安全策略重写的追踪类型

RONGXIAO FU，英国爱丁堡大学ORNELA DARDHA，英国格拉斯哥大学MICHEL STEUWER，英国爱丁堡大学

arXiv:2304.14154v1 [cs.PL] 27 Apr 2023

策略语言使程序员能够将重写规则组合成策略，并控制其应用。这在程序设计语言中非常有用，例如，可以组合地描述程序变换；在自动定理证明中也同样有用，因为相关思想已通过战术语言得到研究。显然，并非所有重写的组合都是正确的，但我们如何帮助程序员编写正确的策略?

本文提出了一种用于策略语言的静态类型系统。我们将捕获重写策略如何改变被重写语法形状的结构类型系统，与一种新颖的追踪系统相结合，后者会跟踪所有可能的合法策略执行路径。当组合中的某些部分在运行时必然失败时，我们的类型系统会发出警告；当某个策略不存在合法执行路径时，则会报告错误。

我们给出了策略语言和新颖追踪类型系统的形式化，并正式证明了其类型可靠性。我们还给出了形式化结果，表明追踪不良的策略在运行时必然失败，而良好追踪的策略执行“不会出错”，这意味着它们必然存在一条可能成功的执行路径。

1 引言

重写系统在许多领域都有应用，从逻辑 [Marchiori 1994] 和定理证明器 [Hsiang 等人 1992] 到程序设计语言语义的定义 [Rosu 和 Serbanuta 2010] 以及编译器中的程序变换 [Visser 2001]。在许多领域，只需简单地指定一组重写规则即可——每条规则都规定一个单独的重写步骤——这些规则由重写系统应用（可能以非确定方式），直至达到正规形式或不再有规则适用。

但在许多领域，这样做并不够。例如，重写规则是编码简单程序变换的直接选择，但不足以编码更复杂的程序变换，也不足以在实践中控制其应用。可能需要仅对程序的某个子部分应用规则、按特定顺序应用多条规则，或在某个特定阶段启用或禁用规则。为了控制重写规则的应用，策略语言（例如 Stratego）[Bravenboer 等人，2008；Visser 等人 1998] 提出了策略语言。Kirchner [2015]对该领域进行了最新概述。这些策略语言通过提供将重写规则组合成更大策略的组合子，实现了策略重写。

Stratego 是 Spoofax 语言工作台的重要组成部分，由 Kats 和 Visser [2010]设计，用于以声明式方式指定语言及与之配套的 IDE。本文使用 Stratego 策略语言，仅通过重写的组合来编写解释器和编译器。

Hagedorn 等人 [2023, 2020] 介绍了 ELEVATE 策略语言，以及如何使用它以组合方式对传统编译器优化（如循环分块）进行编码和控制。所实现的性能与传统设计的 TVM 编译器 [Chen 等人 2018] 在深度学习领域中的表现相当。事实上，ELEVATE 展示了如何重新思考“用户可调度语言”的设计，将其设计为策略语言，正如Ragan-Kelley [2023]所强调的那样。调度方案允许专家精确控制要应用哪些编译器优化。这一理念由领域专用编译器 Halide 推广开来 [Ragan-Kelley 等人 2013]，如今已被其他优化编译器广泛采用，例如 TVM。

作者地址：Rongxiao Fu，爱丁堡大学，英国，s1742701@ed.ac.uk；Ornela Dardha，格拉斯哥大学，英国，ornela.dardha@glasgow.ac.uk；Michel Steuwer，爱丁堡大学，英国，michel.steuwer@ed.ac.uk。

与策略语言密切相关的是策略语言在自动定理证明领域中的应用，它们允许控制各个证明步骤的排列。这些思想可以追溯到Gordon 等人 [1978]，如今策略语言已出现在许多证明助手中，例如 Coq[Delahaye 2000]。

策略语言（如 Stratego 和 ELEVATE）使程序员能够将复杂策略描述为重写的组合。但显然，并非所有重写组合都是正确的。举个简单的例子，考虑以下两个重写规则的顺序组合（;）：**规则**：

```
let broken_composition = rule x + y -> y + x ; rule 1 * z -> z
```

在这一组合中，我们先应用交换律重写，然后尝试移除作为乘法幺元的 1。但显然，这里出了问题！第一条重写会生成包含加法运算符的表达式，而第二条重写却要求乘法运算符。这个重写组合绝不可能奏效，并且对于任何可能的输入，都会在运行时失败。虽然这是一个简单的例子，却引出了一个普遍问题：我们如何帮助程序员避免此类错误，并编写正确的重写组合策略?

本文提出一种新颖而强大的策略语言静态类型系统，能够拒绝诸如上述示例那样组合不当的策略。为此，我们使用行多态变体类型，将被重写表达式的语法（例如 `x+y`）编码为结构类型。接着，我们将重写规则及其组合表示为类似函数的策略类型**策略类型**，以反映重写所描述的语法程序变换。将策略应用于输入时，我们的类型系统会为结果值赋予一个**策略结果类型**，以反映重写是成功执行并生成重写后的表达式，还是无法应用而导致运行时失败。最后，为了处理以策略为参数的高阶组合子，我们为其定义了专用的**策略组合子类型**。我们将结构类型系统与一种新颖的**追踪类型系统**结合起来，静态检查重写组合是否存在成功的执行路径，从而拒绝那些对于所有可能输入都必然导致运行时失败的重写组合。我们的追踪类型系统不仅能像上文所示那样保证顺序组合的这一性质，也能处理选择组合子（||），它会在两种可能的重写策略中择一。

我们对静态类型核心策略语言 *Typed ELEVATE* 进行了形式化，该语言包含类似于 ELEVATE（以及 Stratego）中已有的组合子。我们介绍了该语言新颖的类型系统：将结构类型与轨迹相结合，并证明了其类型可靠性。此外，我们给出了形式化结果，表明结构类型良好且追踪良好的策略不会产生组合错误，例如上述示例中的错误；并且对于适当的输入，其执行保证存在一条可能成功的执行路径。相反，我们形式化地证明了，结构类型良好但未进行良好追踪的策略必然会导致运行时失败，因此被我们的类型系统合理地拒绝。

总结而言，本文作出以下贡献：

- 我们为策略语言引入了一种强大的静态类型系统，将结构类型与新颖的带跟踪类型相结合，以拒绝组合不当的策略；
- 我们讨论了若干示例，展示该类型系统在实践中的应用（第2节）；
- 我们给出静态类型核心策略语言的形式化定义*Typed ELEVATE* 及其类型系统（第3.1–3.6节），并证明其类型可靠性（第3.7–3.8节）；

- 我们正式定义类型良好且轨迹良好的策略所代表的含义：它们“不会出错”；并证明类型良好但不具备良好轨迹的策略必定会导致运行时失败（第3.9节）。

我们将在第4节讨论局限性和设计选择，然后在第5节介绍相关工作，并在第6节总结。

2 Typed ELEVATE 示例

本节将通过示例介绍 Typed ELEVATE。首先在第2.1 节讨论有效的重写组合，并展示我们的类型系统为这些策略赋予的类型。随后，我们将在第2.2 节讨论有问题的组合，这些组合会被我们的类型系统标记为警告；并在第2.3节讨论类型系统会以错误拒绝的组合。

2.1 有效的策略组合、策略应用和策略组合子

2.1.1 示例 *1*。 先从三个重写规则的简单组合开始：

```
let e1 = rule m * n -> n * m ; (rule 1 * v -> v || rule 2 * w -> w + w)
```

要定义策略 `e1`，我们首先交换两个因子 `n` 和 `m`，然后移除与 `1` 进行的不必要乘法，或者将与 `2` 进行的乘法转换为加法。我们使用顺序组合运算符（`;`），它将第二个策略应用于第一个策略的输出；还使用选择组合子（`||`），它会应用两个策略中的一个；如果两个策略都不适用，则失败。

为了将我们重写的表达式（例如 `m * n`）表示为类型，我们使用行多态变体类型对其进行编码，具体说明见第3节。在本节中，我们将以非正式方式书写这些结构类型，就像向类型系统的用户展示它们一样。作为策略的重写规则及其组合会被赋予一个策略类型，该类型反映了重写所描述的语法转换。例如，第一个重写规则具有以下策略类型：

`rule m * n -> n * m` : $a_0 * a_1 \xrightarrow{[a]} a_1 * a_0$

类型中出现的变量是轨迹变量。这里所有变量都属于由 a 标识的单个轨迹，这表示执行该策略所描述的重写只有一种可能的方式：交换乘法的两个操作数，由两个轨迹变量 a_0 和 a_1 表示。

两个其他重写的选择组合所对应的策略类型更有意思：

`(rule 1 * v -> v || rule 2 * w -> w + w)` : $1 * b_0 \mid 2 * c_0 \xrightarrow{[b,c]} b_0 \mid c_0 + c_0$

这里的策略类型包含两条轨迹，b 和 c，这表明该重写策略有两种可能的执行方式：要么执行第一条重写规则，将 $1 * b_0$ 转换为 b_0；要么执行第二条重写规则，将乘法 $2 * c_0$ 转换为 $c_0 + c_0$。当一个策略类型中包含多条轨迹时，我们会分别写出每条轨迹的输入和输出，并以 $\mid$ 分隔，以表示这些是该策略彼此分离的可能执行方式。在我们的形式系统中，类型并不是以这种方式表示的。我们将在第 3 节说明，为什么这种简化且直观的类型表示能够如实反映形式类型中可用的信息。

按顺序组合这两个策略后，我们得到的策略类型反映了整体策略的效果：

```
let e1 = rule m * n -> n * m ;
         (rule 1 * v -> v || rule 2 * w -> w + w) : d0 * 1 | e0 * 2 -[d,e]-> d0 | e0 + e0
```

策略类型现在表明，有两种可能的方式可以成功执行组合策略：输入表达式的形式为 $d_0 * 1$，并将其重写为 d_0；

或者，输入的形式为 $e_0 * 2$，并将其重写为 $e_0 + e_0$。对于任何其他作为输入的表达式，组合策略都将失败。

2.1.2 示例 2. 我们也可以检查重写策略对输入表达式的应用吗？请看下面的示例：我们将表达式 `5 * 2` 作为输入传递给示例 *1* 中的策略 `e1`，然后将所得表达式作为输入传递给另一条重写规则：

```
let e2 = (rule 5 + 5 -> 10) (e1 (5 * 2))
```

将 `5 * 2` 应用于 `e1` 的结果具有策略结果类型：$[b] \blacktriangleright 5 + 5$。在此情形下，它表示重写成功的表达式，写在黑色三角形的右侧。右侧没有出现轨迹变量b_i，因为该表达式只包含常量，不包含变量。

以此类型作为规则 `(rule 5 + 5 -> 10)` : $5+5 \xrightarrow{[a]} 10$的输入，我们得到：

```
let e2 = (rule 5 + 5 -> 10) (e1 (5 * 2)) : [c] ▶ 10
```

此类型确认该策略执行有效，并将生成表达式`10`。

2.1.3 示例 *3*。接下来，让我们考虑两个选择组合的顺序组合：

```
let e3 = (rule m * n -> n * m || rule m + n -> n + m) ;
         (rule 1 * v -> v     || rule 0 + w -> w)
```

这里，第一行中的选择组合交换乘法或加法的操作数，第二行中的选择组合移除乘法或加法中的单位元。通常，将两个选择组合进行顺序组合时，有四条可能的执行路径：左-左、左-右、右-左和右-右。但显然，在这个示例中，只有两条路径会成功执行：一条交换操作数并移除乘法中的幺元（左-左），另一条对加法执行相同操作（右-右）。为体现这一点，这个示例也可以写成：

```
(rule m * n -> n * m ; rule 1 * v -> v) || (rule m + n -> n + m ; rule 0 + w -> w)
```

我们的追踪类型系统能够在类型中精确反映所有可能的执行路径。各个选择组合的类型为：

`(rule m * n -> n * m || rule m + n -> n + m)` : $a_0 * a_1 \mid b_0 + b_1 \xrightarrow{[a,b]} a_1 * a_0 \mid b_1 + b_0$和

`(rule 1 * v -> v     || rule 0 + w -> w)` : $1 * c_0 \mid 0 + d_0 \xrightarrow{[c,d]} c_0 \mid d_0$

按顺序组合它们时，我们的追踪类型系统会连接这两种策略类型的轨迹，以计算表示所有可能执行路径的组合的可能轨迹。

```
let e3 = (rule m * n -> n * m || rule m + n -> n + m) ;
         (rule 1 * v -> v     || rule 0 + w -> w) : e0 * 1 | f0 + 0 -[e,f]-> e0 | f0
```

组合的最终类型清楚地表达出，整体策略会将 $e_0 * 1$重写为 e_0，或将 $f_0 + 0$ 重写为 f_0。对于所有其他输入，该策略都必须失败。

我们将在第 3 节正式讨论我们的类型系统如何计算策略组合的连通轨迹。

2.1.4 示例4。 到目前为止，我们只见过使用两个基本组合子构建的策略 `;` 和 `||`。不过，我们也可以构建自定义策略组合子，它们以策略为参数并扩展其行为。例如，考虑 `swapOps` 组合子：它接受一个策略 `s`，并在交换任意二元运算符 `op`的操作数后应用该策略。

```
let swapOps = st s => rule m op n -> n op m ; s
```

此自定义组合子可用于交换转换二元表达式的任何策略的操作数顺序，例如：`swapOps (rule 1 * v -> v)` 其类型为：$b_0 * 1 \xrightarrow{[b]} b_{00}$ 策略组合子具有专用的箭头类型 $\Rightarrow$ 对于我们的自定义组合子，该类型为：

```
let swapOps = st s => rule m op n -> n op m ; s : ((a0 a2 a1) -[a]-> a3) ⇒ ((a1 a2 a0) -[a]-> a3)
```

此类型表示策略输入表达式将其顺序从 $a_0\ a_2\ a_1$ 变为 $a_1\ a_2\ a_0$，清楚地展示了操作数的交换。

2.2 导致警告的问题策略组合

在展示了一些有效策略的示例之后，我们现在来研究有问题的策略。在本节中，我们将研究一个仍能正确执行的示例，但其中包含一个死执行分支，而我们的类型系统能够检测到该分支并向用户发出警告。

2.2.1 示例 *5*。 让我们考虑以下策略：

```
let e5 = (rule m + n -> n + m ; rule m * n -> n * m) || rule 1 * v -> v
```

这是一个左侧为顺序组合、右侧为重写规则的选择组合。该顺序组合显然存在问题，因为第一个重写规则产生的表达式与第二个规则不兼容。我们的类型系统会为该顺序组合赋予一个带有空轨迹（$\_ \xrightarrow{[]} \_$）的策略类型，这表示不存在可能的执行路径。

但是，使用右侧的规则时，整体策略 `e5` 仍然可以正常工作。因此，在这种情况下，整体组合与右侧规则具有相同的类型：$1 * a_0 \xrightarrow{[a]} a_{00}$ 这表明，对于作为右侧规则输入的表达式，组合策略将能够正常工作。

为了提醒用户注意存在问题的顺序组合，我们的类型系统实现可以向用户发出警告：

```
let e5 = (rule m * n -> n * m ; rule m + n -> n + m) || rule 1 * v -> v : 1 * a0 -[a]-> a0
//       ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//       Warning: this strategy is guaranteed to fail at runtime,
//                but the overall strategy can still succeed
```

在第3节中，我们将看到类型规则**T-S-App**，当结论中的轨迹集合变为空时，该规则可以立即触发这些警告。

2.3 导致错误的问题策略组合

最后，在本节中，我们将讨论一些有问题的策略示例：对于所有可能的输入，它们都保证会在运行时失败，因此会被我们的类型系统拒绝。

2.3.1 示例 *6*。 首先，考虑一个有问题的顺序组合：

```
let e6 = rule 2 * n -> n + n ; rule 2 + 3 -> 5
```

乍看之下，这似乎是一个有效的组合，因为第一条规则生成了一个乘法表达式，而第二条规则正需要这种表达式。但是，显然我们不可能同时将同一个变量 `n` 实例化为 `2` 和 `3`！在我们的类型系统中，这一点会被形式化地检测出来，因为统一类型后不会存在轨迹。这与前面的示例情况类似，在前面的示例中我们只报告了一个警告。我们如何区分何时发出警告，以及何时因错误而拒绝一个程序？这取决于使用空轨迹策略的上下文。如果它被用于更大的组合中（如前面的示例），我们只会发出警告；但如果我们尝试使用 `let` 将空轨迹策略绑定到一个名称，就会引发错误：

```
let e6 = rule 2 * n -> n + n : 2 ∗ a0 --[a]--> a0 + a0
       ; rule 2 + 3 -> 5      : 2 + 3 --[b]--> 5
//     ^~~~~~~~~~~~~~~~~~~~~~
//     Error: this sequential composition is guaranteed to fail at runtime
//            resulting in a guaranteed failure of the overall strategy
//
//            There is no trace for the composed strategy type
//            after unification of a0 + a0 and 2 + 3
```

2.3.2 示例 *7*。 最后，让我们考虑一个稍微复杂些的示例，其中包含两个选择组合的顺序组合：

```
let e7 = (rule m * n -> n * m || rule m + n -> n + m) ;
         (rule v - 0 -> v     || rule w / 1 -> w)
```

在这里，不可能成功执行，因为第一条策略产生的可能表达式必须包含乘法或加法，而这与第二条策略所期望的减法或除法不兼容。我们可以为每个选择组合赋予一个有效的策略类型，但在将第一条策略的输出类型与第二条策略的输入类型统一后，无法为该组合找到轨迹：

```
let e7 = (rule m * n -> n * m) || (rule m + n -> n + m) : a0 ∗ a1 | b0 + b1 --[a,b]--> a1 ∗ a0 | b1 + b0
       ; (rule v - 0 -> v      || rule w / 1 -> w)      : c0 − 0 | d0/1 --[c,d]--> c0 | d0
//     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//     Error: this sequential composition is guaranteed to fail at runtime
//            resulting in a guaranteed failure of the overall strategy
//
//            There is no trace for the composed strategy type
//            after unification of a1 ∗ a0 | b1 + b0 and c0 − 0 | d0/1
```

在第 3 节中，我们将看到类型规则 **T-Let**，它在遇到空的带轨迹策略时会报错 .

在本节中，我们看到了许多示例，展示了我们为策略和策略组合子赋予的类型。我们看到了成功组合的策略，也看到了会被警告标示为有问题或因错误而被拒绝的策略。关键在于，轨迹 指示了策略在运行时可能采取的执行路径。当轨迹集合为空，表示不存在任何可能的执行路径时，系统就会识别出有问题的策略。在下一节中，我们将把这些直觉形式化，并正式证明，类型良好但空迹的策略在运行时必然失败，从而说明我们的类型系统为何会拒绝此类策略。此外，我们还将证明我们的类型系统是健全的，并正式阐明对于类型良好且带有良好轨迹的策略而言，“不会出错”意味着什么。

3 TYPED ELEVATE，形式化

本节介绍 Typed ELEVATE 的形式化。除非另有说明， *NavyBlue* 均为元层描述，例如，方括号 **[ ]** 表示 EBNF 语法中的可选项；语法结构的带索引的多次出现（可能由、分隔）*expr*统一表示为 $(expr_i)_{i\in N}$，其中索引（写作 i、 j、 k 或 p 或 q）遍历自然数集的某个可能为空的子集（写作 M、N、U 或 V）。当同一公式中出现两个索引集时，默认它们互不相交。

$$
\begin{array}{lll}
\text{模式} & p := & x,\ y,\ z,\ \ldots \mid () \mid (p_m,\ p_n) \mid \ell\ p \\
\text{项} & e := & x,\ y,\ z,\ \ldots \mid \text{规则}\ p \to e \mid () \mid (e_m,\ e_n) \mid \ell\ e \mid \\
 & & \text{seq} \mid \text{choice} \mid \text{st}\ x \Rightarrow e_b \mid e_f \Leftarrow\ e \mid \\
 & & e_s \leftarrow e_i \mid \text{成功}\ e \mid \text{失败} \mid e_m \diamond e_n \mid \\
 & & \text{let绑定}\ x = e_f \text{in}\ e
\end{array}
$$

图 1. 项和模式的语法。

3.1 项语法

图1展示了项和模式的语法。

模式。 模式（记作 p）用于重写规则定义，包括取自有效变量名的模式变量（记作 x、y、z、$\cdots$）、单位模式（()）、对模式（$(p,\ p)$）以及变体模式（$\ell\ p$），其中 ℓ 是要匹配的标签。为简单起见，本文中的所有模式都是线性的，即每个绑定的模式变量在一个模式中只出现一次。模式的线性要求由图5中的类型规则强制执行。

项。项（记作 e）包括变量，这些变量取自有效的变量名（记作x、y、z、$\cdots$）。作为语言的基本组成部分，可以使用类似 lambda 的语法（规则 $p \to e$）定义重写规则，其中 p 是要匹配的模式，即规则的左侧（Left-hand Side），而 e 是使用 p 中的变量及其他语法组件构造的规则右侧（Right-hand Side）。因此，项语法提供了单位构造器（()）、对构造器（(e_m, e_n)）以及变体构造器或标签应用（$\ell\ e$），供规则定义内部使用。不难看出，这些项的语法与模式的语法相似，因此可以很容易地定义一个从模式到项的转换 p2e（见附录中的图13）。这里有两个常量策略组合子：用于顺序组合的 seq，以及用于策略的（非确定性）选择组合的 choice。为了允许程序员自行定义策略组合子，项语法还提供了策略组合子抽象（$\text{st}\ x \Rightarrow e_b$），其中 x 是表示策略的绑定变量，e_b 是组合子定义的主体；同时还提供了策略组合子应用（$e_f \Leftarrow\ e$），其中 e_f 是策略组合子，e 是参数策略。为了获取策略的执行结果，可以将输入 e_i 应用于策略 e_s，写作策略执行（$e_s \leftarrow e_i$）；执行结果可以是（$\text{succ}\ e$），其中 e 是成功重写后的表达式，也可以简单地是失败。为了支持选择组合子，两个结果之间的非确定性选择（$e_m \diamond e_n$）也包含在语法中，但在大多数情况下，程序员不需要直接使用该运算符。最后，项语法为策略、策略组合子和执行结果的多态抽象提供了 let 绑定（$\text{let}\ x = e_f \text{in}\ e$）。

项的非形式化表示。 下面展示的是一个使用形式语法书写的项。它包含一个策略组合子定义（`swapOps` 来自示例4）和策略应用。

$$
\begin{array}{l}
\text{let}\ \ swapOps = \text{st}\ \ s \Rightarrow \text{seq} \Leftarrow (\text{重写规则}\ \ Op(op,(m,n)) \to Op(op,(n,m))) \Leftarrow s\text{in}\ (\\
swapOps \Leftarrow (\text{重写规则}\ \ Op(Mul(),(1\ ()\ \ v)) \to v)) \leftarrow \text{成功}\ (Op(Mul(),(2\ (),1\ ())))
\end{array}
$$

通过在重写规则定义中对 Typed ELEVATE 项（例如数字和二元运算符）应用语法糖，我们得到一个使用第2节中所用简化语法的项。

```
let swapOps = st s => (rule m op n -> n op m) ; s in swapOps (rule 1 * v -> v) (2 * 1)
```

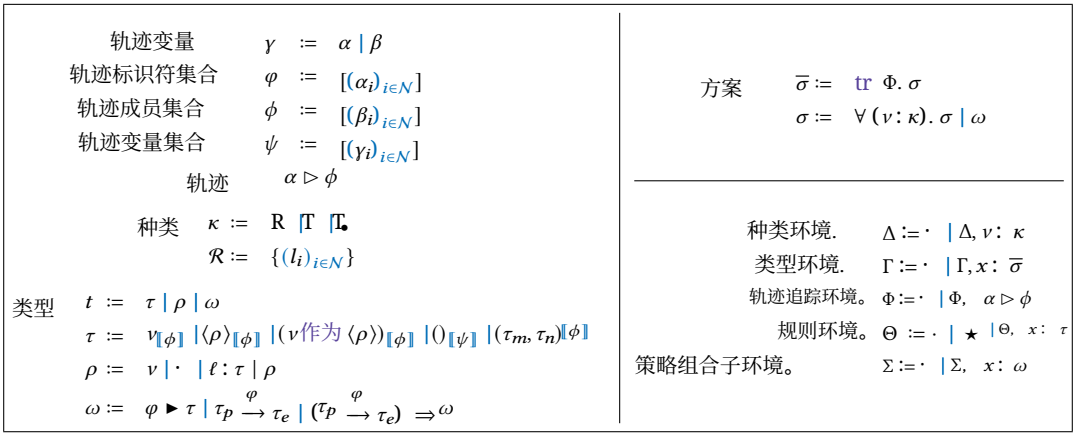

图2 。轨迹语法、种类和类型（左）以及种类、轨迹追踪和类型环境（右） t).

3.2 类型语法

图 2 展示了 TypedELEVATE 中轨迹、种类和类型的语法，以及种类、轨迹和类型环境的语法。

轨迹。 我们类型系统中一个新颖且重要的部分是轨迹追踪。借助轨迹，我们能够观察类型层面的每一种可能的重写方式。轨迹由轨迹变量（γ）构成，这些变量可以划分为两个子集：追踪标识符（α）和跟踪成员（β）。在示例中，我们针对不同种类的轨迹变量采用不同的命名约定，并在形式化表示中使用不同的元变量，以便于表达和理解。不过请注意，轨迹变量在语法上不必存在任何差异，因为跟踪环境能够将它们区分开来。由不同组轨迹变量形成的（由不同元素组成的）集合也用不同的元变量表示： φ 表示追踪标识符集合， ϕ 表示轨迹成员集合， ψ 表示追踪变量集合。最后，我们能够定义轨迹。每条记作 $\alpha \triangleright \phi$ 的轨迹都由追踪标识符 α 标识，而轨迹成员集合 ϕ 包含该轨迹的其余轨迹变量。除了采用常见的集合运算外，为方便起见，我们还使用以下转换规则来书写上述集合。我们以追踪标识符集合的规则为例，同样的规则也适用于另外两种集合。

$$[\alpha, \varphi] = [\alpha, (\alpha_i)_{i\in N}] \quad if \quad \varphi = [(\alpha_i)_{i\in N}] \quad and \quad \alpha \notin \varphi$$
$$[\varphi_a, \varphi_b] = [(\alpha_i)_{i\in N}, (\alpha_j)_{j\in M}] \quad if \quad \varphi_a = [(\alpha_i)_{i\in N}] \quad and \quad \varphi_b = [(\alpha_j)_{j\in M}] \quad and \quad \varphi_a \cap \varphi_b = \emptyset$$

种类和类型。 类型有三种语法类别，共同构成类型宇宙 t 在 Typed ELEVATE 中：可追踪类型（τ）、行类型（ρ）和带跟踪类型（ω）。可追踪类型（τ）的种类为 $\mathcal{T}_{\bullet}$，包括类型变量（ν）、由行构造的变体类型（$\langle\rho\rangle$）、等递归变体类型（ν 其中 $\langle\rho\rangle$） ν 是表示类型自身的变量、单位类型（$()$）以及对类型（$(\tau_m,\ \tau_n)$）。在 τ的定义中，每种情况都可以选择性地在下标位置附加一个追踪变量集合（或跟踪成员集合）。这就是语法层面的轨迹追踪工作方式：轨迹并非与底层类型深度绑定，而是覆盖在底层类型之上的另一层类型级信息。

以下转换规则用于折叠和展开等递归变体类型，其中**擦除** 是一个擦除所有轨迹的函数，稍后将在附录中的图 14 中定义。

$$\nu \ \mathrm{as}\ \langle\rho\rangle \simeq \langle\rho\rangle[\nu \mapsto \mathbf{erase}(\nu \ \mathrm{as}\ \langle\rho\rangle)]$$

由于折叠类型中的所有轨迹都会在展开过程中被擦除，因此在此转换前后轨迹保持不变。如果后续定义中缺少等递归变体类型的情况，则表示所有等递归变体类型都已预先展开。

行类型（ρ）是由标签-类型对组成的序列 $\ell : \tau$ 以行变量（ν）或空行（$\cdot$）结尾，其中标签-项对的顺序无关紧要，且所有标签均不重复。所有可能的标签集合与所有可能的变量名集合互不相交。行通过其种类与普通类型区分开来，这些种类称为行种类（$\mathcal{R}$），它们是行中不存在的标签集合。对于行变量，其种类描述了行变量实例化后不得包含的 [Blume 等人 2006年; Hillerström 和 Lindley 2016]。尽管行类型用于构造可追踪类型，但行类型自身并不可追踪。这表明，我们可以在行的不同表示形式之间切换，甚至可以在结构类型的不同规范之间切换，并为其添加追踪机制。

带跟踪类型（ω）包含策略结果类型（$\varphi \blacktriangleright \tau$）、策略类型（$\tau_p \xrightarrow{\varphi} \tau_e$）和策略组合子类型（$(\tau_p \xrightarrow{\varphi} \tau_e) \Rightarrow \omega$）。带跟踪类型和可追踪类型共同构成 T 种类，它是 $\mathbf{T}_\bullet$ 的超种类。顾名思义，带跟踪类型的每种情况都携带一个或多个追踪标识符集合 φ，用于标识相应可追踪类型中的轨迹。策略结果类型 $\varphi \blacktriangleright \tau$ 是策略执行结果的类型，其结果可以是成功 e 或失败。在成功 e 情况下，τ 中的 $\varphi \blacktriangleright \tau$ 给出 e 的类型。因此，策略结果类型可以看作 Haskell 中 `Maybe a` 类型的类比。策略类型 $\tau_p \xrightarrow{\varphi} \tau_e$ 是策略的类型；策略可以是单个重写规则，也可以是由较小策略组合而成的大型策略。对于单个重写规则，其类型可以理解为“将类型为 τ_p 的项重写为类型为 τ_e 的项的规则”；对于更一般的策略，τ_p 和 τ_e 中的轨迹可能更加复杂，并且每条轨迹都标记一个

可能的重写方式。策略组合子类型 $(\tau_p \xrightarrow{\varphi} \tau_e) \Rightarrow \omega$ 是策略组合子的类型，这些策略组合子要么由语言作为原语提供，例如 `seq` 和 `choice`，要么由程序员使用 `st` 关键字定义。这里有一个细微之处：类型为 $\tau_p \xrightarrow{\varphi} \tau_e$ 的策略组合子并不直接将类型为 $(\tau_p \xrightarrow{\varphi} \tau_e) \Rightarrow \omega$ 的策略作为其参数，而是要求其参数是一个类型与 兼容 的策略 $\tau_p \xrightarrow{\varphi} \tau_e$，并通过将迹计算的结果应用于 ω 来返回一个带跟踪类型。关于迹计算的更多内容将在第 3.5节介绍。最后，带跟踪类型中的轨迹标识符集合仅为清晰起见而写出；实际上，可以使用将在第 3.3节介绍的追踪规则计算出这些集合。

类型的非正式表示。 表达式 (`rule 1 * v -> v || rule 2 * w -> w + w`)（来自示例 1）在此作为示例，用于讨论形式类型与第 2节中使用的类型的非正式表示之间的关系。该表达式的形式项语法为：

choice ⇐ (规则 $Op(Mul(), (1\ ()\ v)) \to v$) ⇐ (规则 $Op(Mul(), (2\ ()\ w)) \to Op(Add(), (w, w))$)

其形式类型如下：

$\langle Op : (\langle Mul : ()_{[b,c]} \mid v_a \rangle, (\langle 1 : ()_{[b]} \mid 2 : ()_{[c]} \mid v_c \rangle, (v \text{ as } Op : (Add : () \mid v_e, (v, v)) \mid v_d)^{[b_0,c_0]}) \mid v_b \xrightarrow{[b,c]} Op : (Add : ()_{[c]} \mid v_e,$
$((v \text{ as } Op : (Add : () \mid v_e, (v, v)) \mid v_d)^{[c_0]}, (v \text{ as } Op : (Add : () \mid v_e, (v, v)) \mid v_d)^{[c_0]}) \mid v_d^{[b_0]}$

有两个轨迹标识符，b 和 c，分别对应由 `choice` 组合子组成的两个规则。我们关注由轨迹变量标识的、由轨迹变量追踪的类型 b，它们是 b 和 b_0，并在忽略 c 轨迹的情况下对整个类型执行自底向上遍历，得到类型：

$Op : (Mul : ()_{[b]} \mid v_a, (1 : ()_{[b]} \mid 2 : () \mid v_c, (v \text{ 作为 } Op : (Add : () \mid v_e, (v, v)) \mid v_d)^{[b_0]}) \mid v_b \xrightarrow{[b]} Op : (Add : () \mid v_e,$
$(v \text{ 作为 } Op : (Add : () \mid v_e, (v, v)) \mid v_d \ \ v \text{ 作为 } Op : (Add : () \mid v_e, (v, v)) \mid v_d) \mid v_d^{[b_0]}$

通过应用等递归类型的转换规则（这里之所以存在该规则，是因为进行了类型统一），我们得到一个更简单的类型，其中明显无关的部分以灰色显示。

$Op\colon (Mul\colon ()_{[b]} \mid v_a, (1\colon ()_{[b]} \mid 2\colon () \mid v_c, (v \text{ as } Op\colon (Add\colon () \mid v_e, (v, v)) \mid v_d)^{[b_0]})) \mid v_b \xrightarrow{[b]} (v \text{ as } Op\colon (Add\colon () \mid v_e, (v, v)) \mid v_d)^{[b_0]}$

对于这种类型，我们已经可以看出它与表达式 规则 $Op\,(Mul\,(), (1\,()\ v)) \to v$ 的联系，以及如何借助轨迹变量提取表示单个重写路径的类型。在使用数字、二元运算符和行变量的语法糖时，这种类型的表示会更加清晰：

$\langle\langle 1_{[b]} \mid 2 \mid\rangle\ \langle *_{[b]} \mid\rangle\ (v \text{ as } \langle v\ \langle + \mid\rangle\ v \mid\rangle)_{[b_0]} \mid\rangle \xrightarrow{[b]} (v \text{ as } \langle v\ \langle + \mid\rangle\ v \mid\rangle)_{[b_0]}$

我们还可以在这里看到轨迹成员与模式变量之间的对应关系。现在，我们可以去掉底层结构类型和灰色的无关部分，从而得到一个简化类型 $1 * b_0 \xrightarrow{[b]} b_0$，它仍然清楚地展示了相应规则所执行的重写。对于由 c标识的另一条轨迹，我们也可以进行相同的处理，并得到另一个简化类型$2 * c_0 \xrightarrow{[c]} c_0 + c_0$。将这两个类型放在一起，并用竖线分隔由 b和 c标识的部分，最终便得到第 2节中最初展示的非正式类型：

$$1 * b_0 \mid 2 * c_0 \xrightarrow{[b,c]} b_0 \mid c_0 + c_0$$

类型方案。 Typed ELEVATE 的类型系统还配备了广义类型。类型方案（σ）表示一个全称量化类型，并携带一个跟踪环境 Φ，用于实例化为新鲜轨迹。不带跟踪环境的类型方案记作 σ，并以标准方式归纳定义。有界类型变量的种类在绑定位置指定。文本中可以使用 $(\forall(\alpha_i\colon\ \kappa_i).)_{i\in\mathcal{N}}$语法，在顺序无关紧要时统称一系列量词。

环境。 图 2给出了各种环境的定义。种类环境（Δ）和类型环境（Γ）是标准的：它们可以是空环境（$\cdot$），也可以扩展为包含类型变量及其种类（$v\colon\ \kappa$）或变量及其类型方案（$x\colon\ \sigma$）。跟踪环境（Φ）是轨迹的集合，并为其他环境和类型提供所需的全部跟踪信息。如果环境中的所有轨迹都不包含轨迹成员，则使用以下转换规则。

$$(\alpha_i \rhd [])_{i\in\mathcal{N}} = \varphi \rhd [] \quad if \quad \varphi = [(\alpha_i)_{i\in\mathcal{N}}]$$

规则环境（Θ）是在定义重写规则时使用的类型环境，其中包含从规则左侧提取的变量-类型对。规则的左侧可能没有任何变量，因此，为了清晰地区分规则的内侧和外侧，规则环境可以从空环境（$\cdot$）或无名占位符（$\star$）归纳构造而成；无名占位符使规则环境非空，但不会引入变量。策略组合子环境（Σ）用于定义策略组合子，其中只保存策略类型的变量。顺便一提，用于擦除类型中的轨迹的**擦除** 函数可以很容易地推广到环境上。

策略变量的线性性。 设计这个类型系统的目标，是精确反映策略如何操作重写后的程序，但有些策略定义很难赋予精确的类型。我们来看一个颇具挑战性的策略组合子 $\texttt{st}\ x \Rightarrow \texttt{seq} \Leftarrow\ x \Leftarrow\ x$，它将一个策略 x 与自身进行顺序组合。为了给这个策略组合子赋予类型，我们只知道 x 具有一般的策略类型 $\tau_p \xrightarrow{\varphi} \tau_e$。由于 x 与自身进行顺序组合，它的输出类型就会成为自身的输入类型，因此，它的输出类型 τ_e 必须与输入类型相匹配

$$\frac{\nu : \kappa \in \Delta}{\Delta \vdash \nu : \kappa}\ \textbf{K-Var} \qquad \frac{}{\Delta \vdash () : \mathcal{T}_\bullet}\ \textbf{K-Unit} \qquad \frac{\Delta \vdash \tau_m : \mathcal{T}_\bullet \quad \Delta \vdash \tau_n : \mathcal{T}_\bullet}{\Delta \vdash (\tau_m, \tau_n) : \mathcal{T}_\bullet}\ \textbf{K-Pair}$$

$$\frac{\Delta \vdash \rho : \mathcal{R}}{\Delta \vdash \langle \rho \rangle : \mathcal{T}_\bullet}\ \textbf{K-Variant} \qquad \frac{\Delta, \nu : \mathcal{T}_\bullet \vdash \rho : \mathcal{R}}{\Delta \vdash \nu \ \mathsf{as}\ \langle \rho \rangle : \mathcal{T}_\bullet}\ \textbf{K-Recursive} \qquad \frac{\Delta \vdash \tau : \mathcal{T}_\bullet}{\Delta \vdash \tau : \mathcal{T}}\ \textbf{K-Subkind} \qquad \frac{\Delta \vdash \tau : \mathcal{T}_\bullet}{\Delta \vdash \tau_\psi : \mathcal{T}_\bullet}\ \textbf{K-Traced}$$

$$\frac{\Delta \vdash \rho : \mathcal{R} \quad \Delta \vdash \tau : \mathcal{T}_\bullet \quad \ell \in \mathcal{R} \quad \mathcal{R}' = \mathcal{R} \setminus \ell}{\Delta \vdash (\ell : \tau \mid \rho) : \mathcal{R}'}\ \textbf{K-RowExt} \qquad \frac{}{\Delta \vdash \cdot : \mathcal{R}}\ \textbf{K-EmptyRow}$$

$$\frac{\Delta \vdash \tau_p : \mathcal{T}_\bullet \quad \Delta \vdash \tau_e : \mathcal{T}_\bullet \quad \Phi \vdash \tau_p \dashv \varphi \quad \Phi \vdash \tau_e \dashv \varphi}{\Delta; \Phi \vdash \tau_p \xrightarrow{\varphi} \tau_e : \mathcal{T}}\ \textbf{K-Strategy} \qquad \frac{\Delta \vdash \tau : \mathcal{T}_\bullet \quad \Phi \vdash \tau \dashv \varphi}{\Delta; \Phi \vdash \varphi \blacktriangleright \tau : \mathcal{T}}\ \textbf{K-Result}$$

$$\frac{\Delta; \Phi \vdash \tau_p \xrightarrow{\varphi} \tau_e : \mathcal{T} \quad \Delta; \Phi \vdash \omega : \mathcal{T}}{\Delta; \Phi \vdash (\tau_p \xrightarrow{\varphi} \tau_e) \Rightarrow \omega : \mathcal{T}}\ \textbf{K-Combinator}$$

图 3. 种类规则

τ_p。经过合一， x（以及它与自身的顺序组合）的类型会变成不精确的策略类型 $\tau_q \xrightarrow{\varphi_x} \tau_q$。按照这一过程，我们可以任意扩展顺序组合链，例如，$\mathtt{st}\ x \Rightarrow \mathtt{seq} \Leftarrow (\mathtt{seq} \Leftarrow x \Leftarrow x) \Leftarrow x$，结果都相同，得到不精确的类型。这显然与我们追求精确类型的目标相冲突。为避免这个问题，我们要求策略变量（例如本例中的 x）具有线性性，即只能使用一次。为确保线性性，我们从跟踪环境和策略组合子环境中移除收缩结构规则，从而保证同一个追踪标识符或同一个策略变量不会重复出现。关于策略组合子类型判断的更多细节将在第3.5节介绍。作为避免此问题的一种变通方法，我们只需通过 `let` 将策略绑定到一个名称，就能以安全的方式将策略与自身进行顺序组合，如第3.6节所述。

类型判断。类型判断的形式为 $\Delta; \Gamma; \Phi^{\downarrow}; \Sigma^{\downarrow}; \Theta \vdash e : t \dashv \Phi^{\uparrow}; \Sigma^{\uparrow}$，表示项 e 具有类型 t，并且在一系列输入环境下进行类型判断，其输出为若干环境。本文中的类型等价允许阿尔法重命名。如果输入环境的名称与输出环境发生冲突，则使用 $\downarrow$和 $\uparrow$上标分别区分输入环境和输出环境。对于转门左侧的输入环境， Δ 提供种类信息，并确保 t 以及所有其他环境中的类型（包括那些输出环境中的类型）都是良种类的； Γ 提供由 let绑定的变量的类型信息； $\Phi^{\downarrow}$ 是输入跟踪环境，并确保 $\Sigma^{\downarrow}$ 和 Θ 中的类型都是良好追踪的； $\Sigma^{\downarrow}$ 保存由 st绑定的变量的输入类型； Θ 保存由 rule绑定的变量的类型。对于反转门右侧的输出环境，$\Phi^{\uparrow}$ 是输出跟踪环境，并确保 t 以及 $\Sigma^{\uparrow}$ 中的所有类型都是良好追踪的； $\Sigma^{\uparrow}$ 保存由 st绑定的变量的输出类型。如果某个环境必须为空，则将其从类型判断中省略。例如， $\Delta; \Gamma; \Phi^{\downarrow}; \Sigma^{\downarrow} \vdash e : t \dashv \Phi^{\uparrow}; \Sigma^{\uparrow}$ 等价于 $\Delta; \Gamma; \Phi^{\downarrow}; \Sigma^{\downarrow}; \cdot \vdash e : t \dashv \Phi^{\uparrow}; \Sigma^{\uparrow}$。如果输出环境与其对应的输入环境完全相同，则将输出环境与反转门一并省略。例如， $\Delta; \Gamma; \Phi^{\downarrow}; \Sigma^{\downarrow}; \Theta \vdash e : t$等价于 $\Delta; \Gamma; \Phi^{\downarrow}; \Sigma^{\downarrow}; \Theta \vdash e : t \dashv \Phi^{\downarrow}; \Sigma^{\downarrow}$。

$$\frac{\Phi \vdash \tau_{[\gamma]} \dashv \varphi_h \quad \Phi \vdash ()_\psi \dashv \varphi_t}{\Phi \vdash \tau_{[\gamma,\psi]} \dashv [\varphi_h, \varphi_t]}\ \textbf{Tr-Set} \qquad \frac{\alpha \rhd [\beta, \phi] \in \Phi \quad \Phi \vdash \tau \dashv \varphi_\tau}{\Phi \vdash \tau_{[\beta]} \dashv [\alpha, \varphi_\tau]}\ \textbf{Tr-Member} \qquad \frac{\alpha \rhd \phi \in \Phi}{\Phi \vdash ()_{[\alpha]} \dashv [\alpha]}\ \textbf{Tr-Identifier}$$

$$\frac{}{\Phi \vdash () \dashv []}\ \textbf{Tr-Unit} \qquad \frac{\Phi \vdash \tau_m \dashv \varphi \quad \Phi \vdash \tau_n \dashv \varphi}{\Phi \vdash (\tau_m, \tau_n) \dashv \varphi}\ \textbf{Tr-Pair} \qquad \frac{}{\Phi \vdash \nu \dashv []}\ \textbf{Tr-Variable}$$

$$\frac{\Phi \vdash \tau \dashv \varphi_\tau \quad \Phi \vdash \langle \rho \rangle \dashv \varphi_\rho}{\Phi \vdash \langle \ell : \tau \mid \rho \rangle \dashv [\varphi_\tau, \varphi_\rho]}\ \textbf{Tr-Variant} \qquad \frac{}{\Phi \vdash \langle \cdot \rangle \dashv []}\ \textbf{Tr-EmptyRow}$$

$$\frac{}{\Phi \vdash \langle \nu \rangle \dashv []}\ \textbf{Tr-RowVar} \qquad \frac{\Phi \vdash \langle \rho \rangle \dashv \varphi}{\Phi \vdash \nu \ \mathsf{as}\ \langle \rho \rangle \dashv \varphi}\ \textbf{Tr-Recursive}$$

图 4. 追踪规则

3.3 类型良构性

图 3 展示了用于检查类型良构性的种类规则。种类判定的形式为 $\Delta; \Phi \vdash t : \kappa$，表示类型 t 在种类环境 Δ 和跟踪环境 Φ中是良构的，并且具有种类 κ。当跟踪环境未被使用时，将其从判断中省略。大多数种类规则都很直接。这里的大多数行种类判定规则与 [Hillerström 和 Lindley 2016], 中的规则相似，但 **K-Variant** 规则除外；该规则在我们的类型系统中更加宽松。构造变体类型时，我们不要求行种类为空，而是取消了这一要求，且 **K-Variant** 中的 ρ 的种类可以是任意的。这仍然保证行中不存在重复标签，同时也使我们能够像 [Blume 等人 2006]那样灵活地表达行中某些标签的确定缺失。除此之外，我们的类型系统还有三个特定的种类规则：**K-Strategy**、**K-Result**和**K-Combinator**。它们是带跟踪类型的种类规则，用于检查轨迹和底层类型的良构性。因此，需要一组跟踪规则来定义它们。

图4 展示了用于检查轨迹良构性并计算类型的追踪标识符集合的追踪规则。追踪判断的形式为 $\Phi \vdash \tau \dashv \varphi$，表示类型 τ在跟踪环境 Φ 中具有良好追踪性，并且 τ对应的追踪标识符集合为 φ。最重要的追踪规则是 **Tr-Set**。对于由追踪变量集合 $[\gamma, \psi]$追踪的任意可追踪类型 τ，我们首先检查由单个追踪变量 γ追踪的 τ的良构性，这将根据 γ由**Tr-Member** 或 **Tr-Identifier** 处理，并得到一个追踪标识符集合 φ_h，该集合标识 γ 以及 τ中的追踪变量。由于 τ中的内部轨迹已经过检查，我们使用单位类型替换 τ，并计算 $()_\psi$的追踪标识符集合 φ_t，然后给出最终结果 $[\varphi_h, \varphi_t]$。对于**Tr-Member** 规则，我们使用跟踪成员来追踪模式变量的类型，而模式变量在语法中属于终结符。换句话说，（变量的类型）不可能具有任何内部（追踪）结构（由同一追踪标识符标识）。因此，对于 $\tau_{[\beta]}$，β必须由变量 α 标识，且该变量不同于 τ的追踪标识符集合中的元素。对于**Tr-Identifier** 规则，单位模式和单位类型分别是模式语法和类型语法中唯一的非变量终结符，因此，单位类型是唯一可以由追踪标识符直接追踪的类型。此外，还有一些追踪规则与模式的结构（以及可由模式匹配的项）密切相关。对模式的两个元素必须同时存在，因此**Tr-Pair** 要求对类型的两个元素具有相同的追踪标识符集合。变体模式中只能出现一个标签，因此**Tr-Variant** 规则要求与每个标签 ℓ 关联的类型的追踪标识符集合必须不同于该行的其余部分。其余追踪规则都很直接。

$$\frac{\textbf{Fresh}\ \alpha \quad \Delta \vDash_\alpha p : \tau_p \dashv \Phi;\ \Theta \qquad \Delta;\ \Gamma;\ \Phi;\ \Theta \vdash e : \tau_e}{\Delta;\ \Gamma;\ \Phi_\Sigma;\ \Sigma \vdash \text{规则}\ p \to e : \tau_p \xrightarrow{[\alpha]} \tau_e \dashv \Phi}\ \textbf{T-R-Lam}$$

$$\frac{\textbf{fresh}\ \beta \quad \Phi = \alpha \rhd [\beta] \qquad \Theta = \star\ x : \tau_{[\beta]}}{\Delta \vDash_\alpha x : \tau_{[\beta]} \dashv \Phi;\ \Theta}\ \textbf{P-Var} \qquad \frac{\Phi = \alpha \rhd [\,]}{\Delta \vDash_\alpha () : ()_{[\alpha]} \dashv \Phi;\ \star}\ \textbf{P-Unit}$$

$$\frac{\Phi = \alpha \rhd [\phi_m, \phi_n] \quad \Theta_m \cap \Theta_n = \emptyset \qquad \Delta \vDash_\alpha p_m : \tau_m \dashv (\alpha \rhd \phi_m); \Theta_m \quad \Delta \vDash_\alpha p_n : \tau_n \dashv (\alpha \rhd \phi_n); \Theta_n}{\Delta \vDash_\alpha (p_m, p_n) : (\tau_m, \tau_n) \dashv \Phi;\ \Theta_m, \Theta_n}\ \textbf{P-Pair}$$

$$\frac{\Delta \vDash_\alpha p : \tau \dashv \Phi; \Theta}{\Delta \vDash_\alpha \ell\ p : \ell : \tau \mid \rho \dashv \Phi;\ \Theta}\ \textbf{P-Label}$$

$$\frac{x : \tau \in \Theta}{\Delta;\ \Gamma; \Phi; \Theta \vdash x : \tau}\ \textbf{T-R-变量} \qquad \frac{\Phi = \alpha \rhd \phi \quad \Theta \neq \cdot}{\Delta;\ \Gamma;\ \Phi;\ \Theta \vdash () : ()_{[\alpha]}}\ \textbf{T-R-单位}$$

$$\frac{\Theta \neq \cdot \quad \Delta;\ \Gamma;\ \Phi;\ \Theta \vdash e_m : \tau_m \qquad \Delta;\ \Gamma;\ \Phi;\ \Theta \vdash e_n : \tau_n}{\Delta;\ \Gamma;\ \Phi;\ \Theta \vdash (e_m,\ e_n) : (\tau_m,\ \tau_n)}\ \textbf{T-R-成对} \qquad \frac{\Theta \neq \cdot \quad \Delta;\ \Gamma;\ \Phi;\ \Theta \vdash e : \tau}{\Delta;\ \Gamma;\ \Phi;\ \Theta \vdash \ell\ e : \ell : \tau \mid \rho}\ \textbf{T-R-标签}$$

图5。重写规则的类型规则

3.4 重写规则的类型规则规则复数

图 5展示了重写规则的类型规则。

重写规则定义的类型规则。我们从用于对重写规则定义进行类型检查的核心规则 **T-R-Lam** 开始。为该规则分配一个新的追踪标识符 α。根据模式 p （即重写规则的左侧）计算规则环境 Θ （以及其跟踪环境 Φ），然后使用 Θ 对重写规则的右侧进行类型检查 e。

左侧的模式类型规则。模式的类型判断形式为 $\Delta \vDash_\alpha p : \tau \dashv \Phi; \Theta$，表示给定种类环境 Δ 和分配的追踪标识符 α后，模式p 具有类型 τ，该判断会生成跟踪环境 Φ 和规则环境 Θ。跟踪环境 Φ 中唯一的轨迹由 α标识，而 Φ 确保 τ 以及 Θ 中的所有类型都被良好追踪。至于模式类型规则，**P-Var** 为模式变量 x的底层类型 τ 分配一个新的跟踪成员 β，然后将 $x : \tau_{[\beta]}$ 扩展到一个非空规则环境中作为其输出；**P-Unit** 则直接使用分配的追踪标识符，并生成一个非空规则环境作为输出；**P-Pair** 通过检查两个元素各自的输出环境是否互不相交，确保模式是线性的；**P-Label** 给出一个变体类型，其中包含匹配的标签 ℓ 及其对应的类型 τ，并根据追踪规则，$\rho = \textbf{擦除}(\rho)$。

右侧类型规则。 图 5 中剩余的类型规则用于为重写规则的右侧进行类型检查，且十分直观。条件 $\Theta \neq \cdot$ 用于禁止在重写规则体中使用其他语言构造。值得注意的是，重写规则定义两侧的类型规则非常相似，除了对模式的线性限制外，它们还共享相同的语法。这是因为在组合过程中，重写规则的右侧可以与另一条重写规则的左侧进行 match，而这种相似性使它们能够相互比较。

3.5 组合器的类型规则 strategyCombinators

图 6 展示了策略组合子的类型规则，包括基本策略组合子的类型规则、策略组合子定义和策略组合子应用的类型规则。

$$\frac{\textbf{fresh}\ \alpha,\beta_i,\beta_j,\beta_k \qquad \Phi = \alpha \rhd [\beta_i,\beta_j,\beta_k]}{\Delta;\Gamma;\Phi_\Sigma;\Sigma \vdash \texttt{seq} : (\tau^i_{[\beta_i]} \xrightarrow{[\alpha]} \tau^j_{[\beta_j]}) \Rightarrow (\tau^j_{[\beta_j]} \xrightarrow{[\alpha]} \tau^k_{[\beta_k]}) \Rightarrow (\tau^i_{[\beta_i]} \xrightarrow{[\alpha]} \tau^k_{[\beta_k]}) \dashv \Phi}\ \textbf{T-Seq}$$

$$\frac{\textbf{fresh}\ \alpha_m,\alpha_j,\beta_m,\beta_n,\beta_j,\beta_k \qquad \Phi = \alpha_m \rhd [\beta_m,\beta_n],\alpha_j \rhd [\beta_j,\beta_k]}{\Delta;\Gamma;\Phi_\Sigma;\Sigma \vdash \texttt{choice} : (\tau^p_{[\beta_m]} \xrightarrow{[\alpha_m]} \tau^e_{[\beta_n]}) \Rightarrow (\tau^p_{[\beta_j]} \xrightarrow{[\alpha_j]} \tau^e_{[\beta_k]}) \Rightarrow (\tau^p_{[\beta_m,\beta_j]} \xrightarrow{[\alpha_m,\alpha_j]} \tau^e_{[\beta_n,\beta_k]}) \dashv \Phi}\ \textbf{T-Choice}$$

$$\frac{\textbf{fresh}\ \alpha,\beta_m,\beta_n \qquad \Phi^{\downarrow}_x = \alpha \rhd [\beta_m,\beta_n] \qquad \omega_x = \textbf{erase}(\tau_m)_{[\beta_m]} \xrightarrow{[\alpha]} \textbf{erase}(\tau_n)_{[\beta_n]} \qquad \Delta;\Gamma;\Phi^{\downarrow},\Phi^{\downarrow}_x;\Sigma^{\downarrow},x:\omega_x \vdash e_b : \omega_b \dashv \Phi^{\uparrow};\Sigma^{\uparrow},x:\tau_m \xrightarrow{\varphi} \tau_n}{\Delta;\Gamma;\Phi^{\downarrow};\Sigma^{\downarrow} \vdash \texttt{st}\ x \Rightarrow e_b : (\tau_m \xrightarrow{\varphi} \tau_n) \Rightarrow \omega_b \dashv \Phi^{\uparrow};\Sigma^{\uparrow}}\ \textbf{T-S-Lam}$$

$$\frac{\omega = \tau_m \xrightarrow{[\alpha]} \tau_n \quad \alpha \rhd \phi \in \Phi^{\downarrow} \quad x:\omega \in \Sigma^{\downarrow}}{\Delta;\Gamma;\Phi^{\downarrow};\Sigma^{\downarrow} \vdash x : \omega \dashv \alpha \rhd \phi; x:\omega}\ \textbf{T-S-Var}$$

$$\frac{\Delta;\Gamma;\Phi^{\downarrow};\Sigma^{\downarrow} \vdash e_f : (\tau_m \xrightarrow{\varphi_f} \tau_n) \Rightarrow \omega_f \dashv \Phi^{\uparrow}_f;\Sigma^{\uparrow}_f \qquad \Delta;\Gamma;\Phi^{\downarrow};\Sigma^{\downarrow} \vdash e : \tau_j \xrightarrow{\varphi_e} \tau_k \dashv \Phi^{\uparrow}_e;\Sigma^{\uparrow}_e \qquad \textbf{erase}(\tau_m \xrightarrow{\varphi_f} \tau_n) = \textbf{erase}(\tau_j \xrightarrow{\varphi_e} \tau_k) \qquad (\Phi^{\uparrow};\Sigma^{\uparrow};\omega) = \textbf{compTrace}(\Phi^{\uparrow}_f;\Sigma^{\uparrow}_f;\tau_m \xrightarrow{\varphi_f} \tau_n;\omega_f;\ \Phi^{\uparrow}_e;\Sigma^{\uparrow}_e;\tau_j \xrightarrow{\varphi_e} \tau_k)}{\Delta;\Gamma;\Phi^{\downarrow};\Sigma^{\downarrow} \vdash e_f \Leftarrow e : \omega \dashv \Phi^{\uparrow};\Sigma^{\uparrow}}\ \textbf{T-S-App}$$

图 6. 策略组合子的类型规则

基本组合子的类型规则。 基本策略组合子 seq 和choice的类型规则很直接。对于 seq，它按顺序组合两个策略，因此要求两个策略具有相同的轨迹，且第一个策略的输出与第二个策略的输入相匹配。对于 choice，它在两个策略之间进行非确定性选择，因此要求它们具有不同的轨迹，但底层结构类型必须相同，并且两条轨迹会合并到组合的结果类型中。

自定义策略组合子的类型规则。 为了构造策略组合子，我们引入了 st 关键字，相应的类型规则是 **T-S-Lam**。根据该类型系统的类型规则，我们区分输入和输出环境，并且只有输出环境包含最终的跟踪信息（在完成所有轨迹计算之后）。因此，对于表达式 st x ~~$\Rightarrow e_b$~~，我们必须从输出策略组合子环境中提取 x 的类型，以确保该类型准确反映 x 在 e_b中的使用情况。因此，这要求为 x 在输入策略组合子环境中的类型进行精心设计。在 **T-S-Lam** 规则中，x 被赋予一个输入类型，该类型在策略类型的左侧和右侧分别包含一个新的追踪标识符和两个新的轨迹成员。此外， x 的底层类型在输入和输出环境中始终保持不变，因此可以直接使用，无需任何更改。对于 x的这一输入类型，可以直观地理解为：该类型不对轨迹追踪作任何假设，并且具有最一般的轨迹，因此当 x 以线性方式使用并参与轨迹计算时，输出类型将从上下文中捕获跟踪信息。从形式上看，**T-S-Lam** 规则的正确性由引理 3.6 得以证明，该引理表明策略组合子的 β 归约保持类型不变。

规则 **T-S-Var** 用于策略组合子定义主体中绑定变量的使用。它从输入策略组合子环境中获取变量 x 的类型，并基于线性性将输出策略组合子环境设为仅包含 x 的单元素环境。

策略组合子应用的类型规则。 **T-S-App** 规则是该类型系统中最复杂的类型规则之一。它适用于形如 $e_f \Leftarrow\ e$的策略组合子应用，

```
1  compTrace(Φm; Σm; ωm; ωc; Φn; Σn; ωn) =
2    φm =traceIds(Φm;ωm)
3    φn =traceIds(Φn;ωn)
4    (Φ⁻m;Σ⁻m;ω⁻c) = 删除 φm 在 (Φm;Σm;ωc)
5    (Φ⁻n;Σ⁻n;ωn⁻) = 删除 φn于 (Φn;Σn;ωn)
6    (Φr; Σr;ωr) := ((Φ⁻m Φ⁻n); (Σ⁻m Σ⁻n); ω⁻c)
7    对 αm 于 φm
8      对 αn 于 φn
9        (αm ▷ ϕm; Σ′m;ω′m) = 选择 [αm] 于 (Φm; Σm;ωm)
10       (αm ▷ ϕm; ·; ω′c) = 在 (Φm; ·; ωc)中选择 [αm]
11       (αm ▷ ϕn; Σ′n; ω′n) =(在 (Φn; Σn; ωn)) 中选择 [αn] [αn ↦ αm]
12       S‾ =unifyTrace(αm ▷ [ϕm, ϕn];ω′m;ω′n)
13       如果S‾ ≠failTrace
14         (Φi; Σi; ωi) =Fresh (αm ▷ [ϕm, ϕn]; (Σ′m Σ′n) [S‾];ω′c [S‾])
15         (Φr; Σr;ωr) := 添加 (Φi; Σi;ωi) 至 (Φr; Σr;ωr)
16   返回 (Φr; Σr; ωr)
```

清单 1。**compTrace** 的定义

即将策略 e 应用于策略组合子 e_f。该规则调用**compTrace**，通过枚举将参数 e_f所需类型中的每一对轨迹与作为参数传入的 e 的类型进行连接的所有可能方式，来计算带迹返回类型和环境。随后，它将计算结果应用于 e_f的主体类型以及策略组合子环境中的类型。形式上，**T-S-App** 规定，对于形如 $e_f \Leftarrow e$ 的策略组合子应用，其中e_f的类型为$(\tau_m \xrightarrow{\varphi_f} \tau_n) \Rightarrow \omega_f$，而 e的类型为 $\tau_j \xrightarrow{\varphi_e} \tau_k$，我们首先检查参数 e_f和参数 e 的底层结构类型是否相同，也就是检查它们在轨迹擦除后的类型是否相等。然后，我们使用**compTrace** 函数计算 $e_f \Leftarrow e$的类型和输出环境。由于轨迹的存在，这一过程比普通函数应用的类型检查复杂得多。

迹计算 为了理解 compTrace 的工作原理，我们来看看它在清单 1 中的定义。该定义接受七个输入，可分为两组：前四个参数（Φ_m; Σ_m; ω_m; ω_c）是来自策略组合子 e_f 的输出环境和带跟踪类型，其中 ω_m 是参数类型， ω_c 是策略组合子主体的类型；另外三个参数（Φ_n; Σ_n;ω_n）是来自参数 e 的输出环境和带跟踪类型。由于线性性， ω_m 不能与 ω_n 共享任何追踪标识符。迹计算在 ω_m 和 ω_n 上执行，然后将计算结果应用于 Σ_m、 Σ_n 和 ω_c，以合成得到的环境和带跟踪类型（Φ_r; Σ_r; ω_r）。我们用 (:=) 表示可变变量赋值，用 (=) 表示常量初始化。在使用带跟踪类型 ω 进行计算时，必须将其与对应的轨迹追踪及策略组合子环境 Φ 和 Σ 一并处理。因此，我们将它们合称为带迹三元组（Φ; Σ;ω），其中 Σ 和 ω 在 Φ 中是良好追踪的。

compTrace 的基本思想是枚举连接 ω_m和 ω_n 中的轨迹并将结果应用于 ω_c 和策略组合子环境的所有可能方式。我们首先在第 2–3 行中，使用 traceIds 函数从 ω_m 和 ω_n 的带迹三元组中提取追踪标识符集合 φ_m 和 φ_n；该函数的形式化定义见附录中的图 17。提取出的追踪标识符集合包含后续计算中涉及的轨迹。在第 4–6 行中，我们从 ω_c 的带迹三元组和 ω_n 的带迹三元组中移除所涉及的轨迹，从而得到只包含计算中未涉及轨迹的带迹三元组。我们将这些三元组合并为 $(\Phi_r; \Sigma_r; \omega_r)$，它同时也是后续循环中累加器的初始值。第 7–8 行中的两个嵌套循环枚举来自 φ_m 和φ_n的所有追踪标识符对 (α_m, α_n)。在循环体中，我们从输入类型的带迹三元组中选择由 α_m 和 α_n 追踪的带迹三元组（第 9–11 行），以获得单迹切片。

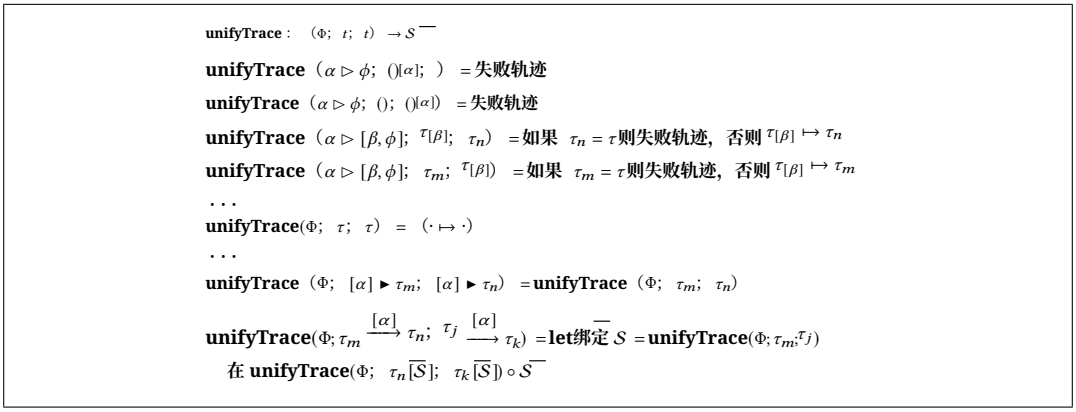

图 7. 轨迹合一的定义

我们在第 11 行将 α_n 重命名为 α_m，以确保所有轨迹都暂时采用统一的标识，并为后续计算做好准备。Select 函数从跟踪环境中选择轨迹，并从其返回的带迹三元组中移除所有未选择的轨迹。其形式化定义见附录中的图 18。Select 是此前使用的 Delete 运算符的对偶。在第 12 行中，我们将带迹类型 ω'_m和 $\omega'n$进行合一。下面将更深入地讨论轨迹合一。如果合一在第 13 行成功并得到结果 S，则存在一种连接由 α_m 和 α_n标识的轨迹的方式，这意味着该策略存在一条可能的执行路径。随后，在第 14 行中，我们将 S 应用于一个包含策略组合子环境 (Σ'_m, Σ'_n) 的带迹三元组；由于线性性，这些环境可以安全地组合在一起，同时还包含 e_f的单迹返回类型（$\omega'c$）。在第 15 行将其添加到累加器之前，我们对带迹三元组进行 Fresh 重命名，以确保追踪标识符之间保持区分。最后，累加器作为 compTrace 的结果返回（第 16 行）。Fresh 和 Add 的形式化定义见附录中的图 19 和图 20。

轨迹合一。与普通类型统一类似，对于带跟踪类型 ω_m 和 ω_n，轨迹合一会生成一个轨迹替换 S，使得 $\omega_m[S] = \omega_n[S]$。轨迹替换是映射的集合，但在普通替换中的映射将变量替换为项时，轨迹替换中的每个映射则将由跟踪成员跟踪的可追踪类型替换为另一个可追踪类型，即如果 $\tau_{[\beta]} \in dom(S)$，则 $\tau_{[\beta]}[S] = S(\tau_{[\beta]})$。实际上，在 $dom(S)$ 中查找 $\tau_{[\beta]}$ 时无需检查 τ，因为 β已经是相应映射的唯一键（见引理 3.1）。此外，在此类型系统中，轨迹替换始终由 unifyTrace 生成，而 unifyTrace 只能对底层类型相同的类型执行，因此我们有 $\tau = \text{erase}(S(\tau_{[\beta]}))$。换言之，轨迹替换不会影响底层类型。轨迹替换自然地适配了普通替换的操作，但为了简洁地进行形式化，我们使用底值 failTrace 扩展轨迹替换，以表示 unifyTrace 的失败，并且任意轨迹替换与 failTrace 的组合都会得到 failTrace。

函数 **unifyTrace** 的定义见图 7 ，图中着重展示了最重要的部分；完整定义见附录中的图 16。该定义直观明了，与标准的类型统一定义非常接近。 **failTrace** 会在尝试将带跟踪类型与其未跟踪对应类型统一时返回，否则会返回轨迹替换。值得注意的是， **unifyTrace** 未针对策略组合子类型定义，也从未应用于策略组合子类型。引理 3.2 表明， **unifyTrace** 是一个定义正确的轨迹合一。

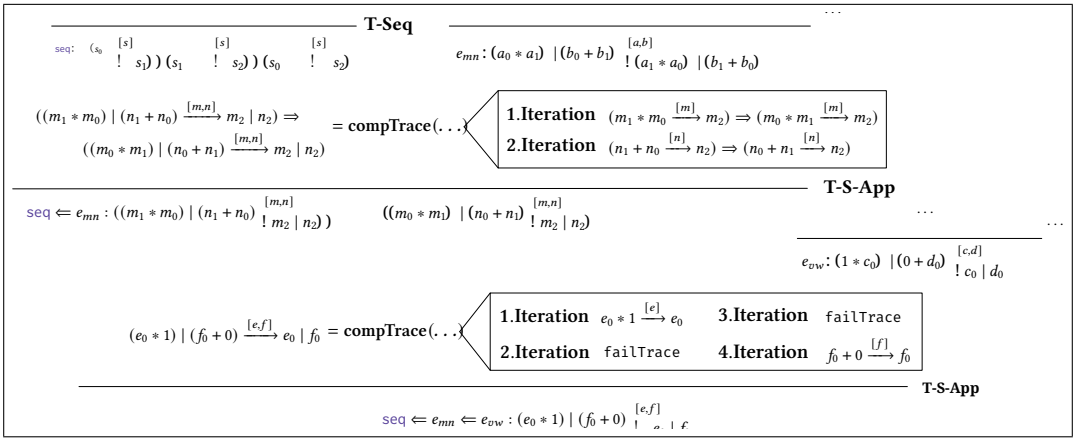

图 8. 示例 3 的简化类型推导

引理 3.1（唯一的底层类型）。 如果 $\Delta \vdash e_m : \omega \dashv \Phi$，对于每个 $\alpha \rhd \phi \in \Phi$，都存在$(\alpha \rhd \phi; \cdot; \omega_m)$ = **选择** $[\alpha]$ **属于** $(\Phi; \cdot; \omega)$，并且对于每个 $\beta \in \phi$，它总是会在 ω_m 中跟踪一个唯一的 τ。

证明。对 $\Delta \vdash e_m$ 的类型推导进行归纳： $\omega \dashv \Phi$。 □

引理 3.2（合一）。如果 **unifyTrace**$(\Phi; \omega_1; \omega_2) = \mathrm{S} \neq$ **failTrace**，则 $\omega_1[\mathrm{S}] = \omega_2[\mathrm{S}]$。

证明。 对 ω_1 进行归纳，并根据 **unifyTrace** 的定义以及引理 3.1。 □

类型策略示例。 图 8 展示了一个简化的类型推导，以说明示例 3 中顺序组合的类型推导过程。在图中，我们将 e_{mn} 和 e_{vw} 记作两个组合的策略，并使用简化类型以便清晰呈现。

要为表达式 seq $\Leftarrow$ $e_{mn} \Leftarrow e_{vw}$推导类型，我们分别为 seq $\Leftarrow$ e_{mn} 和 e_{vw} 推导类型，然后根据它们的类型计算所得的带跟踪类型。我们仔细看看图左上方 seq $\Leftarrow$ e_{mn} 的类型推导。这里，我们在最顶端为 seq 和 e_{mn} 推导类型，并将它们的类型作为 **compTrace** 函数的输入。在此例中，我们进行两次迭代：将 seq (*s*) 的一条轨迹与 e_{mn} (*a* 和 *b*) 的两条轨迹相连接。每次迭代都会进行轨迹合一，在此得到两个带跟踪类型，并将它们组合成策略应用所得的带跟踪类型。对于图下半部分的第二个策略应用，我们计算两个各自包含两条轨迹的策略的轨迹，因此 **compTrace** 会执行 4(= 2 × 2) 次迭代，枚举将第一个类型中的一条轨迹与第二个类型中的一条轨迹相连接的所有可能方式。这里，并非所有迭代中的轨迹合一都能成功，因为并非所有轨迹都能连接。两个成功合一的带跟踪类型将组合起来，得到整体策略应用的最终带跟踪类型。

3.6 策略执行和let绑定的类型规则

策略执行的类型规则。 能够定义重写规则并使用策略组合子将其组合起来后，我们仍然需要执行这些策略并获得结果。图 9展示了策略执行与结果的类型规则。执行策略会产生两种可能的结果：失败 或 成功 e，并且执行过程应当保持类型。因此，类型规则 **T-Fail** 允许 失败 结果具有任意的底层类型和任意的轨迹，因为对于返回 失败的策略执行，我们一无所知。值得注意的是，输出跟踪环境不包含任何跟踪成员，因为闭项中不可能存在模式变量。实际上，策略结果类型中的所有表达式相对于规则环境和策略组合子环境都是闭的，因为在实践中我们总是将策略应用于

策略语言中的闭项抽象语法树。 在规则 **T-Succ**中，如果 成功 e 是策略的执行结果，那么表达式 e 必定由重写的右侧构造而成，因此必须存在一个关于 e 的轨迹，并且 e 的类型与重写规则的右侧相同。类似地，为了保持类型，我们也允许添加任意轨迹。**T-Alt** 规则用于非确定性选择运算符 $\diamond$，该运算符从两个操作数中选择一个。直观地说，应当要求两个操作数具有相同的类型。最后，相对复杂的规则 **T-Exec** 用于策略执行。该类型规则的结构和机制与用于策略组合子应用的 **T-S-App** 规则几乎相同，并且同样使用 **compTrace** 函数来计算执行结果中的轨迹。和之前一样，为了保持类型，可以向最终类型添加任意轨迹。

Let 绑定的类型规则。 最后的类型规则如图10所示，其中包括 `let`绑定、`let`绑定变量的使用，以及类型变量实例化规则。所有这些规则都很直接。考虑到轨迹的抽象与底层类型变量在语法上彼此分离，这些类型规则中对底层类型的操作也都是标准的。由于我们的目标是让轨迹精确反映策略的内部重写路径，因此轨迹的抽象与实例化应尽可能保留跟踪信息，同时避免违反跟踪环境的子结构规则。因此，在 **T-Let** 规则中，输出跟踪环境 $\Phi_f$of e_f由类型方案中的 `tr` 关键字直接绑定；在 **T-Var** 规则中，绑定的跟踪环境会被重新命名为全新的名称（带跟踪类型中的轨迹也一并如此）。根据 α 等价，跟踪信息不会改变，

$$\frac{}{\Delta;\Gamma \vdash \texttt{fail} : \varphi \blacktriangleright \tau \dashv \varphi \rhd [\,]}\ \textbf{T-Fail}$$

$$\frac{\begin{array}{c}\textbf{fresh}\ \alpha \quad \Delta;\Gamma;\alpha \rhd [\,];\star \vdash e : \tau_e \qquad \textbf{erase}(\tau_a) = \textbf{erase}(\tau_e) \\ (\varphi_s \rhd [\,];\cdot;\varphi_s \blacktriangleright \tau_s) = \textbf{Add}\ (\varphi_a \rhd [\,];\cdot;\varphi_a \blacktriangleright \tau_a)\ \textbf{to}\ (\alpha \rhd [\,];\cdot;[\alpha] \blacktriangleright \tau_e)\end{array}}{\Delta;\Gamma \vdash \texttt{succ}\ e : \varphi_s \blacktriangleright \tau_s \dashv \varphi_s \rhd [\,]}\ \textbf{T-Succ}$$

$$\frac{\Delta;\Gamma \vdash e_a : \varphi \blacktriangleright \tau \dashv \varphi \rhd [\,] \quad \Delta;\Gamma \vdash e_b : \varphi \blacktriangleright \tau \dashv \varphi \rhd [\,]}{\Delta;\Gamma \vdash e_a \diamond e_b : \varphi \blacktriangleright \tau \dashv \varphi \rhd [\,]}\ \textbf{T-Alt}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_f : \tau_p \xrightarrow{\varphi_f} \tau_e \dashv \Phi_f \quad \Delta;\Gamma \vdash e_i : \varphi_i \blacktriangleright \tau_i \dashv \varphi_i \rhd [\,] \qquad \textbf{erase}(\tau_p) = \textbf{erase}(\tau_i) \quad \textbf{erase}(\tau_a) = \textbf{erase}(\tau_e) \\ (\varphi_r \rhd [\,];\cdot;\varphi_r \blacktriangleright \tau_r) = \textbf{Add}\ (\varphi_a \rhd [\,];\cdot;\varphi_a \blacktriangleright \tau_a)\ \textbf{to compTrace}(\Phi_f;\cdot;\varphi_f \blacktriangleright \tau_p;\varphi_f \blacktriangleright \tau_e;\varphi_i \rhd [\,];\cdot;\varphi_i \blacktriangleright \tau_i)\end{array}}{\Delta;\Gamma \vdash e_f \leftarrow e_i : \varphi_r \blacktriangleright \tau_r \dashv \varphi_r \rhd [\,]}\ \textbf{T-Exec}$$

图 9。策略执行与结果的类型规则

$$\frac{\Delta, (\alpha_i : \kappa_i)_{i\in\mathcal{N}};\Gamma \vdash e_f : \omega_f \dashv \Phi_f \qquad \Phi_f \neq \cdot \qquad \Delta;\Gamma, x : \texttt{tr}\ \Phi_f.(\forall\ (\alpha_i : \kappa_i).)_{i\in\mathcal{N}}\ \omega_f \vdash e : \omega_e \dashv \Phi_e}{\Delta;\Gamma \vdash \texttt{let}\ x = e_f\ \texttt{in}\ e : \omega_e \dashv \Phi_e}\ \textbf{T-Let}$$

$$\frac{x : \texttt{tr}\ \Phi_x.\ \sigma_x \in \Gamma \qquad \Delta \vdash \sigma_x\ \preceq\ \omega_x \quad (\Phi;\cdot;\omega) = \textbf{Fresh}\ (\Phi_x;\cdot;\omega_x)}{\Delta;\Gamma;\Phi_\Sigma;\Sigma \vdash x : \omega \dashv \Phi}\ \textbf{T-Var}$$

$$\frac{\Delta \vdash \rho : \mathcal{R}' \quad \mathcal{R} \subseteq \mathcal{R}' \quad \rho = \textbf{erase}(\rho) \quad \Delta \vdash \sigma[\nu \mapsto \rho]\ \preceq\ \omega}{\Delta \vdash \forall\ (\nu : \mathcal{R}).\ \sigma\ \preceq\ \omega}\ \textbf{I-Row}$$

$$\frac{\Delta \vdash \tau : \mathcal{T}_\bullet \quad \tau = \textbf{erase}(\tau) \quad \Delta \vdash \sigma[\nu \mapsto \tau]\ \preceq\ \omega}{\Delta \vdash \forall\ (\nu : \mathcal{T}_\bullet).\ \sigma\ \preceq\ \omega}\ \textbf{I-Type}$$

图 10。let 绑定的类型规则

$$
\begin{array}{l}
\text{组合子值}\quad cv ::= \mathsf{seq} \mid \mathsf{choice} \mid cv \Leftarrow v\text{值} \quad v ::= \quad cv \mid \mathsf{st}\ x \Rightarrow e_b \mid \text{规则} \\
p \to e \mid \ell\, v \mid () \mid (v_m, v_n) \mid \text{成功}\ v \mid \text{失败}\ \text{求值上下文} \quad E ::= \quad [\,] \mid E \Leftarrow e \mid v \Leftarrow \\
E \mid E \leftarrow e \mid v \leftarrow E \mid \ell\, E \mid (E, e) \mid (v, E) \mid \mathsf{let}\ x = E \mathsf{in}\ e \mid \text{成功}\ E \mid E \diamond e \mid v \diamond E
\end{array}
$$

图 11. 值与求值上下文的语法。

$$
\begin{array}{l}
E[e_a] \rightarrowtail E[e_b] \text{ iff } e_a \rightsquigarrow e_b \\
(v \leftarrow \text{失败}) \rightsquigarrow \text{失败} \\
((\mathsf{seq} \Leftarrow v_a \Leftarrow v_b) \leftarrow v_i) \rightsquigarrow (v_b \leftarrow (v_a \leftarrow v_i)) \\
((\mathsf{choice} \Leftarrow v_a \Leftarrow v_b) \leftarrow v_i) \rightsquigarrow (v_a \leftarrow v_i) \diamond (v_b \leftarrow v_i) \\
((\text{规则}\ p \to e) \leftarrow \text{成功}\ v) \rightsquigarrow \text{失败 如果 } \mathbf{match}(p;\, v) = \mathbf{失败模式} \\
((\text{规则}\ p \to e) \leftarrow \text{成功}\ v) \rightsquigarrow \text{成功}\ e[\mathcal{S}] \text{ 如果 } \mathbf{match}(p;\, v) = \mathcal{S} \neq \mathbf{失败模式} \\
\text{失败} \diamond \text{成功}\ v_b \rightsquigarrow \text{成功}\ v_b \\
\text{成功}\ v_a \diamond \text{成功}\ v_b \rightsquigarrow \text{成功}\ v_a \\
\text{成功}\ v_a \diamond \text{成功}\ v_b \rightsquigarrow \text{成功}\ v_b \\
\text{成功}\ v_a \diamond \text{失败} \rightsquigarrow \text{成功}\ v_a \\
\text{失败} \diamond \text{失败} \rightsquigarrow \text{失败} \\
(\mathsf{st}\ x \Rightarrow e) \Leftarrow v \rightsquigarrow e[x \mapsto v] \\
\text{let绑定}\ x = v \mathsf{in}\ e \rightsquigarrow e[x \mapsto v]
\end{array}
$$

$$
\begin{array}{l}
\mathbf{match}: \ p \to v \to \mathcal{S} \\
\mathbf{match}(x; v) = x \mapsto v \\
\mathbf{match}(\ell\, p; \ell\, v) = \mathbf{match}(p;\, v) \\
\mathbf{match}(\,();\,()) = (\cdot \mapsto \cdot) \\
\mathbf{match}(\,(p_a, p_b);\, (v_a, v_b)) = \\
\quad \mathbf{match}(p_b; v_b) \circ \mathbf{match}(p_a; v_a) \\
\mathbf{match}(p; v) = \mathbf{失败模式}
\end{array}
$$

图 12。操作语义

并且不会发生任何名称冲突，因为所有轨迹变量都会被重新命名为 Fresh。此外，**T-Let** 规则会通过要求 Φ_f 非空来触发示例 6 和示例 7 中的错误。类型变量实例化规则的形式为 $\Delta \dashv \sigma \leq \omega$，说明 σ 是不带跟踪环境的类型方案，并且可以在种类环境 Δ 中实例化为带跟踪类型 ω。行和类型的实例化是标准的，但变量所实例化到的类型或行不得包含任何轨迹变量，因而我们可以清楚地区分轨迹变量抽象和类型变量抽象，并保留跟踪信息。

策略变量线性性的变通方案。 回顾第 3.2 节中提到的线性限制，我们禁止定义包含重复使用策略变量的策略组合子，例如 $\mathsf{st}\ x \Rightarrow \mathsf{seq} \Leftarrow x \Leftarrow x$。对此限制的一种变通方案是，可以在 `let`绑定中定义策略，并重复使用多态的`let`绑定变量，f例如，这是因为每次使用 f时，绑定的跟踪环境都会被重新命名，因此 f的所有出现形式的类型中不会存在公共轨迹变量。

3.7 操作语义

图 11 展示了值的语法（记作 v）和求值上下文（记作 E），其定义采用了 [Wright 和 Felleisen 1994] 所使用的风格。在 Typed ELEVATE 中，原始策略组合子及其对值的应用被视为值的一个子集，即组合子值（记作 cv）。从这个角度看，原始策略组合子类似于 Haskell 中的数据构造器。其余的值和求值上下文定义都很直接。

图 12 展示了 Typed ELEVATE 的操作语义中的归约规则，以及执行模式匹配的函数 **match** 的定义。关系 $e_m \rightsquigarrow e_n$ 包含归约的概念，其中有趣的情形是 e_m 归约为 e_n；将这一关系嵌入

求值上下文后，我们得到步进关系 $e_m \rightarrowtail e_n$，它适用于任意可约式 [Wright 和 Felleisen 1994]。所有这些定义都很直接。与 **unifyTrace** 类似， **match** 产生的代换也扩展了一个底值 **failPat**，用于表示模式匹配失败。引理 3.3 表明， **match** 函数定义正确。

引理 3.3（模式匹配）。如果 $\mathbf{match}(p;\ v)\ = \mathrm{S}\ \neq$ **失败模式**，那么 $\mathbf{p2e}(p)[\mathrm{S}] = v$。

证明。根据 **match** 的定义，并对 p进行归纳。 □

值得注意的是，引入组合子值是我们做出的一项设计选择。如图 9所示，我们在策略执行期间为类型保持性的轨迹精度作出了折中。同时，如 `seq` 和 `choice`的归约规则所示，原始策略组合子被解释为策略执行的组合。因此，为了将精度上的折中限制在有限范围内，我们总是在执行之前，将策略的组合组织为具有精确轨迹的组合子值。这项设计选择的一个积极副作用是，它也使原始策略组合子的扩展更加容易。

3.8 可靠性与强正规化

本节将展示我们的追踪类型系统的可靠性及其强正规化性质。在本节以及后续各节中，对于关系 R，我们将 R^+ 记作其传递闭包，将 R^* 记作其自反传递闭包。

在给出类型健全性定理之前，我们先介绍以下一系列替换引理。引理 3.4 和引理 3.5 针对重写规则的执行。引理 3.4 指出，如果非空迹的重写规则执行中的模式匹配成功，则该执行保持类型不变；而引理 3.5 指出，空迹的重写规则执行必定失败。

引理 3.4（规则替换）。如果 $\Delta;\ \Gamma \vdash (\text{规则}\ \ p \to e) \leftarrow \text{成功}\ \ v :\ \ [\alpha_r] \blacktriangleright \tau_r \dashv \alpha_r \triangleright []$，且 $\mathbf{match}(p;\ v)\ = \mathcal{S}\ \neq$ **失败模式**，则 $\Delta;\ \Gamma \vdash \text{成功}\ \ e[\mathcal{S}] :\ \ [\alpha_r] \blacktriangleright \tau_r \dashv \alpha_r \triangleright []$。

证明。 证明通过对 p 的类型判断推导进行归纳展开。详见完整证明 第 31 页。 □

引理 3.5（失败规则）。若 $\Delta;\ \Gamma \vdash$（规则 $p \to e$）$\leftarrow$成功 $v :\ \ [] \blacktriangleright \tau_r$，则（规则 $p \to e$）$\leftarrow$成功 $v \rightsquigarrow$ 失败。

证明。证明对 p的类型判断推导进行归纳。完整证明见第31页。□

引理3.6表明，策略组合子应用的 β 归约保持类型不变。它还说明了类型规则 **T-S-Lam** 的设计依据。

引理 3.6（策略替换）。若 $\Delta;\ \Gamma;\ \Phi^{\downarrow};\ \Sigma^{\downarrow} \vdash$（策略 $x \Rightarrow e_f$）$\Leftarrow e :\ \ \omega \dashv \Phi^{\uparrow};\ \Sigma^{\uparrow}$，则$\Delta;\ \Gamma;\ \Phi^{\downarrow};\ \Sigma^{\downarrow} \vdash e_f[x \mapsto e] :\ \ \omega \dashv \Phi^{\uparrow};\ \Sigma^{\uparrow}$。

证明。证明对 e_f的类型判断推导进行归纳。完整证明见第32页。□

借助此处所示的引理以及附录中更为详细的内容，类型可靠性的两个部分，即保持性（推论 3.8）和进展性（定理 3.9）的证明是直接的。

定理 3.7（主体归约）。如果 $\Delta \vdash e_m :\ \omega \dashv \Phi$，并且 $e_m \rightsquigarrow e_n$，那么 $\Delta \vdash e_n :\ \omega \dashv \Phi$。

证明。 对归约进行分类讨论 $e_m \rightsquigarrow e_n$。无法通过常规方法证明的情形，其证明见引理 3.4、引理3.6和引理 C.1。 □

推论 3.8（保持性）。若 $\Delta \vdash e_m : \omega \dashv \Phi$，且 $e_m \rightarrowtail e_n$，则 $\Delta \vdash e_n : \omega \dashv \Phi$.

证明。 这是定理3.7根据 $\rightarrowtail$的定义直接得到的推论。 □

定理 3.9（进展性）。 如果 $\Delta \vdash e_m : \omega \dashv \Phi$，那么 e_m 要么是一个值，要么存在一个 e_n 使得 $e_m \rightarrowtail e_n$。

证明。我们需要证明，如果 $\Delta \vdash e_m : \omega \dashv \Phi$，那么 e_m 要么是一个值，要么存在一个求值上下文 E以及两个表达式 e_j和 e_k，使得 $e_m = E[e_j]$和 $e_j \rightsquigarrow e_k$。证明通常通过对 $\Delta \vdash e_m : \omega \dashv \Phi$的类型推导进行归纳来完成。□

推论 3.10（类型可靠性）。 如果 $\Delta \vdash e_m : \omega \dashv \Phi$，那么 e_m 要么是一个值，要么存在一个e_n 使得 $e_m \rightarrowtail^+ e_n$ 且 $\Delta \vdash e_n : \omega \dashv \Phi$。

证明。 这是推论 3.8 和定理 3.9的直接推论。 □

除了类型可靠性之外，Typed ELEVATE 还具有强正规化性质。STLC 的强正规化证明中所使用的证明技术 [Pierce 2002] 也可以应用于此处。

定理 3.11（强正规化）。如果 $\Delta \vdash e : \omega \dashv \Phi$，那么 $e \rightarrowtail^* v$ 其中 v 是一个值。

证明。 容易证明 $\Delta;\ \Gamma \vdash$ （规则 $p \rightarrow e$） $\leftarrow e_i :\ \varphi_r \blacktriangleright \tau_r \dashv \Phi_r$ 总是会强正规化，因此无需考虑从 Θ 中取变量进行代换。于是，对于一个项 e，我们需要证明：在 Γ（通过 $\mathcal{S}_\Gamma$）和 Σ（通过 $\mathcal{S}_\Sigma$）中，将强规范化的项代换给变量后，在 e 中，$e[\mathcal{S}_\Gamma][\mathcal{S}_\Sigma]$ 也同样是强规范化的。该证明通过构造按类型 ω索引的强规范化项上的逻辑关系，然后对 e进行归纳来完成。 □

3.9 良类型策略的性质

本节正式陈述良类型策略的性质。推论 3.12和推论 3.13 讨论的是类型良好但空迹的表达式。推论 3.12指出，空迹结果类型的表达式求值后必定得到失败。推论 3.13 进一步细化了推论 3.12 中的条件，并指出，如果某个策略是空迹的，那么对于该策略的任何类型良好的执行，其结果都会是失败。推论 **T-Let** 中对跟踪环境进行的非空检查有推论 3.13 作为依据，因为一个已知总是失败的策略毫无用处，或是无效产出的。

推论 3.12（空迹失败结果）。若 $\Delta \vdash e : [] \blacktriangleright \tau$，则 $e \rightarrowtail^*$ 失败。

证明。这直接由定理 3.11 和推论 3.10 得出，因为失败 是唯一能够以类型 $[] \blacktriangleright \tau$表示的值。 □

推论 3.13（空迹的无效产出策略）。如果 $\Delta \vdash e_f :\ \tau_a \xrightarrow{[]} \tau_b$，那么对于任意 e_i满足 $\Delta \vdash e_f \leftarrow e_i :\ \phi_r \blacktriangleright \tau_r \dashv \Phi_r$的，有 $(e_f \leftarrow e_i) \rightarrowtail^+$失败。

证明。这是推论3.12 和**T-Exec**的直接推论。 □

引理3.14 表明，类型良好且良好追踪的规则执行必须成功。由于**T-Exec**在精度方面有所折衷，我们在此使用形如 $\Delta;\Gamma \nvdash e :\ [] \blacktriangleright \tau$ 的记号，来正式说明“良好追踪”的含义，即 e 不能使用空迹的类型进行类型检查$[] \blacktriangleright \tau$。具体而言，在引理3.14中，这意味着**compTrace** 在**T-Exec** 中必须返回一个至少包含一条轨迹的带迹三元组。结合 $\Delta;\Gamma \vdash e_r :\ \varphi_r \blacktriangleright \tau_r \dashv \varphi_r \rhd []$，我们便得到“类型良好且良好追踪”的完整形式化表述。

引理 3.14（成功规则）。如果对于 $e_r =$ （规则 $p \rightarrow e$） $\leftarrow$ 成功 v，存在 $\Delta;\ \Gamma \vdash e_r :\ \varphi_r \blacktriangleright \tau_r \dashv \varphi_r \rhd []$ 并且 $\Delta;\ \Gamma \nvdash e_r :\ [] \blacktriangleright \tau$，那么 $e_r \rightarrowtail^+$ 成功 v_s。

证明。证明通过对 p的类型判断推导进行归纳来完成。请参见完整证明 第 35页。□

最后，推论 3.15 表明，类型良好且良好追踪的策略执行必定成功。更准确地说，鉴于非确定性，应将其表述为“至少存在一个成功的执行结果”。该推论依赖于类型可靠性（推论 3.10）和引理C.2；由于命题较长，该引理被移至附录。引理 C.2 的非正式描述是：它表明每条轨迹都对应于策略执行中的一条可能执行路径，这可以视为将引理 3.14推广到策略。

推论 3.15（成功重写）。若 $\Delta \vdash e_f \leftarrow e_i : \quad \varphi_r \blacktriangleright \tau_r \dashv \Phi_r$ 且 $\Delta \nvdash e_f \leftarrow e_i : \quad [] \blacktriangleright \tau$，则存在一个归约 $(e_f \leftarrow e_i) \rightarrowtail^{+}$成功 v_s。

证明。 这直接推论自引理 C.2 和推论 3.10。 □

4 讨论

本节讨论我们在选择组合子方面所做的一项核心设计选择。在 Typed ELEVATE 中，我们仅支持非确定性选择版本；而 ELEVATE 和 Stratego 还提供一种称为左选择的确定性版本：它先在左侧应用策略，只有在应用失败时才尝试在右侧应用策略。

我们通过考察策略 `e8`来理解这一设计选择。该策略定义为先进行选择组合，随后依次应用另一条重写规则：

```
let e8 = (rule m * n -> n * m || rule 1*v -> v) ; rule m + n -> n + m
```

该选择组合包含两条针对乘法的规则，其类型如下：

$(\texttt{rule m * n -> n * m || rule 1*v -> v}) : (a_0 * a_1) \ | (1 * b_0) \xrightarrow{[a,b]} (a_1 * a_0) \ | \ b_0$

这条唯一的重写规则交换加法的操作数，其类型如下：

$\texttt{rule m + n -> n + m} : c_0 + c_1 \xrightarrow{[c]} c_1 + c_0$

因此，w 根据我们的类型系统，其顺序组合具有以下类型：

$\texttt{let e8 = (rule m * n -> n * m || rule 1*v -> v) ; rule m + n -> n + m} : 1 * (d_0 + d_1) \xrightarrow{[d]} 1 * (d_1 + d_0)$

最终策略类型中只有一条由 d 标识的轨迹，这反映出该策略期望输入的形式为 $1 * (d_0 + d_1)$。对于此类输入，推论3.15 保证，整体策略执行必须包含一条成功的执行路径，即 `rule 1 * v -> v`后接 `rule m + n -> n + m`。然而，也许令人意外的是，当我们使用左偏确定性选择组合子时，情况并非如此！该策略执行将始终返回失败。左偏选择组合子首先将第一个策略应用于输入；由于模式 `m * n` 与 `1 * v`重叠，右侧策略在运行时无法访问，因此始终会选择左侧策略。但是遗憾的是，左侧策略无法与附加的重写规则进行顺序组合，因此整体策略执行总是失败。

如果我们希望像第3.9节中那样，对类型良好的策略的性质作出类似的形式化陈述，但采用左选择，那么我们会发现，推论 3.12 和推论 3.13 仍然成立，因为它们只依赖于类型可靠性和强正规化，而这两者不会因切换到左选择而受到影响。另一方面，推论 3.15 将不再成立，因为如上面的示例所示，在左选择下，轨迹失去了与执行路径之间的对应关系。换言之，我们将无法在左选择下证明引理 C.2。为了获得这样一个性质，即类型良好且良好追踪的策略执行必须成功，我们需要从类型中静态移除包含不可访问模式的轨迹，这将使

追踪机制的设计变得更具挑战性。此外，对于包含左选择的策略组合子，移除不可访问模式的轨迹意味着，策略组合子定义中现有策略的类型可能会受到策略参数类型的影响。这可能会带来可组合性问题，并可能使类型可靠性的证明更加困难。

值得注意的是，关于类型良好但空迹策略的论述对类型系统的变化具有更强的适应性，因为它们仅依赖于类型可靠性和强正规化。诸如切换到确定性选择版本、允许重写规则定义中存在无追踪易失败条件，以及添加无法被精确追踪的基本类型等变化，可能会使关于类型良好且良好追踪的策略执行成功的论述不再成立；但只要类型系统是可靠的并且具有强正规化性质，我们仍然可以断言，类型良好但空迹策略的执行结果必定是 失败。

5 相关工作

重写系统的类型系统。 项重写系统 [Dershowitz 1985] 已被证明可用于各种应用，例如程序变换、语言语义，以及定理证明系统中的计算。在专门为程序变换设计的策略语言中，例如 Borovanský 等人 [1996],的 ELAN、Visser 等人 [1998] 和Bravenboer 等人 [2008], 的 Stratego，以及 Winter 和 Subramaniam [2004], 的 TL，用户可以使用组合运算符定义策略，以控制归约序列；这一设计后来被 Hagedorn 等人 [2020]应用于 ELEVATE 的设计与实现。基础的类型系统和策略项重写的形式化由 Lämmel [2003] 以及 Kaiser 和 Lämmel [2009]提出，涵盖了泛型遍历。 Smits 和 Visser [2020] 展示了 Core Gradual Stratego 的类型系统，该系统使用渐进式类型系统，将动态类型的灵活性与静态类型所提供的安全性结合起来。 Koppel [2023] 使用组合式数据类型 Haskell 库，该库最初旨在解决表达式问题，并用于泛型编程； [Bahr 和 Hvitved 2011], 则利用该库确保重写过程中存在适用的语言构造，并实现保类型程序转换。与此同时，我们的工作聚焦于重写后程序的具体形态，强调策略的有效组合，并使用带有轨迹的类型系统在静态层面排除无用策略。

行多态语言。 由 Rémy [1989] 和 Wand [1991], 行多态是一种参数多态，允许在类型中将可扩展结构表示为行，通常是一系列标签-类型对（尽管也存在其他概念，例如 Morris 和 McKinna [2019]提出的概念），可将其作为（多态）变体和记录的基础。行多态与结构子类型具有同等的表达能力，但能与 Hindley-Milner 风格类型推断顺畅协作，并广泛应用于类型系统建模。OCaml 中结构多态类型的实现采用了行多态，并保证类型可靠性 [Garrigue 2015]。在 Links 中，Hillerström 和 Lindley [2016]使用行多态来实现代数效应和效应处理器，为带效应计算提供模块化抽象。 Blume 等人[2006] 使用行来实现多态变体和记录，并将分支或模式匹配分支视为一等值，从而使模式匹配表达式能够随时扩展新的分支。这类似于在 Typed ELEVATE 中使用 `choice` 组合子组合策略，但缺少轨迹所提供的精度。近期，Morris 和 McKinna [2019] 为现有的行多态类型系统提出了一套通用理论，该理论聚焦于行连接，并由此构建了语言 ROSE，作为一种灵活的可扩展数据类型编程工具。

6 结论

本文介绍 Typed ELEVATE，这是一种具有强静态类型系统的策略语言，能够拒绝组合不当的策略。我们讨论了若干实际的策略组合，展示了我们的类型系统为策略组合推导精确类型，以及对有问题的策略组合发出警告和错误的能力。

我们的类型系统将基于行多态的结构类型系统与新颖的追踪类型系统相结合。我们使用结构类型来表示在类型层面被重写的表达式的形状。追踪系统被设计为结构类型之上的附加层，用于表示策略可能被执行的方式。

我们确认了类型系统的可靠性，形式化地论证了为何拒绝空追踪策略（因为它们在运行时必然失败），并证明良好追踪的策略在运行时必须具有一条可能的执行路径。最后，我们讨论了非确定性选择组合子对于我们强形式化结果的重要性。

参考文献

Patrick Bahr 和 Tom Hvitved。2011年。组合数据类型。载于第七届 ACM SIGPLAN 泛型编程研讨会论文集（日本东京）（WGP '11）。计算机协会，美国纽约州纽约市，83–94。https://doi.org/10.1145/2036918.2036930 Matthias Blume、Umut A. Acar 和 Wonseok Chae。2006年。一等案例的可扩展编程。载于第十一届 ACM SIGPLAN 函数式编程国际会议论文集（美国俄勒冈州波特兰）（ICFP '06）。计算机协会，美国纽约州纽约市，239–250。https://doi.org/10.1145/1159803.1159836 Peter Borovanský、Claude Kirchner、Hélène Kirchner、Pierre-Etienne Moreau 和 Marian Vittek。1996年。ELAN：一种基于计算系统的逻辑框架。《理论计算机科学电子札记》4（1996），35–50。https: //doi.org/10.1016/S1571-0661(04)00032-5 RWLW96，第一届重写逻辑及其应用国际研讨会。Martin Bravenboer、Karl Trygve Kalleberg、Rob Vermaas 和 Eelco Visser。2008年。Stratego/XT 0.17：用于程序变换的语言和工具集。《计算机程序设计科学》72，1（2008），52–70。https://doi.org/10.1016/j.scico.2007.11.003 专题：实验性软件与工具包第二期（EST）。Tianqi Chen、Thierry Moreau、Ziheng Jiang、Lianmin Zheng、Eddie Q. Yan、Haichen Shen、Meghan Cowan、Leyuan Wang、Yuwei Hu、Luis Ceze、Carlos Guestrin 和 Arvind Krishnamurthy。2018年。TVM：用于深度学习的自动化端到端优化编译器。载于 OSDI。USENIX 协会，578–594。David Delahaye。2000年。Coq 系统的策略语言。载于 LPAR（《计算机科学讲义》，第1955卷）。Springer，85–95。Nachum Dershowitz。1985年。使用重写系统进行计算。《信息与控制》65，2-3（1985），122–157。Jacques Garrigue。2015年。一种具有结构多态和递归类型的 ML 认证实现。《数学结构与计算机科学》25，4（2015），867–891。https://doi.org/10.1017/S0960129513000066 Michael J. C. Gordon、Robin Milner、F. Lockwood Morris、Malcolm C. Newey 和 Christopher P. Wadsworth。1978年。用于 LCF 交互式证明的元语言。载于 POPL。ACM 出版社，119–130。Bastian Hagedorn、Johannes Lenfers、Thomas Koehler、Xueying Qin、Sergei Gorlatch 和 Michel Steuwer。2023年。以函数式方式实现高性能：将高性能优化表达为重写策略。《ACM 通讯》66，3（2023），89–97。Bastian Hagedorn、Johannes Lenfers、Thomas Kundefinedhler、Xueying Qin、Sergei Gorlatch 和 Michel Steuwer。2020年。以函数式方式实现高性能：将高性能优化表达为重写策略的函数式珍珠。Proc. ACM Program. Lang. 4，ICFP，文章92（2020年8月），29页。https://doi.org/10.1145/3408974 Daniel Hillerström 和 Sam Lindley。2016年。借助行与处理器释放效应。载于第一届类型驱动开发国际研讨会论文集（日本奈良）（TyDe 2016）。计算机协会，美国纽约州纽约市，15–27。https://doi.org/10.1145/2976022.2976033 Jieh Hsiang、Hélène Kirchner、Pierre Lescanne 和 Michaël Rusinowitch。1992年。自动定理证明的项重写方法。《逻辑程序杂志》14，1&2（1992），71–99。Markus Kaiser 和 Ralf Lämmel。2009年。基于 Isabelle/HOL 的 Stratego 式遍历策略模型。载于第11届 ACM SIGPLAN 声明式编程原理与实践会议论文集（葡萄牙科英布拉）（PPDP '09）。计算机协会，美国纽约州纽约市，93–104。https://doi.org/10.1145/1599410.1599423 Lennart C. L. Kats 和 Eelco Visser。2010年。Spoofax 语言工作台。载于 SPLASH/OOPSLA Companion。ACM，237–238。

Hélene Kirchner。2015年。重写策略与策略式重写程序。载于《逻辑、重写与并发：献给 José Meseguer 的论文集，纪念其65岁生日》，Narciso Martí-Oliet、Peter Csaba Ölveczky 和 Carolyn Talcott（编）。Springer International Publishing，Cham，380–403。https://doi.org/10.1007/978-3-319-23165-5_18 James Koppel。2023年。类型化多语言策略组合子。载于 Eelco Visser 纪念研讨会（EVCS 2023）（《开放获取信息学丛书》（OASIcs），第109卷），Ralf Lämmel、Peter D. Mosses 和 Friedrich Steimann（编）。Schloss Dagstuhl – Leibniz-Zentrum für Informatik，德国 Dagstuhl，16:1–16:9。https://doi.org/10.4230/OASIcs.EVCS.2023.16 Ralf Lämmel。2003年。使用项重写策略进行类型化泛型遍历。《逻辑与代数程序设计杂志》54，1（2003），1–64。https://doi.org/10.1016/S1567-8326(02)00028-0 Massimo Marchiori。1994年。作为项重写系统的逻辑程序。载于 ALP（《计算机科学讲义》，第850卷）。Springer，223–241。J. Garrett Morris 和 James McKinna。2019年。抽象化可扩展数据类型：或者，换个名字称作行。《Proc. ACM Program. Lang.》3，POPL，文章12（2019年1月），28页。https://doi.org/10.1145/3290325 Benjamin C. Pierce。2002年。《类型与程序设计语言》（第1版）。麻省理工学院出版社，第12章，149–152。Jonathan Ragan-Kelley。2023年。技术视角：重新思考用户可调度语言的设计。《ACM 通讯》66，3（2023），88。Jonathan Ragan-Kelley、Connelly Barnes、Andrew Adams、Sylvain Paris、Frédo Durand 和 Saman P. Amarasinghe。2013年。Halide：用于优化图像处理流水线中并行性、局部性和重新计算的语言与编译器。载于 PLDI。ACM，519–530。Didier Rémy。1989年。在 ML 的自然扩展中对记录和变体进行类型检查。载于第16届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集（美国得克萨斯州奥斯汀）（POPL ’89）。计算机协会，美国纽约州纽约市，77–88。https://doi.org/10.1145/75277.75284 Grigore Rosu 和 Traian-Florin Serbanuta。2010年。K 语义框架概述。《逻辑与代数方法程序设计杂志》79，6（2010），397–434。https://doi.org/10.1016/j.jlap.2010.03.012 Jeff Smits 和 Eelco Visser。2020年。渐进式类型化策略。载于第13届 ACM SIGPLAN 软件语言工程国际会议论文集（线上会议，美国）（SLE 2020）。计算机协会，美国纽约州纽约市，1–15。https://doi.org/10.1145/3426425.3426928 Eelco Visser。2001年。程序变换系统中的重写策略综述。《理论计算机科学电子札记》57（2001），109–143。https://doi.org/10.1016/S1571-0661(04)00270-1 WRS 2001，第一届重写与程序设计中的归约策略国际研讨会。Eelco Visser、Zine-el-Abidine Benaissa 和 Andrew Tolmach。1998年。使用重写策略构建程序优化器。载于第三届 ACM SIGPLAN 函数式编程国际会议论文集（美国马里兰州巴尔的摩）（ICFP ’98）。计算机协会，美国纽约州纽约市，13–26。https://doi.org/10.1145/289423.289425 Mitchell Wand。1991年。记录串接与多重继承的类型推断。《信息与计算》93，1（1991年7月），1–15。https://doi.org/10.1016/0890-5401(91)90050-C Victor L. Winter 和 Mahadevan Subramaniam。2004年。瞬态组合子、高阶策略与分布式数据问题。《计算机程序设计科学》52，1（2004），165–212。https://doi.org/10.1016/j.scico.2004.03.006 专题：程序变换。A.K. Wright 和 M. Felleisen。1994年。类型可靠性的句法方法。《信息与计算》115，1（1994年11月），38–94。https://doi.org/10.1006/inco.1994.1093

辅助函数的定义

$$
\begin{aligned}
&\mathbf{p2e}\colon\quad p \to e\\
&\mathbf{p2e}(x) \;= x\\
&\mathbf{p2e}(\ell\ p) \;= \ell\ \mathbf{p2e}(p)\\
&\mathbf{p2e}(\ ()\ ) \;= ()\\
&\mathbf{p2e}(\ (p_a,\ \ p_b)) \;= (\mathbf{p2e}(p_a),\ \ \mathbf{p2e}(p_b))
\end{aligned}
$$

图 13. 模式表达式转换的定义

B 轨迹操作的定义

$$
\begin{aligned}
&\textbf{擦除}\colon\quad t \to t\\
&\textbf{擦除}(\tau_{\psi}) \;= \textbf{擦除}(\tau)\\
&\textbf{擦除}(v) \;= v\\
&\textbf{擦除}(\ \rho) \;= \textbf{擦除}(\rho)\\
&\textbf{擦除}(\ v\text{作为}\ \langle\rho\rangle) \;= v\text{作为}\ \langle\textbf{擦除}(\rho)\rangle\\
&\textbf{擦除}(\ ()) \;= ()\\
&\textbf{擦除}((\tau_m, \tau_n)) \;= (\textbf{擦除}(\tau_m), \textbf{擦除}(\tau_n))\\
&\textbf{擦除}(\cdot) \;= \cdot\\
&\textbf{擦除}(\ell : \tau \mid \rho) \;= (\ell : \textbf{擦除}(\tau) \mid \textbf{擦除}(\rho))\\
&\textbf{擦除}(\varphi \blacktriangleright \tau) \;= [\,] \blacktriangleright \textbf{擦除}(\tau)\\
&\textbf{擦除}(\tau_p \xrightarrow{\varphi} \tau_e) \;= \textbf{擦除}(\tau_p) \xrightarrow{[\,]} \textbf{擦除}(\tau_e)\\
&\textbf{擦除}((\tau_p \xrightarrow{\varphi} \tau_e) \Rightarrow \omega) \;= \textbf{擦除}(\tau_p \xrightarrow{\varphi} \tau_e) \Rightarrow \textbf{擦除}(\omega)
\end{aligned}
$$

图 14. 轨迹擦除的定义

$$
\begin{aligned}
&-[-]\colon\ t \to \overline{S}\ \to t\\
&\tau_{[\beta]}[\overline{S}] = S\ (\overline{\tau[\beta]})\ \textbf{如果}\ \ \tau_{[\beta]} \in dom(S)^{-}\\
&\tau_{[\beta]}[\overline{S}] = \tau_{[\beta]}\ \textbf{如果}\ \ \tau_{[\beta]} \notin dom(S)^{-}\\
&v[\overline{S}] = v\\
&\rho[\overline{S}] = \rho[S]^{-}\\
&()[\overline{S}] = ()\\
&(\tau_m,\ \ \tau_n)^{-}[S] = (\tau_m[\overline{S}],\ \ \tau_n[\overline{S}])\\
&(\ell\colon\ \tau \mid \rho)\ [\overline{S}] = (\ell\colon\ \tau[\overline{S}] \mid \rho[\overline{S}])\\
&\cdot[\overline{S}] = \cdot
\end{aligned}
$$

图 15. 轨迹替换的定义

$$\textbf{unifyTrace} : \ (\Phi;\ \iota;\ \iota) \ \to \overline{\mathcal{S}}$$

$$\textbf{unifyTrace}(\alpha \rhd \phi;\ ()[\alpha];\ ()) = \textbf{failTrace}$$

$$\textbf{unifyTrace}(\alpha \rhd \phi;\ ();\ ()[\alpha]) = \textbf{失败轨迹}$$

$$\textbf{unifyTrace}(\alpha \rhd [\beta, \phi];^{\tau[\beta]};\ \tau_n) = \textbf{如果}\ \ \tau_n = \tau \textbf{那么 失败轨迹 否则}\ \tau[\beta] \mapsto \tau_n$$

$$\textbf{unifyTrace}(\alpha \rhd [\beta, \phi];\ \tau_m;^{\tau[\beta]}) = \textbf{如果}\ \ \tau_m = \tau \textbf{则 失败Trace 否则}\ \tau[\beta] \mapsto \tau_m$$

$$\textbf{unifyTrace}(\Phi; \langle \rho_m \rangle; \langle \rho_n \rangle) = \textbf{unifyTrace}(\Phi;\ \rho_m;\ \rho_n)$$

$$\textbf{unifyTrace}(\Phi; (\tau_m, \tau_n); (\tau_j, \tau_k)) = \textbf{let绑定}\overline{\mathcal{S}} = \textbf{unifyTrace}(\Phi; \tau_m;^{\tau_j})$$

$$\quad \textbf{在 unifyTrace}(\Phi; \tau_n[\overline{\mathcal{S}}]; \tau_k[\mathcal{S}\overline{]}) \circ \mathcal{S}\ \overline{\ }$$

$$\textbf{unifyTrace}(\Phi; \tau; \tau) \ = (\cdot \mapsto \cdot)$$

$$\textbf{unifyTrace}(\Phi; (\ell : \tau_m \mid \rho_m); (\ell : \tau_n \mid \rho_n)) = \textbf{let绑定}\mathcal{S} = \overline{\textbf{un}}\textbf{ifyTrace}(\Phi; \tau_m; \tau_n)\textbf{】【}。$$

$$\quad \textbf{在 unifyTrace}(\Phi; \rho_m[\overline{\mathcal{S}}]; \rho_n[\mathcal{S}\overline{]}) \circ \mathcal{S}\ \overline{\ }$$

$$\textbf{unifyTrace}(\Phi;\ \rho;\ \rho) \ = \ (\cdot \mapsto \cdot)$$

$$\textbf{unifyTrace}(\Phi;\ [\alpha] \blacktriangleright \tau_m;\ [\alpha] \blacktriangleright \tau_n) = \textbf{unifyTrace}(\Phi;\ \tau_m;\ \tau_n)$$

$$\textbf{unifyTrace}(\Phi; \tau_m \xrightarrow{[\alpha]} \tau_n;\ \tau_j \xrightarrow{[\alpha]} \tau_k) = \textbf{let绑定}\ \overline{\mathcal{S}} = \textbf{unifyTrace}\ (\Phi;\ \tau_m;\ \tau_j)$$

$$\quad \textbf{在 unifyTrace}\ (\Phi;\ \tau_n[\overline{\mathcal{S}}];\ \tau_k[\mathcal{S}\overline{]})\ \circ\ \mathcal{S}\ \overline{\ }$$

图 16. 轨迹合一的定义

$$\textbf{traceIds} : (\Phi; \omega) \ \to \varphi$$

$$\textbf{traceIds}(\Phi;^{\tau_p} \xrightarrow{\varphi} \tau_e) = \varphi\ \textbf{若}\ \ \Phi \vdash \tau_p \dashv \varphi\ \textbf{且}\ \ \Phi \vdash \tau_e \dashv \varphi$$

$$\textbf{traceIds}(\Phi; \varphi \blacktriangleright \tau) = \varphi\ \textbf{如果}\ \ \Phi \vdash \tau \dashv \varphi$$

图 17. traceIds 的定义

选择 $-$ **在** $-$: $\varphi \to (\Phi;\ \Sigma;\ \omega) \to (\Phi;\ \Sigma;\ \omega)$

选择 φ **在** $(\Phi;\ \Sigma;\ \omega) =$

$(\Phi_r;\ \Sigma_r; \omega_r)$: $=$ **(·; 擦除(Σ); 擦除(ω))**

对于 α **在** φ

如果 $\alpha \rhd \phi \in \Phi$

Σ_i **:=·**

对于 $(x:\ \omega_x)$ **在** Σ

$\Sigma_i : = \Sigma_i,\ x:$ **按类型选择**$(\Phi;\ \alpha;\ \omega_x)$

$(\Phi_r;\ \Sigma_r;\ \omega_r) :=$ **Add** $(\alpha \rhd \phi;\ \Sigma_i;$ **按类型选择**$(\Phi;\ \alpha;\ \omega))$ **到** $(\Phi_r;\ \Sigma_r;\ \omega_r)$

返回 $(\Phi_r;\ \Sigma_r;\ \omega_r)$

按类型选择 : $(\Phi;\ \alpha;\ t) \to t$

按类型选择$(\alpha \rhd \phi, \Phi_r; \alpha; ()[\alpha,\psi_r]) =$ **()**$[\alpha]$

按类型选择$(\alpha \rhd [\beta, \phi_r], \Phi_r; \alpha; ^{\tau}[\beta,\psi_r]) =$ **擦除**$(\tau)[\beta]$

按类型选择$(\alpha \rhd \phi, \Phi_r; \alpha; ^{\tau_\psi}) =$ **按类型选择**$(\alpha \rhd \phi, \Phi_r; \alpha; \tau)$ **如果** $\phi \cap \psi = \emptyset$ **并且** $\alpha \notin \psi$

按类型选择$(\Phi;\ \alpha; \langle\rho\rangle) =$ **⟨按类型选择**$(\Phi;\ \alpha;\ \rho)\rangle$

按类型选择$(\Phi;\ \alpha;\ \nu$ 作为 $\langle\rho\rangle) = \nu$ 作为 **⟨按类型选择**$(\Phi;\ \alpha;\ \rho)\rangle$

按类型选择$(\Phi; \alpha; \cdot) = \cdot$

按类型选择$(\Phi; \alpha;\ \nu) = \nu$

按类型选择$(\Phi;\ \alpha; (\ell:\ \tau \mid \rho)) =$ **(**ℓ**:按类型选择**$(\Phi; \alpha; \tau)$ **|按类型选择**$(\Phi; \alpha;\ \rho))$

按类型选择$(\Phi; \alpha; ()) =$ **()**

按类型选择$(\Phi;\ \alpha; (\tau_m, \tau_n)) =$ **(按类型选择**$(\Phi;\ \alpha; \tau_m)$**, 按类型选择**$(\Phi;\ \alpha; \tau_n))$

按类型选择$(\Phi; \alpha;\ [\alpha, \varphi_r] \blacktriangleright \tau) = [\alpha] \blacktriangleright$ **按类型选择**$(\Phi;\ \alpha; \tau)$

按类型选择$(\Phi;\ \alpha; \varphi \blacktriangleright \tau) = [\,] \blacktriangleright$ **擦除**(τ) **如果** $\alpha \notin \varphi$

按类型选择$(\Phi;\ \alpha;^{\tau_p} \xrightarrow{[\alpha,\varphi_r]} \tau_e) =$ **按类型选择**$(\Phi; \alpha;^{\tau_p}) \xrightarrow{[\alpha]}$ **按类型选择**$(\Phi; \alpha; \tau_e)$

按类型选择$(\Phi;\ \alpha;^{\tau_p} \xrightarrow{\varphi} \tau_e) =$ **擦除**$(\tau_p) \xrightarrow{[\,]}$ **擦除**(τ_e) **如果** $\alpha \notin \varphi$

按类型选择$(\Phi; \alpha; (^{\tau_p} \xrightarrow{\varphi} \tau_e) \Rightarrow \omega) =$ **按类型选择**$(\Phi; \alpha;^{\tau_p} \xrightarrow{\varphi} \tau_e) \Rightarrow$ **按类型选择**$(\Phi;\ \alpha; \omega)$

图 18. **选择** $-$ **中** $-$的定义

$$\textbf{Fresh}\ \_ : (\Phi;\ \Sigma; \omega)\ \to (\Phi;\ \Sigma; \omega)$$

$$\textbf{Fresh}\ (\Phi;\ \Sigma; \omega)\ =$$

$$(\Phi_r;\ \Sigma_r; \omega_r) := (\cdot; \textbf{擦除}(\Sigma); \textbf{擦除}(\omega))$$

$$\textbf{对于}\ \alpha \rhd \phi,\ \textbf{其中}\ \Phi$$

$$(\alpha \rhd \phi;\ \Sigma_\alpha; \omega_\alpha)\ = \textbf{选择}\ [\alpha],\ \textbf{其中}\ (\Phi;\ \Sigma; \omega)$$

$$\phi_i :\ = \textbf{移除成员}\ (\alpha \rhd \phi;\ \omega_\alpha;\ [])$$

$$\textbf{对于}\ (x :\ \omega_x),\ \textbf{在}\ \Sigma_\alpha\ \textbf{中}$$

$$\phi_i :\ = \textbf{移除成员}\ (\alpha \rhd \phi;\ \omega_x;\ \phi_i)$$

$$(\Phi_n;\ \Sigma_n;\ \omega_n) :\ = (\alpha \rhd \phi_i;\ \Sigma_\alpha;\ \omega_\alpha)\ [\alpha \mapsto \textbf{新鲜}\ \alpha_n]$$

$$\textbf{对于}\ \beta,\ \textbf{在}\ \phi_i\ \textbf{中}$$

$$(\Phi_n;\ \Sigma_n;\ \omega_n) :\ = (\Phi_n;\ \Sigma_n;\ \omega_n)\ [\beta \mapsto \textbf{新鲜}\ \beta_n]$$

$$(\Phi_r;\ \Sigma_r; \omega_r) := \textbf{Add}\ (\Phi_n;\ \Sigma_n; \omega_n)\ \textbf{至}\ (\Phi_r;\ \Sigma_r; \omega_r)$$

$$\textbf{返回}\ (\Phi_r;\ \Sigma_r;\ \omega_r)$$

$$\textbf{移除成员} :\ (\Phi;\ t;\ \phi)\ \to \phi$$

$$\textbf{移除成员}(\alpha \rhd \phi;\ ()[\alpha];\ \phi_s)\ = \phi_s$$

$$\textbf{移除成员}(\alpha \rhd [\beta, \phi_r];\ \tau[\beta];\ \phi_s)\ = \textbf{如果}\ \beta \in \phi_s\ \textbf{则}\ \phi_s\ \textbf{否则}\ [\beta, \phi_s]$$

$$\textbf{移除成员}(\Phi;\ \langle\rho\rangle;\ \phi_s)\ = \textbf{移除成员}(\Phi;\ \rho;\ \phi_s)$$

$$\textbf{移除成员}(\Phi;\ \nu \mathsf{as} \langle\rho\rangle; \phi_s)\ = \textbf{移除成员}(\Phi;\ \rho; \phi_s)$$

$$\textbf{移除成员}(\Phi;\ \cdot; \phi_s)\ = \phi_s$$

$$\textbf{移除成员}(\Phi;\ \nu; \phi_s)\ = \phi_s$$

$$\textbf{移除成员}(\Phi; (\ell :\ \tau \mid \rho); \phi_s)\ = \textbf{移除成员}(\Phi;\ \rho; \textbf{移除成员}(\Phi; \tau;\ \phi_s))$$

$$\textbf{移除成员}(\Phi; (); \phi_s)\ = \phi_s$$

$$\textbf{移除成员}(\Phi; (\tau_m, \tau_n);\ \phi_s)\ = \textbf{移除成员}(\Phi; \tau_n; \textbf{移除成员}(\Phi; \tau_m; \phi_s))$$

$$\textbf{移除成员}(\Phi;\ \varphi \blacktriangleright \tau; \phi_s)\ = \textbf{移除成员}(\Phi; \tau; \phi_s)$$

$$\textbf{移除成员}(\Phi; \tau_p \xrightarrow{\varphi} \tau_e; \phi_s)\ = \textbf{移除成员}(\Phi; \tau_e; \textbf{移除成员}(\Phi; \tau_p;\ \phi_s))$$

$$\textbf{移除成员}(\Phi; (\tau_p \xrightarrow{\varphi} \tau_e) \Rightarrow \omega; \phi_s)\ = \textbf{移除成员}(\Phi; \omega; \textbf{移除成员}(\Phi; \tau_p \xrightarrow{\varphi} \tau_e; \phi_s))$$

图 19. **Fresh** –的定义

Add －到－: $(\Phi;\ \Sigma;\ \omega) \to (\Phi;\ \Sigma;\ \omega) \to (\Phi;\ \Sigma;\ \omega)$

$\textbf{Add}\ (\Phi_m;\ \Sigma_m;\omega_m)\ \textbf{to}\ (\Phi_n;\ \Sigma_n;\omega_n)\ =$

$\Sigma_r := \cdot$

对于$(x:\ \omega_x)$在 Σ_m 中

$\Sigma_r := \Sigma_r,\ x:\ \textbf{addType}(\Sigma_m(x);\ \Sigma_n(x))$

返回 $(\Phi_m,\ \Phi_n;\ \Sigma_r;\ \textbf{addType}(\omega_m;\omega_n))$

$\textbf{addType}:\ (t;\ t) \to t$

$\textbf{addType}\ (\tau^m_{\psi_m};\ \tau^n_{\psi_n})\ = \textbf{addType}\ (\tau^m;\ \tau^n)\ [\psi_m,\psi_n]$

$\textbf{addType}\ (\tau^m_{\psi_m};\ \tau^n)\ = \textbf{addType}\ (\tau^m;\ \tau^n)\ \psi_m$

$\textbf{addType}\ (\tau^m;\ \tau^n_{\psi_n})\ = \textbf{addType}\ (\tau^m;\ \tau^n)\ \psi_n$

$\textbf{addType}\ (\rho_m\rangle;\ \langle\rho_n)\ = \langle\textbf{addType}\ (\rho_m;\ \rho_n)$

$\text{addType}\ (v\,\mathsf{as}\,\langle\rho_m\rangle;\ v\,\mathsf{as}\,\langle\rho_n)\ = v\,\mathsf{as}\,\langle\text{addType}\ (\rho_m;\ \rho_n)$

$\textbf{addType}(\cdot;\cdot)\ = \cdot$

$\textbf{addType}(v;v)\ = v$

$\textbf{addType}((\ell:\ \tau_m \mid \rho_m);\ (\ell:\ \tau_n \mid \rho_n)) = (\ell:\ \textbf{addType}(\tau_m;\ \tau_n) \mid \textbf{addType}(\rho_m;\ \rho_n))$

$\text{addType}(\ ();\ ()\ = ()$

$\textbf{addType}\ (\ (\tau_m,\ \tau_n)\ ;\ (\tau_j,\ \tau_k)\ )\ = (\textbf{addType}\ (\tau_m;\ \tau_j)\ ,\ \textbf{addType}\ (\tau_n;\ \tau_j)\ )$

$\textbf{addType}(\varphi_m \blacktriangleright \tau_m;\varphi_n \blacktriangleright \tau_n)\ = [\varphi_m,\varphi_n] \blacktriangleright \textbf{addType}(\tau_m;\tau_n)$

$\textbf{addType}(\tau_m \xrightarrow{\varphi_m} \tau_n;\tau_j \xrightarrow{\varphi_j} \tau_k)\ = \textbf{addType}(\tau_m;\tau_j) \xrightarrow{[\varphi_m,\varphi_j]} \textbf{Add}(\tau_n;\tau_k)$

$\textbf{addType}((\tau_m \xrightarrow{\varphi_m} \tau_n) \Rightarrow \omega_m;(\tau_j \xrightarrow{\varphi_j} \tau_k) \Rightarrow \omega_j)\ = \textbf{addType}(\tau_m \xrightarrow{\varphi_m} \tau_n;\tau_j \xrightarrow{\varphi_j} \tau_k) \Rightarrow \textbf{addType}(\omega_m;\omega_j)$

图2。Add －到－

C 证明

在以下证明中，我们将 $mems(\Phi;\ \omega)$(定义于图 21) 表示由跟踪成员追踪的类型集合，该集合从单跟踪类型中提取 ω，且位于跟踪环境 Φ中。若参数Φ 可从上下文中明确得知，则省略不写。此记法也可轻松推广至环境。

$$
\begin{aligned}
&mems: (\Phi; t) \ \to \{\tau\} \\
&mems(\alpha \rhd [\beta, \phi]; ^{\tau\lfloor\beta\rfloor}) \ = \{^{\tau\lfloor\beta\rfloor}\} \\
&mems(\Phi;\ v) \ = \emptyset \\
&mems(\Phi; \rho) \ = mems(\Phi;\ \rho) \\
&mems(\Phi; ()) \ = \emptyset \\
&mems(\Phi; (\tau_m, \tau_n)) \ = mems(\Phi; \tau_m) \ \cup mems(\Phi; \tau_n) \\
&mems(\Phi; \cdot) \ = \emptyset \\
&mems(\Phi; (\ell:\ \tau \mid \rho)) \ = mems(\Phi; \tau) \ \cup mems(\Phi;\ \rho) \\
&mems(\Phi; \varphi \blacktriangleright \tau) \ = mems(\Phi; \tau) \\
&mems(\Phi; ^{\tau_p} \xrightarrow{\varphi} \tau_e) \ = mems(\Phi; ^{\tau_p}) \ \cup mems(\Phi; \tau_e) \\
&mems(\Phi; (^{\tau_p} \xrightarrow{\varphi} \tau_e) \ \Rightarrow \omega) \ = mems(\Phi; ^{\tau_p} \xrightarrow{\varphi} \tau_e) \ \cup mems(\Phi; \omega)
\end{aligned}
$$

图 21。 $mems$的定义

引理 3.4（规则替换）。若 $\Delta;\ \Gamma \vdash$ (规则 $p \to e$) $\leftarrow$ 成功 $v:\ [\alpha_r] \blacktriangleright \tau_r \dashv \alpha_r \rhd []$，且 $\mathbf{match}(p;\ v) = \mathcal{S} \neq$ **失败模式**，则 $\Delta;\ \Gamma \vdash$ 成功 $e[\mathcal{S}]:\ [\alpha_r] \blacktriangleright \tau_r \dashv \alpha_r \rhd []$。

引理 3.4 的证明。对 T-Exec 和 T-R-Lam 进行逆变换，可知 $\Delta \vDash_{\alpha_f} p:\ \tau_p \dashv \Phi; \Theta$，以及$\Delta;\ \Gamma;\ \Phi; \Theta \vdash e:\ \tau_e$，并且 $\Delta;\ \Gamma;\ \alpha_v \rhd []; \bigstar \vdash v:\ \tau_v$，且 $\mathrm{erase}(^{\tau_p}) = \mathrm{erase}(\tau_v)$，并且存在一个由 $\mathrm{unifyTrace}(\alpha_f \rhd \phi_f;\ [\alpha_f] \blacktriangleright \tau_p;\ [\alpha_f] \blacktriangleright (\tau_v[\alpha_v \mapsto \alpha_f]))$ 生成的 S。对 $\Delta \vDash_{\alpha_f} p:\ \tau_p \dashv \Phi;\ \Theta$ 的类型推导进行归纳，结合 $\mathrm{match}(p; v) \neq$失败模式以及 unifyTrace 的定义，可知 $S \neq \mathrm{failTrace}$。然后根据引理 3.2 和引理 3.3，可知 $\tau_v[\alpha_v \mapsto \alpha_f] = \tau_p[S],$，并且对于 p 中且也在 $dom(S)$ 中的每个变量 x，在 $^{\tau_p}$ 中且也在 $dom(S)$ 中都存在相应的 $^{\tau\lfloor\beta\rfloor}$，并且 $\Delta;\ \Gamma;\ \alpha_f \rhd []; \bigstar \vdash S(x) : S(^{\tau\lfloor\beta\rfloor})$。于是很容易证明 $\Delta;\ \Gamma;\ \alpha_f \rhd \overline{[]}; \bigstar \vdash e[S]:\ \tau_e[S],,$ 经过适当的重命名后即可为证明目标提供证据。□

引理 3.5（失败规则）。如果 $\Delta;\ \Gamma \vdash$ （规则 $p \to e$） $\leftarrow$ 成功 $v:\ [] \blacktriangleright \tau_r$，那么（规则 $p \to e$） $\leftarrow$ 成功 $v \rightsquigarrow$ 失败。

引理 3.5 的证明。通过对 T-Exec 和 T-R-Lam 进行逆变换，我们知道 $\mathrm{failTrace} = \mathrm{unifyTrace}(\alpha_f \rhd \phi_f;\ [\alpha_f] \blacktriangleright \tau_p;\ [\alpha_f] \blacktriangleright (\tau_v[\alpha_v \mapsto \alpha_f]))$，其中 $\Delta \vDash_{\alpha_f} p:\ \tau_p \dashv \Phi;\ \Theta$，以及 $\Delta;\ \Gamma;\ \Phi;\ \Theta \vdash e:\ \tau_e$，并且$\Delta;\ \Gamma;\ \alpha_v \rhd []; \bigstar \vdash v:\ \tau_v$，以及 $\mathrm{erase}(^{\tau_p}) = \mathrm{erase}(\tau_v)$。根据对$\Delta \vDash_{\alpha_f} p:\ \tau_p \dashv \Phi;\ \Theta$ 的类型推导进行归纳，并结合 unifyTrace 的定义，我们知道 failTrace 的唯一来源是 p 和 v 中的不匹配标签。根据 match 的定义，我们知道 $\mathrm{match}(p;\ v)$ 的结果必定是 failPat。因此，（规则 $p \to e$） $\leftarrow$ 成功 $v \rightsquigarrow$ 失败。□

引理 3.6（策略替换）。若 $\Delta;\ \Gamma;\ \Phi^{\downarrow};\ \Sigma^{\downarrow} \vdash$ （$\mathtt{st}\ x \Rightarrow e_f$） $\Leftarrow e:\ \omega \dashv \Phi^{\uparrow};\ \Sigma^{\uparrow}$，则$\Delta;\ \Gamma;\ \Phi^{\downarrow};\ \Sigma^{\downarrow} \vdash e_f[x \mapsto e]:\ \omega \dashv \Phi^{\uparrow};\ \Sigma^{\uparrow}$。

引理 3.6 的证明。通过对 T-S-App 和 T-S-Lam 进行逆变换，我们知道：Δ； Γ； $\Phi^{\downarrow}$、$\Phi_x^{\downarrow}$； $\Sigma^{\downarrow}$、 x： $\omega_x^{\downarrow} \vdash e_f$： $\omega_f \dashv \Phi_f^{\uparrow}$； $\Sigma_f^{\uparrow}$、 x： $\omega_x^{\uparrow}$ Δ； Γ； $\Phi^{\downarrow}$； $\Sigma^{\downarrow} \vdash e$： $\omega_e \dashv \Phi_e^{\uparrow}$； $\Sigma_e^{\uparrow}$擦除$(\omega_x^{\uparrow})$ = 擦除 (ω_e) （ $\Phi^{\uparrow}$； $\Sigma^{\uparrow}$； ω） =compTrace$(\Phi_f^{\uparrow}$； $\Sigma_f^{\uparrow}$； $\omega_x^{\uparrow}$； ω_f； $\Phi_e^{\uparrow}$； $\Sigma_e^{\uparrow}$； $\omega_e)$ 我们的目标是证明 Δ； Γ； $\Phi^{\downarrow}$； $\Sigma^{\downarrow} \vdash e_f[x \mapsto e]$： $\omega \dashv \Phi^{\uparrow}$； $\Sigma^{\uparrow}$。证明通过对 Δ； Γ； $\Phi^{\downarrow}$、 $\Phi_x^{\downarrow}$； $\Sigma^{\downarrow}$、 x： $\omega_x^{\downarrow} \vdash e_f$： $\omega_f \dashv \Phi_f^{\uparrow}$； $\Sigma_f^{\uparrow}$、 x： $\omega_x^{\uparrow}$ 的类型推导进行归纳来完成。

- **情形** $e_f = x$由 **T-S-Var** 可知，此情形显然成立。
- 情形 $e_f = e_g \Leftarrow e_y$★ 当 x在线性使用于 e_y时，通过对 T-S-App 作逆变换，我们知道：Δ； Γ； $\Phi^{\downarrow}$， $\Phi_x^{\downarrow}$； $\Sigma^{\downarrow}$， x： $\omega_x^{\downarrow} \vdash e_g$： $\omega_g \Rightarrow \omega_h \dashv \Phi_g^{\uparrow}$； $\Sigma_g^{\uparrow}\Delta$； Γ； $\Phi^{\downarrow}$， $\Phi_x^{\downarrow}$； $\Sigma^{\downarrow}$， x： $\omega_x^{\downarrow} \vdash e_y$： $\omega_y \dashv \Phi_y^{\uparrow}$； $\Sigma_y^{\uparrow}$， x： ω_x^{y}擦除(ω_g) =擦除(ω_y) $(\Phi_f^{\uparrow}$； $\Sigma_f^{\uparrow}$， x： $\omega_x^{\uparrow}$； $\omega_f)$ =compTrace$(\Phi_g^{\uparrow}$； $\Sigma_g^{\uparrow}$； ω_g； ω_h； $\Phi_y^{\uparrow}$； $\Sigma_y^{\uparrow}$， x： ω_x^{y}； $\omega_y)$ 根据归纳假设，我们知道：$(\Phi_u^{\uparrow}$； $\Sigma_u^{\uparrow}$； $\omega_y^{\uparrow})$ =compTrace$(\Phi_y^{\uparrow}$； $\Sigma_y^{\uparrow}$； ω_x^{y}； ω_y； $\Phi_e^{\uparrow}$； $\Sigma_e^{\uparrow}$； $\omega_e)\Delta$； Γ； $\Phi^{\downarrow}$； $\Sigma^{\downarrow} \vdash e_y[x \mapsto e]$： $\omega_y^{\uparrow} \dashv \Phi_u^{\uparrow}$； $\Sigma_u^{\uparrow}$我们的目标是证明 $(\Phi^{\uparrow}$； $\Sigma^{\uparrow}$； $\omega)$ =compTrace$(\Phi_g^{\uparrow}$； $\Sigma_g^{\uparrow}$； ω_g； ω_h； $\Phi_u^{\uparrow}$； $\Sigma_u^{\uparrow}$； $\omega_y^{\uparrow})$。

由于在枚举过程中，**compTrace** 对每条轨迹执行的操作都是一致的，因此证明只需考虑一条轨迹（忽略重命名）。我们在名称上加波浪号（例如 $\tilde{\omega}_e$），表示只考虑一条轨迹。

∗ 对于 $\Phi^{\uparrow}$ 中列出的每条轨迹，以下是我们目前已知的证据：我们在某些方程中添加注解，表明该方程也定义了一个（换句话说，例如，$\tilde{\omega}_e[S_e] := \tilde{\omega}_x^{\uparrow}[S_e]$ 表示 S_e 是通过使用 $\tilde{\omega}_x^{\uparrow}[S_e]$ 进行轨迹 $\tilde{\omega}_e$ 而定义的）。我们需要证明，unifyTrace 会给出一个代换 S_h（如果合 $\tilde{\omega}_h$ 成功）使得 $\tilde{\omega}_g[S_h] := \tilde{\omega}_y^{\uparrow}[S_h]$、 $\tilde{\omega}_h[S_h] = \tilde{\omega}_g[S_h]$ 和 $(\tilde{\Sigma}_g^{\uparrow}, \tilde{\Sigma}_u^{\uparrow})[S_h] = \tilde{\Sigma}^{\uparrow}$，即 $\tilde{\omega}_g[S_h] := \tilde{\omega}_y[S_y][S_h]$ $\tilde{\omega}_h[S_h] = \tilde{\omega}_h[S_g][S_e]$ $(\tilde{\Sigma}_g^{\uparrow}, (\tilde{\Sigma}_y^{\uparrow}, \tilde{\Sigma}_e^{\uparrow})[S_y])[S_h] = ((\tilde{\Sigma}_g^{\uparrow}, \tilde{\Sigma}_y^{\uparrow})[S_g], \tilde{\Sigma}_e^{\uparrow})[S_e]$

◦ 为证明 $\tilde{\omega}_h[S_h] = \tilde{\omega}_h[S_g][S_e]$，其中 S_h定义为 $\tilde{\omega}_g[S_h] := \tilde{\omega}_y[S_y][S_h]$，我们需要研究 S_h 和 S_e 的定义。– 关于 S_h 的定义，我们知道：$dom(S_y) \subseteq mems(\tilde{\omega}_e) \cup mems(\tilde{\omega}_x^{y})$ $mems(\tilde{\omega}_e) \cap mems(\tilde{\omega}_x^{y}) = \emptyset$ $dom(S_g) \subseteq mems(\tilde{\omega}_g) \cup mems(\tilde{\omega}_y)$ $mems(\tilde{\omega}_g) \cap mems(\tilde{\omega}_y) = \emptyset$ $mems(\tilde{\omega}_y) \cap mems(\tilde{\omega}_e) = \emptyset$ $mems(\tilde{\omega}_g) \cap mems(\tilde{\omega}_e) = \emptyset$ $mems(\tilde{\omega}_y) \cap mems(\tilde{\omega}_x^{y}) \supseteq \emptyset$因此， $mems(S_y(dom(S_y) \cap dom(S_g))) \cap (dom(S_g) \cup mems(\tilde{\omega}_y)) = \emptyset$，根据 unifyTrace 的定义，我们可以将 S_h 拆分为两个阶段，分别命名为 S_h^{yg} 和 S_h^{u}，其中 S_h^{yg} 可根据 S_g 定义如下。$S_h^{yg} = \forall \tau_{[\beta]} \in dom(S_g) \setminus dom(S_y)$ $\tau_{[\beta]} \mapsto S_y(S_g(\tau_{[\beta]}))$。

对于 S_h^u，我们可以推导出 $dom(S_h^u) \subseteq mems(S_y(dom(S_g) \cap dom(S_y))) \cup mems(S_g(dom(S_g) \cap dom(S_y)))$，并且对于所有 $\tau_{[\beta]} \in dom(S_h^u)$，$S_y(\tau_{[\beta]})\ [S_h^u] = S_g(\tau_{[\beta]})\ [S_h^u]$。最终，我们得到 $\tilde{\omega}_g[S_h^{yg}][S_h^u] = \tilde{\omega}_y[S_y][S_h^{yg}][S_h^u]$。关于 S_e 的定义，根据证据我们知道 $\tilde{\omega}_e[S_e] := \tilde{\omega}_x^y[S_g][S_e]$。同样地，我们有 $mems(S_g(dom(S_g) \cap dom(S_y))) \cap (dom(S_y) \cup mems(\tilde{\omega}_x^y)) = \emptyset$，根据 unifyTrace 的定义，我们可以将 S_e 拆分为两个阶段，分别命名为 S_e^{gy} 和 S_e^u，其中 S_e^{gy} 可根据 S_y 定义如下。$S_e^{gy} = \forall \tau_{[\beta]} \in dom(S_y) \setminus dom(S_g).\ \tau_{[\beta]} \mapsto S_g(S_y(\tau_{[\beta]}))$。至于 S_e^u，我们可以推导出 $dom(S_e^u) \subseteq mems(S_y(dom(S_g) \cap dom(S_y))) \cup mems(S_g(dom(S_g) \cap dom(S_y)))$，并且对于所有 $\tau_{[\beta]} \in dom(S_e^u)$，$S_y(\tau_{[\beta]})\ [S_e^u] = S_g(\tau_{[\beta]})\ [S_e^u]$。最终，我们得到 $\tilde{\omega}_e[S_e^{gy}][S_e^u] = \tilde{\omega}_x^y[S_g][S_e^{gy}][S_e^u]$。现在我们需要证明 $\tilde{\omega}_h[S_h^{yg}][S_h^u] = \tilde{\omega}_h[S_g][S_e^{gy}][S_e^u]$。由于 $mems(\tilde{\omega}_h) \cap dom(S_y) = \emptyset$，可以将 S_h^{yg} 分解，从而得到 $\tilde{\omega}_h[S_g][S_y][S_h^u]$。另一方面，由于 $mems(\tilde{\omega}_h[\_][S_g]) \cap dom(S_g) = \emptyset$ 和 $mems(\tilde{\omega}_h[\_][S_y][S_e^u]) \cap dom(S_y) = \emptyset$，可以将 S 替换为 S，从而得到 $S\ S\ S$。最后，根据定义，S 和 S_e^u 相同，因此得到方程 $\tilde{\omega}_h[S_g][S_y][S_h^u] = \tilde{\omega}_h[S_g][S_y][S_h^u]$，即 $\tilde{\omega}_h[S_h] = \tilde{\omega}_h[S_g][S_e]$。

◦ 为了说明 $(\tilde{\Sigma}_g^{\uparrow}, (\tilde{\Sigma}_y^{\uparrow}, \tilde{\Sigma}_e^{\uparrow})\ [S_y])\ [S_h] = ((\tilde{\Sigma}_g^{\uparrow}, \tilde{\Sigma}_y^{\uparrow})\ [S_g], \tilde{\Sigma}_e^{\uparrow})\ [S_e]$，我们可以将方程分解为以下三个部分。$\tilde{\Sigma}_g^{\uparrow}[S_h] = \tilde{\Sigma}_g^{\uparrow}[S_g][S_e]$ 可以用与 $\tilde{\omega}_h[S_h] = \tilde{\omega}_h[S_g][S_e]$ 类似的方式进行分析。$\tilde{\Sigma}_y^{\uparrow}[S_y][S_h] = \tilde{\Sigma}_y^{\uparrow}[S_g][S_e]$ 可以写成 $\tilde{\Sigma}_y^{\uparrow}[S_y][S_h^{yg}][S_h^u] = \tilde{\Sigma}_y^{\uparrow}[S_g][S_e^{gy}][S_e^u]$。将 S_y 表示为 $S_y^g \circ S_g^y$，其中 $S_y^g = \forall \tau_{[\beta]} \in dom(S_y) \cap dom(S_g).\ \tau_{[\beta]} \mapsto S_y(\tau_{[\beta]})$，以及 $S_g^y = \forall \tau_{[\beta]} \in dom(S_y) \setminus dom(S_g).\ \tau_{[\beta]} \mapsto S_y(\tau_{[\beta]})$，并类似地将 S_g 表示为 $S_g^y \circ S_g^g$，我们得到 $\tilde{\Sigma}_y^{\uparrow}[S_g^y][S_y^g][S_h^{yg}][S_e^u]$。对于 S_h^{yg}，$dom(S_h^{yg}) \cap mems(\tilde{\Sigma}_y^{\uparrow}) = dom(S_h^{yg}) \cap mems(\tilde{\Sigma}_y^{\uparrow}[S_y])$。此外，对于所有 $\tau_{[\beta]} \in (dom(S_g) \setminus dom(S_y)) \cap mems(\tilde{\Sigma}_y^{\uparrow})$，存在 $S_y(S_g(\tau_{[\beta]})) = S_g(\tau_{[\beta]})$ 以及 $mems(S_g(\tau_{[\beta]})) \cap dom(S_y) = \emptyset$。因此，$S_h^{yg}$ 可以替换为 S_g^g，并且可以将其位置与 S_y^g 和 S_y^y 交换。对于 S_e^{gy} 也可以进行同样的处理，将其替换为 S_y^y。于是得到 $\tilde{\Sigma}_y^{\uparrow}[S_y^y][S_g^g][S_y^g][S_h^u] = \tilde{\Sigma}_y^{\uparrow}[S_y^y][S_g^g][S_g^y][S_e^u]$。根据 S_h^u 和 S_e^u 的定义，该方程成立。$\tilde{\Sigma}_e^{\uparrow}[S_y][S_h] = \tilde{\Sigma}_e^{\uparrow}[S_e]$ 可以写成 $\tilde{\Sigma}_e^{\uparrow}[S_y][S_h^{yg}][S_h^u] = \tilde{\Sigma}_e^{\uparrow}[S_e^{gy}][S_e^u]$，并且可以用与分析 $\tilde{\omega}_h[S_h] = \tilde{\omega}_h[S_g][S_e]$ 类似的方法进行分析。

∗ 对于因为在相应的 compTrace 中的轨迹迭代中，unifyTrace 函数返回 failTrace

◦ 如果 S_g 成功但 S_e 失败，S_y 可以

– **失败**。在这种情况下，不会存在相应的轨迹。– **成功**。在这种情况下，S_h 必须失败。如果 S_g 和 S_y 都成功，那么可以将 S_e^{gy} 定义为 S_e的一部分，而 S_e^u 的失败将成为 S_e 失败的唯一原因。如果 S_e^u 失败，与 S_e^u 共享相同定义的 S_h^u 也将失败。因此，依赖于 S_h^u 的 S_h必须失败。

◦ **如果**S_g 失败，那么 S_h 必然失败，因为如果统一 $\tilde{\omega}_g$ 和 $\tilde{\omega}_y$ 的 S_g 失败，那么统一 $\tilde{\omega}_g$ 和 $\tilde{\omega}_y$ [的 S_h S_h] 就不可能成功。

★ **当** x 在线性使用于 e_g中时，证明类似。

• 情形 $e_f = \mathsf{st}\ y \Rightarrow e_b$我们需要证明 $\Delta;\ \Gamma;\ \Phi^{\downarrow};\ \Sigma^{\downarrow} \vdash (\mathsf{st}\ y \Rightarrow e_b)[x \mapsto e] :\ \omega \dashv \Phi^{\uparrow};\ \Sigma^{\uparrow}$。根据线性性，我们知道 $y \neq x$、x 在线性使用于 e_b中，y也是如此。因此，通过对 T-S-Lam 进行逆变换，我们知道：$\Delta;\ \Gamma;\ \Phi^{\downarrow}$、$\Phi_x^{\downarrow}$、$\Phi_y^{\downarrow};\ \Sigma^{\downarrow}$、$x :\ \omega_x^{\downarrow}$、$y :\ \omega_y^{\downarrow} \vdash e_b :\ \omega_b \dashv \Phi_f^{\uparrow};\ \Sigma_f^{\uparrow}$、$x :\ \omega_x^{\uparrow}$、$y :\ \omega_y^{\uparrow}(\Phi^{\uparrow};\ \Sigma^{\uparrow};\ \omega) = \mathsf{compTrace}(\Phi_f^{\uparrow};\ \Sigma_f^{\uparrow};\ \omega_x^{\uparrow};\ \omega_y^{\uparrow} \Rightarrow \omega_b;\ \Phi_e^{\uparrow};\ \Sigma_e^{\uparrow};\ \omega_e)$，并且根据归纳假设，我们知道：$(\Phi_b^{\uparrow};\ \Sigma_b^{\uparrow},\ y :\ \omega_y^b;\ \omega_b^{\uparrow}) = \mathsf{compTrace}(\Phi_f^{\uparrow};\ \Sigma_f^{\uparrow},\ y :\ \omega_y^{\uparrow};\ \omega_x^{\uparrow};\ \omega_b;\ \Phi_e^{\uparrow};\ \Sigma_e^{\uparrow};\ \omega_e)\Delta;\ \Gamma;\ \Phi^{\downarrow}、\Phi_y^{\downarrow};\ \Sigma^{\downarrow},\ y :\ \omega_y^{\downarrow} \vdash e_b[x \mapsto e] :\ \omega_b^{\uparrow} \dashv \Phi_b^{\uparrow};\ \Sigma_b^{\uparrow}$、$y :\ \omega_y^b$根据 compTrace 的定义，我们知道 $\Phi_b^{\uparrow} = \Phi^{\uparrow}$、$\Sigma_b^{\uparrow} = \Sigma^{\uparrow}$，并且 $(\omega_y^b \Rightarrow \omega_b^{\uparrow}) = \omega$。然后根据 T-S-Lam，我们知道 $\Delta;\ \Gamma;\ \Phi^{\downarrow};\ \Sigma^{\downarrow} \vdash \mathsf{st}\ y \Rightarrow e_b[x \mapsto e] :\ \omega_y^b \Rightarrow \omega_b^{\uparrow} \dashv \Phi_b^{\uparrow};\ \Sigma_b^{\uparrow}$，由此得到证明目标。

□

引理 C.1（序列归约）。若 $\Delta;\ \Gamma \vdash (\mathsf{seq} \Leftarrow e_m \Leftarrow e_j) \leftarrow e_i :\ \varphi_r \blacktriangleright \tau_r \dashv \Phi_r$，则 $\Delta;\ \Gamma \vdash e_j \leftarrow (e_m \leftarrow e_i) :\ \varphi_r \blacktriangleright \tau_r \dashv \Phi_r$。

引理 C.1 的证明。对 **T-Seq**、**T-S-App**和**T-Exec**进行逆变换，可知：$\Delta; \Gamma \vdash e_m : \tau_m \xrightarrow{\varphi_m} \tau_n \dashv \Phi_m \quad \Delta; \Gamma \vdash e_j : \tau_j \xrightarrow{\varphi_j} \tau_k \dashv \Phi_j \quad \Delta; \Gamma \vdash e_i :\ \varphi_i \blacktriangleright \tau_i \dashv \varphi_i \rhd []$**擦除**$(\tau_n)$ = **擦除**(τ_j) **擦除**(τ_m) = **擦除**(τ_i) **擦除**(τ_k) = **擦除**(τ_r)
计算的**compTrace** 完整展开过于复杂，无法展示。作为替代，由于枚举过程中**compTrace** 中的操作对每条轨迹都是统一的，因此证明时只需考虑单条轨迹（忽略重命名的差异）。我们用名称上的波浪号（例如 $\tilde{\tau}_m$）表示只考虑单条轨迹。

• 对于 Φ_r 中存在的每条轨迹，我们有$\tilde{\tau}_n[S_{jn}] := \tilde{\tau}_j[S_{jn}]$ $\tilde{\tau}_m[S_{jn}][S_{imjn}] := \tilde{\tau}_i$ $\tilde{\tau}_k[S_{jn}][S_{imjn}] = \tilde{\tau}_r$我们在某些方程中写作 $:=$，表示该方程同时定义了一个代换。例如，$\tilde{\tau}_n[S_{jn}] := \tilde{\tau}_j[S_{jn}]$表示，$S_{jn}$ 由使用 unifyTrace 对 $\tilde{\tau}_n$ 和 $\tilde{\tau}_j$ 进行轨迹合一所定义。根据已知证据，必定存在一个 S_{im}，使得 $\tilde{\tau}_m[S_{im}] := \tilde{\tau}_i$，因为如果我们有 $\tilde{\tau}_m[S_{jn}][S_{imjn}] := \tilde{\tau}_i$，根据 unifyTrace 的定义，我们必须能够得到这样的 S_{im}。此外，还必定存在另一个 S_{nimj}，使得 $\tilde{\tau}_j[S_{nimj}] := \tilde{\tau}_n[S_{im}]$，因为我们已经知道 $\tilde{\tau}_m[S_{jn}][S_{imjn}] = \tilde{\tau}_i = \tilde{\tau}_m[S_{im}]$，并且可以得到$\tilde{\tau}_n[S_{jn}][S_{imjn}] = \tilde{\tau}_n[S_{im}]$，因为 $mems(\tilde{\tau}_n) \subseteq mems(\tilde{\tau}_m)$，然后将 S_{imjn} 应用于$\tilde{\tau}_n[S_{jn}] = \tilde{\tau}_j[S_{jn}]$的两边，我们得到 $\tilde{\tau}_n[S_{jn}][S_{imjn}] = \tilde{\tau}_j[S_{jn}][S_{imjn}]$，再根据传递性得到$\tilde{\tau}_j[S_{jn}][S_{imjn}] = \tilde{\tau}_j[S_{nimj}]$，最后因为 $mems(\tilde{\tau}_k) \subseteq mems(\tilde{\tau}_j)$，我们得到 $\tilde{\tau}_k[S_{jn}][S_{imjn}] = \tilde{\tau}_k[S_{nimj}]$。这也给出了我们的证明目标，即 $\tilde{\tau}_k[S_{nimj}] = \tilde{\tau}_r$。

• **对于** $\Phi^{\uparrow}$ 中不存在的每条轨迹，因为在相应的 **compTrace** 迭代中，**unifyTrace**函数返回 **failTrace**。★ **如果**S_{jn}成功而 S_{imjn} 失败，则 S_{im} 可以

∗ **失败**。在这种情况下，不存在对应的轨迹。
∗ **成功**。仅当 S_{imjn}失败时，这种情况才可能发生，因为对于每个 $\tau_{[\beta]} \in mems(\tilde{\tau}_m)$，$S_{jn}(\tau_{[\beta]})$ 都无法与 $S_{im}(\tau_{[\beta]})$ 统一。在这种情况下，S_{nimj}也必定失败，因为$mems(\tilde{\tau}_n) \subseteq mems(\tilde{\tau}_m)$。★ **如果** S_{jn} 失败，那么 S_{im} 可以

∗ **失败**。在这种情况下，不存在对应的轨迹。
∗ **成功**。在这种情况下，S_{nimj} 必须失败，因为如果统一 $\tilde{\tau}_n$ 和 $\tilde{\tau}_j$ 的 S_{jn} 失败，那么统一 S_{im}]、$\tilde{\tau}_j$ 和 $\tilde{\tau}_n$[的 S_{nimj} 就不可能成功。

□

引理 3.14（成功规则）。如果 $e_r = (\text{规则}\ p \to e) \leftarrow \text{成功}\ v$，存在 $\Delta;\ \Gamma \vdash e_r :\ \varphi_r \blacktriangleright \tau_r \dashv \varphi_r \rhd []$ 并且 $\Delta;\ \Gamma \nvdash e_r :\ [] \blacktriangleright \tau$，则 $e_r \rightarrowtail^{+} \text{成功}\ v_s$。

引理 3.14 的证明。通过对 T-Exec 和 T-R-Lam 进行逆变换，我们知道 $\Delta \vDash_{\alpha_f} p :\ \tau_p \dashv \Phi;\ \Theta$，以及 $\Delta;\ \Gamma;\ \Phi;\ \Theta \vdash e :\ \tau_e$，还有 $\Delta;\ \Gamma;\ \alpha_v \rhd [];\ \bigstar \vdash v :\ \tau_v$，并且 $\text{erase}(\tau_p) = \text{erase}(\tau_v)$，且存在一个由 $\text{unifyTrace}(\alpha_f \rhd \phi_f;\ [\alpha_f] \blacktriangleright \tau_p;\ [\alpha_f] \blacktriangleright (\tau_v[\alpha_v \mapsto \alpha_f]))$ 生成的 $S \neq \text{failTrace}$，因为$\Delta;\ \Gamma \nvdash e_r :\ [] \blacktriangleright \tau$。根据对 $\Delta \vDash_{\alpha_f} p :\ \tau_p \dashv \Phi;\ \Theta$ 的类型推导进行归纳，并结合 unifyTrace 的定义，我们知道 p 和 v 中不存在不匹配标签。根据 match 的定义可知，不匹配标签不仅是失败模式的来源，也是唯一的来源。因此，$\text{match}(p;\ v)$ 必然成功，并且 $e_r \rightarrowtail \text{succ}\ v_s$。□

引理 *C.2*（枚举）。如果 $\Delta; \Gamma \vdash v_f: \tau_p \xrightarrow{\varphi_f} \tau_e \dashv \Phi_f$且 $\Delta; \Gamma \vdash$成功 $v_i: [\alpha_i] \blacktriangleright \tau_i \dashv \alpha_i \rhd []$且擦除$(\tau_p)$ =擦除(τ_i)，并且 $(\varphi_r \rhd [];\cdot;\ \varphi_r \blacktriangleright \tau_r)$ $= compTrace(\Phi_f;\cdot;\ \varphi_f \blacktriangleright \tau_p;\ \varphi_f \blacktriangleright \tau_e;\ \alpha_i \rhd [];\cdot;\ [\alpha_i] \blacktriangleright \tau_i)$，其中 $\varphi_r \neq []$，那么对于每个 α_r属于 φ_r，都存在一个归约 $(v_f \leftarrow (\text{成功}\ v_i)) \rightarrowtail^{+}$成功 v_r；对于 $(\alpha_r \rhd [];\cdot;\ [\alpha_r] \blacktriangleright \tau' r)$ =选择 $[\alpha_r]$属于 $(\varphi_r \rhd [];\cdot;\ \varphi_r \blacktriangleright \tau_r)$，则有$\Delta;\ \Gamma;\ \alpha_r \rhd [];\ \bigstar \vdash v_r :\ \tau'_r$。

引理 C.2 的证明。证明对 v_f 进行归纳。需要考虑三种情况：v_f =（规则 $p \to e$）、v_f =（seq $\Leftarrow v_a \Leftarrow v_b$）以及 v_f =（choice $\Leftarrow v_a \Leftarrow v_b$）。规则情况由引理 3.14 证明。通过应用归纳假设，可以很容易地证明 choice 情况。通过应用归纳假设和引理 C.1，可以证明 seq 情况。□