Mason：类型和名称引导的程序合成

Jasper Geer不列颠哥伦比亚大学 温哥华，加拿大
jasper.geer@ubc.ca

Fox Huston 塔夫茨大学 梅德福，美国
hello@fox.boston

Jeffrey S. Foster 塔夫茨大学 梅德福，美国 j
effrey.foster@tufts.edu

arXiv:2602.16981v1 [cs.PL] 19 Feb 2026

摘要

面向对象程序通常采用许多常见的编码惯用法来编写，例如设计模式所概括的那些惯用法。设计模式虽然有用，但实现起来往往繁琐且重复，需要编写样板代码，使程序员难以专注于更重要的细节。本文介绍了一种名为 Mason 的工具，它能够从部分程序片段合成面向对象程序；我们还利用它自动将设计模式插入程序。

Mason 的核心是一种新颖的类型和名称引导的合成技术：枚举求解器遍历部分程序以生成类型约束；根据受约束类型和成员的名称，通过程序变换消解约束；并在约束遭到违反或候选程序未能通过单元测试时进行回溯。我们还为 Mason 引入了两项扩展：一种利用执行轨迹的非局部回溯启发式，以及一种对缺失名称施加语法限制的模式语言。我们在一组基准测试上评估 Mason，要求它将各种知名设计模式添加到程序中，这些设计模式以程序片段库的形式实现。我们发现，当满足其类型约束的候选程序极少时，Mason 表现良好；而在情况并非如此时，我们的扩展能够显著提升 Mason 的性能。我们认为，Mason 在利用设计模式合成多类面向对象程序方面迈出了重要一步。

1 引言

面向对象编程语言的使用者经常会采用设计模式，即软件工程中常见惯用法的解决方案 [5]。设计模式有助于构建更易于维护的软件，也能促进软件工程师之间的交流。然而，许多设计模式要求程序员先编写大量乏味的样板代码，才能着手处理关键细节。

解决这一问题的一种潜在方案是程序合成，它旨在通过根据用户提供的规范自动生成代码，提高程序员的工作效率。此前，面向对象语言中的合成工作能够利用轻量级形式规范，在 Ruby 中合成方法 [7, 8]，合成程序通过草图式编程 [17] 在 Java 中 [11, 13]，并为程序分析生成模拟库实现 [10] 然而，这类技术要么高度依赖特定领域，要么合成底层实现细节，同时要求程序员提供样板代码。

本文介绍了 Mason，这是一种用于合成面向对象程序的新工具，并展示了如何使用它自动将设计模式插入程序中。Mason 的关键创新在于，它能够在类型和名称的共同引导下，根据部分程序片段合成面向对象程序。在 Mason 中，程序员提供一个部分候选程序、一组必须通过的单元测试，以及一个用于描述设计模式的部分程序片段库。我们将这些程序片段称为片段，它们可以包含空位，以代替类型或成员名称。Mason 首先遍历候选程序以生成类型约束。随后，Mason 通过以下两种方式之一求解类型约束：填充空洞，使受约束类型或成员的名称与候选程序中的名称合一；或者通过将片段合并到候选程序中，为缺失的类型或成员合成定义。每次转换后，Mason 都会重新检查约束；如果发现矛盾，则回溯。解决所有目标后，Mason 会运行候选程序；如果任何单元测试失败，则会回溯。

我们针对一种核心面向对象语言 L_M，将 Mason 形式化；该语言允许用空洞替代类型和成员名称。我们分两部分定义 L_M 的类型推断算法：首先，定义一个语法制导过程，从 L_M 程序中生成类型约束；其次，定义一个约束重写系统，在候选程序的上下文中检查并简化约束。程序可能是不完整的，因此某些约束可能因约束未定义的类型或成员而无法满足；我们称这类约束为挂起约束。随后，我们定义类型和名称引导的合成 规则，通过组合片段并用具体名称填充空位来解决挂起约束。

我们在 Scala 中实现了 Mason，将其作为一种能够接受带有空位的部分 Java 程序的工具。我们通过对挂起约束施加顺序，将非确定性合成规则改造为搜索过程。我们还实现了两种扩展，用于剪枝那些无法通过类型约束排除的候选项：轨迹引导的回溯 是一种非局部回溯启发式，它利用失败单元测试的执行轨迹和名称模式；名称模式可用于

作者联系方式：Jasper Geer，不列颠哥伦比亚大学，温哥华，加拿大，jasper.geer@ubc.ca；FoxHuston，塔夫茨大学，梅德福，美国，hello@fox.boston；Jeffrey S.Foster，塔夫茨大学，梅德福，美国，jeffrey.foster@tufts.edu。

```
类 Canvas继承 对象 {
  整数 brushX;
  整数 brushY;
  图像 image;
  . . . // 其他字段
  Canvas(...) { ... }
  图像 当前图像() { ... }
  void 开始绘图() { ... }
  void 停止绘图() { ... }
  void moveBrush(整数 x,    整数 y) {
    . . .
    此对象.setBrushX(此对象.brushX + x);
    此对象.setBrushY(此对象.brushY + y);
    . . .
} }
```

```
类 CanvasLogger {
  List log = 新建 ArrayList();
  // 状态变更处理程序
  void onUpdateX(整数 newX) {
    这个.log.add("x:
      .concat(newX.toString()));
  }
  void onUpdateY(Integer 新 Y 坐标) {
    此.log.add("y:
      .concat(newY.toString()));
  }
  // 其他方法
  getLogEntry(整数 i) {
    返回 this.log.get(i);
} }
```

图1. 输入Mason的部分程序

为各空位添加约束，限定可填入其中的名称集合。

我们评估Mason 及其扩展，使用我们开发的一套基准测试，其中Mason 必须应用四人帮 [5] 设计模式，这些设计模式以片段库的形式实现。每个基准测试均可相对于一个自然数进行规模扩展，我们测量Mason在规模不断增大时找到正确解所需的时间。我们发现，基线Mason在良类型可综合程序数量保持较低的基准测试中表现良好。对于其余基准测试中除一个以外的所有基准测试，跟踪引导的回溯使Mason 能够在超时之前综合出更大的程序，并且名称模式将两个基准测试的时间从120秒以上降至6秒以下。

我们相信，Mason在使用设计模式综合多类面向对象程序方面向前迈出了重要一步。

2 动机示例

现在，我们使用 Mason 通过实现观察者 [5]设计模式来补全一个不完整的 Java 程序。Mason 获得一个初始程序片段以及一个程序片段库，并在类型、名称和单元测试的指导下组合出解决方案。

图 1 展示了Canvas 和CanvasLogger的定义。Canvas 是一个类，允许客户端使用虚拟画笔操作图像。CanvasLogger 的实例使用两个处理程序记录该画笔的移动： onUpdateX 和onUpdateY。我们还要求单个 Canvas 能够通知多个订阅的CanvasLogger。这是观察者设计模式所解决的一种常见依赖关系 [5] 。

在观察者模式中，主题对象会带有一组观察者。每当状态发生有意义的变化时，主题都负责通知它的每个观察者。在我们打算合成的模式版本中，主题的某些状态变更方法将负责通知所有已订阅的观察者状态发生了变化。

具体来说，我们设想为 brushX 和 brushY 各提供一个特殊的 setter方法，并提供一个将新的 CanvasLogger 注册到 Canvas 的方法。为了向 Mason 传达这一意图，我们提供一个<片段库>，其中包含一组程序片段，Mason 可据此推导出新的类型和方法声明；此外还提供一组单元测试，合成的程序必须通过这些测试。

片段。其中一个程序片段如图2所示。@MemberFragment 注解表明，我们希望 Mason 使用此片段合成成员实现。第 2、3、5、4、6 和 7 行中以 ? 为前缀的标识符是空位，即 Mason 预期会填入的句法元变量。空位指出 Mason 可以调整哪些细节，以便将片段适配到新的上下文中。例如，类名 ?Update 将填入一个现有类的名称，并将它们的声明合并。根据方法定义的结构，我们还可以看出，此片段将用于合成一个 setter方法：使用某个类型 ?T 的新值更新某个字段 ?field，并在每个 ?Observer 上调用某个处理程序 ?notify，以通知其发生了更改。将 newValue 传递给 ?notify，体现了这种设计模式形态的一个关键细节：主体中的每个状态变更方法都对应于观察者的一个特定处理程序。

```
@MemberFragment
类?Update {
  void ?{(set)(?字段)}(?T newValue) {
    此.?field = 新值;
    对于 (int i = 0; i < 此.?{(num)(?)}(); i++) {
      ?观察者o = this.?{(get)(?)}(i));
      o.?通知(newValue);
}}}
```

图2。 提供给 Mason 的片段库示例元素。空洞以 ? 开始，并以**红色**显示。

约束生成。Mason遍历初始片段，收集类型约束。请注意，图1中的 Canvas 实现 并不完整。具体来说，moveBrush的函数体引用了未定义的方法setBrushX 和 setBrushY。然而，Mason 不会报告失败，而是发出两个暂挂的类型约束：

$$⟦\texttt{Canvas} \leq [\texttt{setBrushX} : (\texttt{Integer}) \rightarrow \alpha]⟧$$
$$⟦\texttt{Canvas} \leq [\texttt{setBrushY} : (\texttt{Integer}) \rightarrow \beta]⟧$$

通俗地说，只有当Canvas 定义了一个从Integer 到某种类型的方法，且该类型由类型变量表示时，该约束才算满足 α 和 β。分隔符⟦_⟧ 表示，虽然这两个约束都尚未满足，但各自都包含某个未定义部分，我们可以在后续步骤中合成该部分。在本例中，setBrushX和setBrushY 都尚未定义。我们发现，这类约束对应于合成目标；在生成可执行程序之前，我们必须分别为setBrushX 和setBrushY 合成实现。Mason 会在片段库中搜索元数相同且类型兼容的方法，例如可以选择图2中的片段。

请注意，方法名称 ?{(set)(?field)} 的语法独特。为了向 Mason 提供更详细的规范，我们采用了命名约定，并使用了名称模式。此模式要求，任何用于填充空洞的名称（例如 setBrushX）都必须由 set 加上 ?field 所对应的名称组成，大小写不限。不过，?field 目前尚未确定。为确保填入 ?field 的名称与该模式一致，Mason 会生成一个名称约束：

$$\texttt{setBrushX} \dashleftarrow (\texttt{set})(?\texttt{field})$$

约束的左操作数是一个名称，右操作数是一个名称模式。若其左操作数为具体值且与右侧的模式匹配，则该约束得到满足；像?field 这样的空位被视为通配符。我们将在第4.2节详细描述名称模式。Mason 并不将名称约束用作合成目标，但它们可以非常有效地剪枝搜索空间，我们稍后将对此进行说明。

合成。 Mason 通过深度优先回溯搜索，查找满足所有类型约束和名称约束的程序，如图 3所示。搜索空间中的每个元素都由一个候选程序和一组约束构成。Mason 首先执行约束求解，在候选程序的上下文中重写约束。约束求解有两个目的：第一，对部分程序进行类型检查；如果后续步骤不可能得到类型正确的程序，则检查失败。第二，更新 Mason的合成目标：消除所有已解决的约束，并挂起所有符合条件的约束。如果约束求解失败，Mason 就会回溯。我们可以在s2中看到这一点：字段 image 的类型Image 被赋值为newX ，其类型为整数；约束求解失败，Mason 于是回溯到s0。如果约束求解成功，Mason 会选择一个待求解的挂起约束，并通过应用综合规则来探索可能的解；这些规则通过填充空洞或合成定义来生成新的候选程序。

假设Mason已将图2中的片段实例化为set BrushX和setBrushY二者的函数体，并正确地将?观察者映射到CanvasLogger、将?{(num)(?)}映射到numLoggers、将?{(get)(?)}映射到getLogger。该状态对应于图3中的s0。Mason首先探索 $s1$，在setBrushX的函数体中用brushY填充?field，以求解一个挂起约束（Canvas $\leq$ [?field : γ]）。然而，名称约束会随程序的其余部分一起被重命名，而这一替换将前述约束变换为setBrushX $\dashleftarrow$ (set)(brushY)，从而导致约束求解失败，使Mason拒绝s1。对s2以及一系列被省略的候选，约束求解同样失败。最终，Mason到达唯一有效的选择brushX，我们便来到s3。此处同样只有一个选择有效，我们必然到达s4。

轨迹引导的回溯。 当静态信息无法充分缩小搜索空间时，Mason 会依赖单元测试提供的动态信息，执行非局部回溯。下面是这样一个测试，我们提供给 Mason 以指定解决方案的行为。参见图1，只有在 setBrushX 调用onUpdateX时，该测试才会通过。

```
1  @Test2  静态 voidtestUpdateX() {3
CanvasLogger cl =新建CanvasLogger(); 4
Canvas c =新建Canvas(...); 5
c.registerLogger(cl); 6  assert(c.getLogger(0) ==
cl); 7  assert(c.numLoggers() == 1) 8
c.setBrushX(10); 9  assert(cl.log.contains("x: 10"));
10 }
```

有三种类型正确的选项可以填入 ?notify：onUpdateX、onUpdateY 和 getLogEntry。假设 Mason 选错了，选择 getLogEntry 并到达 s5。尽管 setBrushY 的函数体已正确完成，s6 仍未通过测试 testUpdateX。在朴素的回溯策略下，Mason 可能会尝试用 s7 和 s8 实现不同的 setBrushY，但这徒劳无功；在执行 testUpdateX 期间，setBrushY 从未被调用。

Mason 将这一思路机制化为轨迹引导的回溯。对于每个综合步骤，Mason 都会计算该步骤可能影响其实现的方法的干扰集。然后，对于每项测试，Mason 会跟踪一个依赖集，其中包含测试结果可能依赖的方法。如果某个综合步骤的干扰集与某个单元测试的依赖集相交，那么 Mason 就确定该综合步骤可能会影响该测试的结果。

合成出完整且类型正确的候选程序后，Mason 会按顺序执行单元测试，以发现依赖关系，并在遇到第一个失败的测试时提前停止。Mason

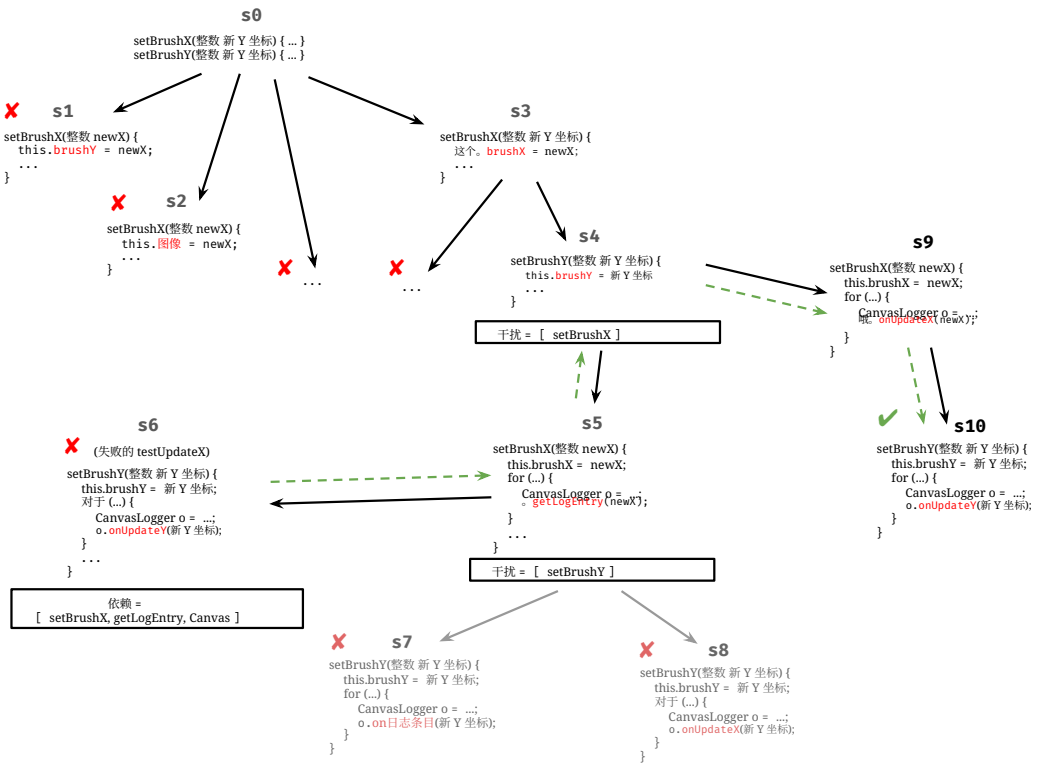

图3. Mason对正确观察者模式实现的搜索。节点表示综合步骤，边表示在给定综合步骤中所做的特定选择。每个节点都标注了执行该综合步骤之前的候选程序，所选步骤的干扰集或依赖集显示在下方框中。标注有 ✘ 的叶节点是因约束求解失败或单元测试失败而被拒绝的候选。灰色节点和边是跟踪引导的回溯未探索的路径。绿色虚线箭头追踪轨迹引导下的搜索路径。

会记录执行最后一个单元测试期间调用的方法集合，即其执行轨迹，并将这些方法加入该测试的依赖集。随后，Mason 会进行非局部回溯，直到到达某个综合步骤，其干扰集与失败测试的依赖集相交；接着选择一个尚未探索的候选项，并重放此前路径中能够重放的选择。

我们在图3中观察到这一点。由于在s4的出边处所做的选择会影响或干扰setBrushX的实现，因此将其加入s4的干扰集（见第4.2 节中干扰的定义）。同样，s5 的干扰集包含setBrushY。在s6处执行程序后，testUpdateX 失败，并以setBrushX、getLogEntry和 Canvas 作为依赖，因为这些是失败之前所做的函数调用。

回溯到 s5 后，Mason 再次回溯，跳过 s7 和 s8，因为 s5 只会干扰 setBrushY，而 testUpdateX 并不依赖它。到达 s4 后，Mason 正确地选择 onUpdateX，并重放 (s5，s6)，从而找到解 s10。通过这种方式，Mason 只重新评估可能导致测试失败的选择，因而剪去了搜索空间中的大片区域。在这个场景中，从 s4 出发可到达 3^2 个可执行候选项。使用跟踪引导的回溯，我们最多访问 3×2。

3 形式化

在本节中，我们定义一种面向对象核心语言$\mathcal{L}_M$，以形式化类型和名称引导的程序合成。图 4 的上半部分展示了 $\mathcal{L}_M$的语法。这里，记号 a 表示零个或多个 a的序列。程序由类型声明 d组成，每个声明定义一个类或接口。类包含方法 m 和字段 f，而接口包含方法签名 s。每个类都有一个父类，并实现零个或多个接口。

L_M 方法体由单个表达式 e组成，该表达式可以是变量引用 x、字段访问 $e.M$、方法调用 $e.M(e)$ 或类型转换 $(A)\ e$。我们省略了赋值和顺序组合等形式，但将系统扩展以包含这些特性是直接的。类型名称和方法名称可以分别是类和方法的具体名称 **A** 与**M**，也可以是空位 **?A** 和**?M**，这些空位会在合成过程中被具体名称替换。

$$
\begin{array}{rcl}
P & ::= & \overline{d} \\
d & ::= & \textbf{类}A\ \textbf{扩展}\ A\ \textbf{实现}\ A\ \{\ f\ m\ \} \\
 & \mid & \textbf{接口}\ A\ \{\ s\ \} \\
m & ::= & A\ M(A\ x)\ \{\ e\ \} \\
s & ::= & A\ M(A\ x) \\
f & ::= & A\ M \\
e & ::= & x \mid e.M \mid e.M(e) \mid (A)e \\
A & ::= & \mathbf{A} \mid \mathbf{?A} \\
M & ::= & \mathbf{M} \mid \mathbf{?M}
\end{array}
$$

$$
\begin{array}{rcl}
t & ::= & A \mid \alpha \mid t \to t \\
C & ::= & C \wedge C \mid t \le t \mid t \le [M\colon\ t] \mid \text{实现}(A, A) \\
 & \mid & \text{存在}(A) \mid (\!|C|\!)
\end{array}
$$

图4. L_M 语法、类型与约束。

类型约束。 类型检查 $\mathcal{L}_M$ 程序的合成包括生成和求解类型约束，其语法见图4的下半部分。约束对定义的结构和继承图作出断言。

类型 t 可以是类型名称 A、变量 α或方法类型 $t \to t$。约束可由 $\wedge$合取，并有多种形式。如果t_1 继承自 t_2 或实现 t_2（可能通过继承实现），则满足名义子类型 约束 $t_1 \le t_2$。当程序中包含 A 的类或接口定义时，存在存在量化 约束 exists(A) 成立。最后，我们有两个结构子类型 约束。如果 t定义或继承了类型为t_M的成员 M，则约束$t \le [M\colon\ t_M]$ 成立。如果 A_1 为 A_2 的每个成员定义或继承了同名且类型兼容的成员，则约束 implements(A_1, A_2) 成立。

最后，约束也可以被挂起，记作 $(\!|C|\!)$。如果约束因引用了某个未定义的类型或成员而无法满足，我们就会将其挂起。例如，如果当前不包含 **A** 的定义，我们可能会将 exists(**A**) 挂起为 $(\!|\text{exists}(\mathbf{A})|\!)$，如果 P。如下文第3.2节所述，在合成过程中，我们会在向程序中添加内容时恢复这类约束，使其有可能得到满足。我们的实现还支持名称约束，详见第4节。

3.1 $\mathcal{L}_M$的类型推断

我们的综合算法将类型推断用作子程序，以生成并求解类型与类型变量之间的约束。图 5 给出了 L_M 的类型推断规则。前几条规则为表达式定型。TVar 以通常方式为变量定型。TFld 为字段访问 $e.M$ 定型，方法是生成一条约束：对象的类型 t 必须包含一个类型为 α的字段 M，其中 α 是一个新鲜类型变量。TCast 为类型转换 (A)e 定型，方法是生成两条约束：A $\le t$ 要求A 是 t的子类型，而 exists(A) 要求 A 存在于

$$\textsc{TVar}\ \frac{}{\Gamma \vdash x\colon\ \Gamma(x)} \qquad \textsc{TFld}\ \frac{\Gamma \vdash e\colon\ t \qquad t \le [M\colon\ \alpha] \qquad \alpha\ \text{新鲜}}{\Gamma \vdash e.M\colon\ \alpha}$$

$$\text{类型转换}\ \frac{\Gamma \vdash e\colon\ t \qquad \text{一个} \le t \qquad \text{存在（一个）}}{\Gamma \vdash (\text{一个})\ e\colon\ \text{一个}}$$

$$\textsc{TCall}\ \frac{\Gamma \vdash e_0\colon t_0 \qquad \Gamma \vdash e\colon\ t \qquad t_0 \le [M\colon\ t \to \alpha] \qquad \alpha\ \text{新鲜}}{\Gamma \vdash e_0.M(e)\colon\ \alpha}$$

$$\textsc{TMethD}\ \frac{\text{此}\colon A_0, \text{父类}\colon \text{父类}(P,\ A_0)\ \ x\colon A_2 \vdash e\colon\ t \qquad t \le A_1 \qquad \text{存在}(A_1) \qquad \text{存在}(A_2)}{P;\ \text{甲}_0 \vdash \text{甲}_1\ M(\text{甲}_2\ x)\ \{e\}} \qquad \textsc{TFldD}\ \frac{\text{存在}(A)}{P \vdash A\ M}$$

$$\textsc{TCls}\ \frac{P \vdash \overline{f} \qquad P; A_0 \vdash \overline{m} \qquad \text{存在}(A_1) \qquad \overline{\text{存在}(A)} \qquad \text{实现}(A_0, A)}{P \vdash \textbf{类}A_0\ \textbf{继承}\ A_1\ \textbf{实现}\ A\ \{f\ m\}}$$

$$\textsc{TIntf}\ \frac{\overline{\text{存在}(A_{i0})} \qquad \overline{\text{存在}(A_{i1})}}{P \vdash \textbf{接口}\ A\ \{\ A_{i0}\ M_i\ \overline{(A_{i1}\ x_i)}\ \}}$$

图 5. L_M 的类型推断与约束生成。

程序。请注意，在更为典型的类型推断算法中，这两个条件会被立即检查，但在我们的系统中，由于程序可能是部分的，我们需要生成约束。最后，TCall 为方法调用 $e_0.M(e)$定型，方法是约束 t_0（即接收者的类型）包含一个方法，该方法接受一个类型为 t的参数（即 e的类型），并返回某个类型 α。

其余规则用于类型声明。TMethD 通过确保方法体 e 符合其规范来为方法定类型。请注意，此规则会为方法定义中提及的类型生成存在约束。TFldD 同样会为字段类型生成存在约束。TCls 通过为类定义中的字段和方法定类型来为类定义定类型；为其中提及的类生成存在约束（但不包括正在定义的 A_0,）；并生成实现约束，以确保 A_0 与其父类和接口兼容。请注意，这里我们稍微滥用了记号：用 exists(A) 表示约束集合 $\bigwedge \text{exists}(A_i)$ for$A_i \in A$，implements(A_0, A) 也同理。最后，TIntf 通过生成约束来为接口定类型，以确保接口中提及的所有类都存在。

约束求解。 类型推断生成约束后，约束求解会迭代重写这些约束，直到它们达到求解形式。在标准约束求解中，不可满足的约束总会导致类型推断失败。相比之下，在 Mason 中有两种可能：矛盾的约束永远无法变为可满足，因此会迫使综合算法回溯

$\boxed{P \vdash C \twoheadrightarrow C'}$

Nominals

$P \vdash C \wedge A \ \leq A$	$\twoheadrightarrow$	C		
$P \vdash C \wedge$ 一个 $\leq A'$	$\twoheadrightarrow$	$C \wedge ([A \ \leq A'])$	**如果**$A \notin P$	
$P \vdash C \wedge A \ \leq A'$	$\twoheadrightarrow$	$\bot$	**如果** $A \in$ 接口(P) 和 $A' \in$ 类(P)	
$P \vdash C \wedge$ 一项 $\leq A'$	$\twoheadrightarrow$	C	**如果**A 继承自或实现 A'	
$P \vdash C \wedge$ 一个 $\leq A'$	$\twoheadrightarrow$	$C \wedge ([A \ \leq A'])$	**如果**A 继承自或实现某个 A_i，其中 A_i 是一个空洞或 $A_i \notin P$	
$P \vdash C \wedge A \ \leq A'$	$\twoheadrightarrow$	$\bot$	否则	

Existentials

$P \vdash C \wedge$ 存在（A）	$\twoheadrightarrow$	C	**如果**$A \in P$
$P \vdash C \wedge$ 存在（A）	$\twoheadrightarrow$	$C \wedge$（存在（A））	**如果**$A \notin P$

Interfaces

$P \vdash C \wedge$ 实现(A,A')	$\twoheadrightarrow$	$C \wedge$ 实现$(A,A') \ \wedge A \ \leq [M: \ A \rightarrow A_r]$	**如果** $A' \in$ 接口(P) **其中**$(A'.M: \ A \rightarrow Ar) \ \in P$
$P \vdash C \wedge$ 实现(A,A')	$\twoheadrightarrow$	$\bot$	**如果** $A' \in$ 类(P)
$P \vdash C \wedge$ 实现$(A, \ A')$	$\twoheadrightarrow$	$C \wedge$ (实现$(A, \ A')$)	否则

Variables

$P \vdash C \wedge A \ \leq \alpha \wedge \alpha \leq t \ \twoheadrightarrow \ C \wedge A \ \leq \alpha \wedge \alpha \leq t \wedge A \ \leq t$

Methods

$P \vdash C \wedge A \ \leq [M: \ t \rightarrow t_r]$	$\twoheadrightarrow$	$C \wedge t \leq t' \wedge t'_r \leq t_r$	**如果** $(A.M: t' \rightarrow t'_r) \ \in P$
$P \vdash C \wedge$ 一个 $\leq [M: t \rightarrow t_r]$	$\twoheadrightarrow$	$C \wedge t \leq t' \wedge t'_r \leq t_r$	**if** $(A'.M: \ t' \rightarrow t'_r) \ \in P$ **其中** A 继承自 A'
$P \vdash C \wedge A \ \leq [M: \ t \rightarrow t_r]$	$\twoheadrightarrow$	$C \wedge \ (A \ \leq [M: \ t \rightarrow t_r])$	否则

图 6. 所选约束解析规则。

并尝试其他解。否则，如果这些约束在合成过程中可能变为可满足状态，就会将其挂起，并继续进行约束求解。

图 6 展示了我们所选的约束解析规则，并将其形式化为关系 $P \vdash C \twoheadrightarrow C'$，其含义是：给定程序 P，约束 C 被重写为 C'。我们将$\twoheadrightarrow^*$ 记为 $\twoheadrightarrow$的自反传递闭包。矛盾约束是那些被重写为 $\bot$的约束。请注意，我们在传递意义上使用术语实现；我们说A 实现 A'，如果 A **实现** A' 出现在源代码中，或者 A的某个超类实现 A'。

名义重写规则处理形如$A \ \leq A'$的约束。第一条规则处理A 与A'均位于 P 中且该约束可满足的情形，此时该约束被消去。下一条规则在A为接口且A' 为类时产生矛盾。接下来的两条规则在该约束当前不可满足、但继承链或实现链包含一个可能位于从A 到A'的路径上的空洞时挂起该约束，因为填充该空洞可能使此约束可满足。

存在量词重写规则要么消解 exists(A) 约束（如果 $A \in P$），要么挂起该约束，因为合成过程可能会将 A 添加到程序中。接口重写规则会对 implements(A,A') 约束进行推理。如果 A'是程序中的一个接口，那么我们会生成新的约束，要求将 A' 中的所有方法都包含在 A中。如果 A' 是一个类，那么就会产生矛盾。否则，我们会挂起该约束，直到将 A' 添加到程序中。

其余重写规则都是标准的。变量规则实现了传递闭包。

3.2 形式化合成

程序由片段合成，我们将片段定义为程序 P 与约束 C构成的二元组$P, \ C$。我们的目标是从$P, \ C$以及一组片段 F（一个片段库）推导出某个片段$P', \ C'$，其中 C 不包含挂起约束。

合成规则需要若干关键定义。替换 σ 是从空缺名称到具体类型和成员名称的映射。若 $\sigma = \{\mathbf{?A} \mapsto \mathbf{A}, \mathbf{?M} \mapsto \mathbf{M}\}$，则 $\sigma(P, \ C \ )$表示通过将 P 和 C中所有**?A** 替换为**A** 、并将**?M** 替换为**M** 所得到的程序。

我们在声明上定义偏二元运算符 $\oplus$ 。当 d_1 与 d_2 种类相同（类或接口）、扩展同一类型且不包含同名成员时，$d_1 \oplus d_2$ 有定义。结果包含 d_1 与 d_2的成员之并；若 d_1 与 d_2 是类，则该组合实现二者中任一方所实现的全部接口。我们将 $\oplus$ 提升到程序，方法是组合每一对同名声明；没有对应项的声明在结果中予以保留。最后，我们将 $\oplus$ 提升到片段：$P, C \ \oplus P', C' \ = P \oplus P', C \wedge C'$。

类型和名称引导的合成。 Mason 通过求解挂起约束来合成程序，其中使用类型和

$$\boxed{F \vdash \langle P,\ C\rangle \rightsquigarrow \langle P',\ C'\rangle}$$

Fill-Type

$$\frac{\begin{array}{c}A_1 \in P \qquad \sigma(A_0) = \sigma(A_1) \\ \langle P', C_r\rangle = \sigma(\langle P,\ C \wedge \mathsf{exists}(A_0)\rangle) \\ P' \vdash C_r \twoheadrightarrow^* C'\end{array}}{F \vdash \left\langle P,\ C \wedge [\![\mathsf{exists}(A_0)]\!]\right\rangle \rightsquigarrow \langle P',\ C'\rangle}$$

Fill-Member

$$\frac{\begin{array}{c}A' \in \mathsf{super}^*(P, A) \cup \{A\} \\ (A'.M_1) \in P \qquad \sigma(M_0) = \sigma(M_1) \\ \langle P', C_r\rangle = \langle P,\ C \wedge A \le [M_0 : t]\rangle \\ P' \vdash C_r \twoheadrightarrow^* C'\end{array}}{F \vdash \left\langle P,\ C \wedge [\![A \le [M_0 : t]]\!]\right\rangle \rightsquigarrow \langle P',\ C'\rangle}$$

Synth-Type

$$\frac{\begin{array}{c}\left\langle P_f,\ C_f\right\rangle \in F \qquad A_1 \in P_f \qquad \sigma(A_0) = \sigma(A_1) \\ \langle P', C_r\rangle = \sigma(\langle P,\ C \wedge \mathsf{exists}(A_0)\rangle) \oplus \sigma(\left\langle P_f,\ C_f\right\rangle) \\ P' \vdash C_r \twoheadrightarrow^* C'\end{array}}{F \vdash \left\langle P,\ C \wedge [\![\mathsf{exists}(A_0)]\!]\right\rangle \rightsquigarrow \langle P',\ C'\rangle}$$

Synth-Member

$$\frac{\begin{array}{c}A_0 \in P \qquad \left\langle P_f,\ C_f\right\rangle \in F \qquad (A_1.M_1 : t_f) \in P_f \\ A_{s0} = \mathsf{super}(P, A_0) \qquad A_{s1} = \mathsf{super}(P_f, A_1) \\ \sigma(M_0) = \sigma(M_1) \qquad \sigma(A_0) = \sigma(A_1) \qquad \sigma(A_{s0}) = \sigma(A_{s1}) \\ \langle P', C_r\rangle = \sigma(\langle P,\ C \wedge A_0 \le [M_0 : t]\rangle) \oplus \sigma(\left\langle P_f,\ C_f\right\rangle) \\ P' \vdash C_r \twoheadrightarrow^* C'\end{array}}{F \vdash \left\langle P,\ C \wedge [\![A_0 \le [M_0 : t]]\!]\right\rangle \rightsquigarrow \langle P',\ C'\rangle}$$

图7。 合成规则。

名称引导的综合，其规则见图7。规则证明形如 $F \vdash \langle P,\ C\rangle \rightsquigarrow \langle P',\ C'\rangle$ 的判断，表示$\langle P',\ C'\rangle$ 可以使用片段库 F 从$\langle P,\ C\rangle$ 。

每条合成规则都会匹配片段 $P,\ C$ 中的一个挂起约束；对 $P,\ C$ 应用某种转换，生成 $P',\ C_r$；通过在 P' 的上下文中解析 C_r 来对结果进行类型检查，从而生成 C'；并生成一个新的类型正确的片段 $P',\ C'$，其中的约束已被解除。Fill- 规则通过填充空洞来实现这一点，而 Synth- 规则则通过合成新的类型和成员来实现这一点。

规则 Fill-Type 通过将一个空洞与一个具体类型名称合一，来解除一个挂起的存在量词约束（exists(A_0)）。第一个名称 A_0, 在约束中被引用，而另一个名称 A_1, 必须作为类型声明的名称出现。该规则将 σ 定义为使 A_0 与 A_1 合一的替换；请注意，替换必须将空洞映射为具体名称。随后，它将 σ 应用于 $P,\ C$，从而创建一个程序，其中存在针对 $\sigma(A_0)$ 的定义。

规则 Fill-Member 通过将M_0 与A的某个成员名称合一，来解除一个挂起的结构化子类型约束（$A \le [M_0 : t]$）；该成员的类型与 t 兼容。由于 A 可以从其超类型继承成员，Fill-Member 会将A的超类型中的所有成员名称都作为候选项，与M_0合一。

规则 Synth-Type 通过合成新生成代码，来解除一个挂起的存在量化约束（exists(A_0)）。该规则从片段库 F 中选取一个片段 $\left\langle P_f,\ C_f\right\rangle$，该片段包含一个名为$A_1$ 的类型声明，并使用 σ 将其与$P,\ C$中的A_0 以及 $\left\langle P_f,\ C_f\right\rangle$ 中的A_0 合一。它将所得片段组合起来，生成P', C_r，其中现在存在 $\sigma(A_0)$ 的定义。

Synth-Member 规则通过向 A_0 添加一个新成员来解除一个挂起的结构子类型约束 ($A_0 \le [M_0 : t]$)，该成员的类型与 t 兼容。该规则从 F 中选择一个片段 $\left\langle P_f,\ C_f\right\rangle$，并从该片段中选择方法 $A_1.M_1$，通过片段组合将其添加到 A_0 中。为了使 A_0 和 A_1 能够合并并满足该约束，σ 会统一三对名称：方法名称 M_0 和 M_1；类型名称 A_0 和 A_1；以及两种类型 A_{s0} 和 A_{s1} 的超类型。将 σ 应用于这两个片段并将它们组合后，得到 $\sigma(A_0)$，其中包含 $\sigma(M_0)$ 的定义。

这些规则具有很强的非确定性；给定一组挂起的约束，这些规则可以按许多不同的顺序应用。在第 4 节中，我们将介绍 Mason 实现如何系统地应用这些规则，以产生一个搜索过程。

4 实现

Mason 使用约 5,000 行 Scala3 代码实现。它接收一个包含单元测试的初始片段，以及一个用于补全该初始片段的片段库作为输入。两者均使用经过修改的 Java 语法编写，该语法支持空位和名称模式。

初始片段不能包含任何空位，但可以引用尚不存在的类型和方法。我们将声明程序主方法的类指定为其测试框架。主方法可以包含任意数量的单元测试；这些测试是带有注解 @Test 的静态 void 方法。单元测试不得依赖其方法体之外执行的任何代码的结果，因为轨迹引导的回溯在回溯过程中依赖这一特性；此外，单元测试必须抛出终止执行异常以表示失败。

尽管在形式化定义中，片段是完整的程序，但在 Mason 的实现中，片段被限制为仅包含一个类型声明。虽然将片段作为完整程序进行类型检查很有用，但在表示我们选定的任何设计模式时，我们都没有发现多声明片段有什么用处。

我们还将片段分为类型片段和成员片段，也就是说，前者用于Mason 使用 Synth-Type 合成新类型，后者则用于Mason 使用Synth-Member合成新方法。我们在片段源代码中使用类型注解来标明这一区别：`@TypeFragment` 和`@MemberFragment`。这一区分可避免 Mason 在不恰当的上下文中应用片段，从而浪费时间。例如，在第 2节中，片段

? Update （见图2 ）旨在为现有类合成设置器方法。使用此片段合成一个仅包含单个成员的新类，会徒然扩大搜索空间，因此我们将？ Update 指定为成员片段。

最后，为避免不同综合步骤中引入的空位之间发生意外的别名，Mason 实现会对新合成代码中的空位重新命名。α

4.1 实际中的合成

本小节将介绍我们施加的限制，以便从图中描述的高度非确定性合成规则中恢复出搜索过程7。

回溯搜索。在解析初始片段和片段库并对二者进行类型检查后，Mason会在可合成候选程序的空间中执行深度优先回溯搜索。对于每个部分候选程序，它会选择一个尚未求解的存在量化约束或结构子类型约束，并尝试按照图中的规则合成一个新的候选程序7。请回想，我们使用两类规则来求解子类型约束：填充规则和合成规则。由于我们优先探索规模较小的候选程序，Mason 会先穷尽探索通过应用填充规则可达的候选程序空间，然后再以同样的方式探索合成规则。如果这里也找不到解，Mason 就会回溯到前一个候选程序。

约束排序。 选择约束的顺序会影响可综合程序的范围。如果可以按任意顺序处理挂起约束，Mason 可能会尝试解决一个当前尚无法解决的约束。例如，假设 Mason 选择了以下代码片段中的挂起约束 (Foo ≤ [f : Integer])：

```
1 类 Bar { 整数 f; }类 Foo 继承 ?
2 X { }
```

两条规则 Fill-Member 和 Synth-Member 都匹配此约束。这里的第一个选项是应用 Fill-Member，将字段名具体化，但 Foo 没有符合条件的字段。第二个选项是用 Synth-Member 合成一个新字段，但如果我们的片段库中没有兼容的成员片段呢？此处的解决方案是将 Foo 的父类 ?X 填为 Bar，从而通过继承使 f 可用。然而，Mason 会尝试前两种策略，然后回溯，错过这个解决方案。因此，只有当某个类继承了特定成员后，才可能解决某个约束。

这种依赖关系的变体也可能存在于两个结构子类型约束之间。考虑以下代码片段和约束 (Foo ≤ [f : 整数]) 与 (Bar ≤ [f : 整数])：

```
1 类Bar { Integer ?f; }2 类Foo继承
Bar { }
```

首先满足 onBar 上的约束会将空洞 ?f 具体化，并间接解决 Foo 上的约束。然而，鉴于不存在可用于合成新字段的片段，优先选择 Foo 上的约束会导致失败，因为对 Foo 所能做的任何操作都无法满足该约束。

第三个例子是，考虑这样一种情况：包含空洞的约束，只有在包含具体名称的约束得到满足后才能满足。考虑约束 (Foo ≤ [f : 整数]) 和 (Foo ≤ [?g : 整数])，以及以下代码片段：

```
1 类Bar { 2 整数 foo() {返回 this.?g }
3 整数 ?h;4 }
```

如果 Mason 先选择满足 (Foo ≤ [f : 整数])，我们就可以在这一步将 ?h 填为 f，然后通过将 ?g 填为 f 来满足 (Foo ≤[?g : 整数])。然而，按相反顺序处理这些约束会失败，因为我们的规则不允许我们合一 ?g 和 ?h。如果通过合成一个新字段来满足 (Foo ≤ [f : 整数])，也可能出现类似情况。

对于这三种依赖关系，我们制定了三条规则。首先，存在量化约束总是先于子类型约束得到满足。其次，关于超类型的约束总是先于关于子类的约束。第三，对于无法根据前两条规则比较先后顺序的约束对，不含空洞的约束总是先于含有空洞的约束。

因此，如果当前约束集中存在某个类型或成员的具体名称，Mason 会先合成其定义，再填充所有可由该定义解决的空洞。无论排序如何，只要某个片段中存在带有具体名称的类型或成员定义，就可以通过 Synth 规则使用该定义。不过，如果空洞的正确替代项仅存在于库片段中的某个约束里，Mason 可能会在引入该片段之前尝试填充空洞，从而无法解决它。因此，我们的约束排序保证，只要解中每个合成类型和成员的名称，要么作为具体引用出现在初始片段中，要么是库片段中某个定义的名称，Mason 就能找到一个解。

4.2 扩展

现在我们介绍 Mason 的两项扩展，旨在缓解组合爆炸，尤其是在 Mason 必须在多个类型正确、但运行时行为各不相同的选项中进行选择时。

名称模式。 Mason 中的合成Mason围绕着用具体名称填充空位展开，因此，我们可以施加一种命名约定，要求候选替换遵循该约定，从而大幅缩小搜索空间。如第 2 节所述，Mason 允许程序员使用名称模式来编码此类约定。

名称模式可以出现在源代码中，代替类型或成员名称，并由 ?{_} 括起来。它们使用一种

受限的正则表达式语法，并增加了以 ? 为前缀的空位。就本文而言，我们仅保留选择运算符 |、分组运算符 () 和隐式连接运算符。名称模式通过将每个以 ? 为前缀的字母数字序列替换为 .*，并将所有大写字母转换为对应的小写字母，编译为正则表达式。

对于源代码中出现的每个类型名称模式 ?{A} 或成员名称模式 ?{M}，Mason 都会生成相应的名称约束 $M \dashleftarrow \mathrm{M}$ 或 $A \dashleftarrow \mathrm{A}$，其中$M$、$A$ 是由 ?{_} 括起、被解释为类型或成员名称的完整序列。当名称约束左侧是一个符合右侧模式的具体名称时，该约束即得到满足。

名称模式中包含的空位会在合成过程中与片段的其余部分一起重命名。例如，应用于模式{?field ↦ 名称} 的替换(set)(?field) 会得到(set)(名称)。名称模式可以编码不同空位之间的句法关系，例如设置器的名称与其所修改字段的名称之间的关系，如第2节所示。

轨迹引导的回溯。Mason 实现了一种匿名局部回溯启发式，轨迹引导的回溯，我们已在第2节中简要介绍过。为实现这一点，Mason 会为每个执行过的综合步骤跟踪一个干扰集，并为每次运行的单元测试跟踪一个依赖集。

综合步骤的干扰集记录了其结果可能影响实现的方法集合。每个综合步骤开始时该集合为空，并在尝试合成规则时，加入这些规则所干扰的方法。

如果使用 Fill-Member 重命名空洞**?M**，它会干扰以下方法：

- 在重命名之前出现过 ?M 的任何成员
- 如果该步骤为类型 t中名为**?M** 的成员指定了具体名称，则会干扰任何具有空洞名称、且曾经存在于 t中的成员（无论其当前已存在，还是之后才被综合出来）。

第二项是必要的，因为为类型声明中的某个方法指定特定名称后，该名称将永远不能再指定给声明中当前存在的任何其他方法，也不能指定给未来添加到该声明中的任何方法。

如果使用 Synth-Member 在类型 t中综合方法**M** ，它会干扰以下内容：

- 重命名前曾出现过 **?M** 的任何成员
- 此步骤中合成的任何新成员
- 在 t、其子类或其超类型中曾经存在过的、具有空洞名称的任何成员

如果某个综合步骤应用了 Fill-Class 或 Synth-Class，干扰集就是程序中的所有方法。

单元测试的依赖集记录了其结果可能取决于哪些方法的正确性。任何单元测试失败时，都会将其执行过程中调用的方法加入该测试的依赖集。

我们进一步扩展依赖集，将 Fill-Member 重命名之前的所有方法名称也纳入其中。

当 Mason 遇到单元测试失败时，会更新该测试的依赖集，并进行非局部回溯，直到到达干扰集与依赖集相交的综合步骤。随后搜索恢复，且不再进行非局部回溯；不过，如果 Mason 遍历了所有出边仍未找到可执行程序，就会使用同一个依赖集恢复非局部回溯。在此期间，Mason 会优先采用合成上一个可执行程序时所做的选择。

为了修正某个单元测试 T_2的失败，Mason 可能会做出导致先前通过的测试 T_1 失败的更改。在这种情况下，我们会回溯并搜索新的解，力求同时通过T_1 和 T_2。Mason 必须修改会影响这两个测试的步骤中的选择，因此 T_1 的依赖集将变为 T_1 和 T_2的依赖集之并集。执行轨迹和轨迹引导的回溯照常进行。

5 评估

我们在一组涉及使用常见设计模式合成程序的基准测试上评估 Mason。我们在评估中旨在回答的问题如下：

RQ1 Mason 在多大程度上实现了其既定目标，即自动合成设计模式的实现？合成问题的哪些具体方面会导致 Mason 表现良好或不佳？RQ2 Mason 的两项扩展在多大程度上、在哪些情形下能够有效提升其性能？

5.1 基准测试

我们开发了自己的合成基准测试套件，旨在揭示 Mason 的优势和不足。在每个基准测试中，Mason 都会通过添加一个四人帮 [5] 设计模式来合成程序；我们已将该模式实现为一个片段库。每个基准测试的初始片段都包含一组单元测试，用于实例化类并检查设计模式是否得到了正确实现。例如，*Accessor* 基准测试中的单元测试会检查设置器是否修改了正确的字段，以及获取器能否正确观察到这些修改。每个基准测试的规模都可以通过缩放因子 n进行调整，以便我们评估基准测试规模增大时 Mason 的性能。举例来说，在规模为 32 时，Mason 合成了额外的 298 行代码（由 cloc [2]测得），从而完成了 *VirtualProxy*基准测试。下面我们将介绍各个基准测试。

- **Factory**。 初始片段由实现同一接口的 n 产品类组成。对于每个类，

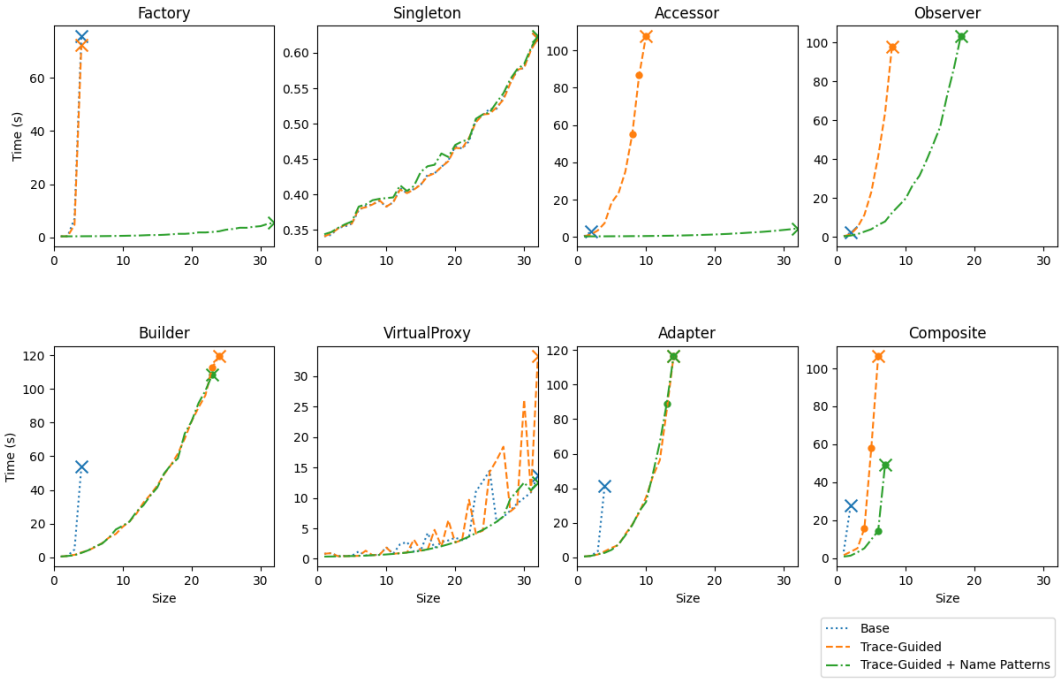

图 8. 我们 8 个基准测试中，每个基准测试进行 21 次试验所得的基准测试规模与中位墙钟时间。请注意，各子图的 y 轴刻度不同。当中位试验时间超过超时时间后，我们便停止绘制数据点。每组测试序列的最后一个数据点以 × 标记。至少有一次试验超时的运行以 • 标记。

Mason 必须合成一个具体工厂，用于实例化该类的对象。Mason 还必须合成一个抽象工厂接口，供具体工厂实现。片段库包含用于合成具体工厂类和抽象工厂接口的类型片段。

• **单例**。对于初始片段中的每个 n 类，Mason 都必须添加一个字段_instance，以及一个方法：如果_instance 尚未初始化，该方法就调用该类的构造函数将其初始化；否则返回其值。片段库包含一个同时包含该字段和方法的成员片段。

• **Accessor**。初始片段包含一个类，其中有n 个字段，且类型均相同。对于每个字段，Mason 都必须合成设置器和获取器。片段库包含一个用于合成设置器的成员片段，以及另一个用于合成获取器的成员片段。

• **观察者**。此基准测试将第 2 节中的示例推广了。除了在 n subject 类上合成设置器外，*subject* 类，Mason 还必须合成固定数量的辅助方法来管理观察者。片段库包含一个包含这些辅助方法的成员片段，以及图 2 所示的设置器片段。

• **建造者**。初始片段包含一个带有 n构造函数参数的类。Mason 合成一个具体的建造者用于这个类，它为每个参数提供一个设置器，并提供一个用于实例化该类的方法。片段库包含一个用于合成建造者的类片段、用于设置器的成员片段，以及用于实例化类的成员片段。

• **虚拟代理**。Mason 必须合成一个虚拟代理，它公开与给定类相同的接口（该类有 n 个方法），但会将调用转发给它延迟实例化的该类实例。片段库包含用于合成代理类的类型片段，以及用于代理方法的方法片段。

• **适配器**。Mason 必须合成一个适配器，它包装给定类的一个对象（该类有 n 个方法），同时实现一个不兼容的接口，并将方法调用转发给被包装的对象。片段库包含用于合成适配器类的类型片段，以及用于合成其方法的方法片段。

• **组合**。给定 $2n$ 定义树形结构的类，Mason 必须合成一个共享接口，并为每个类实现该接口，使客户端能够对叶节点和非叶节点执行相同的操作。片段库包含一个用于合成和填充共享接口的类型片段和成员片段，还包含一个具有“非叶节点”方法的成员片段，以及另一个具有“叶节点”方法的成员片段。

5.2 结果

图 8展示了在基准测试套件上运行 Mason 的结果。数据采集自一台配备 M2 Pro 处理器和 16GB 内存的 2023 年 MacBook Pro。基准测试本身运行在 JVM 上，其栈空间上限为 1GB，堆上限为 8GB。

对于每个基准测试，我们运行并绘制了三组测试。第一组是我们的基线，在图例中标记为基线，运行时未启用轨迹引导的回溯或名称约束。第二组标记为 轨迹引导，仅启用轨迹引导的回溯。第三组标记为 轨迹引导 + 名称模式，同时启用了这两项扩展。

在每组测试中，我们逐步提高基准测试的缩放因子，从 1 开始，每次递增 1，直至 32（含 32）。对于每个缩放因子，我们进行了 21 次试验，超时时间设为 120 秒，并记录所有单元测试通过之前用于合成和执行单元测试的总挂钟时间。随后，我们绘制了这些时间的中位数。如果 21 次试验中有多数发生超时，我们便不再对更大的缩放因子运行测试，也不绘制相应数值。

基线。 当类型正确的候选程序往往也是正确程序时，基线 Mason 的表现良好。单例 基准测试便是如此：每个类的特殊构造函数都使用单个成员片段进行合成，其两个空洞都会立即由 Synth-Member 填充。同样，对于*VirtualProxy*，类型正确的候选程序始终只有两种，其中一种会错误地将一个对象代理到其自身类的虚拟代理中。因此，单例的综合时间不超过 0.7 秒，而 VirtualProxy 的综合时间不超过 15 秒。

对于其余基准测试，规模增大会大幅增加类型正确且可综合的程序数量，进而导致性能下降。例如，在*Accessor*基准测试中，当规模因子为三时，基线 Mason 在大多数试验中都会超时。考察该基准测试便不难理解原因：当*Accessor* 的规模因子为三时，Mason 会为三个设置器方法和三个获取器方法分别选择要访问的正确字段，总计有 $3^3 \times 3^3$，即 729 种可能的程序。

使用跟踪引导的回溯。 在加入跟踪引导的回溯后，Mason 能够在超时时间内以更大的规模因子为*Accessor*、观察者、建造者、适配器和组合合成正确的解决方案。与 观察者 在第2节中的示例类似，这些基准测试中组合爆炸的来源，是在合成不相交方法时做出的一系列选择。

这一优势在建造者中最为显著：采用轨迹引导的回溯时，Mason 能达到规模 24，并且在规模 22 及以下均未遇到超时。基线 Mason 在规模超过四之后，过半试验都会超时。尽管表现优于基线，Mason 在组合中的表现却不那么出色：在这里，Mason 更容易做出导致原本通过的测试失败的更改，从而扩大依赖集，降低轨迹引导的回溯的有效性。还要注意，初始组合 程序比其他程序更大，其中包含$2n$ 不完整类。

使用名称模式。 当同样允许 Mason 使用名称约束（由嵌入片段中的名称模式生成）时，我们在四个基准测试中看到了明显的性能提升：*Factory*、*Accessor*、观察者和组合。其中，*Factory* 和*Accessor* 受益最大，*Factory* 的综合时间中位数从未超过六秒，*Accessor* 的中位数时间从未超过五秒。

这在一定程度上符合预期；在导致这两个基准测试出现组合爆炸的综合步骤中，名称约束会强制只剩单一选项。在*Accessor*中，这是每个设置器和获取器所要操作的字段选择（类似于第2节中的示例），而在*Factory* 中，这是每个具体工厂所要实例化的类选择。对于观察者和组合，名称约束未能完全约束每一个重要选择，但仍消除了大量不必要的搜索：在观察者 规模为八时，名称约束将综合时间中位数从98秒降至13秒；在组合规模为六时，它们将综合时间中位数从107秒降至14秒。

6 未来工作

未来工作有若干有趣的方向。

名称模式。 名称模式在图8中带来了最为显著的性能提升，尤其是在*Factory* 和 *Accessor* 基准测试中。然而，名称模式语言较为有限，它们只能用于表达类型与成员名称之间的约束。未来工作可以将名称模式扩展为引用抽象语法树其他部分中的名称，例如方法参数的名称。

合成底层细节。 随着 Mason 抽象层次的提高，表达能力有所损失。例如，Mason 无法合成任意表达式或语句序列。未来工作可以考虑将 Mason 与现有的程序综合技术相结合，以使这一点成为可能。我们可以允许空位出现在表达式位置，由 Mason 生成草图 [17]，再使用类似 JSketch [11] 的工具来求解。

更具表达力的规范。 我们的评估表明，当静态信息紧密约束搜索空间时，Mason 的表现尤为出色。然而，Java 类型是一种相当粗粒度的规范。LiquidJava [4]以静态检查的谓词扩展了 Java 类型系统。未来工作可以扩展 Mason 的类型

约束，以纳入这些更丰富的类型，从而在搜索过程中剪除更多候选。

7 相关工作

面向对象程序的合成。 Pasket [10]也会使用设计模式合成 Java 程序。然而，尽管 Mason的输出旨在供程序员使用，Pasket生成的框架模型则用于符号执行。Pasket 使用示例框架客户端生成的日志，Pasket 会实例化设计模式，以生成解决方案的 草图 [17]，再使用名为JSketch [11]的工具将其补全。与片段类似，草图也是带有空位的部分程序，但空位的取值范围是表达式而非名称。为了解决草图，JSketch 会推理动态语义，但不推理静态语义，因此 JSketch 和 Pasket可能会生成类型错误的程序，而类型引导的Mason则不会。推理动态语义的一个后果是，JSketch 和 Pasket 无法合成使用缺乏语义模型的库方法的代码；而Mason 只需要类型签名，这些签名默认即可获取。

已有多种技术 [3, 7, 8, 13] 可用于合成 Ruby 和 Java 等面向对象语言中的单个函数。这些技术能够合成表达式或语句序列，但 Mason 不能创建新的类、接口和成员。在这些技术中，RbSyn 可根据 API 调用合成 Ruby 函数，与 Mason 有相似之处。RbSyn 也利用单元测试和测试失败来引导搜索，不过它要求候选程序执行额外的副作用，并由 RbSyn 直接对这些副作用进行推理。然而，因此 RbSyn 要求库方法包含效应注解。

类型引导的合成。 Synqid [16], Myth [15], 和 Scythe [14] 代表了类型引导合成的传统方法，即根据类型检查和类型推断的证明系统推导合成规则。相比之下，尽管类型推断是 Mason 合成的关键部分，但 Mason 的合成规则并不遵循其结构。此外，这三种方法都在函数式语言中合成单个函数或表达式。与 Mason 类似，Synqid 使用子类型约束引导合成，但采用了表达能力更强的精化类型 ，并以逻辑谓词对其进行注解。与 Mason 类似，Myth 也使用测试用例引导合成，但其输入输出示例的表达能力较弱。不过，Myth 能够在合成过程中不断细化示例，并利用它们在搜索时排除候选项。Scythe 在合成过程中保留一组类型约束，原因与 Mason 类似：在类型推断的语境下，填充程序的某些部分可能会产生非局部影响。不过，Scythe 只对相等约束进行推理。

设计模式自动化。 软件工程文献中的一个相关主题是代码的自动重构以使用设计模式。与 Mason 不同，这些技术应用于完整程序，因此旨在开发过程的后期使用。Remodel [9] 利用遗传算法来应用微型转换 [1], 能够组合设计模式的小型代码更改，并使用软件工程度量来确定适应度。JungGL [19]和 R^2 [12]是用于编写重构脚本的工具，这些脚本使用专用 API 来应用设计模式。与 Mason 的片段类似，脚本允许用户配置，但重构脚本的作者必须设计算法，而 Mason 则无需如此。生成类型正确的程序也是该领域的一个关注点；Tip 等人 [18] 使用类型约束来确保重构的类型正确性。与我们一样，他们为 Java 子集定义了一套类型约束系统，并用其约束可能的程序变换。然而，他们对约束的处理因每次重构而异：某些约束必须在转换之后成立，而其余约束则不必。

8 结论

本文介绍了 Mason，这是一种由类型和名称引导、能够根据程序片段合成面向对象程序的工具。我们为这一方法建立了形式化模型，提出了一种适用于核心面向对象语言的基于约束的类型推断算法，以及一组由类型和名称引导的合成规则。我们还为 Mason 引入了两项扩展，旨在提升性能。

我们使用 Mason 将设计模式应用于一套基准测试。我们发现，当能够合成的类型正确的程序很少时，基线 Mason 表现良好。如果并非如此，但组合爆炸的根源分散在不同的方法和单元测试中，那么轨迹引导的回溯可显著提升性能，优于基线 Mason。当类型和成员可以采用命名约定时，名称模式还能进一步提升性能。我们认为，Mason 在使用设计模式合成多类面向对象程序方面迈出了重要一步。

数据可用性声明

我们已将Mason的源代码、论文中使用的基准测试套件、用于复现第5节结果的脚本和说明，以及一个 Docker 图像作为制品提供 [6]。

参考文献

[1] M.O. Cinneide。2000年。通过自动化重构引入设计模式。载于 *2000*年国际软件工程会议论文集。*ICSE 2000*：新千年。722–724。doi：10.1145/ 337180.337612[2] Albert Danial。2021 年。 *cloc*：*v1.92*。 doi：10.5281/zenodo.5760077

[3] Yu Feng、Ruben Martins、Yuepeng Wang、Isil Dillig 和 Thomas W. Reps。2017年。面向复杂 API 的基于组件的合成。载于第44届 ACM SIGPLAN 编程语言原理研讨会论文集（法国巴黎）（POPL '17）。计算机协会，纽约，美国，599–612。doi:10.1145/3009837.3009851[4] Catarina Gamboa、Paulo Canelas、Christopher Timperley 和 Alcides Fonseca。2023年。面向 Java 液体类型的面向可用性的设计。载于第45届国际软件工程会议论文集（墨尔本，维多利亚州，澳大利亚）（ICSE '23）。IEEE 出版社，1520–1532。doi:10.1109/ICSE48619.2023.00132[5] E. Gamma、R. Helm、R. Johnson 和 J. Vlissides。1994年。《设计模式：可复用的面向对象软件的要素》。培生教育。https://books.google.com/books?id=6oHuKQe3TjQC[6] Jasper Geer 和 Fox Huston。2026年。Mason：类型和名称引导的程序合成制品。doi: 10 5281 18676122 。/[7] zenodo。Sankha Narayan Guria、Jeffrey S. Foster 和 David Van Horn。2021年。RbSyn：类型和效应引导的程序合成。载于第42届 ACM SIGPLAN 编程语言设计与实现国际会议论文集（线上，加拿大）（PLDI 2021）。计算机协会，纽约，美国，344–358。doi:10.1145/3453483.3454048[8] Sankha Narayan Guria、Jeffrey S. Foster 和 David Van Horn。2023年。Absynthe：抽象解释引导的合成。《ACM 程序语言会议录》 7, 171 2023 24 10 1145 3591285[9] 2010 PLDI，文章（6月），页码。doi: 。/ Adam C. Jensen 和 Betty H.C. Cheng。 。关于将遗传编程用于自动化重构和引入设计模式。载于第12届遗传与进化计算年会论文集（波特兰，俄勒冈州，美国）（GECCO '10）。计算机协会，纽约，美国，1341–1348。doi:10.1145/1830483.1830731[10] Jinseong Jeon、Xiaokang Qiu、Jonathan Fetter-Degges、Jeffrey S. Foster 和 Armando Solar-Lezama。2016年。为符号执行合成框架模型。载于第38届国际软件工程会议论文集（奥斯汀，得克萨斯州）（ICSE '16）。计算机协会，纽约，美国，156–167。doi:10.1145/ 2884781.2884856[11] Jinseong Jeon、Xiaokang Qiu、Jeffrey S. Foster 和 Armando Solar-Lezama。2015年。JSketch：面向 Java 的草图式编程。载于2015年第10届软件工程基础联合会议论文集（贝加莫，意大利）（ESEC/FSE 2015）。计算机协会，纽约，美国，934–937。doi:10.1145/2786805.2803189[12] Jongwook Kim、Don Batory 和 Danny Dig。2015年。在 Java 中编写参数化重构脚本，以改造设计模式。载于2015年 IEEE 软件维护与演化国际会议论文集（ICSME '15）。IEEE 计算机学会，美国，211–220。doi:10.1109/ICSM.2015.7332467[13] Benjamin Mariano、Josh Reese、Siyuan Xu、ThanhVu Nguyen、Xiaokang Qiu、Jeffrey S. Foster 和 Armando Solar-Lezama。2019年。基于代数库规范的程序合成。《ACM 程序语言会议录》3，OOPSLA，文章132（2019年10月），25页。doi:10.1145/ 3360558[14] Peter-Michael Osera。2019年。基于约束的类型引导程序合成。载于第4届 ACM SIGPLAN 类型驱动开发国际研讨会论文集（柏林，德国）（TyDe 2019）。计算机协会，纽约，美国，64–76。doi:10.1145/3331554.3342608[15] Peter-Michael Osera 和 Steve Zdancewic。2015年。类型和示例引导的程序合成。载于第36届 ACM SIGPLAN 编程语言设计与实现大会论文集（波特兰，俄勒冈州，美国）（PLDI '15）。计算机协会，纽约，美国，619–630。doi:10.1145/2737924.2738007[16] Nadia Polikarpova、Ivan Kuraj 和 Armando Solar-Lezama。2016年。从多态精化类型合成程序。载于论文集 第*37*届 *ACM SIGPLAN* 编程语言设计与实现大会论文集 （美国加利福尼亚州圣巴巴拉，2016年6月2日）。ACM，522–538页。doi：10.1145/2908080.2908093[17] Armando Solar-Lezama。2009年。程序合成的草图方法。载于 编程语言与系统。Springer Berlin Heidelberg，柏林，海德堡，4–13页。[18] Frank Tip、Adam Kiezun 和 Dirk Bäumer。2003年。使用类型约束实现泛化的重构。载于 第*18*届 *ACM SIGPLAN* 面向对象编程、系统、语言与应用年会论文集 （美国加利福尼亚州阿纳海姆）（*OOPSLA '03*）。计算机协会，纽约，纽约州，美国，13–26页。 doi：10.1145/949305.949308[19] Mathieu Verbaere、Ran Ettinger 和 Oege de Moor。2006年。JunGL：一种用于重构的脚本语言。载于 第*28*届国际软件工程大会论文集 （中国上海）（*ICSE '06*）。计算机协会，纽约，纽约州，美国，172–181页。 doi：10.1145/1134285.1134311