，程序 {x(I) += 1. x(I) += 0.5 * x(I).} 是线性的，但遗憾的是，由于自由变量 I，它并不是基的。缩写将其变换为 {x' += 1. x' += 0.5 * x'.}，后者可由 solve_linear 求解。它还提供一条恢复规则 x(I) += x'，用以重建原来的非基项。[109]强连通分量分解是求解有环方程的一种常见优化（例如，Goodman（1998）、Nederhof和Satta（2008年）、Esparza 等人（2007）、Schlund 等人（2013）、Mohri（2002年）、Eisner（2023年）、Baharev等人（2016）、Chiang、McDonald 等人（2023年）以及 Purdom（1970））。

最有可能适用于求解它。以下是我们的排序：

1. solve_线性：因为它是精确的。不过，它要求使用闭半环（§5.3），至少满足以下条件之一： {右线性、左线性[100], 线性+交换性}，并且程序能够基化（参见第12.8 行）。

2. 牛顿法：因为它比不动点迭代方法收敛得更快。不过，它要求能够高效地求解所导出的线性子问题。

3. 议程：当其他方法都失败时，可以使用议程求解该子程序，因为它适用的条件最为宽泛。

算法13 给出了混合算法的伪代码，该算法按拓扑序求解每个子程序，并将先前子程序的解作为输入传递给后续子程序。它使用名为 best_solve的子例程，根据上述求解器的排名，选择求解给定子程序的最佳策略。[110]

讨论。

- Th关于支撑集终止性的讨论适用于此处（§5.1.4）。

- 使用牛顿法或不动点算法求解强连通分量（SCC）时，可能需要像本章前面所述那样使用数值收敛阈值（或迭代上限），因为

[110]一种简单的实现方式是best_solve 使用异常处理将偏好的求解器列表向下移动。当可以在任何昂贵计算之前检查必要属性时，这一过程可以加快。例如，在算法12中，可以在不实例化程序规则的情况下检查线性。

算法 13 solve：使用混合算法求解程序 $\mathfrak{p}$ 。

```
def solve(p):
  m̄ ← 支撑集(p)                                  ▷ 估计 p的支撑集（算法 9）
  m ← ⟅⟆
  对于 σ ∈ 强连通分量(输入 m):                    ▷ 强连通分量的拓扑有序列表（算法5）
    ▷ 限制 p 以计算当前分量里面的项， σ。
    p′ ← ⟅⟆
    对于 x ∈ σ:
      对于 (y_1 ∘ ⋯ ∘ y_K) ∈ p[x]:               ▷ 将特化规则添加到子程序；
        p′[x] ⊕= y_1 ∘ ⋯ ∘ y_K                    注意： p的规则可能参与多个强连通分量。
    ▷ solve 子程序 p′ 和最佳适用方法，并将解取至
      先前子程序作为输入，经由 m。[111]
    m ← 最佳_solve(p′ ⊕ m)
  返回 m
```

否则，算法13 可能会在某个子程序上无限运行，而其他子程序却取得进展。

• 展望其他章节，我们注意到，通常最好在执行前优化程序。我们特别注意到（§6.2）中的程序特化技术以及程序优化器（§8）。这些方法会生成一个语义等价程序，其执行效率更高。

总结

本章介绍了多种用于求解 Dyna 程序的算法。我们给出了不动点迭代的三种实现。我们扩展了 Eisner、Goldlust 等人（2005年）的前向链接算法，使其支持非地面更新和延迟约束。我们将 Esparza 等人（2007）的半环加权牛顿算法改编为适用于 Dyna 程序的算法，从而提供了一种强大的新求解器，[111]在 ⊕加法下面应谨慎实现，以避免复制带来的代价。

Dyna 工具箱。我们还为具有特定线性结构的程序提供了专用求解器。最后，我们给出了一种统一的混合算法，它将大型程序拆分为较小的子程序，并使用最佳适用方法分别求解这些子程序。

第 6 章

程序分析

本章介绍静态程序分析：无需实际在所有可能的输入数据集合上运行程序，就能证明程序满足某些属性。在不使用具体数据实际运行程序的情况下分析其行为，这在编程语言社区中称为抽象解释（Cousot 和 Cousot，1977年）；这可以看作对程序进行部分求值，以获取关于其可能的语义的约束。

• 抽象分析对代码生成很有用，因为代码生成可以利用任何关于某一项将具有特定类型或属性的保证。[112] 在§6.2中，我们利用抽象分析的输出来生成程序的特化（即 Dyna 到 Dyna 代码生成），这些特化可以通过移除不必要的中间项，更高效地计算项的一个子集。

[112]在面向对象宿主语言中进行代码生成时，可以利用这些保证，将该项高效地表示为一个类的实例，该类专门针对这种类型的对象，并可加快测试（例如模式匹配）。

• 抽象分析对复杂度分析很有用：如果程序存储并迭代某一特定类型的不同对象所组成的集合，那么了解该类型在具体数据上的基数可以 yield 该程序在空间和运行时需求方面的界限（§6.3）；我们的程序优化器（§8）将此类界限用作代价函数。

以下是本章内容目录。

6.1 类型分析.102 6.1.1 形式化.102 6.1.2 抽象前向链接.106 6.2 基于类型的程序特化.120 6.2.1 推断有用项的类型.122 6.2.2 特化变换.123 6.2.3 缩写变换.125 6.3 自动空间与时间分析.129 6.3.1 空间分析.129 6.3.2 时间分析.133 6.4 应用.137

6.1 类型分析

6.1.1 形式化

本节介绍一种系统，它能够推断程序 $\mathfrak{p}$ 中可能具有非0值的项集合，适用于满足用户提供的输入约束 $\mathfrak{D}$ 的所有输入数据 $\mathfrak{D}$ 。我们将非0值项的集合称为程序的支撑集。令 $\mathrm{nz}(\mathfrak{D})$ 和 $\mathrm{nz}(\mathfrak{p}(\mathfrak{D}))$ 分别表示相应参数的支撑集。由于我们不知道用于执行 $\mathfrak{p}(\mathfrak{D})$ 的具体数据 $\mathfrak{D}$ ，因此通过抽象解释计算其支撑集的一个上界 $\mathfrak{m}$ ，使得[114]

$$\underbrace{\text{新西兰}(\mathfrak{D}) \subseteq \overline{\mathfrak{D}}}_{\text{输入满足约束}} \implies \underbrace{\text{新西兰}(\mathfrak{p}(\mathfrak{D})) \subseteq \overline{\mathfrak{m}}}_{\text{支撑集上的有效上界}}$$

换句话说， $\mathfrak{D}$ 刻画了以下子集： $\mathbb{H}$ 其中输入项 $\mathfrak{D}$ 非0，而 $\mathfrak{m}$ 刻画了以下子集： $\mathbb{H}$ 其中导出项 $\mathfrak{p}(\mathfrak{D})$ 非0。因此，我们称 $\mathfrak{D}$ 为**输入类型**，而 $\mathfrak{m}$ 为**导出类型**。

现在，我们将介绍类型规范语言，以及如何用它来推理可能的项。接着 CKY 示例（示例6），我们给出输入数据的规约。不过，由于我们关注的是某个项可能具有非0 值的可能性，而不是求出其具体值，因此我们先按照下文所示转换程序。

示例18。回想一下，要分析的程序是

[113]回想一下， $\mathfrak{p}(\mathfrak{D})$ 是 $\mathfrak{p}'[\![\mathfrak{D}]\!]$ 的简写，且 $\mathfrak{p}' = \mathfrak{p} \oplus \mathfrak{D}$。[114]对于自然序半环（定义26），我们应该能够改进这种技术，从而根据其输入项值的上界，得出每个导出项值的上界。

```
135 参数：单词；长度；  γ.136  β(X,I,K) += γ
(X,Y,Z) * β(Y,I,J) *  β(Z,J,K).137  β(X,I,K) += γ(
X,Y) * β(Y,I,K).138  β(X,I,K) += γ(X,W) * 单词
(W,I,K). 139 目标  += β(s,0,N) * 长度(N)。
```

由于没有指定任何规则来定义单词、长度和 γ 项，因此可能会认为这些项的值为0。[115] 不过，参数声明指出，程序的输入将定义这些项。

CKY 程序 𝔭 在……上面可以自动“布尔化”，从而得到**类型程序** 𝔭 在……下面。[116] 该 :- 和 , 符号传统上表示布尔半环中的 ⊕= 和运算。

。

```
140 参数：单词；长度；  γ.141  β(X,I,K) :- γ
(X,Y,Z), β(Y,I,J), β(Z,J,K).142  β(X,I,K) :- γ(X,Y), β
(Y,I,K).143  β(X,I,K) :- γ(X,W), 单词(W,I,K). 144
目标 :- β(s,0,N), 长度(N)。
```

其意图是，对于所有满足 β(X,I,K) 在 CKY 程序中具有非0值的 X、I、K， β(X,I,K) 在类型程序中的值都为 true。当然，每个程序都需要输入：如果在数据 𝔇上运行 𝔭 ，那么就必须在其布尔化版本 𝔇上运行 𝔭 。赋值 𝔭(𝔇) 现在实际上指定了程序可能推导出的项的推断类型。[117]

为了不受特定输入数据的影响 𝔇，用户可以直接指定布尔化后的输入数据 𝔇（如上所述）。这 𝔇 可以是其支撑集的任意上界。 𝔇 可作为未知输入的类型声明 𝔇。

[115]换句话说，需要输入的程序只得到了部分指定。查询输入项时，会将其作为延迟约束返回（而不是错误地将其赋值为 0）。[116]此外，翻译还会将任何常量值映射为 false 或 true。[117]例如，如果 𝔭(𝔇)⟦β(np,3,5)⟧ ≠ 0̸，那么 𝔭(𝔇)⟦β(np,3,5)⟧ = true；如果 𝔭(𝔇)⟦β(np,3,5)⟧ = 0，也可能如此。例如，在 ⊕ 下，对非0值进行加法可能得到 0。由于布尔程序不跟踪实际值，因此可能出现这种高估。

由 p（𝔇）定义的类型随后仍包含nz（p（𝔇））中的所有项，适用于任何满足此上界的数据集 𝔇 。例如， p（𝔇）仍会将值true 赋给 β（np,3,5），即使 β（np,3,5） 可能具有非0 值。

CKY 的输入类型声明示例如下，这是一个布尔程序 𝔇:

```
145 参数：k；w；n。146
word(W:w,I:n,K:n) :- I <K。147
len(N:n)。148 γ(X:k,Y:k,Z:k)。
149 γ(X:k,Y:k)。150 γ(X:k,
W:w)。
```

第 #146–150 行指定了 5 个集合，它们的并集是一个包含所有输入项的类型。我们在 #146 行使用符号表示 word(W:w,I:n,K:n) :- I <K，作为 word(W,I,K) :- w(W), n(I), n(K), I < K 的简写。该类型表明，集合 {word(W,I,K) | w(W), n(I), n(K), I < K} 中的项在输入数据 𝔇.[118,119] 中可能具有非0值。像 n 这样的谓词可以通过额外的行来定义，例如 n(I) :- 0 ≤ I ≤ 20，该定义给出了长度为 20 的句子的可能句子位置。不过，为了不局限于任何特定的句子长度，我们省略了此类定义，将 n 保留为参数（#145）。因此，上述程序 𝔇 实际上定义了一种参数化类型，正如许多编程语言中的 List[α] 是一种参数化类型，它依赖于具体的选择，例如

[118]请注意，这种输入类型与格解析兼容（例如，Tomita（1986年）和 Hall（2005年））；在这种情况下，格（即加权无环有限状态自动机）的节点将用输入拓扑序中的位置来标记，而 word(W,I,K) 则编码弧 I W → K 的权重。[119]#146 的一种自然替代写法是 word(W:w,I:n,K:n) :- (K is I + 1)，它使用内置的加法约束。此类型表示集合 {word(W,I,I+1) | w(W), n(I), n(I+1)}，并直接编码了我们在例6中给出的 CKY 输入数据描述。尽管这种输入刻画对某些应用而言可能更精确，但由此得出的结论似乎相同（也就是说，我们无法推导出更紧的导出类型）。

α = 字符串。它依赖于另外三个关系，这些关系指定了允许的终结符、非终结符和句子位置。具体输入类型的形式为 $\mathfrak{D}$(k, w, n)。我们的程序分析器以符号方式推理这些用户参数；因此，它执行符号求值 (例如，Baldoni等人（2018）)。

我们的抽象前向链接算法（§6.1.2）将导出以下规则，为 p($\mathfrak{D}$(k, w, n)) 的派生项指定有效类型：

```
151 β(X:k, I:n, K:n):-I <K。
152 目标。
```

加入这些规则和 $\mathfrak{D}$ 后，就得到了所需的类型描述，它覆盖了所有非零项。将生成的程序称为 m。程序 m 使用与 $\mathfrak{D}$ 相同的参数，并为 p($\mathfrak{D}$(k, w, n)) 提供一个上界。该上界意味着我们只会构建 β 项，其中 X 是非终结符（类型为 k），而 I 和 K 变量是句子位置（类型为 n），且 I < K。此外，该上界还说明（令人欣慰的是）可能可以构建一个目标项。如果目标不可能实现，我们就会发现代码中有错误。同样，如果上界中包含 β(X:k,I:k,K:n)，那这个上界就会宽松得令人怀疑，因为 I 不应当是非终结符。

请注意，p′ = p $\oplus$ $\mathfrak{D}$ 也可以是 p($\mathfrak{D}$(k,w, n))的有效类型。然而，p′下游应用不易使用。[120] 我们的分析器（§6.1.2）保证输出一种导出类型，它是一个程序（如上面的 m），其中每条规则都是简单的。如果一条规则的**体**只包含常量、用户参数和内置约束，我们就称其为简单规则。简单规则可以视为公理（§的符号推广）。2。

[120]例如，时空复杂度分析（§6.3）和程序特化（§6.2).

总之，类型分析器接收程序 $\mathfrak{p}$ 以及参数化输入类型 $\mathfrak{D}$，并计算出一种参数化派生类型，例如 #151–152，用于表示由 $\mathfrak{p}$派生的项。实际上，它会在输入上界 $\mathfrak{p}$ 上运行布尔化程序 $\mathfrak{D}$。不过，由于派生类型只需是一个上界，我们允许分析器在必要时逐步“向上取整”，以保持类型简单。[121]

6.1.2 抽象前向链

现在，我们将介绍基于布尔半环中的不动点迭代（§4.3）的类型推断算法。[122] 在这里，我们将不动点迭代的赋值估计 ν 推广为抽象赋值，其中用户的类型参数（例如n）用于符号化地定义 ν，正如我们在 CKY 示例的导出类型中所见。这种推广是必要的，因为我们不再拥有一个具体的输入数据上界，可用来初始化前向链：n、k和w 参数都是未知关系。相反，我们指定参数化规则，为未知输入数据提供上界。前向链会推导出新的简单规则，这些规则也可能引用n 和$<$等关系。事实上，这些关系可能会出现在导出类型 $\mathfrak{m}$中，正如我们在 #151中所见。

我们在 §6.1.2.1 中给出抽象前向链接算法。我们在 §§6.1.2.2–6.1.2.5 中探讨约束累积带来的挑战。

[121]抽象解释文献通常将向上取整称为拓宽。[122]请回想，不动点迭代也是常规前向链的基础（Ullman，1988年；Eisner、Goldlust 等人，2005年），也是我们在 §5中开发的许多算法的基础。

6.1.2.1 类型推断算法

为了推广布尔半环中的前向链，我们将把 § ν 中的赋值表示为4 中的程序 $\mathfrak{m}$——就像我们在 §5 中所做的那样。我们将在布尔半环中进行推理。这种表示很有帮助，因为我们可以用程序表示未知集合和无限集合。我们要求 $\mathfrak{m}$ 中的每条规则都具有特定的句法形式：每条规则都必须是简单类型。简单**类型**是如下形式的规则： x: - $c_1, \ldots, c_M$ 其中每个 c_m 都在外部定义为内置约束或参数化约束，并且可以包含变量。请注意， x不能出现在任何其他简单类型的体中。我们看见了 CKY 示例的导出类型 #146–152 是简单类型的集合。

从某种意义上说，程序规则 #141–150 已经描述了导出类型，但它们是间接描述的，而不是作为简单类型的集合。这种形式确保每个简单类型都是自包含的，并且可以作为（非递归的）面向对象类来实现。

类型推断通过对符号步操作进行不动点迭代来完成。应将其视为 §5中前向链接算法的一个“提升”版本：它从其他简单类型推导出简单类型，而不是从其他基项推导出基项。这些简单类型中出现的约束会在执行过程中传递。在约束逻辑编程文献中，这类约束称为“延迟约束”，以区别于 或 这类可以立即求值并替换为真或假（分别）的约束。

算法 14 abstract_fc：类型程序的抽象前向链接算法
$\overline{\mathfrak{p}}$. 该过程假定输入类型已被包含在 $\mathfrak{p}$中。

```
def摘要_fc(p):
    ▷ 本算法是对不动点迭代（算法 6）的直接修改
      以寻找松弛后的步操作 relax(传播(step_p(·)))的不动点。
    m ← {}[124]
    当 真:
        m′ ←松弛(传播(步_p(m)))        ▷ 执行松弛步操作
        如果 m′ = m: 返回 m
        m ← m′
```

定义于算法6 [123]

然而，反复应用步操作所带来的挑战在于，约束可能会不断累积。[123] 对于递归规则，约束可能无限累积，因为它们实际上会展开成无界的递归调用链，§6.1.2.2将对此进行说明。为防止发散，我们提出以下策略：约束松弛（relax；§6.1.2.2）和约束传播（propagate；§6.1.2.3）。我们在 §6.1.2.5讨论终止问题。算法14给出了整体方法的伪代码。

6.1.2.2 松弛

考虑以下输入类型 $\mathfrak{D}$ 例如 4。

```
153 参数: n。 154 stop(S):- n(S)。
155 edge(S,S'):-n(S),n(S')。
```

应用步骤操作一次后（该操作会重命名变量以避免冲突），我们

[123]请注意，当在 $\mathfrak{p}$ 中查询用户参数时，算法 2 会将它们作为延迟约束返回。例如，设 $\mathfrak{p}$ 为布尔化 CKY 程序及其输入类型。那么，如果我们执行查找 $\mathfrak{p}$[n(K)]，得到的结果流将包含 n(K) 作为唯一元素。[124]我们将 $\mathfrak{m}$表示为一组规则。在我们的实现中，这个集合会用最一般的版本替换规则；也就是说，如果存在一对简单类型 r, r' 使得 $r' \subseteq r$，我们就移除r' 并保留 r。这确保了等价的规则（在变量重命名下）不会同时出现在集合中，除此之外还有其他作用。它也有助于实现收敛。更多讨论请参见 §6.1.2.4。

推导出以下适用于 β 项的简单类型（未显示其他类型）：156 β(S1):-
n(S1).。

应用步骤 第二次后，我们推导出

```
157 β(S1) :- n(S1).
158 β(S1) :- n(S1), n(S2).
```

注意第二条规则，其中 n(S2)作为递归调用留下的约束。在遵循布尔程序的（类似 Prolog 的）语义时，变量 S2 是存在量化的，因为它是一个只出现在（:-）规则体中的“局部”变量。应用 步骤 k 次后，我们推导出

159 β(S1) :- n(S1).160 β(S1) :- n(S1),n(S2). . 161 ..
162 β(S1) :- n(S1),n(S2),n(S3), ..., n(Sk).

该过程通过逐步扩展类型来描述任意长度的路径，因此会发散，而不是收敛到一个不动点（即一种描述所有能够沿任意长度路径到达终止状态的节点的类型）。我们消除先前递归层级约束累积的策略是：丢弃（即 relax）任何涉及局部变量的约束：

定义 30 (relax)。 令 x:-$c_1, \ldots, c_M$ 为一种简单类型。操作 relax 会删除每个引用了某个变量的延迟约束 c_m，该变量没有出现在 x中。换句话说，我们得到一个头部相同的新简单类型，但其约束子集如下 $\{c_m \mid \mathsf{vars}(c_m) \subseteq \mathsf{vars}(x)\ m \in [1{:}M]\}$。要对一个简单类型集合进行 relax 操作，只需将relax 映射到该集合中的每个元素。

将 relax 应用于 step 的第二次应用#157–158，我们看到它会丢弃第二个简单类型的第二个约束，因为它依赖于

非头部变量S_2，yielding:

```
163  β̄(S1) :- n(S1).
164  β̄(S1) :- n(S1).
```

我们可以合并这些规则，因为它们是重复项。现在得到以下类型

```
165  β̄(S1) :- n(S1)。
```

我们现在得到了一个不动点：本次迭代的类型（松弛之后）等于上一次迭代的类型（#156）。请注意，由于 β(S)即使对于满足条件的 节点 S，也不存在以停止结尾的路径，因此精度有所损失。

为什么 relax 是有效的？一般来说，从简单类型中删除约束只会使类型变大，从而提高上界。删除依赖于非头部变量的约束，只是一种方便的选择。

现在我们来看一个示例，其中 relax 会导致类型过于宽松。回顾 CKY（示例 18）。来自 𝔇 (#145–150) 的简单类型在第一 m 步中被添加到.....上，例如

```
166  γ̄(X,W) :- k(X), w(W). 167 单词(W,I,K) :-
w(W), n(I), n(K), I < K.
```

在下一步，#143 来自 p

```
168  β̄(X,I,K) :- γ̄(X,W), word(W,I,K).
```

为 β给出了一个初始类型：

```
169  β̄(X,I,K) :- k(X), w(W), n(I), n(K), I < K.
```

在relax之后，我们得到

```
170  β̄(X,I,K) :- k(X), n(I), n(K), I < K.
```

不过，在接下来的步骤，#141 来自 p

```
171  β̄(X,I,K) :- γ̄(X,Y,Z), β̄(Y,I,J), β̄(Z,J,K).
```

将展开 m 为以下类型：172 β(X,I,K) :- k(X), k(Y), k(Z), n(I), n(J), n(K), I < J, J <K. relax将删除所有针对局部变量的约束：173 β(X,I,K) :- k(X), n(I), n(K).

遗憾的是，松弛导致顺序信息（<）丢失。在下一节中，我们将看到如何组合 I < J 和 J < K（在对它们进行松弛之前），从而推导出 I < K。由于 I < K 只约束头部变量，因此它会在松弛步骤中保留下来，使分析器能够推导出 β(X,I,K) 项都具有 I < K 这一性质（而不是第一代具有 I < K，第二代具有 (∃J) I < J、J < K，第三代具有 (∃J1, J2, J3) I < J_1，J_1 < J_2，J_2 < J_3，J_3 < K，等等）。

6.1.2.3 约束传播

约束传播规则 (Frühwirth，1998），常用于约束逻辑编程（例如 ECLiPSe（Apt 和 Wallace，2007年））、计算机代数系统（例如 Mathematica（Wolfram，2003年年））和定理证明器（例如 Z3（Moura 和 Bjørner，2008年年））。在我们的场景中，它们为用户提供了一种机制，使其能够提供关于类型参数和内置约束的领域专门知识。它们用于在推理过程中推导新的约束，这有助于抵消过度松弛（§6.1.2.2）。

回想一下在 CKY 示例中推断出的宽松类型（#172；在.....下面重复列出）：174 β(X,I,K) :- k(X), k(Y), k(Z), n(I), n(J), n(K), I < J, J <K。

我们看见了 t hat relax 会导致序关系 I < K 在 #173。约束

传播规则通过推导额外的约束来弥补这一损失，例如根据 <:[125]
175 β(X,I,K) :- k(X), k(Y), k(Z), n(I), n(J), n(K), I <J, J <K, I <K的传递性得出 I < K。

–

应用relax后，我们得到所需的类型

176 β̄(X,I,K) :- k(X), n(I), n(K), I < K.

下面，我们给出一些示例，说明约束传播规则的动机，以及它们对我们的贯穿示例有何用处。约束传播规则使用 <== 下方示例中的语法[126] 来指定。它们也可以与输入类型一起指定。

- 在 CKY（示例 6）中，文法起始符号（常量）s 是参数化非终结符类型k的一个元素，而0 是参数化句子位置类型n的一个元素：

177 k(s) <== 为真。
178 n(0) <== 为真。

- 要断言k 和n 类型互不相交：

179 失败 <== k(X), n(X)。

这里失败是一种永远无法满足的特殊约束（即矛盾）。如果k(X) 和n(X) 同时出现在简单类型的体中，则该简单类型为空，可以将其删除。请注意，#177–179 足以从n(s) 或 k(0) 推出矛盾（即二者都为假）。

- 断言类型d 是类型c的子集：180 c(X) <== d(X)。

[125]指定传递性属性的具体语法在 #181。[126]我们使用 <== 来区分传播规则与程序规则和输入类型。

- 断言 < 是严格偏序:

```
(I  <K)  <== (I  < J), (J  < K)。
失败 <==(I  <I)。
```

- 要推理递归类型，例如非终结符[127] 列表（即类型k）：

```
ks([])  <==真。
k(X) <==ks([X|Xs])。
ks(Xs)  <==ks([X|Xs])。
```

内置项的传播规则（例如 < 在.....上面的规则）由标准库提供，而用户必须为用户参数提供规则。如果用户未提供某些规则，我们的系统仍会生成一个有效的上界，但该上界可能比必要的更宽松。因此，编写传播规则可能是一个交互式过程。然而，如果用户提供了不可靠的传播规则，系统很可能会生成错误的上界。

用于传播抽象前向链接（抽象_fc）的方法运行一个标准的约束传播算法（Frühwirth，1998）——这只是另一种前向链接算法——以推导简单类型体中的所有传递后果。

检查约束。在传播步骤中，我们会检查每个内置约束是否得到满足或不可满足。任何已满足的约束都可以安全地从其简单类型中删除，而且不会放宽该类型。如果给定的简单类型中存在不可满足的约束，那么该简单类型可以安全地从 m中删除。

—

[127]方括号竖线符号表示是处理列表时使用的简写（借自 Prolog（Colmerauer 和 Roussel，1996））。例如，列表 [1,2,3] 可展开为 cons(1,cons(2,cons(3,nil)))。符号表示 [X, Y|Rest] 会与以下形式匹配：X=1, Y=2, Rest=[3]。

6.1.2.4 移除冗余简单类型

在算法 14中，如果存在一个超类型，且它是（另见脚注 $\bar{s}$），可以从 m 中移除简单类型 124）。下面介绍一种利用 传播来检验此类子类型关系的方法。我们还会介绍一种合并简单类型的方法，本章后面会用到。

算法 15 简单类型的子类型、交集和合并算法。

```
▷如果 s ⊆ t，则返回 true；可能会出现假阴性。
定义 子类型(s, t):
    ▷(s ⊆ t) ⟺ (s ∩ t) = s
    r ← 交集(s, t)
    R ←传播(r)
    S ←传播(s)
    返回 R = S                                   ▷ 在变量重命名后相等
▷返回一个表示 s 和 t交集的简单类型。
定义 交集(s, t):
    θ ← 合一(头部(s), 头部(t))
    s′ ← θ(s)
    t′ ← θ(t)
    返回 (头部(s′) :- 体(s′), 体(t′))
▷给定简单类型 r 和 s，返回一个简单类型 t，使得 r ⊆ t 和 s ⊆ t。请注意，t 可能
  是 r 和 s的松界。
定义 合并(r, s):
    (r :- R) ← r
    (s :- S) ← s
    x ← fresh()                                  ▷ 新变量；用于表示公共头部形状
    R′ ←传播(R ∘ (x=r))
    S′ ←传播(S ∘ (x=s))
    T ← R′ ∩ S′                                  ▷ 仅保留共同约束
    返回(x :- T)
```

请注意，如果传播规则不完备，此测试可能无法检测出子集关系。未来的系统或许会使用功能更强大的定理证明器（例如 Z3）（Moura 和 Bjørner，2008年）来测试子集关系。

6.1.2.5 终止

本节介绍一种防止抽象前向链接无限运行的技术。该技术可确保在有限时间内终止并得到有效上界，但可能会牺牲精度。为了继续进行下游分析（例如空间和时间复杂度分析器（§6.3）、程序特化系统（§6.2）以及程序优化器（§8）），我们需要有效上界。[128]

确保终止的挑战在于，抽象前向链必须提供一个有效的上界，因此我们不能简单地运行有限次迭代后返回图表，因为这样可能会低估类型。我们的方法是将类型向上取整（正如我们在 relax 中所做的那样）。作为一种极端情况，请注意，我们可以将 $\mathfrak{m}$ 一路向上取整到无约束类型 `X:-true`，因为它始终是一个有效的上界。遗憾的是，这种类型完全没有信息量，因为它对于所有子类型都过于宽松；最好是至少为部分子类型提供更好的界限，而对其他子类型使用更宽松的类型。

—

我们的终止策略涉及根据三个用户可控制的参数 $\langle D_{\text{depth}}$、D_{builtin}、$D_{\text{propagation}}\rangle$ 进行**截断**，其工作方式如下：

• **项**：我们可以构造出无限多个与越来越大的项对应的简单类型。为避免这种情况，我们将其截断至用户指定的（可选）**深度限制** $D_{\text{深度}}$。此深度限制将按以下方式重写简单类型：在深度 D 处出现在头部中的任何子项都会被替换为一个 fresh，

[128]它们还可用于确保算法9（在具体数据上执行支持度估计）返回一个有效的上界。

无约束变量。[129]

• **内置项**：内置约束的挑战在于，它们可能引入数量不受限制的新符号，而这些符号可用于简单类型。为防止这种情况，我们会跟踪内置关系在所有迭代中引入的不同符号数量；一旦引入的内置项超过$D_{\text{内置项}}$，当任何内置约束试图引入新符号时，我们都会阻止其求值。

• **传播**：我们还必须确保约束传播规则不会无限地产生结果（即，它们不会产生无界数量的新约束）。为此，我们限制一次调用传播时，约束传播规则最多可产生的不同约束数量。若推导出的 $D_{\text{传播}}$ 条约束（针对某个给定的简单类型）超过这一数量，迭代约束传播过程就会停止，并保留截至此时得出的结论。

这些终止启发式（以及 relax）足以确保算法 14 终止并给出有效的上界。[130] 然而，启发式一经启用，得到的分析结果就会比较宽松。因此，系统会向用户提供一条警告消息。如果需要更好的结果，程序员可以与系统交互（例如使用交互式

[129]我们根据结构相等约束应用深度限制，因此，`f([1,2,3,4,5])` 和`f([1,2|Xs]) :- Xs=[3,4,5]` 会以相同方式处理，因为后者会被转换为前者。[130]参数的取值有限时，只能生成有限种简单类型。由于松弛步操作是单调非递减的，抽象前向链接不可能 m永远持续增长。

定理证明器），通过细化输入类型或补充缺失的传播规则来改进分析。

一般来说，截断参数的限制应设得保守一些（即设为较大的值）；然而，选择较小的值可以提高效率，因为找到的类型可能会呈指数级增长 D_{depth}（见示例 19）。我们不能在所有情况下都简单地将 D_{depth} 的值设得很小。例如，如果 $D_{\text{depth}}=4$，那么简单类型 $f_1(f_2(f_3(f_4(f_5(X)))))$:- n(X) 将被近似为 $f_1(f_2(f_3(f_4(\text{Trunc}))))$:- true，这是一个宽松得多的类型。如果 D_{depth} 原本是≥ 5，那么分析就不会损失任何精度。

示例 19. 抽象前向链接会为下面的程序生成一个呈指数级增长的类型（相对于 D_{depth} 参数）。

```
位串([]). 位串([X|Xs]) += 位(X) 位串(Xs)
位(0). 位(1).
```

因为截断之前，它会枚举数量呈指数级增长的列表（相对于 D_{depth}）开始生效。[131] 在下面的示例有 $D_{depth}=5$:

```
位(0).位(1). 位串([]). 位串([0,0,0]). 位串([0,0,0,Trunc1 |Trunc2]). 位串([]). 位串([0,0,1]).
位串([0,0,1,Trunc1 |Trunc2]). 位串([1]). 位串([0,1,0]). 位串([0,1,0,Trunc1 |
Trunc2]). 位串([0]). 位串([0,1,1]). 位串([0,1,1,Trunc1 |Trunc2]). 位串([0,0]). 位
串([1,0,0]). 位串([1,0,0,Trunc1 |Trunc2]). 位串([0,1]). 位串([1,0,1]). 位串(
[1,0,1,Trunc1 |Trunc2]). 位串([1,0]). 位串([1,1,0]). 位串([1,1,0,Trunc1 |
Trunc2]). 位串([1,1]). 位串([1,1,1]). 位串([1,1,1,Trunc1 |Trunc2]).
```

在位串示例中，没有任何有限值 D_{depth} 能够在不向上取整的情况下覆盖该类型。虽然增大 D_{depth} 确实能让分析更精确，但不太可能改善任何下游分析。例如，[131]请注意，此示例不需要任何输入数据。

我们将在 *§6.3.1* 中看到的空间复杂度分析仍会（正确地）预测，不动点迭代所需的空间量为 ∞，无论D_{depth} 对 *50* 亿来说都是 *5*。[132]

我们可以得到更精确的分析，将列表参数约束为良构。我们使用以下内容：

```
197 % 声明用户参数 198 参数: b; bs.
199 % 位串的类型建议 200 位串(Xs) :-
bs(Xs). 201 % 传播规则 202 b(0)  <==
true. 203 b(1)  <==true. 204 bs([])
 <==  true. 205 b(X)  <==bs([X|Xs]).
206 bs(Xs)  <==  bs([X|Xs]).
```

请注意 #*200* 为以下内容初始化类型：位串(Xs)。由此，抽象前向

链接证明它是一个有效的上界。输出为

```
207  位(0).208  位(1).209  位串
(Xs):-bs(Xs).
```

我们的符号求值方法（抽象前向链接）需要这类终止策略并不奇怪，因为更广义而言，符号求值技术饱受路径爆炸之苦，并采用截断技术（如我们的方法）使其方法具备实用性（Baldoni等人，2018，§5）。状态合并（Baldoni等人，2018，§5）是另一种实用技术，即合并两个或更多分析器状态，也就是用一个表示它们并集的状态来替代这些状态，该并集可能是有损并集。未来工作可以考虑探索状态合并方法，以减轻这里采用的激进截断

[132]根据我们作为类型分析器用户的经验，我们发现使用较小的截断参数值重新运行类型分析器具有优势，因为这会得到较小的输出，从而说明相同的发散问题。

方法。[133]

总之，我们描述了一种截断技术，可确保抽象前向链接在有限时间内产生有效的上界。该上界是可靠的，但可能相当宽松。我们希望，宽松的上界总比“类型分析未收敛”错误更有用。用户可以检查系统的输出，并提供额外的约束传播规则以避免截断。因此，类型分析系统可以用作定理证明助手。[134] 最后，截断使下游分析得以继续进行，尽管使用的是宽松的类型。

[133]我们在算法 15 中给出了一种简单类型合并方法，可能对此目的有用。[134]探索将动态分析用作辅助静态分析的手段会很有趣。例如，未来的系统可以综合出一些看似合理的不变量，而这些不变量恰好在程序使用具体数据运行时对程序项成立。在此基础上，系统可以向用户推荐一些断言，以收紧对程序的静态程序分析。

6.2 基于类型的程序特化

本节介绍一个系统，该系统可将给定程序特化为特定的输入–输出 API。该系统利用 §6.1 中的类型分析器执行程序变换（即 Dyna 到 Dyna 代码生成），从而生成专用程序。下面来看一个示例。

```
210 目标 += a(I,I).211 a(I,K)  +=
b(I,J) * c(J,K).212 目标 +=
dead(X).
```

和输入类型[134]

```
213 b̄(_,_).
214 c̄(_,_).
```

和输出类型[134]

```
215 ĉ(_,_).
```

我们可以看到，dead(X) 未定义；因此，对于给定的声明输入类型，底部规则 (#212) 可以确定永远不会触发。现在，假设我们从输出（即示例中的目标）反向推理。在这种情况下，我们发现只需要 a(I,K) 的一个子集——具体来说，只需要它的对角线 a(I,I)。我们称 a(I,I) 为有用项，而非对角线元素则是无用的。[135] 有用项也应当非 0。有用性分析是敏感性分析的一种形式：如果某项非 0，且将其值设为 0 会改变任一输出项的值，那么该项就是有用的；否则就是无用的。计算有用项的精确集合需要具体输入数据，因此，与 §6.1 类似，我们将估算一个（符号）

[134]回想一下，下划线（_）会创建一个fresh、未命名变量。[134]这里的标记 ̂类似于 · 来自§6.1的标记，除了有用类型（见在……下面）¯。[135]用有限状态机和上下文无关文法的术语来说，一个有用项是既可达且共可达的。

针对特定输入–输出类型的有用项集合的上界。

下面，我们利用对有用项的了解，手动对示例程序的每条规则进行特化，使其更高效地计算目标：

```
216 目标 +=a(I,I)。217 a(I,I)  +=b(I,J) * c(J,I)。 % 仅构建 a 的对角线
```

请注意，由于非对角线上的 a(I,K) 项现在取值为 0，因此该程序与前一个程序并不完全等价。[136] 但是，声明的输入–输出 API 得以保留：输出值（即 目标）保证与原始程序的输出值相等。

我们还可以通过将 a(I,I) 项缩写为 a'(I) 来进一步改进该程序，因为它们始终带有一个冗余变量：

```
218 目标 += a'(I)。219 a'(I)
 +=b(I,J) * c(J,I)。
```

我们还可以选择性地加入规则，例如a(I,I) += a'(I)，从而从其缩写版本中恢复原始项（假设它们是有用项）。稍后，我们将展示一些缩写示例：展平嵌套项、移除冗余变量，以及删除从未实例化的变量。

本节将开发一个用于执行此类特化变换的自动化系统。

- 首先（§6.2.1），我们将介绍对类型分析方法（§6.1）的扩展，以估计有用项的集合。

- 接下来（§6.2.2），我们给出一种特化方法，对每条规则进行变换——

[136]在论文的某些部分，这种改变语义的变换会要求我们重命名特化版本 a，因为它具有不同的含义。为使论述简单起见，我们将允许这种情况。

算法 16 有用：对于给定的程序 $\mathfrak{p}$、输入类型 $\mathfrak{D}$ 和输出类型 $\overline{\widehat{\mathfrak{D}}}$，定义一个新的项集合$\widehat{\cdot}$其中，若 x 可能有用，则 $\widehat{x}$ 为真。

1. ▷ 注意：函数$\widehat{\cdot}$会用一个新颖的函数符号包装一个项。
2. **定义** 有用$(\mathfrak{p}, \overline{\mathfrak{D}}, \widehat{\mathfrak{D}})$:
3. $\widehat{\mathfrak{p}} \leftarrow \wr\int$ ▷ 新程序
4. $\widehat{\mathfrak{p}} \oplus\!= \overline{\mathfrak{p}} \oplus \overline{\mathfrak{D}}$ ▷ 定义可能的项的集合 $\cdot$; 与 *§6.1* 一
5. $\widehat{\mathfrak{p}} \oplus\!= \widehat{\mathfrak{D}}$ ▷ 基本情况：输出类型是有用的
6. **适用于**$(x \oplus\!= y_1 \circ \cdots \circ y_K) \in \mathfrak{p}$:
7. **适用于** $k \in [1{:}\ K]$: [137]
8. $\widehat{\mathfrak{p}}[\widehat{y_k}]$: $\widehat{x}$, $\overline{y_1}$, ..., $\overline{y_K}$ ▷ 如果子目标 y_k 出现在一个.....里面，它就是有用的
9. **返回** $\widehat{\mathfrak{p}}$ 活跃规则和有用头部

即按每种可能的类型将其展开为多个版本——旨在减少无用项和无用规则触发的总数。

- 最后（§6.2.3），我们细化特化方法以执行缩写，如在.....上面的示例中那样。

6.2.1 为有用项推断类型

为了推理特定 API 的有用项集合（即，一个$\widehat{\ }$输入类型程序 $\mathfrak{D}$ 和一个输出类型程序 $\mathfrak{D}$的有序对），我们变换程序$\mathfrak{p}$，使用算法 16。输出将是一个新程序 $\widehat{\mathfrak{p}}$，其定义了用$\widehat{\cdot}$标记的新项，使得 $\widehat{x}$为 真 如果 x可能有用。请审阅在.....下面的带注释的伪代码以作进一步解释。

对 $\widehat{\mathfrak{p}} =$ 有用$(\mathfrak{p},\ \mathfrak{D},\ \widehat{\mathfrak{D}})$ 使用抽象前向链式推导（§6.1）进行类型分析，将推导出 $\widehat{\mathsf{x}}$ 的类型。下面是 $\widehat{\mathfrak{p}}$

[137]跳过 k 是安全的，当 y_k 是常量或内置项时。

```
220 % 基本情况：输出类型有用 221 目标。 222 % 目标 +=a(I,I).223 ^a(I,I) :- 目标, a(I,I). 224 % 目标 +=dead(X).
225 dead(X) :- 目标, dead(X). 226 % a(I,K) +=b(I,J) * c(J,K).227 ^b(I,J) :- ^a(I,K), b(I,J), c(J,K).228 ^c(J,K) :- ^a(I,K), b(I,J), c(J,K).
```

```
229 % 布尔化程序 230 goal:- a(I,I). 231 goal:- dead(X). 232 a(I,K) :- b(I,J), c(J,K). 233 % 输入类型 234 b(I,J). 235 c(J,I).
```

推断类型 $\hat{\mathtt{m}} = \text{抽象_fc}(\hat{\mathtt{p}})$ 是

```
236目标. 目标.
237 ^a(I,I). a(I,K).~238 b(I,J). b(I,J).239 ^c(J,K). c(J,K).
```

我们在#237 中看到，只有对角线a(I,I) 是有用的，而整个a(I,K) 都是可能的，正如我们在之前的手动分析中所观察到的。

最后，我们注意到 $\hat{\mathtt{m}} \subseteq \mathtt{m}$ 因为我们向推断问题添加了更多约束。我们可以在第 16.8 行看到 $\hat{y_k}$ 仅当 y_k 时才为真。

6.2.2 特化变换

给定一个特定类型 $\hat{\mathtt{m}}$，[138] 我们可以通过针对该类型实例化其规则，来特化程序 $\mathtt{p}$，从而看出哪些规则永远不会（有效地）触发。更好的是，我们可以将每条规则特化为其有效的实例化。算法17给出了相应的伪代码。

正确性。

互不相交 $\hat{\mathtt{m}}$。 我们要求 $\hat{\mathtt{m}}$ 的各个分支两两互不相交。互不相交性保证我们可以将规则 r 替换为互不重叠的实例-

[138]请注意，§6.1 中的类型 $\mathtt{m}$ 也适用（在需要的地方将 $\hat{\cdot}$ 替换为 $\cdot$）；不过，由此得到的专用程序效率较低，因为 $\hat{\mathtt{m}} \subseteq \mathtt{m}$。

算法 17 特化：特化程序 $\mathfrak{p}$ 为类型 $\widehat{\mathfrak{m}}$。

定义 特化($\mathfrak{p}$, $\widehat{\mathfrak{m}}$)：
  断言 $\widehat{\mathfrak{m}}$ 里面的简单类型两两互不相交
  $\mathfrak{p}' \leftarrow \wr\int$ ▷ 新程序
  对于 $(x \oplus\!= y_1 \circ \cdots \circ y_K) \in \mathfrak{p}$：
    对于 $\langle \mathcal{C}_0, \mathcal{C}_1, \ldots\ldots, \mathcal{C}_K \rangle \in \widehat{\mathfrak{m}}[\widehat{x}] \times \widehat{\mathfrak{m}}[\widehat{y_1}] \times \cdots \times \widehat{\mathfrak{m}}[\widehat{y_K}]$： ▷ 根据类型展开
      $\mathcal{C} \leftarrow$传播$(\mathcal{C}_0 \quad \circ \mathcal{C}_1 \circ \cdots \circ \mathcal{C}_K)$
      如果 失败 $\in \mathcal{C}$：**继续** ▷ 可以安全地删除，因为它不会触发
      $\mathfrak{p}'[x] \oplus\!= y_1 \circ \cdots \circ y_K \circ \mathcal{C}$ ▷ 这些约束 $\mathcal{C}$ 变为布尔侧条件
  返回 $\mathfrak{p}'$

化。这样做是为了避免重复计数。给定一个互不相交的有用类型 $\widehat{\mathfrak{m}}$，我们可以对程序的规则及其不同情形进行特化。特化变换的工作方式是利用规则的抽象实例化，收集该规则可能具有的类型所对应的约束。根据这些类型，我们可以向规则添加（具体的）约束，将其划分为可能的各种情形。如果存在重叠类型，我们可以取一个推断类型 $\widehat{\mathfrak{m}}$，并通过应用 析取化($\widehat{\mathfrak{m}}$)（算法 9）将其变为互不相交，作为预处理步骤。

⊂**需要用户参数**。要利用用户的参数化类型，用户必须在输入中提供这些类型。请看 CKY 中以下两种可能的类型：`γ(X,Y)`项（示例 18）：

```
γ(X,Y) :- k(X), k(Y)
γ(X,Y) :- k(X), w(Y)
```

当用户提供集合 `γ(X,Y)` 时，我们不知道其中哪些项属于`k(Y)`，哪些属于`w(Y)`。因此，我们要求用户提供这些集合，以便系统区分它们的类型。[139]

我们要求 要求用户除了输入之外，还要提供他们的参数 ut

[139]在某些情况下，可能可以根据输入数据自动确定用户的参数集。如果用户忘记为每个参数至少提供一个实例，系统会打印一条警告。

数据。例如，CKY 示例要求用户提供实现n、k和w 约束的方法。为此，需要将它们定义为与输入数据一起提供的项。由于这些项是约束，因此它们的值为{0, 1}；只有值为1的项才需要在输入数据中枚举𝔇。[140] 请注意， < CKY 的输入类型中出现的约束是系统提供的内置关系。

6.2.3 缩写变换

我们将特化变换（§6.2.2）扩展为 缩写 项的名称，就像前面的矩阵对角线示例中所做的那样。缩写是一种按照每个项可能具有的简单类型对其进行划分的方案。它会定义新项，并以其简单类型及任意（必要的）参数作为名称。算法18中给出了伪代码。

正确性。 缩写与特化（§6.2.2）在相同条件下都是正确的。

缩写与特化。 与特化变换不同，缩写变换可以从缩写后的程序中省略某些约束，因为根据规则体中的类型，这些约束在静态上已保证成立（见第18.27行）。特化变换

[140]可以用规则来定义它们，例如，如果句子位置（n）编码为整数{1, … , 10}，我们可以写成 n(X) :- 1 ≤ X ≤ 10.[141] 元组 ⟨X1, … , XM⟩ 的顺序可以任意，但对每个 t 都必须保持一致。完全自由的变量可以从元组中省略。[142]添加第 18.30 行的规则时，如果缩写中对任何自由变量求和，就必须应用重数校正（§3.1.2.3）。我们计算原规则中的局部自由变量集合与缩写后规则中的局部自由变量集合。如果这两个变量集合不相等，就将新规则的规则体乘以 (∞ · 1)。

。

算法 18 缩写：按类型缩写程序 $\mathfrak{p}$，根据类型 $\widehat{\mathfrak{m}}$。

定义 缩写$(\mathfrak{p},\ \widehat{\mathfrak{m}})$:
  断言 $\widehat{\mathfrak{m}}$ 中的简单类型两两互不相交
  $\mathfrak{p}' \leftarrow \wr\int$ ▷ 将作为输出程序
  ▷ 将项 x 映射到由 t 标识的类型对应的缩写名称
  定义 缩写$_t(x)$:
    如果 x 是内置项和常量：**返回** x
    $(x' :\text{-}\ \mathcal{C}') \leftarrow \mathsf{fresh}(\widehat{\mathfrak{m}}_t)$
    $\langle \mathsf{x}_1, \ldots, \mathsf{x}_M \rangle \leftarrow$ 按顺序排列的[141]集合中的变量 $(x' :\text{-}\ \mathcal{C}')$
    $\theta \leftarrow$合一(x, x')
    返回 $\theta($生成$_t(\mathsf{x}_1, \ldots, \mathsf{x}_M))$ ▷ 这里生成$_t$ 是简单类型 t的新符号
  ▷ *API* 保留规则
  对于 $(x :\text{-}\ \mathcal{C}) \in \widehat{\mathfrak{m}}$:
    为简单类型 $(x :\text{-}\ \mathcal{C})$ 分配唯一标识符 t
    如果 t 是一个输入:
      ▷ 输入项不应缩写，以保留 $\mathfrak{p}$的 *API*。下面的规则将该类型的输入项作为输入，得到其缩写项 t。
      $\mathfrak{p}'[$缩写$_t(x)] \oplus\!= x \circ \mathcal{C}$ ↰ 此处，约束 $\mathcal{C}$ 是必需的，以确保正确的集合输入项被传递到此缩写项。
    else
      ▷ 以下规则可从缩写项还原原始项；注意：下面添加的规则在 $\mathfrak{p}'$ 除非 x 是一个输出项。
      $\mathfrak{p}'[x] \oplus\!=$ 缩写$_t(x)$
  针对 $(x \oplus\!= y_1 \circ \cdots \circ y_K) \in \mathfrak{p}$:
    对于 $\langle \mathcal{C}_0, \mathcal{C}_1, \ldots\ldots, \mathcal{C}_K \rangle \in \widehat{\mathfrak{m}}[\widehat{x}] \times \widehat{\mathfrak{m}}[\widehat{y_1}] \times \cdots \times \widehat{\mathfrak{m}}[\widehat{y_K}]$:
      $\mathcal{C} \leftarrow$ 传播$(\mathcal{C}_1 \circ \cdots \circ \mathcal{C}_K)$
      如果 失败 $\in \mathcal{C}$: **继续**
      $\mathcal{C}' \leftarrow$传播$(\mathcal{C}_0 \quad \circ \mathcal{C})$
      如果 失败 $\in \mathcal{C}'$: **继续**
      ▷ 必须检查头部的类型约束，但 $\mathcal{C}$ 中的约束可以已被移除，因为它们保证 会被满足。
      $\mathcal{C}'' \leftarrow \mathcal{C}_0 \setminus \mathcal{C}$
      $\langle t_0, t_1, \ldots, t_K \rangle \leftarrow$ 当前简单类型的标识符
      ▷ 将 x 和每个 y_k 替换为其类型专用缩写。[142]
      $\mathfrak{p}'[$缩写$_{t_0}(x)] \oplus\!=$ 缩写$_{t_1}(y_1) \quad \cdots$缩写$_{t_K}(y_K) \circ \mathcal{C}''$
  返回 $\mathfrak{p}'$

无法做到这一点，因为这些子目标没有被拆分成各自不同的类型。因此，一个具有不止一种简单类型的子目标可能只在部分情况下满足这些约束。对于缩写而言，这并不是问题，因为这些项被拆分成了互不相交的简单类型。再次考虑CKY。子目标rewrite(X,Y)有两种可能的简单类型： $_{\gamma}$(X,Y) :- k(X), k(Y)和$_{\gamma}$(X,Y) :- k(X), w(Y)。因此，特化无法对Y上的约束给出多少保证，但缩写可以，因为rewrite(X,Y)被拆分成了它的两种类型。[143]

_

_

我们在自然语言处理中提供关于缩写 的示例，见例 21。

例20. 考虑如下程序，用于计算加权图中节点有序对之间所有路径的总权重：

[242] 路径(I,I)。[243] 路径(I,K) += 边(I,J) * 路径(J,K)。

假设输入和输出类型为

$$\overline{\mathfrak{D}} = \{\texttt{params: n. edge(I,J) :- n(I), n(J).}\} \textit{ and } \widehat{\mathfrak{D}} = \{\texttt{path(I,K) :- n(I), n(J).}\}$$

缩写生成如下程序：

244 % 将每种输入类型映射到其缩写 245 edge$_0$(I,J) += edge(I,J) * n(I) * n(J). 246 % 有用输出项的恢复规则 247 path(I,K) += path$_1$(I,K). 248 % 缩写规则 249 path$_1$(I,I) +=n(I). 250 path$_1$(I,K) += edge$_0$(I,J) * path$_1$(J,K).

请注意，非基的 path(I,I) 项已经消失，因为变量 I 现在受到约束n(I)。

*N*现在，假设我们为有用的起始状态引入一种类型（

[143]话虽如此，我们至少知道 k(X) 在两种情况下都成立，因此我们可以去掉该约束。我们的特化变换不会执行这项优化，但后续分析可以做到。

$\overline{}$ $\sim\mathfrak{D}$ = {参数: a; ~~n。~~ 边(I,J) :- n(I), n(J)。} 以及 $\mathfrak{D}$ = {路径(I,K) :- a(I), n(J)。}

在此 *API* 下，缩写 产生：

```
251 % 将每个输入类型映射到其缩写 252 边0(I,J)
 +=边(I,J) * n(I) * n(J). 253 % 有用项的恢复规则
254 路径(I,K) +=路径1(I,K). 255 % 缩写规则 256
路径1(I,I) +=a(I) * n(I). 257 路径1(I,K) +=边0(I,J) *
路径1(J,K).
```

此版本进一步约束 `path(I,K)` 项，因为它们还必须满足`a(I)`。

6.3 自动空间与时间分析

本节基于 §6.1中的方法论所提供的类型分析，构建一种自动化方法，用于分析 Dyna 程序在给定数据 $\mathfrak{p}$ 时 $\mathfrak{D}$的空间和时间复杂度。给定与 $\mathfrak{m}$ 相关联的 $\mathfrak{p}$，关键是求出其基数的参数化上界。随后，我们将利用这些上界预测 $\mathcal{O}$界，即采用 §5中的基于议程的不动点迭代算法执行 $\mathfrak{p}(\mathfrak{D})$前向链执行的运行时间参数化上界。

6.3.1 空间分析

为了界定我们用于求值 $\mathfrak{m} = \mathfrak{p}(\mathfrak{D})$ 的不动点迭代算法的空间复杂度，只需界定 $\mathsf{nz}(\mathfrak{m})$ 的大小，因为这些算法只需存储 $\mathfrak{m}$。[144] 我们将以 §6.1 给出的上界 $\mathfrak{m} \supseteq \mathsf{nz}(\mathfrak{m})$ 为基础。我们用 $|s|$表示集合 s 的实际基数，用 $\lceil s \rceil \geq |s|$ 表示上界；我们将该符号表示扩展到程序，方法是通过$\mathsf{nz}(\cdot)$ 将程序隐式强制转换为集合。我们的高层方法是 $\lceil \mathfrak{m} \rceil \stackrel{\text{def}}{=} \sum_{s \in \mathfrak{m}} \lceil s \rceil \geq \sum_{s \in \mathfrak{m}} |s| \geq |\mathfrak{m}| \geq |\mathfrak{m}|$，[145]其中，简单类型 s的大小 $|s|$ 计数的是满足其变量的满足赋值数量

[144]我们假设，任何额外的数据结构（例如用于加快查找操作的数据结构）只会使空间复杂度增加一个常数因子。此外，我们假设 $\mathbb{V}$中的每个值都占用常量大小的空间。[145]如果 $\mathfrak{m}$ 中的简单类型有大量重叠，那么最后一个不等式就会比较宽松，因为求和会对重叠区域重复计数。删除子类型（§6.1.2.4）可以解决最严重的情况，但未来工作也可以考虑将 $\mathfrak{m}$ 转换为简单类型的划分，方法如下：将任意一对重叠的简单类型 s 和 t 替换为 $s \cap t$、$s \setminus t$和 $t \setminus s$。重复此过程，直到不存在重叠的类型对 s 和 t。集合差 $\setminus$ 可以通过基本约束的否定形式来实现，包括相等约束（即合一）；相等约束通常以隐式形式表示。

，而 $\lceil s \rceil$ 是其上界。

众所周知，对满足一个约束集合的变量赋值进行计数在计算上可能代价高昂。然而，在我们的情形中，更重要的问题是我们甚至不知道参数化约束是如何定义的。本节的其余部分考虑如何根据关于参数化约束的某些用户提供的信息来构造一个界 $\lceil s \rceil$ 。

输入规模说明。 考虑简单类型 `γ(X,W) :- k(X), w(W)`。一个直接的上界可以求得：

$$\left|\bar{\gamma}(X,W) \text{ :- } k(X),\ w(W)\right| \leq \lceil k(X) \rceil \cdot \lceil w(W) \rceil = k \cdot w$$

其中 k 和 w 是用户指定的规模参数。（我们稍后将解释规模参数是如何指定的。）然而，当同一变量上存在多个约束时，情况会变得更加复杂。考虑简单类型[258] `β(X,I,K):-k(X),n(I),n(K),I <K`。

–

显然，至多 $k \cdot n \cdot n$ 个赋值能够满足前3个约束。此外，施加第四个约束 `I <K` 只会减少该数量——尽管第四个约束单独来看 会有无穷多个满足赋值（$|\texttt{I <K}| = \infty$）。

我们将说明如何从以下内容推导出一个上界：序列约束 c_1，...， c_M 在简单类型的体中 s （包括传播添加的任何约束，因为它们可能会改进上界）。重新排列约束不会影响满足赋值的数量；但是， $M!$ 种序中的每一种都可能得出不同的上界，因此我们选择其中最紧的一个。

思路是使用**条件基数**。令 $\mathcal{V}$ 为任意变量集合。虽然 c 可能有许多变量赋值满足，但其中只有一些与给定的赋值 $\mathcal{V}$兼容。假设用户提供一个数$\lceil c \mid \mathcal{V} \rceil$，使得对 $\mathcal{V}$ 的任何赋值，扩展后得到的 $\lceil c \mid \mathcal{V} \rceil$变量$\mathsf{vars}(c)$赋值都不超过该数。情况 $\lceil c \mid \mathcal{V} \rceil = 1$ 尤为值得关注：如果$\mathsf{vars}(c) \subseteq \mathcal{V}$，那么这必然成立；否则，它断言存在函数依赖。

对于形式为 x :- $c_1, \ldots, c_M$ 的简单类型，我们可以得到：$|s| \leq \prod_{m=1}^{M} \lceil c_m \mid \mathcal{V}_{<m} \rceil$ 的基数上界，其中 $\mathcal{V}_{<m} = \bigcup_{i=1}^{m-1}$ 变量 (c_i)。[146] 更一般地，给定对某个变量集合 $\mathcal{V}$ 的赋值，s 的条件基数上界为 $\prod_{m=1}^{M} \lceil c_m \mid \mathcal{V} \cup \mathcal{V}_{<m} \rceil$。对所有序进行最小化，即可得到最终的上界 $\lceil s \mid \mathcal{V} \rceil$（其中 $\lceil s \rceil \overset{\text{定义}}{=} \lceil s \mid \emptyset \rceil$）。

不过，这些上界 $\lceil c \mid \mathcal{V} \rceil$ 是从哪里来的？考虑内置约束times，它是一个包含三个数字的无限大关系。由于 times 具有函数依赖，因此只要 $\mathcal{V}$ 至少包含 {X、Y、Z} 中的两个，我们就可以取$\lceil$times(X,Y,Z) $\mid \mathcal{V} \rceil = 1$。我们允许用户用以下符号表示声明这种条件基数：

```
259 |times(+X,  +Y, Z)| ≤ 1.
260 |times(X,  +Y,  +Z)| ≤ 1.
261 |times(+X, Y,  +Z)| ≤ 1.
```

例如，#261 声明，给定X 和Z的值，times(X, Y, Z) 约束最多只会由Y的一个值满足。

给定所有此类声明，我们得到 $\lceil c \mid \mathcal{V} \rceil$ 作为与 c 合一的条件基数声明所提供上界的最小值，且这些声明的

[146]前提是头部中的所有变量 x 都在体中的某处受到约束。如果不是，$|s| = \infty$。

标记为 + 的变量都在 $\mathcal{V}$ 中。这包括一个隐式声明，即 $\lceil c \mid \text{变量}(c) \rceil \leq 1$。（另一个隐式声明是 $\lceil \text{失败} \rceil \leq 0$。）

同样的符号表示也适用于界定输入关系，例如单词关系，见示例6。

```
|单词(W, +I, K)| ≤ 1。
|单词(W, I, +K)| ≤ 1。
|单词(W, I, K)| ≤ n。
```

第一个声明表示，每个位置I至多对应一个单词W，并与一个结束位置K配对。第二个声明与之恰好相反。第三个声明表示，输入句子中总共至多有 n 个单词，其中符号 n 是一个**大小参数**。

我们还可以使用大小参数来界定类型参数的基数：`|k(X)| ≤ k。 |n(I)| ≤ n。 |w(W)| ≤ w。`

它将非终结符的数量界定为 k，句子位置界定为 n，终结符界定为 w。这里，k、n 和 w 都解释为符号。我们将组合这些符号，构成一个**参数表达式**，以此为程序中的项的基数提供上界。例如，正如我们之前所见，$\lceil \beta(X,I,K) \rceil = kn^2$。将简单类型的大小相加后，我们的系统可以计算所得表达式的渐近上界，例如，
$kn^2 + kn + k^2n = \mathcal{O}(kn\max(n,k))$.

最后一个技巧是指定复合约束的基数或条件基数，使其可以与 c_1、……、c_M 的某个子序列进行匹配，以便构造上界。例如，`|even(X),prime(X)| ≤ 1` 表示只有一个偶素数，而 `|γ(X,Y), k(Y)| ≤ g` 表示没有非终结符号X 可以 rewrite 为超过 g 个不同的非终结符号 Y。（但这并不

会限制 x 可以通过多少种方式 rewrite 为终止状态符号y，因为这些方式不满足 k(y) 约束。）

局限性。

• 某些无限关系（例如可以用自由变量和延迟约束表示的关系），在这种方法下估计大小为 ∞（参见脚注 146）。不过，求解器可以将某些无限关系高效地表示为有限大小的程序，见 §5 。然而，由于我们的方法依据非 0 项的数量而非其表示大小来确定界限，我们最终得到的大小估计极其宽松。

• 我们假设所有内置关系都会将其中出现的所有变量实例化为基。未来工作应整合更精细的模式系统（例如 Overton（2003年）和 Filardo（2017年）），以推理变量的实例化状态。模式分析也可用于推理延迟约束的集合，因此有助于解决前文提到的局限性。

• 我们的空间界始终是输入大小参数的（分段）多项式；例如，它们无法表示指数级大小界。

6.3.2 时间分析

假设 Dyna 程序由基于待办队列的算法（算法 8）执行。该算法维护两个程序：**图表** m 和**议程** ∂。图表表示当前对不动点的估计（ §4.3 ）。

议程是一个更新队列，用于更新 $\mathfrak{m}$中的值。当 y' 的更新出队时，其值会在图表中更新。随后，这一变化会传播到其他项 x。这些 x 项的更新会被放入议程，稍后才执行。关键在于，通过将 x 项与任意规则的子目标进行匹配来找出受影响的项（ y'（即**驱动程序**），然后查询图表，找出与该规则其他子目标匹配的项（**乘客**）。

从给定的驱动程序传播更新 y' 穿过规则 r 需要我们找出所有满足以下条件的赋值： $体(r)$ 这些赋值与将驾驶者子目标绑定到 y'相一致。虽然 r 不是一种类型，但其体中的子目标（例如$\gamma(X,Y)$ ）确实会对 X 和Y施加约束——具体而言，是类型分析（§6.1）得出的约束 (X,Y) ，只有当 $\gamma(X,Y)$ 可能在图表中有值时，该约束才成立。

假设我们已将驱动程序 y' 与 r 的一个子目标匹配，并且已将零个或多个其他子目标与从图表中检索到的乘客匹配。这些匹配已实例化规则中的一些变量 $\mathcal{V}$。对于每个子目标匹配，我们还可以从静态类型分析（§6.1.2）得知，该匹配涉及的变量必须满足某些约束 $\mathcal{C}$。如果 γ(X,Y) 是我们接下来要与图表匹配的子目标，那么最多会得到 $N \stackrel{\text{def}}{=} \lceil \mathcal{C}, \gamma(X,Y) \mid \mathcal{V} \rceil$个答案。上一节介绍的方法可以计算出这个上界，即使 γ 不是内置约束或参数化约束：只需通过与 γ(X,Y) 合一来细化项类型 $\mathfrak{m}$ ，其中与之合一的是所有简单类型 $s \in \mathfrak{m}$ 的头部，然后对这个细化后的类型 $\mathfrak{m}'$的条件基数求上界即可。

得益于图表的索引，获取和处理 N答案的运行时间在最坏情况下与 $1+N$ 成正比（其中 1 代表

即使没有答案，尝试查询也会产生开销）。对于每个答案，接下来我们必须以所有可能的方式匹配其余乘客。

算法19 中的伪代码明确说明了这一点。有两点细节值得指出。首先， $\mathfrak{m}$ 具有析取性。我们根据先前匹配的乘客累积的约束 $\mathcal{C}$ 取决于覆盖这些乘客的具体简单类型，因此我们必须遍历所有可能性。其次，虽然驾驶者子目标由议程选定，但其余子目标（乘客）的实例化顺序由我们决定。伪代码总是根据先前匹配的乘客的简单类型，选择能够使运行时间上界最小化的下一个子目标。换言之，我们没有遵循规则 r中规定的固定子目标顺序，而是利用静态分析优化子目标排序，从而获得更好的运行时间。[147]

正确性与局限性。 算法 19 给出的运行时间上界假设每个项驱动的次数是 常量。如果程序无环，那么按拓扑顺序排列的 议程 (§5.1.4) 可保证每个项最多弹出一次，因此该上界是正确的。不过，如果程序有环，项的驱动次数可能超过常量次。在有环的情况下，我们可以将该上界理解为一次迭代的最坏情况复杂度，

[147]未来工作可以探索一些方法，直接优化乘客查询顺序的平均执行时间，而不是优化其上界。可以利用机器学习技术估算并优化平均执行时间。Vieira、Francis-Landau 等人（2017年）介绍了一个用于此类优化的通用框架。按照他们的术语，具体的乘客排序是回答乘客查询这一任务的策略。他们的方法还允许由一个策略动态决定排序选择；该策略可以访问具体的乘客查询（也就是说，不局限于查询的类型）以及可用索引数据结构的状态。

算法 19 运行时间：为给定导出类型 $\mathfrak{m}$ 的 $\mathfrak{p}$ 推断运行时界。

```
定义 运行时间(p, m):
    t ← 0                                                    ▷ 累加器
    对于 (y' : C') ∈ m:                                      ▷ 可能的驱动类型
        对于 (x ⊕= y_1 ∘ ··· ∘ y_K) ∈ p:                     } 将匹配规则与驱动程序 y' 进行匹配
            对 k ∈ [1 : K]:                                  }
                for _ ∈ 合一(y_k, y'):                       }
                    加上执行当前规则的代价
                    t += 运行时间'({y_1, ..., y_K} − {y_k}, C', 变量(y_k))   ▷ 注：y' = y_k 由于合一
                                  乘客
    返回 t
▷ 这个递归函数（可通过记忆化提高效率）告诉我们子目标集合的最优运行时长（
  ，P和 C 分别为已累积的合取约束集合和
  绑定变量集合。 V
定义 运行时间'(P, C, V):
    如果 P = ∅: 返回 1                                       ▷ 基本情况：没有剩余子目标
    最佳 ← ∞
    ▷ 针对下一个子目标的选择进行优化 p ，从剩余子目标的集合中选取
    对于 p ∈ P:
        t ← 0                                                ▷ 累计总后缀运行时间（给定选择 p）
        对于 ⟨C', θ⟩ ∈ m[p]:                                 ▷ 在 m 中查找与 p匹配的简单类型。注意：我们
                                                               要求此处使用显式的结果流（§3.1.2.1）。
            C'' ←传播 (θ (C ∘ C'))
            q ← ⌈C'', p | V⌉                                 ▷ 为乘客查找条件基数 p
            如果 q = 0: 继续                                 ▷ 为提高效率，请避免递归调用
            t += q·运行时间'(θ(P − {p}) C'', V ∪ 变量(p))     ▷ 递归
        最佳最小值= t
    返回 1 + 最佳
```

半朴素 (算法 7)，它是输入规模参数的函数，而非执行的总复杂度。要得到总复杂度，我们必须调整上界，以计入每种简单类型可能弹出的次数；该次数可能取决于输入规模参数和所采用的优先级策略。[148,149]

⌐其他局限性：由于我们的运行时间界6.3.1基于 § 中的空间复杂度分析，因此也继承了其局限性。特别是，我们的运行时间界

[148]关于近似不动点和收敛速度的讨论见 §4.4。[149]回想一下（§5.4）：通过将循环程序分解为强连通分量，可以更高效地 solve。未来用于推理循环程序运行时间的系统应包含静态强连通分量分解分析。

是输入规模参数的（分段）多元多项式。例如，这意味着我们的系统无法推断指数运行时间上界；它只会预测运行时间无限大，因为多项式不可能给指数函数提供上界。我们还要指出，只要图表中可能出现无限基数延迟约束，我们的运行时间分析器就会给出无限上界，尽管我们的执行算法在处理这些约束时并不会产生无限运行时代价。最后，我们的运行时间界假设半环运算都能在常数时间内完成。

⊏未来工作的想法：静态分析程序的循环性，或更一般地说，分析程序的强连通分量，显然有助于改进运行时间分析。若能为第5 节中的其他算法（例如 牛顿法）提供运行时间界，也会很有用。我们通过模拟这些算法每次迭代所执行工作的通用方法，应该很容易推广到这些情况。针对特定半环族（例如交换幂等半环）提供专门的运行时间界也会很有意思，因为它们可能具有有限时间收敛保证。

6.4 应用

本节提供了更多关于我们静态分析器的示例。**例21**。*Kuhlmann*、*Gómez-Rodríguez*等（*2011*年）在评分函数采用弧分解形式时，为弧先行句法分析（*Nivre*，*2003*年）引入了一种精确动态规划算法（*Eisner*，*1996*）。下面给出了他们算法的 *Dyna* 编码：

```
β(I,I^0,K) +=单词(I,K). β(I,I^1,K) +=单词(I,K). β(I,H^
B,K) += β(I,H^B,J) * 评分(H ← H') * β(J,H'^0,K). β(I,H^B,K)
 += β(I,H^B,J) * 评分(H → H') * β(J,H'^1,K). 目标 += β(0,
0^0,N) * 长度(N). 参数：单词；评分；长度。
```

朴素地看，人们可能会认为弧急切解析器的运行时间为 $\mathcal{O}(n^5)$。然而，更精细的分析表明，它实际上的运行时间为 $\mathcal{O}(n^3)$。为了推导这一点，请考虑以下类型声明：

```
参数: n(N). score(H → H') :-
n(H), n(H'). score(H ← H') :-
n(H), n(H'). 单词(I,K) :- n(I),
n(K). 长度(N) :- n(N).
```

导出类型为

```
β(I,I^0,K) :- n(I), n(K). β
(I,I^1,K) :- n(I), n(K). 目标.
```

导出类型令人惊讶的一点是，尽管规则表明 `H` 在 `β(I,H^B,K)` 中可以独立地在 `n`上取值，但结果发现它始终等于 `I`，因为只有与之合一的类型才是 `β(I,I^0,K)` 和 `β(I,I^1,K)`。现在，我们结合以下输入规模声明来考虑运行时间：

```
|n(N)| ≤ n.
```

推断出的运行时间界为 $\mathcal{O}(n^3)$。

缩写（在……下面）成功去掉了不必要的变量，从而以常数因子缩短了运行时间。下面的程序与 *Kuhlmann*、*Gómez-Rodríguez*等人（*2011*年）手动发现的程序同构。我们去掉了`n(I)` 约束，使输出更加易读。我们选择了与原始名称相似的缩写名称；使用下标来表示简单类型的标识符。

281 % 将每种输入类型映射到其缩写 282 $score_0$(H,H') += score(H $\rightarrow$ H'). 283 $score_1$(H,H') += score(H $\leftarrow$ H'). 这些项已被扁平化（嵌套项已移除） 284 $word_2$(I,K) +=word(I,K). 285 len_3(N) += len(N). 286 % 恢复规则287 β(I,I^0,K) += β_4(I,K). 288 β(I,I^1,K) += β_5(I,K). 这些是无用的恢复规则，可以删除 289 目标 += 目标$_6$. 290 % 缩写程序规则 291 β_4(I,K) +=$word_2$(I,K).292 β_4(H,K) += β_4(H,H') * $score_0$(H,H') * β_5(H',K).293 β_4(H,K) += β_4(H,H') * $score_1$(H,H') * β_4(H',K).294 β_5(I,K) += $word_2$(I,K).295 β_5(H,K) += β_5(H,H') * $score_0$(H,H') * β_5(H',K).296 β_5(H,K) += β_5(H,H') * $score_1$(H,H') * β_4(H',K). 297 目标$_6$ += β_4(0,N) * len_3(N).

推断出的运行时界不受缩写变换的影响。

示例22。 下面我们给出 *CKY* 的另一个版本（示例 *6*），该版本将输入句子编码为一个列表，例如 `sentence([Papa,吃了,鱼子酱])`。此版本将加权产生式规则编码为 `γ(X,Y,Z)` 和 `γ(X,Y)`，与之前一样，但不要求句子使用`word(W,I,K)` 或 `length(N)` 项进行编码。和之前一样，项目标 会对所有完整解析的总权重求和。

298 参数：输入； γ。输出：目标。 299 buffer(Ws) += sentence(Ws). % 初始化缓冲区 300 buffer(Ws) += buffer([W|Ws]). % 移进301 β(X,[W|Ws],Ws) += γ(X,W) * buffer([W|Ws]). 302 β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K).303 β(X,I,K) += γ(X,Y) * β(Y,I,K). 304 目标 += β(s,Ws,[]) * sentence(Ws).

我们指定以下输入类型：

305 参数：w；ws；k。306 γ(X,Y,Z) :- k(X), k(Y), k(Z). 307 $\overline{\gamma}$(X,Y) :- k(X), k(Y). sameasExample18308 γ(X,W) :- k(X), w(W). 309 句子(Ws) :- ws(Ws)。

回想一下 *th* *at* k 是非终结符的集合，而w 是单词的集合。类型为 *r*

语法规则与我们在§6.1中分析的 *CKY* 版本相同。

我们将输入定义为类型ws，它是一个由单词组成的列表。添加以下传播规则后，即可编码表示：我们列表中的每个后缀也都是一个由单词组成的列表。

```
ws([]) <== 真。
ws(Xs) <== ws([X|Xs])。
```

最后，我们添加以下传播规则

```
w(X) <== ws([X|Xs]).
失败 <== k(X), w(X).
```

第一条规则表示，单词列表的第一个元素是一个单词；第二条规则表示，非终结符集合 k 与单词集合w互不相交。

抽象 *act* 前向链为派生项推导出以下类型 s

```
缓冲区(Xs):-ws(Xs).
(X,I,K):-k(X),ws(I),ws(K).
```
β
目标.

运行时界与普通 *CKY* 相同， $\mathcal{O}(k^3\, n^3)$不过，输入长度不是编码为句子位置的数量 $|\mathtt{n(I)}| \leq n$

而是编码为输入句子中的后缀数量 $|\mathtt{ws(Ws)}| \leq n$。

在这种情况下，缩写变换并无意义；它所做的只是更改现有项的名称，没有任何益处。

总结

本章介绍了多种技术，用于独立于特定输入数据，对给定程序进行推理。我们描述了如何构建系统，根据由未知符号用户参数刻画的输入数据，推导程序非0值项集合的符号上界。基于这项技术，我们估算了有用集合，以便回答有关特定输出项集合的查询。随后，我们利用估算出的有用项集合对程序进行特化，通过省略无用项，使程序能够更高效地计算这些项。接着，我们转而分析空间和时间复杂度。我们介绍了一种方法，用用户指定的规模参数来界定非0 项集合的大小。我们将这一上界与基于议程的不动点迭代的空间复杂度联系起来。随后，我们给出一种技术，用于根据输入规模参数分析基于议程的不动点迭代的运行时复杂度。我们将在程序优化器（§8）中重新探讨复杂度上界和程序特化技术。

第 7 章

程序变换

在本章中，我们将构建一组保持含义不变的程序到程序变换。这些变换将用于系统地将给定的初始程序转换为一个计算相同值但速度更快的程序，相关内容见§8。

直观地说，程序变换可以被视为用于将一组递归定义（一个 Dyna 程序）转换为另一组不同的（可能求值更快的）递归定义（另一个 Dyna 程序）的可复用技巧。由于 Dyna 程序本质上是定义赋值函数（§4）的方程组，因此我们可以将程序变换视为类似于用于处理数学方程组的数学恒等式（例如，用于 solve 方程组或证明新的恒等式）。程序变换既可以手动应用，也可以自动应用。

程序　m变换——不同于执行算法（§5）——可以操作　e

无需具体输入数据。[150] 因此，许多变换对应于抽象解释的步骤（Cousot 和 Cousot，1977年），因为它们模拟某个具体求解器所采取的步骤。因此，我们引入**输入和输出声明**：我们假设变换序列中的初始程序将某些项声明为**输入**和/或**输出**项。其余的被视为**中间项**。下面是一个带有输入–输出声明的CKY示例。

```
319  β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K).320  β
(X,I,K) += γ(X,Y) * β(Y,I,K).321  β(X,I,K) += γ(X,Y) *
word(Y,I,K). 322 目标 += β(s,0,N) * len(N). 323 324
输入：word(X,I,K); γ(X,Y,Z); γ(X,Y); len(N). 325 输
出：目标。
```

相同的代码输入–输出声明

若要被视为正确，程序变换必须保留从输入项的赋值到原始程序输出项赋值的映射（§4）。然而，程序变换可以引入、删除或更改中间项。[151]

概要。 接下来，我们将非正式地概述通过程序变换可以实现的加速，并在后续章节中对每种变换作更正式的论述。

7.1 非正式介绍.1447.2 保语义变换.1527.3 定义引入变换.1567.4 折叠与展开变换.160

[150]静态分析（§6.1）是另一个例子。[151]我们将在 §7.2中形式化地定义变换正确性和程序等价。

7.5 提升变换.1867.6 讨论.201

7.1 非正式介绍

本节概述本章将要探讨的各类变换，并介绍其大致特点。每种变换的正式细节将在后续章节中说明。我们还指出，§6.3 中的方法可用于自动推导$\mathcal{O}$ 下面示例中空间和时间复杂度的渐近界。

定义与折叠。我们从非递归折叠开始讨论，因为我们认为大多数读者都熟悉这种方法。

示例 23。下面的程序计算图中长度为 *4* 的路径的总权重，图的边权重为`w`:

```
326  total += w(Y1,Y2) * w(Y2,Y3) * w(Y3,Y4) * w(Y4,Y5).
```

利用分配律，我们可以将此程序改写为一个程序，使其针对每种路径后缀长度重用乘积的部分和:

```
327 总计  +=其余部分1(Y1)。328 其余部分1(Y1)
 += w(Y1,Y2) * 其余部分2(Y2)。329 其余部分2(
Y2)  += w(Y2,Y3) * 其余部分3(Y3)。330 其余部分
3(Y3)  += w(Y3,Y4) * 其余部分4(Y4)。331 其余部
分4(Y4)  +=w(Y4,Y5)。
```

该程序计算出相同的`total`值，但与原始程序相比，求值效率显著更高。具体而言，原始程序的运行时间为 $\mathcal{O}(N^5)$ ，其中 N 表示图中的节点数，而

变换后的程序运行时间为 $\mathcal{O}(N^2)$。这是定义变换的一个示例；该变换提出新的 $\texttt{rest}_i$ 定义，而折叠变换则将这些定义嵌入程序中。

读者可能会注意到，连接树算法在概率图模型推理和数据库查询规划中的连接顺序优化方面，采用了相同的变换方案（Lauritzen 和 Spiegelhalter，1988年；Shafer 和 Shenoy，1990）（Chekuri 和 Rajaraman，1997年）。Gildea（2011年）讨论了连接树算法在自然语言处理中所用加权演绎系统中的诸多用途。然而，这些方法并不变换递归计算——它们针对的是固定的变量集合，而非递归计算中那样无界的集合。[152]

示例 24。回顾一下，*CKY*（示例*6*）有以下耗时较高的规则

```
β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K).
```

通过对变量Y 单独求和，而不与K一起求和，可以加速这条规则：

```
β(X,I,K) +=临时项(I,J,X,Z) * β(Z,J,K).
临时项(I,J,X,Z) += γ(X,Y,Z) * β(Y,I,J).
```

变换后的规则将运行时间从 $\mathcal{O}(K^3N^3)$ 缩短至 $\mathcal{O}(K^2N^3 + K^3N^2)$，其中 K 表示非终结符的数量，N 表示句子长度。[153]

展开。展开变换本质上是折叠变换的逆变换，对应于代码内联。它将一个特定的

[152]值得注意的是，GMTK（Bilmes和Zweig，2002年）提供了一个连接树优化器，可以优化递归图模型；但据我们所知，他们的系统并未像本章这样，在递归层级之间进行静态优化。[153]在自然语言处理领域，这种变换通常称为钩子技巧 （Eisner和Satta，1999a；Huang、Zhang等，2005年；Gildea，2011年）。

子目标 $y_k \in$体$(\mathbf{R})$作为某条规则$\mathbf{R}$的输入。目标是用 y_k 的定义替换$\mathbf{R}$ 里面的内容；然而，其定义可能跨越多条规则$\mathbf{R}'$。因此，我们将从程序中移除$\mathbf{R}$ ，并针对头部与 y_k合一的每条规则$\mathbf{R}'$ ，用$\mathbf{R}$ 的一个特化版本替换它。这些规则$\mathbf{R}'$ 定义了 y_k——但有一种特殊情况除外：当 y_k 匹配任意输入项时，我们无法展开y_k ，因为其完备定义不可用。请注意，展开可能会使某些规则变得多余。[154]

示例 25。 假设用户提供了下面这个低效的程序，它可能是通过以次优的方式折叠示例*23*而得到的。

335 目标 += 临时项$_1$(X$_1$,X$_4$,X$_5$)。336 临时项$_1$(X$_1$,X$_4$,X$_5$) +=临时项$_2$(X$_1$,X$_4$) * w(X$_4$,X$_5$)。337 临时项$_2$(X$_1$,X$_4$) +=临时项$_4$(X$_1$,X$_2$) * 临时项$_3$(X$_2$,X$_4$)。338 临时项$_3$(X$_2$,X$_4$) += w(X$_2$,X$_3$) * w(X$_3$,X$_4$)。339 临时项$_4$(X$_1$,X$_2$) +=w(X$_1$,X$_2$)。

虽然上面的程序是正确的，但其运行时间为 $\mathcal{O}(N^3)$，比我们之前看到的$\mathcal{O}(N^2)$ 版本更差。它没有可以消除的变量，因此没有折叠操作或新定义能够改善其运行时复杂度。要改进它，我们首先必须撤销这些糟糕的选择。如果我们多次展开示例*25*，就可以得到示例*23*中的版本。然后，我们可以重新折叠，从而获得同样高效的版本。

展开变换通常会提高或保持程序的运行时间阶数，但也存在异常：

示例 26。

[154]请记住，在 §6.2 中介绍的方法可以识别并删除基于已声明的输入而无法触发，或未被任何已声明的输出使用的规则。

```
340 a(I,K)  +=b(I,J) * c(J,K).341
轨迹  +=a(L,L).
```

展开 a(L,L) 在.....里面 #341 将程序的运行时间从 $\mathcal{O}(N^3)$ 降至 $\mathcal{O}(N^2)$假设 N 程序中任意变量基数的上界。

```
342  轨迹 +=b(L,J)*c(J,L)。343 a(I,K)
 +=  b(I,J) * c(J,K)。
```

因此，这是一个立即有用的展开示例。在本例中，程序计算矩阵乘积的迹，而展开的作用是 特化计算整个矩阵乘积的子程序a(I,K) 到乘积的使用位置；此处只需要其对角线 a(L,L)。程序员很容易忽略这类优化。[155]

递归转换。 在下面的示例中，我们转换一个归纳定义——实际上消除了一个在（无界）递归中传播的变量。仅凭概率图模型和数据库查询规划中的工具，无法进行这样的rewrite。

示例 27。 我们从以下程序开始，它计算图中所有路径的总权重（目标）；这些路径的起点属于集合 起始，终点属于集合 停止。

```
344 目标  +=  起始(I) * 路径(I,K) * 停止(K). 345
路径(I,I). 346 路径(I,K)  +=  路径(I,J) * 边(J,K).
```

该程序的运行时间是 $\mathcal{O}$（N^3），其中 N 是图中节点的数量。我们将通过替换中间的 $\mathcal{O}$（N^2）程序来推导出更高效的路径(I,K) 项，这些项计算从I 到

[155]许多数值库（例如 Theano开发团队（2016）、Abadi 等人（2016）以及 Smith 和 Gray（2018））都实现了类似这样的优化。然而，与图模型的情况一样，这些系统也存在局限，因为它们无法对递归定义的计算进行推理。

K 的所有节点有序对的路径总权重，其中startpath(K)计算从起始中的任意节点到节点K的路径总权重。

假设我们生成一组新的项，这些项至少在最初由以下规则定义，并将该规则附加到程序中：

```
347 startpath(K)  += 起始(I) *路径(I,K).
```

接下来，我们通过 #344 #347：

```
348  目标 += startpath(K) * 停止(K)。
```

与示例 23 中一样，通过引入一个中间项——在本例中是 startpath——我们成功降低了 #344 的运行时间，因为它的替换项 #348 和 #347 的运行时间为 $\mathcal{O}(N + N^2)$，而不是 $\mathcal{O}(N^3)$。遗憾的是，startpath 仍然依赖于 path，而它目前是该程序运行时间的瓶颈。因此，我们将尝试用 startpath 本身而非 path 来改写 startpath 。下一步是展开 path 子目标，该子目标位于 startpath 规则（#347）中：

```
349 startpath(K) += 起始(K)。 350 startpath(K) += 起始(I) *
路径(I,J) * 边(J,K)。
```

接下来，我们认识到 start(I) * path(I,J) 在.....上 #350 可以通过根据其先前的定义进行折叠（#347）替换为 startpath(J)。我们现在有

```
351  startpath(K) +=起始(K)。 352  startpath(K) +=
startpath(J)*边(J,K)。
```

最后，无用规则消除过程（§6.2）检测到，路径规则无法从目标到达，而目标是我们唯一声明的输出。我们的最终程序是

```
353 目标 += startpath(K) * 停止(K)。 354
startpath(K) +=start(K)。 355  startpath(K) +=
startpath(J)*edge(J,K)。 356  path(I,I).357
path(I,K) += path(I,J) * edge(J,K)。
```

改进后程序的运行时间为 $\mathcal{O}(N^2)$。

请注意，并非所有折叠都是安全的。例如，如果我们不展开 #347，而是立即将 startpath 的定义与其自身折叠，就会得到如下非良基定义：`startpath(K) += startpath(K)`。在 §7.4中，我们将描述用于拒绝这些错误折叠并确保安全折叠的充分条件。

请注意，要在此示例中替换 path，我们需要生成一个新的关系startpath来替代它。尽管 startpath 最初是根据 *path*来定义的——经过几次变换后，它变得可以自给自足。由于发现新定义（例如 startpath）往往需要“灵光一现”，因此人们亲切地称这类定义为尤里卡定义 *(Pettorossi* 和 *Proietti, 1994)*。幸运的是，人们有许多常用策略来提出合理的尤里卡定义，例如像我们所做的那样，将现有子目标连接起来。我们将在 §7.3中讨论本系统采用的策略。

下一个示例采用与示例27相同的总体思路，只不过这里我们要从一个低效程序合成一个高效的动态规划算法。该程序会显式地对图中无限多个路径求和，其中每条路径都表示为一个包含其访问过的节点的列表。

示例 28. 以下程序计算图中所有路径的总权重。它采用非常直观的写法，将沿路径已访问的节点列成列表。

```
358 目标 += 起始(X) * β([X|Xs]).359 β([X1,X2|Xs])
 += 边(X1,X2) * β([X2|Xs]).360 β([X]) +=停止(X).
```

由于它是 w 如此写成时，该程序的效率极低，因为可能存在无限多条 y

路径，这会转化为无限多个非-0 β 项。为了提高该程序的速度，我们引入以下定义，它对列表的尾部求和Xs，同时暴露出路径中的第一个节点X：

```
361 b(X) += β([X|Xs]).
```

Next，展开 #*361* 并得到以下两条替代它的规则：

```
362 b(X1) += 边(X1,X2) * β([X2|Xs])。 363
b(X) += 停止(X)。
```

随后我们将其折叠 #*362* 通过 #*361*：

```
364 b(X) += b(Y)* 边(X,Y)。 365
b(X) += 停止(X)。
```

现在，b 是以自身而非其原始规约#*361* 递归定义的；原始规约使用了低效的 β 项。此外，由于图中节点有限，非零的0 b 项也有限。最后，我们通过#*361* 折叠 #*358*，将b *graft* 到目标规则中：

```
366 目标 += 起始(X) * b(X)。
```

最终程序如下；我们注意到，β 项可用，但现在规则无法用于推导目标，因此求解器不需要运行它们。

```
367 目标 += 起始(X) * b(X)。 368 b(X) +=
b(Y) * 边(X,Y)。 369 b(X) += 停止(X)。370 β
([X,Y|Xs]) += 边(X,Y) * β([Y|Xs])。371 β([
X]) +=停止(X)。
```

推测与左角变换。 下面的示例简要展示了如何通过推测变换（§7.5）对递归方程进行有用的重新排列。

示例 29（单元环分解；改编自 Eisner and Blatz（2007））**.**

再次看一下 *CKY* 程序（示例 *6*）：

372 β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K)。373 β(X,I,K) += γ(X,Y) * β(Y,I,K)。374 β(X,I,K) += γ(X,W) * 单词(W,I,K)。375 个目标 += β(s,0,N) * 长度(N)。

这个程序存在一项低效之处：对于所有跨度 ⟨I，K⟩，它都会重复进行相同的一元链闭包。我们运用推测对程序进行重构（见下文），从而可以预先计算一元链闭包（不依赖于句子），并在每个跨度处重复使用。这项实现技巧归功于 *Stolcke*（*1995*），是概率句法分析中的标准做法。

376 u(X,X') += γ(X,Y) * u(Y,X'). 377 u(X,X') += γ(X,X').378 β(X,I,K) +=u(X,X') * β'(X',I,K).379 β(X,I,K) += β'(X,I,K).380 β'(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K).381 β'(X,I,K) += γ(X,W) * word(W,I,K). 382 目标 += 长度(N) * β(s,0,N).

（通过密切相关的左角变换（§7.5.2），也可以实现几乎相同的程序优化。）

下一步。 本节概述了本章将介绍的程序变换。在下文中，我们将对每种变换进行形式化定义，并进一步举例说明其用途及适用场景。不过，首先我们将对程序等价和变换安全性进行形式化定义。

7.2 保语义变换

我们称程序到程序变换 $\mathfrak{p}^{(s)} \to \mathfrak{p}^{(t)}$ 具有 **语义保持性**（简称**安全的**），如果程序 $\mathfrak{p}^{(s)}$ 和 $\mathfrak{p}^{(t)}$ 语义上**等价**，也就是说，它们定义了完全相同的赋值映射（§4）。[156] 简写为 $\mathfrak{p}^{(s)} \equiv \mathfrak{p}^{(t)}$，表示（或规定）这两个程序语义等价。换句话说，对于 $\mathfrak{p}^{(s)} \equiv \mathfrak{p}^{(t)} \iff \mathfrak{p}^{(s)}[\![\boldsymbol{x}]\!] = \mathfrak{p}^{(t)}[\![\boldsymbol{x}]\!]$ 所有 $\boldsymbol{x} \in \mathbb{H}$，我们还将使用以下更强的语义等价概念，它蕴含前述概念（命题5）。

定义31。 我们称 $\mathfrak{p}^{(s)}$ 和 $\mathfrak{p}^{(t)}$ 是强等价，如果存在一个保值双射 $\phi : \mathfrak{p}^{(s)}\{\cdot\} \to \mathfrak{p}^{(t)}\{\cdot\}$，其中 $\mathfrak{p}^{(s)}\{\cdot\}$ 和 $\mathfrak{p}^{(t)}\{\cdot\}$ 分别是$_{s\to t}$中所有推导的集合 $\mathfrak{p}^{(s)}$ 和 $\mathfrak{p}^{(t)}$。

定义32。 函数 $\phi : \mathfrak{p}^{(s)}\{\cdot\} \to \mathfrak{p}^{(t)}\{\cdot\}$ 是一个保值双射，当且仅当对所有 $\boldsymbol{\delta}' \in \mathfrak{p}^{(t)}\{\cdot\}$，都存在唯一的 $\boldsymbol{\delta} \in \mathfrak{p}^{(s)}\{\cdot\}$ 使得 $\phi(\boldsymbol{\delta}) = \boldsymbol{\delta}'$，$[\![\boldsymbol{\delta}]\!] = [\![\boldsymbol{\delta}']\!]$，并且 $\boldsymbol{\delta}$ 和 $\boldsymbol{\delta}'$ 具有相同的根标签。

直观来说，对于每个项 $\boldsymbol{x}$，函数 ϕ 描述了从$_{s\to t}$使用 $\mathfrak{p}^{(s)}\{\boldsymbol{x}\}$ 中的规则的推导，$\mathfrak{p}^{(s)}$ 到使用 $\mathfrak{p}^{(t)}\{\boldsymbol{x}\}$ 中的规则的推导的翻译，$\mathfrak{p}^{(t)}$。然而，值保持对这种翻译施加了限制，要求 $\boldsymbol{x}$ 的每个推导所贡献的值都相同。双射性确保没有推导丢失，也不会产生虚假推导。

[156]请注意，§4 并未直接讨论部分指定程序的语义，即带有输入关系的程序。幸运的是，输入关系可以归约为内置关系，因为它们只是一些项，其含义是在当前程序之外指定的。

例30。考虑几何级数程序。我们将说明，如何系统地为不同程序的不同推导集合建立索引，从而共同构成给定项x的值所对应的贡献多重集。

$$\mathfrak{p}^{(0)} = \begin{cases} 383 & \mathtt{x \mathrel{+}= b.} \\ 384 & \mathtt{x \mathrel{+}= a * x.} \end{cases}$$

值 x 等于一个由（无穷多个）值组成的多重集之和，$[\![\mathtt{x}]\!] = \mathtt{b} + \mathtt{a\,b} + \mathtt{a}^2\,\mathtt{b} + \mathtt{a}^3\,\mathtt{b} + \mathtt{a}^4\,\mathtt{b} + \mathtt{a}^5\,\mathtt{b} + \mathtt{a}^6\,\mathtt{b} + \cdots$。在.....下面，我们展示了项x的 7 个（无穷多个）推导。我们看到，每个推导的值（即其叶子的乘积）都对应于上面的加项序列。

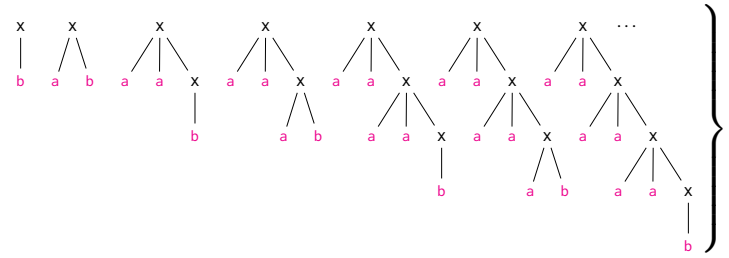
$$\mathfrak{p}^{(0)}\{\mathtt{x}\} = \left\{ \begin{array}{l} \mathtt{x} \\ \mathtt{b} \end{array} \quad \begin{array}{l} \mathtt{x} \\ \mathtt{a\ x} \end{array} \quad \begin{array}{l} \mathtt{x} \\ \mathtt{a\ x} \\ \mathtt{b} \end{array} \quad \begin{array}{l} \mathtt{x} \\ \mathtt{a\ x} \\ \mathtt{a\ x} \\ \mathtt{b} \end{array} \quad \begin{array}{l} \mathtt{x} \\ \mathtt{a\ x} \\ \mathtt{a\ x} \\ \mathtt{a\ x} \\ \mathtt{b} \end{array} \quad \begin{array}{l} \mathtt{x} \\ \mathtt{a\ x} \\ \mathtt{a\ x} \\ \mathtt{a\ x} \\ \mathtt{a\ x} \\ \mathtt{b} \end{array} \quad \begin{array}{l} \mathtt{x} \\ \mathtt{a\ x} \\ \mathtt{a\ x} \\ \mathtt{a\ x} \\ \mathtt{a\ x} \\ \mathtt{a\ x} \\ \mathtt{b} \end{array} \quad \cdots \right\}$$

程序 $\mathfrak{p}^{(1)}$ 在.....下面（通过展开 #384 得到）与上面的程序强等价，$\mathfrak{p}^{(1)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(0)}$。

$$\mathfrak{p}^{(1)} = \begin{cases} 385 & \mathtt{x \mathrel{+}= b.} \\ 386 & \mathtt{x \mathrel{+}= a * b.} \\ 387 & \mathtt{x \mathrel{+}= a * a * x.} \end{cases}$$

为了理解原因，我们检查项的推导x。下面，我们已对推导进行排序$\mathfrak{p}^{(1)}\{\mathtt{x}\}$，使从左到右的序展示出与$\mathfrak{p}^{(0)}\{\mathtt{x}\}$的对应关系（即双射）。

$$\mathfrak{p}^{(1)}\{\mathrm{x}\} = \left\{ \right.$$



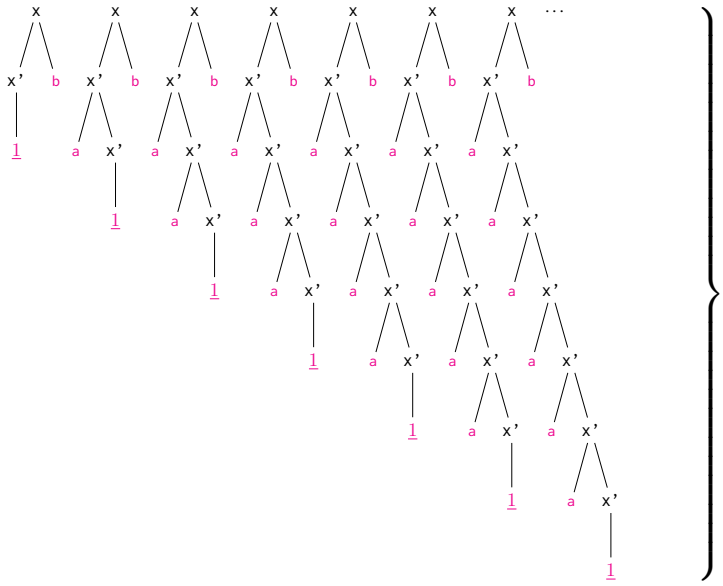
$\left. \right\}$

检查这些树的叶子后，我们可以看到，每个推导的值都能与原始推导中的一个且仅一个对应起来（反之亦然）。换句话说，$\mathfrak{p}^{(0)}\{\mathrm{x}\}$ 与$\mathfrak{p}^{(1)}\{\mathrm{x}\}$之间存在保值双射。

在下面的程序中，我们已系统地将 b 移至每个推导的根部——留下1 在 b原来的位置上。

$$\mathfrak{p}^{(2)} = \begin{cases} 388 & \mathrm{x} \mathrel{+}= \mathrm{x'} * \mathrm{b}. \\ 389 & \mathrm{x'} \mathrel{+}= \mathrm{a} * \mathrm{x'}. \\ 390 & \mathrm{x'} \mathrel{+}= \underline{1}. \end{cases}$$

$$\mathfrak{p}^{(2)}\{\mathrm{x}\} = \left\{ \right.$$



$\left. \right\}$

注意， $\mathfrak{p}^{(2)}$ 定义了一个新项x’，其值为$[\![\mathrm{x'}]\!] = 1 + \mathrm{a} + \mathrm{a}^2 + \cdots = \mathrm{a}^*$。不过，$\mathfrak{p}^{(0)}[\![\mathrm{x'}]\!] = 0$ 因为x’ 未在 $\mathfrak{p}^{(0)}$中定义。为了保持语义，我们要求x’ 在 $\mathfrak{p}_0$中定义。由于名称 x’ 尚未被占用，

$\mathfrak{p}^{(0)}$ 可以安全地向 $\mathfrak{p}_0$ 添加以下（未使用的）规则，以定义x'项。

$$\mathfrak{p}^{(0')} = \begin{cases} \texttt{x += b.} \\ \texttt{x += a * x.} \\ \texttt{x' += a * x'.} \\ \texttt{x' += 1.} \end{cases}$$

将这些定义添加到 $\mathfrak{p}^{(0)} \not\equiv \mathfrak{p}^{(2)}$后，我们得到 $\mathfrak{p}^{(0')} \equiv \mathfrak{p}^{(2)}$。用新定义扩充程序，就是我们所说的定义引入变换*(§7.3)*。引入新的中间量，是我们用来为给定程序设计更高效版本的最有力方法之一。一般而言，只要变换引入了新定义，我们就会确保新命名的项不会与程序全集中的其他任何项冲突。此外，我们会将这些定义添加到全集中的所有程序里，使它们保持语义等价。

定义引入。我们按照 Tamaki 和 Sato（7.3）的做法处理定义引入（§1984）。当变换引入一个定义时，我们假定它是在原始程序中引入的，并将其传播到由该程序派生的所有程序中。按照这一方案，较早的程序会被追溯性地补入该定义，尽管它们在创建时实际上并没有这个定义。为程序全集中的所有程序维护一个全局符号表十分重要，这样可以确保变换不会引入名称与任何先前定义（包括输入和内置项）冲突的新定义。

等价关系。 顾名思义，语义等价（$\equiv$）是一种等价关系。因此，我们可以用无向图表示通过变换找到的程序，其中节点是程序，且

程序之间存在一条边 $\mathfrak{p}$ 和 $\mathfrak{p}'$ 当 ($\mathfrak{p} \equiv \mathfrak{p}'$)。然而，由于 $\equiv$ 具有自反性、对称性和传递性，因此不必显式表示所有边。相反，$\mathfrak{p}$ 和 $\mathfrak{p}'$ 等价，当且仅当它们之间存在一条（无向）路径。由于判断任意一对程序在语义上是否等价是不可判定的，[157] 等价程序图将是不完备的。换句话说，我们只能通过添加程序之间能够证明其等价的（有向）边来对其进行近似（例如，如果存在一个从 $\mathfrak{p}$ 到 $\mathfrak{p}'$的语义保持变换，我们就添加这条边）——这意味着，如果存在一条边，我们就能确定这些程序等价；如果没有边，那么这些程序可能等价。因此，通过闭包推导出的任何等价关系都是可靠的。

7.3 定义—引言变换

定义引入变换通过添加一组定义新项的规则来扩展现有程序。我们假设这些项的名称与现有项不同。随后，将这些新定义 graft 到程序中便成为 折叠和展开变换的任务。我们通常可以用新定义有效地替换原有定义。示例 27演示了如何将新定义 `startpath` 变换为自洽的-

[157]即使检验一个 Dyna 程序的等价关系且不含变量也是不可判定的。这是因为 Dyna 程序可以直接编码上下文无关文法（CFG）生成的语言。因此，判定两个 Dyna 程序是否等价，至少与判定两个上下文无关文法生成的语言是否相同一样困难，而后者是一个不可判定问题（Hopcroft 等人，2007）。

定义，并通过将其折叠进目标规则中，将其 graft 到主程序里——替换代价更高的路径关系。一般来说，发现替代定义可能需要“神来之笔”。因此，正如我们之前提到的，这类规则在程序变换文献的许多著作中被亲切地称为尤里卡定义（Pettorossi 和 Proietti，1994）。

提出合理的尤里卡定义最常见的技术是**元组化技术**（Pettorossi 和 Proietti，1994）。[158] 元组化具有很强的通用性；其参数化方式如下：给定一个非基项乘积 $z_1 \circ \cdots \circ z_N$和一个变量集合 $\{x_1、\ldots\ldots、x_L\}$（它们可以是 $\mathrm{vars}(z_1 \circ \cdots \circ z_N$的任意子集，也可以是新鲜变量），元组化技术所构造的新定义为

$$\mathtt{tmp}(x_1, \ldots, x_L) \oplus\!= z_1 \circ \cdots \circ z_N$$

其中，`tmp` 是一个新颖的函数符号；名称 `tmp` 及其参数x_1、……、x_L 的顺序均可任意选择。本章前面示例（23、24、27和28）中提出的新定义，都是一种称为变量消除技术（见下文）的元组化技术实例。[159]

变量消除技术是元组化技术的一个实例，其中（1）我们使用程序中已有的子乘积；（2）我们选择

[158]Cohen、Simmons 等人（2011年）结合使用元组化、折叠和展开，提出了一种通用方法，用于推导计算逻辑程序加权交集的程序。他们利用这一方法，将若干复杂程序推导为较简单程序的乘积（例如，CKY 是加权上下文无关文法与加权有限状态自动机的交集）。[159]这种技术有时称为循环吸收（Pettorossi 和 Proietti，1994）。在自然语言处理中，这种变量消除形式称为钩子技巧（Eisner和Satta，1999a；Huang、Zhang 等人，2005年；Gildea，2011年）。

所选子乘积中的最小变量子集，使新定义能够正确替代该子乘积。更正式地说，对于给定的程序 $\mathfrak{p}$，变量消除技术会接收一条规则 $R \in \mathfrak{p}$。假设 R 的形式为 $x \oplus\!= \alpha \circ \mu \circ \beta$，其中 α、μ 和 β 是子目标序列。如果是交换的，那么匹配此模式时可以对 R 的子目标重新排序。如果 μ 的任何变量子集都没有出现在 x、α 和 β 中的任何一个里，那么这些变量可以在新定义中局部求和 —— 这样就能得到一条更高效的规则，因为它包含的变量更少。变量集合 $\{X1, \ldots, X_L\} = \text{vars}(\mu) \setminus (\text{vars}(\alpha) \cup \text{vars}(\beta) \cup \text{vars}(x))$ 是新定义要成功折叠到 $\mathfrak{p}$ 中此位置所需的最小变量集合。该技术的其余部分遵循元组化。

我们已经在§7.1中见过不少变量消除的例子。尤其是，示例23–25、27 和 28 都使用变量消除来提出能够实现加速的新定义。下面重点介绍其中一个例子。

示例 31. 我们重新来看示例 *24*，看看它如何使用变量消除。

回想一下，示例 *24*加速了这条规则

```
395  β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K).
```

方法是将其转换为两条规则，分别对变量 Y 和 K 求和：

```
396  β(X,I,K)  +=临时项(I,J,X,Z) *  β(Z,J,K).
397 临时项(I,J,X,Z)  +=  γ(X,Y,Z) *  β(Y,I,J).
```

我们将展示规则 #*397* 是变量消除的输出。[160]

[160]下面我们在体中补上一个 1，以匹配_($x \oplus \alpha \circ \mu \circ \beta$) 模式。

$$\underbrace{\beta(X,I,K)}_{x} \mathrel{+}= \underbrace{1}_{\alpha} * \underbrace{\gamma(X,Y,Z) * \beta(Y,I,J)}_{\mu} * \underbrace{\beta(Z,J,K)}_{\beta}$$

We 将我们的新关系命名为临时项，并按如下方式确定其参数 ：

$$\{I,J,X,Z\} = \mathrm{vars}(\gamma(X,Y,Z) * \beta(Y,I,J)) \setminus (\mathrm{vars}(1) \cup \mathrm{vars}(\beta(Z,J,K)) \cup \mathrm{vars}(\beta(X,I,K)))$$

现在我们可以看到#*397* 是变量消除技术的输出。请注意，随后我们必须用新规则折叠#*395* ，才能得到#*396* ，从而完成示例。每次定义引入后都应进行折叠，将新定义并入程序。

稍后我们将看到，推测变换和左角变换（§7.5）也是基于现有程序生成新定义的方法。

7.4 折叠和展开变换

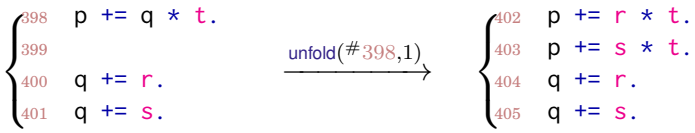
本节介绍一对称为折叠和展开的变换。这对变换在非加权逻辑编程中有着悠久的历史[161]。它们后来被推广到半环加权逻辑编程中（Eisner and Blatz，2007）。从较高层次来看，这些变换类似于沿展开（展开）或因子分解（折叠）方向应用半环分配律（定义17）。折叠变换对于将新定义嫁接到程序现有规则中，并改写这些定义，使其成为高效的自支持递归定义至关重要（如我们在§7.1中所见）。我们先介绍这些变换的基本版本（§7.4.1），然后再介绍广义版本（§7.4.2）。广义变换在加权情形下是新颖的。广义变换本身与无权设置中的变换大体相同。不过，为了证明这些变换能够保持我们的推导求和语义（§4），我们不得不彻底修改正确性证明——无权设置下的现有理论只考虑至少保留一个推导，而非保留所有推导（定义31）。正如无权设置中所指出的，广义折叠与展开变换功能相当强大，但需要谨慎处理技术细节，以免出错。

[161]例如，Tamaki 和 Sato（1984）、Kawamura 和 Kanamori（1990）、Bossi 等人（1992年）、Gergatsoulis 和 Katzouraki（1994）、Pettorossi 和 Proietti（1994）、Roychoudhury 等人（1999）。

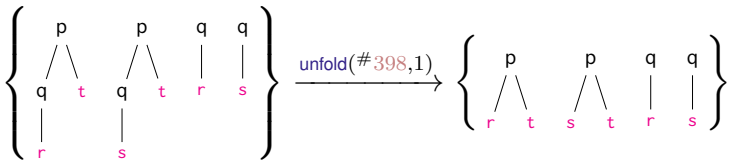
7.4.1 基本折叠与展开

展开。 从较高层次来看，展开变换 $\mathfrak{p}^{(s)} \xrightarrow{\text{展开}} \mathfrak{p}^{(t)}$所做的是，将给定规则 $r \in \mathfrak{p}^{(s)}$ 替换为该规则的多个版本 $\mathcal{S}$ 这些版本对应于根据定义特定子目标值的规则，展开该子目标。折叠的结果是一个新程序 $\mathfrak{p}^{(t)} = \mathfrak{p}^{(s)} \setminus \{r\} \cup \mathcal{S}$。

示例 32。 下面，我们展开q中的第一个子目标（$^{\#}398$）：



由于有两条规则 $^{\#}400$ 和 $^{\#}401$ 定义了q，因此在新程序中，我们得到$^{\#}398$ 的两个变体（$^{\#}402$ 和$^{\#}403$）。在此示例中，$r = {}^{\#}398$ 以及$\mathcal{S} = \{{}^{\#}402, {}^{\#}403\}$ 下面，我们展示p 和q的推导。



请注意，在第一个推导中，$^{\#}398$ 被替换为$^{\#}402$；在第一个推导中，$^{\#}398$ 被替换为$^{\#}403$。q 的两个推导保持不变，因为其中出现的规则均未被修改。

折叠。 折叠是展开的逆操作。从高层次来看，折叠变换 $\mathfrak{p}^{(s)} \xrightarrow{\text{折叠}} \mathfrak{p}^{(t)}$ 会将一组规则 $\mathcal{S} \subseteq \mathfrak{p}^{(s)}$ 替换为一条规则 r。换句话说，折叠的结果是一个新的程序 $\mathfrak{p}^{(t)} = \mathfrak{p}^{(s)} \setminus \mathcal{S} \cup \{r\}$。

示例 33. 延续示例 *32*，我们使用 折叠 来反转 展开 变换 -

变换，来自示例32。我们同时将 #406 通过 #408折叠，并将 #407 通过 #409折叠。

```
406  p += r * t.          410  p += q * t.
407  p += s * t.   fold   411
408  q += r.      ⟶       412  q += r.
409  q += s.              413  q += s.
```

这会将规则集合 $\mathcal{S} = \{^{\#}406, ^{\#}407\}$ 替换为单条规则 $r = {}^{\#}410$。要检查折叠是否正确，请验证展开 #410 后得到左侧的程序（这称为可逆性测试）。

定义 33。 基本展开变换 接受 $\langle \mathfrak{p}^{(s)}、\ r、\ j \rangle$，其中 $\mathfrak{p}^{(s)}$ 是一个程序，r 是一条规则（$r \in \mathfrak{p}^{(s)}$），而 j 是 r 的体（$0 \le j \le |\,体(r)\,|$）中的一个索引。并生成程序 $\mathfrak{p}^{(t)}$，其生成过程如下：

算法 20 展开：应用基本展开变换（定义 33）。

定义 展开($\mathfrak{p}^{(s)}$, r, j)：$\left(x \oplus\!= y_{[:j)} \circ y_j \circ y_{(j:]}\right) \leftarrow r$

$\mathfrak{p}^{(t)} \leftarrow \mathfrak{p}^{(s)} \setminus \{r\}$ ▷ 移除规则 r 的新程序

▷ y_j r $\mathfrak{p}^{(s)}$。在规则中，使用其定义规则展开子目标，以添加新规则。

$\mathfrak{p}^{(t)}[x] \oplus\!= y_{[:j)} \circ \mathfrak{p}^{(s)}[y_j] \circ y_{(j:]}$

返回 $\mathfrak{p}^{(t)}$

令 $\mathcal{S}$ 表示由该变换添加的规则集合。存在一一对应$\overset{\sigma}{t \to s}$：$\mathcal{S} \to \mathfrak{p}^{(s)}$；每条规则在 $\mathcal{S}$ 中映射到 $\mathfrak{p}^{(s)}$ 中一条产生它的不同规则。该对齐的逆记为$\overset{\sigma}{s \to t}$：$\mathfrak{p}^{(s)} \to \mathcal{S}$。[162]

定义 **34.** 基本折叠变换取 $\langle \mathfrak{p}^{(s)}, \ \mathcal{S}, \ r \rangle$，其中 $\mathfrak{p}^{(s)}$ 是 s 一个程序 m，$\mathcal{S} \subseteq \mathfrak{p}^{(s)}$，以及 r 是一条新规则。并且会生成程序 $\mathfrak{p}^{(t)} =$

[162]以下行为继承自 $\mathfrak{p}[\cdot]$。若子目标 j 是常量或输入，则该展开为空操作。若子目标 j 是内置关系，则在其实例化充分时可以求值。回顾§3.1.2.3 可知，展开一条规则可能导致对自由变量求和。该情形在此得到处理，这得益于结果流机制，其建立于§3.1.2.1。

$(\mathfrak{p}^{(s)} \setminus \mathcal{S}) \cup \{r\}$。然而，当且仅当折叠通过可逆性

测试：$\exists j\colon\ \mathfrak{p}^{(s)} = \text{展开}(\mathfrak{p}^{(t)}, r, j)$。[163]

算法 21 折叠：应用基本折叠变换（定义34）。

定义 折叠$(\mathfrak{p}^{(s)}, \mathcal{S}, r)$:
  $\mathfrak{p}^{(t)} \leftarrow (\mathfrak{p}^{(s)} \setminus \mathcal{S}) \cup \{r\}$
  断言 $\exists j\colon\ \mathfrak{p}^{(s)} = \text{展开}(\mathfrak{p}^{(t)}, r, j)$ ▷ 可逆性测试
  返回 $\mathfrak{p}^{(t)}$

请注意，新规则 r 必须经过仔细合成，才能使这种变换方法有用并通过可逆性测试。[164] 可逆性测试最初由 Maher（1987 年）研究。

定理 4. 假设 $\mathfrak{p}^{(s)} \xrightarrow{\text{展开}} \mathfrak{p}^{(t)}$ 根据定义 *33*，则 $\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$。**定理 5.** 假设 $\mathfrak{p}^{(s)} \xrightarrow{\text{折叠}} \mathfrak{p}^{(t)}$ 根据定义 *34*，则 $\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$。

无权设置下的正确性证明可见于许多著作（例如 Maher（1987 年））。然而，加权情形（最早在 Eisner and Blatz（2007）中提出）缺乏针对折叠与展开的正确性的形式化证明；我们在§B.2中填补了文献中的这一空白。此外，我们的理论结果强化了 Eisner and Blatz（2007）的（非正式）正确性主张，即通过证明强等价关系之于折叠与展开，而非（弱）等价关系。

7.4.2 广义折叠与展开

在本节中，我们将推广基本展开变换（定义 33），允许用于展开子目标的规则来自（可能不同的）

[163]对于此测试，句法相等就足够了。[164]对于给定的 r，我们可以在 $\mathcal{S} \subseteq \mathfrak{p}^{(s)}$ 上搜索，查看这些折叠中是否有任何一个通过可逆性测试。

程序 $\mathfrak{p}^{(d)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(s)}$。[165] 我们称之为 $\mathfrak{p}^{(d)}$ 这些 **辅助定义**。正如我们将在示例 34中看到的，辅助定义为我们提供了额外的灵活性，因为我们可以有效地混合搭配给定程序的语义等价版本。

示例 34。 考虑展开几何级数 $\mathfrak{p}^{(0)}$ 得到 $\mathfrak{p}^{(1)}$:

$$\mathfrak{p}^{(0)} = \begin{cases} 414 & \texttt{x += b.} \\ 415 & \texttt{x += x * a.} \end{cases} \xrightarrow{\text{unfold}(\#415,1)} \mathfrak{p}^{(1)} = \begin{cases} 416 & \texttt{x += b.} \\ 417 & \texttt{x += b * a.} \\ 418 & \texttt{x += x * a * a.} \quad \texttt{\% two a's} \end{cases}$$

到目前为止，还 不错。如果我们再展开一次呢？令人沮丧的是，展开($\mathfrak{p}^{(1)}$, #418, 1) 得到

419 x +=b。420 x +=b * a。421 x += b * a * a。422 x += b * a * a * a。423 x += x * a * a * a。% 四个 a

这将递推展开了 *4* 层，而不是 *3* 层。[166] 使用广义展开，展开($\mathfrak{p}^{(1)}$, #418, 1, $\mathfrak{p}^{(0)}$) 和 $\mathfrak{p}^{(0)}$ 作为辅助定义，得到

424 x +=b。425 x += b *a。426 x += b *a * a。427 x += x *a * a * a。% 三个 a

将递推展开至 *3* 层。[167]

广义折叠。我们还考虑一种广义折叠变换，它放宽了基本折叠中的可逆性测试（定义34），允许在可逆性测试中针对辅助定义进行展开。具体来说，广义折叠变换将定义34 中的可逆性测试从 $\exists j:\ \mathfrak{p}^{(s)} = \text{展开}(\mathfrak{p}^{(t)},\ r,\ j,\ \mathfrak{p}^{(t)})$ 改为 $\exists j:\ \mathfrak{p}^{(s)} = \text{展开}(\mathfrak{p}^{(t)},\ r,\ j,\ \mathfrak{p}^{(d)})$。这种额外的灵活性增强了

[165]未来工作可以探索强等价之外的其他等价关系，因为强等价显然并非正确性所必需的。[166]再次展开底部规则会将递推展开至 8 层。[167]再次展开底部规则（针对 $\mathfrak{p}^{(0)}$）会将递推展开至 4 层。

我们的变换系统的能力：示例27 和示例28 仅使用基本折叠将无法实现。我们将在27下面重新讨论示例。

示例 35。 回顾示例27，该示例计算图中所有路径的总权重（目标）：这些路径从集合 start 起始，并在集合 stop中结束。假设我们从 $\mathfrak{p}^{(0)}$ 开始，并将其展开为 $\mathfrak{p}^{(1)}$:

$\mathfrak{p}^{(0)} =$

```
428
429
430
431 目标 += startpath(K) * stop(K)。startpath(K) += start(I) *
```

path(I,K)。path(I,I) += 1。path(I,K) += path(I,J) * edge(J,K)。

$\xrightarrow{\text{unfold}(\#429,2)}$ $\mathfrak{p}^{(1)} =$

```
432 goal += startpath(K) * stop(K).
433 startpath(K) += start(K).
434 startpath(K) += start(I) * path(I,J) * edge(J,K).
435 path(I,I) += 1.
436 path(I,K) += path(I,J) * edge(J,K).
```

接下来，我们使用广义折叠，将 startpath 更高效地 *rewrite* 为用其自身表示，而不是用 path表示。我们注意到 $\mathcal{S} = \{\#434\}$ 可以替换为

$$r = (\texttt{startpath(K)} \mathrel{+}= \texttt{startpath(J)} * \texttt{edge(J,K)})$$

此变换将 $\#434$ 通过 $\#429$进行折叠。由于 $\#429$ 在 $\mathfrak{p}^{(0)}$里面，因此该变换是广义折叠。具体而言，变换 $\text{fold}^{0}_{\{\#434\}\to r}$ 给出

$\mathfrak{p}^{(2)} =$

```
437
438
439
440
441 目标 += startpath(K) * stop(K)。 startpath(K) += start(K)。
```

startpath(K) += ~~startpath(J) * edge(J,K)~~。 path(I,I) += 1。 path(I,K) += path(I,J) * edge(J,K)。

我们停用路径规则，因为它们不再被目标使用。其运行时间为$\mathfrak{p}^{(2)}$，即 $\mathcal{O}(N^2)$，其中 N 表示图中的节点数，这明显优于初始程序的 $\mathcal{O}(N^3)$ 运行时间 $\mathfrak{p}^{(0)}$。

警告! 广义的折叠和展开变换要保持语义，需要满足一些技术限制。上面的定义仅足以——

保证**部分正确性**——这意味着我们可以丢失推导，但不能产生虚假推导。举个简单的例子，请看示例35。如果我们不展开#429，而是直接将#429 本身折叠，就会得到非良基定义：`startpath(K) += startpath(K)`。本节余下部分将给出充分条件，以确保特定的折叠和展开变换能够保持语义。

7.4.2.1 形式化定义

广义展开。 下面，我们将基本展开变换（定义 33）推广为允许使用辅助定义（$\mathfrak{p}^{(d)}$，见下文）。[168]

定义 35。 广义展开变换适用于 $\langle \mathfrak{p}^{(s)}、r、j、\mathfrak{p}^{(d)} \rangle$，其中 $\mathfrak{p}^{(s)}$ 是一个程序，r 是一条规则（$r \in \mathfrak{p}^{(s)}$），j 是 r 的体中的一个索引（$0 \leq j \trianglelefteq |体(r)|$），而 $\mathfrak{p}^{(d)}$ 是一个程序（$\mathfrak{p}^{(d)}$ 强≡ $\mathfrak{p}^{(s)}$）。它生成程序 $\mathfrak{p}^{(t)}$，其定义如下。

算法22 展开：应用广义展开变换（定义 35）。

---

1. 该伪代码与基本展开（算法 *20*）类似。区别在于，我们引入了一个额外参数 $\mathfrak{p}^{(d)}$，并在 $\mathfrak{p}^{(d)}[y_j]$ 第 *22.6* 行上使用 $\mathfrak{p}^{(s)}[y_j]$，而不是。2. **定义** 展开$(\mathfrak{p}^{(s)}, r, j, \mathfrak{p}^{(d)})$：3.
$\left(x \oplus\!\!= y_{[:j)} \circ y_j \circ y_{(j:]}\right) \leftarrow r$ 4. $\mathfrak{p}^{(t)} \leftarrow \mathfrak{p}^{(s)} \setminus \{r\}$ ▷ 移除规则 r
后得到的新程序5 ▷ y_j r $\mathfrak{p}^{(d)}$。
添加新规则：对于规则中位于 $\mathfrak{p}^{(t)}[x] \overset{\circ}{\oplus\!\!=} y_{[:j)} \overset{\circ}{} \mathfrak{p}^{(d)}[y_j]$ $y_{(j:]}$[6][7] **返回** $\mathfrak{p}^{(t)}$

---

[168] Pettorossi 和 Proietti（1994）提出了广义展开，但 Roychoudhury 等人（1999）没有。因此，Roychoudhury 等人（1999）的安全性框架中缺少这一变换；我们的框架以该框架为基础，因此我们将在 §7.4.2.4 中加入它。

令 $\mathcal{S}$ 表示该变换所添加的规则集合。存在一一对应 $\overset{\sigma}{t \to d}$： $\mathcal{S} \to \mathfrak{p}^{(s)}$； $\mathcal{S}$ 中的每条规则都映射到 $\mathfrak{p}^{(d)}$ 中一条由其产生的不同规则。该对齐的逆为 $\overset{\sigma}{d \to t}$： $\mathfrak{p}^{(d)} \to \mathcal{S}$。

与基本展开（定义 33）不同，并非所有广义展开都是安全的（参见示例 36）；定理 1和 2给出了充分条件。注意，当 $\mathfrak{p}^{(d)} = \mathfrak{p}^{(s)}$时，我们可以得到基本展开。我们通常将 $\mathfrak{p}^{(s)} \xrightarrow[\text{unfold}]{\substack{d\ j \\ r \to \mathcal{S}}} \mathfrak{p}^{(t)}$ 写作 $\mathfrak{p}^{(s)}$ 与 $\mathfrak{p}^{(t)}$之间特定展开关系的描述。这提供了一种便捷的简写方式，用于引用 $\mathcal{S}$ 和 r，它们是元组$\langle \mathfrak{p}^{(s)}$、$r$、$j$、$\mathfrak{p}^{(d)} \rangle$的函数。当省略定义索引 d时，例如 $\mathfrak{p}^{(s)} \xrightarrow[\text{unfold}]{\substack{j \\ r \to \mathcal{S}}} \mathfrak{p}^{(t)}$，我们将$d = s$ 作为辅助定义参数的默认值。

广义折叠。下面，我们将基本折叠变换（定义 *34*）推广为允许在其可逆性测试中使用辅助定义。定义 *36*。广义折叠变换的工作方式如下：给定$\langle \mathfrak{p}^{(s)}$、$r$、$\mathcal{S}$、$\mathfrak{p}^{(d)} \rangle$，其中 r 是一个将规则子集 $\mathcal{S} \subseteq \mathfrak{p}^{(s)}$ 替换掉的规则，而$\mathfrak{p}^{(d)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(s)}$ 是辅助定义。结果是一个程序 $\mathfrak{p}^{(t)} =$ $(\mathfrak{p}^{(s)} \setminus \mathcal{S}) \cup \{r\}$。但是， $\mathfrak{p}^{(t)}$ 必须满足广义可逆性测试： $\exists j$： $\mathfrak{p}^{(s)}$ =展开 $(\mathfrak{p}^{(t)}$、r、j、$\mathfrak{p}^{(d)})$ 。

算法 23 折叠：应用广义折叠变换（定义36）。

定义 折叠$(\mathfrak{p}^{(s)}, r, \mathcal{S}, \mathfrak{p}^{(d)})$:
  $\mathfrak{p}^{(t)} \leftarrow (\mathfrak{p}^{(s)} \setminus \mathcal{S}) \cup \{r\}$ ▷ 将规则 $\mathcal{S}$ 替换为规则 r
  断言 $\exists j$： $\mathfrak{p}^{(s)} =$展开$(\mathfrak{p}^{(t)}, r, j, \mathfrak{p}^{(d)})$ ▷ 广义可逆性测试
  返回 $\mathfrak{p}^{(t)}$

不同于基本折叠（定义34），并非所有广义折叠都是安全的（参见示例 25）；定理 1 和 3给出了充分条件。类似于

对于展开的符号表示，我们通常写作 $\mathfrak{p}^{(s)} \overset{\text{折叠}}{\underset{\mathcal{S}\to r}{\overset{d\ j}{\longrightarrow}}} \mathfrak{p}^{(t)}$。注意，取$\mathfrak{p}^{(d)} = \mathfrak{p}^{(s)}$并不会恢复基本折叠。具体而言，广义可逆性条件将不会相同，因为基本折叠将相对于 $\mathfrak{p}^{(t)}$展开，而广义折叠将相对于 $\mathfrak{p}^{(d)}$展开，后者等于$\mathfrak{p}^{(s)}$，而不是 $\mathfrak{p}^{(t)}$。

7.4.2.2 挑战

不幸的是，并非所有广义展开和折叠都是语义保持的。我们在本节中分别给出示例。

示例 36（不安全的广义展开）。我们起始于

$$\mathfrak{p}^{(s)} = \begin{cases} 442 & \mathtt{p\ +\!\!=\ q.} \\ 443 & \mathtt{q\ +\!\!=\ r.} \\ 444 & \mathtt{r\ +\!\!=\ 1.} \end{cases} \qquad \mathfrak{p}^{(d)} = \begin{cases} 445 & \mathtt{p\ +\!\!=\ 1.} \\ 446 & \mathtt{q\ +\!\!=\ r.} \\ 447 & \mathtt{r\ +\!\!=\ p.} \end{cases}$$

显然，$\mathfrak{p}^{(s)} \equiv \mathfrak{p}^{(t)}$ 因为它们都具有 $[\![\mathtt{p}]\!] = [\![\mathtt{q}]\!] = [\![\mathtt{r}]\!] = 1$。此外，$\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$因为我们可以看到其间存在保值双射

$$\mathfrak{p}^{(s)}\{\cdot\} = \left\{ \begin{array}{ccc} \mathtt{p} & \mathtt{q} & \mathtt{r} \\ | & | & | \\ \mathtt{q} & \mathtt{r} & 1 \\ | & | & \\ \mathtt{r} & 1 & \\ | & & \\ 1 & & \end{array} \right\} \qquad \textit{and} \qquad \mathfrak{p}^{(d)}\{\cdot\} = \left\{ \begin{array}{ccc} \mathtt{p} & \mathtt{q} & \mathtt{r} \\ | & | & | \\ 1 & \mathtt{r} & \mathtt{p} \\ & | & | \\ & \mathtt{p} & 1 \\ & | & \\ & 1 & \end{array} \right\}$$

然而，展开 $^{\#}\mathit{443}$ 相对于 $\mathfrak{p}^{(d)}$ 会得到以下不正确的程序：

$$\mathfrak{p}^{(t)} = \begin{cases} 448 & \mathtt{p\ +\!\!=\ q.} \\ 449 & \mathtt{q\ +\!\!=\ p.} \\ 450 & \mathtt{r\ +\!\!=\ 1.} \end{cases} \qquad \mathfrak{p}^{(t)}[\![\cdot]\!] = \begin{cases} 451 & \mathtt{p\ +\!\!=\ 0.} \\ 452 & \mathtt{q\ +\!\!=\ 0.} \\ 453 & \mathtt{r\ +\!\!=\ 1.} \end{cases} \qquad \mathfrak{p}^{(t)}\{\cdot\} = \left\{ \begin{array}{c} \mathtt{r} \\ | \\ 1 \end{array} \right\}$$

检查这些规则后，我们发现 $^{\#}\mathit{448}$ 与$_{\#}\mathit{449}$ 之间存在一个非良基循环，导致其赋值发生变化，推导集的规模也随之缩小。

示例 37 （不安全的广义折叠）。我们从[169]

$$\mathfrak{p}^{(0)} = \begin{cases} {\scriptstyle 454} & \texttt{p += q.} \\ {\scriptstyle 455} & \texttt{q += 1.} \end{cases} \qquad \mathfrak{p}^{(0)}\{\cdot\} = \left\{ \begin{array}{cc} \texttt{q} & \texttt{p} \\ | & | \\ 1 & \texttt{q} \\ & | \\ & 1 \end{array} \right\}$$

由于 $[\![\texttt{p}]\!] = [\![\texttt{q}]\!] = 1$，我们可能会尝试将右侧的所有 q 替换为 p。因此，我们通过折叠 $\mathfrak{p}^{(*)}$ 来定义以下新程序$\mathcal{S} = \{^{\#}454\}$，$r = (\texttt{p += p})$ $j = 0$，以及定义 $\mathfrak{p}^{(d)} = \mathfrak{p}^{(0)}$。输出如下（部分正确的）程序。

$$\mathfrak{p}^{(*)} = \begin{cases} {\scriptstyle 456} & \texttt{p += p.} \\ {\scriptstyle 457} & \texttt{q += 1.} \end{cases} \qquad \mathfrak{p}^{(*)}\{\cdot\} = \left\{ \begin{array}{c} \texttt{q} \\ | \\ 1 \end{array} \right\}$$

尽管广义可逆性条件得到满足（展开$(\mathfrak{p}^{(0)}, ^{\#}456,\ 1,\ \mathfrak{p}^{(0)}) = \mathfrak{p}^{(0)}$），但此变换丢失了p的推导。从较高层次来看，发生的情况是我们为p创建了一个非良基定义；这个有缺陷的定义是有环的，没有任何基例。

7.4.2.3 什么导致部分正确性?

为了更好地理解折叠和展开变换，我们将构造它们的推导映射。回顾定义31，推导映射是一个函数 ϕ，它将来自 $\mathfrak{p}^{(s)}\{\cdot\}$ 的推导映射为来自$_{s\to t}\mathfrak{p}^{(t)}\{\cdot\}$的推导。要使变换$_{s\to t}$ $\mathfrak{p}^{(s)} \longrightarrow \mathfrak{p}^{(t)}$ 强语义保持，我们要求 ϕ 是一个保值双射。广义折叠/展开中有趣之处在于，辅助定义在构造过程中所起的作用

[169]这是对 Pettorossi 和 Proietti（1994）中例4.1的 Dyna 改编 .

这些映射。粗略地说，推导映射 ϕ 将通过枢轴变换$_{s\to t}$经由 $\mathfrak{p}^{(d)}\{\cdot\}$，使用 ϕ 和 ϕ。枢轴变换的挑战在于，它可能导致$_{s\to d}$ $_{d\to s}$无限递归，从而（可能）丢失一条推导（即，部分正确性）。我们将利用从推导映射的失败模式中获得的认识，推导出变换何时安全的充分条件（也就是说，映射何时是保值双射）。

请读者参阅算法24 和算法25，其中给出了广义展开和折叠变换的推导映射伪代码。我们想强调两点：（1）推导映射绝不会作为变换的一部分运行——它们仅用于分析这些变换；（2）算法24 和算法25 都不保证定义了保值双射；事实上，它们甚至不保证是函数，因为对于某些输入，它们不会终止。

为了了解这些变换为何出错，我们将研究推导映射如何在以下示例中陷入困境：

示例 38。继续看不安全折叠示例（示例*37*）。回想一下 ：

$$r = (\mathsf{p}\ \texttt{+=}\ \mathsf{p}) \qquad \mathfrak{p}^{(s)}\{\cdot\} = \left\{ \begin{array}{cc} \mathsf{q} & \mathsf{p} \\ | & | \\ 1 & \mathsf{q} \\ & | \\ & 1 \end{array} \right\} \qquad \boldsymbol{\delta}^{(s)} = \left(\begin{array}{c} \mathsf{p} \\ | \\ \mathsf{q} \\ | \\ 1 \end{array} \right)$$
$$\mathcal{S} = \{\mathsf{p}\ \texttt{+=}\ \mathsf{q}\}$$
$$\mathfrak{p}^{(d)} = \mathfrak{p}^{(s)}$$

而且我们看见，推导 $\boldsymbol{\delta}^{(s)}$ 在这次变换中丢失了。考虑通过算法 $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$ 对 *25*进行求值。由于 $\boldsymbol{\delta}^{(s)}$ 的顶层规则是

$\mathbf{S}^{(\ell)} = (\mathsf{p} \ \mathsf{+=}\ \mathsf{q})$和 $\mathbf{S}^{(\ell)} \in \mathcal{S}$，我们落入底部情况（第 *25.7* 行）。

$$\phi_{s\to t}\left(\begin{array}{c}\mathsf{p}\\|\\\mathsf{q}\\|\\1\end{array}\right) = \begin{array}{c}\mathsf{p}\\|\\\phi_{s\to t}\left(u^{-1}\left(\begin{array}{c}\mathsf{q}\\|\\1\end{array}\right)\right)\end{array}$$

对 u^{-1} 的调用返回[170]

$$\left(\begin{array}{c}\mathsf{p}\\|\\\mathsf{q}\\|\\1\end{array}\right) = u^{-1}\left(\begin{array}{c}\mathsf{q}\\|\\1\end{array}\right)$$

现在，当我们回到第 *25.7* 行时，我们发现了这个问题：

$$\phi_{s\to t}\left(\begin{array}{c}\mathsf{p}\\|\\\mathsf{q}\\|\\1\end{array}\right) = \begin{array}{c}\mathsf{p}\\|\\\phi_{s\to t}\left(u^{-1}\left(\begin{array}{c}\mathsf{q}\\|\\1\end{array}\right)\right)\end{array} = \begin{array}{c}\mathsf{p}\\|\\\phi_{s\to t}\left(\begin{array}{c}\mathsf{p}\\|\\\mathsf{q}\\|\\1\end{array}\right)\end{array} = \begin{array}{c}\mathsf{p}\\|\\\mathsf{p}\\|\\\mathsf{p}\\|\\\mathsf{p}\\|\\\vdots\end{array}$$

这个递推永不终止，因为 ϕ 会立即以相同的$_{s\to t}$参数调用自身。它试图构造一棵无限高的树，却永远无法成功，因为它不会命中基例。因此，推导 $\boldsymbol{\delta}^{(s)}$ 会在推导映射中丢失——这意味着我们用于 ϕ 的过程无法找到任何对应的 $\boldsymbol{\delta}^{(t)}{}_{s\to t}$在 $\mathfrak{p}^{(t)}\{\cdot\}$里面。这意味着我们未能定义一个保值双射（ ϕ 甚至不是一个函数）。因此，我们无法保证这个广义折叠实例$_{s\to t}$的安全性，也不应该保证！

[170]要了解原因，我们来追踪 u^{-1}的步骤。由于 $\mathfrak{p}^{(d)} = \mathfrak{p}^{(s)}$ ϕ（以及 ϕ）变换都是$_{s\to t}$ $_{t\to s}$空操作。折叠规则 $\mathbf{D}^{(\ell)} =$ （p $\mathsf{+=}$ q） （#454）。因此，当我们重构时，得到 $y_j = \mathsf{p}$。我们还得到以下对齐关系：$\overset{\sigma}{}_{d\to t}(\mathbf{D}^{(\ell)}) =$ （p $\mathsf{+=}$ q）。

从这个例子中得到的教训是，要使推导映射（算法24 和算法25）正确，我们需要确保不会发生循环递归——就像我们刚才在例38中看见的那样。我们将把ϕ的不终止作为诊断广义折叠/展开是否$_{s\to t}$出错的依据。当 ϕ （$\boldsymbol{\delta}$）（算法24和算法25）在某个$_{s\to t}\boldsymbol{\delta} \in \mathfrak{p}^{(s)}\{\cdot\}$上未能终止时，有问题的 $\boldsymbol{\delta}$ 对应于一个可能会丢失的推导。因此，为了确保正确性，我们需要拒绝那些 ϕ 可能无法$_{s\to t}$始终终止的变换。更正式地说，我们有以下定理，证明见第B.2节：

定理 2. 给定 $\mathfrak{p}^{(s)}$ 展开 $\xrightarrow{\substack{d\ j\\ r\to\mathcal{S}}}$ $\mathfrak{p}^{(t)}$，以及 $\mathfrak{p}^{(s)}$ 强$\equiv$ $\mathfrak{p}^{(d)}$。假设函数 $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$，如算法 *24* 所规定，对所有 $\boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$终止。则，$\underset{s\to t}{\phi}$ 是 $\mathfrak{p}^{(s)}\{\cdot\}$ 与 $\mathfrak{p}^{(t)}\{\cdot\}$之间的保值双射。此外，$\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$。

定理 3. 给定 $\mathfrak{p}^{(s)}$ 折叠 $\xrightarrow{\substack{d\ j\\ \mathcal{S}\to r}}$ $\mathfrak{p}^{(t)}$，以及 $\mathfrak{p}^{(s)}$ 强$\equiv$ $\mathfrak{p}^{(d)}$。假设函数 $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$，如算法 *25* 所规定，对所有 $\boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$终止。则，$\phi$ 是保值$_{s\to t}$双射（$\mathfrak{p}^{(s)}\{\cdot\}$ 与 $\mathfrak{p}^{(t)}\{\cdot\}$之间）。此外，$\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$。

终止条件体现在我们的正确性证明（§B.2）中；终止确保存在一个良基序，用于我们建立保值双射性的归纳证明。验证这些推导映射是否总能终止可能很困难。我们在 §7.4.2.4 中介绍了一种基于静态分析、用于确保终止的相对简单的系统。

算法 24 推导映射，用于 $\mathfrak{p}^{(s)}$ 展开 $\xrightarrow[r\to\mathcal{S}]{d\ j}$ $\mathfrak{p}^{(t)}$。

1. ▷伪代码假定各规则互不相同。可以通过为每条规则添加一个标识符，并使用相等性测试检查该标识符来实现这一点。将规则从一个程序复制到另一个程序时，其标识符保持不变。创建规则时，则为其分配一个唯一标识符。

2. ▷我们使用推导映射 ϕ 和 ϕ，以建立 $\mathfrak{p}^{(s)}_{s\to d}$ 强等 $\mathfrak{p}^{(d)}$。对于所有，它们必须是固定的。

3. ▷ α 。在下面的代码中，μ,和 β 都是序列。

4. **定义** $\phi_{s\to t}\left(\boldsymbol{\delta}^{(s)}\right)$:

5. **情形：常量或内置项：返回** $\boldsymbol{\delta}^{(s)}$

6. **case** $y_1 \circ\cdots\circ y_K$ **：返回** $\phi_{s\to t}\left(y_1\right)\circ\cdots\circ\phi_{s\to t}\left(y_K\right)$

7. **case** $\left(\begin{array}{c} x \\ | \\ \mu \end{array}\right)$ **以及** 其顶层规则 $\in \mathfrak{p}^{(t)}$: **返回** $\begin{array}{c} x \\ | \\ \phi_{s\to t}\left(\mu\right) \end{array}$

8. **case** $\left(\begin{array}{c} x \\ \alpha\ \ y_j\ \ \beta \end{array}\right)$ **以及**其顶层规则 $\notin \mathfrak{p}^{(t)}$:

9. **返回** $\begin{array}{c} x \\ \phi_{s\to t}\left(\alpha\right)\ \ \phi_{s\to t}\left(u\left(y_j\right)\right)\ \ \phi_{s\to t}\left(\beta\right) \end{array}$ ▷变换后的推导的顶层规则由 2nd 从 u返回的有序对中的元素给出（符号表示滥用）。对 u 的调用会将 y_j 子树替换为 $\mathfrak{p}^{(s)}\{\cdot\}$ 中的值等价推导（ 1st 有序对中的元素），随后对其进行递归变换。

10. ▷ 枢轴函数：返回一个值等价推导和其中新的顶层规则组成的有序对 $\mathfrak{p}^{(t)}$。参见引理 *2*，了解正式陈述和证明。

11. **定义** $u\left(y_j\right)$: ▷ 注意：此函数引用了调用上下文中的 $\langle s、\ t、\ d\rangle$ 。

12. $\boldsymbol{\delta}_1^{(s)} \leftarrow y_j$; $\boldsymbol{\delta}_1^{(d)} \leftarrow \phi_{s\to d}\left(\boldsymbol{\delta}_1^{(s)}\right)$ ▷ 将子树转换为 $\mathfrak{p}^{(d)}\{\cdot\}$。

13. $\left(\begin{array}{c} y_j \\ | \\ \mu \end{array}\right) \leftarrow \boldsymbol{\delta}_1^{(d)}$; $\boldsymbol{\delta}_2^{(d)} \leftarrow \mu$ ▷ 移除中间项y_j

14. $\mathbf{D}^{(\ell)} \leftarrow$ 的顶层规则 $\boldsymbol{\delta}_1^{(d)}$

15. $\mathbf{T}^{(\ell)} \leftarrow \sigma_{d\to t}\left(\mathbf{D}^{(\ell)}\right)$ ▷ 使用对齐来检索新的顶层规则

16. $\boldsymbol{\delta}_2^{(s)} \leftarrow \phi_{d\to s}\left(\boldsymbol{\delta}_2^{(d)}\right)$ ▷ 转换回 $\mathfrak{p}^{(s)}\{\cdot\}$

17. **返回** $\left\langle \boldsymbol{\delta}_2^{(s)},\ \mathbf{T}^{(\ell)} \right\rangle$

算法25 推导映射（针对 $\mathfrak{p}^{(s)}$ 折叠 $\xrightarrow{\substack{d\ j\\ \mathcal{S}\to r}}$ $\mathfrak{p}^{(t)}$ 。

▷算法 *24* 顶部的注释同样适用于此处。**定义** $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$:

情况：常量或内置项： 返回 $\boldsymbol{\delta}^{(s)}$

case $\overset{y_1}{\triangle}\circ\cdots\circ\overset{y_K}{\triangle}$: **返回** $\underset{s\to t}{\phi}\left(\overset{y_1}{\triangle}\right)\circ\cdots\circ\underset{s\to t}{\phi}\left(\overset{y_K}{\triangle}\right)$

case $\left(\begin{array}{c}\boldsymbol{x}\\ |\\ \mu\\ \triangle\end{array}\right)$ **及其顶层规则： 返回** $\begin{array}{c}\boldsymbol{x}\\ |\\ \underset{s\to t}{\phi}\left(\begin{array}{c}\mu\\ \triangle\end{array}\right)\end{array}$

case $\left(\begin{array}{c}\boldsymbol{x}\\ \alpha\ \ \mu\ \ \beta\\ \triangle\ \ \triangle\ \ \triangle\end{array}\right)$ **及其顶层规则：**

返回 $\begin{array}{c}\boldsymbol{x}\\ \underset{s\to t}{\phi}\left(\alpha\right)\quad \underset{s\to t}{\phi}\left(u^{-1}\left(\mu\right)\right)\quad \underset{s\to t}{\phi}\left(\beta\right)\end{array}$

▷ 传入 u^{-1} 的规则是 $\mathbf{S}^{(\ell)}$ 的顶层规则$\boldsymbol{\delta}^{(s)}$（此处滥用符号表示），而 σ中使用的对齐 u^{-1} 来自折叠变换的广义可逆性测试。

▷ 枢轴函数 $\boldsymbol{u}$的逆。有关形式化保证，请参见引理 *3*。

定义 $u^{-1}\left(\boldsymbol{\delta}_2^{(s)},\mathbf{T}^{(\ell)}\right)$: ▷ 注意：谨致歉意，参数名称 $\mathbf{T}^{(\ell)}$ 是为了与 $\boldsymbol{u}$的参数名称保持一致而选定的。（当我们从折叠的推导映射调用它时，传入来自 $\mathfrak{p}^{(s)}$ 、变量名称为 $\mathbf{T}^{(\ell)}$ 的规则并非错误。）

$\boldsymbol{\delta}_2^{(d)} \leftarrow \underset{s\to d}{\phi}\left(\boldsymbol{\delta}_2^{(s)}\right)$

$\mathbf{D}^{(\ell)} \leftarrow \underset{t\to d}{\sigma}\left(\mathbf{T}^{(\ell)}\right)$

$\boldsymbol{y}_j \leftarrow$ 重构$\left(\mathbf{T}^{(\ell)},\ \mathbf{D}^{(\ell)}\right)$

$\boldsymbol{\delta}_1^{(d)} \leftarrow \left(\begin{array}{c}y_j\\ |\\ \mu\\ \triangle\end{array}\right)$ ▷ 所添加的规则是如下内容的实例化：$\mathbf{D}^{(\ell)}$

$\boldsymbol{\delta}_1^{(s)} \leftarrow \underset{d\to s}{\phi}\left(\boldsymbol{\delta}_1^{(d)}\right)$

return $\boldsymbol{\delta}_1^{(s)}$

▷ 重构本会被擦除的中间项，其擦除操作是 $\mathfrak{p}^{(s)}$ 展开 $\xrightarrow{\substack{d\ j\\ r\to\mathcal{S}}}$ $\mathfrak{p}^{(t)}$ 。

def reconstruct$\left(\mathbf{T}^{(\ell)},\mathbf{D}^{(\ell)}\right)$:

$\left(\boldsymbol{x}' \oplus\!\!= \boldsymbol{\alpha}'\circ\boldsymbol{y}'\circ\boldsymbol{\beta}'\right) \leftarrow r$ $\left(\boldsymbol{y}' \oplus\!\!= \boldsymbol{\mu}''\right) \leftarrow$

$\mathbf{D}^{(\ell)}$ $\boldsymbol{\theta} \leftarrow$ 覆盖$\left(\left(\boldsymbol{x}' \oplus\!\!= \boldsymbol{\alpha}'\circ\boldsymbol{\mu}''\circ\boldsymbol{\beta}'\right),\mathbf{T}^{(\ell)}\right)$

▷ *solve* $\mathbf{T}^{(\ell)} = \boldsymbol{\theta}\left(\left(\boldsymbol{x}' \oplus\!\!= \boldsymbol{\alpha}'\circ\boldsymbol{\mu}''\circ\boldsymbol{\beta}'\right)\right)$

返回 $\boldsymbol{\theta}(\boldsymbol{y}')$

7.4.2.4 确保正确性的实用充分条件

本节将介绍一种实用方法，用于检验我们在 §7.4.2.3 的定理中所确定的那些不切实际的充分条件。最终，我们将得到一个系统，能够保证 $\phi(\boldsymbol{\delta})$ 的所有推导 $\boldsymbol{\delta}$（其中 ϕ 要么是$_{s\to t}$ $_{s\to t}$算法 24 ，要么是算法 25）终止。我们将使用一种简单的抽象解释技术，确保 ϕ 中的每次递归调用都沿着某个$_{s\to t}$良基序下降，从而确保递归调用最终到达基本情况。

我们避免不安全的折叠和展开变换的策略改编自 Roychoudhury 等人（1999）。从整体上看，该策略会跟踪每条规则的来源（即每条规则如何通过先前的变换推导而来）。规则来源数据会为不同程序中的规则之间的折叠和展开操作带来额外约束，以防止循环定义环。这些仅仅是充分条件，但足够通用，能够覆盖大多数用例。值得强调的一个条件是$\mathfrak{p}^{(s)}$ 和 $\mathfrak{p}^{(d)}$ 必须通过一系列变换相关联；它们不能是互不相关的程序。

我们从一个初始程序开始 $\mathfrak{p}^{(0)}$。初始程序的每个推导都有一个有限大小，我们可以将其用作归纳法的良基序。通过变换从初始程序推导出新程序时，我们可以使用相对于相对大小 $\mathfrak{p}^{(0)}$ ，对后续程序的推导进行归纳。相对大小使我们能够比较不同程序中的推导（例如，在

u 和 u^{-1}中进行枢轴变换所产生的推导）。不过，我们需要确保每次折叠或展开之后，新程序中的相对大小度量仍然准确且良基。

建模决策。 为了使静态分析易于处理，我们作出以下建模决策：

1 对于任意广义折叠或展开，程序 $\mathfrak{p}^{(s)}$、 $\mathfrak{p}^{(d)}$和 $\mathfrak{p}^{(t)}$ 都可以通过一系列折叠或展开变换与某个初始程序 $\mathfrak{p}^{(0)}$相关联。

2 每次变换都会（近似地）保留每个推导的相对大小。我们将定义一个函数 Ψ，使得 $\Psi\left(\boldsymbol{\delta}^{(s)}\right) = \Psi\left(\boldsymbol{\delta}^{(t)}\right)$ 对于所有 $\left\langle \boldsymbol{\delta}^{(s)},\ \boldsymbol{\delta}^{(t)} \right\rangle$ 在 $\mathfrak{p}^{(s)}\{\cdot\}$ 与 $\mathfrak{p}^{(t)}\{\cdot\}$之间的双射中。

3 推导 $\boldsymbol{\delta} \in \mathfrak{p}\{\cdot\}$ 的相对大小是一个加性函数 Ψ，其参数是与程序规则相关联的值。

现在我们将3 说更精确一些；我们来定义推导的测度。

定义37。 假设给定的程序 $\mathfrak{p}$ 为每条规则赋予一个度量 $\psi(\mathbf{R})$，赋予对象为$\mathbf{R} \in \mathfrak{p}$。[171] 对于 $\boldsymbol{\delta} \in \mathfrak{p}\{\cdot\}$，其推导度量 $\Psi(\boldsymbol{\delta})$ 定义为

$$\Psi(\boldsymbol{\delta}) \stackrel{\text{def}}{=} \begin{cases} 0 & \textit{if } \boldsymbol{\delta} \textit{ is a constant or builtin} \\ \Psi(\boldsymbol{\delta}_1) + \cdots + \Psi(\boldsymbol{\delta}_K) & \textit{if } \boldsymbol{\delta} \textit{ is a product } \boldsymbol{\delta}_1 \circ \cdots \circ \boldsymbol{\delta}_K \\ \psi(\mathbf{R}) + \Psi(\mathsf{body}(\boldsymbol{\delta})) & \textit{if } \boldsymbol{\delta} \textit{ is a derivation with top rule } \mathbf{R} \end{cases} \tag{7.1}$$

我们将仔细选择规则度量参数（ 3 ），使得2 成立。在说下面的示之后，我们将给出精确的步骤。

[171] 为简便起见，我们假设规则度量取整数值。不过，也可以将其推广到某些良序交换群（ Roychoudhury 等人，1999，定义1）。

示例 39。 考虑下面这个展开，其中规则度量由以下内容提供

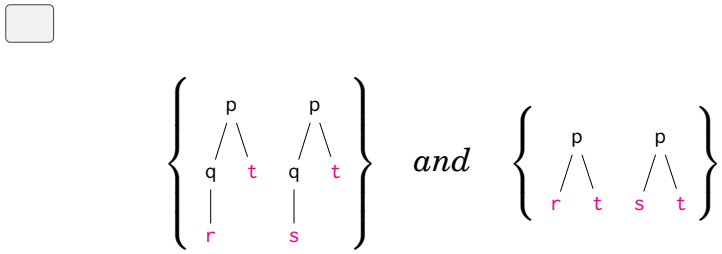
每条规则的左侧：

```
458  γ0: p += q * t.
459
460  γ1: q += r.
461  γ2: q += s.
```

unfold ⟶

```
462  γ0 + γ1: p += r * t.
463  γ0 + γ2: p += s * t.
464  γ1: q += r.
465  γ2: q += s.
```

我们将变量 γ_0、 γ_1 和 γ_2 分别与左侧程序中的 *3* 条规则相关联。右侧的规则度量经过选择，以保持推导度量（ *2* ）。考虑每个程序中p 的 *2* 个推导的推导：



我们验证推导度量得以保持：

$$\Psi\left(\begin{array}{c}\mathsf{p}\\ \mathsf{q}\quad\mathsf{t}\\ \mathsf{r}\end{array}\right) = \gamma_0 + \gamma_1 = \Psi\left(\begin{array}{c}\mathsf{p}\\ \mathsf{r}\quad\mathsf{t}\end{array}\right)$$

$$\Psi\left(\begin{array}{c}\mathsf{p}\\ \mathsf{q}\quad\mathsf{t}\\ \mathsf{s}\end{array}\right) = \gamma_0 + \gamma_1 = \Psi\left(\begin{array}{c}\mathsf{p}\\ \mathsf{s}\quad\mathsf{t}\end{array}\right)$$

直观地说，该方案之所以有效，是因为由 展开 产生的规则每次出现在右侧的推导中时，都各自计为两条规则。

现在我们考虑折叠，其作用方向相反。在这里，每条规则的度量都出现在其左侧。

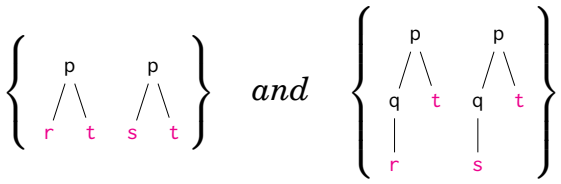
```
466  γ0: q += r.
467  γ1: q += s.
468  γ2: p += r * t.
469  γ3: p += s * t.
```

fold ⟶

```
470  γ0: q += r
471  γ1: q += s
472
473  {γ3 − γ1, γ2 − γ0}: p += q * t.
```

度量 的测度 #*473* 很有意思：我们不确定它是否应该 *b* *e*

$\gamma_3-\gamma_1$ 或 $\gamma_2-\gamma_0$。因此，我们用一个不确定性集合来表示新规则的测度：$\{\gamma_3-\gamma_1,\gamma_2-\gamma_0\}$。这么做是因为我们已经丢失了关于这两条规则中哪一条被折叠的信息。现在，我们可以针对双射中的每一对推导，将其大小保守地估计为一个不确定性集合



我们验证推导度量得以保留（即它们包含在不确定性集合里面）：

$$\Psi\left(\begin{array}{c}\mathrm{p}\\ \mathrm{r}\quad\mathrm{t}\end{array}\right)=\{\gamma_2\}\subseteq\{\gamma_3-\gamma_1+\gamma_0,\gamma_2\}=\{\gamma_3-\gamma_1,\gamma_2-\gamma_0\}+\{\gamma_0\}=\Psi\left(\begin{array}{c}\mathrm{p}\\ \mathrm{q}\quad\mathrm{t}\\ \mathrm{r}\end{array}\right)$$

$$\Psi\left(\begin{array}{c}\mathrm{p}\\ \mathrm{s}\quad\mathrm{t}\end{array}\right)=\{\gamma_3\}\subseteq\{\gamma_3,\gamma_2-\gamma_0+\gamma_1\}=\{\gamma_3-\gamma_1,\gamma_2-\gamma_0\}+\{\gamma_1\}=\Psi\left(\begin{array}{c}\mathrm{p}\\ \mathrm{q}\quad\mathrm{t}\\ \mathrm{s}\end{array}\right)$$

我们通过定义一个变换序列，使[1] 更加精确：

定义 *38.* 变换序列 $\mathfrak{p}^{(0)},\ldots,\mathfrak{p}^{(T)}$ 递归定义如下。基本情况（$T=0$）：允许 $\mathfrak{p}^{(0)}$ 是任意程序。我们称 $\mathfrak{p}^{(0)}$ 为参照程序。递归情形（$T\geq 1$）：假设 $\mathfrak{p}^{(0)},\ldots,\mathfrak{p}^{(T-1)}$ 是一个变换序列，那么只要满足以下条件，$\mathfrak{p}^{(0)},\ldots,\mathfrak{p}^{(T-1)},\mathfrak{p}^{(T)}$ 就是一个变换序列：$\mathfrak{p}^{(T)}=$ 折叠 $\overset{d\ j}{\mathcal{S}\to r}$，$(\mathfrak{p}^{(s)})$ 或 $\mathfrak{p}^{(T)}=$ 展开 $\overset{d\ j}{r\to\mathcal{S}}$，$(\mathfrak{p}^{(s)})$ 其中 $\mathfrak{p}^{(s)},\mathfrak{p}^{(d)}$ 是序列中较早的程序，新变换是良构的（$\langle\mathcal{S},j,r\rangle$ 是有效参数），且新变换是安全的。

定义 39 (规则度量更新)。 对于变换序列中的每个程序 $\mathfrak{p}^{(t)}$，我们为它的每条规则定义一个规则度量 $\psi_t(\mathbf{R})$，即每条规则$\mathbf{R} \in \mathfrak{p}^{(t)}$。规则度量的值通常是一个不确定性集合。规则度量以归纳方式定义。

基本情况：对于每条规则 $\mathbf{R} \in \mathfrak{p}^{(0)}$，我们将其测度 $\psi_0(\mathbf{R}) > 0$ 作为输入参数。[172]

递归情况：假设我们序列中的最后一个程序是 $\mathfrak{p}^{(t)}$。那么它必定由以下情形之一推导而来：

- 情形 $\mathfrak{p}_{(t)} = \text{展开}\,{}^{d\,j}_{r\to\mathcal{S}}\,(\mathfrak{p}_{(s)})$：此处，我们在每条新规则 $T_{(\ell)} \in \mathfrak{p}_{(t)}$ 与某个 $D_{(\ell)} \in \mathfrak{p}_{(d)}$ 之间存在一个对齐。该对齐为 $\{\langle D_{(\ell)},\ T_{(\ell)}\rangle\}_{\ell=1}^{L} =^{\sigma}_{d\to t}$。于是，每条新规则 $T_{(\ell)}$ 的规则度量则为

$$\psi_t\big(\mathbf{T}^{(\ell)}\big) = \psi_s(r) + \psi_d\big(\mathbf{D}^{(\ell)}\big) \quad \textit{for } \ell \in [1{:}L]$$

- 情形 $\mathfrak{p}_{(t)} = \text{折叠}\,{}^{d\,j}_{\mathcal{S}\to r}\,(\mathfrak{p}_{(s)})$：令 r 为通过折叠得到的新规则$\mathcal{S} = \{\mathbf{S}_{(\ell)}\}_{\ell=1}^{L}$，其中使用了定义 $\{\mathbf{D}_{(\ell)}\}_{\ell=1}^{L}$。那么，$r$ 其规则度量如下[173,174]

$$\psi_t(r) = \bigcup_{\ell=1}^{L} \big(\psi_s\big(\mathbf{S}^{(\ell)}\big) - \psi_d\big(\mathbf{D}^{(\ell)}\big)\big)$$

[172]选择 $\psi_0(\mathbf{R}) = 1$ 对于所有 $\mathbf{R}$ 都是可以的；然而，巧妙的选择可能会使某些变换可用，而另一些则不可用。我们将使用一个符号系统来推断初始度量的一个值，若存在这样的值，则将某个变换指定为安全的。[173]此处使用的对齐 ℓ 由 σ 给出，它来自 展开 变换的对齐，该变换用于广义可逆性测试。[174]请注意，当我们 折叠 多条 规则 $|\mathcal{S}| > 1$时，我们可能会引入近似，因为多个不确定性集合被合并到 $\psi_t(r)$。

在这两种递归情形中，任何未更改的规则 $\mathbf{R} \in \left(\mathfrak{p}^{(t)} \cap \mathfrak{p}^{(s)}\right)$ 都会复制其 ψ，该测度来自 $\mathfrak{p}^{(s)}$。其中规则的规则度量 $\mathfrak{p}^{(s)}$ 和 $\mathfrak{p}^{(d)}$ 均以归纳方式定义。

使用相对大小确保终止。 我们将设计一个简单的测试，以确保 ϕ 的每次递归调用都会下降（即相对大小减小）。$_{s\to t}$我们将分别考虑折叠和展开的情况。

考虑折叠的情况。在这种情况下，我们会得到如下形式的对齐：

$$\left\{\left\langle \mathbf{S}^{(\ell)}, \mathbf{D}^{(\ell)} \right\rangle\right\}_{\ell=1}^{L} = \left\{\left\langle \left(\boldsymbol{x}^{(\ell)} \oplus\!\!= \boldsymbol{\alpha}^{(\ell)} \circ \boldsymbol{\mu}^{(\ell)} \circ \boldsymbol{\beta}^{(\ell)}\right), \left(\boldsymbol{y}'^{(\ell)} \oplus\!\!= \boldsymbol{\mu}'^{(\ell)}\right) \right\rangle\right\}_{\ell=1}^{L}$$

现在我们检查算法 25的递归调用。由于其他情况都很直接，我们将重点关注 第 25.7 行。下面，为避免符号表示过于繁杂，我们省略 ℓ 注释。在这种情况下，我们进行如下递归调用：

$$\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right) = \underset{s\to t}{\phi}\left(\begin{array}{c} x \\ \alpha\ \ \mu\ \ \beta \end{array}\right) = \underset{s\to t}{\phi}\left(\begin{array}{c} \alpha \end{array}\right) \underset{s\to t}{\phi}\left(u^{-1}\left(\begin{array}{c} \mu \end{array}\right)\right) \underset{s\to t}{\phi}\left(\begin{array}{c} \beta \end{array}\right) \quad \text{(with } x \text{ above)}$$

这些递归调用棘手之处在于，我们需要确保对 $\underset{s\to t}{\phi}\left(u^{-1}\left(\begin{array}{c}\mu\end{array}\right)\right)$ 的递归调用最终不会再次调用 $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$。我们将通过使用本节前面定义的推导度量来实现这一目标。假设每个推导的度量都相等（ 2 ），这意味着我们可以专注于证明 $\Psi\left(\boldsymbol{\delta}^{(s)}\right) > \Psi\left(u^{-1}\left(\begin{array}{c}\mu\end{array}\right)\right)$ 或者等价地，$\Psi\left(\boldsymbol{\delta}^{(s)}\right) - \Psi\left(u^{-1}\left(\begin{array}{c}\mu\end{array}\right)\right) > 0$。如果能做到这一点，$\phi$ 就不可能无限递归：每一步$_{s\to t}$都会减少一个非零正整数，而推导度量是 ≥ 0。很容易验证，同级调用 $\underset{s\to t}{\phi}\left(\begin{array}{c}\alpha\end{array}\right)$

和 $\underset{s\to t}{\phi}\left(\overset{\beta}{\triangle}\right)$都是递减的，因为它们是 $\boldsymbol{\delta}^{(s)}$的子树。

现在我们来推导这个测试。首先，根据定义37，我们展开 $\boldsymbol{\delta}^{(s)}$的推导度量

$$\Psi\left(\boldsymbol{\delta}^{(s)}\right)=\Psi\left(\begin{array}{c}\boldsymbol{x}\\ \boldsymbol{\alpha}\ \ \boldsymbol{\mu}\ \ \boldsymbol{\beta}\\ \triangle\ \ \triangle\ \ \triangle\end{array}\right)=\psi_s(\mathbf{S})+\Psi\left(\overset{\boldsymbol{\alpha}}{\triangle}\right)+\Psi\left(\overset{\boldsymbol{\mu}}{\triangle}\right)+\Psi\left(\overset{\boldsymbol{\beta}}{\triangle}\right)$$

回顾算法25可知

$$u^{-1}\left(\overset{\boldsymbol{\mu}}{\triangle}\right)=\underset{d\to s}{\phi}\left(\begin{array}{c}\boldsymbol{y}\\ |\\ \underset{s\to d}{\phi}\left(\overset{\boldsymbol{\mu}}{\triangle}\right)\end{array}\right)$$

其中，规则$\mathbf{D}=(\boldsymbol{y}\oplus\!=\boldsymbol{\mu})$是 $\mathfrak{p}^{(d)}$中的一条规则的实例化。由于该操作保持度量（根据假设），我们知道新推导的度量为 $\Psi\left(u^{-1}\left(\overset{\boldsymbol{\mu}}{\triangle}\right)\right)=\psi_d(\mathbf{D})+\Psi\left(\overset{\boldsymbol{\mu}}{\triangle}\right)$。

现在，我们可以将这些部分组合起来：

$$\begin{aligned}\Psi\left(\boldsymbol{\delta}^{(s)}\right)&=\psi_s(\mathbf{S})+\Psi\left(\overset{\boldsymbol{\alpha}}{\triangle}\right)+\Psi\left(\overset{\boldsymbol{\mu}}{\triangle}\right)+\Psi\left(\overset{\boldsymbol{\beta}}{\triangle}\right)\\ &=\psi_s(\mathbf{S})+\Psi\left(\overset{\boldsymbol{\alpha}}{\triangle}\right)+\left(\Psi\left(u^{-1}\left(\overset{\boldsymbol{\mu}}{\triangle}\right)\right)-\psi_d(\mathbf{D})\right)+\Psi\left(\overset{\boldsymbol{\beta}}{\triangle}\right)\end{aligned}$$

$$\Psi\left(\boldsymbol{\delta}^{(s)}\right)-\Psi\left(u^{-1}\left(\overset{\boldsymbol{\mu}}{\triangle}\right)\right)=\psi_s(\mathbf{S})+\Psi\left(\overset{\boldsymbol{\alpha}}{\triangle}\right)-\psi_d(\mathbf{D})+\Psi\left(\overset{\boldsymbol{\beta}}{\triangle}\right)$$

因此，如果以下条件对所有可能的兄弟节点 $\overset{\boldsymbol{\alpha}}{\triangle}$和 $\overset{\boldsymbol{\beta}}{\triangle}$均成立，我们就有下降。 $\overset{}{\triangle}$ $\overset{}{\triangle}$

以及不确定性集合中的所有可能测度均成立：[175,176]

$$(\psi_s(\mathbf{S}) - \psi_d(\mathbf{D})) + \Psi\left(\overset{\alpha}{\triangle}\right) + \Psi\left(\overset{\beta}{\triangle}\right) > 0$$

在实践中，我们用估计值 $\Psi\left(\overset{\alpha}{\triangle}\right)$ 和 $\Psi\left(\overset{\beta}{\triangle}\right)$ 替代 $\widehat{\Psi}(\boldsymbol{\alpha})$ 和 $\widehat{\Psi}(\boldsymbol{\beta})$，这些估计值仅取决于推导的顶部。[177] 这些估计值必须满足

$$\widehat{\Psi}(\boldsymbol{x}) \leq \min_{\boldsymbol{\delta} \in \mathfrak{p}^{(s)}\{x\}} \Psi(\boldsymbol{\delta}) \quad \textbf{for all } \boldsymbol{x} \in \mathbb{H}$$

请注意，测度保持意味着我们可以使用任意程序的推导$\widehat{\ }$集合来作为最小化的基准。因此，Ψ 可以根据初始程序 $\mathfrak{p}^{(0)}$进行估计，这还有一个额外的好处：其推导测度具有最高的保真度。[178] 我们在定义40中介绍了可能的估计器。

对于 展开变换，通过类似的论证，我们得出以下条件：[179]

$$\psi_s(\mathbf{S}) + \psi_d(\mathbf{D}) + \Psi\left(\overset{\alpha}{\triangle}\right) + \Psi\left(\overset{\beta}{\triangle}\right) > 0$$

请注意，$\psi_s(\mathbf{S}) + \psi_d(\mathbf{D})$ 不一定为正。

最后，我们必须确保整个来源链 $\mathfrak{p}^{(t)}$ 是安全的。这意味着，当我们沿着先前 折叠 和 展开 变换的序列向上追溯时，需要检查每个变换是否安全，以及它们是否源自同一个 $\mathfrak{p}^{(0)}$。

[175]当 x 是不确定性集合时，约定 $x > 0$ 是 $\forall x_i \in x$ 的简写：$x_i > 0$。[176]这一条件等价于 Roychoudhury 等人（1999）提出的条件。不过，我们是通过不同的论证得出这一条件的。[177]请注意，这里选择使用推导的相对大小而非相对高度，确实带来了好处，因为推导中的兄弟节点可以为满足该条件提供一些余地。[178]其不确定性集合应当是单元素集合。[179]Roychoudhury 等人（1999）没有提出这一条件，因为他们没有考虑广义的 展开。

定义 40. 以下是用于估计某项的最小派生测度的可能估计器。

- 最简单的估计器是 $\Psi(\cdot) = 0\widehat{\circ}$

- 一个简单的估计器是

$$\widehat{\Psi}(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \textbf{\textit{if }} x \textit{ is a constant, builtin, or input} \\ \widehat{\Psi}(x_1) + \cdots + \widehat{\Psi}(x_K) & \textbf{\textit{if }} x \textit{ is a product } x_1 \circ \cdots \circ x_K \\ 1 & \textbf{\textit{otherwise}} \end{cases}$$

- 更精确的 Ψ 可以通过以下线性不等式系统表示，该系统对可行估计器集合进行了充分刻画：[180]

$$\begin{cases} \widehat{\Psi}(\cdot) \geq 0 \text{对于 } x \text{常量、内置项或输入：} \quad \widehat{\Psi}(x) \leq 0 \text{对于} \mathbf{R} \\ \\ \\ \\ \end{cases}$$
$$\in \mathfrak{p}^{(0)}\colon \ (x \oplus\!\!= y_1 \circ \cdots \circ y_K) \leftarrow \mathbf{R}\widehat{\Psi}(x) \leq \psi_0(\mathbf{R}) + \widehat{\Psi}(y_1) + \cdots + \widehat{\Psi}(y_K)$$

对于$\mathbf{R} \in \mathfrak{p}^{(0)}$，$\psi_0(\mathbf{R})$ 应为单元素集合；因此，在这些不等式中，我们可以将其视为标量。

我们在算法 26 中总结了完整系统。

定理 1（算法 26的正确性）。设 $\mathfrak{p}^{(0)}, \mathfrak{p}^{(1)}, \ldots, \mathfrak{p}^{(t)}$ 是一个（测度安全）变换序列。然后，我们可以使用测度_安全$\left(\mathfrak{p}^{(?)}\right)$来评估候选折叠或展开 变换 $\mathfrak{p}^{(t)} \to \mathfrak{p}^{(?)}$ 的安全性。如果结果为真，则该变换是安全的。

[180]为了提高效义，我们可以对程序进行粗化，例如，采用最大深度限制，正如我们在 §6.1 中所做的那样。

证明概要。测度安全性测试（算法26）确保推导映射的递归过程（算法24 和算法25）总能终止。方法测度_安全的$\left(\mathfrak{p}^{(?)}\right)$ 只有在能够证明递归调用在候选折叠或展开下方安全地递减时，才会返回 真。本节前文已详细说明了这一点。因此，根据定理2和定理3，候选变换必定是安全的。■

总结。 本节基于 Roychoudhury 等人（1999）针对无权设置的先前工作，为加权情形提供了一种有效的安全性测试。我们发现，当测度_安全性利用约束求解器时，该系统具有出色的召回率。[181] 无需初始规则测度规范，使整个系统实用得多。

[181]得益于约束求解器，我们发现 Roychoudhury 等人（1999）提出的复杂分层计数器并非必需。

算法 26 折叠-展开变换序列安全性测试。我们假设将一个初始参照程序 $\mathfrak{p}^{(0)}$ 及其规则度量 $\psi_0(\cdot)$ 作为输入。

1. $\Psi(\cdot) \leftarrow$ {定义 40} 中的一个估计器

2. def $\psi_t(R)$*: 3. case* $\mathfrak{p}^{(t)} = \mathfrak{p}^{(0)}$*: return* $\psi_0(R)$ *4. case* $\mathfrak{p}^{(t)} =$ 折叠$^{d\,j}_{\mathcal{S}\to r}$, $(\mathfrak{p}^{(s)})$ 和对齐 σ*: 5. if* $R = r$*: 6. return* $\bigcup \langle \psi_s(S) - \psi_d(D) \rangle_{S,D\rangle\in\sigma}$ *7. else 8. return* $\psi_s(R)$ ▷ 复制未更改的规则 *9. case* $\mathfrak{p}^{(t)} =$ 展开 $^{d\,j}_{r\to\mathcal{S}}$, $(\mathfrak{p}^{(s)})$ 和对齐 σ*: 10. if* $R \in \sigma : t\to d$ *11. return* $\psi_s(r) + \psi_d\left(\underset{t\to d}{\sigma}(R)\right)$ *12. else 13. return* $\psi_s(R)$ ▷ 复制未更改的规则

14. **定义** $\varphi(t)$:
15. **情况** $\mathfrak{p}^{(t)} = \mathfrak{p}^{(0)}$: **返回**真
16. 情况 $\mathfrak{p}^{(t)} =$ 折叠$^{d\,j}_{\mathcal{S}\to r}$, $(\mathfrak{p}^{(s)})$ 和对齐 σ: 17. ▷ 在下面的代码中，S 的形式为 $\langle x \oplus\!= \alpha \circ \mu \circ \beta \rangle$，其中 α、μ 和 β 由 S 的折叠方式决定。18. 返回 $\varphi(s) \wedge \varphi(d) \wedge \bigwedge \left(\psi_s(S) - \psi_d(D) + \widehat{\Psi}(\boldsymbol{\alpha}) + \widehat{\Psi}(\boldsymbol{\beta}) > 0 \right)_{\langle S,D\rangle\in\sigma}$
19. 情况 $\mathfrak{p}^{(t)} =$ 展开 $^{d\,j}_{r\to\mathcal{S}}$, $(\mathfrak{p}^{(s)})$ 和对齐 σ: 20. $\left(x \oplus\!= \boldsymbol{\alpha} \circ \boldsymbol{y}_j \circ \boldsymbol{\beta}\right) \leftarrow r$ ▷ 这里，α、y_j 和 β 由 r 的展开方式决定。21. 返回 $\varphi(s) \wedge \varphi(d) \wedge \bigwedge \left(\psi_s(S) + \psi_d(D) + \widehat{\Psi}(\boldsymbol{\alpha}) + \widehat{\Psi}(\boldsymbol{\beta}) > 0 \right)_{\langle D,T\rangle\in\sigma}$

22. **def**测度_安全的$(\mathfrak{p}^{(t)})$:
23. ▷ 为获得最佳结果，请使用约束求解器为 $\{\gamma_{\mathbf{R}}\}_{\mathbf{R}\in\mathfrak{p}^{(0)}}$ 寻找满足命题 $\varphi(t)$ 的值。请记住，每个 $\gamma_{\mathbf{R}}$ 都必须是 > 0 且为整数。我们的实现使用流行的求解器（Moura 和 Bjørner，2008年），但我们指出，任何整数线性求解器都足以胜任。24. ▷（为更快获得结果，我们可以使用 $\gamma_{\mathbf{R}} = 1$ 检查所有 R $\in\mathfrak{p}^{(0)}$的 $\varphi(t)$ 是否可行。）25. ▷（返回 false 总是可以的，例如，如果我们想为 SMT 求解器设置超时。）26. 如果 $\varphi(t)$ 可满足，则返回 true，否则返回 false

7.5 提升变换

本节介绍两种相关的变换：推测变换（Eisner and Blatz，2007）和左角变换（Rosenkrantz 和 Lewis，1970；Johnson，1998；Johnson 和 Roark，2000）。我们的左角变换将上下文无关语法中著名的左角变换推广到了 Dyna 程序。[182] 我们发现左角变换和推测变换之间存在一种有趣的联系：两种变换都会将子目标沿树向上“提升”，但左角变换还会转置每个推导的结构——将左递归替换为右递归。

7.5.1 推测变换

推测变换最早由 Eisner and Blatz（2007）提出。我们的表述将与他们不同，因为我们将从它如何变换推导的角度来描述它。

例40. 再次考虑 *CKY* 程序（示例*6*）。[183]

```
474 p(X,I,K) +=p(Y,I,J) * p(Z,J,K) * r(X,Y,Z). 475 p(X,I,K)
 +=p(Y,I,K) * r(X,Y). 476 p(X,I,K) +=w(Y,I,K) * r(X,Y).
```

这个程序存在一处低效：它会对所有跨度⟨I,K⟩重复执行相同的一元链闭包。我们利用推测来重构程序，使一元链闭包能够预先计算（独立于句子），并在

[182]左角变换消除上下文无关文法中的左递归，从而使它们适用于特化的自顶向下句法分析算法。[183]下面的示例改编自 Eisner and Blatz（2007）。

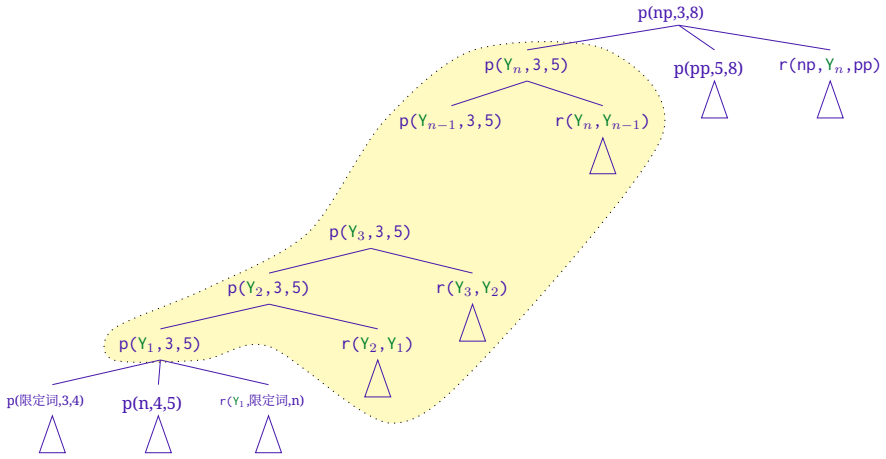
每个跨度上重复使用。这一技巧归功于 *Stolcke*（*1995*），是概率

句法分析。最终程序如下：

```
477 u(X,X') +=u(Y,X') * r(X,Y). 478 u(X,X') +=r(X,X').
479 p(X,I,K) += p'(X',I,K) * u(X,X'). 480 p(X,I,K) +=
p'(X,I,K). 481 p'(X,I,K) += p(Y,I,J) * p(Z,J,K) *
r(X,Y,Z). 482 p'(X,I,K) += w(Y,I,K) * r(X,Y).
```

但我们是如何得出它的呢？下面是原始程序的推导

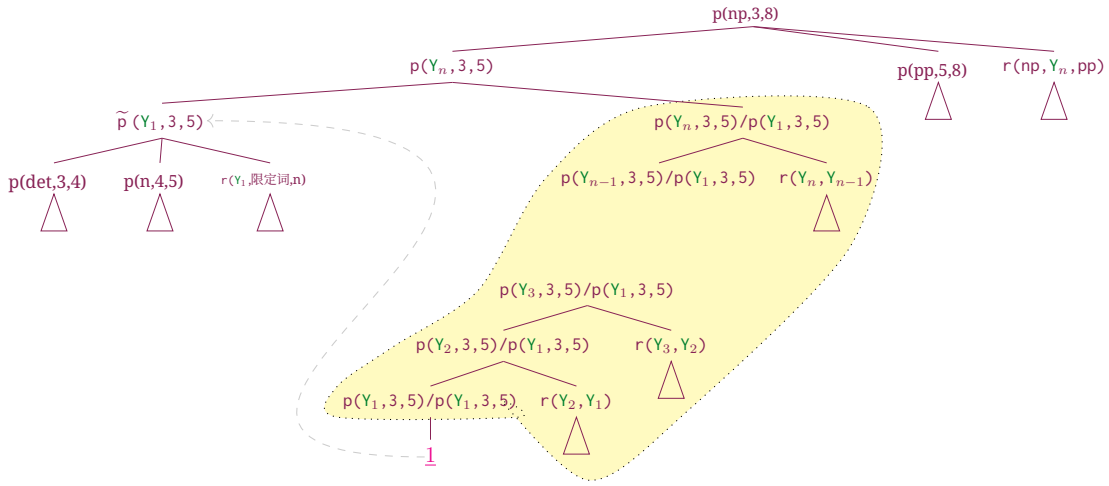
其中包含一条很长的#*475* 全部带有I=3、K=5（高亮显示）



我们的变换后的程序将使用一个斜线项[184] p(Yn,3,5)/p(Y1,3,5)，它计算中间链的总权重——具体来说，是那些从p(Y1,3,5)开始，到 p(Yn,3,5)结束，并经过规则子集（此处为 {#475}）。[185]这些不同的链将对应于斜线项的不同推导。我们可以把这个新项看作是对从特定起点 p(Y1,3,5)出发的链的抽象。稍后（即在推导树中更高的位置），我们可以使用一条恢复规则，确定一个形如p(Y1,3,5) 的具体起点，例如

[184]使用/ 并没有什么特殊之处——它不是一种特殊运算符，只是新项的名称，旨在表明 p(Yn,3,5) = p(Y1,3,5) * p(Yn,3,5)/p(Y1,3,5)的含义，就像/ 在实数算术中表示除法时一样。[185]具体的规则子集是该变换的一个参数。

p(Y$_n$,3,5) += ~p(Y$_1$,3,5) * p(Y$_n$,3,5)/p(Y$_1$,3,5)。出于技术原因，我们通过标记链底部的元素~（此处为 ~p(Y$_1$,3,5)），使其与原元素区分开来。我们的变换后的推导如下所示。



从图示中可以看出，链已被收纳进斜线项的一个推导中。事实证明，该子推导不依赖跨度 ⟨3, 5⟩的具体值——这意味着不同跨度可以重复使用它，并且在应用缩写（§6.2.3）后能带来运行时改进。以泛型三角形表示的子推导将递归地进行变换。

△

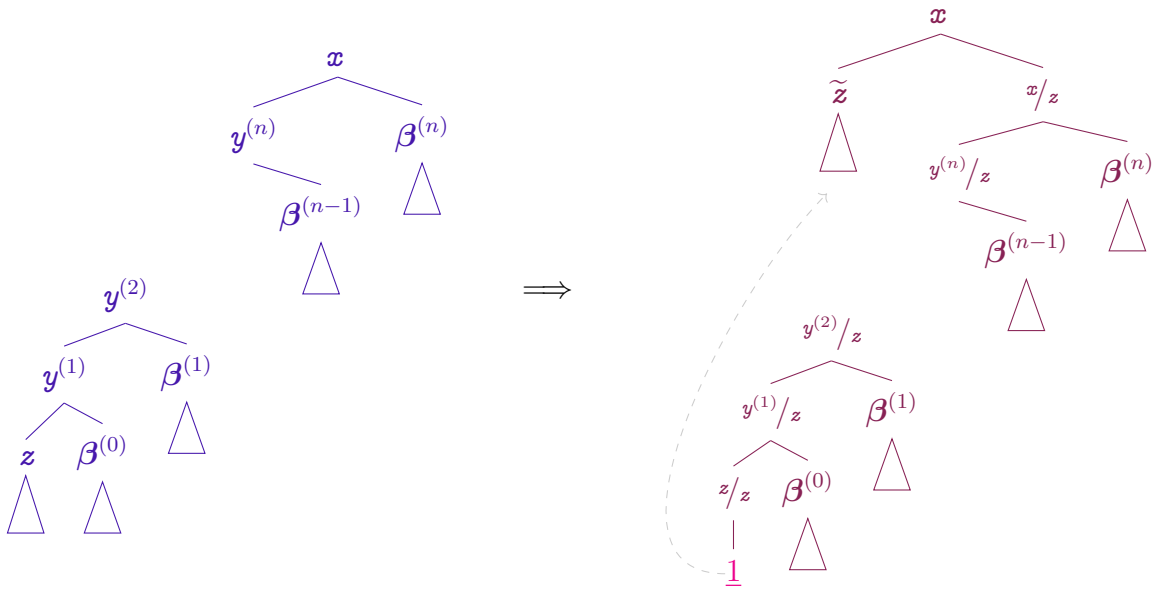
用推测记法表示，变换后的程序为

483 p(X,I,K) / p(X',I',K') += p(Y,I,K) / p(X',I',K') * r(X,Y). 484 p(X',I', K') / p(X',I',K'). 485 p(X,I,K) += ~p(X',I,K) * p(X,I,K) / p(X',I',K'). 486 ~p(X,I,K) += p(Y,I,J) * p(Z,J,K) * r(X,Y,Z).487 ~p(X,I,K) += w(Y,I,K) * r(X,Y).

程序特化（§6.2）可识别出 I=I' 和 K=K'。此外，它还识别出 I、I'、K和 K' 始终是斜线项中的自由变量，因此可以将它们删除。特化之后，我们便得到前面给出的最终程序。

推测变换由以下内容参数化：$\boldsymbol{\sigma} \subseteq \mathfrak{p}$ 以及一个非基项 z。令 $\boldsymbol{\delta} \in \mathfrak{p}\{\cdot\}$ 为程序 $\mathfrak{p}$ 的任意推导树。我们可以

将 δ 示意性地表示为左侧形式的推导。推测变换将把 δ 变换为右侧形式的推导。



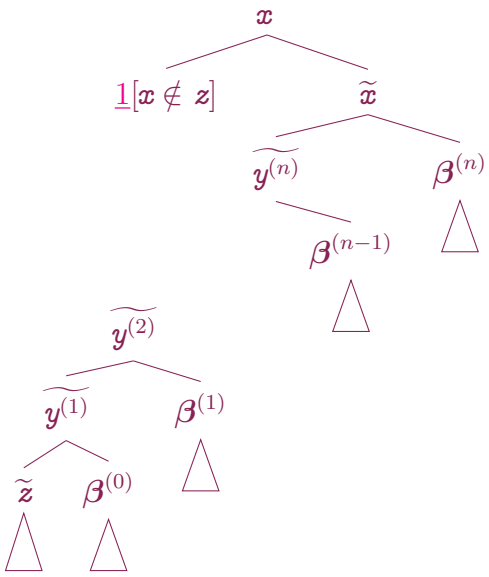
上面的示意图展示了推导的**脊柱**：沿着推导左边缘、仅使用某个指定规则子集的（最长）路径 $x \rightsquigarrow z$， σ 并在与非基项 z匹配的项处结束。[186] 正如右侧的推导所示，推测变换将以向上提起（提升）以 z 为根的子推导，留下**轨迹**$x/z \rightsquigarrow z/z$ 沿着脊柱 $x \rightsquigarrow z$，并在被提升的子推导原来的位置放置一个1 。我们将每个这样的 $\cdot/\cdot$ 项称为**斜线项**。新的推导仍以 x为根，但现在 z 在顶端连接着一个关于 x 不含z 的项的推导，该项位于其脊柱的基部。每棵以 $\beta^{(i)}$ 为根的子树仍然是关于相同 $\beta^{(i)}$ 项的推导，但由于变换会递归地应用于每棵子树，其内部结构可能发生变化。

回顾　例如 40 标记 $\tilde{z}$ 在 z 项上表示该项

[186]因此，如果 z 出现多次，我们取最底端出现的位置。

是链的底部。我们将把此标记用于另一种用途。在 $\mathfrak{p}$中，也可能通过其他方式推导出 x，使用 σ 中的规则，使得脊柱基部与 z不匹配。我们希望保留这些推导，因此创建一条标记为~的路径，允许使用 σ 中的规则构建、且脊柱上没有 z 的推导继续为 x贡献值。

这些被标记的项将对 x的值做出贡献，并使用一条规则（$x \oplus\!= \widetilde{x}$），但请注意，我们需要确保 $x \notin z$，因为 $\widetilde{x}$ 应当对 x 的推导求和，这些推导要么没有 z 处于其脊柱上——这要么是因为（a）它们不经过 σ，要么是因为（b）它们确实经过 σ，但它们的末尾并不是一个 z 项。如果 x 是一个 z 项，它便到达 z 时带有一条零长度脊柱。为了去掉这些不当的贡献，Eisner and Blatz（2007）增加了一条约束，即 $x \notin z$。



我们需要考虑的最后一种情况是，多个 z 项可能会出现在脊柱上。如果不区分这种情况，就可能会重复计数，因为有很多位置都可以让 $\widetilde{z}$ 该项尝试附着。我们采用 Eisner and Blatz 提出的解（2007）：将最底部的 z 项尽可能向上提

沿着脊柱能提到的最高位置（即通过将其附着在脊柱上最高的位置，消除附着位置的歧义）。

定义 *41*。给定程序 $\mathfrak{p}$，推测变换 $\mathfrak{p}' = \textit{speculation}(\mathfrak{p}, z, \sigma)$ 由非基项 z 和 σ 参数化，其中 z 表示要删去的项集，而 σ 是要从中删去规则的规则索引集合。为简便起见，下面给出的变换要求：*(1)* $\mathfrak{p}$ 没有输入，[187] *(2)* z 中的变量不能出现在 $\mathfrak{p}$ 中；*(3)* 对于每个 $i \in \sigma$，$(x \oplus\!= y_1 \circ \cdots \circ y_K) = \mathfrak{p}_i$：*(a)* $K > 0$；*(b)* y_1 不是常量或内置项；[188] *(c)* $y_1 \subseteq z$ 或 $y_1 \cap z = \emptyset$。[189]

算法 27 推测：应用推测变换（定义 41）。

```
1.  定义 推测(𝔭, z, σ):
2.      𝔭′ ← ⟅⟆
3.      对于 i = 1 … |𝔭|:
4.          如果 i ∉ σ:
5.              (x ⊕= μ)  ← 𝔭_i
6.              𝔭′[x̃] ⊕= μ
7.          else
8.              (x ⊕= y ∘ β)  ← 𝔭_i
9.              𝔭′[x/z] ⊕= y/z ∘ β
10.             如果 y ∩ z = ∅:
11.                 𝔭′[x̃] ⊕= ỹ ∘ β
12.     ▷ 斜线项的基本情况
13.     𝔭′[z/z] ⊕= 1
14.     ▷ 用于恢复 𝔭中原始项的规则
15.     𝔭′[X] ⊕= z̃ ∘ X/z
16.     𝔭′[X] ⊕= 不匹配_z(X)  ∘ X̃
17.     返回 𝔭′
```

[187]令人恼火的是，当程序具有输入时，所给出的变换会重命名这些输入（将它们包装在˜中）——从而破坏现有的程序 API。当输入的名称与程序所定义的项互不相交时，可以直截了当地 relax 这一假设：(1) 覆盖 $\tilde{x}$ 使其不标记 x 当它是输入关系时，以及 (2) 约束 第 27.16 行使其不会产生有问题的规则 $x \oplus\!=$ not-matches$_z(x)$ x.[188]为了支持划掉常量和内置关系，我们可以将它们包装在临时项中，然后在推测变换完备后（通过展开）消除它们。[189]有时，使用布尔侧条件将 $\mathfrak{p}_i$ 划分为三条更特化的规则是有用的：$(y_1 \cap z)$、$(y_1 \setminus z)$ 和 $(z \setminus y_1)$。

算法 28 不匹配：不匹配关系的实现。

```
▷此方法实现了 不匹配z（Y）                    def
  由非基项 z所表示的项集中的元素。重要的是， z中的变量 = 1[Y ∉ z]，一种用于检查 Y 不是以下项的
  是内置项的名称 的一部分，因此在内置项被用于一个
  规则中时，不会对它们进行求和。此方法应包含在内置关系库中。其实现如下：
  一个结果流以便和§3.1.2.2。
定义 不匹配z(y)：
  如果 合一(z, y) = 失败：yield 1                ▷ 确定，满足： z 和 y 无法匹配
  否则如果 覆盖(z, y) ≠ 失败：返回               ▷ 确定，失败： z ⊇ y 意味着 y 必须在 z里面
  否则 yield不匹配z(y)                          ▷ 不确定，延迟： y 仍有可能成为 z的一个实例
```

变换将生成新的函数符号，用于表示 $\not\cdot$ 和 $\tilde{\cdot}$ 项。这些名称必须与 $\mathfrak{p}$ 中各项的名称互不相交。伪代码使用 not-matches$_z(Y)$ $\stackrel{\text{def}}{=}$ $1[Y \notin z]$，这是一种内置关系，用于检查 Y 是否不是由非基项 z 所表示的项集中的元素。需要特别注意的是，z 中的变量不应进行求和——它们应当是内置项的局部变量。[190] 为完整起见，我们在算法 not-matches*28* 中给出了 。变换之后通常应继续使用 缩写 *(§6.2.3)* 来细化其输出，因为输出中通常包含无用的规则和变量。

扩展。 我们还注意到，对定义 41 的一种直接改写，是对最右侧子目标而非最左侧子目标进行推测。我们的实现同时提供最左侧与最右侧两种选项。当半环是交换的时，我们可以（作为预变换）置换这些子目标，使任意子目标成为其规则中的第一个子目标。即使没有交换性，某些子目标也可能彼此交换——特别是，内置约束可以被置换，因为它们乘以 1 或 0，而借助半环的性质，这些值可以自由移动。最后，我们注意到，许多

[190]使用内置关系的策略是一种变通方法，可以避免采用混合半环程序，正如 Eisner and Blatz（2007）的演示中所做的那样。

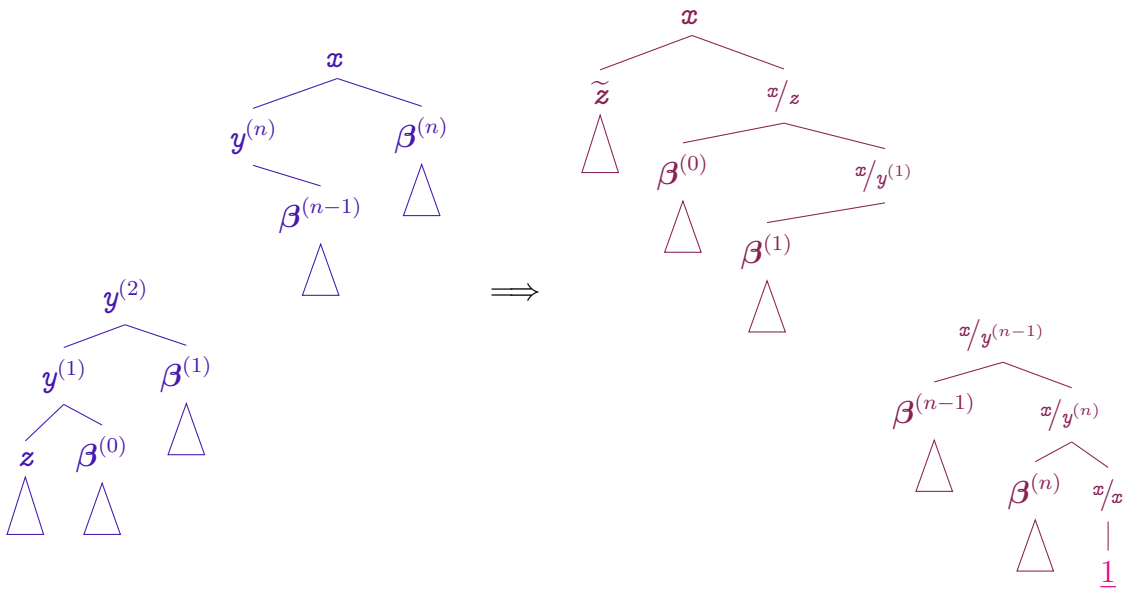
这些 $\tilde{\cdot}$ 由推测变换创建的项是不必要的；因此，常常值得在推测之后进行展开和无用规则消除，以去除一些不必要的中间 $\tilde{\cdot}$ 项。一个简单的选择是避免创建 $\tilde{\cdot}$ 规则，对象为任何不在 $\{z\} \cup \{y \mid (x \oplus\!\!= y \circ \beta) \quad = \mathfrak{p}_i$ 里面的项，$i \in \sigma\}$。[191]

正确性。我们认为推测变换是强语义保持的。$/\cdot$ 和 $\tilde{\cdot}$ 项应被视为新定义（§7.3），因为它们未在 $\mathfrak{p}$ 中定义。我们在§B.3中提供了该变换的推导映射。然而，我们尚未提供完整的正确性证明。

7.5.2 左角变换

我们将左角变换（Rosenkrantz 和 Lewis，1970；Johnson，1998）扩展到 Dyna 程序。为精细控制变换的应用方式，我们以选择性左角变换（Johnson 和 Roark，2000）为基础，该变换允许仅对规则子集应用变换。左角变换与推测密切相关：这两种变换都会将子推导提升，但左角变换还会将左递归转换为右递归（即转置推导主干）。下面的示意图以先前关于推测的讨论为基础。

[191]此 s　组项可用于专门化第27.15行和第27.16行上的恢复规则　.



与推测不同，变换后的推导轨迹中的斜线项采用了不同的命名方式；它们不是根据被提升的项 z 来命名，而是根据目的地 y来命名。变换正是借此转置推导树。当程序定义了许多与一组输出项无关的项时，这种效果可能会有所帮助。[192] 我们采用的左角变换不限制推导主干的基部，因此无需创建项的递归规则~。我们保留~项，以区分推导主干基部和顶部的项。

定义 42. 给定一个程序 $\mathfrak{p}$，左角变换 $\mathfrak{p}' = \mathsf{leftcorner}(\mathfrak{p}, \boldsymbol{\sigma})$由规则索引集合 $\boldsymbol{\sigma}$ 参数化。为简单起见，该变换要——

[192]这类似于前向模式与反向模式自动微分之间的区别（Griewank 和 Walther，2008年）。具体来说，推测与左角的关系，正如前向模式与反向模式的关系。微分方法构建形式为 $\frac{\partial x}{\partial z}$ 的项，这与提升方法构建形式为 x/z的项颇为相似。不过，它们构建的是不同的子集：反向模式和左角针对分子，而前向模式和推测针对分母。在自动微分领域，众所周知，当所需分子的数量少于所需分母的数量时，反向模式效率更高；而在相反情况下，前向模式效率更高。

要求 *low* 满足：*(1)* $\mathfrak{p}$ 没有输入，[193] *(2)* 对每个 $i \in \boldsymbol{\sigma}$，$(x \oplus\!= y_1 \circ \cdots \circ y_K) = \mathfrak{p}_i$：*(a)* $K > 0$，*(b)* y_1 不是常量、内置项或输入。[194]

算法29 leftcorner：应用左角变换（定义42）。

```
1.  定义 leftcorner(𝔭, σ):
2.    𝔭′ ← ʃʃ
3.    对于 i = 1 … |𝔭|:
4.      如果 i ∉ σ:
5.        (x ⊕= μ) ← 𝔭_i
6.        𝔭′[x̃] ⊕= μ
7.      else
8.          (x ⊕= y ∘ β)  ← 𝔭_i
9.        𝔭′[X/y] ⊕= β ∘ X/x
10.   ▷ 斜线项的基本情况
11.   𝔭′[X/X] ⊕= 1
12.   ▷ 恢复 𝔭中原始项的规则
13.   𝔭′[X] ⊕= Z̃ ∘ X/Z
14.   𝔭′[X] ⊕= X̃
15.   返回 𝔭′
```

定义底部段落中的注释*41*在此处适用。

扩展。 推测情况下的相同扩展也适用于此处：将最右侧子目标提升的右角变换，以及对可交换子目标进行置换。

正确性。 我们认为，左角变换能够强语义保持。它与文法变换足够相似，因此我们预计不会出现任何问题。不过，我们尚未正式检查所有细节。

[193]参见脚注 187 以了解变通方法。[194]参见脚注 188 以了解变通方法。

7.5.3 示例

示例 41。 如果我们对示例 *40*应用左角变换，会发生什么？变换后的程序几乎完全相同：两者生成的项都完全一致。不过，正如预期的那样，左角变换会转置递归：带斜线的项相同，但其定义使用的是右递归规则，而非左递归规则。具体来说，新程序与#*483*–*487*相同，只是我们将 #*483* （在下面重复）

```
488  p(X,I,K) / p(X',I',K') += p(Y,I,K) / p(X',I',K') * r(X,Y).
```

和规则

```
489  p(X',I',K') / p(Y,I,K) += r(X,Y) * p(X',I',K') / p(X,I,K).
```

例42. 本例探讨了推测与我们在 *§7.3*中看到的变量消去策略之间的连接。考虑以下单规则程序，它可视为三个矩阵的乘积：

```
490  a(I,L) += b(I,J) * c(J,K) * d(K,L).
```

将最后两个子目标因子分解为一个中间项tmp(J,L)：

```
491 a(I,L)  +=b(I,J) * 临时项(J,L).492
临时项(J,L) +=c(J,K)*d(K,L).
```

新程序更高效，因为规则的最大变量数减少了。我们可以通过对 $\boldsymbol{\sigma} = \{\#490\}$和 $\boldsymbol{z}$ =b(I', J') 进行推测来模拟这一变换：

493 a(I,L) / b(I',J') += b(I,J) / b(I',J') * c(J,K) * d(K,L)。 494 a(I,L) +=a(I,L) / b(I',J') * b(I',J')。 495 b(I',J') / b(I',J')。

接下来，我们通过 #*495* 展开来消除：

496 a(I,L) +=a(I,L) / b(I,J) * b(I,J). 497 a(I,L) /
b(I,J) += c(J,K) * d(K,L).

请注意，第二条规则总是有 I 空闲。因此，推测的项a(I,L) / b(I,J)实际上只是对 tmp(J,L) 项的一种命名方案，这是我们可以通过缩写变换（*§6.2.3*）自动完成的一步。

示例 43。 假设我们有如下程序

498 目标 += x(I)* f(I).499 目标
+= x(I)* g(J) * z.

并且我们希望将其因式分解为以下形式：

500 goal += x(I) * (???).

我们可以通过推测 $z = \texttt{x(I)}$ 和 $\sigma = \{^{\#}498, ^{\#}499\}$来实现：

501 目标 +=x(I') * 目标/x(I'). 502 目标/
x(I') += x(I)/x(I') * f(I). 503 目标/x(I')
+= x(I)/x(I') * g(J). 504 x(I')/x(I').

接下来，我们展开x(I)/x(I') 子目标：

505 目标 +=x(I')*目标/x(I')。 % unchanged506 目标/
x(I') += f(I')。507 目标/x(I') += g(J) * z。

注意，展开会将I 绑定到 I′ 在 #*506*中，而 #*507* 不受范围限制（即其头部含有自由变量）。我们看到，目标/x(I') 是一种巧妙的命名方案，用于表示这两条规则的贡献。这个程序具有我们所期望的形式。

例44。 拆分双词汇语法要求中心词必须先与所有右子节点结合，然后才能与任何左子节点结合。下面给出的朴素算法的运行时间为 $\mathcal{O}(n^5)$。*Eisner*和*Satta*（*1999a*）将其加速至$\mathcal{O}(n^3)$。*Eisner and Blatz*（*2007*）用推测来描述这种加速。为

简化这个示例时，我们假设使用无标记语法。

508 r(I,I,K) += 单词(I,K). % 尚无右子节点 509 r(P,I,K) += r(P,I,J) * arc(P → C) * l(C,J,K). % 添加右子节点 510 511 l(P,I,K) += r(P,I,K). % 尚无左子节点 512 l(P,I,K) += l(C,I,J) * arc(C ← P) * l(P,J,K). % 添加左子节点 513 514 目标 += l(P,0,N) * len(N).

加速的关键在于变换左成分，使其无需特定的右成分即可构建。这项工作可以独立于右成分完成，而我们将在下面的变换后的程序中推导出的斜线项不会实例化 K——从而缩短运行时间。以下是从右侧推测 $z = \texttt{r(H', I', K')}$ 和 $\sigma = \{^{\#}511, ^{\#}512\}$ 并省略无用的 goal 项并执行无用规则消除后的输出：

515 ~r(I,I,K) +=单词(I,K).516 ~r(P,I,K) +=r(P,I,J) * arc(P → C) * l(C,J,K). 517 518 l(P,I,K) / r(H',I',K') += r(P,I,K) / r(H',I',K'). 519 l(P,I,K) / r(H',I',K') += l(C,I,J) * arc(C ← P) * l(P,J,K) / r(H',I',K'). 520 521 目标 += +=l(H,0,N) * len(N). 522 523 % 斜线项的基本情况 524 (r(H',I',K') / r(H',I',K')). 525 526 % 恢复规则 527 r(P,I,K) += ~r(H',I',K') * r(P,I,K) / r(H',I',K'). 528 l(P,I,K) += ~r(H',I',K') * l(P,I,K) / r(H',I',K').

接下来，我们使用展开消除原始r 和l 项的所有用法；然后运行程序特化，揭示彼此相等的变量：

529 ~r(I,I,K) +=单词(I,K)。530 ~r(I,I,K) += ~r(I,I,J) * arc(I → C) * ~r(C,C,K) * l(C,J,K) / r(C,C,K)。 531 l(I',I',K') / r(I',I',K')。 532 l(I',I,K') / r(I',I',K') += ~r(C,C,J) * l(C,I,J) / r(C,C,J) 533 * arc(C ← I') * l(I',J,K') / r(I',I',K')。 534 目标 += ~r(H,H,N) * l(H,0,N) / r(H,H,N) * 长度(N)。

我们看到 th 的前两个参数 ~r(H,H,N) 总是相等的；因此我们可以

缩写 ~r(H,H,N) ⟼ rtrap(H,N)。

535 右梯形(I,K) += 单词(I,K)。536 右梯形(I,K) += 右梯形(I,J) * 弧(I → C) * 右梯形(C,K) * l(C,J,K) / r(C,C,K)。537 l(I',I',K') / r(I',I',K')。538 l(I',I,K') / r(I',I',K') += 右梯形(C,J) * l(C,I,J) / r(C,C,J) 539 * 弧(C ← I') * l(I',J,K') / r(I',I',K')。540 目标 += 右梯形(H,N) * l(H,0,N) / r(H,H,N) * 长度(N)。

我们选择这个名称，是因为它对应于*Eisner*和*Satta*（*1999a*）提出的"右梯形"概念；接下来我们将推导他们的"左梯形"。首先，我们注意到l(I',I,K') / r(I',I',K')中的冗余，其中三个变量被赋予相同值。我们可以将l(I',I,K') / r(I',I',K') ⟼ lr(I',I,K')。

541 rtrap(I,K) += 单词(I,K). 542 rtrap(I,K) += rtrap(I,J) * arc(I → C) * rtrap(C,K) * lr(C,J,K). 543 lr(I',I',K'). 544 lr(I',I,K') +=rtrap(C,J) * lr(C,I,J) * arc(C ← I') * lr(I',J,K'). 545 目标 +=lr(H,0,N) * rtrap(H,N) * 长度(N).

接下来，我们 注意到 K' 在……里面 lr(I',I,K') 始终是自由的，因此我们通过简写将其省略 g

lr(I',I,K') ⟼ ltrap(I',I).

546 rtrap(I,K) +=单词(I,K). 547 rtrap(I,K) +=rtrap(I,J) * arc(I → C) * rtrap(C,K) * ltrap(C,J). 548 ltrap(I',I'). 549 ltrap(I',I) += rtrap(C,J) * ltrap(C,I) * arc(C ← I') * ltrap(I',J). 550 目标 += ltrap(H,0) * rtrap(H,N) * 长度(N).

最后，我们应用变量消除来推导出以下新项：

551 临时项(C,I) +=ltrap(I,J)*rtrap(C,J)。

将这个新定义立即折叠进程序中，就得到了最终的 $\mathcal{O}(n^3)$ 时间

算法：

552 rtrap(I,K) +=单词(I,K) 553 rtrap(I,K) +=临时项(I,C) * 弧(I → C) * rtrap(C,K). 554 ltrap(I',I'). 555 ltrap(I',I) += ltrap(C,I) * 弧(C ← I') * 临时项(C,I'). 556 临时项(C,I) += ltrap(I,J) * rtrap(C,J). 557 目标 += ltrap(H,0) * rtrap(H,N) * 长度(N).

示例 45。 考虑以下用于计算.....的左递归程序

图中所有路径的总权重。

```
startpath(X) +=start(X)。 startpath(Y) +=
startpath(X)*edge(X,Y)。 目标 += startpath(X) *
stop(X)。
```

如果对它应用左角变换，就会得到下面的右递归

程序：

```
(目标 / start(X)) += 目标 / startpath(X)。 (目标 /
startpath(X)) += edge(X,Y) * 目标 / startpath(Y)。 (目标 /
startpath(X)) += stop(X)。 目标 += start(X) * 目标 / start(X)。
```

接下来，我们展开目标 / start(X) 底部规则中的内容，并缩写(目标 / startpath(X))

⟼ 停止路径(X)：

```
停止路径(X) +=边(X,Y)*停止路径(Y)。 停止路径
(X) +=停止(X)。 目标 += 起始(X) * 停止路径(X)。
```

该程序为目标计算出的值相同，但它使用的是从给定节点出发的路径总权重，而不是以给定节点为终点的所有路径的总权重。这两种方法都是计算所有路径总权重的知名方法。这里，我们介绍了一种从一种方法推导出另一种方法的机械化技术。（注意，如果我们对这个最后的程序应用右角变换，也会得到原始程序。）

示例 46。 回顾示例 *28*。该程序（在下面重复）对图中所有路径的总权重求和，其中路径以一种非常直接的方式表示为沿路径访问过的节点列表。

```
目标 += 起始(X1) * β([X1|Xs])。 β([X1,X2|Xs])
+=边(X1,X2) * β([X2|Xs])。 β([X]) += 停止(X)。
```

我们对该程序应用右角变换，其中 σ 等于所有规则。经过无用规则消除后，得到如下结果：

571 (Z / β([X$_1$ |Xs])) +=起始(X$_1$) * Z / 目标. 572 (Z / β([X$_2$ | Xs])) += 边(X$_1$,X$_2$) * Z / β([X$_1$,X$_2$ |Xs]). 573 (Z / 停止(X)) += Z / β([X]). 574 (Z / Z). 575 目标 += 停止(X$_1$) * 目标 / 停止(X$_1$).

特化后：

576 (目标 / 路径([X$_1$ |Xs])) +=起始(X$_1$) * 目标 / 目标. 577 (目标 / 路径([X$_2$ | Xs])) +=边(X$_1$,X$_2$) * 目标 / 路径([X$_1$,X$_2$ |Xs]). 578 (目标 / 停止(X)) +=目标 / 路径([X]). 579 (目标 / 目标). 580 目标 += 停止(X$_1$) * 目标 / 停止(X$_1$).

当我们运行这个程序时，列表的尾部Xs 在……里面 目标/路径([X$_2$ |Xs]) 从未被实例化。因此，我们可以缩写 目标/路径([X|Xs]) ⟼ 星路径(X)，并由此恢复高效算法示例 *27*。

7.6 讨论

实现。 我们已经实现并测试了这些变换。它们可在我们的系统中获取：https://github.com/timvieira/dyna-pi。在自动优化器（§8）的语境中使用这些变换，对它们进行了压力测试，因为有缺陷的变换往往会产生运行速度很快的程序（例如，空程序），而搜索会探测这些变换可能使用的参数。

Eisner and Blatz（2007）。关于变换半环加权逻辑程序的原始论文是 Eisner and Blatz（2007）。他们将折叠/展开

变换（Tamaki 和 Sato，1984），以及魔集变换（Ramakrishnan，1991）从无权设置推广到加权情形，并提出了一种新颖的推测变换。

⊂**单个与多个半环**。 Eisner and Blatz（2007）允许程序使用多个半环来组合值。他们通过转换半环程序片段来支持这种情况；我们的工作也应当兼容半环片段。正如我们在 §4中所讨论的，我们的语义基于对推导求和，而 Eisner and Blatz（2007）的语义则基于较弱的不动点概念。推导求和语义与半环的分配律密切相关。混合使用半环意味着这一性质并非总是适用。[195]

⊂**将广义折叠与展开引入加权情形**。Eisner and Blatz（2007）对加权情形下的折叠和展开变换所做的扩展仅限于基本折叠。因此，他们对折叠和展开能够实现什么的探索受到严重限制。特别是，他们从未发现像`startpath` 这样的自支持递归定义（例27）。我们将 Eisner and Blatz（2007）对折叠/展开的处理加以推广，使得可以相对于辅助程序进行折叠和展开，例如变换序列中程序的先前版本。此前已发现，将这一推广应用于辅助定义在无权设置中颇为强大（例如，Tamaki 和 Sato（1984）、Roychoudhury 等人（1999）、Bossi 等人（1992）以及 Pettorossi

[195]一个显著的例外是混合半环程序为 分层的情况——这意味着我们可以将其非 0 项划分为强连通分量（如 §5.4 中所做的那样），使得每个分量都使用单个半环。

和 Proietti（1994））。仅使用基本折叠，无法实现例28 和例 28这样的例子。我们给出了变换后程序的示例，展示了程序如何从不终止变为终止，以及如何从指数时间变为多项式时间。与 Eisner and Blatz（2007）（以及 Gildea（2011年））提出的有限示例不同，那些示例只采用了简单、表面的“钩子技巧”；我们的方法促成了自支持递归定义的发现。然而，为避免病态循环定义，还需要格外谨慎。我们在§7.4.2.2 中将病态定义问题刻画为推导映射的不终止。我们在§7.4.2.4 中提出了一种实用的系统来避免非良基定义，该系统建立在 Roychoudhury 等人（1999）的工作之上。

推测作为提升变换。 我们将推测变换描述为对程序推导结构的改变，从而进一步深入讨论这一变换。我们还将它与更广为人知的左角文法变换联系起来（ Rosenkrantz 和 Lewis，1970；Johnson，1998 年；Johnson 和 Roark，2000），并将其推广到了 Dyna 程序（§7.5.2）。

正确性证明。 Eisner and Blatz（2007）从未对（基本的）折叠或展开的正确性作出形式化证明；我们填补了半环加权逻辑编程文献中的这一空白。

对折叠与展开文献的贡献。 我们分析折叠–展开变换保留每个项的全部推导集合的能力。以往工作[196] 侧重于保留至少一个推导

[196]例如，Tamaki 和 Sato（1984）、Maher（1987 年）、Kawamura 和 Kanamori（1990）、Bossi 等人（1992年）、Roychoudhury 等人（1999）以及Pettorossi 和Proietti（1994）。

针对每个项，但这不足以满足我们的半环加权设置。

⌐**在 Roychoudhury 等人（1999）的工作基础上。** 我们的实用 折叠/展开 安全性测试（§7.4.2.4）直接建立在 Roychoudhury 等人（1999）针对非加权逻辑编程提出的工作之上。我们证明了，他们的技术可以直接扩展到加权逻辑编程。然而，我们采用了不同的正确性概念：强等价。因此，我们从另一个角度解释了他们的技术为何正确。我们还对论述（和证明）进行了组织，以便将变换的结构与用于检查其安全性的静态分析技术区分开来。此外，§7.4.2.4 比其他相关工作更明确地说明了他们方法中的静态分析建模假设。一项较小的贡献是，我们扩展了他们的方法，使其能够支撑广义的 展开变换；这一扩展很直接，但他们的工作中并未涉及。最后，从实践角度来说，我们提议使用约束求解器来免去指定初始度量的需要，从而提升了我们批准某些变换的能力，否则这些变换会被拒绝。

一个值得注意的遗漏：魔集变换。 Eisner and Blatz（2007）提出了魔集变换的加权推广（Ramakrishnan，1991年）。魔集变换是 Datalog 社区熟知的一种变换，因为它是一种简单的方法，可以针对特定查询对程序进行部分特化。魔集变换的一个令人振奋的应用是，将其用于 CKY 时，可以恢复 Earley 算法中的自顶向下过滤部分（Minnen，1996；Eisner and Blatz，2007）。遗憾的是，我们不支持 Eisner and Blatz（2007）版本的魔集，因为变换后的程序混合了

半环。支持混合半环程序是一个有趣的未来工作方向。不过，从另一个角度看，魔集变换是一种为基于议程的求值（§5.1.3）安排优先级的方法。我们可以推导出比魔集所提供的简单布尔启发式更紧的启发式。Felzenszwalb 和 McAllester（2007）以及 Prieditis（1993）展示了如何利用松弛程序的外部上下文，为细粒度内部遍历安排优先级。通过搜索要舍弃哪些约束和变量，可以自动发现并优化松弛程序。它为安排优先级而计算出的值可以比布尔值表达得更丰富。此外，对于优越半环，我们甚至可以自动推导出用于 A^*类解码的可采纳启发式。未来工作中，我们希望探索这一方向。

总结

我们介绍了若干可用于优化给定程序的程序变换。我们有许多示例表明，这些变换能够显著缩短运行时间。在 §8中，我们将使用搜索，在搜索预算内自动找到一个能使程序运行时间最短的变换序列。

第8章

搜索更快的动态程序

本章将介绍一种用于自动发现更快程序的工具。从总体上看，我们的方法很直接：给定一个初始的正确程序，我们将搜索一系列保持语义不变的源到源变换（§7），以最小化给定的代价函数，例如 §6.3 中的最坏情况运行时间界，或程序在给定输入工作负载上的平均执行时间。我们将讨论几种代价函数的优缺点，并介绍如何构建变换序列的搜索空间。最后，我们将通过实验表明，我们的概念验证系统不仅能够重现论文中给出的示例，还能重现文献中发现的许多被遗漏的加速（详见示例 1）。

8.1 图搜索问题

我们考虑一个抽象的**图搜索问题**，$\langle \mathcal{S},\ s_0,\ \mathcal{T},$ 转移、代价$\rangle$，其中 $\mathcal{S}$ 是状态空间，转移：$\mathcal{S} \to 2^{\mathcal{S}}$ 是转移函数，$s_0 \in \mathcal{S}$是初始状态，$\mathcal{T} \subseteq \mathcal{S}$ 是终止状态的集合，而代价：$\mathcal{T} \to \mathbb{R}_{\geq 0}$ 是定义在终止状态上的代价函数。[197] 搜索算法会将代价和转移函数视为黑箱。搜索问题的目标是在图中找到代价最低的（可达）终止状态，$s^* = \min_{s \in \mathcal{T}}$ 代价(s)。

我们的问题是程序优化问题（如图 8.1所示），可以直接映射到这种符号表示中。我们的状态 $s \in \mathcal{S}$ 是程序。[198] 初始状态 s_0 就是初始程序。每次状态转移都是应用一次保持语义的程序变换（§7）。这意味着，在此图中，从 s_0 可达的每个状态都是语义等价程序。在我们的设定中，每个状态都是终止状态。我们将在§8.1.1中介绍用于遍历此图的搜索算法。在 §8.1.2中讨论实验（§第8卷第3期）中所用搜索空间的构造细节。在 §8.2中讨论代价函数。

[197]原则上，代价函数不必取实数值；它可以是从状态到一个全序集合的任意映射。[198]要使用广义折叠/展开变换，必须为状态添加检查基于测度的安全条件所需的元数据，具体见§7.4.2.4。

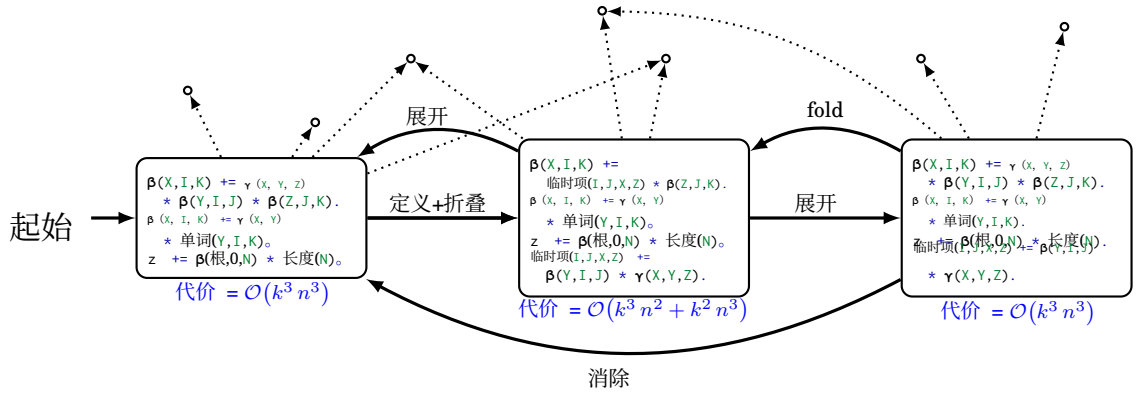


图 8.1： 程序优化图搜索问题的示意图。本图中使用的程序是 CKY（示例 6）。节点是 Dyna 程序（§2）。指向“起始”的节点表示用户的程序。边是程序变换（§ 7）。代价由程序分析得出（§6）。图中仅显示了搜索图中极小一部分节点和边。未标注的虚线出边代表其他变换；为避免图示过于杂乱，我们未在图中详细展开这些变换。

8.1.1 优先搜索算法

我们的搜索算法基于最佳优先搜索，这种方法常用于求解困难的组合优化问题（Russell 和 Norvig，2020年）。算法30给出了伪代码。最佳优先搜索算法的基本思想是维护一组候选状态 $\mathcal{Q}$ 以供后续探索，并根据搜索策略 π从候选状态中选出预计最有希望的状态并对其进行扩展。一旦选定某个候选状态，就根据转移枚举其后继状态，并将它们加入候选集合 $\mathcal{Q}$（如果之前尚未加入）。

我们整体的搜索算法search$_\pi$ 是一种随时可用的算法（Zilber- style id='6'>1996），它会遍历图，直到搜索预算耗尽。在search$_\pi$ 遍历图的过程中，会持续记录迄今找到的代价最低的终止状态。如果在算法探索完

算法 30 搜索：给定图搜索的算法问题 $\langle \mathcal{S}, \mathcal{A}, s_0, \mathcal{T}, \text{转移}, \text{代价} \rangle$。需要一种搜索策略 π 来选择要展开的候选状态。我们还要求指定搜索预算，例如一个搜索所花费的总执行时间上限。

```
1.  定义搜索π(s_0):
2.      Q ← {s_0}                                  ▷ 接下来要探索的候选状态集合
3.      已访问 ← {s_0}                             ▷ 已访问过的状态集合
4.      当 |Q| > 0 且尚未超出搜索预算时:
5.          s ← π(Q)                               ▷ 从候选项中任选一个状态
6.          Q.移除(s)
7.          如果 s ∈ T:
8.              计算 代价(s)
9.          对 s' ← 转移(s):
10.             如果s' ∉ 已访问:                   ▷ 如果我们之前还没有这样做过，就将 s' 添加为候选项
11.                 Q.添加(s')
12.                 已访问。添加(s')
        返回已评估的最低成本状态
```

整张图之前就达到搜索预算，它将返回截至那一刻找到的最优解。这使它成为一种随时可用的算法，因为即使尚未完全探索搜索空间（也就是在 $|\mathcal{Q}| = 0$之前），它也能提供一个解。搜索策略 π 决定了图中各状态的探索顺序。从抽象层面看，π 会接收候选状态集合 $\mathcal{Q} \subseteq \mathcal{S}$，并从中返回一个元素 $\mathcal{Q}$，也可能随机返回。它可以编码多种搜索顺序（例如深度优先搜索、广度优先搜索、贪心搜索和 A*）。在实践中，我们使用优先队列来实现$\mathcal{Q}$ 和 π，并在将某个状态加入$\mathcal{Q}$(第 30.11 行）时计算该状态的优先级。

如果搜索预算足够大，且 π 不会让任何候选状态挨饿，那么该算法可以保证最终找到可达的最佳状态。[199] 一种能够保证这一点的简单策略是广度优先策略，

[199]饥饿是指某个状态无限期地留在优先队列中。

该策略按照先进先出（FIFO）的顺序枚举状态；或者说，按照首次发现该状态时所经过路径的长度进行枚举。这种策略的随时性能并不好。另一方面，总是选择代价最低状态的贪心策略通常具有良好的随时性能，但可能会让某些状态挨饿。因此，我们采用一种交替使用贪心选择和先进先出选择的混合策略，以兼得两者的优势。

8.1.2 搜索空间详情

最基本的搜索图版本会认为，§7中的所有有效变换始终都可用。[200] 然而，这会导致需要探索的状态数量呈天文数字般增长，同时也会忽略一些有用的问题结构。本节将详细介绍搜索空间的组织方式。

尤里卡定义。 为了提出尤里卡定义（§7.3），我们的自动化系统会考虑变量消除操作。[201] 为了枚举可用的变量消除定义，我们逐一考察每条规则。对于给定的规则 $\mathbf{R}$，如果变量 $v \in \mathsf{vars}(\mathbf{R})$ 未出现在规则体的所有 子目标中，且未出现在规则头部中，那么就可以有效地消除该变量。[202] 因此，为了枚举所有变量消除定义，我们

[200]在 § 7 中介绍各项变换的章节里，我们讨论了这些变换有效的条件。[201]原则上，可以将其他尤里卡定义 §7.3 直接添加到我们的搜索框架中，作为额外的变换。[202]更形式化地说，可消除变量的集合为 elim(R) def= $\{v \mid \text{factors(R } v) \neq \text{body(R)} v \notin \text{head(R)}\}$ 其中 factors(R v) def= $\{y \mid y \in$ body(R) $v \in$ vars(y)$\}$。如果程序 $\mathtt{p}$ 中的某条规则 R 包含可消除的变量（$|\text{elim(R)}| > 0$），那么通过将 factors(R v) 折叠出 R 并纳入一条新规则，消除任何变量 $v \in$ elim(R)，都会降低该规则 R 的运行时界，因此可能降低（但绝不会提高）程序的运行时界。当

找出每个可以消除的变量，并使用 §7.3 中详述的方法提出一个新定义。

展开。 回顾一下，广义展开变换（定义 35）由源程序 $\mathfrak{p}^{(s)}$、规则索引 $r \in \mathfrak{p}^{(s)}$、r的子目标索引以及辅助定义 $\mathfrak{p}^{(d)}$作为参数。在这种参数化方式下，可以直接枚举展开的所有实例：任意规则中的任意非输入子目标都是展开的候选对象。关于 $\mathfrak{p}^{(d)}$ 的选择将在后文讨论。广义展开必须通过基于测度的安全性测试。

折叠。 回顾一下，广义折叠变换（定义36）以源程序 $\mathfrak{p}^{(s)}$、辅助定义 $\mathfrak{p}^{(d)}$、新规则 r以及将被替换的一组规则 $\mathcal{S} \subseteq \mathfrak{p}^{(s)}$ 作为参数。

• 我们按如下方式生成候选规则 r：对于每个有序对 $\langle \mathbf{S}、\mathbf{D} \rangle \in \mathfrak{p}_{(s)} \times \mathfrak{p}_{(d)}$，我们尝试用$\mathbf{D}$替换$\mathbf{S}$的体中的子乘积。[203] 由此，我们得到一组 r 候选项，并筛除不满足折叠变换可逆性条件的候选项。

• 集合 $\mathcal{S}$ 由以下内容决定： r 以及在广义可逆性测试中，我们在 j 的体中展开的位置： r。在广义可逆性测试中，我们进行广义展开变换。该展开变换会添加一组规则，而这组规则是 $\mathcal{S}$。

可以消除多个变量（$|\text{消除}(\mathbf{R})| > 1$），变量消除的顺序会影响最终的运行时界。遗憾的是，通过从概率图模型中的变量消除顺序问题归约可知，寻找最优的变量消除步骤序列是 NP困难（Eisner and Blatz，2007；Gildea，2011年）。[203]枚举子乘积的相关算法见 §C.2。规则体对齐后，我们将匹配的子乘积替换为 $\mathbf{D}$ 的头部。随后，我们将使用可逆性条件检查此替换是否有效。

• 广义折叠必须通过基于测度的安全性测试。

用于广义折叠/展开的辅助定义。为提高效率，我们将广义折叠和展开变换所考虑的辅助定义限制为每个项的原始定义。[204] 我们发现，对于我们的基准测试问题，这种选择已经足够。

减少安全性检查的次数。总体而言，我们发现，在由 π 选定变换后的程序之后再检查折叠/展开安全条件，效率更高。这样做显著减少了对约束求解器的调用次数。如此一来，我们只需检查最有希望的变换的安全条件，而不是每一个变换。从技术上说，这意味着只有通过安全性测试后，状态才是终止状态（即 $\mathcal{T}$ 的成员）。此外，对于测试判定为不安全的状态，我们不会继续对其进行变换。

提升变换。 回顾一下，提升变换（即推测和左角变换）以源程序 $\mathfrak{p}^{(s)}$ 和规则子集 $\boldsymbol{\sigma} \subseteq \mathfrak{p}^{(s)}$为参数。对于推测，我们还有一个额外参数 $\boldsymbol{z} \in \mathbb{H}$，用于描述项的类型。在非交换情形下，选择左侧还是右侧也是一个参数。在交换情形下，我们将规则 $\boldsymbol{\sigma}$ 中的任意子目标移动为最左侧子目标，也作为一个额外参数。遗憾的是，提升变换的搜索难度很大，因为可用的变换数量实在太多：有效提升变换的数量会随规则数量呈指数级增长——而且

[204]原则上，整个搜索空间中任何已访问的程序都可以用作辅助定义。然而，这会产生过多选项，而且这些选项似乎并不会增加实践中可达状态的集合。

如果程序的半环是交换的，情况甚至更糟！对于一小组程序（每个程序只有几条规则）来说，这或许尚可接受，但将其纳入搜索会导致候选状态过多。我们已在软件库中加入用于枚举这些变换的方法，但必须谨慎使用，否则搜索空间会变得过大。

程序特化。 回想一下，缩写变换（§6.2）可以加速给定的程序，但可能通过消除无用项（即，使其值为 0）来实现这一点。然而，我们的变换框架（§7）假定所有程序都是（强）等价的；因此，我们只能将缩写作为后处理步骤来应用。另一方面，代价函数可能希望计入缩写后的程序的代价。然而，经常运行缩写可能会减慢搜索。

规则待办列表。 我们考虑的展开和 Eureka 生成方法，都是围绕变换后程序当前版本中的特定规则展开的。按任意顺序对规则 $\mathbf{R}$,$\mathbf{R}'$ $\in \mathfrak{p}^{(s)}$ 应用变换，只要其中一种变换不会使另一种变换无法进行，得到的结果就相同。因此，我们按照规范顺序考虑变换。现在，每个状态都由一个程序以及一个 **待办列表** 组成，其中列出仍可进行变换的规则。该状态下可能执行的操作包括：从列表中移除顶层规则 $\mathbf{R}$ （选择不对 $\mathbf{R}$进行变换），或者应用下一个变换。应用此类变换可能会删除和/或添加程序规则，相应地，这些规则也会从列表中删除和/或添加到列表末尾。（如果列表为空，则无法再执行任何操作。）这种设计将分支因子降低了与规则数量成正比的幅度

。当然，一个潜在的缺点是，这会使搜索树的深度增加与规则数量成正比的倍数。

8.2 代价函数

代价函数大致分为两类：经验型和解析型。我们的实验只会使用解析型上界；我们将在此说明我们的立场。

经验成本函数。 最直接的优化目标是经验执行时间，即在特定计算机（例如你的笔记本电脑）上，使用特定的求解器配置（例如采用基于强连通分量的优先级排序的优先级不动点迭代），针对具有代表性的输入构成的特定工作负载运行给定程序所需的时间（例如，在宾夕法尼亚树库的实际句子上运行经过转换的句法分析器（Marcus 等，1993））。

解析成本函数。[205] 解析运行时间界，例如最坏情况运行时间界，是计算机科学中用于理论比较算法的首选方法。事实上，在整篇博士论文中，我们一直用这种方式比较算法。在我们的设定中，可以通过 §6.3 中给出的方法推导最坏情况解析运行时间界；该方法将 McAllester（2002年）著名的元定理自动化。[206] 我们还将提供一种

[205]虽然本文不对此进行探讨，但我们简要提及，可以根据输入数据的统计模型推导出平均情况运行时间估计。这类模型用于数据库查询优化器中（Deshpande 等，2007；Chekuri 和 Rajaraman，1997；Leis 等，2017）。[206]这一理论在自然语言处理中被广泛用于分析算法，例如 Gildea（2011）、Nederhof 和 Satta（2011）、Gilroy 等（2017）、Melamed（2003）、Kuhlmann（2013）、Nederhof 和 Sánchez-Sáez（2011）、Büchse 等（2011）、Lopez（2009）以及 Eisner and Blatz（2007）。

稍后在本节中介绍一种计算成本更低的代理指标。

直接执行的代价可能很高。 经验成本函数的评估代价可能非常高——尤其是在初始程序效率低下时。例如，初始程序的运行时间可能是无限的，但其优化版本的运行时间却可能有限。同样，评估$\mathcal{O}(n^{1000})$ 程序的执行时间，显然需要 $\mathcal{O}(n^{1000})$ 时间。为了避免某些程序在特定输入上运行过久，通常会对任何给定程序的执行设置时间限制。我们还面临一个问题：必须对多次运行的运行时测量结果取平均值，才能准确估算期望运行时间。通常，确定性代理测量更可取。在我们的场景中，观测到的前缀触发总次数或程序规则（§6.3）与不动点迭代算法的执行时间高度相关。

工作负载过拟合。 直接执行方法要求我们为程序指定特定输入；其缺点是我们可能会对 这些输入过拟合 。我们也可能过拟合于程序执行方式的特定细节（例如硬件特定细节）。优化解析度量或其他执行时间代理指标，可能会导致欠拟合。

实践效率与保真度的缺失。实践效率很容易超过保真度的不足。考虑评估一个 $\mathcal{O}(n^{1000})$ 程序的代价：评估墙钟时间需要 $\mathcal{O}(n^{1000})$ 时间，而评估运行时复杂度则不以这种方式取决于 n 。优化解析运行时间可以相较于直接执行带来关键性的加速，从而实现更

实践中的穷举搜索。此外，它还避开了指定具体工作负载的需要。当然，比较分析界的有效性高度取决于用于推导该界的分析的保真度：比较非常松的界可能会导致关于哪个算法更高效的错误结论。[207] 界背后的假设同样重要，例如输入类型规约的紧致性。

求解器配置。 我们讨论的是程序优化，而不是求解器配置。不过，我们想强调，在为给定问题开发高效的端到端算法时，求解器配置（包括求解器本身的选择）是一项重要决策。[208]

• §5 中求解器的选择非常重要：优先级不动点迭代和牛顿法都是不错的选择，但它们各有优缺点，具体取决于程序及其输入数据。此外，每种求解器都有自己的配置选项。例如，优先级不动点迭代算法的

[207]我们提醒读者，比较上界并不能可靠地得出哪个系统更好的结论。原则上，给定 $f_{lo} \leq f \leq f_{hi}$ 和 $g_{lo} \leq g \leq g_{hi}$ 我们只能在以下条件成立时，才能正确断定 f 一定小于 g：当且仅当 $f_{hi} \leq g_{lo}$。换句话说，要得出正确结论，既需要上界，也需要下界。遗憾的是，我们没有一种方法可以推导出有意义的运行时间下界。不过，如果运行时间界足够紧，并且能刻画典型行为，那么比较结果仍然可以准确。[208]许多高性能计算系统都经过调优，以优化执行时间。著名的例子包括数值库，如 ATLAS（Whaley、Petitet 等，2001年）、PhiPAC（Bilmes、Asanović 等，1996）、FFTW（Frigo 和 Johnson，1998）以及 SPIRAL（`http://www.spiral.net`），这些库利用执行时间，针对特定硬件调整底层实现细节。这些细节包括循环的{order, tiling, unrolling}，以及是否使用指令级并行。这些库如此受欢迎的主要原因之一是，在一台计算机上获得的性能提升未必能推广到其他架构或所有工作负载；因此，它们会针对特定硬件调整实现。一个可移植性能研究群体已经形成，致力于针对给定计算平台调优软件，使其尽可能快。我们还要提及针对计算困难问题的求解器配置。Hutter（2009年）采用黑盒优化技术，为混合整数规划求解器和可满足性求解器调优求解器启发式方法的参数，以针对特定问题工作负载优化效率。这种方法在多项竞赛中获胜，证明了针对特定工作负载调优求解器的有效性。

运行时间取决于优先级函数的选择。对于采用基于强连通分量的方法的循环程序，收敛阈值（如适用）的选择会影响运行时间。牛顿法存在不动点迭代所没有的开销，但求解循环程序的速度更快。

• 对于任意给定的程序和求解器，许多底层细节都会影响执行时间。例如，内存布局（如稠密数组在内存中的行优先或列优先布局）、关系的稀疏表示与稠密表示（如哈希表与数组），以及关系上的索引（包括排序顺序），在实际运行中都可能对运行时间产生巨大影响。然而，这些因素不会体现在解析运行时间界中。

幸运的是，对程序的改进通常与较低层次的求解器配置相互独立且相辅相成。我们的观点是，应先确保高层算法正确——也就是说，找到一个大-$\mathcal{O}$ 运行时间良好的算法——然后再优化较低层次的选择。之后，我们可以研究内存布局等底层技巧，以改善常数因子和平均执行时间。

8.2.1 细粒度运行时间界

在 §6.3中，我们设计了一种自动化方法，用于评估给定程序关于用户定义输入规模参数的最坏情况时间复杂度。[209]

[209]我们在 §6.3中讨论过这个界的有效性，但在此提醒读者，该界只有在程序无环时才成立。此外，该界还假设

比较界。需要对该方法给出的分析界进行比较。最直观的方法是针对输入规模参数的特定值计算运行时间表达式。对于单变量的运行时界，可以借助计算机代数系统（如 SymPy（Meurer 等人，2017年）或 Mathematica（Wolfram，2003年）），按照大-的意义，确定两个运行时界中哪个增长更快。[210] 以这种方式比较多变量运行时间表达式则更为棘手。一种选择是由用户将输入规模参数定义为单个参数 t（例如 $k = \log(t)$，$n = t$），并将问题归约为单变量情形。

效率。§6.3 中的方法可能会因几个原因而变慢：

- 对于某些程序，抽象前向链接可能需要做大量工作。极端情况下，程序可能没有输入数据；因此，抽象解释实际上要解决该程序的宽松布尔化版本。因此，与具体执行类似，抽象前向链接也可能需要对执行设置限制，详见（§6.1.2.5）。

- 开销：如果约束传播规则的生成能力很强（会产生大量推论），约束传播系统可能会很慢。此外，§6.3 中的方法会优化子目标排序决策，但对于体部较长的规则，这种优化可能效率不高。[211]

采用基于强连通分量的优先级排序进行优先级不动点迭代（§5.1.4）。[210]这些计算机代数系统可以按照大 $f(n)$的意义比较两个函数 $g(n)$的渐近行为：$\mathcal{O}$: $f \leq g \iff f = \mathcal{O}(g) \iff \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$。如果运行时界是简单多项式，这只需比较最高次项的度数。然而，§6.3 中的运行时间界可能包含 min，以及求和和乘法，因此比较会更加复杂；所以我们建议使用计算机代数系统。[211] 可以采用次优的子目标排序来加快处理。

这些低效率促使我们提出一种粒度更粗的分析界限，我们将在§8.2.2中介绍。

8.2.2 粗粒度运行时界

本节将介绍一种基于 Datalog 社区流传已久的民间定理、速度极快的代价函数。该民间定理指出，在满足某些条件（稍后讨论）的情况下，给定程序的前向链执行的运行时间至多与其实例化规则的方式数呈线性关系，也就是说，与将变量绑定到常量的方式数呈线性关系。[212] 如果能确定程序中的每个变量最多可以通过 η 种不同方式进行绑定，那么就可以得到一个相对简单的规则实例化数量上界。再加上本节末尾讨论的其他条件，包含 k 个变量的规则，其实例化方式数上界为 $\mathcal{O}(\eta^k)$。程序 $\mathfrak{p}$ 的总运行时间为 $\mathcal{O}(\eta^{\text{degree}(\mathfrak{p})})$，其中 度($\mathfrak{p}$) 是 $\mathfrak{p}$ 中任意规则所含变量数的最大值。因此，我们可以在搜索过程中将度 用作待最小化的代价函数；它的计算效率很高，也很容易实现。

举例来说，考虑 CKY。其度界限表明，下面的程序运行时间为 $\mathcal{O}(\eta^6)$，因为第一条规则必须对 6 个变量求和（X、Y、Z、I、J、K）。

600 β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K).601 β(X,I,K) += γ(X,Y) * β(Y,I,J).602 β(X,I,K) += γ(X,Y) * 单词(Y,I,K).

这是一种粗粒度分析：运行时间通常更具体地表示为 $\mathcal{O}(n^3\, k^3)$ 其中 n 是句子位置的数量，而 k 是[212]的数量。请勿将其与规则的非0 实例化次数混淆。

语法符号。（这种更精细的界限可以通过类型分析 §6.3实现：直观地说，变量 I、J、K 各自可以有 $\mathcal{O}(n)$ 种绑定方式，而 X、Y、Z 各自可以有 $\mathcal{O}(k)$ 种绑定方式。）然而，更简单的分析$\mathcal{O}(\eta^6)$只需归约一个指数，即 6。下面给出一个加速版本，它将变量Y 与 K分开求和：

```
603  β(X,I,K) += 临时项(I,J,X,Z) * β(Z,J,K)。604 临
时项(I,J,X,Z) += γ(X,Y,Z) * β(Y,I,J)。605  β(X,I,K)
 += γ(X,Y) * β(Y,I,J)。606  β(X,I,K) += γ(X,Y) * 单
词(Y,I,K)。
```

这个变体的运行时间为 $\mathcal{O}(\eta^5)$，因此效率更高。在更细粒度的分析下，它也更高效，$\mathcal{O}(k^2\, n^3 + k^3\, n^2)$。

对于包含多条规则的程序，我们可以细化规则次数分析，以测度程序中每条规则的次数，而不是用最大次数来概括。这样，我们就可以说，如果一个程序的规则次数列表（按降序排列）在字典序上小于另一个程序的规则次数列表，那么前者运行得更快。这样做的好处是可以减少运行时间中的低阶项，而不仅仅是最高阶项。

界的正确性。Dyna 程序的规则次数分析只有在某些条件下才能得到有效的 $\mathcal{O}$-表达式，下面我们将对此进行讨论。

1. 度 界假设基 程序无环。循环会拖慢前向链，因为我们必须迭代到不动点（§5）。通常，达到不动点所需的迭代次数取决于数据。

2. 度界假设程序中的所有关系都是有限的。这会阻止用户编码无限集合，例如皮亚诺整数。

虽然严格来说，这个界并没有错；只是 η 是无限的。

3. 该度 界还假设半环运算为常数时间 .

回顾第 §6.3 节，细粒度运行时间界作出了假设（1）和（3）。不过，细粒度界通常更紧，因为它能够考虑某些类型的稀疏性，并且其界不要求所有变量具有相同的域大小。细粒度系统可以通过类型推断检查（2）是否成立，因此不会假设它。

宽松性。 度分析可能因诸多原因而不够严格。使用我们的方法论推导出的上界假设关系是稠密的，并且所有变量都只有一种类型。我们注意到，度分析在进行程序特化（§6.2）后会更加严格。例如，如果程序中的两个变量始终相等，我们就不应将它们视为两个独立变量，但度界会将它们分别计算。幸运的是，特化可以使这些变量相等，从而改进度界。[213] 同样，如果某个变量始终被绑定到一个有限值集合（或者始终未实例化），那么计算度时就不应考虑它；特化会转换程序，使其不再包含这类变量。尽管这只是一种粗略的近似，降低程序的度通常效果不错。我们将在§第8卷第3期 看到，仅优化度就足以实现自然语言处理文献中提到的若干渐近加速（见§1）。

[213]为了加快搜索速度，我们的实验仅对最终程序运行特化 m.

第8卷第3期 实验

我们的目标是设计一个系统，自动改进典型的动态规划问题，尤其是自然语言处理应用中遇到的问题。为了评估我们是否实现了这一目标，我们设计了一组基准程序，以考察所提出的方法效果如何。我们的测试涵盖了§1 中提到的大多数未能实现的加速，以及博士论文中各处描述的许多示例；具体的程序集合见§E。

实验详情。

- 结果汇总于表8.1。

- 每个基准测试的名称都链接到§E 中的一个章节，该章节包含有关基准问题的更多信息，包括初始程序的 Dyna 代码，以及该程序在文献中出现位置的相关注释。对于每个基准问题，我们都手动指定了其最优度。在实验中，我们假设使用交换半环。

- 我们列出了每个基准测试的初始、找到的和最优度，以及搜索时间：搜索找到最优解（如果找到）的耗时（秒）。标记为“—”的单元格表示在 1 分钟的搜索时间预算内未找到最优度解。

- 一项检查 标记（✔）表示基准测试问题已求得最优解 l-

性（相对于度）代价函数。

• 对于搜索策略 π，我们采用混合贪心–先进先出策略（§8.1.1中提出），在贪心操作和先进先出操作之间交替进行。[214] 贪心策略按度优先排序。

讨论。

• 这些测试绝非完备，但它们表明，我们的概念验证系统能够自动实现多种加速。我们看到，除两种情况外，搜索都能在几秒内找到最优度数程序。我们认为，通过更高效地实现每种程序变换，可以显著缩短这一已经很快的耗时。例如，在用于这些实验的实现中，存在大量不必要的复制操作；随着程序规模增大，这些操作的代价可能很高。提高程序变换的效率，将使我们能够更快、更完备地搜索状态空间。

• 基准测试{path-list、even-odd-peano、explicit-pda}与其他基准测试不同，它们各自的初始版本都具有无限大小的变量域（即$\eta = \infty$）。不过，它们都可以经过优化，使变量域变为有限。尽管度数上界未考虑 η的变化，搜索为path-list 和 even-odd-peano找到的程序都是最优的。

[214]我们没有优化贪心操作与先进先出操作的比例，因为这可能会导致在这组规模较小的基准问题上过拟合。我们确实发现，完全采用先进先出策略会显著增加平均搜索时间。

name	初始度数	找到的度数	最优度数	搜索时间（秒）
弧优先	6	✔ 3	3	1.375
bad-chain-05	3	✔ 2	2	0.398
bad-chain-10	3	✔ 2	2	1.289
巴-希勒尔	10	✔ 8	8	0.013
双词汇标注	8	✔ 7	7	0.014
bilexical-unlabeled	5	✔ 4	4	0.049
chain-05	6	✔ 2	2	0.016
chain-10	11	✔ 2	2	0.090
链期望	3	✔ 3	3	0.003
CKY+语法	6	✔ 3	3	1.424
cky3	6	✔ 5	5	0.003
cky4	8	✔ 6	6	0.012
edit	6	✔ 4	4	0.008
even-odd-peano	1	✔ 0	0	5.812
explicit-pda	8	6	5	—
hmm	5	✔ 4	4	0.003
itg	9	✔ 8	8	0.011
path-list	3	✔ 2	2	1.164
路径起点	3	✔ 2	2	8.707
半马尔可夫	4	✔ 3	3	0.002
split-head-EB	5	4	3	—
split-head-J	4	✔ 3	3	0.780
tensor-decomp-parser	6	✔ 4	4	1.916

表 8.1： 在§中所述基准测试上对我们的基于搜索的程序优化器进行E实验演示。

• 我们还要强调，我们的系统无法将 split-head-EB 基准测试提升至最优。我们确实成功加快了它的速度（$\mathcal{O}(\eta^5) \to \mathcal{O}(\eta^4)$）。然而，我们无法实现最优的 $\mathcal{O}(\eta^3)$ 算法，因为我们的搜索空间中没有提升变换（见 §8.1.2）。遗憾的是，我们尚未找到一种高效的策略来搜索提升变换。不过，我们已经验证，手动用一轮针对初始程序的推测变换来初始化变换搜索，就足以让我们的算法找回最优度算法。问题在于，在其他状态中加入这些变换会使搜索空间膨胀得过大。

• 这个explicit-pda 基准测试似乎超出了我们的程序变换空间所能达到的范围。我们将在 §E.12中说明如何对其进行优化。缺失的技巧似乎是对推测的一种泛化。

总结

本章中，我们开发了一种工具，用于自动发现更快的程序。具体做法是搜索能够使给定代价函数最小化的变换序列，并将此问题归结为一种图搜索。我们给出了一种简单的随时算法，用于在该图上执行程序优化。我们讨论了在优化问题中使用各种代价函数的利弊。最后，我们通过实验演示说明了原型系统如何优化若干取自有关被忽视的加速效果的文献的基准问题（示例 1）。在我们的基准测试中，除两个之外，搜索都能在几秒钟内找到快速算法。不过，仍有改进空间。实验中遇到的一个突出问题是，由于可用的提升变换太多，搜索这些变换十分困难。我们还展示了一个例子，它超出了当前变换空间所能达到的范围，这表明未来工作需要扩展可用的变换。尽管如此，我们很高兴看到，文献中发现的许多加速效果都能通过自动搜索迅速找到。因此，我们认为这个工具已经具有显著价值：使用它时，这些此前被忽视的加速效果及类似效果将不再被遗漏。

第9章

相关工作与未来工作

9.1 相关工作

Dyna 1。 我们大量借鉴了 Eisner、Goldlust 等人（2005年）以及 Eisner and Blatz（2007）的工作。我们在本博士论文的各处对这些连接进行了广泛讨论。

Dyna 2。 Dyna 2语言（Eisner and Filardo，2011年）与我们在本文中讨论的 Dyna 1 语言密切相关，因为 Dyna 1程序包含于 Dyna 2程序集合中。我们在§D 中用一小节介绍了该语言以及我们工作的可能扩展。

算法 = 逻辑 + 控制。将算法（用于句法分析或其他用途）分解为逻辑规约和控制器，源自 Kowalski（1979年），他写道：“算法 =逻辑 +控制。”后来，在句法分析算法的背景下，Shieber 等人（1995）写道：

> “尽管演绎系统并未规定详细的控制结构，但将它们转变为完整句法分析器所需的控制信息是统一的，因此可以由单个演绎引擎提供，该引擎能够对推理规则进行可靠且完备的自底向上解释。”

算法的逻辑规约还在许多著作中得到进一步研究，例如 McAllester（2002年）、Ganzinger 和 McAllester（2002年）。然而，在 Pereira 和 Warren（1983）以及 Shieber 等人（1995）发表了开创性论文之后，自然语言解析社区才真正接受了这一方法。在自然语言处理社区之外，编程语言社区广泛使用逻辑演绎记法来规约操作语义（Plotkin，2004年）和程序分析（Whaley、Avots 等，2005年；Ullman，1988年；Dawson 等，1996；Reps，1993；Esparza 等，2010年）。

解析即演绎。Pereira 和 Warren（1983）发表了一篇面向自然语言处理和句法分析社区的开创性论文，将句法分析视为逻辑演绎。之后，Shieber 等人（1995）确立了演绎系统作为一种有价值的框架，可用于简洁地规约句法分析算法。

Goodman（1998）改进了演绎框架，以适应当时刚刚兴起的对统计句法分析的兴趣。他指出，许多值得关注的量，例如总概率、最可能的句法分析结果以及最可能的句法分析结果集合，都可以使用同一种加权推理算法计算，差异则通过选择不同的半环来体现。

权重。[215] 这是一个重要的发现，也与大致同时期图模型领域的研究成果相吻合（Aji 和 McEliece，2006）。此时，显然需要一种用于描述加权演绎系统的编程语言。于是，Dyna 表示法应运而生（Eisner、Goldlust 等人，2005年；Eisner、Goldlust 等人，2004年）。

演绎系统已被用于表达多种形式系统：

- Earley句法分析（Pereira和Warren，1983）
- 投射依存句法分析（Eisner和Satta，1999b）
- 非射影依存句法分析（Cohen、Gómez-Rodríguez等人，2011年）
- 树附加语法（Alonso等人，1999）
- 组合范畴文法（Kuhlmann和Satta，2014年）
- 线性上下文无关重写系统（Burden 和 Ljunglöf，2005年）
- 机器翻译模型（Melamed 等，2004年；Lopez，2009年）

我们还提到了 Sikkel（1997年）、Gómez-Rodríguez（2010年）和 Goodman（1998），他们的著作包含了大量与自然语言处理中句法分析相关的演绎系统实例。

演绎系统表示法与 Dyna 表示法。 半环加权演绎系统与 Dyna 之间的差异在很大程度上是表面的。

[215]我们还提到了 Teitelbaum（1973年），他指出 CKY算法可以扩展到适用于任意半环。Goodman（1998）将这篇论文视为大多数关于半环加权演绎的研究的奠基之作。

考虑如下形式的 Dyna规则：

$$x \oplus\!= y_1 \circ \cdots \circ y_K$$

用演绎系统记法表示为

$$\frac{y_1 \quad \cdots \quad y_K}{x}$$

演绎记法常用于研究论文和书籍中。然而，它并不是一种便于用户使用的编程语言。[216] 此外，Dyna 表示法更明确地说明了如何组合项值。[217]

演绎数据库与逻辑编程。Dyna 表示法是 Prolog（以及 Datalog）的自然推广，它允许项具有非布尔值。Green、Karvounarakis 等人（2007）——独立于 Eisner、Goldlust 等人（2005年）——发现了为 Datalog规则附加半环值的优点及其用途。然而，他们的工作关注的是数据溯源[218]，而不是动态规划。我们在许多方面借鉴了逻辑编程文献：我们的语义、执行以及源到源转换都根植于逻辑编程文献。这些联系将在本博士论文中逐一讨论。

einsum记法。 Dyna 程序类似于 einsum记法，后者广泛用于科学计算库工具包，例如 NumPy（Harris

[216]Gómez-Rodríguez 等人（2009年）构建了一个系统，用于解释以纯文本形式编写的无权规则，其记法仿照演绎系统记法。[217]在 Dyna 2（见§D）中，可以用任意方式组合项值（即，不限于两种半环运算），而这在演绎记法中是不可能的。[218]数据溯源是一种记录轨迹，用来追踪某些数据库项为何存在。实际上，数据溯源提供了一种推导，说明某个项为何属于该数据库。

等，2020年），作为张量积的简洁符号表示。

示例 47. 下面是 `einsum` 代码（在`% 注释` 上面）及其等价 *Dyna* 代码的有序对示例。

```
607 % out0 =einsum('i->', a) 608 out0 +=
a(I). 609 % out1 =einsum('i,i->i', a, b) 610
out1(I) += a(I) * b(I). 611 % out2 =
einsum('i,j->ij', a, b) 612 out2(I,J) +=a(I) *
b(J). 613 % out3 =einsum('ii->', c) 614 out
3 +=c(I,I). 615 % out4 =einsum('ij,ji->ij', c,
d) 616 out4(I,J) +=c(I,J) * d(J,I). 617 % out
5 = einsum('ij,kl->ijkl', c, d) 618 out5
(I,J,K,L) += c(I,J) * d(K,L). 619 % out6 =
einsum('ij,jk->ik', c, d) 620 out6(I,K) +=
c(I,J) * d(J,K).
```

Dyna 表示法以多种方式扩展了 einsum——其中最重要的一点是，它支持相互交织的递归方程。我们希望有朝一日，Dyna——或其某种变体——能取代 einsum，成为张量操作的默认领域专用语言。

上下文无关文法。 将 Dyna 程序与上下文无关文法（CFG）联系起来很有意思。映射如下

(i) 终结状态 $\Longrightarrow$ 值 (ii) 非终结符 $\Longrightarrow$ 基项 (iii) x $\rightarrow$ y z $\Longrightarrow$`x += y * z`(iv) 语法分析树 $\Longrightarrow$ 推导 (v) 语法分析树的 yield $\Longrightarrow$ 推导的值 (vi)（由某个非终结符）生成的语言 $\Longrightarrow$ （该项的）所有推导的总值

主要区别在于，Dyna 不会将终结符和非终结符的集合限制为有限集合。我们发现，探索 Dyna 程序与 CFG 之间的连接，促使我们发现了左角变换（§7.5.2）和牛顿求解器（§5.2）。最后，我们注意到，我们的程序变换或许可用作文法变换，而我们的搜索方法或许能有效减小文法大小。

9.2 未来工作

超越半环。在未来的工作中，探索 Dyna 1 语言的扩展将会很有趣。

- 我们在 §D 中讨论了扩展到 Dyna 2 的内容。

- 我们还提到，允许非结合乘积运算的一种半环松弛（例如 Gildea（2020年））或许值得探索。这样一来，项值就可以是树，而不是由值组成的字符串。只要注意一些细节，我们认为许多结果很可能都能扩展到这个更通用的框架中；不过，我们将这项探索留待未来工作。

提高效率。在本博士论文中，我们尚未探索许多提高效率的方法。下面列举其中几种。

- 代码生成和编译对于高效运行 Dyna 程序至关重要。Eisner、Goldlust 等人（2005年）为 Dyna 提供了一个编译器，但该编译器已不再积极开发。我们希望有朝一日能提供类似的编译器。

• 针对某些半环的优化能够带来更高效的执行策略。例如，优越半环适用于 A^* 类执行算法（Huang，2008年；Felzenszwalb 和 McAllester，2007；Knuth，1977年）。我们希望在未来工作中探索这些方法。

• 用于优先级固定点迭代算法（§5.1.3）的优先级函数优化可以显著缩短我们的执行时间，因为我们将不再需要进行 SCC 分解遍历。或许可以像 Bagnara 等人（2010年）以及 Podelski 和 Rybalchenko（2004年）那样，使用 SMT 合成优先级函数；他们将这种方法应用于命令式程序。合成有效的优先级启发式方法可以大幅提升我们合成高效代码的能力。

• 数据结构优化可以显著提升我们在执行算法最内层循环中存储和检索值的能力。以下数据结构选择在实际应用中会对执行时间产生重大影响：内存布局（例如，稠密数组在内存中的行优先或列优先布局）、关系的稀疏表示与稠密表示（例如，哈希表与数组），以及关系上的索引（包括排序顺序）。[219]

• 探索 Filardo 和 Eisner（2012 年；Filardo，2017年）提出的内存高效的混合链接算法，也会很有意义。

[219] Vieira、Francis-Landau 等人（2017年）描述了一种强化学习技术，用于在执行过程中动态优先选择更快的数据结构。该技术使用一种策略，在不复制数据的情况下，动态调整存储在各个相互竞争的数据结构中的数据量。

改进变换搜索。 我们提供了一种概念验证搜索算法，但它并不十分复杂。以下是一些可能的改进思路：

• 将所探索的程序集合分解为可能彼此依赖的子任务，便能实现一种动态规划，从而应当提高搜索效率，因为每个子任务都可以被递归优化，并组合成能够高效计算所需输出任务的程序。

• Vieira、Francis-Landau 等人（2017年）提出了动态优化，使求解器适应特定工作负载，在执行过程中使用强化学习，而不是像我们在§8中那样预先进行。他们还提出分解为子任务以加快学习。

拓展程序变换的集合。算法设计者使用的许多可复用技巧，至今仍未被系统化为程序变换；另有一些研究正在探索组合算法的高效编译。一个值得关注的近期成果是 PERPL（Chiang、McDonald 等人，2023年），这是一种概率编程语言，能够高效地编译掉某些类型的“线性”数据结构，例如栈和列表。我们希望将他们的编译技术移植为一种适用于 Dyna 程序的源到源变换。这样一来，我们便能获得有用的加速，从而提升自然语言处理算法的性能（例如推栈自动机和线性索引语法）。原则上，PERPL 使用的高效编译策略可以变换下面这个低效程序，该程序用于对一个推

下推自动机

621 推栈自动机([],0)。 622 推栈自动机([X|Xs],K) += 推栈自动机(Xs,J) * rewrite(X,Y) * 单词(Y,J,K)。 623 推栈自动机([X|Xs],K) += 推栈自动机([Z,Y|Xs],K) * rewrite(X,Y,Z)。 624 目标 += 长度(N) * 推栈自动机([s],N)。 625 626 输入：rewrite(_,_,_); rewrite(_,_); 长度(_); 单词(_,_,_)。 627 输出：目标。

转化为一种与 CKY 等价的高效算法。

628 p(I,X,K) += rewrite(X,Y) * word(Y,I,K)。629 p(I,X,K) += 推栈自动机(I,Y,J) * p(J,Z,K) * rewrite(X,Y,Z)。630 目标 += 长度(N) * p(0,s,N)。

我们在§E.12中进一步讨论了这个示例。

第10章

结论

我们将 Dyna 介绍为一种动态规划的符号表示（§2）。我们在示例 1 和示例 2 中展示了，自然语言处理领域的作者历来忽视了加速效果。我们认为，这说明需要借助工具来促进自然语言处理算法的分析与开发。

我们研究了 Dyna 的基础：语义（§ 4）和执行（§5）。我们提出了一种新的推导求和语义。我们发现了一种新的执行算法——牛顿法，并将 Eisner、Goldlust 等人（2005年）的前向链式算法推广，以支撑非范围受限程序。我们识别并探索了若干丰富的源到源变换族，它们将许多论文中用于推导出更高效算法的加速技巧系统化，其中包括广义折叠/展开、推测和左角变换。

我们提供了一种基于搜索的系统（§8），用于为给定的初始程序寻找一系列保留语义的源到源变换，以

最小化其最坏情况下的运行时间。我们的程序分析器（§6）能够自动完成 NLP 论文中的许多算法分析工作，包括推导出关于运行时间和空间复杂度的精确渐近界$\mathcal{O}$ 。我们展示了该系统能够找出 NLP 文献中发表的许多有趣疏漏，包括分析中的疏漏（示例2）以及被忽视的加速效果（示例1）。

我们发布了一个可免费使用的原型实现 https://github.com/timvieira/dyna-pi。我们发现，我们的系统的实用之处在于，它有助于发现正确的（即高效的）高层算法。换句话说，对于算法开发者和自然语言处理系统的作者而言，我们的工具最有助于找到正确的递推——类似于定理证明助手。我们的实现尚不能像 Eisner、Goldlust 等人（2005年）那样进行高效的代码生成。该实现还包括一些有用的开发实用工具，例如推导过程可视化，这些内容我们在此未作讨论。

我们相信，这篇博士论文标志着朝着自动化开发高效动态程序迈出了重要一步，尤其是自然语言处理应用中出现的动态程序。

附录 A

推理算法

本节展示了我们的算法如何 solve 某些具有无限多个非0项。

例 48。 在……下面，我们有三个版本的几何级数程序。左侧是基本的几何级数。中间版本贯穿了一个无用的变量 `I`。右侧版本则在 `I` 上添加了一个不等式约束。每个程序变体下方都展示了它们分别在第 *1*、*2*、*3* 次迭代以及最后一次迭代时的图表。

Basic:	*Uninstantiated:*	*Delayed constraint:*
$\mathfrak{p} = \begin{cases} \texttt{x += 1.} \\ \texttt{x += 0.5 * x.} \end{cases}$	$\mathfrak{p} = \begin{cases} \texttt{x(I) += 1.} \\ \texttt{x(I) += 0.5 * x(I).} \end{cases}$	$\mathfrak{p} = \begin{cases} \texttt{x(I) += (I > 10).} \\ \texttt{x(I) += 0.5 * x(I).} \end{cases}$
$\mathfrak{m}^{(1)} = \{\texttt{x += 1}\}$	$\mathfrak{m}^{(1)} = \{\texttt{x(I) += 1}\}$	$\mathfrak{m}^{(1)} = \{\texttt{x(I) += 1.0 * (I > 10)}\}$
$\mathfrak{m}^{(2)} = \{\texttt{x += 1.5}\}$	$\mathfrak{m}^{(2)} = \{\texttt{x(I) += 1.5}\}$	$\mathfrak{m}^{(2)} = \{\texttt{x(I) += 1.5 * (I > 10)}\}$
$\mathfrak{m}^{(3)} = \{\texttt{x += 1.75}\}$	$\mathfrak{m}^{(3)} = \{\texttt{x(I) += 1.75}\}$	$\mathfrak{m}^{(3)} = \{\texttt{x(I) += 1.75 * (I > 10)}\}$
$\vdots$	$\vdots$	$\vdots$
$\mathfrak{m}^{(\star)} = \{\texttt{x += 2}\}$	$\mathfrak{m}^{(\star)} = \{\texttt{x(I) += 2}\}$	$\mathfrak{m}^{(\star)} = \{\texttt{x(I) += 2 * (I > 10)}\}$

示例 49 (约束累积)。然而，我们的方法并不 *solve*

$$\mathfrak{p} = \begin{cases} \texttt{x(I) += (I > 0).} \\ \texttt{x(I) += 0.5 * (I > J) * x(J).} \end{cases}$$

经过五次迭代，我们看到约束只是在不断累积，

$$\mathfrak{m}^{(5)} = \begin{cases} \texttt{x(I) += (I > 0).} \\ \texttt{x(I) += 0.5 * (I > J) * (J > 0).} \\ \texttt{x(I) += 0.25 * (I > J) * (J > } J_1\texttt{) * (}J_1\texttt{ > 0).} \\ \texttt{x(I) += 0.125 * (I > J) * (J > } J_1\texttt{) * (}J_1\texttt{ > } J_2\texttt{) * (}J_2\texttt{ > 0).} \\ \texttt{x(I) += 0.0625 * (I > J) * (J > } J_1\texttt{) * (}J_1\texttt{ > } J_2\texttt{) * (}J_2\texttt{ > } J_3\texttt{) * (}J_3\texttt{ > 0).} \end{cases}$$

要解决这个程序，我们需要简化规则体。具体来说，我们需要一种方法，用于计算延迟约束的满足赋值数量，并投影头部变量上的任意约束。例如，要简化底部规则，我们需要一种高效机制来表示以下内容的解

644 多重性(I) += (I > J)* (J > J_1) * (J_1 > J_2) * (J_2 > J_3) * (J_3 > 0)

以一般方式高效地表示并推理不动点迭代期间遇到的多重性问题的解颇具挑战性。*SMT*求解器（例如 *Moura* 和 *Bjørner*（*2008*年））或许能够帮助处理某些约束域，例如算术不等式。我们认为这是一个有趣的未来工作方向。

示例 50。 考虑在最小加半环中计算图的自反传递闭包的程序：

645 路径(I, I)最小= 0646 路径(I, K) 最小= 路径(I, J) +边(J, K)。

请注意，顶层规则并非范围受限，也就是说，其头部包含一个自由变量。正如 *§5.1.3* 中所讨论的，这类自由变量会给.....带来问题

Eisner、*Goldlust* 等人（*2005*年）的前向链接算法。

*B*在.....下面，我们展示对以下输入数据进行的朴素不动点迭代：

```
647 边(a,b)最小= 1。 648 边
(b,c)最小= 1。 649 边(c,a)最
小= 1。
```

每次迭代的图表如下所示：

$$\mathfrak{m}^{(1)} = \begin{cases} \texttt{edge(a,b) min= 1.} \\ \texttt{edge(b,c) min= 1.} \\ \texttt{edge(c,a) min= 1.} \\ \texttt{path(I,I) min= 0.} \end{cases}$$

$$\mathfrak{m}^{(2)} = \begin{cases} \texttt{edge(a,b) min= 1.} \\ \texttt{edge(b,c) min= 1.} \\ \texttt{edge(c,a) min= 1.} \\ \texttt{path(I,I) min= 0.} \\ \texttt{path(a,b) min= 1.} \\ \texttt{path(b,c) min= 1.} \\ \texttt{path(c,a) min= 1.} \end{cases}$$

$$\mathfrak{m}^{(3)} = \begin{cases} \texttt{edge(a,b) min= 1.} \\ \texttt{edge(b,c) min= 1.} \\ \texttt{edge(c,a) min= 1.} \\ \texttt{path(I,I) min= 0.} \\ \texttt{path(a,b) min= 1.} \\ \texttt{path(b,c) min= 1.} \\ \texttt{path(c,a) min= 1.} \\ \texttt{path(a,c) min= 2.} \\ \texttt{path(b,a) min= 2.} \\ \texttt{path(c,b) min= 2.} \end{cases}$$

$$\mathfrak{m}^{(4)} = \begin{cases} \texttt{edge(a,b) min= 1.} \\ \texttt{edge(b,c) min= 1.} \\ \texttt{edge(c,a) min= 1.} \\ \texttt{path(I,I) min= 0.} \\ \texttt{path(a,b) min= 1.} \\ \texttt{path(b,c) min= 1.} \\ \texttt{path(c,a) min= 1.} \\ \texttt{path(a,c) min= 2.} \\ \texttt{path(b,a) min= 2.} \\ \texttt{path(c,b) min= 2.} \\ \texttt{path(a,a) min= 3.} \\ \texttt{path(b,b) min= 3.} \\ \texttt{path(c,c) min= 3.} \end{cases}$$

我们看到，不动点迭代经过 *4* 次迭代后收敛。这些图表值得注意之处在于，它们包含一条非基规则 `path(I,I) min= 0`。这意味着，例如，当我们查询 `path(a,a)` 的值时，$\mathfrak{m}^{(4)}$ 中的两条规则会共同决定其值，得到 $\min(0,3) = 0$。

附录 B

程序变换

B.1 强等价

命题 5. 如果 $\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$，那么 $\mathfrak{p}^{(s)} \equiv \mathfrak{p}^{(t)}$。

证明。 任取一个项 $\boldsymbol{x} \in \mathbb{H}$。

$$
\begin{aligned}
\mathfrak{p}^{(s)}[\![\boldsymbol{x}]\!] &= \bigoplus_{\boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\boldsymbol{x}\}} \mathfrak{p}^{(s)}\left[\!\!\left[\boldsymbol{\delta}^{(s)}\right]\!\!\right] && \text{[Definition 22]} && \text{(B.1)} \\
&= \bigoplus_{\boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\boldsymbol{x}\}} \mathfrak{p}^{(t)}\left[\!\!\left[\underset{s \to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)\right]\!\!\right] && \text{[value preserving]} && \text{(B.2)} \\
&= \bigoplus_{\boldsymbol{\delta}^{(t)} \in \mathfrak{p}^{(t)}\{\boldsymbol{x}\}} \mathfrak{p}^{(t)}\left[\!\!\left[\boldsymbol{\delta}^{(t)}\right]\!\!\right] && \text{[bijection]} && \text{(B.3)} \\
&= \mathfrak{p}^{(t)}[\![\boldsymbol{x}]\!] && \text{[Definition 22]} && \text{(B.4)}
\end{aligned}
$$

■

B.2 折叠变换与展开变换的正确性

本附录给出了若干结果，用以刻画 {基本、广义} × {展开、折叠} 变换的正确性。我们选择对广义展开的情形给出最详细的正确性证明，因为广义折叠变换的正确性证明采用相同的论证。此外，基本展开与折叠变换的正确性证明（定理 4）和定理 5）是广义展开证明的推论。广义展开变换的正确性证明由两个引理给出，即引理 1 和引理 4；二者共同构成主定理：定理 2。引理 1 给出了使推导映射 ϕ（由算法 24 提供）成为保值映射的一般化条件。$s \to t$随后，引理 4 证明了 ϕ（由算法 32 提供）是 ϕ 的逆$t \to s$。广义折叠变换也有类似的结果。$s \to t$提醒读者，这里给出的条件并不容易验证（§7.4.2.3）。§7.4.2.4 给出了一组实用充分条件。

概述。 下面给出结果概述。

	展开	fold
推导映射 (ϕ $_{s\to t}$)		
↳ 算法	算法24 算法25	
↳ ϕ $_{s\to t}$ 是一个保值映射	引理 1	引理 5
逆推导映射（ ϕ $_{t\to s}$ ）		
↳ 算法	算法31 算法32	
↳ ϕ $_{t\to s}$ 是 ϕ $_{s\to t}$ 的逆	引理 4	引理 6
语义保持		
↳ 基本	定理 4	定理 5
↳ 广义版（不实用）	定理 2	定理 3
↳ 广义化（实用）	定理 1	

补充定义。 我们的一些证明会用到以下推导上的序关系：

定义 43. 我们定义子树序关系。设 δ、$\delta' \in \mathfrak{p}\{\cdot\}$ 是推导（可能对应不同的项）。当且仅当 δ' 是 δ的（严格）子树时，我们称 $\delta' \prec_{\text{子树}} \delta$ 。

引理 1。假设

A1 $\mathfrak{p}^{(s)} \xrightarrow{\text{unfold}\ r\to\mathcal{S}}{}^{d,j} \mathfrak{p}^{(t)}$

A2 $\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(d)}$

A3 方法 ϕ : $\mathfrak{p}^{(s)}\{\cdot\} \to \mathfrak{p}^{(t)}\{\cdot\}$ 算法*24* 中指定的方法总是会终$_{s\to t}$止。

那么， ϕ 满足以下条件， $\forall \boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$: $_{s\to t}$

C1 $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$ *exists*

C2 $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right) \in \mathfrak{p}^{(t)}\{x\}$

C3 $\left[\!\left[\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right) \right]\!\right] = \left[\!\left[\boldsymbol{\delta}^{(s)} \right]\!\right]$.

其中 x 是 $\boldsymbol{\delta}^{(s)}$的根标签。

证明。我 们的正确性论证采用归纳法，按照 $\prec$ 步骤的序在.....上方 $\mathfrak{p}^{(s)}\{\cdot \qquad\}$.

↪ 由于 ϕ 依据 A3 会终止，而且它是一个无[ ]用的确定性过$s\to t$程，因此每次调用 $\underset{s\to t}{\phi}$ 所需的步数都是有限的。令 $\text{steps}\left\{\underset{s\to t}{\phi}(\boldsymbol{\delta})\right\}$表示在 ϕ 上运行 $\boldsymbol{\delta}$ 时所需的步数。序关系 $\prec$ steps 定义如下: $\boldsymbol{\delta}' \prec \text{steps}\,\boldsymbol{\delta} \iff \text{steps}\left\{\underset{s\to t}{\phi}(\boldsymbol{\delta}')\right\} < \text{steps}\left\{\underset{s\to t}{\phi}(\boldsymbol{\delta})\right\}$。我们可以假设每次函数调用都有正的开销，因此任何子例程调用的代价都不可能为零；这确保了 $\prec$ 是非自反的。

显然，如果 steps $\underset{s\to t}{\phi}(\boldsymbol{\delta})$ 递归调用 $\underset{s\to t}{\phi}(\boldsymbol{\delta}')$（即 $\underset{s\to t}{\phi}(\boldsymbol{\delta}')$ 是 $\underset{s\to t}{\phi}(\boldsymbol{\delta})$ 的子例程），并且 $\boldsymbol{\delta}' \prec \boldsymbol{\delta}$。此外， $\prec$ 在 steps 上构成良基序 $_{s\to t}$A3 成立，那么

[ ]

ϕ $(\cdot)$ 因为不存在无限的递降比较链。因此，$\prec_{s\to t}$ 可安全地用于采用诺特归纳原理的归纳证明步数归纳法（Cohn，1969）。

任意选择 $\boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$。

归纳假设：

IH $\forall \boldsymbol{\delta}' \underset{\text{steps}}{\prec} \boldsymbol{\delta}^{(s)}$: $\underset{s\to t}{\phi}(\boldsymbol{\delta}')$ exists, $\underset{s\to t}{\phi}(\boldsymbol{\delta}') \in \mathfrak{p}^{(t)}\{x\}$, and $[\![\boldsymbol{\delta}']\!] = \left[\!\!\left[\underset{s\to t}{\phi}(\boldsymbol{\delta}') \right]\!\!\right]$.

基本情况：（$\boldsymbol{\delta}^{(s)}$ **是常量或内置项**）：此情况显然成立。

C1 : $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right) \overset{\text{定义}}{=} \boldsymbol{\delta}^{(s)}$ C2: $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right) \in \mathfrak{p}^{(t)}\{\cdot\}$ 因为根据定义 23，常量和内置项始终包含在推导集合中。

□

C3 : $\left[\!\!\left[\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right) \right]\!\!\right] = [\![\boldsymbol{\delta}^{(s)}]\!]$ 根据赋值映射的定义（定义 22）。

归纳情形：需要考虑三种情况：

情形 1（$\boldsymbol{\delta}^{(s)}$ **是一个乘积** $\overset{y_1}{\triangle} \circ \cdots \circ \overset{y_K}{\triangle}$）：这种情形很简单。注意 $\overset{y_1}{\triangle}$，……，$\overset{y_K}{\triangle} \prec_{\text{步}} \boldsymbol{\delta}^{(s)}$，因为它们是子程序。

C1: $\underset{s\to t}{\phi}\left(\overset{y_1}{\triangle} \circ \cdots \circ \overset{y_K}{\triangle}\right) \overset{\text{def}}{=} \underset{s\to t}{\phi}\left(\overset{y_1}{\triangle}\right) \circ \cdots \circ \underset{s\to t}{\phi}\left(\overset{y_K}{\triangle}\right)$

C2:

$$\overset{y_1\circ\cdots\circ y_K}{\triangle} \in \mathfrak{p}^{(s)}\{y_1 \circ \cdots \circ y_K\} \Longrightarrow \overset{y_1}{\triangle} \in \mathfrak{p}^{(s)}\{y_1\}, \ldots, \overset{y_K}{\triangle} \in \mathfrak{p}^{(s)}\{y_K\}$$

$$\overset{\text{IH}}{\Longrightarrow} \underset{s\to t}{\phi}\left(\overset{y_1}{\triangle}\right) \in \mathfrak{p}^{(t)}\{y_1\}, \ldots, \underset{s\to t}{\phi}\left(\overset{y_K}{\triangle}\right) \in \mathfrak{p}^{(t)}\{y_K\}$$

$$\Longrightarrow \underset{s\to t}{\phi}\left(\overset{y_1}{\triangle} \circ \cdots \circ \overset{y_K}{\triangle}\right) \in \mathfrak{p}^{(t)}\{y_1 \circ \cdots \circ y_K\}$$

C3: $\left[\!\left[\phi_{s\to t}\left(y_1 \circ \cdots \circ y_K \right) \right]\!\right] = \left[\!\left[\phi_{s\to t}\left(y_1 \right) \circ \cdots \circ \phi_{s\to t}\left(y_K \right) \right]\!\right] \overset{\text{IH}}{=} \left[\!\left[y_1 \circ \cdots \circ y_K \right]\!\right]$

情况 2（$\boldsymbol{\delta}^{(s)}$ **的顶层规则在……里面** $\mathfrak{p}^{(t)}$）：这种情形很简单。注意到 $\mu \prec \boldsymbol{\delta}^{(s)}$ 因为它是一个子例程。步骤

C1: $\phi_{s\to t}\left(x(\mu) \right) \overset{\text{def}}{=} x\left(\phi_{s\to t}\left(\mu \right) \right)$

C2：由于 $\boldsymbol{\delta}^{(t)} = \phi_{s\to t}\left(\boldsymbol{\delta}^{(s)} \right)$ 的顶层规则是在 $\mathfrak{p}^{(t)}$里面的一条规则的特化，且其子树在 $\mathfrak{p}^{(t)}\{\cdot\}$ 里面，根据归纳假设，则 $\boldsymbol{\delta}^{(t)} \in \mathfrak{p}^{(t)}\{x\}$。□

C3: $\left[\!\left[\phi_{s\to t}\left(x(\mu) \right) \right]\!\right] = \left[\!\left[x\left(\phi_{s\to t}\left(\mu \right) \right) \right]\!\right] \overset{\text{IH}}{=} \left[\!\left[x(\mu) \right]\!\right] = \left[\!\left[\boldsymbol{\delta}^{(s)} \right]\!\right]$

情形3（$\boldsymbol{\delta}^{(s)}$ **的顶层规则不在** $\mathfrak{p}^{(t)}$里面）：根据定义35，推导 $\boldsymbol{\delta}^{(s)}$ 必须具有如下形式：

$\boldsymbol{\delta}^{(s)} = x(\alpha, y_j, \beta)$

现在我们展开推导 y_j：$\left\langle \mu, \mathbf{T}^{(\ell)} \right\rangle = u\left(y_j \right)$。这一步是为了让变换后的推导中（在……下面）的（不同的）替换规则$\mathbf{T}^{(\ell)} \in \mathcal{S}$ 生效所必需的。注意 α，μ，$\beta \prec \boldsymbol{\delta}^{(s)}$，因为它们是子程序步骤。

△ △ △

C1: $\phi_{s\to t}\left(x(\alpha, y_j, \beta) \right) \overset{\text{def}}{=} x\left(\phi_{s\to t}\left(\alpha \right), \phi_{s\to t}\left(\mu \right), \phi_{s\to t}\left(\beta \right) \right)$

C2：引理 2 保证了重映射树 $\overset{\mu}{\triangle} \in \mathfrak{p}^{(s)}\{\boldsymbol{\mu}\}$。此外，替换规则 $\mathbf{T}^{(\ell)} \in \mathcal{S}$ 是 $\mathfrak{p}^{(t)}$ 中的一条（不同的）规则，形式为 $\boldsymbol{x} \oplus\!= \boldsymbol{\alpha} \circ \boldsymbol{\mu} \circ \boldsymbol{\beta}$。现在，由于 $\underset{s\to t}{\phi}\left(\overset{\alpha}{\triangle}\right)$，$\underset{s\to t}{\phi}\left(\overset{\mu}{\triangle}\right)$，$\underset{s\to t}{\phi}\left(\overset{\beta}{\triangle}\right) \in \mathfrak{p}^{(t)}\{\boldsymbol{x}\}$ 根据归纳假设，可得 $\boldsymbol{\delta}^{(t)} \in \mathfrak{p}^{(t)}\{\boldsymbol{x}\}$。

□

C3：

$$
\begin{aligned}
&\left[\!\!\left[\underset{s\to t}{\phi}\left(\overset{x}{\overset{\alpha \quad y_j \quad \beta}{\triangle}} \right) \right]\!\!\right] \\
&= \left[\!\!\left[\underset{s\to t}{\phi}\left(\overset{\alpha}{\triangle}\right) \right]\!\!\right] \circ \left[\!\!\left[\underset{s\to t}{\phi}\left(u\left(\overset{y_j}{\triangle}\right) \right) \right]\!\!\right] \circ \left[\!\!\left[\underset{s\to t}{\phi}\left(\overset{\beta}{\triangle}\right) \right]\!\!\right] && \text{[由 □]} \\
&= \left[\!\!\left[\overset{\alpha}{\triangle} \right]\!\!\right] \circ \left[\!\!\left[u\left(\overset{y_j}{\triangle}\right) \right]\!\!\right] \circ \left[\!\!\left[\overset{\beta}{\triangle} \right]\!\!\right] && \text{[由 归□]} \\
&= \left[\!\!\left[\overset{\alpha}{\triangle} \right]\!\!\right] \circ \left[\!\!\left[\overset{y_j}{\triangle} \right]\!\!\right] \circ \left[\!\!\left[\overset{\beta}{\triangle} \right]\!\!\right] && \text{[由引理 2]} \\
&= \left[\!\!\left[\boldsymbol{\delta}^{(s)} \right]\!\!\right]
\end{aligned}
$$

■

引理 2。 假设 u 由算法 *24* 和 $\mathfrak{p}^{(s)}$ 展开 $\xrightarrow{d\ j\ \mathcal{S}}_{r}$ $\mathfrak{p}^{(t)}$，*and*

$$\boldsymbol{\delta}^{(s)} = \begin{array}{c} \boldsymbol{x} \\ \boldsymbol{\alpha}\quad \boldsymbol{y}_j\quad \boldsymbol{\beta} \\ \triangle\ \ \triangle\ \ \triangle \end{array}$$

其中 $\boldsymbol{\delta}^{(s)}$ 的顶层规则是待展开规则 r。那么，$\left\langle \begin{array}{c}\boldsymbol{\mu}\\ \triangle\end{array},\ \mathbf{T}^{(\ell)} \right\rangle \leftarrow u\left(\begin{array}{c}\boldsymbol{y}_j\\ \triangle\end{array}\right)$具有以下性质：

C1 $\begin{array}{c}\boldsymbol{\mu}\\ \triangle\end{array} \in \mathfrak{p}^{(s)}\{\cdot\}$

C2 $\left[\!\left[\begin{array}{c}\boldsymbol{\mu}\\ \triangle\end{array}\right]\!\right] = \left[\!\left[\begin{array}{c}\boldsymbol{y}_j\\ \triangle\end{array}\right]\!\right]$

C3 $\mathbf{T}^{(\ell)} \in \mathfrak{p}^{(t)}$ 和 $\mathbf{T}^{(\ell)}$ 具有以下形式 $\boldsymbol{x} \oplus\!= \boldsymbol{\alpha} \circ \boldsymbol{\mu} \circ \boldsymbol{\beta}$

证明略。 结论 C1 和 C2 是 $\mathfrak{p}(\square) \overset{\text{strong}}{\equiv} \mathfrak{p}(\square)$由调用上下文假设）。

C3: 根据展开变换（定义 35）的构造，我们知道 r 具有 $\boldsymbol{x} \oplus\!= \boldsymbol{\alpha} \circ \boldsymbol{y}_j \circ \boldsymbol{\beta}$的形式。通过检查定义 35，我们可以看到，规则 $\mathrm{T}^{(\ell)}$ 必须具有 $\boldsymbol{x} \oplus\!= \boldsymbol{\alpha} \circ \boldsymbol{\mu} \circ \boldsymbol{\beta}$ 的形式，其中某个 $\boldsymbol{\mu}$。该变换会将项 $\boldsymbol{y}_j$ 替换为 $\mathfrak{p}^{(d)}$中定义规则的各个主体。此外，当我们知道所使用的定义规则 $\mathrm{D}^{(\ell)}$ 时，就知道 $\mathfrak{p}^{(t)}$ 中的哪条规则替换了 r。具体而言，$\mathrm{T}^{(\ell)} = \underset{d\to t}{\sigma}(\mathrm{D}^{(\ell)})$使用一一对应 $\overset{\sigma}{}_{d\to t}$（定义 35 中对此进行了描述）。在展开创建 $\mathrm{T}^{(\ell)}$时，所有变量绑定都已经传播。

■

引理 3。 方法 u^{-1} 正确实现了 u的逆变换。

证明（略）。我们的论证思路是 u^{-1} 通过从 u 的输出反向推导u，并逐行依次反向还原。[220]

假设调用上下文包含 $\boldsymbol{\delta}^{(s)}$ 顶层规则为$\mathbf{T}^{(\ell)} \in \mathcal{S}$。因此，$\mathbf{T}^{(\ell)}$ 的形式为 $\boldsymbol{x} \oplus\!= \boldsymbol{\alpha} \circ \boldsymbol{\mu} \circ \boldsymbol{\beta}$，而展开规则 r 的形式为 $\boldsymbol{x} \oplus\!= \boldsymbol{\alpha} \circ \boldsymbol{y}_j \circ \boldsymbol{\beta}$。

反向还原 u 时唯一棘手之处在于，我们必须重建中间项$\boldsymbol{y}_j$，这是u在第 24.13 行抹去的。幸运的是，变换中包含的信息刚好足以根据$\mathbf{T}^{(\ell)}$的标识重建它。广义展开变换（定义 35）通过跟踪对齐$\overset{\sigma}{}_{d \to t}$，谨慎地确保了这一点能够实现。

第 25.15 行重建被抹去的项的方法如下：我们查找在广义展开变换（定义 35）期间用于创建 $\mathrm{T}^{(\ell)}$ 的不同展开器规则 $\mathrm{D}^{(\ell)} \in \mathfrak{p}^{(d)}$。我们有 $\mathrm{D}^{(\ell)} = \overset{\sigma}{}_{t \to d}(\mathrm{T}^{(\ell)})$。此外，$\mathrm{T}^{(\ell)}$ 必须具有形式 $\boldsymbol{y}' \oplus\!= \boldsymbol{\mu}'$。现在，必定存在一个中间子树（即展开操作在第 24.13 行抹去的子树），它由规则 $\mathrm{D}^{(\ell)}$ 的一个实例化构成。为了构建这个节点，我们确定使折叠器主体 $\boldsymbol{\mu}' \boldsymbol{\theta} = \boldsymbol{\mu}$ 的替换。我们可以确定一个唯一替换 $\boldsymbol{\theta} =$，使其覆盖 $\boldsymbol{\mu}'$ 和 $\boldsymbol{\mu}$。现在，我们可以确定缺失的中间项 $\boldsymbol{y}_j = \boldsymbol{\theta}(\boldsymbol{y}')$ 的名称 $\boldsymbol{y}_j$。

其余运算正确，这是基于以下假设：$\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(d)}$。■

[220]回想一下 如果 $h(x) = f(g(x))$，那么 $h^{-1}(x) = g^{-1}(f^{-1}(x))$（假设这些逆存在）。

算法 31 用于展开的逆推导映射 $\mathfrak{p}^{(s)} \xrightarrow{d,j \;\; r \to \mathcal{S}} \mathfrak{p}^{(t)}$。

1. ▷算法 *24* 顶部的注释同样适用于此处。
2. **定义** $\underset{t\to s}{\phi}\left(\boldsymbol{\delta}^{(t)}\right)$: ▷ 逆推导映射
3. **情况** 常量或内置项：**返回** $\boldsymbol{\delta}^{(t)}$
4. **case** $\overset{y_1}{\triangle}\circ\cdots\circ\overset{y_K}{\triangle}$：**返回** $\underset{t\to s}{\phi}\left(\overset{y_1}{\triangle}\right)\circ\cdots\circ\underset{t\to s}{\phi}\left(\overset{y_K}{\triangle}\right)$
5. **case** $\left(\begin{array}{c}x\\|\\\mu\end{array}\right)$ **并且** 其顶层规则 $\in \mathfrak{p}^{(s)}$：**返回** $\begin{array}{c}x\\|\\\underset{t\to s}{\phi}\left(\mu\right)\end{array}$
6. **case** $\left(\begin{array}{c}x\\\alpha\;\;\mu\;\;\beta\end{array}\right)$ **并且** 其顶层规则 $\notin \mathfrak{p}^{(s)}$: ▷ 顶层规则必须在 $\mathcal{S}$
7. **返回** $\begin{array}{c}x\\\underset{t\to s}{\phi}\left(\alpha\right)\quad u^{-1}\left(\underset{t\to s}{\phi}\left(\mu\right)\right)\quad\underset{t\to s}{\phi}\left(\beta\right)\end{array}$ ▷ 变换后的推导的顶层规则是 r

算法32 逆推导映射，用于 $\mathfrak{p}^{(s)}$ 折叠 $\xrightarrow{d\;j \;\; \mathcal{S}\to r} \mathfrak{p}^{(t)}$。

1. ▷下列位于算法 *24* 顶部的注释同样适用于此。
2. **定义** $\underset{t\to s}{\phi}\left(\boldsymbol{\delta}^{(t)}\right)$: ▷ 逆推导映射
3. **情况** 常量或内置项：**返回** $\boldsymbol{\delta}^{(t)}$
4. **case** $\overset{y_1}{\triangle}\circ\cdots\circ\overset{y_K}{\triangle}$：**返回** $\underset{t\to s}{\phi}\left(\overset{y_1}{\triangle}\right)\circ\cdots\circ\underset{t\to s}{\phi}\left(\overset{y_K}{\triangle}\right)$
5. **case** $\left(\begin{array}{c}x\\|\\\mu\end{array}\right)$ **并且** 其顶层规则 $\in \mathfrak{p}^{(s)}$：**返回** $\begin{array}{c}x\\|\\\underset{t\to s}{\phi}\left(\mu\right)\end{array}$
6. **case** $\left(\begin{array}{c}x\\\alpha\;\;y_j\;\;\beta\end{array}\right)$ **并且** 其顶层规则 $\notin \mathfrak{p}^{(s)}$: ▷ 顶层规则必须是 r
7. **返回** $\begin{array}{c}x\\\underset{t\to s}{\phi}\left(\alpha\right)\quad u\left(\underset{t\to s}{\phi}\left(y_j\right)\right)\quad\underset{t\to s}{\phi}\left(\beta\right)\end{array}$ ▷ 变换后推导的顶层规则由对 u的调用给出

引理 4. 设

A1 $\mathfrak{p}^{(s)} \xrightarrow{\text{unfold } {}^{d,j}_{r\to\mathcal{S}}} \mathfrak{p}^{(t)}$

A2 $\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(d)}$

A3 $\underset{s\to t}{\phi}$ ： $\mathfrak{p}^{(s)}\{\cdot\} \to \mathfrak{p}^{(t)}\{\cdot\}$ 是算法 [221] 中指定的部分*24*。

那么， $\forall \boldsymbol{\delta}^{(t)} \in \mathfrak{p}^{(t)}\{\cdot\}$ ： $\exists!\ \boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$ ： $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right) = \boldsymbol{\delta}^{(t)}$。此外，函数 $\underset{t\to s}{\phi}$ 在算法 *31* 中指定，计算 $\boldsymbol{\delta}^{(s)} = \underset{t\to s}{\phi}\left(\boldsymbol{\delta}^{(t)}\right)$。

证明。 我们的证明对 $\prec_{\text{子树}}$ 在 $\mathfrak{p}^{(t)}\{\cdot\}$上使用归纳法。[222] 固定任意一个 $\boldsymbol{\delta}^{(t)} \in \mathfrak{p}^{(t)}\{\cdot\}$.

归纳假设：

IH $\forall \boldsymbol{\delta}_1^{(t)} \underset{\text{subtree}}{\prec} \boldsymbol{\delta}^{(t)}$ ： $\exists! \boldsymbol{\delta}_1^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$ ： $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}_1^{(s)}\right) = \boldsymbol{\delta}_1^{(t)}$. **Furthermore,** $\boldsymbol{\delta}_1^{(s)} = \underset{t\to s}{\phi}\left(\boldsymbol{\delta}_1^{(t)}\right)$.

基本情况：**($\boldsymbol{\delta}^{(t)}$ 是一个常量或内置项)**：这种情况很简单。

我们要找出所有满足以下方程的 $\boldsymbol{\delta}^{(s)}$ ：

$$\boldsymbol{\delta}^{(t)} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$$

从 ϕ （参见 第 24.5 行）的实现可以清楚地看出， $\boldsymbol{\delta}^{(t)}$ 具有这种形式，当且仅当 $_{s\to t}$时

$$\boldsymbol{\delta}^{(t)} = \boldsymbol{\delta}^{(s)}$$

[221]这里，我们将给定输入上的不终止定义为函数在该输入上未定义。[222]我们能够在一个 $\prec_{\text{子树}}$序上进行归纳，而无需额外假设就能保证该序是良基的，这就是 ϕ 是一个全函数，而 $\underset{t\to s}{\phi}$ 仅 $_{s\to t}$能保证是一个偏函数的原因。引理1的证明中使用的归纳方案要求我们规定 ϕ 会终止，以便按$_{s\to t}$算法24执行的步数进行归纳。

这显然是一个唯一解。此外，$\boldsymbol{\delta}^{(s)} = \underset{t\to s}{\phi}\left(\boldsymbol{\delta}^{(t)}\right) = \boldsymbol{\delta}^{(t)}$。

归纳情形：正如预期的那样，ϕ 的构造本质上逆转了$t\to s$的每一步构造 ϕ。因此，与引理$s\to t$1 中一样，需要考虑三个类似的情形：

情形 1（$\boldsymbol{\delta}^{(t)}$ **是一个乘积** $\overset{y_1}{\triangle}\circ\cdots\circ\overset{y_K}{\triangle}$）：这种情形很简单

.

我们要找出所有满足下列方程的 $\boldsymbol{\delta}^{(s)}$：

$$\boldsymbol{\delta}^{(t)} = \overset{y_1}{\triangle}\circ\cdots\circ\overset{y_K}{\triangle} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$$

从 ϕ 的实现中可以清楚地看出（参见 第 24.6 行），$\boldsymbol{\delta}^{(t)}$ 具有这种$s\to t$形式当且仅当

$$\overset{y_1}{\triangle}\circ\cdots\circ\overset{y_K}{\triangle} = \underset{s\to t}{\phi}\left(\overset{y_1}{\triangle}\right)\circ\cdots\circ\underset{s\to t}{\phi}\left(\overset{y_K}{\triangle}\right)$$

现在，根据 归纳假设，对于每个 $k\in[1\colon K]\colon\ \exists!\ \boldsymbol{\delta}_k^{(s)}\colon\ \overset{y_k}{\triangle} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}_k^{(s)}\right)$；此外，$\boldsymbol{\delta}_k^{(s)} = \underset{t\to s}{\phi}\left(\overset{y_k}{\triangle}\right)$。因此，方程 $\boldsymbol{\delta}^{(t)} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$对 $\boldsymbol{\delta}^{(s)}$恰有一个解。此外，

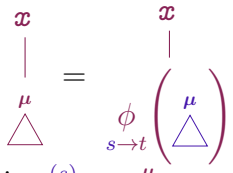
$$\boldsymbol{\delta}^{(s)} = \boldsymbol{\delta}_1^{(s)}\circ\cdots\circ\boldsymbol{\delta}_K^{(s)} = \underset{t\to s}{\phi}\left(\overset{y_1}{\triangle}\right)\circ\cdots\circ\underset{t\to s}{\phi}\left(\overset{y_K}{\triangle}\right) = \underset{t\to s}{\phi}\left(\boldsymbol{\delta}^{(t)}\right)$$

情况 2（$\boldsymbol{\delta}^{(t)}\in\mathfrak{p}^{(s)}$）：这种情况很简单。

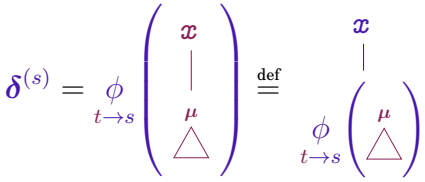
我们试图找出所有满足以下方程的 $\boldsymbol{\delta}^{(s)}$：

$$\boldsymbol{\delta}^{(t)} = \begin{array}{c}\boldsymbol{x}\\|\\\mu\\\triangle\end{array} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$$

从 ϕ 的实现（参见第24.7行）可以清楚地看出， $\boldsymbol{\delta}^{(t)}$ 具有这种$_{s\to t}$形式当且仅当

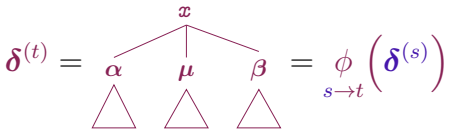


现在，根据归纳假设，我们有 $\exists!\ \boldsymbol{\delta}_1^{(s)}:\ \overset{\mu}{\triangle} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}_1^{(s)}\right)$；此外，还有 $\boldsymbol{\delta}_1^{(s)} = \underset{t\to s}{\phi}\left(\overset{\mu}{\triangle}\right)$。根据第 24.7 行，显然， $\boldsymbol{\delta}^{(s)}$ 和 $\boldsymbol{\delta}^{(t)}$中的顶层规则必定相同；因此，方程 $\boldsymbol{\delta}^{(t)} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$ 恰有一个解 $\boldsymbol{\delta}^{(s)}$。此外，



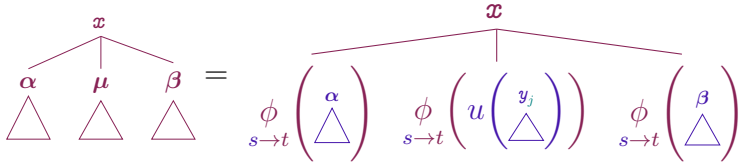
情形3（$\boldsymbol{\delta}^{(t)} \notin \mathfrak{p}^{(s)}$**的顶层规则**）：

我们要找出所有满足以下方程的 $\boldsymbol{\delta}^{(s)}$：



从 ϕ 的实现（参见第 24.9 行）显然可知， $\boldsymbol{\delta}^{(t)}$ 具有此$_{s\to t}$

形式当且仅当



根据[illegible]纳假设，以下方程有唯一解：

$$\underset{\triangle}{\boldsymbol{\alpha}} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}_0^{(s)}\right) \quad ; \quad \underset{\triangle}{\boldsymbol{\mu}} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}_1^{(s)}\right) \quad ; \quad \underset{\triangle}{\boldsymbol{\beta}} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}_2^{(s)}\right)$$

此外，这些解由以下内容给出 ϕ： $t\to s$

$$\underset{t\to s}{\phi}\left(\underset{\triangle}{\boldsymbol{\alpha}}\right) = \boldsymbol{\delta}_0^{(s)} \quad ; \quad \underset{t\to s}{\phi}\left(\underset{\triangle}{\boldsymbol{\mu}}\right) = \boldsymbol{\delta}_1^{(s)} \quad ; \quad \underset{t\to s}{\phi}\left(\underset{\triangle}{\boldsymbol{\beta}}\right) = \boldsymbol{\delta}_2^{(s)}$$

接下来，我们solve $\boldsymbol{\delta}_1^{(s)} = u\left(\boldsymbol{\delta}_3^{(s)}\right)$。根据引理 3，该方程有一个由 $\boldsymbol{\delta}_3^{(s)} = u^{-1}\left(\boldsymbol{\delta}_1^{(s)}\right)$ 和 第 25.7 行计算得到的唯一解。最后，我们有，唯一解 $\boldsymbol{\delta}^{(s)}$ 由下式给出

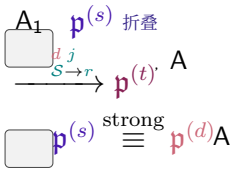
$$\boldsymbol{\delta}^{(s)} = \underset{t\to s}{\phi}\left(\begin{array}{c}\boldsymbol{x}\\ \boldsymbol{\alpha}\quad \boldsymbol{\mu}\quad \boldsymbol{\beta}\\ \triangle\ \ \triangle\ \ \triangle\end{array}\right) \overset{\text{def}}{=} \begin{array}{c}\boldsymbol{x}\\ \underset{t\to s}{\phi}\left(\underset{\triangle}{\boldsymbol{\alpha}}\right)\quad u^{-1}\left(\underset{t\to s}{\phi}\left(\underset{\triangle}{\boldsymbol{\mu}}\right)\right)\quad \underset{t\to s}{\phi}\left(\underset{\triangle}{\boldsymbol{\beta}}\right)\end{array}$$

■

定理 2。给定 $\mathfrak{p}^{(s)}$ 展开 $\xrightarrow[r \to \mathcal{S}]{d\ j} \mathfrak{p}^{(t)}$，以及 $\mathfrak{p}^{(s)}$ 强$\equiv$ $\mathfrak{p}^{(d)}$。假设函数 $\underset{s \to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$，正如在算法 24里面所规定的那样，对所有 $\boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$终止。则 $\underset{s \to t}{\phi}$ 是 $\mathfrak{p}^{(s)}\{\cdot\}$ 与 $\mathfrak{p}^{(t)}\{\cdot\}$ 之间的保值双射。此外，$\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$。证明。在上面提到的条件的下方，由引理 1和引理 4可知，ϕ 是 $\mathfrak{p}^{(s)}\{\cdot\}$与 $\mathfrak{p}^{(t)}\{\cdot\}$之间的保值双射。这样的双射$_{s \to t}$根据定义31 确保 $\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$ 。

■

引理 5. 假设



3 方法 ϕ ： $\mathfrak{p}^{(s)}\{\cdot\} \to \mathfrak{p}^{(t)}\{\cdot\}$ 算法中指定的*25* 总是终止$_{s\to t}$。

那么， ϕ 满足以下条件，$\forall \boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$: $_{s\to t}$

C1 $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$ *exists*

C2 $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right) \in \mathfrak{p}^{(t)}\{\boldsymbol{x}\}$

C3 $\left[\!\left[\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)\right]\!\right] = \left[\!\left[\boldsymbol{\delta}^{(s)}\right]\!\right]$.

其中 $\boldsymbol{x}$ 是 $\boldsymbol{\delta}^{(s)}$ 的根标签。

证明。 我们的正确性论证采用归纳法，按照我们在引理 $\prec_1$ 的证明中定义的 $\mathfrak{p}^{(s)}\{\cdot\}$步骤序1。

Choose $\boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$ arbitrarily.

归纳假设:

IH $\forall \boldsymbol{\delta}' \underset{\text{steps}}{\prec} \boldsymbol{\delta}^{(s)}$: $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}'\right)$ exists, $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}'\right) \in \mathfrak{p}^{(t)}\{\boldsymbol{x}\}$, and $[\![\boldsymbol{\delta}']\!] = \left[\!\left[\underset{s\to t}{\phi}\left(\boldsymbol{\delta}'\right)\right]\!\right]$.

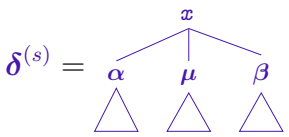
基本情况: （$\boldsymbol{\delta}^{(s)}$ **是常量或内置项**）： 此情况与引理 1 的证明完全相同。

归纳情形: 需要考虑三种情况:

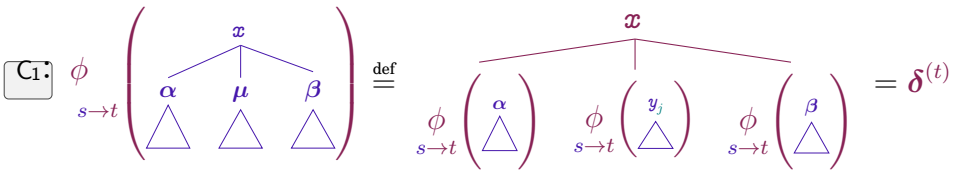
情形 1 （$\boldsymbol{\delta}^{(s)}$ **是一个乘积** $\overset{y_1}{\circ}\cdots\circ\overset{y_K}{\triangle}$）: 此情形与引理 1 的证明完全相同。

情况 2（**$\boldsymbol{\delta}^{(s)}$ 的顶层规则在……里面 $\mathfrak{p}^{(t)}$**）：此情形与引理 1 的证明完全相同。

情形3（**$\boldsymbol{\delta}^{(s)}$ 的顶层规则不在 $\mathfrak{p}^{(t)}$**）：推导 $\boldsymbol{\delta}^{(s)}$ 必须具有以下形式，顶层规则为$\mathbf{S}^{(\ell)} \in \mathcal{S}$:



现在折叠推导 $\overset{\mu}{\triangle}$：$u^{-1}\left(\begin{array}{c}\mu\\ \triangle\\ \triangle\end{array}, \mathbf{S}^{(\ell)}\right) = \overset{y_j}{\triangle}$。注意，$\overset{\alpha}{\triangle}$，$\overset{y_j}{\triangle}$，$\overset{\beta}{\triangle}$ $\prec_{\text{步骤}} \boldsymbol{\delta}^{(s)}$，因为它们是子程序。

C1: 

C2：引理3 保证重映射树 $\overset{y_j}{\triangle} \in \mathfrak{p}^{(s)}\{y_j\}$。此外，折叠规则 r 是 $\mathfrak{p}^{(t)}$ 中的一个（不同的）规则，形式为 $\boldsymbol{x} \oplus\!= \boldsymbol{\alpha} \circ y_j \circ \boldsymbol{\beta}$。现在，由于 $\underset{s\to t}{\phi}\left(\overset{\alpha}{\triangle}\right)$，$\underset{s\to t}{\phi}\left(\overset{y_j}{\triangle}\right)$，$\underset{s\to t}{\phi}\left(\overset{\beta}{\triangle}\right) \in \mathfrak{p}^{(t)}\{\boldsymbol{x}\}$ 根据归纳假设，可得$\boldsymbol{\delta}^{(t)} \in \mathfrak{p}^{(t)}\{\boldsymbol{x}\}$。 □

C_3:

$$\left[\!\!\left[\underset{s\to t}{\phi}\left(\begin{array}{ccc} & \boldsymbol{x} & \\ \boldsymbol{\alpha} & \boldsymbol{\mu} & \boldsymbol{\beta} \\ \triangle & \triangle & \triangle \end{array} \right) \right]\!\!\right]$$

$$= \left[\!\!\left[\underset{s\to t}{\phi}\left(\begin{array}{c} \boldsymbol{\alpha} \\ \triangle \end{array} \right) \right]\!\!\right] \circ \left[\!\!\left[\underset{s\to t}{\phi}\left(u^{-1}\left(\begin{array}{c} \boldsymbol{\mu} \\ \triangle \end{array} \right) \right) \right]\!\!\right] \circ \left[\!\!\left[\underset{s\to t}{\phi}\left(\begin{array}{c} \boldsymbol{\beta} \\ \triangle \end{array} \right) \right]\!\!\right] \qquad \text{[由]}$$

$$= \left[\!\!\left[\begin{array}{c} \boldsymbol{\alpha} \\ \triangle \end{array} \right]\!\!\right] \circ \left[\!\!\left[u^{-1}\left(\begin{array}{c} \boldsymbol{\mu} \\ \triangle \end{array} \right) \right]\!\!\right] \circ \left[\!\!\left[\begin{array}{c} \boldsymbol{\beta} \\ \triangle \end{array} \right]\!\!\right] \qquad \text{[由归纳]}$$

$$= \left[\!\!\left[\begin{array}{c} \boldsymbol{\alpha} \\ \triangle \end{array} \right]\!\!\right] \circ \left[\!\!\left[\begin{array}{c} y_j \\ \triangle \end{array} \right]\!\!\right] \circ \left[\!\!\left[\begin{array}{c} \boldsymbol{\beta} \\ \triangle \end{array} \right]\!\!\right] \qquad \text{[根据引理 3]}$$

$$= \left[\!\!\left[\boldsymbol{\delta}^{(s)} \right]\!\!\right]$$

■

引理 6. 假设

A1 $\mathfrak{p}^{(s)} \xrightarrow{\text{fold } \mathcal{S}\to r}{}^{d,j} \mathfrak{p}^{(t)}$

A2 $\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(d)}$

A3 ϕ： $\mathfrak{p}^{(s)}\{\cdot\} \to \mathfrak{p}^{(t)}\{\cdot\}$ 是算法 *25* 中指定的偏函数。$_{s\to t}$

那么， $\forall \boldsymbol{\delta}^{(t)} \in \mathfrak{p}^{(t)}\{\cdot\}$： $\exists!\ \boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$： $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right) = \boldsymbol{\delta}^{(t)}$。此外，函数 $\underset{t\to s}{\phi}$ 算法 *32* 计算 $\boldsymbol{\delta}^{(s)} = \underset{t\to s}{\phi}\left(\boldsymbol{\delta}^{(t)}\right)$。

证明。 我们的证明使用在 $\prec_{\text{子树}}$ 上方 $\mathfrak{p}^{(t)}\{\cdot\}$ 上的归纳法。本证明中除一个情形外，其余所有情形都与引理 4 相同。任取一个 $\boldsymbol{\delta}^{(t)} \in \mathfrak{p}^{(t)}\{\cdot\}$。

归纳假设：

IH $\forall \boldsymbol{\delta}_1^{(t)} \underset{\text{subtree}}{\prec} \boldsymbol{\delta}^{(t)}$: $\exists! \boldsymbol{\delta}_1^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$: $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}_1^{(s)}\right) = \boldsymbol{\delta}_1^{(t)}$. **Furthermore,** $\boldsymbol{\delta}_1^{(s)} = \underset{t\to s}{\phi}\left(\boldsymbol{\delta}_1^{(t)}\right)$.

基本情况：**($\boldsymbol{\delta}^{(t)}$ 是常量或内置项)**：与引理4的证明相同。

归纳情形：不出所料， ϕ 本质上逆转了$_{t\to s}$构造 ϕ 的每一步。因此，与引理$_{s\to t}$中一样，有三个类似的情形需要考虑：5：

情形 1 **($\boldsymbol{\delta}^{(t)}$ 是一个乘积** $\overset{y_1}{\triangle}\circ\cdots\circ\overset{y_K}{\triangle}$**)**：与引理 4的证明相同。

情况 2 **($\boldsymbol{\delta}^{(t)} \in \mathfrak{p}^{(s)}$的顶层规则)**：与引理 4的证明相同。

情形 3 **($\boldsymbol{\delta}^{(t)} \notin \mathfrak{p}^{(s)}$的顶层规则**：

我们试图找出所有 $\boldsymbol{\delta}^{(s)}$ 满足以下方程的项：

$$\boldsymbol{\delta}^{(t)} = \begin{array}{c} x \\ \alpha \quad y_j \quad \beta \\ \triangle \quad \triangle \quad \triangle \end{array} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$$

从 ϕ 的实现（参见 第 25.7 行）可清楚看出，$\boldsymbol{\delta}^{(t)}$ 具有这种$_{s\to t}$形式当且仅当

$$\begin{array}{c} x \\ \alpha \quad y_j \quad \beta \\ \triangle \quad \triangle \quad \triangle \end{array} = \begin{array}{c} x \\ \underset{s\to t}{\phi}\left(\alpha\right) \quad \underset{s\to t}{\phi}\left(u^{-1}\left(\mu\right)\right) \quad \underset{s\to t}{\phi}\left(\beta\right) \\ \triangle \quad \triangle \quad \triangle \end{array}$$

根据[illegible]纳假设，以下方程组有唯一解：

$$\underset{\triangle}{\alpha} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}_0^{(s)}\right) \quad ; \quad \underset{\triangle}{y_j} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}_1^{(s)}\right) \quad ; \quad \underset{\triangle}{\beta} = \underset{s\to t}{\phi}\left(\boldsymbol{\delta}_2^{(s)}\right)$$

此外，这些解由 ϕ 给出：$_{t\to s}$

$$\underset{t\to s}{\phi}\left(\underset{\triangle}{\alpha}\right) = \boldsymbol{\delta}_0^{(s)} \quad ; \quad \underset{t\to s}{\phi}\left(\underset{\triangle}{y_j}\right) = \boldsymbol{\delta}_1^{(s)} \quad ; \quad \underset{t\to s}{\phi}\left(\underset{\triangle}{\beta}\right) = \boldsymbol{\delta}_2^{(s)}$$

接下来，我们 solve $\boldsymbol{\delta}_1^{(s)} = u^{-1}\left(\boldsymbol{\delta}_3^{(s)}\right)$。根据引理3，该方程具有由 $\boldsymbol{\delta}_3^{(s)} = u^{-1}\left(\boldsymbol{\delta}_1^{(s)}\right)$ 和 第 25.7 行计算得到的唯一解。最后，唯一解 $\boldsymbol{\delta}^{(s)}$ 由下式给出

$$\boldsymbol{\delta}^{(s)} = \underset{t\to s}{\phi}\left(\begin{array}{c} x \\ \alpha \quad y_j \quad \beta \\ \triangle \quad \triangle \quad \triangle \end{array}\right) \overset{\text{def}}{=} \begin{array}{c} x \\ \underset{t\to s}{\phi}\left(\alpha\right) \quad u\left(\underset{t\to s}{\phi}\left(y_j\right)\right) \quad \underset{t\to s}{\phi}\left(\beta\right) \\ \triangle \quad \triangle \quad \triangle \end{array}$$

■

定理 3. 给定 $\mathfrak{p}^{(s)}$ 折叠 $\xrightarrow[\mathcal{S}\to r]{d\ j}$ $\mathfrak{p}^{(t)}$，以及 $\mathfrak{p}^{(s)}$ 强≡ $\mathfrak{p}^{(d)}$。假设函数 $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$，依照算法 *25*的规定对所有 $\boldsymbol{\delta}^{(s)} \in \mathfrak{p}^{(s)}\{\cdot\}$终止。则 ϕ 是一个值-$_{s\to t}$保持双射，介于 $\mathfrak{p}^{(s)}\{\cdot\}$ 与 $\mathfrak{p}^{(t)}\{\cdot\}$之间。此外，$\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$。

证明。 在上述条件下，由引理 5 和引理 6可知ϕ 是 $\mathfrak{p}^{(s)}\{\cdot\}$与 $\mathfrak{p}^{(t)}\{\cdot\}$之间的保值双射$_{s\to t}$，该双射确保 $\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$ 符合定义 31。

■

定理 4。 假设 $\mathfrak{p}^{(s)} \xrightarrow{\text{展开}} \mathfrak{p}^{(t)}$ 符合定义 *33*，则 $\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$。证明。 任何基本 展开 变换都是广义展开变换的实例，其中 $\mathfrak{p}^{(d)} = \mathfrak{p}^{(s)}$。因此，它继承了广义展开 变换的正确性证明（定理2）。我们可以加强这一结论：所有基本展开 变换都保持语义，因为我们可以去掉定理2中关于推导映射（算法 24）会终止的条件，因为对于基本展开变换，该条件总是成立。为说明原因，我们注意到，对 u 的调用总是返回的一个（严格）子树，而子树构成良基序。因此， ϕ 保证会对任意基本展开$_{s \to t}$变换终止。
■

定理 5。 假设 $\mathfrak{p}^{(s)} \xrightarrow{\text{折叠}} \mathfrak{p}^{(t)}$ 根据定义 *34*，则 $\mathfrak{p}^{(s)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(t)}$。

证明。 定理 4 确立了一个对称的结果，我们可以在此利用它。定义 34 要求具备可逆性： $\mathfrak{p}^{(t)} \xrightarrow[r \to \mathcal{S}]{\text{展开}}{}^{t\ j} \mathfrak{p}^{(s)'}$。由于 $\mathfrak{p}^{(t)} \xrightarrow[r \to \mathcal{S}]{\text{展开}}{}^{t\ j} \mathfrak{p}^{(s)'}$ 意味着 $\mathfrak{p}^{(t)} \overset{\text{strong}}{\equiv} \mathfrak{p}^{(s)}$ (根据定理 4)，且语义等价具有对称性，因此可得 $\text{折叠}_{\mathcal{S} \to r}{}^{s\ j}$，也具有语义保持性，因为 任何 从 $\mathfrak{p}^{(s)}$ 到 $\mathfrak{p}^{(t)}$ (或反过来) 的变换，只要有一个强等价变换将其关联起来，就会保持语义 $\mathfrak{p}^{(t)}$ 和 $\mathfrak{p}^{(s)}$。 ■

B.3 推测

我们提供了一个推导映射，作为推测变换（定义 41）正确性的（不完备）论证。要完成正确性证明，我们需要证明这个推导映射是一个双射保值映射（正如我们在 §B.2 中对折叠和展开进行正确性证明时所做的那样）。通过结构归纳法（基于 子树关系）进行论证应该相当直接；不过，我们并未给出这样的论证。

算法 33 推测的推导映射 $\mathfrak{p}^{(s)}$ 推测 $\xrightarrow{z\ \sigma}$ $\mathfrak{p}^{(t)}$。

定义 $\underset{s\to t}{\phi}\left(\boldsymbol{\delta}^{(s)}\right)$：

  若 $\boldsymbol{\delta}^{(s)}$ 是内置项或常量：**返回** $\boldsymbol{\delta}^{(s)}$

  case $y_1 \circ \cdots \circ y_K$：**返回** $\underset{s\to t}{\phi}\left(y_1\right) \circ \cdots \circ \underset{s\to t}{\phi}\left(y_K\right)$

  case $\left(x \mid \mu\right)$ **并且**顶层规则 $\notin \boldsymbol{\sigma}$： ▷ 脊柱底部

    返回 $\widetilde{x}\left(\underset{s\to t}{\phi}\left(\mu\right)\right),\ x/x\,(1)$ **如果** $x \subseteq z$ **否则** $\widetilde{x}\left(\underset{s\to t}{\phi}\left(\mu\right)\right)$

  case $\left(x\left(y, \beta\right)\right)$ **和** 顶层规则 $\in \boldsymbol{\sigma}$：

    匹配 $\underset{s\to t}{\phi}\left(y\right)$：

      case $\widetilde{y}$：**返回** $\widetilde{x}\left(\widetilde{z},\ \underset{s\to t}{\phi}\left(\beta\right)\right)$

      case $\widetilde{z}, y/z$：**回报** $x\left(\widetilde{z},\ x/z\left(y/z,\ \underset{s\to t}{\phi}\left(\beta\right)\right)\right)$

附录 C

模式匹配算法

C.1 合一

本节将介绍合一算法；我们在 §3.1.1中对其进行了简要描述。[223] 关于合一的论述很多（Herbrand，1930年；Robinson，1965年；Knight，1989年；Norvig，1991年；Martelli 和 Montanari，1982），因此我们不会过多讨论该算法——之所以纳入它，主要是为了完整性。我们对显式合一 unify 的实现见算法 34；它严格遵循 Russell 和 Norvig 的教科书（2020年）中的伪代码。总体而言，该算法是一个相对直接的递归过程：它同时自顶向下遍历一对项，获取使这些项相等所需的最小变量绑定集合（如果无法实现相等，则失败）。算法 35 给出了一种“单

[223]本节的实现使用显式替换映射，而在 §3.1.2.1 中，该映射是通过可变数据结构隐式表示的。

边”合一算法（覆盖），用于解决形如$\theta(x) = y$的问题。该算法可用于确定由 x表示的项集是否是由 y表示的项集的超集：$(x \supseteq y) \iff$ (覆盖$(x, y) \neq$ 失败）。在伪代码中，解引用*(x, θ) 会递归应用 $\theta(x)$，直到 $x \notin \theta$。

算法34 合一：合一算法。求解以下形式的任意方程 $\theta(x) = \theta(y)$，其中 θ 给定的可能为非基项的项 x 和 y。

```
定义 合一(x, y, θ = {}):
    如果 θ = 失败: 返回 失败
    否则，如果 x = y: 返回 θ
    否则，如果 x是一个变量:
        如果 x ∈ θ: 返回 合一(θ[x], y, θ)
        否则，如果 y ∈ θ: 返回 合一(x, θ[y], θ)
        否则，如果 x在.....里面出现于 y替换之后: 返回 失败
        else
            θ[x] ← y                                  ▷ 进行变更 θ
            返回 θ
    否则，如果 y是一个变量: 返回 合一(y, x, θ)
    否则如果 x和 y是名称和元数相同的项:
        对于 i ∈ 1 ... |x|: θ← 合一(x_i, y_i, θ)
        返回 θ
    else
        返回 失败
```

算法 35 覆盖：单向合一算法。对任何形式为 $\theta(x) = y$ 的方程，在给定可能为非基项的项 x 和 y的情况下，求解 θ 。

```
1. def 覆盖(x, y, θ = {}): 2. x ← 解引用*(x, θ); y ← 解引用*(y, θ) 3. if x = y:
return θ4. else if x是一个变量： 5. if x ∈ θ: 6. if θ[x] = y: return θ7. else if
x不出现在 θ(y)中： 8.
θ[x] ← y                                              ▷ 变异 θ9. return θ
10. return 失败 11. else if x和 y是名称和元数相同的项： 12. for i ∈ 1 ... |x|:
θ ← 覆盖(x_i, y_i, θ) return θ
```

C.2 子乘积枚举

C.2.1 交换子乘积

算法36 是一个过程，用于枚举一个子目标集合与另一个子目标集合之间的对齐 $x_1 \circ \cdots \circ x_N \quad y_1 \circ \cdots \circ y_M$ 假设 a 是交换的。主要用途是在折叠过程中，其中 $x_1 \circ \cdots \circ x_N$ 是折叠规则的右侧，而 $y_1 \circ \cdots \circ y_M$ 是被折叠的规则的右侧。例如，。

- 覆盖_乘积(f(X) * f(Y),f(A) * f(B) * f(C)) 有六种对齐：

$$\langle \{\mathtt{X} \mapsto \mathtt{A}, \mathtt{Y} \mapsto \mathtt{B}\}, \{0 \mapsto 0, 1 \mapsto 1\}\rangle$$
$$\langle \{\mathtt{X} \mapsto \mathtt{A}, \mathtt{Y} \mapsto \mathtt{C}\}, \{0 \mapsto 0, 1 \mapsto 2\}\rangle$$
$$\langle \{\mathtt{X} \mapsto \mathtt{B}, \mathtt{Y} \mapsto \mathtt{A}\}, \{0 \mapsto 1, 1 \mapsto 0\}\rangle$$
$$\langle \{\mathtt{X} \mapsto \mathtt{B}, \mathtt{Y} \mapsto \mathtt{C}\}, \{0 \mapsto 1, 1 \mapsto 2\}\rangle$$
$$\langle \{\mathtt{X} \mapsto \mathtt{C}, \mathtt{Y} \mapsto \mathtt{A}\}, \{0 \mapsto 2, 1 \mapsto 0\}\rangle$$
$$\langle \{\mathtt{X} \mapsto \mathtt{C}, \mathtt{Y} \mapsto \mathtt{B}\}, \{0 \mapsto 2, 1 \mapsto 1\}\rangle$$

- 覆盖_乘积(f(X,Y), g(Y,Z),f(I,J), g(J,I)) 有一种对齐：

$$\langle \{\mathtt{X} \mapsto \mathtt{I}, \mathtt{Y} \mapsto \mathtt{J}, \mathtt{Z} \mapsto \mathtt{K}\}, \{0 \mapsto 0, 1 \mapsto 1\}\rangle$$

每次迭代都是一个由替换映射 θ 和索引对齐σ组成的有序对。替换映射 θ 和对齐 σ 满足

1. 每个对齐 σ都是一个函数 $[1{:}N] \to [1{:}M]$；换句话说，它从 x 的索引映射到 y 的索引。

2. $\theta(x_i) = y_{\sigma(i)}$ 对于 $i \in [1{:}\ N]$；与 覆盖 类似， θ 只能绑定 x中的变量。

算法 36 覆盖_乘积：从一个.....惰性枚举交换匹配子目标集合 $x_1 \circ \cdots \circ x_N$ 到另一个子目标集合 $y_1 \circ \cdots \circ y_M$。每个迭代项是一个替换映射和索引对齐组成的有序对。

```
定义覆盖_乘积(x_1 ∘ ⋯ ∘ x_N, y_1 ∘ ⋯ ∘ y_M, θ_0 = {}):
  def f(i, β, θ):
    如果 θ = 失败: 返回                                          ▷ 失败
    否则如果 |x_[i:]| > |y_β|: 返回                              ▷ 失败：子目标过多
    else if |x_[i:]| = 0: yield ⟨θ, {}⟩                          ▷ 成功：已到达 x的末尾
    else
      对于 j ∈ β:
        对于 ⟨θ', σ'⟩ ∈ f(i + 1, β ∖ {j}, 覆盖(x_i, y_j, 复制(θ))):
          yield ⟨θ', σ' ∪ (i → j)⟩
  yield 自 f（1, {1, ..., M}, θ_0）
```

C.2.2 不可交换子乘积

算法 37是一个用于枚举子乘积的过程；它会找出所有包含在 $x_1 \circ \cdots \circ x_N$ 中的 $y_1 \circ \cdots \circ y_M$子乘积。与交换情形不同，匹配中不允许重新排序或出现间隙。

算法37 覆盖_乘积：从一个子目标集合中惰性枚举到另一个子目标集合的不可交换匹配 $x_1 \circ \cdots \circ x_N \circ$ $y_1 \circ \cdots \circ y_M$每次迭代的结果都是一个替换映射与匹配起始位置构成的有序对。

```
def 覆盖_product(x_1 ∘ ⋯ ∘ x_N, y_1 ∘ ⋯ ∘ y_M, θ = {}):
for j ∈ [1:M − N]: θ' ← 覆盖(x, y_[j+i:j+N], copy(θ))
if θ' ≠ FAIL: yield ⟨θ', j⟩
```

附录 D

Dyna 2

我们简要概述 Dyna 2语言，并探讨如何将本博士论文的结果扩展到该语言。

Dyna 2 (Eisner 和 Filardo，2011年）扩展了 Dyna 1（参见 §2），允许以单个半环所支持的两种方式之外的更多方式组合值。因此，Dyna 2 程序可以作为一种简洁的符号表示，用于指定一般的计算图[224]，其规则仍采用类Prolog记法书写（与 Dyna 1 一样）。每条规则仍然是一个模板，其中的 变量 可以用任意项进行实例化。规则实例化共同定义了许多项的值所对应的贡献。这些贡献通过一个聚合器进行聚合算子（聚合器，简称）聚合，方式与 Dyna 1 中相同。Dyna 程序支持一般的数学表达式（而不仅仅是半环运算）。前馈神经网络及其训练目标可以通过以下方式定义：

[224]用于定义计算图的其他符号表示法和库包括 Theano（Theano开发团队，2016）、PyTorch（Paszke 等，2017年）、TensorFlow（Abadi 等，2016）、JAX（Bradbury 等，2018）以及命名张量表示法（Chiang、Rush 等，2021年）。

```
684 σ(X) = 1/(1+exp(-X)). % 定义 sigmoid函数 685 out(J) = σ
(in(J)). % 应用 sigmoid函数 686 in(J) +=out(I) * edge(I,J). % 向量-
矩阵乘积 687 loss += (out(J) - target(J))^2. % L2损失
```

网络的结构和权重通过定义以下形式的项的值来指定：`edge(I, J)`。这可以通过为每条边列出一条规则来完成，但也可以通过根据结构化节点名称编写边定义规则，以系统地完成，其中这些名称将实例化上面的 `I`、`J` 。例如：

```
688 边(输入(X,Y),隐藏层(X+DX,Y+DY)) =权重(DX,DY)。 689 权重(DX,DY) := 随机
(*,-1,1) 对于 DX:-4..4, DY:-4..4。
```

定义了一个初始权重为随机值的卷积层[225] 和一个 9×9 卷积滤波器。例如，名为 `hidden(10,10)` 的节点与名为`input(8,11)` 的节点之间有一条连接，权重为`weight(-2,1)`。网络的输入在运行时通过更新`out(input(...))`的值来指定；否则，这些项没有值，因为输入节点没有入边。

Dyna求解器寻求一个不动点，其中所有项都保持一致，即每个项的值都与其根据其他值给出的定义相匹配。循环程序可能没有唯一的不动点。如果存在多个不动点，求解器可以任意选择一个。一般来说，求解器可能无法终止，因为不存在不动点，或者因为它无法发现不动点。有时，求解器只能在数值收敛时终止，例如几何级数求和`690 a += 1。 691 a +=a`

```
/2.
```

或者这个一个实现巴比伦方法以进行数值计算的程序 y

[225]每个表达式`random(*,-1,1)` 都命名了一个不同的随机变数，因为特殊参数`*` 会生成一个符号（“生成符号”），该符号在规则的每次实例化中都不同。因此，最后一行定义了 81 个不同的权重。

平方根近似：$y = \sqrt{x}$：

```
692 y := 1. % 任意初始值 ≠ 0693 y := 1/2 *(x/y + y).694
x:= 2.
```

这里的 : = 聚合器是“最后一条规则优先”聚合器。顶层规则指定了不动点迭代的初始化。

Eisner 和 Filardo（2011年）通过更复杂的示例，对 Dyna 2 作了更为完备的介绍，其中包括变分推断和图模型中的消息传递、回溯搜索中的约束传播、反向传播以及马尔可夫链蒙特卡洛。

将这篇博士论文扩展至 Dyna 2

执行。 §5 中的朴素、半朴素和优先级不动点迭代算法，在 Dyna 2 设定中也有对应形式。Filardo（2017年）以及 Eisner and Blatz（2007）在一定程度上研究了这些算法。[226] 在存在半环子程序的设定中，我们的任何执行算法都可用于求解该子程序。尽管如此，Dyna 2 框架似乎正朝着以项重写系统作为其执行系统基础的方向发展（Francis-Landau 等人，2020年）。

类型分析。 我们的类型分析器 §6.1 可以扩展到 Dyna 2。主要区别在于，我们需要为某个项可能具有的值 证明其类型，因为它不再是同质的。换句话说，我们将

[226] Eisner and Blatz（2007）介绍了混合半环上的前向链，这涵盖了大多数但并非所有 Dyna2 程序。

讨论可能的值–项有序对。原则上，这种扩展应该并不困难。我们用于时空复杂度分析的方法（§6.3）也应适用于 Dyna 2，因为执行 Dyna 2 程序的复杂度同样与实例化其规则的代价成正比。我们的程序特化方法（§6.2）稍作修改后也应适用。

程序变换。 尽管 Eisner and Blatz（2007）早于 Dyna 2，但他们的方法颇有远见。他们设计了可应用于混合半环程序的半环变换。诀窍很简单：仔细检查程序片段中是否存在局部分配律。本质上，我们的变换仍应有效，但需要检查局部分配性。我们还注意到，Dyna 2程序需要其他类型的变换，这些变换运用了更广泛的数学重写。这类变换也用于其他计算图库。

通过变换搜索进行程序优化。 我们的变换搜索方法可直接应用于针对 Dyna 2 程序定制的变换搜索空间。

附录 E

实验所用程序

E.1 链（chain-05、chain-10、bad-chain-05、bad-chain-10） . . . . . 275E.2 投射依存句法分析.276E.2.1 双词汇无标注（bilexical-unlabeled）.276E.2.2 双词汇有标注（bilexical-labeled）.277E.2.3 弧先行（arc-eager）.277E.2.4 分裂式双词汇依存句法分析.278E.2.4.1 Johnson（2007）版本（split-head-J） . . . . . 278E.2.4.2 Eisner 和 Blatz （ 2007 ）版本（ split-head-EB ） 278E.3 成分句法分析.279

E.3.1 CKY（cky3）.279E.3.2 使用四元产生式的 CKY（cky4）.279E.3.3 使用固定文法的 CKY（cky+grammar） . . . . . 280张量分解分析器（tensor-decomp-parser） . . . 281

E.4 反向转写文法（itg）.281

E.5 编辑距离（edit）.282E.6 Bar-Hillel 构造（bar-hillel）.282E.7 线性链条件随机场下的期望（chain-expect）.283E.8 偶数和奇数 Peano 数的交集（even-odd-peano）. 284E.9 隐马尔可夫模型（hmm）.285E.10 半马尔可夫模型（semi-markov）.285E.11 图中的最短路径.286E.11.1 显式路径（path-list）.286E.11.2 起始路径（path-start）.286E.12 带显式栈的下推自动机（explicit-pda） 287

E.1 链（chain-05、chain-10、bad-chain- 05、bad-chain-10）

链结构在自然语言处理中很常见，因为它们用于建模序列中相邻单词或标签之间的交互（例如，Lafferty 等人（2001年））。chain-N 程序是示例3的改编版本，

```
1 目标 += w(X1,X2) *w(X2,X3) * w(X3,X4) * w(X4,X5)。
```

该示例针对长度为 4 的特定情况，使长度为 N。原程序的度为 N。无论 N 如何，链的最优度数都是 2。

我们还来看以下变量消除顺序不佳的示例 s

§7.3。下面这两个示例都可以加速到 2 度。

bad-chain-05:

1 目标 += gen6(X5) * 临时项(X1,X_4,X_5). 2 临时项(X_1,X_4,X_5) += gen2(X_1,X_4) * w(X_4，X_5). 3 gen2(X_1,X_4) += gen5(X_1,X_2) * gen4(X_2，X_4). 4 gen4(X_2,X_4) +=w(X_2，X_3) * w(X_3，X_4). 5 gen5(X_1，X_2) +=w(X_1,X_2). 6 gen6(X_5) += w(X_5，X_6).

bad-chain-10:

1 gen56(X_4，X_6) += w(X_4，X_5) * w(X_5,X_6). 2 gen57(X_2) +=w(X_1，X_2). 3 gen59(X_4，X_7) += gen56(X_4，X_6) * w(X_6，X_7). 4 目标 += gen124(X_3) * gen122(X_3). 5 gen124(X_3) +=gen123(X_4) * w(X_3，X_4). 6 gen123(X_4) += gen125(X_{10}) * gen121(X_{10}，X_4). 7 gen125(X_{10}) +=w(X_{10},X_{11}). 8 gen122(X_3) += gen57(X_2) * w(X_2，X_3). 9 gen121(X_{10}，X_4) +=gen119(X_4，X_8) * gen120(X_{10}，X_8). 10 gen120(X_{10}，X_8) += w(X8, X9) * w(X_9，X_{10}). 11 gen119(X_4,X_8) += gen59(X4, X7) * w(X_7，X_8).

E.2 投射依存句法分析

此程序实现了 CKY（§E.3）的一种变体，其中非终结符已用其中心词进行词汇标注。与 CKY 的 $\mathcal{O}(n^3)$ 运行时间相比，该算法最初的描述版本运行时间为 $\mathcal{O}(n^5)$（Collins，1996）。在后续工作中，Eisner和Satta（1999a）给出了该算法的一个更快版本，其运行时间为 $\mathcal{O}(n^4)$。

E.2.1 双词汇无标注（bilexical-unlabeled）

1 短语(I,H,K) += 短语(I,H,J) * 短语(J,H',K) * score(H ← H'). 2 短语(I,H',K) += 短语(I,H,J) * 短语(J,H',K) * score(H → H'). 3 短语(I,I,K) +=单词(I,_,K).

4 目标(N) += 短语(0,0,N).56 输入：单词(_,_,_)；score(_ → _)；score(_ ← _).7 输出：目标(_).

度：5。最优：4。

E.2.2 双词汇有标注（bilexical-labeled）

在 bilexical-unlabeled 的基础上扩展，加入标签（即语法关系）。

1 phrase(I,H,X,K) += phrase(I,H,Y,J) * phrase(J,H',Z,K) * rewrite(H,Y, H',Z, H,X). 2 phrase(I,H',X,K) += phrase(I,H,Y,J) * phrase(J,H',Z,K) * rewrite(H,Y, H',Z, H',X). 3 phrase(I,I,X,K) +=rewrite(X,W) * word(I,W,K). 4 goal(N) +=phrase(0,0,s,N). 5 6 输入：word(_,_,_); rewrite(_,_,_,_,_,_); rewrite(_,_). 7 输出：goal(_).

度：8。最优：7。

E.2.3 ArcEager（弧优先）

This 基准测试基于示例21；请参阅该讨论。

1 β(I,I^0,K) +=单词(I,K).2 β(I,I^1,K) +=单词(I,K).3 β(I,H^B,K) += β(I,H^B,J) * 分数(H ← H') * β(J,H'^0,K).4 β(I,H^B,K) += β(I,H^B,J) * 分数(H → H') * β(J,H'^1,K). 5 目标(N) += β(0,0^0,N). 6 7 输入：单词(_,_); 分数(_ → _); 分数(_ ← _). 8 输出：目标(_).

度：6，最优：3。

E.2.4 拆分双词汇依存句法分析

拆分双词汇语法是指中心词必须先与所有右子节点结合，然后才能与任何左子节点结合的语法。Johnson（2007）以及 Eisner和Satta（1999a）分别提出了初始的低效算法，并展示了如何将其加速到 $\mathcal{O}(n^3)$。他们最初提出的算法各不相同，因此我们将它们作为独立的基准测试提供。

E.2.4.1 Johnson（2007）版本（split-head-J）

以下程序是 Johnson（2007）最初提出的低效算法。

1 目标(N) +=x(0,V,N). 2 3 % 单词会重复，如 Johnson（2007）中的情况。 4 l(I,K) +=单词(left,I,K). 5 r(I,K) +=单词(right,I,K). 6 7 % l(I, K) 是一个以 K 为头部的跨度 8 l(I,K) += x(I,V,J) * score(V ← K) * l(J,K). 9 10 % r(I, K) 是一个以 I 为头部的跨度 11 r(I,K) += r(I,J) * score(I → V) * x(J,V,K). 12 13 % J 是 l(I, J) 的头部，也是 r(J, K) 的头部 14 x(I,J,K) +=l(I,J) * r(J,K). 15 16 输入: 单词(_,_,_); score(_ → _); score(_ ← _). 17 输出: 目标(_).

度：4，最优：3。

E.2.4.2 Eisner 和 Blatz（2007）的版本（split-head-EB）

以下程序是 Eisner and Blatz（2007）提出的初始低效算法，我们在示例44中讨论过。

```
r(I,I,K)  +=单词(I,K)。                    % 尚无右子节点
```

2 r(P,I,K) +=r(P,I,J) * arc(P → C) * l(C,J,K). % 添加右子节点 3 4 l(P,I,K) += r(P,I,K). % 尚无左子节点 5 l(P,I,K) += l(C,I,J) * arc(C ← P) * l(P,J,K). % 添加左子节点 6 7 目标(N) +=l(H,0,N). 8 9 输入: arc(C ← P); arc(P → C); 单词(_, _). 10 输出: 目标(N)。

度：5，最优：3。

E.3 成分句法分析

以下程序是 CKY（Cocke 和 Schwartz，1970年；Younger，1967年；Kasami，1965年；Lange 和 Leiß，2009年；Eisner and Blatz，2007；Tomita，1985年；Tomita，1991年；Johnson，1989年；Baker，1979年；Jelinek，1985年）的变体。我们在正文示例6中简要讨论了 CKY。

E.3.1 CKY（cky3）

1 β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * 短语(Z,J,K)。2 β(X,I,K) += γ(X,Y) * β(Y,I,K)。3 β(X,I,K) += γ(X,Y) * 单词(Y,I,K)。4 z += β(根, 0, N) * 长度(N)。5 输入：单词(W,I,K)；长度(N)； γ(X,Y,Z)； γ(X,Y)。6 输出：z。

度：6，最优：5。

E.3.2 带四元产生式的 CKY（cky4）

以下程序实现了 Tomita（1985年）的算法，其中语法可以包含长度最多为 4 的产生式规则。这只是一个简单的修改

以实现 CKY4。我们只需添加以下规则和输入声明。

7 短语(X,I_1,I_4) += γ(X,Y_1,Y_2,Y_3) * 短语(Y_1,I_1,I_2) * 短语(Y_2,I_2,I_3) 8 * 短语(Y_3,I_3,I_4)。
9 输入: γ(X,Y_1,Y_2,Y_3)

度：7，最优：6。

E.3.3 带固定语法的CKY (cky+grammar)

对于 CKY+grammar，我们使用 CKY3，但移除 γ 的输入声明

并为程序声明以下要特化到的语法 .

```
10 γ("S", "NP", "VP") += 1.011 γ
("NP", "限定词","N") += 1.012 γ("
NP", "NP", "PP") += 1.013 γ("VP",
"V", "NP") += 1.014 γ("VP", "V")
 += 1.015 γ("VP", "VP", "PP") += 1.
0.16 γ("PP", "P", "NP") += 1.017 γ
("<.>", ".") += 1.018 γ("NP", "
Papa") += 1.019 γ("N", "鱼子酱")
 += 1.020 γ("N", "勺子") += 1.0.
21 γ("N", "叉子") += 1.022 γ("N",
"望远镜") += 1.023 γ("N", "男孩")
 += 1.024 γ("N", " 女孩 ") += 1.0.
25 γ("N", "婴儿") += 1.026 γ("N",
"男人") += 1.027 γ("N", "女人")
 += 1.028 γ("N", "月亮") += 1.0.
29 γ("N", "猫") += 1.030 γ("V", "
吃了") += 1.031 γ("V", "看见了")
 += 1.032 γ("V", "喂了") += 1.0.
33 γ("V", "说了") += 1.034 γ("V",
"跳了") += 1.035 γ("P", "和") += 1.
0.36 γ("P", "在……上方") += 1.0.
37 γ("P", "在……下方") += 1.0.
38 γ("P", "在……上面") += 1.0.
39 γ("P", "在……下面") += 1.0.
40 γ("P", "在……上") += 1.041 γ(
"P", "在……里面") += 1.0.
```

度：6，最优：3。

E.3.4 张量分解解析器（tensor-decomp-parser）

以下程序是 CKY 的一种变体，其中语法（即rewrite 关系）经过了因子分解：rewrite(X,Y,Z) += λ(R) * u(R, X) * v(R, Y) * w(R, Z)。这种方法由 Cohen、Satta 等人（2013）提出：他们使用低秩张量分解来近似语法，随后用下面的算法进行句法分析。不过，利用rewrite 关系的结构，可以加快下面的算法。

1 rewrite(X,Y,Z) += λ(R) * u(R,X) * v(R,Y) * w(R,Z).2 β(I,K,X) += β(I,J,Y) * β(J,K,Z) * rewrite(X,Y,Z).3 β(I,K,X) +=单词(I,K,X)。 4 目标 += β(0,N,s) * 长度(N)。 5 6 输入：λ(R); u(R,X); v(R,Y); w(R,Z); 单词(I,K,X); 长度(_)。 7 输出：目标。

度：6，最优：4。

E.4 求逆转写语法（反向转写文法）

吴（1996）提出了反向转换语法，并给出了一种 $\mathcal{O}(n^7)$ 算法，后来 Huang、Zhang 等人（ $\mathcal{O}(n^6)$ 将其加速（2005年）。该模型会同时对一对相关句子进行句法分析，通常是一句目标语言句子和一句源语言句子，如机器翻译中的情况。该模型允许对短语进行有限的句法重排序

称为求逆。

1 β(A,I_1,K_1,I_2,K_2) +=单词$_1$(X,I_1,K_1) * 单词$_2$(X',I_2,K_2) * 转导(A,X,X').2 β(A,I_1,K_1,I_2,K_2) += β(B,I_1,J_1,I_2,J_2) * β(C,J_1,K_1,J_2,K_2) * 重写(A,B,C).3 β(A,I_1,K_1,I_2,K_2) += β(B,J_1,K_1,I_2,J_2) * β(C,I_1,J_1,J_2,K_2) * 逆向重写(A,B,C). 4 目标 += β(s,0,N_1,0,N_2) * 长度(N_1,N_2). 5 6 输入: 单词$_1$(_,_,_); 单词$_2$(_,_,_); 转导(_,_,_); 重写(_,_,_); 7 逆向重写(_,_,_); 长度(_,_). 8 输出: 目标.

度：9，最优：8。

E.5 编辑距离（编辑）

以下程序实现了两个序列之间的加权广义单调对齐：单词和单词'。这本质上就是广为人知的 Levenshtein距离（Levenshtein，1966）。

1 align(0,0) min= 1. 2 align(J,J') min= align(I,I') * word(W,I,J) * word'(W',I',J') * score(W,W'). 3 align(I,J') min= align(I,I') * word'(W',I',J') * score(ε,W'). 4 align(J,I') min= align(I,I') * word(W,I,J) * score(W,ε). 5 目标 min=align(N,N') * 长度(N) * len'(N'). 6 输入: word(_,_,_); word'(_,_,_); score(_,_); len(_); len'(_). 7 输出: 目标.

度数 6，最优：4。

E.6 Bar-Hillel 构造（bar-hillel）

以下程序实现了一个句法分析器，用于计算上下文无关句法分析器与词性序列上的二元模型的（加权）交集。这本质上是 Bar-Hillel 等人（1961年）构造上下文无关语言的方式，该语言接受正则语言与上下文无关语言的交集。

1 目标 += β(0,_,根,_,N) * 长度(N).2 β(I,A,X,D,K) += β(I,A,Y,B,J) * β(J,C,Z,D,K) * γ(X,Y,Z) * 二元组(B,C).3 β(I,X,X,X,K) +=标签(X,W) * 单词(W,I,K). 4 输入: 长度(_); 单词(_,_,_); 二元组(_,_); γ(_,_,_); 标签(_,_). 5 输出: 目标。

度数 10，最优值：8。

E.7 线性链条件随机场下的期望值（链-期望）

本例实现了内外算法加速（Li 和 Eisner，2009年），用于计算可加分解函数 $f: S \times S' \to \mathbb{R}^d$在从加权图中随机抽取的序列上的期望（例如，条件随机场（Lafferty 等人，2001年））。该图由一组权重 w以及 起始 和 末尾 节点指定。关系 α 和 β 实现了前向-后向算法（Rabiner（1989年）中讨论），而 z 是该分布的归一化常数。函数 i^{th} f 的维度 f 的期望为fbar(I)/z。

1 % 前向算法2 α(S) += 起始(S).3 α(S') += α(S) * w(S,S'). 4 % 后向算法5 β(S) += 末尾(S).6 β(S) += w(S,S') * β(S'). 7 % 归一化常数 8 z += α(S) * 末尾(S). 9 % 通过内外算法加速计算未归一化期望值 10 fbar(R) += α(S) * w(S,S') * β(S') * r(S,S',R). 11 输入: w(_,_); r(_,_,_); 起始 (_); 末尾 (_).

```
12 输出: fbar(_). z.
```

度 3，最优：3。

E.8 偶数与奇数皮亚诺数的交集 (even-odd-peano)

下面的程序会统计同时为偶数和奇数的皮亚诺数的数量。这样的数不存在，但按照程序当前的写法，不动点迭代会永无止境地枚举偶数和奇数。因此，它的运行时间是无限的。不过，可以使用折叠/展开和 define 将此程序转换为常数时间运行的版本；最优程序是空程序。

```
1 % 偶数皮亚诺数 2 even(0). 3
even(s(s(X))) += even(X). 4 5 % 奇数
皮亚诺数 6 odd(X) +=even(s(X)). 7
8 % 交集 9 evenodd(X) +=even(X)
* odd(X). 10 11 % 交集的大小 12 目
标 += evenodd(X). 13 14 输出：目标.
```

度：1，最优：0。

E.9 隐马尔可夫模型（隐马尔可夫模型）

隐马尔可夫模型（HMMs）是条件随机场（CRFs）的生成式且局部归一化的对应形式，条件随机场将在 §E.7 中讨论。Rabiner（1989年）提供了一篇经典教程。

1 v(0,起始) += 1。2 v(T’，Y’) +=v(T,Y) * 发射(Y,X) * 转移(Y,Y’) * 观测(T,X,T’)。3 目标 +=v(N,停止) * 长度(N)。4 输入：观测(_,_,_)；长度(_)；发射(_,_)；转移(_,_)。5 输出：目标。

度数 5，最优：4。

E.10 半马尔可夫模型（半马尔可夫）

半马尔可夫模型（Sarawagi 和 Cohen，2004年）将马尔可夫模型推广为对跨度而非单个词进行评分。在运行时间方面，可以在 $\mathcal{O}(n)$ 时间内对马尔可夫模型执行推理（省略对标签数量的依赖）。相比之下，对半马尔可夫模型执行推理需要 $\mathcal{O}(n^2)$。

1 β（起始，0） += 1。2 β（Y，J） += β（X，I）* 转移(X, Y) * 分块(Y, I, J)。3 目标 += β（_，N）* 长度(N)。4 输入：转移(_,_)；分块(_，_，_)；长度(_)。5 输出：目标。

度 4，最优：3。

E.11 图中的最短路径

E.11.1 显式路径（path-list）

下面的代码此前已在示例 28 中讨论过。这个示例很有意思，因为其初始运行时间是无限的，但可以将其加速至$\mathcal{O}(n^2)$ 其中 n 是图中的节点数。这个示例需要使用广义折叠/展开或左角变换（示例 46）

```
1 β([X,Y|Xs]) += 边(X,Y) * β([Y|Xs])。
2 β([X]) += 停止(X)。3 目标 += 起始(X)
* β([X|Xs])。4 5 输出：目标。6 输入：起
始(_); 边(_,_); 停止(_)。
```

E.11.2 路径起点（路径起点）

前面已在示例中讨论过下面的代码27。在这里，我们将初始 $\mathcal{O}(n^3)$ 程序加速为 $\mathcal{O}(n^2)$ 其中 n 是图中的节点数。此示例需要广义折叠 / 展开。

```
1 path(I,I)。 2 path(I,K) +=path(I,J) *
edge(J,K)。 3 4 目标 +=start(I) * path(I,K) *
stop(K)。 5 6 输出：goal。 7 输入：start(_);
edge(_,_); stop(_) 。
```

E.12 带显式栈的下推自动机（explicit-pda）

原则上，可以将下面的程序转换为 CKY。不过，这样做需要的程序变换似乎超出了我们的能力范围。

```
1 pda([],0)。% 初始栈为空 2 pda([X|Xs],K) += pda(Xs,J) * rewrite(X,Y) * word(Y,J,K)。% 移进 3 pda([X|Xs],K) += pda([Z,Y|Xs],K) * rewrite(X,Y,Z)。% 归约 4 goal += pda([s]，N) * len(N)。% 对标记为 s 的整个输入进行句法分析 5 6 输入：rewrite(_,_); rewrite(_,_,_); word(_,_,_); len(_)。 7 输出：goal。‘
```

以下程序是使用一种推测扩展手动指定的，该扩展允许采用类似组合范畴语法的组合规则。[227] 遗憾的是，截至撰写本文时，我们尚未能够完全形式化这一扩展变换的正确性条件。不过，我们可以使用折叠/展开变换，证明下面的程序与原程序等价。

调用原始程序 $\mathfrak{p}^{(0)}$ 以及下面的程序 $\mathfrak{p}^{(1)}$。

```
1 目标 +=长度(N) * 推栈自动机([s],N). 2 推栈自动机(Ys,I)/推栈自动机(Ys,I). % 基本情况 3 推栈自动机([X|Xs],K)/推栈自动机(Xs,I) += 推栈自动机(Xs,J)/推栈自动机(Xs,I) * rewrite(X,Y) * 单词(Y,J,K). 4 推栈自动机([X|Xs],K)/推栈自动机(Xs,I) +=5 推栈自动机([Z,Y|Xs],K)/推栈自动机([Y|Xs],J) * 推栈自动机([Y|Xs],J)/推栈自动机(Xs,I) * rewrite(X,Y,Z). 6 % 恢复规则 7 推栈自动机([X|Ys],K) +=(推栈自动机([X|Ys],K)/推栈自动机(Ys,I)) * 推栈自动机(Ys,I). 8 推栈自动机([],0).
```

还原 CKY。 首先，来看看程序如何转换为 CKY。展开目标规则（两次）后，我们得到一个不再依赖于.....的程序

[227]看起来，我们的提升变换无法实现这种变换，因为它们只有类似组合范畴语法的应用规则。

关于推栈自动机 关系：

```
1 推栈自动机(Ys,I) / 推栈自动机(Ys,I). 2 推栈自动机([X|Xs],K)/推栈自动机(Xs,I)  += 推栈
自动机(Xs,J)/推栈自动机(Xs,I) * rewrite(X,Y) * 单词(Y,J,K). 3 推栈自动机([X|Xs],K)/推栈
自动机(Xs,I)  +=4 推栈自动机([Z,Y|Xs],K)/推栈自动机([Y|Xs],J) * 推栈自动机([Y|Xs],J)/推
栈自动机(Xs,I) * rewrite(X,Y,Z). 5 目标  +=长度(N) * 推栈自动机([s],N)/推栈自动机([],0).
```

这个程序不会实例化`Xs`。我们可以直接运行这个程序，其 $\mathcal{O}(k^3\, n^3)$ 运行时间与 CKY 相同。我们还可以使用程序特化 §6.2 变换，得到以下程序；它在视觉上与 CKY 更为相似。请注意，新关系`p` 的参数顺序是任意选定的。

```
1 p(X,K,I)  +=  rewrite(X,Y) * word(Y,I,K). 2 p(X,K,I)  +=  p(Z,K,J) *
p(Y,J,I) * rewrite(X,Y,Z). 3 目标  +=  长度(N) * p(s,N,0).
```

很容易看出，这个程序与 CKY 等价。

通过折叠/展开证明正确性。 现在，我们来考察手动指定的程序的正确性。关键在于证明，恢复规则可以通过变换转化为原始定义 $\mathtt{p}^{(0)}$。如果我们能通过保语义变换做到这一点，就能证明 $\mathtt{p}^{(0)}$ 和 $\mathtt{p}^{(1)}$等价。

```
1 推栈自动机([X|Ys],K)  +=(推栈自动机([X|Ys],K)/推栈自动机(Ys,I)) *
推栈自动机(Ys,I)。 2  推栈自动机([],0)。
```

可转换为原始规则：

```
1 推栈自动机([],0). 2 推栈自动机([X|Xs],K)  +=  推栈自动机(Xs,J) *
rewrite(X,Y) * 单词(Y,J,K). 3 推栈自动机([X|Xs],K)  +=推栈自动机([Y2,Y1|Xs],
K) * rewrite(X,Y1,Y2).
```

将递归规则展开两次（使用 $\mathtt{p}^{(1)}$ 作为辅助定义）：

```
1 推栈自动机([],0)。 2 推栈自动机([X|Ys],K)  +=  推栈自动机([Z,Y|Ys],K) / 推栈自动机([Y|Ys],J) * 推栈自动机([
Y|Ys],J) / 推栈自动机(Ys,I) *
```

```
    rewrite(X,Y,Z) * 推栈自动机(Ys,I)。 3 推栈自动机([X|Ys],K) +=
rewrite(X,Y) * 单词(Y,I,K) * 推栈自动机(Ys,I)。
```

将此程序折叠两次（使用 $\mathfrak{p}^{(1)}$ 作为辅助定义）正是我们所要的

之后。

```
1 推栈自动机([],0)。2 推栈自动机([X|Ys],K) +=推栈自动机([Z,Y|Ys],K) *
rewrite(X,Y,Z)。3 推栈自动机([X|Ys],K) +=rewrite(X,Y) * 单词(Y,I,K) * 推栈自动
机(Ys,I)。
```

由于我们使用保语义规则得出了这个定义，因此可以推出 $\mathfrak{p}^{(0)} \equiv \mathfrak{p}^{(1)}$。我们使用 Dyna 库处理了这个示例，因此无需手动执行变换或检查其安全条件；库无缝地处理了所有这些工作。我们使用搜索算法枚举有效的折叠，但由我们手动从中进行选择。

引用

Abadi, Martin, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu 和 Xiaoqiang Zheng（2016）。《TensorFlow：大规模机器学习系统》。收录于：*USENIX 操作系统设计与实现研讨会（OSDI 16）*。 统一资源定位符：https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf。

Aït-Kaci, Hassan（1991年）。*Warren 抽象机：教程式重构*。麻省理工学院出版社。统一资源定位符：https://web.archive.org/web/20031004100957/http://www.vanx.org/archive/wam/wambook.pdf。

Aji, S. M. 和 R. J. McEliece（2006年）。“广义分配律”。载于：*IEEE 信息论汇刊* 46.2。 统一资源定位符：https://doi.org/10.1109/18.825794。

Alonso, MiguelA.、DavidCabrero、Eric de la Clergerie 和 Manuel Vilares（1999）。“用于 TAG 句法分析的表格算法”。载于：计算语言学协会

计算语言学协会欧洲分会。 统一资源定位符：https://aclanthology.org/E99-1020。

Althoen, Steven C. 和 Renate McLaughlin（1987 年）。《高斯-若尔当消元：简史》。载于 《美国数学月刊》94.2。 统一资源定位符：
http://www.jstor.org/stable/2322413。

Apt, Krzysztof R. 和 Mark Wallace（2007）。 使用 *ECLiPSe* 进行约束逻辑编程。剑桥大学出版社。 统一资源定位符： https://doi.org/10.1017/CBO9780511607400。

Bagnara, Roberto、Fred Mesnard、Andrea Pescetti 和 Enea Zaffanella（2010年）。《线性秩函数的自动综合：完整未删节版》。载于 计算研究资料库 abs/1004.0944。arXiv：1004.0944。统一资源定位符： http://arxiv.org/abs/1004.0944。

Baharev, Ali、Hermann Schichl 和 Arnold Neumaier（2016）。《求解稀疏非线性方程组的分解方法》。载于 提交中。 统一资源定位符：
https://baharev.info/publications/baharev_撕裂_survey.pdf。

Baker, J. K.（1979年）。“用于语音识别的可训练语法”。收录于：美国声学学会会议上发表的语音通信论文。统一资源定位符：
https://compcalc.github.io/public/baker/baker1979trainable.pdf。

Baldoni, Roberto, Emilio Coppa, Daniele Cono D’elia、Camil Demetrescu 和 Irene Finocchi（2018）。《符号执行技术综述》。载于：*ACM*计算综述51.3。统一资源定位符：https://doi.org/10.1145/3182657。

Bar-Hillel、耶霍书亚、米哈·阿舍尔·佩尔莱斯和埃利·沙米尔（1961年）。《简单短语结构文法的形式属性》。载于：*Zeitschrift fürPhonetik, Sprachwissenschaft undKommunikationsforschung* 14（2）。统一资源定位符：https://www.proquest.com/openview/fb41296047fb7453dcb1de182b4aa0b6/1。

Baras, John S. 和乔治·西奥多拉科普洛斯（2010年）。网络中的路径问题。通信网络综合讲座。Morgan & Claypool出版社。统一资源定位符：https://doi.org/10.2200/S00245ED1V01Y201001CNT003。

Bellman, Richard（1958年）。“关于一个路由问题”。载于：应用数学季刊16.1。统一资源定位符：http://www.jstor.org/stable/43634538。

Bilmes, Jeff、Kste Asanovi´c、吉姆·德梅尔、多米尼克·拉姆和Chee-Whye Chin（1996）。*PHiPAC*：一种可移植的高性能*ANSI C*编码方法论及其在矩阵乘法中的应用。技术报告。田纳西大学。统一资源定位符：https://people.ece.uw.edu/bilmes/p/pgs/b2hd-bilmes1996-lapack.html。

Bilmes, Jeff 和 GeoffreyZweig（2002年）。“图模型工具包：用于语音与时间序列处理的开源软件系统”。载于：*2002 IEEE*声学、语音与信号处理国际会议。第4卷。统一资源定位符：https://ieeexplore.ieee.org/document/5745513。

Bossi，安娜丽莎、妮科莱塔·科科和桑德罗·埃塔莱（1992年）。“论安全折叠”。载于：程序语言实现与逻辑编程国际研讨会。统一资源定位符：https://link.springer.com/chapter/10.1007/3-540-55844-6_135。

Bradbury、James、罗伊·弗罗斯蒂格、彼得·霍金斯、Matthew JamesJohnson、克里斯·利里、杜格尔·麦克劳林、乔治·内库拉、Adam Paszke、杰克·范德普拉斯，

Skye Wanderman-Milne 和 Qiao Zhang（2018）。*JAX*：可组合的 *Python* 变换+*NumPy* 程序。版本 0.3.13。统一资源定位符：http://github.com/google/jax。

Büchse, Matthias、Mark-Jan Nederhof 和 Heiko Vogler（2011年）。《使用同步树邻接文法进行树解析》。载于：解析技术国际研讨会。 统一资源定位符：https://www.aclweb.org/anthology/W11-2903。

Burden, Håkan 和 Peter Ljunglöf（2005年）。《线性上下文无关重写系统的句法分析》。载于：解析技术国际研讨会论文集。 统一资源定位符：https://aclanthology.org/W05-1502。

Burstall, Rod M. 和 John Darlington（1977年）。《用于开发递归程序的变换系统》。载于：*ACM* 期刊 24.1。 统一资源定位符：https://doi.org/10.1145/321992.321996。

Chekuri, Chandra 和 Anand Rajaraman（1997年）。《合取查询包含问题再探》。载于：国际数据库理论会议论文集。 统一资源定位符：https://doi.org/10.1007/3-540-62222-5_36。

Cheney, James、LauraChiticariu和Wang-ChiewTan（2009年）。“数据库中的溯源：为何、如何以及何处”。载于：数据库基础与趋势®1.4. 统一资源定位符：http://dx.doi.org/10.1561/1900000006。

Chiang, David、Colin McDonald和Chung-chieh Shan（2023年）。“精确递归概率编程”。载于：统一资源定位符：https://arxiv.org/abs/2210.01206。

Chiang, David, Alexander M. Rush 和 Boaz Barak（2021）。《命名张量表示法》。载于：计算研究资料库 abs/2102.13196。arXiv：2102.13196。 统一资源定位符：https://arxiv.org/abs/2102.13196。

Cocke, John 和Jacob T. Schwartz（1970）。《程序设计语言及其编译器：初步笔记》。技术报告。库朗数学科学研究所，纽约大学。统一资源定位符：http://www.softwarepreservation.org/projects/FORTRAN/CockeSchwartz_ProgLangCompilers.pdf。

Cohen, Shay B., Carlos Gómez-Rodríguez 和 Giorgio Satta（2011年）。“生成式概率非投射依存句法分析的精确推断”。载于：经验方法与自然语言处理会议论文集。统一资源定位符：https://aclanthology.org/D11-1114。

Cohen, Shay B., Giorgio Satta 和 Michael Collins（2013）。“使用张量分解进行近似概率上下文无关文法句法分析”。载于：计算语言学协会北美分会人类语言技术会议论文集。统一资源定位符：https://aclanthology.org/N13-1052。

Cohen, Shay B., Robert J. Simmons 和 Noah A. Smith（2011年）。“加权逻辑程序的乘积”。载于：逻辑编程理论与实践 11。统一资源定位符：https://link.springer.com/chapter/10.1007/978-3-540-89982-2_18。

Cohn, Paul Moritz (1969)。普遍代数。第6卷。Harper and Row。统一资源定位符：https://link.springer.com/book/10.1007/978-94-009-8399-1。

Collins, Michael John (1996)。“一种基于二元语法词汇依存关系的新型统计句法分析器”。载于：计算语言学协会会议论文集。统一资源定位符：https://www.aclweb.org/anthology/P96-1025。

Colmerauer, Alain和Philippe Roussel（1996）。“Prolog的诞生”。载于：《程序设计语言史》。第II卷。计算机协会。统一资源定位符：https://doi.org/10.1145/234286.1057820。

Colón、Michael和Henny Sipma（2001年）。“线性排序函数的综合”。载于：国际系统构造与分析工具和算法会议论文集。第2031卷。施普林格。统一资源定位符：https://doi.org/10.1007/3-540-45319-9_6。

Cousot, Patrick 和 Radhia Cousot（1977年）。《抽象解释：通过构造或近似不动点实现程序静态分析的统一格模型》。载于：*ACM* 编程语言原理研讨会会议录。统一资源定位符：https://doi.org/10.1145/512950.512973。

Dawson, Steven、C. R. Ramakrishnan 和 David S. Warren（1996）。《使用通用逻辑编程系统进行实用程序分析——案例研究》。载于：*ACM SIGPLAN* 编程语言设计与实现会议论文集。统一资源定位符：https://doi.org/10.1145/231379.231399。

Deshpande, Amol、Zachary G. Ives 和 Vijayshankar Raman（2007年）。《自适应查询处理》。载于：数据库基础与趋势®1.1. 统一资源定位符：https://doi.org/10.1561/1900000001。

Dial, Robert B.（1969年）。“算法360：采用拓扑排序的最短路径森林 [H]”。载于：计算机协会通讯12.11。统一资源定位符：https://doi.org/10.1145/363269.363610。

Dijkstra, Edsger W.（1959年）。“关于图相关的两个问题的一则注记”。载于：《数值数学》1。统一资源定位符：https://doi.org/10.1007/BF01386390。

Dolan, Stephen（2013）。“乐趣和半环：一篇函数式珍珠在……上滥用线性代数”。在……里面：*ACM SIGPLAN*国际*Con-*

*ference*在……上函数式编程。计算机协会。统一资源定位符：https://doi.org/10.1145/2500365.2500613。

Dozat, Timothy 和 Christopher D. Manning（2017年）。“用于神经依存句法分析的 DeepBiaffineAttention”。收录于：国际学习表征会议论文集。统一资源定位符：https://openreview.net/forum?id=Hk95PK9le。

Durrett, Greg 和 Dan Klein（2015）。“神经条件随机场句法分析”。收录于：计算语言学协会年会暨自然语言处理国际联合会议论文集。统一资源定位符：https://aclanthology.org/P15-1030。

Dyer, Chris（2013）。“最小错误率训练与凸包半环”。收录于：计算研究资料库 abs/1307.3675。arXiv：1307.3675。统一资源定位符：http://arxiv.org/abs/1307.3675。

Eisner, Jason（2001年）。“期望半环：适用于有限状态转换器的灵活期望最大化”。收录于：*ESSLLI*有限状态自然语言处理方法研讨会论文集。赫尔辛基。统一资源定位符：http://cs.jhu.edu/~jason/papers/#eisner-2001-fsmnlp。

Eisner, Jason（2002年）。《概率有限状态转导器的参数估计》。载于：计算语言学协会会议论文集。 统一资源定位符：https://www.aclweb.org/anthology/P02-1001。

Eisner, Jason（2023年）。《时间与空间高效的加权演绎》。载于：计算语言学协会会刊。统一资源定位符：http://cs.jhu.edu/~jason/papers/#eisner-2023-tacl。

Eisner, Jason 和 JohnBlatz（2007）。《用于优化句法分析算法及其他加权逻辑程序的程序变换》。载于：形式语法会议论文集。统一资源定位符：http://cs.jhu.edu/~jason/papers/#eisner-blatz-2007。

Eisner, Jason 和 Nathaniel W. Filardo（2011年）。“Dyna：将 Datalog 扩展到现代人工智能”。载于：*Datalog Reloaded*。第 6702 卷。计算机科学讲义。施普林格。统一资源定位符：https://link.springer.com/chapter/10.1007/978-3-642-24206-9_11。

Eisner, Jason、Eric Goldlust 和 Noah A. Smith（2004年）。“Dyna：一种用于实现动态程序的声明式语言”。载于：计算语言学协会会议论文集。统一资源定位符：https://www.aclweb.org/anthology/H05-1036/。

Eisner, Jason、Eric Goldlust 和 Noah A. Smith（2005年）。“编译计算语言学：加权动态规划与 Dyna 语言”。载于：人类语言技术联合会议暨自然语言处理经验方法会议论文集。统一资源定位符：https://aclanthology.org/H05-1036/。

Eisner, Jason、Michael Kornbluh、Gordon Woodhull、Raymond Buse、Samuel Huang、Constantinos Michael 和 George Shafer（2006年）。“通过大型有向图和超图进行可视化导航”。载于：*IEEE* 信息可视化研讨会论文集。统一资源定位符：http://cs.jhu.edu/~jason/papers/#DYNASTY-2006。

Eisner, Jason 和 Giorgio Satta（1999a）。“双词汇上下文无关文法和中心词自动机文法的高效句法分析”。载于：计算语言学协会会议论文集。统一资源定位符：https://www.aclweb.org/anthology/P99-1059。

Eisner, Jason 与 Giorgio Satta（1999b）。“双词汇上下文无关语法与中心词自动机语法的高效句法分析”。载于：计算语言学协会会议论文集。统一资源定位符：https://aclanthology.org/P99-1059/。

Eisner, Jason 与 Giorgio Satta（2000）。“词汇化树邻接语法的更快句法分析算法”。载于：树邻接语法及相关框架国际研讨会论文集。统一资源定位符：https://aclanthology.org/W00-2011/。

Eisner, Jason M.（1996）。“依存句法分析的三种新概率模型：一项探索”。载于：计算语言学协会会议论文集。统一资源定位符：https://www.aclweb.org/anthology/C96-1058。

Elidan, Gal、Ian McGraw 与 Daphne Koller（2006年）。“残差信念传播：异步消息传递的知情调度”。载于：人工智能不确定性会议论文集。统一资源定位符：https://arxiv.org/abs/1206.6837。

Esparza, Javier、Stefan Kiefer 和 Michael Luttenberger（2007）。《将牛顿法扩展至 ω-连续半环》。载于：语言理论发展国际会议论文集。统一资源定位符：https://doi.org/10.1007/978-3-540-73208-2_17。

Esparza, Javier、Stefan Kiefer 和 Michael Luttenberger（2010年）。《牛顿式程序分析》。载于：*ACM*学报57.6。统一资源定位符：https://doi.org/10.1145/1857914.1857917。

Etessami, Kousha 和 Mihalis Yannakakis（2009）。“递归马尔可夫链、随机文法和非线性方程单调系统”。载于：美国计算机协会期刊 56.1。 统一资源定位符：https://doi.org/10.1145/1462153.1462154。

Felzenszwalb, P.F. 和 D. McAllester（2007）。“广义 A* 架构”。载于： 人工智能研究学报 29。 统一资源定位符： http://www.jair.org/papers/paper2187.html。

Filardo, Nathaniel Wesley（2017）。“Dyna 2：迈向通用加权逻辑语言”。博士论文。约翰斯·霍普金斯大学。 统一资源定位符：https://jscholarship.library.jhu.edu/bitstream/handle/1774.2/58686/FILARDO-DISSERTATION-2017.pdf。

Filardo, Nathaniel Wesley 和 Jason Eisner（2012）。“有限算术电路的灵活求解器”。载于：国际逻辑编程会议。

第 17 卷。莱布尼茨国际信息学论文集。 统一资源定位符： http://cs.jhu.edu/~jason/papers/#filardo-eisner-2012-iclp。

Floyd, Robert W.（1962 年）。“算法97：最短路径”。载于：计算机协会通讯 5.6。 统一资源定位符： https://doi.org/10.1145/367766.368168。

Ford Jr., Lester Randolph（1956）。网络流理论。统一资源定位符：https://link.springer.com/chapter/10.1007/0-387-22633-8_9。

Francis-Landau、Matthew、Tim Vieira 和 Jason Eisner（2020）。《带内置谓词和聚合的逻辑程序求值：Bag Relations 的一种演算》。载于：国际重写逻辑及其应用研讨会。统一资源定位符：http://cs.jhu.edu/~jason/papers/#francislandau-vieira-eisner-2020-wrla。

Frigo, Matteo与Steven G. Johnson（1998）。“FFTW：面向快速傅里叶变换的自适应软件架构”。载于：*IEEE*声学、语音与信号处理国际会议论文集。第3卷。统一资源定位符：https://doi.org/10.1109%2FICASSP.1998.681704。

Frühwirth, Thom（1998）。“约束处理规则的理论与实践”。载于：《逻辑程序设计杂志》37.1。统一资源定位符：http://www.sciencedirect.com/science/article/pii/S0743106698100055。

Galler, Bernard A.与Michael J. Fischer（1964）。“一种改进的等价算法”。载于：《计算机协会通讯》7.5。统一资源定位符：https://doi.org/10.1145/364099.364331。

Gallo, Giorgio、Giustino Longo、Stefano Pallottino与Sang Nguyen（1993）。“有向超图及其应用”。载于：《离散应用数学》42.2。统一资源定位符：https://www.sciencedirect.com/science/article/pii/0166218X9390045P。

Ganzinger, Harald 和 David McAllester（2002年）。“逻辑算法”。载于：逻辑编程。统一资源定位符：https://link.springer.com/chapter/10.1007/3-540-45619-8_15。

Gergatsoulis、Manolis 与 Maria Katzouraki（1994）。“确定子句程序的展开/折叠变换”。载于：程序设计会议论文集

语言实现与逻辑编程。统一资源定位符：https://doi.org/10.1007/3-540-58402-1_24。

Gildea, Daniel（2011年）。《基于树分解的语法因式分解》。载于：计算语言学37.1。 统一资源定位符： https://www.aclweb.org/anthology/J11-1008。

Gildea, Daniel（2020年）。《高效的外部计算》。载于：计算语言学46.4。 统一资源定位符： https://aclanthology.org/2020.cl-4.2。

Gilroy, Sorcha、Adam Lopez 和 Sebastian Maneth（2017年）。《使用正则图语法解析图》。载于：词汇与计算语义联合会议论文集。 统一资源定位符：https://www.aclweb.org/anthology/S17-1024。

Gómez-Rodríguez, Carlos（2010年）。 实用文本分析的解析图式。

第1卷。数学、计算、语言与生命：数学语言学与语言理论前沿。帝国理工学院出版社。 统一资源定位符：http://www.worldscientific.com/worldscibooks/10.1142/p714。

Gómez-Rodríguez, Carlos、Jesús Vilares 和 Miguel A. Alonso（2009年）。“句法分析图式的编译器”。载于：软件——实践与经验39.5。 统一资源定位符：https://doi.org/10.1002/spe.904。

Gondran, Michel 和 Michel Minoux（2008年）。图、*Dioid* 与半环：新模型与算法。第 41 卷。施普林格科学与商业媒体。统一资源定位符：https://link.springer.com/book/10.1007/978-0-387-75450-5。

Goodman, Joshua（1999）。《半环解析》。载于：计算语言学25.4。统一资源定位符：https://www.aclweb.org/anthology/J99-4004。

Goodman, Joshua T.（1998）。《由内而外解析》。博士论文。哈佛大学。统一资源定位符：https://arxiv.org/abs/cmp-lg/9805007。

Green, Todd J., Grigoris Karvounarakis 和 Val Tannen（2007年）。《溯源半环》。载于：*ACM SIGMOD-SIGACT-SIGART* 数据库系统原理研讨会论文集。统一资源定位符：https://doi.org/10.1145/1265530.1265535。

Green, Todd J. 和 Val Tannen（2017年）。《数据库溯源的半环框架》。载于：*ACM SIGMOD-SIGACT-SIGAI* 数据库系统原理研讨会论文集。统一资源定位符：https://doi.org/10.1145/3034786.3056125。

Griewank, Andreas 和 Andrea Walther（2008年）。导数求值：算法微分原理与技术。第2版。SIAM。统一资源定位符：https://doi.org/10.1137/1.9780898717761。

Hall, Keith B.（2005年）。“用于集成句法语言建模的最佳优先单词格句法分析：技术”。博士论文。布朗大学。统一资源定位符：https://www.proquest.com/openview/eeb76c0001e6d80db7c72be0ea977440/。

Harris, Charles R., K. Jarrod Millman, St’efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern,Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern’andez del R’io, Mark Wiebe, Pearu Peterson, Pierre G’erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, 和

Travis E. Oliphant（2020年）。《使用 NumPy 进行数组编程》。载于：《自然》585.7825。统一资源定位符：https://doi.org/10.1038/s41586-020-2649-2。

Herbrand, Jacques（1930年）。“证明理论研究”。博士论文。巴黎大学。统一资源定位符： http://www.numdam.org/issue/THESE_1930__110__1_0.pdf。

Hong, Juneki 和 Liang Huang（2018）。“基于循环神经网络和动态规划的线性时间成分句法分析”。载于：计算语言学协会年会论文集。 统一资源定位符：https://aclanthology.org/P18-2076。

Hopcroft, John E.、Rajeev Motwani 和 Jeffrey D. Ullman（2007年）。自动机理论、语言与计算导论。第3版。Addison– Wesley。统一资源定位符：https://dl.acm.org/doi/book/10.5555/1196416。

Huang, Liang（2008年）。“半环和超图框架中的高级动态规划”。载于：国际计算语言学会议：计算语言学中的高级动态规划：理论、算法与应用——教程讲义。统一资源定位符：https://aclanthology.org/C08-5001。

Huang、Liang 和 David Chiang（2005年）。《更好的 k-最佳句法分析》。在：国际句法分析技术研讨会论文集。统一资源定位符：https://aclanthology.org/W05-1506。

Huang、Liang 和 Kenji Sagae（2010年）。《线性时间增量句法分析的动态规划》。载于：计算语言学协会会议论文集。 统一资源定位符：https://aclanthology.org/P10-1110。

Huang、Liang、Hao Zhang 和 Daniel Gildea（2005年）。《作为带钩子的词汇化句法分析的机器翻译》。载于：解析技术国际研讨会。 统一资源定位符：https://www.aclweb.org/anthology/W05-1507。

Hutter, Frank（2009年）。“用于解决困难计算问题的算法自动配置”。博士论文。英属哥伦比亚大学（温哥华）。统一资源定位符：https://www.cs.ubc.ca/~hutter/papers/Hutter09PhD.pdf。

Iwano, Kazuo 和 Kenneth Steiglitz（1990）。“凸多边形上的半环与零和环问题”。载于：*SIAM* 计算杂志 19.5。 统一资源定位符：https://doi.org/10.1137/0219061。

Jelinek, Frederick（1985年）。“文本生成的马尔可夫源建模”。载于：北约高级研究所：处理技术对通信的影响。 统一资源定位符：https://link.springer.com/chapter/10.1007/978-94-009-5113-6_28。

Jiang, Jiarong、Adam Teichert、Hal Daumé III 和 Jason Eisner（2012 年）。“用于权衡准确率与速度的学习得到的优先级排序”。载于：神经信息处理系统进展。统一资源定位符： http://cs.jhu.edu/~jason/papers/#jiang-et-al-2012-nips。

Johnson，Mark（1989年）。《Tomita算法的计算复杂性》。在：解析技术国际研讨会。 统一资源定位符：https://www.aclweb.org/anthology/W89-0221。

Johnson, Mark (1998). “使用左角语法变换对基于约束的语法进行有限状态近似”。发表于：年会

计算语言学协会与国际计算语言学会议。 统一资源定位符：
https://aclanthology.org/P98-1101。

Johnson, Mark（2007年）。“通过展开-折叠将投射性双词汇依存语法转换为可高效解析的上下文无关文法”。载于：计算语言学协会会议论文集。统一资源定位符：
https://www.aclweb.org/anthology/P07-1022。

Johnson, Mark 和 Brian Roark（2000）。“使用选择性左角变换和因子分解构建紧凑的非左递归语法”。载于：国际计算语言学会议论文集。统一资源定位符：
https://aclanthology.org/C00-1052。

Jurafsky, Dan 和 James H. Martin（2020年）。《语音与语言处理：自然语言处理、计算语言学与语音识别导论》第三版。第三版。（草稿）。统一资源定位符：
https://web.stanford.edu/~jurafsky/slp3/。

Kasami, Tadao（1965年）。上下文无关语言的高效识别与句法算法。技术报告AF-CRL-65-758。空军剑桥研究实验室。统一资源定位符：
https://www.ideals.illinois.edu/handle/2142/74304。

Kawamura, Tadashi 与 Tadashi Kanamori（1990）。“展开/折叠逻辑程序变换中更强等价性的保持”。载于：理论计算机科学75.1。统一资源定位符：
https://www.sciencedirect.com/science/article/pii/030439759090065P。

Kim、Yoon、Carl Denton、Luong Hoang 与 Alexander M. Rush（2017年）。“结构化注意力网络”。载于：国际学习表征会议论文集。统一资源定位符：https://openreview.net/forum?id=HkE0Nvqlg。

Kleene, Stephen C.（1977年）。《事件在神经网络和有限自动机中的表示》。载于：自动机研究34。 统一资源定位符：https://apps.dtic.mil/sti/pdfs/ADA596138.pdf。

Klein, Dan 和 Chris D. Manning（2001年）。《句法分析与超图》。载于：第七届国际句法分析技术研讨会论文集。 统一资源定位符：https://aclanthology.org/W01-1812。

Knight, Kevin（1989年）。《合一：一项跨学科综述》。载于：*ACM* 计算调查21.1。 统一资源定位符： https://doi.org/10.1145/62029.62030。

Knuth, Donald E.（1977年）。《Dijkstra 算法的一种推广》。载于：信息处理快报6.1。 统一资源定位符：https://www.sciencedirect.com/science/article/pii/0020019077900023。

Koo、Terry 和 Michael Collins（2010年）。“高效三阶依存句法分析器”。载于：计算语言学协会会议论文集。统一资源定位符：https://www.aclweb.org/anthology/P10-1001。

Kowalski、Robert（1979年）。“算法 = 逻辑 + 控制”。载于：计算机协会通讯22.7。统一资源定位符：https://dl.acm.org/citation.cfm?id=359136。

Kuhlmann, Marco（2013）。“轻度非投射依存语法”。收录于：计算语言学39.2. 统一资源定位符： https://www.aclweb.org/anthology/J13- 2004年。

Kuhlmann, Marco、Carlos Gómez-Rodríguez 和 Giorgio Satta（2011年）。“用于基于转移的依存句法分析器的动态规划算法”。收录于：计算语言学协会年会：人类语言技术论文集。 统一资源定位符： https://aclanthology.org/P11-1068。

Kuhlmann, Marco 和 Giorgio Satta（2014年）。“组合范畴语法的新句法分析算法”。载于：计算语言学协会会刊2. 统一资源定位符：https://aclanthology.org/Q14-1032。

Kuhlmann, Marco、Giorgio Satta 和 Peter Jonsson（2018）。“CCG句法分析的复杂性”。载于：计算语言学44.3. 统一资源定位符：https://www.aclweb.org/anthology/J18-3004。

Kuich, Werner（1997年）。“半环与形式幂级数：它们与形式语言和自动机的关联”。载于：《形式语言手册：词、语言、语法》。第1卷。施普林格。 统一资源定位符： https://doi.org/10.1007/978-3-642-59136-5_9。

Lafferty, John、Andrew McCallum 和 Fernando Pereira（2001年）。“条件随机场：用于序列数据分割和标注的概率模型”。载于：国际机器学习会议论文集。统一资源定位符： https://dl.acm.org/doi/10.5555/645530.655813。

Lange、Martin 和 Hans Leiß（2009年）。“转为乔姆斯基范式，还是不转？一种高效且易于呈现的 CYK算法版本”。载于：*Informatica Didactica*8。 统一资源定位符： http://www.informatica-didactica.de/cmsmadesimple/index.php?page=LangeLeiss2009。

Lauritzen, S. L. 和 D. J. Spiegelhalter（1988年）。“图结构上的概率局部计算及其在专家系统中的应用”。载于：皇家统计学会会刊。B辑（方法论）50.2。 统一资源定位符：
https://rss.onlinelibrary.wiley.com/doi/10.1111/j.2517-6161.1988.tb01721.x。

Lee, Kenton, Mike Lewis 和 Luke Zettlemoyer（2016）。《具有最优性保证的全局神经CCG句法分析》。载于：自然语言处理实证方法会议论文集。统一资源定位符：
https://aclanthology.org/D16-1262。

Lehmann, Daniel J.（1977年）。《传递闭包的代数结构》。载于：理论计算机科学4.1。统一资源定位符：
https://www.sciencedirect.com/science/article/pii/0304397577900561。

Leis, Viktor, BernhardeRadke, AndreyGubichev, Alfons Kemper 和 Tho mas Neumann（2017年）。《正确进行基数估计：基于索引的连接采样》。载于：创新数据系统研究会议论文集。统一资源定位符：
https://www.cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf。

Levenshtein, Vladimir I.（1966）。《能够纠正删除、插入和反转的二进制码》。载于：苏联物理学报告8（10）。统一资源定位符：
https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf。

Li, Zhifei 和 Jason Eisner（2009年）。“用于翻译森林最小风险训练的一阶和二阶期望半环”。载于：经验方法与自然语言处理会议论文集。统一资源定位符：
https://www.aclweb.org/anthology/D09-1005。

Lopez, Adam（2009年）。“作为加权演绎的翻译”。载于：欧洲计算语言学协会会议论文集。统一资源定位符：https://www.aclweb.org/anthology/E09-1061。

Łukasiew icz, Jan（1920）。《关于三值逻辑》。载于：《哲学评论》5 .

Luttenberger, Michael（2009年）。《半环上的多项式系统求解：牛顿法的推广》。博士论文。慕尼黑工业大学。统一资源定位符：https://d-nb.info/100090086X/34。

Mächler, Martin（2012 年）。准确计算$\log(1 - \exp(-|a|))$ 由 *Rmpfr* 软件包评估。技术报告。统一资源定位符：
https://cran.r-project.org/web/packages/Rmpfr/vignettes/log1mexp-note.pdf。

Maher, Michael J.（1987 年）。逻辑程序变换系统的正确性。技术报告 RC13496。IBM 研究报告，T.J. Watson 研究中心。统一资源定位符：
https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.2964。

Mannila, Heikki 和 Esko Ukkonen（1986年）。《论合一序列的复杂性》。载于：国际逻辑编程会议论文集。统一资源定位符：
https://link.springer.com/chapter/10.1007/3-540-16492-8_69。

Marcus, MitchellP.、Beatrice Santorini 和 Mary Ann Marcinkiewicz（1993）。《构建大型英语标注语料库：宾夕法尼亚树库》。载于：计算语言学19.2。统一资源定位符：https://www.aclweb.org/anthology/J93- 2004年。

Martelli, Alberto 和 Ugo Montanari（1982年）。《一种高效的合一算法》。载于：*ACM* 编程语言与系统汇刊 4.2。统一资源定位符：
https://dl.acm.org/doi/10.1145/357162.357169。

McAllester, David（2002年）。《关于静态分析的复杂度分析》。载于：*ACM*学报 49.4。 统一资源定位符：https://doi.org/10.1145/581771.581774。

Melamed, I. Dan（2003年）。“多文本语法与同步解析器”。载于：计算语言学协会北美分会人类语言技术会议论文集。统一资源定位符：https://www.aclweb.org/anthology/N03-1021。

Melamed, I. Dan、Giorgio Satta与Benjamin Wellington（2004年）。“广义多文本语法”。载于：计算语言学协会年会论文集。西班牙巴塞罗那。统一资源定位符：https://aclanthology.org/P04-1084。

Aaron Meurer、Christopher P. Smith、Mateusz Paprocki、Ondřej Čertík、Sergey B. Kirpichev、Matthew Rocklin、AMiT Kumar、Sergiu Ivanov、Jason K. Moore、Sartaj Singh、Thilina Rathnayake、Sean Vig、Brian E. Granger、Richard P. Muller、Francesco Bonazzi、Harsh Gupta、Shivam Vats、Fredrik Johansson、Fabian Pedregosa、Matthew J. Curry、Andy R. Terrel、Štěpán Roučka、Ashutosh Saboo、Isuru Fernando、Sumith Kulal、Robert Cimrman与Anthony Scopatz（2017年）。“SymPy：Python中的符号计算”。载于：《*PeerJ*计算机科学》3。统一资源定位符：https://doi.org/10.7717/peerj-cs.103。

Minnen, Guido（1996）。“动态自底向上处理中的过滤器优化 Magic”。载于：计算语言学协会年会论文集。统一资源定位符：https://aclanthology.org/P96-1033。

Mohri, Mehryar（1997）。“语言与语音处理中的有限状态转换器”。载于：计算语言学23.2。统一资源定位符：https://www.aclweb.org/anthology/J97-2003。

Mohri, Mehryar（2002年）。“最短路径问题的半环框架与算法”。载于：《自动机、语言与组合学杂志》7.3。统一资源定位符：https://cs.nyu.edu/~mohri/pub/jalc.pdf。

Mohri, Mehryar（2009年）。“加权自动机算法”。载于：《加权自动机手册》。施普林格，柏林·海德堡。统一资源定位符：https://doi.org/10.1007/978-3-642-01492-5_6。

Moore, Edward F.（1959年）。“迷宫中的最短路径”。载于：《国际开关理论研讨会论文集》。统一资源定位符：https://books.google.com/books/about/The_最短_路径_穿过_一个_迷宫.html?id=IVZBHAAACAAJ。

Moura, Leonardo Mendonça de 和 Nikolaj Bjørner（2008年）。“Z3：一种高效的 SMT 求解器”。载于：《国际系统构造工具与算法分析会议论文集》。统一资源定位符：https://doi.org/10.1007/978-3-540-78800-3_24。

Nederhof, Mark-Jan 和 Ricardo Sánchez-Sáez（2011年）。“用于解析选择的部分括号化结构的解析”。在：国际句法分析技术研讨会。统一资源定位符：https://www.aclweb.org/anthology/W11-2928。

Nederhof, Mark-Jan 和 Giorgio Satta（2008年）。《计算 PCFG 的配分函数》。载于：语言与计算研究。统一资源定位符：https://link.springer.com/content/pdf/10.1007/s11168-008-9052-8.pdf。

Nederhof, Mark-Jan 和 Giorgio Satta（2011年）。《概率同步上下文无关文法的前缀概率》。载于：计算语言学协会年会论文集：人类语言技术。统一资源定位符：https://www.aclweb.org/anthology/P11-1047。

Nivre, Joakim（2003年）。“投射性依存句法分析的高效算法”。载于：国际句法分析技术会议论文集。统一资源定位符：https://aclanthology.org/W03-3017。

Norvig, Peter（1991年）。“修正合一算法中一个普遍存在的错误”。载于：*Software: Practice andExperience* 21.2. 统一资源定位符：http://norvig.com/unify-bug.pdf。

Och, Franz Josef（2003年）。“统计机器翻译中的最小错误率训练”。载于：计算语言学协会年会论文集。计算语言学协会。统一资源定位符：http://www.aclweb.org/anthology/P03-1021。

Overton, David（2003年）。“带类型逻辑编程语言的精确且富有表现力的模式系统”。博士论文。墨尔本大学。统一资源定位符：https://www.semanticscholar.org/paper/Precise-and-Expressive-Mode-Systems-for-Typed-Logic-Overton/53a9c78610eda8ab617fa47d1f60d4c6c9f31957。

Paszke、Adam、Sam Gross、Soumith Chintala、Gregory Chanan、Edward Yang、Zachary DeVito、Zeming Lin、Alban Desmaison、Luca Antiga 和 Adam Lerer（2017年）。“PyTorch 中的自动微分”。发表于：*NeurIPS* 自动微分研讨会论文集。 统一资源定位符：
h
t
t
p://
p
apers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf。

Pereira、Fernando C. N. 和 David H. D. Warren（1983）。“作为演绎的句法分析”。发表于：计算语言学协会会议论文集。 统一资源定位符：https://aclanthology.org/P83-1021。

Pettorossi, Alberto 和 Maurizio Proietti（1994）。《逻辑程序变换：基础与技术》。载于：《逻辑编程杂志》19–20。 统一资源定位符：https://www.sciencedirect.com/science/article/pii/0743106694900280。

Plotkin, Gordon D（2004）。《结构操作语义的起源》。载于：《逻辑与代数编程杂志》60-61。 统一资源定位符：https://www.sciencedirect.com/science/article/pii/S1567832604000268。

Plotkin, Gordon D.（1970）。《归纳泛化札记》。载于：《机器智能》。 统一资源定位符：https://homepages.inf.ed.ac.uk/gdp/publications/MI5_札记_归纳_泛化.pdf。

Podelski, Andreas 和 Andrey Rybalchenko（2004）。《线性排序函数综合的完备方法》。载于：国际验证、模型检验与抽象解释会议论文集。第2937卷。施普林格。 统一资源定位符：https://doi.org/10.1007/978-3-540-24622-0_20。

Prieditis, Armand E.（1993）。“有效可采纳启发式的机器发现”。载于：《机器学习》12.1–3。统一资源定位符：https://doi.org/10.1007/BF00993063。

Purdom Jr., Paul Walton（1970）。“传递闭包算法”。载于：《*BIT* 数值数学》10（1）。统一资源定位符：https://link.springer.com/article/10.1007/BF01940892。

Qi、Peng、Yuhao Zhang、Yuhui Zhang、Jason Bolton 和 Christopher D. Manning（2020年）。“Stanza：面向多种人类语言的 Python 自然语言处理工具包”。载于：计算语言学协会年会论文集：系统演示。统一资源定位符：https://aclanthology.org/2020.acl-demos.14。

Rabiner, Lawrence R（1989年）。《隐马尔可夫模型及其在语音识别中的若干应用教程》。载于：*IEEE* 会议论文集 77.2。 统一资源定位符：https://ieeexplore.ieee.org/document/18626。

Ramakrishnan, Raghu（1991年）。《魔法模板：一种引人入胜的逻辑程序处理方法》。载于：《逻辑编程杂志》11.3–4。 统一资源定位符：https://doi.org/10.1016/0743-1066(91)90026-L。

Rastogi, Pushpendre、Ryan Cotterell 和 Jason Eisner（2016）。《利用神经上下文对有限状态转换赋权》。载于：计算语言学协会北美分会人类语言技术会议论文集。 统一资源定位符：https://aclanthology.org/N16-1076。

Reps, Thomas W.（1993）。《使用逻辑数据库进行需求驱动的过程间程序分析》。载于：逻辑数据库应用。Kluwer 工程与计算机科学国际丛书。 统一资源定位符：https://link.springer.com/chapter/10.1007/978-1-4615-2207-2_8。

Reps, Thomas W., Emma Turetsky 和 Prathmesh Prabhu（2017年）。《通过张量乘积进行牛顿式程序分析》。载于：《机器智能》程序语言系统汇刊39.2。 统一资源定位符：https://doi.org/10.1145/3024084。

Reynolds, John C.（1970）。“原子公式的变换系统与代数结构”。载于：《机器智能》。统一资源定位符：http://www.cs.cmu.edu/afs/cs/user/jcr/ftp/transysalg.pdf。

Robinson, John Alan（1965年）。“基于归结原理的机器导向逻辑”。载于：*ACM*学报 12.1. 统一资源定位符： http://doi.acm.org/10.1145/321250.321253。

Rosenkrantz, Daniel J. 和 Philip M. Lewis（1970）。“确定性左角句法分析”。载于：开关与自动机理论年度研讨会论文集。 统一资源定位符：https://ieeexplore.ieee.org/document/4569645。

Roy, Bernard（1959年）。“传递性与连通性”。载于：法国科学院院报 249.2. 统一资源定位符： https://gallica.bnf.fr/ark:/12148/bpt6k3201c/f222. 图像。

Roychoudhury, Abhik, K. Narayan Kumar, C. R. Ramakrishnan 和 I. V. Ramakrishnan（1999）。“确定逻辑程序的参数化展开/折叠变换框架”。载于：声明式编程原理与实践。 统一资源定位符：https://www3.cs.stonybrook.edu/~cram/Papers/RKRR_PPDP99/paper.pdf。

Rush, Alexander（2020年）。“Torch-Struct：深度结构化预测库”。在……里面：计算语言学协会年会论文集。统一资源定位符：https://aclanthology.org/2020.acl-demos.38。

Russell, Stuart J. 和 Peter Norvig（2020年）。 人工智能：一种现代方法（第*4*版）。培生。 统一资源定位符：http://aima.cs.berkeley.edu/。

Sarawagi, Sunita 和 William W. Cohen（2004年）。“用于信息抽取的半马尔可夫条件随机场”。载于： 神经信息处理系统进展。 统一资源定位符：
https://proceedings.neurips.cc/paper/2004/hash/eb06b9db06012a7a4179b8f3cb5384d3-Abstract.html。

Schlund, Maximilian（2016）。《半环上的不动点方程代数系统：理论与应用》。博士论文。慕尼黑工业大学。统一资源定位符：
https://mediatum.ub.tum.de/doc/1273784/1273784.pdf。

Schlund, Maximilian、Michal Terepeta 和 Michael Luttenberger（2013）。《将牛顿法付诸实践：半环上的多项式方程求解器》。收录于 国际编程逻辑、人工智能与推理会议论文集。 统一资源定位符：
https://doi.org/10.1007/978-3-642-45221-5_48。

Shafer, Glenn 和 Prakash P. Shenoy（1990）。《概率传播》。收录于：数学与人工智能年鉴2（1）。统一资源定位符： https://doi.org/10.1007/BF01531015。

Shi, Tianze、LiangHuang 和 Lillian Lee（2017年）。《基于最小特征集的基于转移的依存句法分析快速（更快）精确解码与全局训练》。收录于：自然语言处理实证方法会议论文集。 统一资源定位符： https://aclanthology.org/D17-1002。

Shieber, Stuart M.、Yves Schabes 和 Fernando C. N. Pereira（1995）。《演绎式句法分析的原理与实现》。载于：逻辑编程期刊24.1–2。统一资源定位符：
http://www.eecs.harvard.edu/~shieber/Biblio/Papers/infer.pdf。

Sikkel, Klaus（1997年）。《句法分析图式：句法分析算法规约与分析框架》。《理论计算机科学丛书》。施普林格出版社。统一资源定位符：https://link.springer.com/book/10.1007/978-3-642-60541-3。

Smith, DanielG. A. 和 JohnnieGray（2018年）。“opt_einsum——用于优化类似 Einsum 的表达式的收缩顺序的 Python 软件包”。载于：《开放源代码软件期刊》3.26。统一资源定位符：https://doi.org/10.21105/joss.00753。

Stern、Mitchell、Jacob Andreas 和 Dan Klein（2017年）。“基于跨度的神经成分句法分析器”。载于：计算语言学协会年会论文集。统一资源定位符：https://aclanthology.org/P17-1076。

Stolcke, Andreas（1995）。“一种高效的概率上下文无关句法分析算法，可计算前缀概率”。载于：计算语言学21.2。统一资源定位符：https://www.aclweb.org/anthology/J95-2002。

Strassen, Volker（1969）。“高斯消元并非最优”。载于：《数值数学》13.4。统一资源定位符：https://doi.org/10.1007/BF02165411。

Tamaki, Hisao 和 Taisuke Sato（1984）。“逻辑程序的展开/折叠变换”。载于：国际逻辑编程会议论文集。统一资源定位符：
h
t
tps://archive.computerhistory.org/resources/access/text/2022/11/102785625-05-01-acc.pdf。

Tarjan, Robert（1972）。“深度优先搜索与线性图算法”。在……里面：《*SIAM* 计算杂志》1.2. 统一资源定位符：https://doi.org/10.1137/0201010。

Tarjan, Robert Endre（1981年）。“路径问题的统一方法”。在……里面：美国计算机协会期刊 第28卷第3期。 统一资源定位符：
https://doi.org/10.1145/322261.322272。

Teitelbaum, Ray（1973年）。“通过求值代数幂级数进行上下文无关错误分析”。载于：*ACM*计算理论研讨会论文集。统一资源定位符：
https://dl.acm.org/doi/pdf/10.1145/800125.804050。

Theano开发团队（2016）。“Theano：用于快速计算数学表达式的Python框架”。载于：计算研究资料库abs/1605.02688。统一资源定位符：
http://arxiv.org/abs/1605.02688。

Tomita, Masaru（1985年）。“一种适用于自然语言的高效上下文无关句法分析算法”。载于：国际人工智能联合会议论文集。统一资源定位符：
https://dl.acm.org/doi/10.5555/1623611.1623625。

Tomita, Masaru（1986年）。“一种用于连续语音识别的高效词格语法分析算法”。载于：*IEEE*国际声学、语音与信号处理会议论文集。统一资源定位符：
https://ieeexplore.ieee.org/abstract/document/1168663。

Tomita, Masaru（1991年）。广义*LR*语法分析。施普林格。统一资源定位符：
https://www.springer.com/gp/book/9780792392019。

Ullman, Jeffrey D.（1988年）。《数据库与知识库系统原理》。第1卷。计算机科学出版社。统一资源定位符：https://www.worldcat.org/oclc/310956623。Vieira, Tim、Ryan Cotterell 和 Jason Eisner（2021年）。“搜索更高效的动态程序”。载于：计算语言学协会研究成果。统一资源定位符：https://arxiv.org/abs/2109.06966。

Vieira, Tim、Matthew Francis-Landau、Nathaniel Wesley Filardo、Farzad Khorasani 和 Jason Eisner（2017年）。“Dyna：面向机器学习应用的自我优化声明式语言”。载于：*ACM SIGPLAN* 机器学习与编程语言研讨会论文集。统一资源定位符：http://cs.jhu.edu/~jason/papers/#vieira-et-al-2017。

Vijay-Shankar, K. 和 Aravind K. Joshi（1985年）。“树附加语法的一些计算性质”。载于：计算语言学协会年会论文集。统一资源定位符：https://aclanthology.org/P85-1011。

Vijay-Shanker, K. 和 David J. Weir（1989年）。“组合范畴语法和线性索引语法的识别”。载于：解析技术国际研讨会。统一资源定位符：https://www.aclweb.org/anthology/W89-0218。

Vijay-Shanker, K. 和 David J. Weir（1990）。“组合范畴语法的多项式时间解析”。载于：计算语言学协会会议论文集。统一资源定位符：https://www.aclweb.org/anthology/P90-1001。

Vijay-Shanker, K. 和 David J. Weir（1993）。“一些受限语法形式系统的句法分析”。载于：计算语言学 19.4。统一资源定位符：https://www.aclweb.org/anthology/J93-4002/。

维特比，Andrew J.（1967年）。“卷积码的错误界及渐近最优译码算法”。载于：*IEEE* 信息论汇刊13.2。统一资源定位符：https://doi.org/10.1109/TIT.1967.1054010。

Warren, David H. D.（1983）。抽象 *Prolog* 指令集。技术报告。SRI 国际人工智能中心。统一资源定位符：http://www.ai.sri.com/pubs/files/641.pdf。

Warshall, Stephen（1962）。“关于布尔矩阵的一个定理”。载于：《计算机协会期刊》 9.1。 统一资源定位符： https://doi.org/10.1145/321105.321107。

Whaley, John、Dzintars Avots、Michael Carbin 与 Monica S. Lam（2005年）。“使用 Datalog 和二叉决策图进行程序分析”。载于：编程语言与系统。 统一资源定位符： https://link.springer.com/chapter/10.1007/11575467_8。

Whaley, R. Clint、Antoine Petitet 与 Jack J. Dongarra（2001年）。“软件的自动经验优化与 ATLAS 项目”。载于：并行计算27.1。 统一资源定位符：https://www.sciencedirect.com/science/article/abs/pii/S0167819100000879。

Wingate, David 与 Kevin D. Seppi（2005年）。“用于加速 MDP 求解器的优先级排序方法”。载于：《机器学习研究》期刊 6。 统一资源定位符：http://jmlr.org/papers/v6/wingate05a.html。

Wolfram, Stephen（2003年）。《*Mathematica* 指南》。第5版。Wolfram-Media。统一资源定位符：https://www.wolfram.com/books/profile.cgi?id=4939。

Wu, Dekai (1996)。“统计机器翻译的多项式时间算法”。载于：计算语言学协会会议论文集。统一资源定位符：https://www.aclweb.org/anthology/P96-1021。

吴世杰和Ryan Cotterell（2019年）。“字符级转导的精确硬单调注意力”。载于：计算语言学协会年会论文集。统一资源定位符：https://aclanthology.org/P19-1148。

吴世杰、Pamela Shapiro与Ryan Cotterell（2018）。“字符级转导的硬非单调注意力”。载于：自然语言处理经验方法会议论文集。统一资源定位符：https://aclanthology.org/D18-1473。

Younger, D. H.（1967年）。“上下文无关语言的识别与解析（时间 n^3）”。载于：信息与控制第10卷第2期。统一资源定位符：https://core.ac.uk/download/pdf/82305262.pdf。

Zadeh, L.A.（1965年）。“模糊集”。载于：信息与控制第8卷第3期。统一资源定位符：https://www.sciencedirect.com/science/article/pii/S001999586590241X。

Zilberstein, Shlomo（1996）。“智能系统中的随时算法应用”。载于：《*AI Magazine*》第17卷第3期。统一资源定位符：https://doi.org/10.1609/aimag.v17i3.1232。

索引

不匹配$_z(_Y)$，192

议程，81，133

代数路径问题，94

辅助定义，164

巴拿赫不动点定理，69

二元运算，38

二元关系，31

反对称， 32

非对称， 32

等价关系， 32

等价类， 32

无限下降链， 33

偏序，33

预序，33

自反 ，32

严格偏序，33对称，32全序，33传递，32良基序，33

内置关系，14，25

柯西序列，68图表，73，133完备度量空间，68条件基数，131常量折叠，16约束传播规则，111压缩映射，69压缩率，69游标技巧，77

深度限制，115

推导，59高度，60推导集合，59脊柱，189子树排序，244值，59yield，58维度，88测度，176推导集合，13

广义可逆性测试，167

可逆性测试，163

函数，34双射，34单射，34逆，34满射，34图凝缩，35强连通，35强连通分量，35拓扑排序，35拓扑排序，35图搜索问题，207

导出类型，102

距离度量，68

驱动程序，82

驱动项，134Dyna 2，270

群，38

不动点方程，60

不动点迭代，63

折叠变换 基本折叠变换，162广义折叠变换，167

输入和输出声明，143输入数据，14输入类型，102项，10，12输入，143

中间项，143

输出，143有用，120无用，120值，10，12

最小上界，68

左角变换，194

最小推导度量，183

幺半群，38

交换的，38幂等的，38

重数校正，31

参数，104

参数表达式，132

帕累托有效，52

Parikh像，52

划分，32

乘客，82

乘客，134

后验界，70

先验界，69优先级策略，81

程序，12

不动点方程，60推导集合，59内部表示，18语义等价，152强等价，152支撑集，102程序变换 部分正确性，166安全的，152语义保持，152

查询，15

显式结果流，22隐式结果流，23结果，15

规则，12

公理，14

体，12

基化，13

头部，12

所有基例的集合，14

边条件，14

子目标，13

规则度量，176

半环，13，38n折叠乘积，39n折叠求和，39自然序，66性质ω-连续，68完备，67自然有序，66性质K-闭，40闭，39交换的，39幂等的，39单调，41星运算符，39

布尔，44

向量空间中的凸集格，53

计数，45

一阶期望，46形式语言，51模糊集，45整数，45谱系，50对数非负实数，46矩阵，49最大加法，47最大乘法，47最大容量，48最小加法，47最小乘法，47非负实数，46帕累托前沿，52来源，50实数，45二阶期望，半线性集，52集合，50三值逻辑，45

半环实例

K-最佳，48K-折叠自然，46卢卡西维茨，45北极半环，47

46

热带，47

维特比，47加权形式语言，51

简单，105

简单程序，72

简单类型，107

大小参数，132

斜线项，189

专用程序，15

推测变换，191步算子，63

替换，19

替换映射，19

上确界，68

term

基项，12Herbrand 宇宙，12非基项，18

待办列表，213

轨迹，189

变换序列，178参照程序，178截断，115元组化技术，157类型程序，103展开变换 基本展开变换，162广义展开变换，166合一，19

更新，81

赋值，58

赋值函数，12保值双射，152变量消除技术，157变量，13