解析算法分析自动化（以及其他动态规划程序）

TimVieira Ryan Cotterell Jason Eisner约翰·霍普金斯大学 苏黎世联邦理工学院 约翰·霍普金斯大学tim.f.vieira@gmail.comryan.cotterell@inf.ethz.ch jason@cs.jhu.edu

arXiv:2512.23665v1 [cs.PL] 29 Dec 2025

摘要

自然语言处理领域的许多算法研究旨在高效操作复杂的形式结构。算法设计者通常希望为其提出的算法提供保证，例如，保证其运行时间或空间复杂度的上界是输入规模的某个函数。他们还可能希望确定算法所推导出的各项量所需具备的属性，以便合成高效的数据结构并检查类型错误。本文开发了一个系统，帮助程序员执行这类分析。我们将该系统应用于多种自然语言处理算法，发现它能够成功推断类型、死代码和冗余代码，以及参数化的运行时间和空间复杂度界。

https://github.com/timvieira/dyna-pi

1 引言

这项工作的目标是进一步推动自然语言处理中的声明式编程（Pereira 和 Shieber，2002；Gómez-Rodríguez 等人，2009年）。声明式编程的优势在于，它使程序员能够用高级语言描述算法，而不必担心底层实现细节，从而专注于算法本身的正确性。本文聚焦于 Dyna（Eisner 等人，2005），这是一种专为吸引自然语言处理研究者而设计的声明式编程语言。Dyna 是一种简洁的语言，可用于实现动态规划程序；长期以来，这类程序一直是许多自然语言处理问题中结构化预测的主力。具体而言，自然语言处理广泛使用基于丰富语言学形式体系的概率模型，例如有限状态转换（Mohri，1997）、上下文无关句法分析（Goodman，1999年）、依存句法分析（Eisner，1996）和温和上下文敏感句法分析（Vijay-Shanker 和 Weir，1989），其中的推理需要动态规划。这类算法往往难以正确且高效地实现。[1]

现在我们转向对这些声明式程序的静态分析——例如类型分析、运行时间复杂度分析和空间复杂度分析（Cousot 和 Cousot，1977；Pierce，2002）。这类分析对于理解未知输入数据上可能出现的各种程序行为至关重要。[2] 自动化程序分析能够帮助程序员，因为手动程序分析可能既繁琐又容易出错。在自然语言处理的背景下，对动态规划程序（例如在 Dyna 中指定的程序）进行自动化程序分析，有望帮助自然语言处理算法研究人员准确而快速地分析他们的成果。事实上，自然语言处理文献中已有一些例子表明，这类自动化分析本可以发挥作用。例如，Huang 和 Sagae（2010）推导出了一种投射式依存句法分析算法，并给出了其复杂度 $\mathcal{O}(n^7)$；后续工作（Shi 等，2017）改进了分析，发现其实际复杂度为 $\mathcal{O}(n^6)$。我们的自动化系统正确给出了 $\mathcal{O}(n^6)$ 分析。再如，Kuhlmann 等人（2011，§5.4）发现了其表格式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得出的时间复杂度为 $\mathcal{O}(n^5)$，空间复杂度为$\mathcal{O}(n^3)$，但在注意到其转移系统构建的对象总是包含两个相同的变量（这一点可以通过归纳法证明）后，他们得以将分析收紧为时间复杂度 $\mathcal{O}(n^3)$，空间复杂度为 $\mathcal{O}(n^2)$。他们

[1]当结构的分数可分解为由神经网络计算的局部项时，这些经典的自然语言处理算法仍常用于深度结构化预测（Durrett和Klein，2015年；Rastogi等人，2016；Lee等人，2016；Kim等人，2017；Qi等人，2020；Rush，2020）。[2]虽然本文不探讨这一点，但我们还要指出，静态程序分析对于编译高效代码至关重要。例如，类型推断可以让编译器为表示内部对象合成高效的数据结构，并生成用于对其进行模式匹配的高效代码。

作者注： 本文可引用为 Vieira 等人（2021）。我们于 2021年11月 将其投稿至期刊《计算语言学协会汇刊》，该期刊于两个月后有条件地接受了本文。不过，我们从未进行必要或建议的阐述性修改：谨向 TACL 的编辑和审稿人致歉。扩展版本收录于第一作者的博士论文（Vieira，2023）第 6 章。

随后修改了程序表述，以消除冗余变量。在相关程序上运行时，我们的系统会自动发现这一冗余以及更精确的分析结果。

Eisner 和 Blatz (2007) 指出，Dyna 程序可以通过多种方式自动转换，同时保留其语义，但这些转换可能会影响最坏情况下的运行时间。若要自动搜索能够改善最坏情况运行时间的转换序列，我们就需要确定最坏情况下的运行时间，例如，求得尽可能紧确的渐近界。

本文提出了一套用于自动分析 Dyna 程序的理论，并提供了一个可执行该分析的系统。同样，在自然语言处理领域，我们的自动分析可以得出关于运行时间和空间复杂度的大$\mathcal{O}$ 界限——许多自然语言处理研究者曾手动求得过此类界限。具体而言，本文作出了以下贡献：(i) 提出一种用于描述和推理类型的记法（§3）；(ii) 提出一种针对 Dyna 程序进行类型推断的新算法（§4）；(iii) 提出一种用于描述输入规模的记法；(iv) 提出一种时间和空间复杂度分析器（§5），它将自然语言处理文献中普遍采用的运行时间与空间分析技术自动化并加以改进。该技术基于 McAllester(2002)。本文还收录了自然语言处理文献中的大量示例，将其改写为 Dyna 程序并以统一的方式呈现，以证明我们提出的算法得出的分析结果与文献中记载的结果一致。

2 Dyna 中的动态规划

Dyna 是一种高级领域专用语言，可用递归关系简洁地表达动态规划程序。这里简要介绍这门语言；更完整的介绍请参阅 Eisner 等人 (2005)。下面先通过一些简单或熟悉的计算作为示例。

示例 1. 矩阵乘法加上单位矩阵。

```
a(I,K)  += b(I,J)* c(J,K)。
 a(I,I) += 1。
```

该程序定义了一组派生项 a(I,K)，其取值由其他项 b(I,J) 和 c(J,K) 决定；这些项本身也可能由类似表达式或输入推导而来。对于任意一对 (I, K)，程序将 a(I,K) 的值定义为 $\left(\sum_J \text{b(I,J)} \cdot \text{c(J,K)}\right) + 1[\text{I} = \text{K}]$。规则中使用 += 表示该值被定义为一个求和。第一条规则针对J的每个可能取值贡献一个求和项，因为J 只出现在该规则的右侧。第二条规则在 1 的情况下额外贡献一个求和项，即I = K。

Dyna 规则可以是递归的：

示例 2. 图中的最小代价路径。

```
β（J）最小化=stop（J）。 β（I）最小化= cost（I，J） + β（J）。
```

此程序将 β（I）定义为图中从节点I到任意终端节点J的最小代价路径的代价。此处，必须首先将stop（J）定义为在J 处停止时的代价，其中J是终端节点（否则应保持未定义）。此外，必须首先将cost（I，J）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）。

示例3。使用 *CKY* 进行加权上下文无关解析（*Cocke 和 Schwartz，1970年*；*Younger，1967*；*Kasami，1965*），或者更准确地说，内部算法（*Baker，1979年*；*Jelinek，1985年*）：

```
β(X,I,K) += γ(X,Y,Z)* β(Y,I,J) * β(Z,J,K)。 β(X,I,K) += γ(X,Y) * β(Y,I,K)。 β(X,I,K) += γ(X,W) *word(W,I,K)。 目标 += β(s,0,N)*len(N)。
```

此程序要求 γ、word 和 len 项编码特定内容。γ 项应表示上下文无关文法中相应产生式的权重（例如，γ(np,det,n) = 0.8表示产生式 np $\xrightarrow{0.8}$ det n）。当 W 是输入句子中位置 K 处的 word 且 I = K − 1时，项 word(W,I,K) 的值应为 1；否则应保持未定义。最后，如果句子的长度为 N，则 len(N) 的值应为 1。对于这样的输入，项 β(X,I,K) 将表示非终结符 X 在覆盖位置 (I, K] 的子串上的所有语法推导的总权重。此外，goal 将表示整个输入句子的所有语法推导的总权重。

更形式化地说，Dyna**程序** $\mathcal{P}$ 用于定义**项**的**值**。每个项都由“Herbrand 宇宙” $\mathbb{H}$中的一个元素命名——也就是一个嵌套命名元组，例如f(g(z,h(3)))，它是一个类型为f 的 1 元组，其唯一元素是一个类型为g的 2 元组。$\mathbb{H}$ 中的元素称为**基项**，因为它们不包含变量。程序 $\mathcal{P}$ 本身是由**规则**组成的无序集合，通常形式为 $h \oplus= b_1 \otimes \ldots \otimes b_K$。我们将 h称为头部，将 b_1、……、 b_K称为规则的体部。体部中的每个 b_k 都称为**子目标**。

运算 $\oplus$和 $\otimes$ 对集合 $\mathbb{V}$ 中的值进行操作，可以是构成半环的任意一对运算

（Goodman，1999；Huang，2009）。令 0 和 1 分别表示半环的加法和乘法单位元。如果某项的值未定义或没有 $\oplus$项，那么这等价于说它的值为 0，这意味着它不会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 $\langle \mathbb{R}_{\geq 0}, +, * \rangle$、布尔半环 $\langle \{\text{false}, \text{true}\}, \vee, \wedge \rangle$、最小代价 $\langle \mathbb{R} \cup \{\infty\}, \min, + \rangle$，以及维特比 $\langle \mathbb{R}_{\geq 0}, \max, * \rangle$。半环的分配律确保了对每个项的值的一种重要解释：该值是此项所有证明的和，其中每个证明的值是其公理值的乘积。本文假设程序中的所有规则都使用同一个半环。[3,4]

规则无需指定具体项。规则中的项可以包含大写变量，例如X，它们在 $\mathbb{H}$ 上进行全称量化。这使它们能够与多个项进行模式匹配。令 $X_1, \ldots, X_N$ 表示规则 r 中互不相同的变量。那么，r 的**实例化**是一个不含变量的规则 $r[X_1 \mapsto v_1, \ldots, X_N \mapsto v_N]$（也就是说，对于每个 i，将 v_i 替换 r 中所有出现的 X_i 后得到的结果），其中 $v_1, \ldots, v_n \in \mathbb{H}$。$\Gamma(r)$ 表示 r 的所有实例化构成的集合。在 $\mathcal{P}$ 中加入一条非地规则，等价于加入它的所有实例化。

Dyna 允许在规则上添加逻辑侧条件，例如，目标 += f(X) for X < 9。这是一种语法糖，相当于目标 += f(X) * lessthan(X,9)，其中每个 lessthan(a,b) 项的值在 $a < b$ 时为 1，否则为 0。逻辑表达式 ?x 的值在 x 有任意证明时为 1，否则为 0。lessthan 是内置约束的一个例子，简称内置函数。其他内置函数还包括 Z is X+Y，它会展开为 plus(X,Y,Z)。内置函数是无需在程序中定义的项，因为它们属于 Dyna 的标准库。

输入数据 $\mathcal{D}$ 表示为一组**公理**——即形如 $(h \oplus\!= v)$ 的规则，其中 $v \in \mathbb{V}$。将这些特殊规则添加到 $\mathcal{P}$ 后，会得到一个新程序 $\mathcal{P}'$。

语义。赋值 $\nu : \mathbb{H} \to \mathbb{V}$ 是一个映射，将基项映射到它们在半环中的取值。通常，ν 只为有限多个非 $\mathbb{0}$ 值赋值

[3]我们将把对 Dyna 2 的扩展留待未来工作（Eisner 和 Filardo，2011）；这些扩展会放宽这一限制。

[4]我们有一个例外：除了使用不同半环的规则外，也可以向程序中添加布尔规则。将布尔半环转换为其他半环时，我们把 false 和 true 分别映射为 $\mathbb{0}$ 和 $\mathbb{1}$。从半环转换到布尔值空间则需要使用? 运算符（稍后介绍）。

给 $\mathbb{H}$ 中的有限子集赋值。程序 $\mathcal{P}$ 的**步进算子** $\mathbf{T}_{\mathcal{P}}$ 是一个映射，它将赋值 ν 映射为修订后的赋值 ν'。本质上，$\mathbf{T}_{\mathcal{P}}$ 利用 $\mathcal{P}$ 的规则计算出新的赋值。更形式化地说，对于任意 $h \in \mathbb{H}$，

$$\Big[\mathbf{T}_{\mathcal{P}}(\nu)\Big](h) \stackrel{\text{def}}{=} \bigoplus_{\substack{r \in \mathcal{P}, \\ (h \oplus\!= b_1 \otimes \cdots \otimes b_K) \in \Gamma(r)}} \nu(b_1) \otimes \cdots \otimes \nu(b_K) \quad (1)$$

要在数据 $\mathcal{D}$ 上运行 $\mathcal{P}$，我们将 $\mathcal{P}'$ 定义为 $\mathcal{P}$ 与 $\mathcal{D}$ 的连接，并求取 $\mathbf{T}_{\mathcal{P}'}$ 算子的一个不动点 ν^*。[5] 换句话说，ν^* 是由 $\nu^* = \mathbf{T}_{\mathcal{P}'}(\nu^*)$给出的一组非线性方程的解。我们常将项 x 的值写作 $[\![x]\!]$ def= $\nu^*(x)$。该方程组可能有多个不动点，也可能没有不动点。实际上，某种前向链方法会计算出一个合适的不动点 ν^*。本文将重点讨论朴素自底向上算法：它将 ν 初始化为常数 0 函数，然后反复更新 ν，使其变为 $\mathbf{T}_{\mathcal{P}'}(\nu)$，直至收敛到某个值$\nu^*$。[6] 我们将用 $\mathcal{P}(\mathcal{D})$作为简写，表示以这种方式趋近的最终赋值 $\nu^* = \mathbf{T}_{\mathcal{P}'}(\mathbf{T}_{\mathcal{P}'}(\cdots \mathbf{T}_{\mathcal{P}'}(\mathbb{0}) \cdots))$。中间赋值实质上表示动态规划算法运行时备忘录表（例如，解析时的解析图表）的演变。在布尔半环中，始终存在唯一的最小不动点 ν^*，并且对于每个项 x，$\nu(x)$都能在有限时间内达到 $\nu^*(x)$。然而，有些程序中，并不存在一个有限的时间点，使所有项都已收敛。

实用工具。最后，我们定义一些稍后会用到的其他术语和实用函数。令 nz(ν) 表示 ν 的支撑集，即 ν 中具有非 $\mathbb{0}$ 值的项的集合。令 head(r) 和 body(r) 分别表示规则 r 的头部和体部。令 vars(·) 表示一个（可能包含变量的）项中所包含的变量集合，例如，vars(f(g(X), 4,X)) $\mapsto$ {X}。项可以通过结构相等约束与其他项建立相等关系，这类约束用等号运算符表示，例如 f(X) = f(3)。此类约束可以使用合一算法高效求解，unify（例如 Robinson (1965)；Martelli 和 Montanari (1982)）会返回一个代换

[5] $\mathcal{D}$ $\mathbb{V}$ ν v $h \oplus\!= v$. $\mathbb{1}$ v $v \in$ 由于包含如下形式的规则 () 其中 是一个常量，因此我们必须将 () 解释为在 0 中返回 。

[6] 这可以改进为半-朴素自底向上算法（Ullman，1988；McAllester，2002；Eisner 等人，2005），但这种差异对我们的目的并不重要。

映射 θ；如果不存在解，则可能等于 $\emptyset$ 。例如，unify(f(Y, 4), f(g(X), Z)) 的解为 {Y $\mapsto$ g(X), Z $\mapsto$ 4}。[7] 当我们将该代换应用于两侧（使用函数 subst ）时，项会变为相等，subst(f(Y, 4) θ) = subst(f(g(X), Z) θ)。另一方面，f(X,g(X))= f(3,g(4)) 是不可满足的。记号 θ_i 返回集合 i 中变量对应的子映射。记号 θ(X) 返回映射中变量 X 的值（如果 θ 中不存在对应值，则返回 X 本身）。最后，我们使用函数 fresh，将项、规则或整个程序中的所有变量重命名，以避免变量名冲突。

3 类型分析

一种**类型**是 $\mathbb{H}$的子集，也就是说，是由基项组成的集合。本文将类型表示为有限个**简单类型**的并集。每种简单类型通过一个非基项来指定，其中的变量本身可能受特定类型约束，或受到其他方式的约束。例如，`f(3,5)` 是一个基项，但{`f(3,J)`} 是使用变量指定的无限项集合，而{`f(3,J)` | `g(J)`, `3 < J`} 或 {`f(I,J)` | `I < J`} 是一个其描述涉及约束的无限集合。如下文所示，这些集合（简单类型）中的每一个都可以通过一条受限 Dyna 规则（在布尔半环中）方便地指定。

在编程语言领域，不实际运行程序并基于具体数据推理程序行为，称为**抽象解释** (Cousot 和 Cousot, 1977)。随后生成编译代码时，可以利用有关代码操作的对象将具有特定类型的任何证明：既可以将该对象高效地表示为专用于该类型对象的类的实例，也可以加速模式匹配等测试。如果程序存储并遍历某一类型的不同对象集合，那么了解该类型在具体数据上的基数，就能得出程序空间和运行时间需求的界限（§5）。

本节将介绍 Dyna 程序中的抽象解释。我们不计算赋值 $\mathcal{P}(\mathcal{D})$，而是计算其支撑集的上界。换言之，我们推断出一种类型，该类型保证包含所有具有非-⓪值的项。实际上，我们无需完全了解输入 $\mathcal{D}$，便可推断出这种项类型：用户将使用本节介绍的表示法，为输入项提供一种类型；基于这一保证，我们还将推断出一种保证包含所有派生项的类型。

本节介绍我们的类型规范语言，以及如何使用它来推理可能存在的项。继续使用 CKY 示例，我们提供输入数据的规范。不过，由于我们关注的是某个项可能具有非 ⓪ 值的可能性，而不是找出它的具体值，因此我们先按下图所示转换程序。

例4. 回顾一下，要分析的程序是

```
9 参数: word; 长度; γ.10 β(X,I,K) += γ(X,Y,Z)* β(Y,I,J) * β(Z,J,K).11 β(X,I,K) += γ(X,Y)* β(Y,I,K).12 β(X,I,K) += γ(X,W)*word(W,I,K)。13 目标 += β(s,0,N)* 长度 (N)。
```

由于尚未指定任何规则来定义`word`、`长度`和 `γ` 项，人们可能会认为这些项没有 `+=` 加项，因此值为`0`。然而，`params` 声明表明，这些项将由程序输入定义。[8]

上面的 CKY 程序 $\mathcal{P}$ 可以自动“布尔化”，从而得到下面的**类型程序** $\mathcal{P}$。[9]符号 `:-` 和，传统上表示布尔半环中的 $\oplus$=和 $\otimes$ 运算。

```
14 参数: word; 长度; γ.15 β(X,I,K) :- γ(X,Y,Z), β(Y,I,J), β(Z,J,K).16 β(X,I,K) :- γ(X,Y), β(Y,I,K).17 β(X,I,K) :- γ(X,W), word(W,I,K). 18 目标 :- β(s,0,N), 长度(N). ——
```

其意图是，`β(X,I,K)` 对于所有`X,I,K` 满足 `β(X,I,K)`在 CKY 程序中具有 anon- `0` 值的情况，在类型程序中都将取真值。当然，每个程序都需要输入：如果在数据 $\mathcal{D}$上运行 $\mathcal{P}$，那么 $\mathcal{P}$ 必须在其布尔化版本 $\mathcal{D}$ 上运行。赋值 $\mathcal{P}(\mathcal{D})$ 现在实际上指定了程序可能推导出的项的一个推断类型。[10]

[7]我们允许合一 将替换映射作为可选的第三个参数，以便算法能够串联合一调用的结果。

[8]Eisner 和 Filardo (2011)通过“dynabase 扩展”机制为 Dyna 程序提供输入，该机制可以用任意新规则扩充程序，但本文采用更简单的方法，将 Dyna 程序视为指定了某些半环加权关系上的函数。[9]此外，翻译会将任何常量值映射为 false 或 true，效果相当于应用了? 运算符（§2）。此外，由于类型程序的值已经是布尔值，因此可以省略?运算符。[10]例如，$\mathcal{P}(\mathcal{D})$(β(np,3,5)) = 为 true，当且仅当$\mathcal{P}(\mathcal{D})$(β(np,3,5)) ≠ ⓪——甚至可能当

为了抽象掉特定的输入数据 $\mathcal{D}$，用户可以直接指定上一段中的布尔化输入数据 $\mathcal{D}$。这 $\mathcal{D}$可以是 nz($\mathcal{D}$) 的任意上界，并作为未知输入 $\mathcal{D}$ 的类型声明。由 $\mathcal{P}(\mathcal{D})$ 定义的类型随后会继续包含 nz($\mathcal{P}(\mathcal{D})$) 中的所有项，对于任何且每一个满足此上界的数据集 $\mathcal{D}$ 都如此。例如，即使 β(np,3,5) 可能具有非布尔半环值，$\mathcal{P}(\mathcal{D})$ 仍会为 β(np,3,5) 赋予真值。

CKY 的输入类型声明示例如下，这是一个布尔程序 $\mathcal{D}$，其中每条规则分别指定一种简单类型：

```
19 参数: k; w; n。20
word(W:w,I:n,K:n) :-I <K。21
len(N:n).22 γ(X:k,Y:k,Z:k).23 γ
(X:k,Y:k).24 γ(X:k,W:w)。
```

§3–3 指定了 5 个集合（简单类型），它们的并集是一种包含所有输入项的类型。§3 中的记法 word(W:w,I:n,K:n) :- I <K 是 word(W,I,K) :- w(W), n(I), n(K), I < K 的简写，表示集合 { word(W,I,K) |w(W), n(I), n(K), I < K} 中的项在输入数据 $\mathcal{D}$ 中可能具有非 ⓪ 值。像 n 这样的谓词可以通过附加语句来定义，例如 n(I) :- 0 ≤ I ≤ 20，这会定义长度为 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，我们省略了这类定义，将 n 留作参数。因此，上述程序 $\mathcal{D}$ 实际上定义了一种参数化类型，就像许多编程语言中的 List[α] 一样，它是一种依赖于具体选择（例如 α = string）的参数化类型。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。具体的输入类型形式为 $\mathcal{D}$(k, w, n)。

请注意，虽然 $\mathcal{D}$ 写作若干简单类型的并集，但 $\mathcal{P}$ 并非如此。我们的提升式前向链接算法（§4）将推导出以下规则，为$\mathcal{P}$($\mathcal{D}$(k,w, n) $\mathcal{P}$)的派生项指定简单类型：

```
25 β(X:k,I:n,K:n) :- I < K.
26 goal.
```

将这些规则添加到 $\mathcal{D}$ 「它声明了输入 项的类型），即可得到涵盖所有非-⓪-值项的类型的理想描述。这

$\mathcal{P}(\mathcal{D})$(β(np,3,5)) = ⓪ 例如，如果 0 可以由在 ⊕下相加的非 0 值产生。这类高估可能出现，是因为布尔程序不会追踪实际值。

个扩展后的程序使用与 $\mathcal{D}$ 相同的参数，并为 $\mathcal{P}$（$\mathcal{D}$(k, w, n)）提供一个上界。该上界意味着我们只会构造 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，该上界还表明（令人安心的是）可能构造出一个目标项。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)，那就会显得宽松得可疑，因为 I 永远不应是非终结符。

总之，我们的类型分析器接收程序 $\mathcal{P}$ 以及参数化输入类型 $\mathcal{D}$，并计算出参数化项类型，例如 §3–3，用于描述由 $\mathcal{P}$派生的项。实际上，它会在输入上界 $\mathcal{D}$上运行布尔化程序 $\mathcal{P}$。然而，由于项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。

4 抽象前向链

现在我们将介绍类型推断算法，它将布尔半环上的普通前向链（§2）加以推广。在普通前向链中，每一步的赋值 ν 都是有限个基项的集合。我们的推广允许每一步的 ν 都是有限个简单类型的集合（因此是 ν^*）。这项推广是必要的，因为我们不再有有限上界来限定用于初始化前向链的输入数据：参数 n 是一个未知关系，而 <约束是一个无限关系。因此，我们指定参数化简单类型，为未知且可能无限的输入数据提供上界。前向链会推导出新的简单类型，这些类型也可能引用诸如 n 和 <这样的关系。事实上，正如我们在 §3中所见，这些关系可能会出现在结果 ν^*中。

我们在§4.1中介绍抽象前向链算法，并在§4.2–4.5中探讨约束累积带来的挑战。

4.1 类型推断算法

为了推广布尔半环中的前向链，我们将 §2 中的赋值映射表示方式加以推广。对于这个半环，赋值映射通常会表示为一个显式集合，其中包含值为 true 的基项。由于我们需要表示未知且无限的集合，这种表示方式无法满足我们的需求。因此，我们将把赋值映射表示为

有限多个简单类型的并集。一个示例$\mathcal{S}$来自 §3 的 CKY 示例是§3–3上的简单类型集合，它给出 ν^*。

从某种意义上说，§3–3 已经间接描述了 ν^*，但并非以简单类型集合的形式。回想一下，简单类型是形如$\mathsf{h} :- \mathsf{c}_1, \ldots, \mathsf{c}_M$的表达式，其中每个$\mathsf{c}_m$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。注意，h不能出现在任何其他简单类型的体部中。这种形式确保每个简单类型都是独立完整的，并且可以实现为面向对象的类。

下面的伪代码实现了步进操作（1），作用于有限个简单类型的集合 $\mathcal{S}$：[11]

▷回想一下，实用函数 unify、fresh 和 subst 在 §2 中定义。▷抽象步进算子；针对当前估计类型 $\mathcal{S}$，展开 $\mathcal{P}'$ 的所有规则（即 $\mathcal{P}$ 和 $\mathcal{D}$）。返回一个表示类型 $\supseteq \mathcal{S}$ 的简单类型集合。 def step($\mathcal{S}$): return $\bigcup_{r\in\mathcal{P}'}$ step_rule($\mathcal{S}, r$)▷针对简单类型 $\mathcal{S}$，展开 r 中的子目标。返回一个简单类型集合。 def step_rule($\mathcal{S}, r, k$=1, θ={}, $\mathcal{C}$=$\emptyset$):▷ k: r中当前子目标的索引▷ $\mathcal{C}$: 来自子目标的延迟约束集合 $< k$▷ θ: 替换映射（h :- $b_1, \ldots, b_K$） $\leftarrow r$▷基本情况；没有更多子目标时，如果 $k = K+1$: return {subst((h :- $\mathcal{C}$) θ)}▷递归处理剩余子目标 return $\bigcup$ step_rule($\mathcal{S}, r$, $k+1, \theta', \mathcal{C}' \cup \mathcal{C}$)$_{\langle\theta',\mathcal{C}'\rangle\in}$lookup($\mathcal{S}, b_k, \theta$)▷返回 $\mathcal{S}$ 中与 b 合一的简单类型集合（并应用替换映射）。 def lookup($\mathcal{S}, b, \theta$={}): 如果 b 是参数或内置函数： return {$\langle\theta, b\rangle$}return {$\langle$unify(b, h', θ) {$c'_1, \ldots, c'_M$}$\rangle$对于 (h' :- c'_1, ... , c'_M) $\in$ fresh($\mathcal{S}$)}

类型推断通过对step 操作（下文给出代码）进行不动点迭代来完成。应将其视为第§2节中前向链算法的“提升版”：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。执行过程中，这些简单类型中出现的约束会被传递。这类约束在约束逻辑编程文献中称为“延迟约束”，以区别于诸如 3 < 2 这类可立即求值并替换为真或假的约束。

然而，反复应用 step 所带来的挑战在于，约束可能会不断累积。对于递归规则，约束可能无限累积，因为它们实际上会展开一条无限长的递归调用链，正如§4.2中将要说明的那样。为防止发散，我们提出以下策略：约束松弛（放宽；§4.2）和约束传播（传播；§4.3）。我们将在§4.5讨论终止性。下面给出整体算法：

1. **定义**step*(): 2. $\mathcal{S} \leftarrow \emptyset$ 3. **当**true: 4. $\mathcal{S}' \leftarrow$放宽(传播(step($\mathcal{S}$))) 5. **如果** $\mathcal{S}' = \mathcal{S}$: **返回** $\mathcal{S}$ 6. $\mathcal{S} \leftarrow \mathcal{S}'$

4.2 松弛

考虑以下输入类型 $\mathcal{D}$ 对于示例2。

27 参数: n。 28 β(S): -
stop(S)。 29 β(S) :- edge(S,S'), β
(S')。30 stop(S) :- n(S)。 31
edge(S,S') :-n(S),n(S')。

应用 step 操作一次（该操作会重命名变量以避免冲突），我们可以推导出 β 项的以下简单类型（其他类型未显示）：

```
32 β̄(S1) :- n(S1).
```

第二次应用step 后，我们可以推导出

```
33 β̄(S1) :- n(S1).
34 β̄(S1) :- n(S1), n(S2).
```

请注意，在第二条规则中，$\mathsf{n}(\mathsf{S}_2)$ 作为递归调用留下的约束仍然存在。按照布尔程序的（类似 Prolog 的）语义，变量 S_2 是存在量化的，因为它是一个“局部”变量，只出现在规则体部中。应用step k 次后，我们可以推导出

```
35 β̄(S1) :- n(S1).
36 β̄(S1) :- n(S1), n(S2).
   ⋮
37
38 β̄(S1) :- n(S1), n(S2), n(S3), …, n(Sk).
```

这个过程通过不断展开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（即描述所有能沿着任意长度的路径到达终止状态的顶点的类型）。我们消除先前递归层级所累积约束的策略是，丢弃（即放宽）任何引用局部变量的约束：

[11]请注意，代码使用了一组 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，如果存在一对规则 r, r' 满足 $r' \subseteq r$，我们就保留 r。这确保了（除其他事项外）在变量重命名后等价的规则不会同时出现在集合中。这也有助于实现收敛。

定义 1 (放宽)**.** 设$h :- c_1, \ldots, c_M$ 是一个简单类型。操作 放宽会丢弃每个引用了在h中未出现的变量的延迟约束c_m 。换言之，我们得到一个头部相同的新简单类型，但其约束子集如下$\{c_m \mid vars(c_m) \subseteq vars(h)\ m \in \{1, \ldots, M\}\}$。若要放宽一个简单类型集合，只需将放宽映射到整个集合上。

应用放宽到步骤(§4.2–4.2)中的第二次S_2, ，我们看到这会删除第二个简单类型中的第二个约束，因为它依赖于非头部变量：

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

我们可以合并这些规则，因为它们是重复的。现在我们得到以下类型

```
β̄(S1) :- n(S1).
```

现在我们得到了一个不动点：本次迭代的类型（松弛后）等于上一次迭代的类型（§4.2）。

为什么放宽是有效的？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。

现在我们来看一个放宽会导致类型过于宽松的例子。回顾 CKY_(例 4)。来自 $\mathcal{D}$ 的简单类型会在第一个步骤中添加到 $\mathcal{S}$，例如

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

在下一个步骤中，§3 来自 $\overline{\mathcal{P}}$

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

为 β提供一个初始类型：

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

之后放宽，我们得到

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

然而，在下一步，§3 来自 $\mathcal{P}$

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

将把 $\mathcal{S}$ 扩展为以下类型：

```
β̄(X,I,K) :- k(X), k(Y), k(Z),
   n(I), n(J), n(K), I < J, J < K.
```

放宽将删除局部变量上的所有约束：

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

遗憾的是，松弛导致顺序信息（<）丢失了。在下一节中，我们将看到如何结合 `I < J` 和 `J < K` （在放宽它们）来推断 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。这将使分析器能够推断出 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 I < J、J < K，第三代具有例如 I < J_1，J_1 < J_2，J_2 < J_3，J_3 < K，等等）。

4.3 约束传播

约束传播规则（Frühwirth，1998年）通常用于约束逻辑编程，例如 ECLiPSe（Apt 和 Wallace，2007）、计算机代数系统，例如 Mathematica（Wolfram，2003），以及定理证明器，例如 Z3（de Moura 和 Bjørner，2008）。在我们的设定中，它们为用户提供了一种机制，用于提供关于类型参数和内置约束的领域特定知识。它们用于在推理过程中推导出新的约束，例如`I < K` 上述内容。

下面，我们通过运行示例给出约束传播规则的动机示例，并说明它们为何有用。

- 在示例3中，语法起始符号 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：

```
k(s)  <==为真。
n(0)  <==为真。
```

- 断言k 和 n 类型互不相交：

```
fail <== k(X), n(X).
```

这里 fail 是一种永远无法满足的特殊约束（矛盾）。如果 k(X) 和 n(X) 同时出现在简单类型的体部中，那么该简单类型就是空的，可以将其删除。请注意，§4.3–4.3 足以从 n(s) 或k(0) 推出矛盾（即二者均为假）。

- 断言类型d 是类型c的子集：

```
c(X) <== d(X).
```

- 断言 < 是严格偏序：

```
(I < K) <== (I < J), (J < K).
fail <== (I < I).
```

这解决了 §4.2 中的问题示例。

- 要定义递归类型，例如非终结符列表[12] （即类型k）：

[12]方括号竖线记法是处理列表的一种简写方式（借自 Prolog （ Colmerauer 和 Roussel，1996 ））。像 `[1,2,3]` 可展开为 `cons(1,cons(2,cons(3,nil)))`。记法 `[X, Y|Rest]` 将匹配为X=1, Y=2, Rest=[3]。

```
57 ks([]) <== 真。 58 k(X)
 <==ks([X|Xs])。 59 ks(Xs)
 <== ks([X|Xs])。
```

抽象前向链中使用的传播方法（步骤*）会运行标准的约束传播算法（Frühwirth，1998年）——这不过是另一种前向链算法——用于推导简单类型体部中约束的所有传递后果。传播规则应当选择得既可靠又相当完备，同时仍能在合理的时间内终止。

4.4 移除冗余简单类型

我们可以从 s 中移除一个简单类型 $\mathcal{S}$，前提是存在一个超类型 $t \in \mathcal{S}$，也就是说，$s \subseteq t$。下面给出一种（不完整的）方法，用于检测此类子类型关系，该方法利用了传播。

定义 子类型(s, t):

▷($s \subseteq t$) $\Leftrightarrow$ ($s \cap t$) $= s r \leftarrow$ 交集(s, t)$R \leftarrow$ 传播(r)$S \leftarrow$传播(s) 返回 $R = S$ ▷

在重命名意义下相等 定义 交集(s, t):$\theta \leftarrow$ 合一(头部(s),头部(t))$s' \leftarrow$ 替换(s, θ)$t' \leftarrow$ 替换(t, θ) 返回 (头部(s') :- 体部(s'), 体部(t'))

请注意，如果 传播 规则不完整，此测试可能无法检测出子集关系。（例如，为确保终止性，我们可能会省略会反转 §ks([X|Xs]) <== k(X), ks(Xs) 的规则，此规则反转了 §4.3–4.3。）未来的系统或许会使用更强大的定理证明器，例如 Z3（de Moura 和 Bjørner，2008），来检测子集关系。

4.5 终止性

为防止抽象前向链无限构建出与越来越大的项相对应的简单类型，我们可以将它们截断至用户可选指定的 深度限制 D。此深度限制会按以下方式重写简单类型：将头部中深度为D 的任何子项替换为一个新的、无约束的变量。此策略可确保构建出的放宽简单类型数量始终有限。因此，只需确保收敛即可。然而，截断通常会导致严重高估，因为自由变量的类型为 $\mathbb{H}$。

5 自动空间与时间分析

现在，我们将开发一种自动化方法，用于分析给定数据 $\mathcal{D}$时 Dyna 程序 $\mathcal{P}$ 的空间和时间复杂度。给定与 $\mathcal{P}$相关联的类型 $\mathcal{S}$，关键是要找到其基数的参数化上界。然后，我们将利用这些上界，预测 $\mathcal{P}(\mathcal{D})$的前向链执行运行时间的参数化 $\mathcal{O}$界。

5.1 空间分析

我们将用 $|s|$ 表示集合 s 的真实基数，并用 $\lceil s \rceil \geq |s|$表示一个上界。

要界定 $\mathcal{P}(\mathcal{D})$ 的空间复杂度，只需界定项类型 $\nu = \bigcup_{s \in \mathcal{S}} s$ （§4）的总大小。我们的高层方法是$\lceil \nu \rceil$ 定义 $= \sum_{s \in \mathcal{S}} \lceil s \rceil \geq \sum_{s \in \mathcal{S}} |s| \geq |\nu|$，[13] 其中，简单类型 s 的大小 $|s|$ 由其变量的满足赋值数量决定，而 $\lceil s \rceil$ 是该数量的一个上界。

众所周知，统计满足一组约束的变量赋值可能会带来很大的计算开销。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。本节余下部分将探讨如何利用用户提供的有关参数化约束的信息来构造一个界 $\lceil s \rceil$ 。

5.2 输入规模说明

考虑简单类型 γ(X,W) :- k(X), w(W)。可以得到一个直接的上界：$|\gamma$(X,W) :- k(X), w(W) $| \leq \lceil$k(X)$\rceil \cdot \lceil$w(W)$\rceil = k \cdot w$其中，k 和 w 是用户指定的大小参数。（我们将很快说明如何指定大小参数。）然而，当同一变量存在多个约束时，情况会变得更加复杂。考虑简单类型

```
60 β(X,I,K) :- k(X), n(I), n(K), I < K.
```

显然，至多有 $k \cdot n \cdot n$ 个赋值能够满足前 3 个约束。此外，施加第四个约束 I <K 只会减少这个数量——尽管第四个约束

[13] 如果 $\mathcal{S}$ 中的简单类型 $\mathcal{S}$ 存在大量重叠，那么最后一个不等式将会比较宽松，因为求和会对重叠区域重复计数。移除子类型（§4.4）可以处理最严重的情况，但未来的工作可以考虑将 $\mathcal{S}$ 转换为简单类型的划分，如下所示。将任意一对存在重叠的简单类型 s 和 t 替换为$s \cap t$、 $s \setminus t$以及 $t \setminus s$。重复此过程，直到不存在任何一对 s 和 t 重叠。集合差 $\setminus$ 可以使用基本约束的否定版本来实现。

单独来看 会有无穷多个满足赋值（$|\texttt{I} < \texttt{K}| = \infty$）。

我们将说明如何根据序列约束$\texttt{c}_1$、…、$\texttt{c}_M$推导出上界。这些约束位于简单类型 s 的体部（包括传播所添加的任何约束，因为它们可能会改善上界）。重新排列约束不会影响满足赋值的数量；不过，这些约束的 $M!$ 种排列顺序可能各自得出不同的上界，因此我们将选取其中最紧的上界。

这个想法是使用**条件基数**。令V为任意变量集合。虽然c 可能有许多变量赋值能够满足，但其中只有一部分与对 V的给定赋值相容。事实上，假设用户提供了一个数值$\lceil \texttt{c} \mid V \rceil$，使得对 V的任何赋值都无法扩展为超过$\lceil \texttt{c} \mid V \rceil$ 个对vars(c)的赋值。情况$\lceil \texttt{c} \mid V \rceil = 1$ 尤为重要：如果$\texttt{vars}(\texttt{c}) \subseteq V$，那么这必然成立；否则，它断言存在函数依赖。

对于形式为 s h :- $\texttt{c}_1$, …, $\texttt{c}_M$的简单类型，我们可以得到 s 的基数上界：$|s| \leq \prod_{m=1}^{M} \lceil \texttt{c}_m \mid V_{<m} \rceil$，其中 $V_{<m} = \bigcup_{i=1}^{m-1} \texttt{vars}(\texttt{c}_i)$。[14] 更一般地，给定对某个变量集合 s 中部分变量的赋值，V 的$\prod_{m=1}^{M} \lceil \texttt{c}_m \mid V \cup V_{<m} \rceil$条件基数 的上界为。对所有排序取最小值，即可得到最终上界 $\lceil s \mid V \rceil$（其中 $\lceil s \rceil \stackrel{\text{def}}{=} \lceil s \mid \emptyset \rceil$）。

但是，这些上界$\lceil \texttt{c} \mid V \rceil$ 从何而来？考虑内置约束 times，它是一个定义在三个整数上的无限大关系。由于times 具有函数依赖，我们可以在以下条件下取$\lceil \texttt{times(X,Y,Z)} \mid V \rceil = 1$：$V$ 至少包含 {X,Y,Z} 中的两个。我们允许用户用以下记法声明这个条件基数：

```
61 |times(+X, +Y, Z)| ≤ 1.   62 |
times(X, +Y, +Z)| ≤ 1.      63 |
times(+X, Y, +Z)| ≤ 1.
```

例如，第三行表示，给定X 和 Z的值后，times(X, Y, Z) 约束最多只会由一个Y值满足。

给定所有这类声明，我们得到$\lceil \texttt{c} \mid V \rceil$，其值为与 c 合一且其变量标记为 + 的变量都属于 V 的条件基数声明所提供的上界中的最小值。这包括一个隐式声明，即$\lceil \texttt{c} \mid \texttt{vars}(\texttt{c}) \rceil \leq 1$。（另一个隐式声明是$\lceil \texttt{fail} \rceil \leq 0$。）

[14]前提是头部 h 中的所有变量都在体部的某处受到约束。如果不是，$|s| = \infty$。

同样的记法也适用于限定输入关系的上界，例如示例 word 关系 4中的关系。

```
64 |word(W, +I, K)| ≤ 1.
65 |word(W, I, +K)| ≤ 1.
66 |word(W, I, K)| ≤ n.
```

第一条声明表示，每个位置W处最多有一个 word，且它与单个结束位置 K 配对，并从每个位置开始I。第二条声明与此相反。第三条声明表示，输入句子中总共最多有 n 个 word，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：

```
67 |k(X)| ≤ k.   |n(I)| ≤ n.   |w(W)| ≤ w.
```

它将非终结符的数量界定为 k，将句子位置界定为 n，将终结符界定为 w。这里，k、n 和 w 均被解释为符号。我们将这些符号组合起来，形成一个**参数化表达式**，为程序中项的基数提供上界。例如，正如我们之前所见，$\lceil \beta(\texttt{X},\texttt{I},\texttt{K}) \rceil = kn^2$。将简单类型的大小相加后，我们的系统便能计算出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；在构造依赖顺序的上界时，可以将其与$\texttt{c}_1$, … $\texttt{c}_M$ 的子序列匹配。例如，|even(X),prime(X)| ≤ 1表示偶素数只有一个，而|γ(X,Y), k(Y)| ≤ g 表示非终结符X 不能改写为超过 g 个不同的非终结符Y。（它也可能改写为更多终结符符号Y，但这些不符合k(Y) 约束。）

5.3 时间分析

假设 Dyna 程序由 Eisner 等人（2005）的前向链算法维护两种索引数据结构：**图表**和**议程**。图表将 ν 当前的估计表示为一种数据结构，它把 $\mathbb{H}$ 中的基项（具体来说，是项类型 $\bigcup_{s \in \mathcal{S}} s$中的基项）映射到 $\mathbb{V}$中的值。（任何不在图表中的项，其值都视为0。）议程（或“工作列表”）是一个队列，其中存放着要应用于基项值的更新。当 b 的一项更新出队时，其值会在图表中更新。随后，这一变化会传播到其他项 h。对这些 h 项的更新会被放入议程，稍后才执行。关键在于，受影响的 h

前向引用：另请参阅 Eisner（2023）

通过匹配找到项 b （**驱动项**）与任意规则的任意子目标 $h \oplus= \cdots$，然后查询图表，找出与该规则其他子目标匹配的项（**乘客项**）。

将更新从给定的驱动项传播 b经过一条规则 r 要求我们找出所有满足条件的赋值，使其符合体部(r) 并且与将驱动项子目标绑定到 b一致。尽管 r 不是类型，但其体部中的子目标（例如 $\gamma(X,Y)$ 确实会对X 和Y施加约束——即约束 $\gamma(X,Y)$，这是我们通过类型分析（§3）得到的；只有当 $\gamma(X,Y)$ 在图表中可能有值时，该约束才成立。

假设我们已将驱动项 b 与 r 的一个子目标匹配，并且已将 0 个或更多其他子目标与从图表中检索到的乘客项匹配。这些匹配已经实例化了规则中的一些变量 V。对于每次子目标匹配，我们还可以通过静态类型分析（§4）得知，该匹配涉及的变量必须满足某些约束 $\mathcal{C}$。如果我们决定将 $\gamma(X,Y)$作为下一个要与图表匹配的子目标，那么我们最多会得到 $N \stackrel{\text{def}}{=} \lceil \mathcal{C}, \gamma(X,Y) \mid V \rceil$个答案。尽管 γ 不是内置约束或参数化约束，我们仍可用上一节介绍的方法计算这个上界：只需通过将所有简单类型 $s \in \mathcal{S}$ 的头部与 $\gamma(X,Y)$合一，来细化项类型 $\mathcal{S}$，然后求出这个细化后类型 $\mathcal{S}'$的条件基数上界。

得益于对图表进行索引，获取并处理 N 个答案的运行时间仅为 $\mathcal{O}(1+N)$（其中 1 表示即使没有答案，也会尝试执行查询所产生的开销）。然后，对于每个答案，我们都必须继续以所有可能的方式匹配其余乘客项。

以下伪代码对此作了明确说明。有两点细节值得指出。首先，$\mathcal{S}$ 是析取的。我们从先前已匹配的乘客项中累积的约束 $\mathcal{C}$ 取决于覆盖这些乘客项的具体简单类型，因此我们必须遍历所有可能性。其次，虽然驱动子目标由议程选定，但其他子目标的实例化顺序由我们决定。伪代码总是选择下一个子目标，使其在给定先前已匹配乘客项的简单类型时，能够将运行时间上界降至最低。换言之，我们没有遵循规则 r(Eisner 等人，2005）中规定的固定子目标顺序，而是利用静态分析生成更灵活、高效的代码。

假设每个项最多只出栈一次。通过拓扑排序可以做到这一点，前提是没有任何项传递性地依赖于自身的值（Goodman，1999年）。那么，我们最终得到的运行时间上界为$\sum_{r\in\mathcal{P}} \sum_{b\in\text{体部}(r)} \lceil b \rceil \cdot \text{运行时间}(r, \{b\})$。

▷此递归函数（为提高效率，应进行记忆化）会在给定某些子目标（无序集合前缀）已完成地面化的情况下，告知我们某条规则的最优“后缀运行时间”；其中 $\mathcal{C}$ 是由此累积得到的合取约束集合。 *def runtime(r, prefix* $=\emptyset, \mathcal{C}=\emptyset, \theta=\{\}$*):*
▷ 基本情况：所有子目标均已完成地面化 *if body(r)* $=$ *prefix: return 1 best* $\leftarrow \infty$ $V \leftarrow$ *vars(prefix)* ▷ 由较早的子目标绑定的变量▷ 对下一个子目标的选择进行优化 p从 $p \in$ *body(r)* $\setminus$ *prefix:* 中剩余子目标的集合里
$t \leftarrow 0$ ▷ 累积总后缀运行时间（给定选择 p），用于 $\langle \theta', \mathcal{C}' \rangle \in$ *lookup(S, p, θ):* ▷ 从 *§4* 查找
$\mathcal{C}'' \leftarrow$ *propagate*$(\mathcal{C} \cup \mathcal{C}')$ 若在 $\mathcal{C}''$ 中失败：继续
$q \leftarrow \lceil \mathcal{C}', p \mid V \rceil$ ▷ 乘客项的最大数量▷对剩余子目标递归调用t += $q \cdot$*runtime(r, prefix* $\cup \{b\}, \mathcal{C}'', \theta'$*)*
best min= t*return* 1+*best*

6 结论

许多自然语言处理算法都可以用 Dyna 表达；我们提出的分析器可以自动完成自然语言处理论文中许多算法分析工作，包括推导出精确的$\mathcal{O}$ 运行时间和空间复杂度界。除了论文中详细介绍的算法外，我们还将系统应用于各种不同的算法。[15] 在为自然语言处理实现和调试结构化预测算法时，我们发现类型分析非常有价值。正如我们在§1中所提到的，我们的系统能够复现一些我们在自然语言处理文献中发现并已发表的有趣案例（Shi 等人，2017；Huang 和 Sagae，2010）。具体而言，我们能够利用运行时间界推断，复现 Shi 等人(2017) 对 Huang 和 Sagae(2010) 的分析所作的改进，以及 Kuhlmann 等人(2011) 对其精确弧标准序分析算法所作的时间与空间优化。我们认为，这表明需要借助工具来促进自然语言处理算法的分析与开发。

[15]这些程序包括多种依存句法分析变体（Huang 和 Sagae，2010；Kuhlmann 等人，2011；Eisner，1996；Johnson，2007；Eisner 和 Blatz，2007）、上下文无关句法分析（包括 Earley 算法（Earley，1970））、维特比解码（维特比，1967）、隐马尔可夫模型（Rabiner，1989）、半马尔可夫模型（Sarawagi 和 Cohen，2004）、编辑距离（Levenshtein，1966）、Bar-Hillel 算法（Bar-Hillel 等人，1961），以及有向图中最短路径的多种变体。

参考文献

Krzysztof R. Apt 和 Mark Wallace。2007 年。使用 *ECLiPSe* 的约束逻辑编程。剑桥大学出版社。

J. K. Baker. 1979年。 用于语音识别的可训练语法。载于 美国声学学会会议论文集：语音交流论文。

Yehoshua Bar-Hillel、Micha Asher Perles 和 Eli Shamir。1961年。 论简单短语结构语法的形式属性。 《语音学、语言学与传播研究杂志》，14。

John Cocke 和 Jacob T. Schwartz。1970年。编程语言及其编译器：初步笔记。技术报告，纽约大学库朗数学科学研究所。Alain Colmerauer 和 Philippe Roussel。1996。*Prolog* 的诞生，第 7 章。计算机协会。Patrick Cousot 和 Radhia Cousot。1977。抽象解释：通过构造或近似不动点，为程序静态分析建立统一的格模型。载于*ACM* 编程语言原理研讨会论文集。Greg Durrett 和 Dan Klein。2015年。神经条件随机场句法分析。载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。Jay Earley。1970年。一种高效的上下文无关句法分析算法。《*ACM* 通讯》，13(2)。Jason Eisner。2023。时间和空间高效的加权演绎。《计算语言学协会汇刊》（*TACL*），11:960–973。Jason Eisner 和 John Blatz。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。CSLI 出版社。Jason Eisner 和 Nathaniel W. Filardo。2011。Dyna：将 Datalog 扩展至现代人工智能。载于*Datalog* 重装上阵，第 6702 卷，收录于《计算机科学讲义》。施普林格。

Jason Eisner、Eric Goldlust 和 Noah A. Smith。2005。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。

Jason M. Eisner。1996年。 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。

Thom Frühwirth。1998年。 约束处理规则的理论与实践。 《逻辑编程杂志》，37(1)。

Carlos Gómez-Rodríguez、Jesús Vilares 和 Miguel A. Alonso。2009 年。 用于句法分析图式的编译器。 《软件：实践与经验》，39(5)。Joshua Goodman。1999年。 半环句法分析。 《计算语言学》，25(4)。Liang Huang。2009年。 自然语言处理中的基于动态规划的搜索算法。载于 人类语言技术会议论文集：计算语言学协会北美分会年会，会议伴随卷：教程摘要。Liang Huang 和 Kenji Sagae。2010年。 用于线性时间增量式句法分析的动态规划。载于计算语言学协会年会论文集。Frederick Jelinek。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。Mark Johnson。2007。 通过展开-折叠将投射性双词依存语法转换为可高效句法分析的上下文无关文法。载于 计算语言学协会年会论文集。Tadao Kasami。1965。 一种用于上下文无关语言的高效识别与句法分析算法。技术报告，空军剑桥研究实验室。Yoon Kim、Carl Denton、Luong Hoang 和 Alexander M. Rush。2017。 结构化注意力网络。载于 国际学习表征会议论文集。

Marco Kuhlmann、Carlos Gómez-Rodríguez 和 Giorgio Satta。2011。用于基于转移的依存句法分析器的动态规划算法。载于 计算语言学协会年会：人类语言技术论文集。

Kenton Lee、Mike Lewis 和 Luke Zettlemoyer。2016。具有最优性保证的全局神经组合范畴语法分析。载于自然语言处理实证方法会议论文集。

Vladimir I. Levenshtein。1966。可纠正删除、插入和逆转错误的二进制码。苏联物理学文摘，8。

Alberto Martelli 和 Ugo Montanari。1982。一种高效的合一算法。*ACM*编程语言与系统汇刊（*TOPLAS*），4(2)。David McAllester。2002。静态分析复杂度分析。*ACM*会刊，49(4)。Mehryar Mohri。1997。语言与语音处理中的有限状态转换器。《计算语言学》，23(2)。Leonardo Mendonça de Moura 和 Nikolaj Bjørner。2008。Z3：一种高效的SMT求解器。载于系统构建与分析工具和算法国际会议论文集。Fernando C.N. Pereira 和 Stuart M. Shieber。2002。*Prolog*与自然语言分析。Microtome Publishing。Benjamin C. Pierce。2002。类型与编程语言，第1版。麻省理工学院出版社。Peng Qi、Yuhao Zhang、Yuhui Zhang、Jason Bolton 和 Christopher D. Manning。2020。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。

Lawrence R. Rabiner。1989。隐马尔可夫模型及其在语音识别中的精选应用教程。《*IEEE*会刊》，77(2)。

Pushpendre Rastogi、Ryan Cotterell 和 Jason Eisner。2016。结合神经上下文对有限状态转换加权。载于计算语言学协会北美分会会议论文集：人类语言技术。

John Alan Robinson。1965 年。 一种基于归结原理的面向机器的逻辑。 *ACM*会刊，12(1)。

Alexander Rush。2020 年。 Torch-Struct：深度结构化预测库。载于计算语言学协会年会论文集。计算语言学协会。

Sunita Sarawagi 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。Tianze Shi、Liang Huang 和 Lillian Lee。2017 年。通过最小特征集实现基于转移的依存句法分析的快速（更快）精确解码和全局训练。载于自然语言处理经验方法会议论文集。Jeffrey D. Ullman。1988 年。 《数据库与知识库系统原理》第一卷。Computer Science Press。Tim Vieira。2023 年。 使用自然语言处理应用自动分析和改进动态规划算法。博士论文，约翰斯·霍普金斯大学。K. Vijay-Shanker 和 David J. Weir。1989 年。 组合范畴语法和线性索引语法的识别。载于国际解析技术研讨会。Andrew J. Viterbi。1967 年。 卷积码的误差界与渐近最优解码算法。 *IEEE*信息论汇刊，13(2)。Stephen Wolfram。2003 年。 《*Mathematica*之书》，第 5 版。Wolfram-Media。Daniel H. Younger。1967 年。 在时间 n^3 内识别和解析上下文无关语言。 信息与控制，10(2)。