arXiv:2305.04154v1 [cs.AI] 7 May 2023

Score：Scone 知识库系统的规则引擎

Jeffrey Chen

CMU-CS-21-116
2021年5月

计算机科学系 卡内基梅隆大学计算机科学学院 匹兹堡，宾夕法尼亚州 15213

论文委员会：Scott E. Fahlman，卡内基梅隆大学计算机科学系及语言技术研究所

Alessandro Oltramari，博世研究与技术中心，匹兹堡

部分满足理学硕士学位要求而提交。

版权所有 © 2021 Jeffrey Chen

关键词：　Scone、知识库系统、产生式规则系统、产生式系统

摘要

我们介绍了 Score，这是一个为 Scone 知识库系统设计并实现的规则引擎。Scone 是一个知识库系统，旨在以符号形式存储和处理丰富的通用知识表示。它以网络结构中的节点和链接来表示知识，并能高效地对不同元素之间的关系进行基本推理。Scone 本身可视作一种能够与其他软件系统交互的“智能记忆”。Scone 有一个可以改进的方面：如何更有效地向智能体提供知识，使其能够利用这些知识执行动作，并根据观察结果更新知识库。

我们为 Scone 系统增添了一个产生式规则引擎，它会基于 Scone 知识库中已有和新添加的结构自动执行简单推理，从而有望提升构建于 Scone 之上的任何规划系统的能力。产生式规则系统由“如果—那么”产生式规则组成，这些规则会尝试将其谓词与现有知识相匹配，并在谓词满足时触发相应动作。我们提出了两种产生式规则：添加时规则和需要时规则，它们在检查和触发方式上有所不同，以涵盖多种使用场景。随后，我们实现了高效检查和触发这些规则的方法，以用于大型知识库。这个新规则引擎并非旨在成为一个复杂的独立规划器，因此我们讨论了它如何融入 Scone 的背景，以及规划系统的未来研究工作。

致谢

我要感谢 Scott Fahlman 教授，他向我介绍了 Scone，并指导我如何设计 Scone 中的规则引擎。他对 Scone 整体架构以及产生式系统相关工作的见解，对我撰写这篇论文起到了至关重要的作用。我还要感谢 Alessandro Oltramari，感谢他愿意抽出时间加入论文委员会，并为本文提出修改意见和改进建议。最后，我要感谢我的姐姐、父母和朋友，在我需要的时候，他们总是给予我支持。

目录

1 引言与背景 **1**
1.1 Scone.2
1.1.1 元素.2 1.1.2 多重继承与标记传递.3 1.1.3 角色与虚拟复制语义.4 1.1.4 多重上下文与情景推理.5

1.2 论文结构.5

2 相关工作 7
2.1 基于规则的系统.7 2.1.1 Soar.7 2.1.2 ACT-R.8 2.1.3 其他相关工作.9

3 Scone 规划系统 **11**
3.1 产生式规则设计.11 3.1.1 添加时规则和需要时规则.12 3.2 规则结构.13 3.2.1 添加时规则示例.14 3.2.2 需要时规则示例.15 3.3 基于配方的规划器.16 3.4 实现状态.17

4 规则检查引擎 19
4.1 规则检查组件.19 4.1.1 规则触发器.19 4.1.2 规则变量替换.20 4.1.3 规则检查搜索算法.22 4.2 端到端示例规则触发.23 4.2.1 若需要示例.25

5 结论与未来工作 **27**

参考文献 **29**

附录 **31**

图目录

1.1 Scone 元素的可视化.2 1.2 知识“Fido 是 John 的宠物”中角色节点的可视化。此处，{pet of John} 是 {pet} 的虚拟副本，而 {pet} 继承自 {John}。 . . . . . 4 3.1 若添加飞行事件规则的示意图，展示了其中一些相关元素。A 和 B 以虚线圆圈绘制，表示它们是其他元素的占位符。实线表示 is-a 链接，虚线表示 x-y-z 链接（此处为角色 has-link）。双线表示触发规则动作时创建的链接.14 3.2 按需会议时长规则的示意图。从 A 和 B 指向标记为 $f(A, B)$ 的节点的箭头表示 A 和 B 将用于计算；从 $f(A, B)$ 指向“C 的持续时间”的双线表示将创建并返回的 is-a 链接.16 4.1 is-a 类型约束触发器的可视化。Y 在 is-a 层次结构中位于 B 之上，并附有一个指向规则 R 和变量 X 的触发器。通过将 A 与 X 匹配来执行触发器激活.21 4.2 x-y-z 谓词触发器的可视化。Y 在 is-a 层次结构中位于 B 之上，并附有一个指向规则 R 以及变量 X 和 Z 的触发器。通过将 A 与 X、C 与 Z 匹配来执行触发器激活.21 4.3 if-needed 规则触发器的可视化。Y 在 is-a 层次结构中位于 B 之上，并附有一个指向规则 R 和变量 Z 的触发器。通过将 C 与 Z 匹配来执行触发器激活。作为规则动作的一部分，将在计算结果 X 与“C 的 B”之间创建一条 is-a 链接。 . . . . . 21 4.4 规则检查算法.22 4.5 会议示例规则的两次触发器激活的可视化，分别对应两个 x-y-z 角色谓词.24

第 1 章

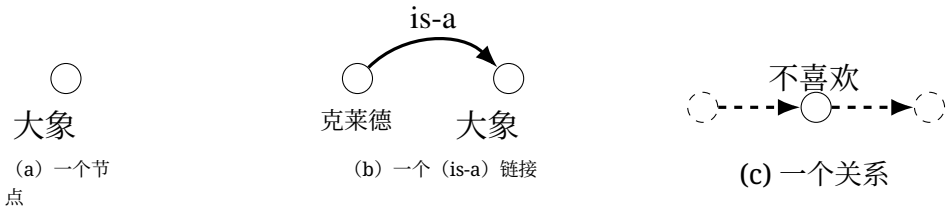
引言与背景

Scone [6, 8] 是一个开源知识库系统（KBS），旨在存储大量知识，包括通用常识知识和特定领域的知识。通过采用多重继承和虚拟复制语义，Scone 的知识库包含大量隐式知识，远超知识网络中明确定义的内容。Scone 还具备快速推理能力，通过并行标记传递算法实现，能够回答有关所存储知识的基本查询 [6]。Scone 使用 Common Lisp 实现，自约 2003 年起一直在卡内基梅隆大学（CMU）语言技术研究所开发。一本介绍 Scone 设计与使用的教程书目前正在编写中。在该书出版之前，更多有关 Scone 的信息可参见 Fahlman 的 Knowledge Nuggets 博客 [8]。

Scone 本身并不是能够做出决策的智能体，而更像是一个“智能记忆”，可用于为其他决策过程提供信息。我们的长期目标之一，是赋予 Scone 对计划和动作进行可靠推理的能力，以便用于引导智能体。Scone 的 ERIS（Scone 中的情景推理）子系统为在 Scone 中表示情景知识（动作、事件、序列和计划）提供了一个起点，使得对其进行基本推理成为可能 [7]。ERIS 为 Scone 引入了一种事件类型，用于表示关于状态如何变化的知识，并配备了其他机制来表示事件发生前后、事件因果关系等概念。

产生式系统由一组产生式规则和规则可以访问的工作记忆组成。每条产生式规则都有一个左侧前提条件，尝试与工作记忆匹配；还有一个右侧动作，用于修改记忆或执行其他动作。产生式系统已被用作 Soar [12, 13, 14] 和 ACT-R [1, 2]等认知架构中的规划系统。不过，这些系统用产生式系统处理各种规划任务，而我们的规划则是让不同系统分别处理简单、低层次的推理和高层次的规划。

我们的主要贡献是 Score，即 Scone Rule Engine（Scone 规则引擎）的简称，它使用产生式规则进行自动规划和推理。这个基于规则的规划器将负责执行检查和触发操作，以处理 Scone 的其他系统尚未处理的底层推理任务。我们还概述了一种基于配方的规划器，但并未实现它。该规划器通过将目标分解为子目标并考虑不同的备选计划，来处理更复杂、更高层次的规划。我们这样做是为了明确规则引擎的目标仅仅是处理简单的自动



（a）一个节点　（b）一个（is-a）链接　(c) 一个关系

图 1.1：Scone 元素的可视化

推理，而非像某些其他产生式系统那样执行各种复杂的规划。

举例来说，为了说明我们希望规则引擎具备哪些能力，假设我们使用 Score 来规划一次旅行，而旅行可以表示为一种事件类型。如果我们知道旅行使用的交通工具是飞机，那么这次旅行就涉及飞行；而如果使用的是汽车，则涉及驾驶。这些推理可以非正式地表述为规则：“如果存在一次旅行事件，且使用的交通工具是飞机，那么这次旅行就是一次飞行事件”，另一个推理也可以类似地表述。我们的规则引擎支持定义这些规则，并执行这些推理，从而自动更新知识库。

在概述我们的贡献细节之前，我们先介绍 Scone 本身的结构所需的背景知识。

1.1 Scone

1.1.1 元素

Scone 中知识的基本单位是元素，它以 Common Lisp 数据结构的形式表示。Scone 元素用花括号表示。元素分为三类：节点、链接和关系，如图 1.1 所示。节点表示关于世界的某种概念性知识的描述，例如 {elephant}（大象）或 {the mother of Clyde}（Clyde 的母亲）。链接描述不同元素之间的关系，例如 {Clyde is an elephant}（Clyde 是一头大象）或 {I dislike brussels sprouts}（我不喜欢抱子甘蓝）。链接有许多不同类型，分别描述其连接的元素。关系表示某种模板关系，可实例化为称为陈述链接的特殊链接。例如，{dislikes}（不喜欢）是一种关系，而其实例 {I dislike brussels sprouts}（我不喜欢抱子甘蓝）则是一个陈述链接。

Scone 中一种特定类型的节点是角色节点。角色节点附属于另一个称为所有者节点的节点，并表示所有者节点可能拥有或具有的某种概念。例如，一个人可能有母亲，一道菜可能有食材，因此在 Scone 中，我们可以将 {mother}（母亲）设为所有者为 {person}（人）的角色节点，并将 {ingredient}（食材）设为所有者为 {dish}（菜肴）的角色节点。

节点分为类型节点和个体节点。类型节点表示某个集合中的典型成员，例如，{大象} 表示典型的大象。个体节点表示某个集合中的特定成员，例如，{大象克莱德} 表示名叫克莱德的一头特定大象。这些概念也适用于角色。类型角色表示所有者可能拥有多个的典型角色，例如一道菜的 {配料}。个体角色表示所有者通常只拥有一个的特定角色，例如某个人的 {母亲} 。

交类型是一种类型节点，表示若干不同类型的交集。例如，类型 {男人} 可以看作类型 {男性}、 {人类}和{成年人}的交集。定义类型是一种交类型，此外还带有一个谓词，节点必须满足该谓词才能成为该类型的实例。稍后我们会进一步探讨，交类型可以看作简单的如果—那么规则：如果一个元素是 {男性}、 {人类}和 {成年人}，那么该元素就是 {男人}。

每个链接都有一组连线，用于控制链接如何将不同元素连接在一起。两种重要的连线是链接的 A 连线和 B 连线。A 连线连接到链接中引用的第一个元素，B 连线连接到链接中引用的第二个元素。例如，在表示 {Clyde is an elephant} 的链接中，A 连线连接到 {Clyde}，B 连线连接到 {elephant}。

最后这个例子是一种称为 is-a 链接的特殊链接。这类链接表示连接到 A 连线的元素是连接到 B 连线的元素的一个更具体的概念。例如，连接 {human} 和 {mammal} 的 is-a 链接表示这样一种知识：人是哺乳动物的一种特定类型。此外还有 eq 链接，表示连接到 A 连线和 B 连线的元素彼此等价。所有 is-a 链接和 eq 链接共同构成所谓的 is-a 层次结构。在 is-a 层次结构中“向上”是指沿 is-a 链接从 A 连线上的元素到 B 连线上的元素，从而抵达更一般的概念；“向下”则是指从 B 连线上的元素到 A 连线上的元素，从而抵达更具体的概念。在 is-a 层次结构中，某个元素上方的元素称为它的上位元素，下方的元素称为它的下位元素。作为类型节点的上位元素称为超类型，作为类型节点的下位元素称为子类型。（所有上位元素实际上都是超类型，因为个体节点不可能有任何下位元素。）is-a 层次结构中最顶层的元素是一种称为 {thing} 的类型：其他所有概念都比 {thing} 更具体。

1.1.2 多重继承与 MarkerPassing

Scone 通过其 is-a 层次结构支持多重继承：每个元素都可以拥有任意数量的传入和传出 is-a 链接。继承使 Scone 能够推断出知识库中许多未被明确陈述的事实。例如，如果知识库包含“鸟类有羽毛”以及“鸡是一种鸟类”的知识，那么“鸡有羽毛”这一事实就会从鸟类类型中继承而来，并可以被推断出。

Scone 还具有取消链接，以支持带有例外的推理。例如，我们可以在知识库中记录：{鸟类} 是 {会飞的事物}的子类型，因为鸟类通常能够飞行；同时， {企鹅} 是 {鸟类}的子类型。然而，由于企鹅不会飞，我们从 {企鹅} 到 {会飞的事物}添加一个取消链接，以表明我们不希望将 {会飞的事物} 从{鸟类}继承而来。取消链接是一个复杂的主题，有时可能导致知识歧义；在规则引擎中，我们不会对这些问题进行深入处理。

Scone 使用标记传递系统来执行高效推理，并处理继承和虚拟复制 [6]。Scone 中的每个节点都配备了一组固定的标记位，可以将其打开或关闭。这些标记可以有条件地并行打开或关闭。例如，如果一组节点被标记了某个标记 m，我们就可以并行查看所有 is-a 链接，并请求每个链接：如果连接到 A 连线的节点带有标记 m，则为连接到 B 连线的节点标记 m。此操作会获取所有带有 m 标记的节点，并为所有

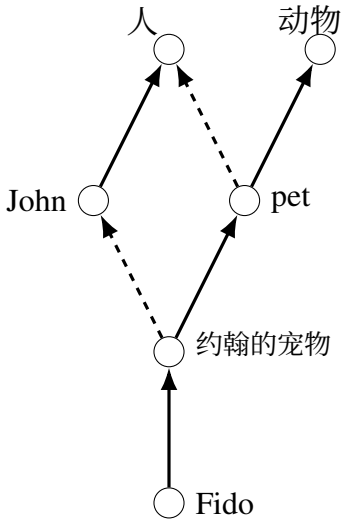


图 1.2：知识“菲多是约翰的宠物”中的角色节点示意图。这里，{约翰的宠物}是 {宠物}的虚拟副本，而 {约翰} 从 {人}继承了它。

位于其上方一层 is-a 层次结构中的节点标记 。重复此操作，直到没有新节点被标记为 m，我们就能快速找到特定节点在 is-a 层次结构中的所有上位节点，这一过程称为向上扫描。与之对应的过程称为向下扫描，即标记特定节点在 is-a 层次结构中的所有下位节点。

1.1.3 角色与 VirtualCopy 语义

角色节点会产生称为 has 链接的特殊链接。为所有者声明新角色时，也会创建一条新的 has 链接，其中 A 连线连接到角色节点，B 连线连接到所有者节点。这条链接表示所有者类型拥有角色类型的某个元素。角色的任何下位元素都称为参与者或填充项，表示“参与者是所有者的角色填充项”。

Scone 通过虚拟复制语义，以一致的方式处理与角色相关的复杂知识 [6]。创建具有多个角色的类型的下位元素时，这些角色会通过继承进行虚拟复制。这意味着，新下位元素会被视为拥有其父类型的每个角色的副本，尽管实际上并未进行复制。例如，假设定义了一个 {宠物} 角色，其所有者是 {人}。这会创建一个 has 链接 {人拥有宠物}。个体 {John} 随后会继承该 has 链接和 {宠物} 角色，从而创建一个 {John 的宠物}虚拟副本。要表示个体 {Fido} 是 John 的宠物，只需添加一条从 {Fido} 到 {John 的宠物}的 is-a 链接即可。图 1.2 展示了这一关系，实线箭头表示 is-a 链接，虚线箭头表示 has 链接。

Scone 还包含一些与角色节点和关系相关的标记操作。对于角色节点，有一个函数会标记每个满足 x 是指定角色和所有者的角色的节点 x，还有一个函数会标记每个满足 PLAYER 是指定角色和 PLAYER 的 x 的角色的节点 x。对于关系，有一些函数会标记每个节点 x，使得当 RELATION 和】【。

已指定 A 或 B 之一。

借助这些标记操作，我们可以高效地回答有关知识库的各种问题，例如 Clyde 是否是动物，或者 Clyde 是否有兄弟姐妹。这些标记传递操作将构成检查 Scone 中的知识是否满足任何可能触发的产生式规则的基础。虚拟继承也将在规则“触发器”的激活中发挥作用。

1.1.4 多重上下文与情景推理

Scone 知识库中的所有知识都存在于某个上下文中，而上下文可以是任意 Scone 元素。默认情况下，所有知识都存在于一个名为 {general} 的大型上下文中。Scone 实现了一种多重上下文机制，可以处理存在于不同上下文中的不同知识，并在上下文之间切换。Scone 中的每个元素都有一条上下文连线，连接到某个上下文，以指明该元素所表示的知识何时存在或有效。

要激活某个上下文，需要在该上下文元素上放置一个特殊的上下文标记，然后向上扫描到该上下文的所有超类型。这样，Scone 就能轻松地考虑不同的“宇宙”；这些宇宙大体上与我们的宇宙相同，但存在一些差异。例如，假设我们想要描述有关哈利·波特宇宙的知识。我们可以创建一个 {哈利·波特宇宙}上下文，它是 {general} 上下文的子类型。激活 {哈利·波特宇宙}上下文时，我们会继承 {general} 中的所有知识，因此“人类有两只手”之类的许多基本事实在该上下文中仍然成立。不过，我们可以只向该上下文添加节点和链接，例如 {巫师可以使用魔法}，这些内容只有在 {哈利·波特宇宙}上下文处于活动状态时才成立。重新激活 {general} 上下文时，新节点和链接的上下文连线指向 {哈利·波特宇宙}，而该上下文已不再带有活动上下文标记，因此这些节点和链接处于非活动状态。Scone 的多重上下文机制使我们能够轻松探索包含略有不同知识的各种可能状态。

Scone 中的情景推理 [7] 在很大程度上依赖多上下文机制。Scone 有一种 {事件}类型，用来表示具有“之前”状态和“之后”状态的事件。我们通过为 {事件}赋予两个角色 {前置上下文} 和 {后置上下文}来表示这一点。顾名思义，{前置上下文} 包含事件发生前为真的知识，而相应地，{后置上下文} 包含事件发生后为真的知识。通常，{前置上下文} 继承自活动上下文，这意味着当前所有活动知识在事件发生前都有效；而 {后置上下文} 继承自 {前置上下文}，并做出少许修改。举例来说，假设有一个事件表示毛毛虫经历蜕变成为蝴蝶。在前置上下文中，某个个体与 {毛毛虫}之间会有一个 is-a 链接；在后置上下文中，我们会取消此链接，并在该个体与 {蝴蝶}之间建立一个新的 is-a 链接。Scone 还有一种 {动作} 类型，它是一种特殊的 {事件}，具有一个 {施事者} 角色，表示引发该事件的事物。施事者可以是某个事件中的一个 {人}，例如投掷球的事件。

1.2 论文结构

论文的其余部分安排如下。第2章讨论产生式系统在认知架构中的应用相关工作。第3章设计一种结构

用于兼容 Scone 的产生式规则，并且我们还介绍了基于配方的规划器，以便为我们的设计目标提供背景。第4章介绍了规则触发器和规则搜索算法，这些机制用于结合知识库，高效地检查并触发这些规则。最后，我们在第5章总结了未来的研究方向。附录中提供了一些具体的代码，以及指向规则引擎代码库的指针。

第2章

相关工作

2.1 基于规则的系统

产生式规则系统已应用于 Soar 和 ACT-R 等现有认知架构。一般而言，产生式系统由一组规则（或产生式）以及一种用于检查和触发规则的机制组成；这些规则（或产生式）可以看作如果—那么语句。产生式系统假定存在某种形式的记忆或知识，产生式可以读取和修改它。产生式的“如果”部分定义了一个条件，该条件必须与记忆的当前状态相匹配。如果匹配成功，该产生式的“那么”部分就会触发并执行某个动作，而该动作通常包括修改当前记忆或知识。产生式触发后，记忆可能会被修改为与另一条规则的“如果”部分相匹配的状态，从而引发一连串的产生式规则触发。

这种通用结构将系统的许多部分留给底层架构来设计和实现，例如如何存储和匹配记忆，以及当多条产生式规则与记忆匹配且均可触发时该如何处理。Soar 和 ACT-R 的产生式系统都采用这种相同的通用结构，但它们对这类设计问题的处理方式不同，并且各自强调系统中略有不同的部分。

2.1.1 Soar

Soar 最初由 Allen Newell 在卡内基梅隆大学计算机科学系的研究小组开发，此后 Soar 的研究主要由密歇根大学的 John Laird 和南加州大学的 Paul Rosenbloom 持续推进 [12, 13, 14]。Soar 旨在成为一种统一认知架构，即一种能够解释认知如何在各种任务中运作的理论。其关键设计目标包括以目标为导向，也就是说，为了实现既定目标而做出理性决策；还包括运用大量知识和抽象，根据当前感知到的状态来指导决策。

Soar 区分两种记忆：长期记忆（长时记忆，LTM）和工作记忆（WM）。长期记忆进一步分为程序性知识、语义知识和情景知识，不过程序性长时记忆是规则系统中主要使用的知识类型。

程序性知识存储了关于如何以及何时执行任务以达成目标的信息，并以 if-then 产生式规则的形式表述。工作记忆包含有关当前状态的信息，例如目标以及任何与环境相关的信息。Soar 还区分规则和算子：规则可以扩充工作记忆状态，并对选择哪个算子提出建议，而只有算子可以修改工作记忆中存储的知识。

长时记忆和工作记忆通过以下五个阶段的序列协同工作，以实现期望的目标：输入阶段、扩展阶段、决策阶段、应用阶段和输出阶段。输入阶段是定义目标，并将关于起始状态的知识从知觉模块引入工作记忆的阶段。扩展阶段是程序性长时记忆中的产生式规则触发的阶段。由于这些规则动作是单调的，这意味着它们只会向当前工作记忆中添加新信息以及关于可以采取哪些可能操作符的建议，因此当前状态下所有适用的规则都会并行触发。当不再有规则可以触发时，扩展阶段结束。

扩展阶段结束、决策阶段开始后，系统会考量所有建议的算子，并选择一个来应用。系统会依据某些偏好规范做出决策，例如优先选择某个算子，或为每个算子分配一个偏好分数并选择分数最高的算子。选择算子后，系统会在应用阶段应用该算子，以修改工作状态。输出阶段只需将算子发送到输出模块，例如控制运动输出的模块。

2.1.2 ACT-R

ACT-R 是 Adaptive Control of Thought-Rational（自适应思想控制-理性）的缩写，由CMU心理学系的 John Anderson 研究团队 [1, 2]主要开发。它是另一种认知架构，旨在模拟和解释人类认知的核心心理过程。ACT-R 的关键假设是，知识分为两种：陈述性知识和程序性知识。陈述性知识以“块”的形式表示，这种结构包含一个指向事实类型的 isa指针，以及指向事实内容的其他指针。程序性知识则以如果-那么语句形式的产生式规则表示。

在 ACT-R 中，许多不同的模块负责处理不同类型的信息，例如处理视觉信息的视觉模块，以及用于检索陈述性知识的陈述性模块。每个模块内部都可以并行处理，处理结果会写入一个小型缓冲区。向缓冲区写入数据和从缓冲区读取数据都是串行进行的，这会在不同模块之间形成通信瓶颈。各模块通过中央程序模块连接，由它协调来自不同缓冲区的数据，并使用产生式规则对其进行处理。程序模块旨在模拟与大脑基底神经节相关的认知机制；据推测，基底神经节会接收来自大脑不同区域的信息，并对其进行处理，从而做出整体决策。

程序模块中的产生式系统经过设计，每次只能触发一条产生式规则，从而形成中央瓶颈，使信息必须在程序层面进行串行处理。设置这一瓶颈的原因之一，是防止多条规则同时触发，导致知识发生相互矛盾的变化；另一个原因是，有认知心理学研究证据表明，人脑在处理

问题状态 [3] 时存在中央瓶颈。当多条规则均适用时，系统会根据某个效用函数，选择效用最高的规则。程序模块通过读取信息并应用产生式规则来完成任务，直至达到目标。研究人员通过实验表明，经过适当设计的 ACT-R 模块能够复现人类在各种任务中的认知处理过程。

此外，也有一些研究探讨了将 Scone 与 ACT-R [15]集成。在这项研究中，Oltramari 和 Lebiere 将 Scone 作为知识模块加入 ACT-R，创建了 ACT-RK（其中 K 代表 Knowledge，即知识）。统一后的 ACT-RK 提升了基于知识的推理能力；将其应用于以视频形式对视觉输入进行语义描述的任务时，便展示了这一点。

2.1.3 其他相关工作

一些规则系统（例如基于 Prolog 的规则引擎） [4], 使用反向链接，而不是正向链接。在 Prolog 系统中，程序由事实和规则构成，规则说明：如果某些前提为真，那么某个结论就为真。程序执行由查询驱动，例如 ?- animal(X)., 用于询问哪些事物是动物。此类查询会在规则之间进行回溯，以找到查询的解答；在本例中，系统会输出所有依据规则可推断为动物的事物。

在知识量庞大且规则众多的系统中，朴素的规则模式匹配往往速度过慢。一种优化方法是 Rete 算法 [10] ，它可以提升匹配性能。Rete 算法会构建一个由节点组成的网络，其中包含用于检测是否有知识部分满足规则谓词的测试。从根节点到叶节点的一条路径表示一个完整的规则谓词已得到满足。添加新事实时，这些事实会沿网络向下传播；任何被触达的叶节点都表明相应动作已准备好触发。

Scone 系统管理并存储语义知识，但它并非唯一的语义知识网络。语义网 [11] 是万维网的扩展，能够对网页中的数据进行语义分析。它由多个层组成，包括 RDF（资源描述框架）和 OWL（网络本体语言）。RDF 使用主语－谓词－宾语三元组来描述数据。一组三元组构成了一个关于网络资源的信息网络。OWL 基于形式逻辑，由用于描述不同术语之间允许存在何种关系的属性断言组成。计算机程序可以利用 OWL 的结构来验证逻辑一致性，并通过演绎将隐含知识显化。

第3章

Scone 规划系统

长期目标是将两个规划系统集成到 Scone 中：一个“基于规则的”规划器和一个“基于配方的”规划器。基于规则的规划器旨在使用产生式执行快速的自动思考，而基于配方的规划器则旨在通过更缓慢、更审慎地考虑如何执行“配方计划”中的步骤来进行规划。这两种规划器都很有价值，并且能够相互补充：基于规则的规划器可以回答大量简单查询，并自动扩充其知识；基于配方的规划器则可以利用这些知识，创建更复杂的计划来实现其目标。

在本章中，我们的主要贡献是为产生式规则设计了一种能够与 Scone 的知识结构恰当衔接的方案。我们已在 Common Lisp 代码中实现了这一设计，使用户能够定义可添加到知识库中的新规则。基于配方的规划器的实现超出了本研究的范围，将留待未来工作；我们描述了它应有的行为，以说明基于规则的规划器并非为执行复杂规划而设计。

Scone 存储了大量知识的丰富表示，并使构建于其上的其他应用能够轻松查询这些知识。从这个意义上说，Scone 与 Soar 和 ACT-R 中的记忆模块有许多共同之处。具体而言，Scone架构及其元素结构可视为类似于 ACT-R 中的陈述性知识，而我们新建的产生式系统则对应于 ACT-R 中的程序性知识。Scone 也可以与 Soar 中的工作记忆相比较，不过其范围要大得多，因为工作记忆通常相当有限；而新的产生式系统则发挥着类似于 Soar 的程序性长期记忆的作用。

Scone 中的基于规则的规划器与 Soar 和 ACT-R 中的产生式系统之间有一个重要区别：Soar 和 ACT-R 依赖产生式规则来执行带有目标的复杂规划。我们预计由基于配方的规划器处理基于目标的规划，而不是由基于规则的规划器处理。相应地，我们的产生式并非旨在朝着特定的目标状态开展工作，而是用于表示快速、反射式的思维。

3.1 产生式规则设计

基于规则的规划器依靠产生式规则进行快速、反射式思考。为此，我们在 Scone知识库系统中引入了一种称为规则的新型知识

（production rule 的简称）。Scone 规则与普通产生式规则类似，都有一个“如果”前提条件，也称为规则的左侧（LHS），以及一个“那么”动作，也称为右侧（RHS）。不过，与之对应的“工作记忆”是整个 Scone 知识库，这与其他产生式系统的工作记忆截然不同，因为它的规模大得多，因此产生式规则必须经过妥善设计，才能与 Scone 及其各种机制良好配合。

规则的“if”部分定义了一些谓词，这些谓词必须与知识库的某一部分匹配，才能满足该规则。Scone 中的匹配比简单的模式匹配稍微复杂一些，因为在匹配时还应考虑 is-a 层次结构中的下位元素。由于 Scone 知识库通常非常庞大，我们希望单条规则只考虑知识库中一个局部的部分。规则最简单的形式只考虑单个元素，并规定关于该元素的一些谓词。例如，一条可能的规则是：如果某个元素既是 {male}，又是 {adult}，那么该元素就是 {man}。这些只依赖于单个元素的规则实际上是 Scone 中已有的定义类型或交类型的示例，因此可以将定义类型和交类型视为规则的特例。

对于在 LHS 中考虑多个元素的规则，这些元素应当通过角色和/或关系以某种方式彼此连接。例如，一条规则可能依赖于某个会议及该会议的开始时间，这对应于两个元素，其中一个是另一个的角色。从概念上讲，一条规则应当描述一组局部元素的某些情况，因此，依赖于多个元素、但这些元素之间又没有通过角色或关系直接连接的规则没有太大意义。

规则的动作定义了当某组元素满足规则的左侧（LHS）时要执行什么操作。动作可以是为规则中的某些元素添加链接。对知识库所做的这些修改会归入活动上下文。如果 Scone 连接了身体运动模块，动作也可以触发外部行为，例如身体运动。动作的结果可能使知识库进入满足其他规则的状态，从而引发规则的前向链式推理，使各条规则依次触发。

3.1.1 添加时规则与需要时规则

我们将 Scone 规则分为两类：添加时规则和需要时规则。添加时规则也称为急切规则，会在向知识库添加新链接时触发。需要时规则也称为惰性规则，当请求知识库中的某个值但未找到该值时触发；它们会计算出所请求的值，并将其作为动作保存到知识库中。

我们区分这两种规则，是为了让它们能够发挥略有不同的作用。如果某条规则的动作应在其左侧条件（LHS）满足后立即执行，那么它就应该是 if-added（急切）规则；无论是因为它可能触发其他规则，还是因为该动作十分紧急、需要立即处理。if-added 规则与现有产生式系统中的产生式规则更为相似。它们会检查记忆——在 Scone 中，记忆就是整个知识库——是否符合其谓词；如果符合，就会触发相应的动作。这些动作随后可以改变记忆，从而使其他规则符合条件并触发。我们只在添加新知识时检查 if-added 规则（因此称为“if-added”规则），因为只要知识保持不变，满足条件的规则集合也就不会改变。只有

添加新知识才会触发规则。发生这种情况时，系统会急切地触发规则，直到没有更多规则可以触发为止，然后暂停，直至将来添加更多知识。采用急切规则的代价是，添加新知识所需的时间会比以前稍长，但从知识库中检索知识仍然很快，不会产生额外开销。

与添加时规则不同，如果某个知识槽为空，且可以通过计算得出其内容，但计算的优先级不高，那么规则就应当是需要时触发（惰性）规则。这些规则会将规则检查推迟到某项知识被明确请求却未找到时。此时，系统会检查知识库中是否存在可与这些谓词匹配的适用知识；如果存在，就用它来计算所请求的知识。执行完成计算的规则动作后，规则引擎会将新知识添加到知识库中，以便日后再次请求相同知识时重用，避免再次触发规则。

3.2 规则结构

现在，我们将详细说明 Scone 中规则的结构，并提供一些示例。规则由若干部分组成：

- 规则中的变量用于创建占位符，以便将 Scone 元素代入规则。这种抽象使规则能够应用于任何有效元素，而不仅限于某些特定个体。对于每个变量，都可以指定一个 Scone 类型约束，要求元素必须是指定类型的下位元素才能满足规则。每个变量最多只能指定一个类型约束，不过，指定交类型实际上可以允许任意多个类型约束。

  某些规则变量还可以选择标记为恰当，这表示只有正规的 Scone 节点才能替代该变量（即不能使用通用角色节点）。稍后我们会看到这样做的必要性。

- 规则中的 x-y-z 谓词定义了规则中不同元素之间的关系。每个谓词由三个元素组成，形式为 (X Y Z)。这里的 Y 元素必须是角色或关系，而 X 和 Z 元素则是变量或 Scone 元素。如果 Y 是角色，该谓词表示“X 是 Z 的某个 Y”。如果 Y 是关系，该谓词表示“陈述 X Y Z 为真”。要满足规则，每个谓词都必须为真。
- 当规则被触发，且知识库中的某些元素满足规则的谓词时，就会执行该规则的动作。添加时规则和需要时规则的可用动作略有不同。
  - 对于添加时规则，动作可以是任意代码，并使用规则中定义的变量。规则会将满足条件的元素代入这些变量，并执行代码，从而改变知识库或在现实世界中执行某种动作。对于需要时规则，动作的形式为 (X Y Z)，其中 X 是任意计算，Y 是个体角色节点，Z 是表示所需值所属者的变量。执行动作时，会进行计算 X，并将其结果设为所需的槽填充值“Z 的 Y”。

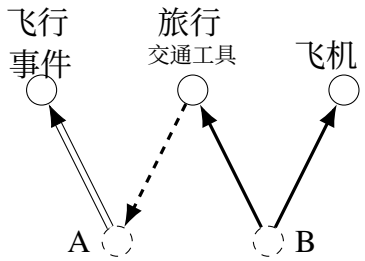


图 3.1：若添加飞行事件规则及其中一些相关元素的示意图。A 和 B 用虚线圆圈绘出，表示它们是其他元素的占位符。实线表示 is-a 链接，虚线表示 x-y-z 链接（此处为角色 has-link）。双线表示触发规则动作时创建的链接。

变量及其类型约束与 x-y-z 谓词共同构成规则的 LHS 谓词，而动作则是规则的 RHS。此外，我们还对规则的 x-y-z 谓词施加约束，以确保规则中的元素彼此相连。若将 X 和 Z 的值视为顶点，并针对每个 x-y-z 谓词在 X 与 Z 之间连一条边，那么这张图必须是连通图，规则才能定义良好。例如，某条规则包含四个变量 A、B、C 和 D，但只有“ A 是 B 的母亲”和“C 是 D 的母亲”这两个 x-y-z 谓词；这条规则定义不当，因为变量 A、B 与变量 C、D 并不相连。如果允许这样的规则，那么每次创建一条母亲链接时，这条新链接与任何一条已有的母亲链接组合起来都会满足该规则，导致规则触发过多。这类规则在概念上也不太合理，因为规则中的元素应当以某种方式彼此关联。

我们通过自然语言描述以下示例规则，以说明这些规则在概念上的含义。定义这些规则的 Common Lisp 代码见附录。

3.2.1 如果添加规则示例

考虑一个 if-added 规则示例：如果某人正在旅行，且其乘坐的交通工具是飞机，那么此人就在飞行。我们将 {旅行} 表示为一种带有 {旅行交通工具}角色的事件类型。此示例规则的各个部分如下：

- 有两个变量 A 和 B，分别代表旅行事件和交通工具，变量 B 的类型约束为“B 是飞机”。
- 有一个 x-y-z 谓词“B 是 A 的旅行交通工具”，用于描述 A 和 B 之间的关系。
- 该动作将链接“A 是一种飞行事件”添加到知识库。

知识库中与此规则匹配的部分如图 3.1 所示。一旦定义并将此 if-added 规则放入系统，规则引擎就会开始监听知识库中的新增内容。当它检测到可能将某些元素与谓词中的 A 和 B 匹配的新知识时，就会开始尝试检查该规则，并在谓词得到满足时执行动作。

规则中需要注意的一点是，我们想为变量 B 指定类型约束，但不为变量 A 指定。原因是，只有当变量的类型对于规则谓词的真值至关重要时，变量才应包含类型约束。规则的 x-y-z 谓词会对变量施加一些隐式类型约束。对于示例规则，可以从 x-y-z 谓词推断出 A 是一个 {旅行}事件，而 B 是一种 {交通工具} 。要确定 A 是否为 {飞行} 事件，必须知道 B 的候选元素是一架飞机，因为如果 B 是汽车等其他交通工具，A 就会是另一种旅行事件。另一方面，无需为 A 指定任何显式类型约束，因为隐式类型约束已足以确定 A 是否为飞行事件。

补充说明一下，即使指定了多余的类型约束，规则引擎仍然可以正常工作，只是可能需要执行更多操作。这是因为每个类型约束都会创建一个触发器，并且在添加新知识时可能会检查这些触发器。有关规则触发器的详细信息，请参见第 4.1.1 节。

3.2.2 按需规则示例

现在考虑一个 if-needed 规则示例：如果我们想知道一场会议的持续时间，并且知道该会议的开始时间和结束时间分别是 t_1 和 t_2，那么就可以将持续时间计算为 $t_2 - t_1$。该规则的组成部分如下：

- 有三个变量 A、B 和 C，分别表示开始时间、结束时间和会议。变量 A 和 B 应标记为恰当，以表明在规则中只能为它们替换恰当的值。
- 有两个 x-y-z 谓词：“A 是 C 的开始时间”和“B 是 C 的结束时间。”
- 该动作使用 A 和 B 的恰当值计算 B–A，并将结果作为“C 的持续时间”返回。

规则的各个部分如 3.2 所示。定义此 if-needed 规则后，系统会将其记作一种可能的计算方式，以便在有人请求时计算会议的持续时间。如果有人请求某场会议的持续时间，系统会先检查是否存在该会议的值；如果存在，则返回该值。如果不存在，规则系统就会尝试将有关该会议的知识与规则相匹配；如果匹配成功，则计算所请求的持续时间，并将其添加到知识库中。

在此规则中，应将变量 A 和 B 标记为恰当，是为了确保只代入具有实际值的节点，因为规则动作需要 A 和 B 的值。可以创建一个角色节点并为其指定类型，但不指定具体值；例如，可以说明某次会议的开始时间以 :30 结束，但确切时间未知。这样，该会议的开始时间就成为一个具体元素，是 {time ending in :30} 的子类型；但如果尝试计算该会议的持续时间，就会失败，因为该会议的开始时间没有恰当的值。

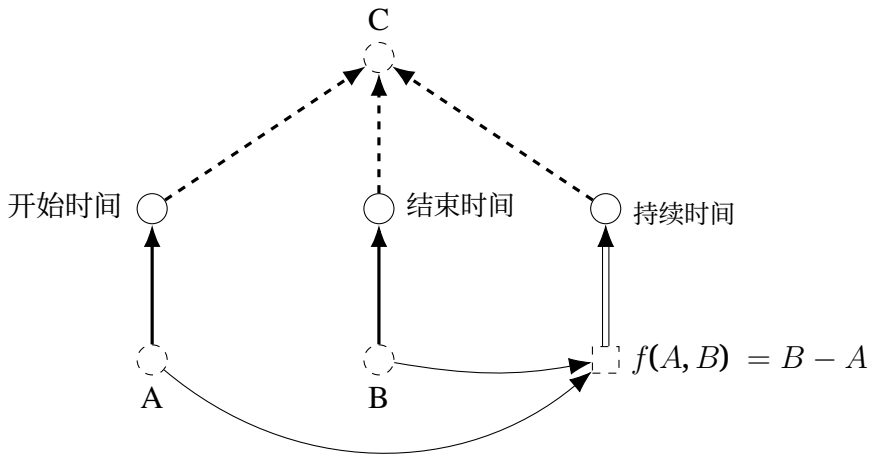


图 3.2：按需会议时长规则示意图。从 A 和 B 指向标记为 $f(A, B)$ 的节点的箭头表示 A 和 B 将用于计算，而从 $f(A, B)$ 指向“C 的持续时间”的双线表示将创建并返回的 is-a 链接。

3.3 基于配方的规划器

基于配方的规划器目前已有部分原型，但尚未完成完整实现；它基于 Fahlman 的 BUILD 规划器 [5]，并适配了 Scone 的框架以及 Eris 中所描述的情景知识表示。BUILD 描述了一种规划系统，其应用场景是在桌面上移动积木。在所考虑的场景中，桌面起初具有某种积木配置，目标是在不造成任何不稳定的情况下，将积木重新排列为某个目标状态。BUILD 规划器通过将目标分解为子目标，并使用不同的动作规划如何实现每个子目标来达成这一目的。如果规划过程中进入因不稳定而无法继续的状态，它就会递归回溯到先前的状态，并尝试不同的动作。通过这种方式，规划器可以避免陷入错误状态，并且通常总能找到推动进展的方案。

BUILD 利用 Scone 的多上下文系统，可以有效实现其回溯到先前状态并考虑不同替代方案的核心能力。在 Scone 中，关于当前状态的信息存储在活动上下文中。在考虑要执行的动作时，活动上下文充当该动作的 {前置上下文}角色，而有关结果状态的新信息则会存储在该动作的 {后置上下文} 角色中。基于配方的规划器可以通过激活 {后置上下文}并查看结果状态，来考虑执行此动作后会发生什么。如果最终进入一个不理想的状态，使得实现子目标变得困难或不可能，那么规划器只需重新激活 {前置上下文}，即可撤销该动作的后果，并尝试其他可能的动作。

Eris 中所述的 Scone 存储情景知识的方式，也指导着基于配方的规划器如何将目标分解为子目标，并尝试不同的策略来实现不同目标。假设规划器正试图实现前往匹兹堡机场这一目标。这个目标 {前往匹兹堡机场} 可以在 Scone 中表示为一种动作类型（即，这个元素是 {动作} 的子类型），并且还可以有进一步的子类型，例如 {开车前往匹兹堡

机场} 或 {乘公交车前往匹兹堡机场}。is-a 层次结构中的分支动作类型代表要采取的截然不同的计划：开车去机场和乘公交车去机场各自需要考虑特定的一组因素，而且这两种计划互不相容。

动作类型也具有由 {part of} 角色所形成的 part-of 层级结构，这些角色在每个计划中对应不同的子目标。例如，{开车前往匹兹堡机场} 这一动作可以包含一些组成部分，例如 {拿车钥匙}、{将车钥匙插入点火装置}、{在某某街左转}，等等。不同的动作有不同的组成部分，因此也有不同的子目标；例如，{乘公交车前往匹兹堡} 这一动作包含 {前往公交车站} 和 {支付车费} 这两个组成部分。动作的每个组成部分都可以继续细分，直到该动作最终达到某个基本动作，例如容易执行的身体运动。如果某个子目标因任何原因无法实现，规划器就会尝试另一组子目标来摆脱困境：如果道路因施工而封闭，无法左转，规划器可以尝试寻找另一条行车路径。如果这也行不通，规划器就可以判定当前动作无法完成，并尝试另一种计划：如果找不到车钥匙，就无法开车，因此必须改用其他方式前往机场。

3.4 实现状态

产生式规则的核心结构已在代码中实现，并加入了 Scone 引擎。Scone 使用 Common Lisp 开发，因此我们通过扩展引擎中的 Common Lisp 代码来实现产生式规则。我们将一条产生式规则表示为一个 Lisp 结构体，其各个组成部分分别对应规则的不同部分，包括变量、谓词和动作。添加时规则和需要时规则共用相同的底层数据结构。最重要的是，我们定义了两个用于创建新规则的 Lisp 宏，一个用于创建 if-added 规则，另一个用于创建 if-needed 规则。这些宏的用法与 Scone 函数类似，例如用于向知识库添加知识的 new-type 和 new-is-a。附录中给出了定义上述示例规则的宏调用示例。

基于配方的规划器的实现超出了本研究的范围，将留待未来工作完成；我们介绍其设计，旨在为理解产生式系统规划器的目标与基于配方的规划器的目标之间的关系提供背景。

现在我们已经有了一个框架，可以向 Scone知识库系统添加新规则，因此还需要一个系统来检查并触发这些规则。在下一节中，我们将介绍规则检查引擎；该引擎已在 Scone 引擎中完整实现。

第4章

规则检查引擎

4.1 规则检查组件

Scone 是一个非常庞大的知识库，规则可以引用知识库中的任意元素子集，因此在检查规则是否得到满足时必须格外谨慎。持续检查所有规则是否得到满足非常浪费，因为只有在添加新知识后，规则才可能得到满足，而且即便如此，得到满足的规则也只有少数（如果有的话）。此外，将每个元素都直接与规则变量进行匹配非常低效，因为需要检查的元素太多了。

为了解决第一个问题，每当创建一条规则时，都会创建一组规则“触发器”，并将其附加到规则涉及的所有元素上。随后，在为添加时规则添加新知识时，以及为需要时规则请求角色值时，都会检查这些触发器。为了解决第二个问题，规则系统会执行一种规则检查搜索算法，利用 Scone 的标记传递操作搜索可能满足规则的相关元素。

4.1.1 规则触发器

添加时规则只应在向知识库添加新知识时检查，而需要时规则只应在请求角色值时检查。此外，我们只希望检查可能与新知识或请求的角色匹配的规则子集。为此，每条新创建的规则都应创建触发器，以控制何时检查该规则。我们创建三种类型的触发器：一种用于添加新的 is-a 链接，一种用于添加新的角色链接或陈述链接，还有一种用于访问角色值。每种触发器都通过元素属性放置在一个 Scone 元素上；元素属性是每个元素都可以拥有的键值对。这三种类型的触发器形式略有不同，以适应不同的触发方式。

- 设 R 是一条 if-added 规则，其中包含一个变量 X，且 X 带有 is-a 类型约束 Y。定义此规则时，会创建一个形如 (R X) 的触发器，其中包含指向 R 和 X 的指针，并将其附加到元素 Y 上。从概念上说，该触发器表示：元素 Y 的任何下位元素都可以替换规则 R 中的变量 X。因此，Y 的所有下位元素都会虚拟继承

此规则触发器。

假设创建了一个从节点 A 指向节点 B 的新 is-a 链接或等式链接。该链接可能会使附加到 B 虚拟继承的某个规则触发器上的任何规则得到满足。因此，我们对 B 执行向上扫描，以检查 B 的任一下位元素上是否存在触发器（R X）。通过将 A 代入规则 R 中的变量 X，可以激活每个此类触发器，然后在 R 上启动规则检查搜索算法。图 4.1 展示了这种触发器激活的可视化示例。

- 设 R 是一个带有 x-y-z 谓词（X Y Z）的 if-added 规则，其中 Y 是角色或关系，而 X 和 Z 是变量或节点。定义该规则时，会创建一个包含指向 R、X 和 Z 的指针的触发器，并将其附加到元素 Y 上。该触发器表示这样一个事实：链接（A B C）中的 B 若是 Y 的下位元素，则可以在规则 R 中将 A 与 X、C 与 Z 进行匹配。因此，Y 的所有下位元素都会虚拟继承该规则触发器。

  假设创建了一个新链接 (A B C)，其中 B 是一个角色节点或关系。如果 B 是角色，这个链接看起来像“A 是 C 的 B”；如果 B 是关系，这个链接看起来像陈述“A B C”。我们对 B 执行向上扫描，以检查是否存在 B 应当继承的触发器 (R X Z)。如果触发器中的 X 是一个 Scone 元素，我们就检查 A 是否是 X 的下位元素；如果不是，则不激活该触发器。如果 X 是变量，我们就在 R 中用 A 替换 X，并允许激活。对于 Z 和 C，我们也执行同样的检查。如果该触发器的激活未被阻止，并且 A 和 C 已替换为对应的变量，那么就会在 R 上启动规则检查搜索算法。图 4.2 展示了这种匹配过程。
- 设 R 是一条 if-needed 规则，其动作是 (X Y Z)，其中 X 是计算，Y 是角色，Z 是变量。定义此规则时，会创建一个形式为 (R Z) 的触发器，并将其附加到元素 Y 上。该触发器表示：如果我们想知道 C 的 B，而 B 是 Y 的下位元素，那么可以通过在 R 中用 C 替换 Z 来激活一条规则，从而得知结果。因此，Y 的所有下位元素都会虚拟地继承这个规则触发器。

  假设请求的值为“C 的 B”，其中 B 是一个角色节点。我们对 B 执行向上扫描，以检查是否存在 B 应继承的任何触发器 (R Z)。找到的每个触发器都会通过在规则 R 中用 C 替换变量 Z，并在 R 上启动规则检查搜索算法来激活。该替换操作以及由此创建的链接如图 4.3 所示。

4.1.2 规则变量替换

规则检查的一个必要组成部分是能够将 Scone 元素替换为规则中的变量。规则 R 包含有关规则中变量、这些变量的类型约束、规则的 x-y-z 谓词以及规则动作的数据。当系统尝试将元素 E 替换为规则 R 中的变量 X 时，必须先验证 E 是否为 X 的任意类型约束的下位元素，以及涉及 X 的任何 x-y-z 谓词是否均由 E 满足。如果其中任一条件不成立，则替换失败，E 替换 X 后无法满足该规则。如果替换成功，规则就会将 E 存储为 X 当前的填充项。当规则中的所有变量都成功完成替换后，该规则即得到满足，并且动作已准备好使用所有已存储的元素触发。

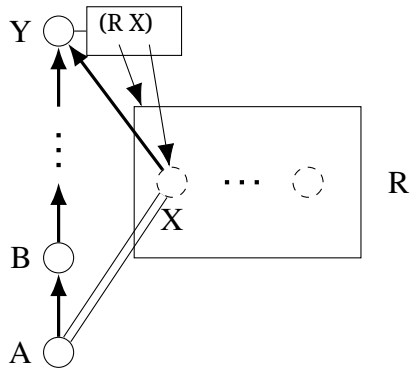


图 4.1：is-a 类型约束触发器的示意图。Y 在 is-a 层次结构中位于 B 之上，并附有一个触发器，其中包含指向规则 R 和变量 X 的指针。通过将 A 与 X 匹配来激活触发器。

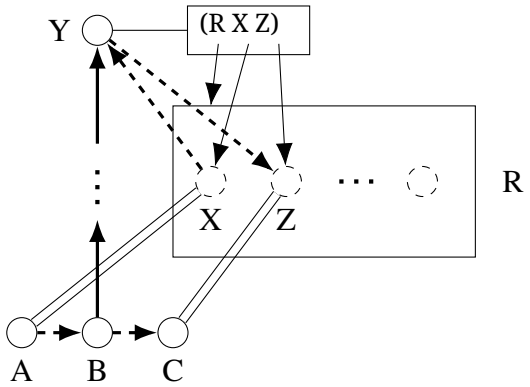


图 4.2：x-y-z 谓词触发器的示意图。Y 在 is-a 层次结构中位于 B 之上，并附有一个触发器，其中包含指向规则 R 以及变量 X 和 Z 的指针。通过将 A 与 X 匹配、将 C 与 Z 匹配来激活触发器。

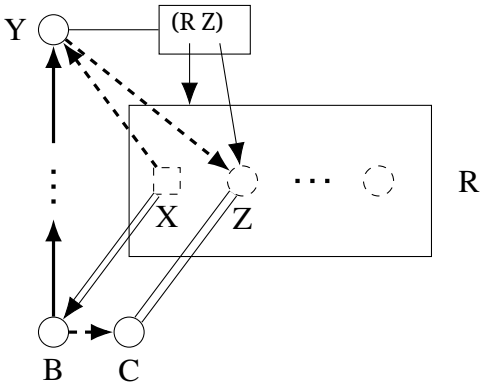


图 4.3：按需规则触发器的示意图。Y 在 is-a 层次结构中位于 B 之上，并附有一个触发器，其中包含指向规则 R 和变量 Z 的指针。通过将 C 与 Z 匹配来激活触发器。作为规则动作的一部分，在所得计算 X 与“C 的 B”之间创建一个 is-a 链接。

```
过程 检查规则(R)
    如果 R 中的每个变量都已替换为一个元素，则
        将 R 添加到待触发队列中
        返回 T
    结束 if
    从 R 中选择一个 x-y-z 谓词（X Y Z），其中 X 或 Z已替换为一个元素
    分配一个新标记 m
    如果 X 被替换为某个元素，则
        用 m 标记所有满足谓词 (X Y E) 的元素 E
    结束 if
    如果 Z 被替换为某个元素，则
        用 m 标记所有满足谓词 (E Y Z) 的元素 E
    结束 if
    对于每个标记有 E 的元素，执行 m
        R′ ← 将 E 替换为未替换的变量 X 或 Z，位于 R中
        如果 R′ 中的任何谓词为假，则
            继续
        结束 if
        检查规则(R′)
    结束循环
    返回 NIL
结束过程
```

图 4.4：规则检查算法

4.1.3 规则检查搜索算法

当规则中的至少一个变量被元素替换时，就开始检查该规则。如前所述，这种情况发生在规则的触发器被激活时。下一步是寻找一组 Scone 元素，使它们能够替换其余变量，并满足所有规则谓词。此过程通过递归搜索算法完成，该算法使用 Scone 标记操作搜索可能的元素。

从整体上看，该算法首先选择一个 x-y-z 谓词，其中一个变量已替换为一个元素，另一个变量尚未替换。然后，它使用 Scone 标记操作，标记所有在将剩余变量替换为相应元素后能够满足该谓词的元素。如果剩余变量被标记为恰当，算法便通过另一个标记操作，将标记限制为恰当元素的子集。对于每个已标记的元素，算法都会尝试将该元素代入规则。如果替换失败，算法就继续尝试下一个已标记的元素。如果没有更多已标记的元素，算法便返回 NIL。如果替换成功，算法会递归执行，以尝试为规则中的另一个变量寻找元素。如果算法成功为规则中的所有变量找到元素，就会使用找到的元素执行该规则的动作。完整的算法伪代码见图 4.4。

请注意，我们对 x-y-z 谓词必须构成连通图的约束，确保了该算法能够正确找出所有可能满足规则的元素集合。该算法的基本做法是反复选择一条连接已替换变量和未替换变量的任意边，并寻找可代入的元素。寻找合适元素的每一步都相对较快，因为只需从一个已替换的元素出发执行一次标记操作。每次递归都会分配一个标记对，因此递归深度受可用标记对数量的限制。这并不会造成问题，因为递归深度等于 x-y-z 谓词的数量，通常不超过三或四个，而可用的标记对通常约有十四个。

在某些情况下，添加一条知识可能会触发多条规则。发生这种情况时，我们会将每条满足条件的规则加入队列，并依次触发它们。在检查触发器后依次触发规则，这使该算法与 ACT-R 所用的算法相似。规则按照检查其触发器的先后顺序触发，通常从层级底部元素上的触发器开始，逐步向上，但这一顺序并没有严格定义。我们采用这种依次触发的方法，以确保规则触发具有确定性和一致性，并由 Scone 负责阻止任何会向知识库添加矛盾知识的规则动作。

4.2 端到端示例：规则触发

为了说明 Scone 的产生式系统在实践中的工作方式，我们将逐步介绍 Scone 检查并触发示例规则的过程。我们使用前面第 3.2.2 节给出的规则，但将其表述为 if-added 规则，而非 if-needed 规则。请注意，在某些情况下（例如本例），规则既可以定义为 if-added 规则，也可以定义为 if-needed 规则；选择哪种规则，取决于我们希望规则结果多快在知识库中更新。提醒一下，该规则的各个部分如下：

- 有三个变量 A、B 和 C，分别代表开始时间、结束时间和会议。
- 有两个 x-y-z 谓词：“A 是 C 的开始时间”和“B 是 C 的结束时间”。
- 该动作会将链接“B–A 是 C 的持续时间”添加到知识库中。

定义此 if-added 规则会为 x-y-z 谓词添加触发器 (R A C) 到角色 {开始时间}，并添加触发器 (R B C) 到角色 {结束时间}。此后，凡是涉及这些角色的知识库新增内容，都会导致系统检查这些触发器。

现在，这条规则是针对一组抽象元素定义的，这些元素满足规则的所有前置条件。要了解这条规则如何针对特定个体进行专门化并触发，假设我们创建一个新的个体 {会议 27}，代表一场特定会议。现在，假设我们首先将链接“10:30 AM 是会议 27 的开始时间”添加到知识库中。这会在 {开始时间} 上进行向上扫描，以查找可能被激活的继承规则触发器。实际上，在 {开始时间}上有一个触发器（R A C），因此规则引擎会检查能否用 {10:30 AM} 替换 A，以及能否用{会议 27} 替换 C。A 和 C 都没有显式类型约束，也没有其他谓词被违反，因此替换成功，触发器随之激活，如图4.5所示。

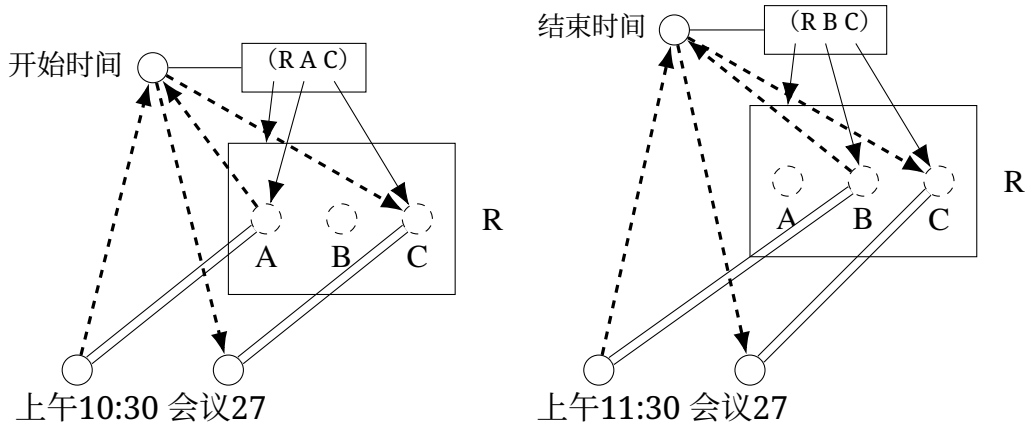


图4.5：会议示例规则的两次触发器激活的可视化，每次对应一个 x-y-z 角色谓词。

此触发器激活后，R 中的变量 A 和 C 会被替换，但变量 B 不会，因此搜索算法需要为 B 找到一个合适的元素。算法会发现规则中有一个部分替换的 x-y-z 谓词“B 是会议 27 的结束时间 ”，于是分配一对标记，并标记所有满足 “E 是会议 27 的结束时间 ” 的元素 E。（对于这条特定规则，由于变量 B 被标记为恰当，算法会进一步将标记范围限制为恰当元素。）此时实际上没有可标记的元素，因此算法返回，并确定该规则不成立。从概念上讲，我们目前只知道会议 27 的开始时间，还不知道结束时间，因此，正如预期的那样，我们还无法计算会议的持续时间。

假设我们发现“上午 11:30 是会议27的结束时间”，并将其添加到知识库中。这会在 {结束时间} 上执行向上扫描，以查找继承规则触发器，从而在 {结束时间} 上找到触发器 (R B C)。通过将 {11上午 11} 替换为 B、将 {会议 27} 替换为 C 来激活规则触发器（如图4.5所示）是成功的，因为没有谓词被违反，因此搜索算法会为尚未替换的变量 A 查找一个元素。

算法找到部分替换后的 x-y-z 谓词“A 是会议27的开始时间”，因此会分配一对标记，并标记所有满足“E 是会议27的开始时间”的元素 E（同样，只标记恰当元素）。这次，它标记了 {10上午 10}，因为知识库中有“上午 10:30 是会议27的开始时间”这一事实。对于每个已标记的元素 E，规则引擎都会尝试在 R 中用 E 替换 B，因此它会尝试在 R 中用 {10上午 10} 替换 B。此替换不会违反任何类型约束或 x-y-z 谓词，因此替换成功。

此时，规则中的所有变量都已替换为元素。这意味着对于这组特定的个体，该规则已满足，因此规则动作会被触发。计算 {11:30 AM} – {10:30 AM}，得出 {1 小时}，并将知识“1小时是会议27的持续时间”添加到知识库中。如果当前定义了任何带有继承{持续时间}触发器的规则，这条新知识可能会导致这些规则触发。

产生式规则触发过程展示了 Scone 在获得新知识时如何进行自动思考。它只会在需要时检查规则，而且只会检查娱乐登录

由于规则触发机制，只需检查特定规则。借助搜索算法，它能够获取相关知识，以检查规则是否得到满足。

再进一步说明一下将此规则设为急切或惰性的选择：如果会议的持续时间会直接影响是否参加该会议的决策阶段，那么将规则设为急切规则就是一个理由。例如，假设我们还有另一条 if-added 规则，表示“如果会议时长不超过 2 小时，就决定参加该会议”，并且默认决策是不参加任何时长超过两小时的会议。我们希望一旦知道会议的持续时间，这条附加规则就立即触发；同时，我们也希望在获得足够信息以确定会议持续时间时，它就立即触发。第一条规则提供了一种计算会议持续时间的方法，因此将其定义为 if-added 规则是合理的。

4.2.1 若需要示例

我们也完全可以将之前用于计算会议持续时间的规则定义为 if-needed 规则。如果我们希望在有人询问 {会议 27} 的持续时间时能够计算该值，但并不急于获取任何会议的持续时间，那么就可以这样做。该规则的结构基本相同，但检查规则的方式会有所不同。将规则定义为 if-needed 规则时，不是添加两个 if-added 触发器，而是在角色节点 {持续时间}上添加一个触发器 (R C)。

需要时规则只有在显式请求某个值时才会检查，因此添加“上午 10:30 是会议27的开始时间”和“上午 11:30 是会议27的结束时间”这两条知识不应激活任何触发器。 {开始时间} 和 {结束时间}上不存在任何继承的触发器，确保了这一行为。一旦请求槽填充值 {会议 27}的持续时间，就会对角色节点 {持续时间}执行向上扫描，从而找到触发器 (R C)。将 {会议 27} 替换到 R 中的 C 后，该触发器便会激活，因为这不会违反任何谓词。

搜索算法现在开始为变量 A 和 B 查找元素。两个 x-y-z 谓词都已被部分替换，因此可以任意选择其中一个，例如“ A 是会议27的开始时间”。省略标记传递和替换的具体细节后，算法发现可以用 {10:30 AM} 替换 R 中的 A。然后，算法查找能够满足“C 是会议27的结束时间”的元素。它找到元素 {11:30 AM} ，并用该元素替换 R 中的 C。

现在，所有变量都已替换，因此可以触发规则动作“B–A 是 B 的持续时间”。该规则动作的计算返回值为 {1 hour}。规则动作将计算得到的链接“1 hour 是会议27的持续时间”添加到知识库，并返回 {1hour} 作为所请求的槽填充值。请注意，该规则动作执行一次后，下次 {请求会议 27} 的持续时间时，所需知识已存在于知识库中，因此无需再次检查该规则。

第5章

结论与未来工作

在本论文中，我们提出了一种与 Scone知识库系统兼容的 if-added 和 if-needed 产生式规则设计方案。我们还实现了触发器和方法，用于高效检查规则是否得到满足。我们的工作主要属于我们为 Scone 设想的“基于规则的规划器”，它会响应新增或请求的知识，执行低层自动思考。我们还概述了“基于配方的规划器”的设计，以说明这两类规划器的设计目标。

未来对 Scone 产生式系统的研究，可以通过多种方式扩展产生式规则的功能。一种扩展是支持 if-removed 规则，在规则谓词中加入负类型约束，例如“A 不是大象”。当链接被移除或取消时，这些规则就会触发。要支持这一功能，需要使用 Scone 的取消链接功能，而确保逻辑行为的一致性将是一项挑战。我们还希望将规则引擎应用于规模大得多的知识库，并使用大量规则，以揭示潜在的扩展性或一致性问题。

规则检查的另一种可能方法是反刍，这是 Learning Reader 用来执行离线推理、从自然语言中学习的一种技术 [9]。在 Scone 的情境中，如果系统目前正在处理一条低优先级规则，但需要将资源用于其他更高优先级的任务，就可以推迟检查该规则，直到未来某个资源充裕的时刻。未完成的规则会加入队列，留待之后处理。在反刍过程中，系统会检查未完成规则的队列，并处理其中的一部分，使其实际触发并更新知识库。

一项重要的未来研究目标是为 Scone 实现“基于配方的规划器”，使其能够将目标拆分为多个子目标，并通过考虑多个情境下的不同选项，规划如何实现每个子目标。结合基于规则的规划器，这些规划器将引导智能体做出决策，以实现既定目标。

通过结构化或非结构化的自然语言，以更简单、更自动化的方式定义规则，是一个值得研究的方向。一个例子是使用结构化语言，以更具声明式的形式定义规则，例如：“会议的开始时间加上持续时间等于该会议的结束时间。”目标是让这种声明式形式生成三条规则，在已知会议的其他两个值时，分别计算开始时间、持续时间和结束时间。目前，研究人员正致力于从非结构化自然语言句子的外部来源，自动向 Scone 知识库添加知识；这项研究

搜索最终也可以扩展到自动添加产生式规则。

最后，能够自动学习产生式规则或许是一个值得追求的长期目标，不过这可能超出 Scone 的能力范围。Soar 和 ACT-R 描述了自动学习产生式规则的方法 [1, 12], ，但这些方法通常涉及来自外部的反馈或环境输入。Soar 的一种学习方法称为组块化：当系统陷入僵局且没有现有的产生式规则适用时，系统便会学习新的产生式规则。在这种情况下，Soar 会创建一个新的子目标来解决僵局，并创建一条新的“组块”规则，以避免今后在类似情境中再次陷入僵局。在 Scone 中，基于配方的规划器会负责规划如何度过此类僵局，也可以采用类似的方法来学习新的辅助产生式规则。

参考文献

[1] John R Anderson。《Act：复杂认知的简单理论》。American psychologist，51 (4):355，1996。[2] John R. (John Robert) Anderson。《人类思维如何存在于物理宇宙中？》Oxford series on cognitive models and architectures。牛津大学出版社，Oxford；，2007。ISBN 0195324250。[3] Jelmer Borst、Niels Taatgen 和 Hedderik Rijn。《问题状态：多任务处理中的认知瓶颈》。Journal of experimental psychology. Learning, memory, and cognition，36: 363–82，03 2010。doi: 10.1037/a0018106。[4] Ivan Bratko。《面向人工智能的 Prolog 编程》。Pearson education，2001。[5] Scott Elliott Fahlman。《机器人施工任务的规划系统》。Artificial Intelligence，5(1):1–49，1974。ISSN 0004-3702。doi: https://doi.org/10.1016/0004-3702(74) 90008-3。URL https://www.sciencedirect.com/science/article/pii/ 0004370274900083。[6] Scott E Fahlman。《Scone 知识库系统中的标记传递推理》。载于《知识科学、工程与管理国际会议》，第 114–126 页。施普林格，2006年。[7] Scott E Fahlman。《Eris：Scone 中的情景表征与推理》。技术报告，Citeseer，2014。[8] Scott E Fahlman。《Scone 教程信息》，2018。URL https://fahlman-knowledge-nuggets.quora.com/Tutorial-Information-on-Scone。[在线；访问日期：18-April-2021]。[9] Kenneth D Forbus、Christopher Riesbeck、Lawrence Birnbaum、Kevin Livingston、Abhishek Sharma 和 Leo Ureel。《整合自然语言、知识表示与推理以及类比处理，通过阅读学习》。载于《全国人工智能会议论文集》，第 22 卷，第 1542 页。Menlo Park, CA；Cambridge, MA；London；AAAI Press；MIT Press；1999，2007。[10] Charles L Forgy。《Rete：一种用于多模式/多对象模式匹配问题的快速算法》。载于《人工智能与数据库读本》，第 547–559 页。Elsevier，1989。[11] Ivan Herman。《W3C 语义网活动》，2013。URL https://www.w3.org/2001/ sw/。

[12] John E Laird。《Soar认知架构》。麻省理工学院出版社，2019年。[13] Jill Fain Lehman、John Laird、Paul Rosenbloom 等。《Soar简介：人类认知架构——2006年更新版》。密歇根大学，第1–37页，2006年。[14] Allen Newell。《Soar作为统一认知理论：问题与解释》。《行为与脑科学》，15(3)：464–492，1992年。[15] Alessandro Oltramari 和 Christian Lebiere。《通过利用 ACT-R 的知识能力追求通用人工智能》。载于《国际通用人工智能会议》，第199–208页。施普林格，2012年。

附录

我们已在主 Scone 引擎中使用 Common Lisp 实现了产生式规则系统[1]。其中有两个新的用户级宏，用于向知识库添加新的添加时规则和需要时规则。new-if-added-rule 是一个宏，它接受绑定、x-y-z-preds 以及任意数量的主体形式作为参数。
new-if-needed-rule 接受绑定、x-y-z-preds 以及一个动作作为参数。

每个绑定要么是单个变量，要么是包含一个变量以及可选关键字参数 :superior 和 :proper 的列表。每个 x-y-z 谓词都是一个三元素列表，其中第二个元素是角色或关系。对于添加时规则，动作是由主体形式剩余部分定义的任意代码。对于需要时规则，动作的形式为 (X Y Z)，其中 X 是计算，Y 是个体角色节点，Z 是绑定中定义的变量。

第 3.2.1 节中描述的示例 if-added 规则可以使用宏定义如下：

(new-if-added-rule(a (b :上级 飞机)) ((b 旅行交通工具 a))
(new-is-a a 飞行事件))

运行此宏会将此规则添加到知识库，并为此规则在{飞机} 和 {旅行交通工具}上添加触发器。假设 Scone 元素定义正确，我们可以使用以下代码片段测试规则是否正确触发：

(new-indv {my trip} {旅行事件}) (new-indv {my vehicle} {飞机})
(x-is-a-y-of-z{my vehicle} {旅行交通工具} {my trip}) (assert
(simple-is-x-a-y? {my trip} {飞行事件}))

这段代码创建了一个新的旅行个体，以及一个作为飞机的新交通工具个体，并将{my vehicle} 设为此次旅行的 {旅行交通工具} 。该断言成功，表明规则成功触发，并将链接“my trip is a flying event”添加到了知识库中。

第 3.2.2 节中描述的示例 if-needed 规则可以使用宏定义如下：

```
(new-if-needed-rule ((a :proper t) (b :proper t) c)
                    ((a {start time} c)
```

[1]产生式规则引擎的代码目前托管在 https://github.com/jchen1352/ scone/tree/develop

(b {结束时间} c))
((scone-subtract b a) {持续时间} c))

为简单起见，假设 {开始时间} 和 {结束时间} 是表示自纪元以来秒数的 Scone 数，而 scone-subtract 是一个适当定义的 Lisp 函数，可以对两个 Scone 时间元素进行减法运算。我们可以用下面这段代码测试规则是否正确触发：

```
(new-indv {my meeting} {会议}) (x-is-the-y-of-z{1609459200} {开始时间} {my meeting}) (x-is-the-y-of-z{1609462800} {结束时间} {my meeting}) (assert (is-x-eq-y? {3600} (the-x-of-y {持续时间} {my meeting})))
```

这段代码会创建一个新的会议个体，并设置其开始时间和结束时间。the-x-of-y 负责检查潜在的需要时规则，而断言成功则表明该 if-needed 规则已成功触发，并将链接“3600 秒是我的会议的持续时间”添加到了知识库中。