egg: 快速和可拓展的等式饱和

MAX WILLSEY, University of Washington, Seattle, USA
CHANDRAKANA NANDI, University of Washington, Seattle, USA
YISU REMY WANG, University of Washington, Seattle, USA
OLIVER FLATT, University of Utah, Salt Lake City, USA
ZACHARY TATLOCK, University of Washington, Seattle, USA
PAVEL PANCHEKHA, University of Utah, Salt Lake City, USA

一个 e-graph （e 图）可以有效地表示许多表达式上的同构关系。虽然它最初是在 20 世纪 70 年代末为用于自动定理证明器而开发的，最近的一项技术被称为 *等式饱和*（*equality saturation*）。重新利用了 e-graphs 来实现最先进的技术（SOTA），用于重写驱动的编译器优化和程序综合器（program synthesizers）。然而，对于这种新的使用情况，e-graphs 仍然没有被专门化。等式饱和的工作量表现出明显的特征，并且往往需要专门的 e-graph 扩展来纳入纯语法重写之外的转换。

我们的工作提供了两项技术，使 e-graphs 快速和方便扩展，使其专门用于等式饱和。一是新的摊销的不变性恢复技术，称为 *重建*（*rebuilding*）。利用了等式饱和的独特工作量。在实践中提供了比目前技术更多的渐进式加速（asymptotic speedups）。二是一个名为 *e-class analyses* （e 类分析）的通用机制，它将特定领域的分析整合到 e-graph 中，减少了临时操作的需要。

我们将这些技术在一个新的开源库实现，叫做 egg 。我们对以前发表的三个等式饱和的应用进行了案例研究，它突出了 egg 的性能和灵活性，在不同的领域可达到最先进的效果。

1 引入

等价图（Equality graphs，即 e-graph ）最初是为了在自动定理证明器（ATPs）中有效地表示同余（congruence）关系。在高层次上，e-graphs [Nelson 1980; Nieuwenhuis and Oliveras 2005] 扩展了并查集（union-find）[Tarjan 1975]，用来紧凑地表示表达式的等价类，同时保持一个关键的不变性：等价关系在同余下是封闭的。[1]

在过去的十年中，一些项目重新利用了 e-graphs 来实现最先进的重写驱动的编译器优化和程序综合器。它们使用一种被称为 *等式饱和* [Joshi et al. 2002; Nandi et al. 2020; Panchekha et al. 2015; Premtoon et al. 2020; Stepp et al. 2011; Tate et al. 2009; Wang et al. 2020] 的技术。给定一个输入程序 p ，等式饱和构建了一个 E 图，它代表与 p 等价的一大组程序，然后可以从 E 中提取” 最好的” 程序。e-graph 的增长是通过反复应用基于模式的重写。最重要的是，这些改写只向 e-graph 增加信息。不需要仔细排序。在达到一个不动点（*饱和*）时，E 将代表 *所有等价的方式来表达表达* p 与给定重写的关系。饱和（或超时）之后，最后的 *萃取*（extraction）程序将分析 E 并根据用户提供的成本函数，选择最佳程序。

理想情况下，用户可以简单地提供一个语言语法和一些重写规则。而等式饱和将产生一个有效的优化器。但是有两个挑战阻碍了我们的期望：首先，随着 E 的增长，维持全等性会变得很昂贵（expensive）。部分原因是传统 ATP（Automated Theorem Proving，自动定理证明）设置中的 e-graphs 仍然没有针对不同的 *等式饱和工作负载*（*equality saturation workload*）特化。其次，许多应用关键性地依赖于 *领域特定的分析*（*domain-specific analyses*），但是集成它们需要对 e-graph 进行特别的扩展。由于缺乏一个通用的扩展机制迫使研究人员多次从头实现等式饱和 [Panchekha et al. 2015; Tate et al. 2009; Wu et al. 2019]。这些挑战限制了等式饱和的实用性。

等式饱和工作负载. ATPs 经常查询和修改 e-graphs ，还需要对文本进行*回溯*（*backtracking*）来撤销修改（例如在 DPLL(T) [Davis and Putnam 1960] 中）。这些要求迫使传统的 e-graph 设计为在每次操作后都要保持同余不变性。与此相反，等式饱和工作负载不需要回溯，并且可以被分解为以下不同的阶段：(1) 查询 e-graph 以同时找到所有的重写匹配 (2) 修改 e-graph 以合并所有匹配术语的等价关系。

[1] 直观地说，同余简单的说就是 $a \equiv b$ 蕴含 $f(a) \equiv f(b)$.

我们提出了一种新的摊销算法称为 *rebuilding* ，该算法将 e-graph 的不变性维护推迟到等式饱和阶段的边界，而不影响健全性。从经验上看，重建算法比传统的方法提供了渐进式的速度提升。

特定领域分析. 等价物饱和主要由句法重写驱动。但许多应用需要额外的解释推理以将领域知识带入 e-graph 中。过去的实现方式是诉诸于临时性的 e-graph 操作来整合那些本来是简单的程序分析，如常量折叠（constant folding）。

为了灵活地纳入这种推理。我们引入了一个新的、通用的机制，叫做 *e 类分析*（*e-class analyses*）。一个 e-class 分析对每个 e-class （term 的等价类）用来自半格域（semilattice domain）的论据（fact）进行推导。随着 e-graphs 的增长。论据被引入、传播和连接以满足 *e-class 分析不变量*的要求。它将分析论据与 e-graph 中的项联系起来。重写与 e-class 分析的合作方式是依赖于分析论据和添加等价物，而这些等价物又建立额外的论据。我们的案例研究和示例（ 5 与 6 节）展示了 e-class 分析，如常量折叠和自由变量分析这些分析需要在以前的等价饱和实现中需要进行专门定制。

我们在一个开源库 [2] 中实现了重建（rebuilding）和 e-class 分析。它称为 egg (**e-g**raphs **g**ood)。egg 专门针对平等饱和。利用其工作负载的特点和支持简单的扩展机制来提供针对 e-graphs 的程序综合和优化。egg 还解决了更多普通的挑战。例如，在用户定义的语言、重写和成本函数上进行参数化。同时还提供了一个优化的实现。我们的案例研究表明，egg 的功能是如何构成了一个通用的、可重复使用的 e-graph 库，可以支持不同领域的等式饱和。

总而言之，本文的工作包括：

- 重建 (Section 3)，它是只在等式饱和算法的选定点恢复关键的正确性和性能不变性的一种技术。我们的评估表明，重建的速度在实践中比现有技术更快。
- E-class 分析 (Section 4)，它是一种整合特定领域分析的技术，不能被表达为纯粹的语法重写。e-class 分析的不变性保证使得重写和分析之间能够协作。
- 一个快速、可拓展的 e-graphs 实现库，称作 egg (Section 5).
- 真实世界的案例研究。使用已发布的 egg 的工具，用于演绎综合（deductive synthesis）和程序优化（program optimization）的案例研究。这些领域包括浮点精度、线性代数优化、和 CAD 程序综合 (Section 6)。相比以前的实现方式 egg 的速度快了几个数量级，并提供了更多的功能。

2 背景

egg 建立在 e-graphs 和等式饱和的基础上。本节描述了这些技术，并介绍了 egg 所要解决的挑战。

2.1 E-graphs （e 图）

一个 *e-graph* 是一个数据结构，它存储了一组项（term）和这些项上的一个同义关系。它最初是为定理证明器开发的，现在仍然被用于定理证明器的核心 [De Moura and Bjørner 2008; Detlefs et al. 2005; Nelson 1980], e-graphs 也被用来支持一种程序优化技术称为 *等式饱和*（*equality saturation*）。[Joshi et al. 2002; Nandi et al. 2020; Panchekha et al. 2015; Premtoon et al. 2020; Stepp et al. 2011; Tate et al. 2009; Wang et al. 2020].。

2.1.1 **定义**. 直观地，一个 e-graph 是一组等价类（*e-classes*）。每个 e-class 是一组代表特定语言中等价项的 *e-nodes* 。而一个 e-node 是一个函数符号与一个子 e-classes 列表的配对。更确切地：

Definition 2.1 (单个 E-graph 的定义). 给定 Figure 1 中的定义与语法, 一个 *e-graph* 是一个元组（tuple）(U, M, H) ，其中:

[2] 主页: https://egraphs-good.github.io, 源码: https://github.com/egraphs-good/egg, 文档: https://docs.rs/egg

函数符号	f, g	
e-class ids	a, b	opaque identifiers
terms	$t ::= f \mid f(t_1, \ldots, t_m)$	$m \geq 1$
e-nodes	$n ::= f \mid f(a_1, \ldots, a_m)$	$m \geq 1$
e-classes	$c ::= \{n_1, \ldots, n_m\}$	$m \geq 1$

Fig. 1. **语法和元变量为一个** e-graph **的组成部分。函数符号可以作为常数** e-nodes **和项单独存在。一个**e-class id **是一个不透明的标识符，可以用** = **来比较是否相等。**

- 一个并查集（union-find [Tarjan 1975]）U 存储了一个 e-class id 等价关系（用 $\equiv_{\text{id}}$ 表示）。
- *e-class map* M 将 e-class id 映射到 e-classes 。所有等价的 e-class id 都映射到相同的 e-class，即，$a \equiv_{\text{id}} b$ 当且仅当 $M[a]$ 与 $M[b]$ 是同一个集合。一个 e-class id a 被称为 *refer to* e-class $M[\texttt{find}(a)]$.
- *hashcons*[3] H 是 e-nodes 到 e-class id 的映射（map）.

请注意，一个 e-class 有一个 ID （它的规范（canonical）e-class id）。但一个 e-node 没有。[4] 无歧义时，我们使用 e-class id a 指代 e-class $M[\texttt{find}(a)]$ 。

Definition 2.2 (规范化). 一个 e-graph 的并查集 U 提供一个 `find` 操作，能规范化（canonicalizes）e-class id 以便 $\texttt{find}(U, a) = \texttt{find}(U, b)$ 当且仅当 $a \equiv_{\text{id}} b$。无歧义时，我们忽略 `find` 的第一个参数。

- 一个 e-class id a 是规范的当且仅当 $\texttt{find}(a) = a$.
- 一个 e-node n 是规范的当且仅当 $n = \texttt{canonicalize}(n)$, 此时 $\texttt{canonicalize}(f(a_1, a_2, \ldots)) = f(\texttt{find}(a_1), \texttt{find}(a_2), \ldots)$.

Definition 2.3 (项（term）的表示). 一个 e-graph 、e-class 或 e-node 被称为 *表示*一个项 t，如果 t 可以在其中被“发现”。表示的方法是递归定义的：

- 一个 e-graph 表示一个项如果它的每一个 e-classes 也是这样的。
- 一个 e-class c 表示一个项如果每一个 e-node $n \in c$ 也是这样的。
- 一个 e-node $f(a_1, a_2, \ldots)$ 表示一个项 $f(t_1, t_2, \ldots)$ 如果它们有一样的的函数符 f 并且 e-class $M[a_i]$ 表示项 t_i.

当每个 e-class 是一个单子（singleton，只包含一个 e-node ），一个 e-graph 本质上是一个共享的项图（term graph）。Figure 2a 显示了一个代表表达式 $(a \times 2)/2$ 的 e-graph 。

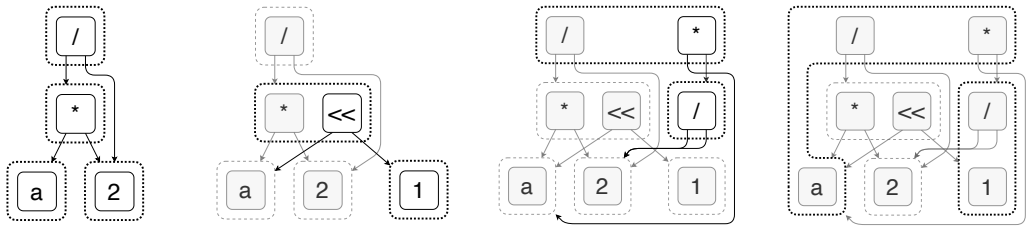
Definition 2.4 (等价，Equivalence). 一个 e-graph 定义了三个等价关系。

- 对于 e-class id: $a \equiv_{\text{id}} b$ 当且仅当 $\texttt{find}(a) = \texttt{find}(b)$.
- 对于 e-nodes: $n_1 \equiv_{\text{node}} n_2$ 当且仅当 e-nodes n_1, n_2 在同样的 e-class 里, 也就是说, $\exists a.\ n_1, n_2 \in M[a]$.
- 对于 terms: $t_1 \equiv_{\text{term}} t_2$ 当且仅当 terms t_1, t_2 以相同的 e-class 表示。

无歧义时，我们使用无下标的 $\equiv$ 。

Definition 2.5 (同余，Congruence). 对于一个给定的 e-graph ，让 $\cong$ 表示在 e-nodes 上的一个全等关系，以便 $f(a_1, a_2, \ldots) \cong f(b_1, b_2, \ldots)$ 当且仅当 $a_i \equiv_{\text{id}} b_i$。让 $\cong^*$ 表示 $\equiv_{\text{node}}$ 的全等闭包（congruence closure）。即 $\equiv_{\text{node}}$ 的最小的超集，同时也是 $\cong$ 的超集。请注意，可能有两

[3]我们使用术语 *hashcons*（哈希康）来引入记忆化技术，因为两者都可以避免创建现有对象的新副本。
[4]我们对 e-graph 的定义反映了 egg 的设计，因此不同于其他一些 e-graph 的定义和实现。我们的定义的特别之处是使用可识别的 e-classes 而不是 e-nodes。



(a) 初始化包含 $(a \times 2)/2$ 的 e-graph (b) 施加重写规则 $x \times 2 \to x \ll 1$ (c) 施加重写规则 $(x \times y)/z \to x \times (y/z)$ (d) 施加重写规则 $x/x \to 1$ 和 $1 \times x \to x$

Fig. 2. 一张 e-graph 由包含等价的 e-nodes（实心框）的 e-classes（虚线框）组成。连接线连结 e-nodes 到它们的 e-classes 。加黑之处是增加和修改的部分。将重写应用于一个 e-graph 会增加新的 e-nodes 和边线。但没有删除任何东西。通过改写增加的表达式会与匹配的 e-class 合并。在 Figure 2d 中，改写不增加任何新节点，只是合并 e-class 。由此产生的 e-graph 有一个循环，代表无限多的表达式：a, $a \times 1$, $a \times 1 \times 1$, 以此类推。

个 e-nodes ，以便 $n_1 \cong^* n_2$ 但是 $n_1 \ncong n_2$ 且 $n_1 \neq_{\text{node}} n_2$. 关系 $\cong$ 只代表一个单步的同余，计算全等闭包可能需要一个以上的步骤。

2.1.2 E-graph **不变量**. e-graph 必须保持不变性，以便正确有效地实现 Section 2.1.3 中的操作。本节只定义了不变量（invariants），关于如何维护这些不变量的讨论推迟到 Section 3 中。这些被统称为 *E-graph* 不变量（*e-graph invariants*）。

*Definition 2.6 (*同余不变量（*Congruence Invariant*）*).* e-nodes 上的等价关系必须是闭合的同余关系，也就是说，$(\equiv_{\text{node}}) = (\cong^*)$。该 e-graph 必须确保同余的 e-nodes 是在同一个 e-class 中。由于相同的 e-nodes 是平凡的同余的（trivially congruent）。这意味着一个 e-node 必须唯一地包含在一个单一的 e-class 中。

*Definition 2.7 (*哈希康不变量（*Hashcons Invariant*）*).* 哈希康 H 必须将所有规范 e-nodes 映射到其 e-class id。换言之：

$$\text{e-node } n \in M[a] \iff H[\texttt{canonicalize}(n)] = \texttt{find}(a)$$

如果哈希康不变量成立，那么一个程序 `lookup` 就可以快速找到哪个 e-class （如果有的话）有一个与给定的 e-node n 同余的 e-node： $\texttt{lookup}(n) = H[\texttt{canonicalize}(n)]$。[5]

2.1.3 **接口和重写**. E-graphs 与经典的数据结构并查集有许多相似之处，它在内部采用了这种结构，继承了很多术语。E-graphs 提供了两个主要的低级变异操作：

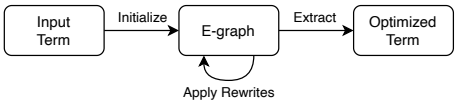
- `add` 接受一个 e-node n 进行：
  - 如果 $\texttt{lookup}(n) = a$, 返回 a;
  - 如果 $\texttt{lookup}(n) = \emptyset$, 则设置 $M[a] = \{n\}$ 并且返回 id a.
- `merge` (有些时候称为 `assert` 或 `union`) 接受两个 e-class id： a 和 b, 在并查集 U 中合并它们, 然后通过设置 $M[a]$ 与 $M[b]$ 到 $M[a] \cup M[b]$ 拼合 e-classes 。

这两种操作都必须采取额外的步骤来维持同余的不变量。不变量维护将在 Section 3 中讨论。

E-graphs 还提供了查询数据结构的操作：

- `find` 使用并查集 U 来进行规范化 e-class id，如定义 2.1 所述。

[5] 【译注】 `canonicalize`： 规范化转换




```
1  def equality_saturation(expr, rewrites):
2    egraph = initial_egraph(expr)
3
4    while not egraph.is_saturated_or_timeout():
5
6      for rw in rewrites:
7        for (subst, eclass) in egraph.ematch(rw.lhs):
8          eclass2 = egraph.add(rw.rhs.subst(subst))
9          egraph.merge(eclass, eclass2)
10
11   return egraph.extract_best()
```

Fig. 3. **等式饱和的框图和伪代码。传统上，等式饱和在整个算法中保持 e-graph 数据结构不变量。**

- `ematch` 执行在 e-graph 中查找模式的 *e-matching* [de Moura and Bjørner 2007; Detlefs et al. 2005] 过程。`ematch` 接受一个具有变量占位符的模式项 p 并返回一个元组列表 (σ, c)，其中 σ 是变量到 e-class id 的替换，使得 $p[\sigma]$ 能在 e-class c 中表示。

这些可以组合起来对 e-graph 执行重写。要在 e-graph 上应用重写 $\ell \to r$，`ematch` 会找到元组 (σ, c)，其中 e-class c 表示 $\ell[\sigma]$。然后，对于每个元组，`merge(`c`, add(`$r[\sigma]`))` 将 $r[\sigma]$ 添加到 e-graph 中，并使其与匹配的 e-class c 统一。

Figure 2 展示了一个经历了一系列重写的 e-graph。注意这个过程只是增加性的；初始项 $(a \times 2)/2$ 仍然在 e-graph 中表示。在 e-graph 中重写也可以饱和，这意味着 e-graph 已经学会了给定重写可以推出的所有等价关系。如果用户试图将 $x \times y \to y \times x$ 应用于 e-graph 两次，第二次将不会添加额外的 e-nodes 并不会进行新的合并；e-graph 可以检测到这一点并停止应用该规则。

2.2 等式饱和（Equality Saturation）

项重写 [Dershowitz 1993] 是一种久经考验的方法，它针对于程序综合 [Joshi et al. 2002; Tate et al. 2009]、定理证明 [De Moura and Bjørner 2008; Detlefs et al. 2005]、程序转换 [Andries et al. 1999], 中的等式推理在这种情况下，一个工具重复选择一组公理改写中的一个。在给定的表达式中搜索与左式相匹配的表达式，并将匹配的实例替换为被替换的右式。

项重写通常是破坏性的并且“忘记”了匹配的左式。考虑应用一个简单的降低计算量（strength reduction）重写：$(a \times 2)/2 \to (a \ll 1)/2$。新项不带有初始项的信息。在此之前应用降低计算量重写阻止我们消除 2/2。在编译器社群中，这个关于何时应用哪个重写的问题被称为*编译优化选择*（*phase ordering*）问题。

解决编译优化选择问题的一种方法是同时应用所有重写，并跟踪每个表达式。这消除了选择正确规则的问题，但是简单实现需要给定重写数量的指数级空间。*等式饱和*（*equality saturation*）[Stepp et al. 2011; Tate et al. 2009] 是一种使用 e-graph 有效地进行此重写的技术。

Figure 3 展示了等式饱和工作流程。首先，从输入的项创建初始 e-graph。算法的核心运行一组重写规则，直到 e-graph 饱和或超时。最后，称为 *extraction*（提取）的程序根据某些成本函数选择最优表示的项。对于简单的成本函数，自底向上贪心遍历 e-graph 足以找到最佳项。其他提取程序已经探索了更复杂的成本函数 [Wang et al. 2020; Wu et al. 2019]。

等式饱和消除了选择何时应用哪些重写的繁琐而容易出错的任务，提供了一种简单易用的工作流程：确定语言相关重写，从给定表达式创建初始 e-graph，运行规则直到饱和，最后提取最优的等价表达式。不幸的是，这种技术仍然是特别定制的，等式饱和用户必须自己实现专门针对语言的 e-graph，避免性能问题，并通过骇入（hack in）的方法实现纯语法重写无法支持的解释性推理。egg 旨在解决这些困难。

2.3 等式饱和和定理证明

等式饱和引擎和定理验证器都有一些能力在对方身上复制是不切实际的。像可满足模理论（SMT）求解器这样的自动化定理证明器是通用工具，除了支持可满足性查询外，还整合了专业的，领域特定的求解器，允许在支持的理论内进行解释性推理。另一方面，等式饱和是专门用于优化的，其提取过程直接产生关于给定成本函数的最优的项。

尽管 SMT 求解器确实是更通用的工具，但等式饱和并不被 SMT 取代；当不需要 SMT 的完全通用性时，特异化的方法可能更快。为了证明这一点，我们复制了最近 TASO 论文的一部分 [Jia et al. 2019]，它优化了深度学习模型。作为工作的一部分，他们必须验证一组合成等式，它们需要遵守一组具有普遍性的公理。TASO 使用 Z3 [De Moura and Bjørner 2008] 执行验证，即使大部分 Z3 的功能（或运算、回溯、理论等）都不需要使用。等式饱和也可以用于验证这些等式，将每个等式的左右两侧添加到 e-graph 中，将公理作为重写规则运行，然后检查两侧是否最终在同一个 e-class 中。Z3 花费 24.65 秒进行验证；egg 在 1.56 秒（快 15×）中完成相同任务，或在使用 egg 的批量评估（Section 5.3）时只需要 0.52 秒（快 47× ）。

3 重建：E-GRAPH 不变量维护的全新视角

传统上 [Detlefs et al. 2005; Nelson 1980]，e-graphs 在每次操作后保持数据结构不变式。我们将这种不变式恢复分为称为 *rebuilding*（重建）的过程。这种分离允许客户端选择何时执行 e-graph 不变式。立即在每次操作后执行重建就复制了传统的不变式维护方法，相比之下，较少的重建可以摊销不变式维护成本，显著提高性能。

在本节中，我们首先描述了 e-graphs 如何传统地维护不变量（Section 3.1）。然后，我们描述了重构框架以及它如何捕获一系列维护不变量的方法，包括传统方法（Section 3.2）。利用这种灵活性，我们然后给出了一种修改的算法，用于等式饱和，它只在特定点强制执行 e-graph 不变量（Section 3.3）。最后，我们证明这种新方法比传统的等价饱和提供了渐进加速（Section 3.4）。

3.1 向上合并

如果不仔细操作，对 e-graph 进行的变异操作（`add` 和 `merge`，Section 2.1.3）可能会破坏 e-graph 不变性。E-graphs 传统上使用 *哈希康化*（*hashconsing*）和 *向上合并*（*upward merging*）来维护同余不变性。

`add` 操作依赖于哈希康不变性 (hashcons invariant，定义 2.7) 来快速检查要添加的 e-node n 或与其同余的节点是否已经存在。如果没有这个检查，`add` 将创建一个新的 e-class ，其中包含 n，即使某些 $n' \cong n$ 已经在 e-graph 中，这将违反同余不变性。

`merge` 操作可能会违反 e-graph 的两个不变性。如果 $f(a,b)$ 和 $f(a,c)$ 分别位于两个不同的 e-classes x 和 y 中，合并 b 和 c 也应该合并 x 和 y 以维护同余不变性。这可能会进一步传播，需要额外的合并。

E-graphs 为每个 e-class 维护一个 *父列表*（*parent list*）以维持同余。e-class c 的父列表包含所有将 c 作为子节点的 e-nodes 。合并两个 e-classes 时，e-graphs 检查这些父节点列表，找到现在同余的父节点，如果需要就递归进行 “向上合并”。

`merge` 程序还必须执行记录（bookkeeping）来保持哈希康不变性。特别地，合并两个 e-classes 可能会改变这些 e-classes 的父 e-nodes 如何规范化。因此 `merge` 操作必须在哈希康中删除、重规范化并替换这些 e-nodes 。在用于等式饱和的现有 e-graph 实现 [Panchekha et al. 2015] 中，在合并时维护不变性可能占用绝大部分运行时间。

3.2 重建细节

传统上，不变量恢复是 `merge` 操作本身的一部分。重建将这些关注点分开，减少 `merge` 的责任并允许摊销不变量维护的工作。在重建范式中，`merge` 保持一个 *工作列表*（*worklist*），其中包含需要” 向上合并” 的 e-class id，即父节点可能同余但尚未在同一 e-class 中的 e-classes

```
def add(enode):
  enode = self.canonicalize(enode)
  if enode in self.hashcons:
    return self.hashcons[enode]
  else:
    eclass_id = self.new_singleton_eclass(enode)
    for child in enode.children:
      child.parents.add(enode, eclass_id)
    self.hashcons[enode] = eclass_id
    return eclass_id

def merge(id1, id2)
  if self.find(id1) == self.find(id2):
    return self.find(id1)
  new_id = self.union_find.union(id1, id2)
  # 传统的 egraph 合并可以通过
  # 在将 eclass 添加到 worklist 之后
  # 立即调用重建来模拟。
  self.worklist.add(new_id)
  return new_id

def canonicalize(enode)
  new_ch = [self.find(e) for e in enode.children]
  return mk_enode(enode.op, new_ch)

def find(eclass_id):
  return self.union_find.find(eclass_id)
```

```
def rebuild():
  while self.worklist.len() > 0:
    # 将 worklist 清空到一个局部变量中
    todo = take(self.worklist)
    # 对 eclass refs 进行规范化和去重,
    # 以节省 repair 调用的次数。
    todo = { self.find(eclass) for eclass in todo }
    for eclass in todo:
      self.repair(eclass)

def repair(eclass):
  # 更新 hashcons,
  # 使其总是指向规范的 enodes 到规范的 eclasses。
  for (p_node, p_eclass) in eclass.parents:
    self.hashcons.remove(p_node)
    p_node = self.canonicalize(p_node)
    self.hashcons[p_node] = self.find(p_eclass)

  # 去重父节点,
  # 注意等价的父节点会被合并并放在 worklist 中。
  new_parents = {}
  for (p_node, p_eclass) in eclass.parents:
    p_node = self.canonicalize(p_node)
    if p_node in new_parents:
      self.merge(p_eclass, new_parents[p_node])
    new_parents[p_node] = self.find(p_eclass)
  eclass.parents = new_parents
```

Fig. 4. add, merge, rebuild **和一些支持的方法的伪代码。在以上每个方法中，** self **指的是正被修改的** e-graph 。

。rebuild 操作处理这个工作列表，恢复去重（deduplication）和同余（congruence）的不变量。重建在如何恢复同余中类似于其他方法，（见 相关工作 与 Downey et al. [1980] 的比较）；但特别的是它允许客户在更大规模的算法（如等式饱和）上选择何时恢复不变量。

Figure 4 展示了主 e-graph 操作和重建的伪代码。请注意，为了完整性起见，给出了 add 和 canonicalize，但它们与传统的 e-graph 实现相同。merge 操作类似，但它只将新的 e-class 添加到工作列表中，而不是立即开始向上合并。在工作列表添加后立即调用 rebuild（Figure 4 第 19 行）将产生传统行为——立即恢复不变性。

rebuild 方法实质上是在工作列表上的 e-classes 上调用 repair，直到工作列表为空。与直接操作工作列表不同，egg 的 rebuild 方法首先将其移动到局部变量中，并对 e-classes 进行去重直到等价（equivalence）。处理工作列表可能会 merge e-classes ，所以将工作列表分成几块，以确保在前一块中被等价的 e-class id 在随后的几块中被去重。

rebuild 的实际工作发生在 repair 方法中。repair 检查 e-class c，首先在 hashcons 中对具有 c 作为子项的 e-nodes 进行规范化。然后，它执行的基本上是一个 “层” 的向上的合并。如果父节点中有任何一个已经变得等价，则它们的 e-classes 被合并，结果被添加到工作列表中。

工作列表的去重，从而减少对 repair 的调用，是延迟重建改善性能的核心。直观地说，重建的上升合并过程追踪 e-graph 中的同余 “路径”。当重建立即发生在 merge 之后（因此经常发生）时，这些路径可能会大量重叠。通过延迟重建，块和去重方法可以合并这些路径的重叠部分，节省了原本冗余的工作。在我们修改的等价饱和算法中（Section 3.3），延迟重建对性能有着显著的改善（Section 3.4）。

3.2.1 **重建的例子**. 延迟重建通过摊销维护 hashcon 不变量的工作来加速同余的维护考虑以下 e-graph 中的 term：$f_1(x), ..., f_n(x), , y_1, ..., y_n$。工作量为 $\texttt{merge}(x, y_1), ..., \texttt{merge}(x, y_n)$。每次

合并可能会更改 $f_i(x)$ 的规范表示，因此传统的不变性维护策略可能需要 $O(n^2)$ 的 hashcons 更新。通过延迟重建，merge 在恢复 hashcons 不变性之前发生，最多只需要 $O(n)$ 个 hashcons 更新。

延迟重建还可以减少调用 repair 的次数。考虑以下 e-graph 中的 w 个 term，每个都嵌套在 d 函数符号之下：

$$f_1(f_2(\ldots f_d(x_1))), \quad \ldots, \quad f_1(f_2(\ldots f_d(x_w)))$$

请注意，w 对应于这组 term 的宽度，d 对应深度。工作量为 $w-1$ 次合并，将所有 x 合并在一起：for $i \in [2, w]$, $\texttt{merge}(x_1, x_i)$。

在传统的向上合并范式中，每次调用 merge 后都会调用 rebuild，每次 $\texttt{merge}(x_i, x_j)$ 都需要 $O(d)$ 次调用 repair 来维护同余，每层 f_i 都需要一次。在整个工作量中，这需要 $O(wd)$ 次调用 repair。

然而，使用延迟重建时，可以在恢复同余之前进行 $w-1$ 次合并。假设 x 全部合并到 e-class c_x 中。在最后调用 rebuild 时，去重工作列表中唯一的元素是 c_x。调用 c_x 上的 repair 将 f_d 的 e-classes 合并到 e-class c_{f_d} 中，并将包含这些 e-nodes 的 e-classes 添加回工作列表。当工作列表再次去重时，c_{f_d} 将是唯一的元素，过程将重复。因此，整个工作量只会产生 $O(d)$ 次调用 repair，消除了与这组术语宽度相关的因素。Figure 8 表明调用 repair 的次数与进行同余维护的时间相关。

3.2.2 **同余性的证明.** 直观上，重建是对向上合并过程的延迟，允许用户选择何时恢复 e-graph 不变性。它们在结构上实质上相似，关键的区别在于代码何时运行。下面我们提供了一个证明，证明重建恢复了 e-graph 同余不变性。

Theorem 3.1. 重建恢复了同余并终止。(*Rebuilding restores congruence and terminates.*)

Proof. 因为重建只合并等价节点，即使 $\equiv_{\text{node}}$ 改变，同余闭包 $\cong^*$ 也是固定的。当 $(\equiv_{\text{node}}) = (\cong^*)$ 时，同余被恢复。注意 $\equiv_{\text{node}}$ 和 $\cong^*$ 都是有限的。因此，我们证明重建导致 $\equiv_{\text{node}}$ 接近 $\cong^*$。我们定义不同余 e-node 对的集合 $I = (\cong^*) \setminus (\equiv_{\text{node}})$；换句话说，当 $(n_1, n_2) \in I$ 时 $n_1 \cong^* n_2$ 但 $n_1 \not\equiv_{\text{node}} n_2$。

由于等价饱和性的加性特征，$\equiv_{\text{node}}$ 只会增加，因此 I 不会增加。然而，在 rebuild 循环中调用 repair 不一定会缩小 I。一些调用只是从工作列表中删除一个元素，但不修改 e-graph 。

设 W 是由 repair 处理的 e-classes 的工作列表，在 Figure 4 中，W 对应于 self.worklist 加上 todo 局部变量的未处理部分。我们证明每次调用 repair 都会在字典序上减少元组 $(|I|, |W|)$，直到 $(|I|, |W|) = (0, 0)$，因此重建以 $(\equiv_{\text{node}}) = (\cong^*)$ 终止。

给定 W 中的 e-class c，repair 检查 c 的父类是否有相同的尚未在同一个 e-class 中的 e-nodes 。

- 如果 c 的至少一对父类是同构的，重建会合并每一对 (p_1, p_2)，这会增加 W，但会减小 I。
- 如果没有找到这样的同构对，则什么都不做。那么，由于 c 来自工作列表，$|W|$ 将减少 1，因为 repair 没有添加任何东西。

由于 $(|I|, |W|)$ 按字典序降低，所以 $|W|$ 最终会达到 0，因此 rebuild 终止。注意，W 仅包含那些需要进行“向上合并”以检查同构父类的 e-classes 。因此，当 W 为空时，rebuild 已经有效地执行了向上合并。根据 Nelson [1980, Chapter 7]，$|I| = 0$。因此，当重建终止时，同余被恢复。

□

```
def equality_saturation(expr, rewrites):
  egraph = initial_egraph(expr)

  while not egraph.is_saturated_or_timeout():


    # 读和写是混合的
    for rw in rewrites:
      for (subst, eclass) in egraph.ematch(rw.lhs):

        # 在传统的等价饱和中，
        # 匹配可以立即被应用，
        # 因为不变量始终保持。
        eclass2 = egraph.add(rw.rhs.subst(subst))
        egraph.merge(eclass, eclass2)

        # 在每次合并后恢复不变量
        egraph.rebuild()

  return egraph.extract_best()
```

(a) 传统的等式饱和和在搜索和应用规则之间交替，而 e-graph 在整个过程中保持其不变性。

```
def equality_saturation(expr, rewrites):
  egraph = initial_egraph(expr)

  while not egraph.is_saturated_or_timeout():
    matches = []

    # 只读阶段，不变性被保留
    for rw in rewrites:
      for (subst, eclass) in egraph.ematch(rw.lhs):
        matches.append((rw, subst, eclass))

    # 写入阶段，暂时破坏不变性
    for (rw, subst, eclass) in matches:
      eclass2 = egraph.add(rw.rhs.subst(subst))
      egraph.merge(eclass, eclass2)

    # 在每次迭代后恢复不变量
    egraph.rebuild()

  return egraph.extract_best()
```

(b) egg 将等式饱和的迭代分为读和写阶段。e-graph 不变性没有被持续维护，而只是在每次迭代结束时由 `rebuild` 方法恢复 (Section 3)。

Fig. 5. 传统版本和 egg 版本的等式饱和算法的伪代码。

3.3 重建 和 等式饱和

重建提供了在何时强制执行 e-graph 不变量的选择，如果因工作列表的去重而延迟，则可能节省工作。客户端在最大化性能而不限制应用的时间负责重建。

egg 提供了一种修改过的等式饱和算法，以利用重建的优势。图 Figure 5 显示了传统等式饱和和 egg 的变体的伪代码，其中体现了两个关键差异：

(1) 每次迭代都分为读取阶段和应用阶段，读取阶段搜索所有重写匹配，应用阶段应用这些匹配。[6]
(2) 重建仅在每次迭代结束时进行一次。

egg 对读取和写入阶段的分离意味着重写是真正无序的。在传统的等式饱和中，给定重写列表中的后续重写更受青睐，因为它们可以在同一次迭代中“看到”先前重写的结果。因此，如果未达到饱和 (这在大重写列表或输入表达式中很常见)，结果将取决于重写列表的顺序。egg 的等式饱和算法对重写列表的顺序不敏感。

将读取和写入阶段分开也允许 egg 安全地推迟重建。如果在传统的等式饱和算法中推迟重建，则重写列表中的后续规则将针对具有破坏不变性的 e-graph 进行搜索。由于可能不满足同余，可能缺少等价性，导致缺少匹配。这些匹配将在下一次迭代的 `rebuild` 期间被看到 (如果发生另一次迭代)，但错误报告可能会影响度量收集、规则调度、[7] 或饱和检测。

3.4 重建的评估

为了证明延迟重建比传统的上升合并更快地提供同余闭包，我们修改了 egg 在每次 `merge` 后立即调用 `rebuild`。这提供了一对一的延迟重建和传统方法之间的比较，并从许多其他使 egg 有效率的因素中隔离开来：整体设计、算法差异、编程语言性能和其他垂直性能改进。

[6]尽管最初的等式饱和论文 [Tate et al. 2009] 没有单独的读取和写入阶段，但一些 e-graph 实现（如 Z3 [De Moura and Bjørner 2008] 中的实现）由于实现细节而将这些阶段分开。我们的算法是第一个利用这一点的算法，通过推迟不变性维护来实现。
[7]在 Figure 5.2 中引入的优化依赖于对重写被匹配次数的准确计数。

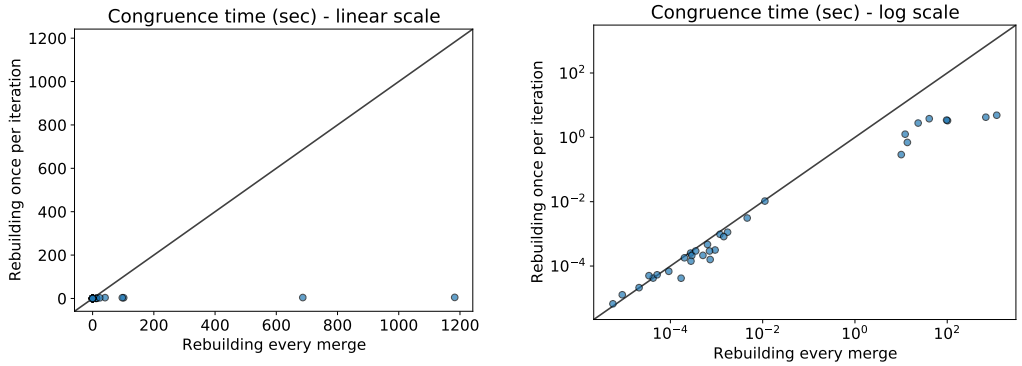


Fig. 6. **每次迭代只重建一次——而不是每次合并后——可显著加快同构维护。两个图显示相同的数据: 每个 32 测试一个点。对角线是 $y = x$; 点在线下面意味着推迟重建更快。在所有测试中的总和 (使用几何平均值), 同构快 $88\times$ ， 等式饱和快 $21\times$ 。线性尺度图显示推迟重建显著更快。对数尺度图表明加速比某些常数倍大; Figure 7更详细地展示这一点。**

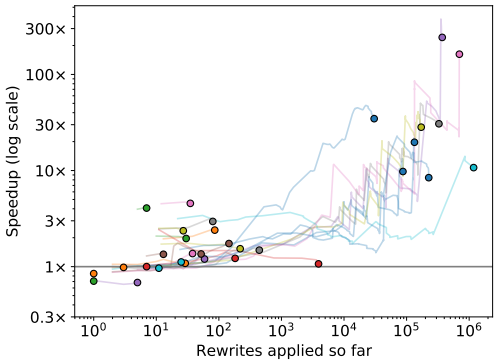


Fig. 7. **当应用更多重写时，推迟重建会带来更大的加速。每条线表示一个测试：每次等式饱和迭代绘制目前已经应用的累计重写与推迟重建的乘法加速率; 点表示该测试结束。整个测试套件（点）和单独的测试（线）均表明随着问题规模的增大而增加的渐近加速。**

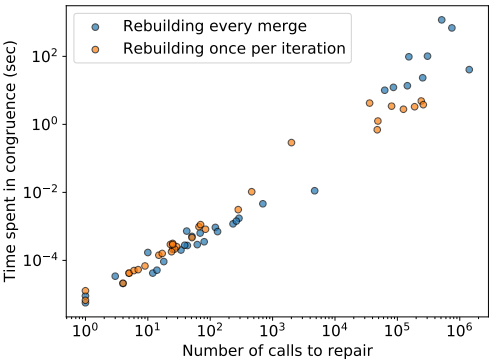


Fig. 8. **同构维护中的时间与 `repair` 方法的调用次数相关。Spearman 相关系数为 $r = 0.98$， p 值为 3.6e-47 ， 表明这两个量确实是正相关的。**

我们使用两种重建策略运行了 egg 的测试套件，并测量了在同余维护上花费的时间。每个测试都包括一次 egg 的等式饱和算法，以优化给定表达式。在总共 32 的测试中，8 达到了 100 次迭代的限制，其余达到了饱和状态。请注意，两种重建策略均使用 egg 的分阶段等式饱和算法，并且所有情况下的 e-graphs 都是相同的。这些实验在带有 2 GHz 四核 Intel Core i5 处理器和 16GB 内存的 2020 Macbook Pro 上进行。

图 Figure 6 显示了重建如何加速同余维护。总体而言，我们的实验显示整体上同余闭包上有 88× 加速，整个等式饱和算法上有 21× 加速。图 Figure 7 显示这种加速是渐近的，随着问题规模变得更大，乘法加速率增加。

egg 的测试套件由两个主要应用组成: `math`，一个小型计算机代数系统，能够进行符号导数和积分；`lambda`，一个无类型 lambda 演算的部分解释器，使用显式替换来处理变量绑

定（如 Section 5 所示）。这两者都是典型的 egg 应用，主要由语法重写驱动，并使用了 egg 复杂的关键功能，如 e-class 分析和动态或条件重写。

可以配置 egg 来捕获运行过程中等式饱和的各种度量，包括读取阶段（搜索匹配）、写入阶段（应用匹配）和重建阶段的花费时间。在 Figure 6 中，同余时间被测量为应用匹配和重建的时间。等式饱和算法的其他部分（创建初始 e-graph，提取最终项）与等式饱和迭代相比花费非常小。

推迟重建对 e-classes 的同余维护检查进行了摊销; 去重工作列表减少了对 `repair` 的调用次数。Figure 8 显示，同余中的时间与 `repair` 方法的调用次数相关。

在 Section 6.1 中的案例研究进一步评估了重建。重建（和其他 egg 特性）也已经在基于 Racket 的 e-graph 中实现，演示重建是为了概念推进，不用与 egg 实现相关联。

4 用 E-CLASS 分析器拓展 E-GRAPHS

正如迄今为止所讨论的，e-graphs 和等式饱和提供了一种高效的方法来实现一个项重写系统。重建增强了效率，但该方法仍然仅限于语法重写。然而，程序分析和优化通常需要的不仅仅是语法信息。相反，转换的计算是基于输入项和该输入项的语义论据（semantic facts），例如，常量值，自由变量，空属性，数字符号，内存大小等。“纯语法”限制迫使现有的等式饱和应用 [Panchekha et al. 2015; Stepp et al. 2011; Tate et al. 2009] 不得不求助于临时措施通过 e-graph 来实现常量折叠等分析。这些临时措施需要手动操纵 e-graph，其复杂性可能会妨碍实施更复杂的分析。

我们提出了一种新技术，称为 *e-class 分析器*（*e-class Analyses*），它允许在 e-graph 上简洁地表达程序分析。一个 e-class 分析器类似于提升到 e-graph 层面的抽象解释，从半格（semilattice）到每个e-class 附加 *分析数据*。在合并 e-classes 和添加新的 e-nodes 时，e-graph 会维护和传播这些数据。分析数据可以直接用于修改 e-graph，以告知重写如何或是否应用其右右部，或在萃取过程中确定项的成本。

e-class 分析器提供了一种通用机制，用来替换以前需要手动操纵 e-graph 的特殊扩展。e-class 分析器也适合等式饱和工作流程，因此它们可以自然地与重写提供的等式推理（equational reasoning）协作。此外，在 e-graph 层面的分析会自动从一种“偏序归约（partial-order reduction）”中免费受益：借由 e-graph 的紧凑表示，大量类似程序可以以用很少的额外成本进行分析。

本节提供了 e-class 分析器的概念解释以及可以使用分析数据的动态和条件重写。接下来的章节将提供具体示例: Section 5 讨论 egg 实现和 lambda 演算中部分求值器的完整示例; Section 6 讨论三个已发布项目如何使用 egg 及其独特特性（如 e-class 分析器）。

4.1 e-class 分析器

e-class 分析器（e-class Analyses）定义了一个域 D 并将一个值 $d_c \in D$ 与每个 e-class c 关联。e-class c 包含关联数据 d_c，即，给定一个 e-class c，可以很容易地获得 d_c，但反过来不行。

e-class 分析器的接口如下，其中 G 指的是 e-graph, n 和 c 指的是 G 中的 e-nodes 和 e-classes：

$\text{make}(n) \to d_c$	当一个新的 e-node n 被添加到 G 并且形成一个新的，单一的 e-class c 时，构造一个新值 $d_c \in D$ 与 n 的新 e-class 关联，通常通过访问 n 的子节点的关联数据。
$\text{join}(d_{c_1}, d_{c_2}) \to d_c$	当 e-classes c_1, c_2 被合并成 c 时，将 d_{c_1}, d_{c_2} 合并成一个新值 d_c 与新 e-class c 关联。
$\text{modify}(c) \to c'$	根据 d_c 可选地修改 e-class c，通常是添加一个 e-node 到 c。如果 e-class 没有其他变化，修改应该幂等，即 $\text{modify}(\text{modify}(c)) = \text{modify}(c)$

```
1  def add(enode):
2    enode = self.canonicalize(enode)
3    if enode in self.hashcons:
4      return self.hashcons[enode]
5    else:
6      eclass = self.new_singleton_eclass(enode)
7      for child_eclass in enode.children:
8        child_eclass.parents.add(enode, eclass)
9      self.hashcons[enode] = eclass
10      eclass.data = analysis.make(enode)
11      analysis.modify(eclass)
12     return eclass
13
14 def merge(eclass1, eclass2)
15   union = self.union_find.union(eclass1, eclass2)
16   if not union.was_already_unioned:
17     d1, d2 = eclass1.data, eclass2.data
18     union.eclass.data = analysis.join(d1, d2)
19     self.worklist.add(union.eclass)
20   return union.eclass
```

```
21 def repair(eclass):
22   for (p_node, p_eclass) in eclass.parents:
23     self.hashcons.remove(p_node)
24     p_node = self.canonicalize(p_node)
25     self.hashcons[p_node] = self.find(p_eclass)
26
27   new_parents = {}
28   for (p_node, p_eclass) in eclass.parents:
29     p_node = self.canonicalize(p_node)
30     if p_node in new_parents:
31       self.union(p_eclass, new_parents[p_node])
32     new_parents[p_node] = self.find(p_eclass)
33   eclass.parents = new_parents
34
35   # 任何对 eclass 的修改
36   # 都会添加到 worklist 中
37   analysis.modify(eclass)
38   for (p_node, p_eclass) in eclass.parents:
39     new_data = analysis.join(
40       p_eclass.data,
41       analysis.make(p_node))
42     if new_data != p_eclass.data:
43       p_eclass.data = new_data
44       self.worklist.add(p_eclass)
```

Fig. 9. **维护 e-class 分析器不变量的伪代码大体上与重建维持同余闭包的方式相似**（Section 3）。**只添加了行 10–11, 17–18, 和 37–44 。灰色或未列出的代码与 Figure 4 相同。**

域 D 与 join 操作一起应该形成一个联并半格（join-semilattice）。半格视角对于定义 分析不变量是很有用的。（其中 $\wedge$ 是 textsfjoin 操作）。

$$\forall c \in G. \quad d_c = \bigwedge_{n \in c} \text{make}(n) \quad \text{and} \quad \text{modify}(c) = c$$

第一部分的分析不变性声明，每个 e-class 的关联的数据必须是每个 e-node 的 make 的 join。由于 D 是联并半格（join-semilattice），这意味着 $\forall c, \forall n \in c, d_c \geq \text{make}(n)$。第二部分的动机更为微妙。由于分析可以通过 modify 方法修改一个e-class ，分析不变性声明这些修改被驱动到一个固定点。当分析不变性成立时，客户端看到的分析数据可以确保分析在重新计算 make，join，modify 不会修改 e-graph 或任何分析数据的意义下是“稳定”的。

4.1.1 **保持分析不变量**（*Analysis Invariant*）. 我们将重建过程从 Section 3 扩展到恢复分析不变性和同余不变性。Figure 9显示了从Figure 4修改重建代码所需的修改。

添加 e-nodes 和合并 e-classes 都有可能以不同的方式打破分析不变性。添加e-nodes 是更简单的情况；行 10–11 恢复了新创建的，单个 e-class 的不变性。合并e-nodes 时，第一个关注点是维护分析不变性的半阶部分。由于 join 在分析数据的域 D 上形成半阶，因此合并的顺序并不重要。因此，线18足以更新合并的e-class 的分析数据。

由于 make(n) 通过查看 n 的子元素创建分析数据，因此合并 e-classes 可能会在同一方式中违反分析不变性。解决方案是使用在 Section 3 中引入的相同工作列表机制。`repair` 方法（它在工作列表的每个元素上 `rebuild`）的行 37–44 重新 make 和 merge 最近合并的 e-classes 父元素的分析数据。新的 `repair` 方法还调用 modify 一次，这足以满足其幂等性（idempotence）。在伪代码中，为了更明晰，modify 被重构为一种可变方法。

egg 实现的 e-class 分析器假设分析域 D 确实是半格，并且 modify 具有幂等性。如果没有这些性质，egg 可能无法在 `rebuild` 上恢复分析不变量，或者它可能永不终止。

4.1.2 **示例: 常量折叠**（*Constant folding*）. e-class 分析器产生的数据可以被其他等式饱和系统组件有效地使用 (参见 Section 4.2), 但是由于 modify 钩子的存在，e-class 分析器可以独立使用典型的 modify 钩子会什么事都不做，检查要合并的 e-classes 的某些不变量，或者将一个 e-node 添加到该 e-class 中 (使用 e-graph 的常规 add 和 `merge` 方法)。

如上所述，其他等式饱和实现已经将常量折叠实现为自定义的，临时通过 e-graph 的步骤。我们可以将常量折叠表示为一个 e-class 分析器, 突出了它与抽象解释之间的相似之处。设域 D = `Option<Constant>`，设 `join` 操作是 `Option` 类型上的 “or” 操作：

```
match (a, b) {
  (None,    None   ) => None,
  (Some(x), None   ) => Some(x),
  (None,    Some(y)) => Some(y),
  (Some(x), Some(y)) => { assert!(x == y); Some(x) }
}
```

注意如何 join 也可以通过检查在 e-graph 中统一的值的属性来辅助调试; 在这种情况下, 我们断言在 e-class 中表示的所有项应该具有相同的常量值。make 操作充当抽象函数，如果它可以从与其子e-classes 相关联的常量值计算出来，返回 e-node 的常量值。modify 操作在这个设置中充当具体化函数。如果 d_c 是一个常量值，那么 modify(c) 就会将 $\gamma(d_c) = n$ 添加到 c 中，其中 γ 将常量值具体化（concretizes）为没有子节点的 e-node 。

常量折叠是一种明显简单的分析，但之前并不适合等式饱和框架。E-class analyses 在通用的方式中支持更复杂的分析，如后面关于 egg 实现和案例研究的章节所讨论的（5 和 6）。

4.2 条件和动态重写

在等式饱和应用中，大多数重写都是纯粹的语法形式。在某些情况下，可能需要额外的数据来确定是否执行重写或如何执行重写。例如，$x/x \to 1$ 重写仅在 $x \neq 0$ 时有效。更复杂的重写可能需要根据左侧的分析事实动态计算右侧。

重写的右侧可以概括为函数 apply，该函数采用替代和通过 e-matching 左侧得到的 e-class ，并产生一个项，该项将被添加到 e-graph 中并与匹配的 e-class 统一。对于纯粹的语法重写，apply 函数无需以任何方式检查匹配的 e-class ； 它只需将替代应用于右侧模式以产生新术语即可。

e-class 分析器大大增加了这种重写的通用形式的效用。apply 函数可以查看匹配的 e-class 的分析数据或替代中的任何 e-classes ，以确定如何构造右侧术语。这些重写可以进一步分为两类：

- 条件重写，如 $x/x \to 1$，它们是纯粹的语法形式，但其有效性取决于检查一些分析数据。
- 动态重写，它们根据分析数据计算右侧。

条件重写是更一般动态重写的子集。我们的 egg 实现支持两者。在 Section 5 中的示例和在 Section 6 中的案例研究中大量使用了广义重写，因为这通常是将领域知识纳入等式饱和框架的最方便的方法。

4.3 萃取（Extraction）

等式饱和通常以萃取阶段结束，该阶段根据某些成本函数从 e-class 中选择最优表示术语。在许多领域 [Nandi et al. 2020; Panchekha et al. 2015] 中，AST（抽象语法树）大小（有时对不同运算符的权重不同）足以作为简单的局部成本函数。当我们可以从函数符号 f 和子结点的成本计算出术语 $f(a_1, ...)$ 的成本时，称成本函数 k 为局部的。使用这样的成本函数（cost function），从 e-graph 中萃取出最佳项可以通过固定点遍历 e-graph 在每个 e-class 中选择具有最小代价的 e-node 来高效地完成 [Panchekha et al. 2015]。

萃取可以在成本函数是局部的情况下被表述为（formulated）为 e-class 分析器。分析数据是一个元组 $(n, k(n))$，其中 n 是该 e-class 中最代价最小的 e-node ，$k(n)$ 是它的代价。make(n) 操作基于 n 的子节点的分析数据（其中包含最小花费）计算 $k(n)$。merge 操作只需取较低代价的元组。分析不变量的半格部分（semilattice portion）保证了分析数据将包含每个类中最低花费的 e-node 。提取可以递归进行；如果 e-class c 的分析数据给出 $f(c_1, c_2, ...)$ 作为最佳 e-node ，则 c 表示的最佳项是 $\text{extract}(c) = f(\text{extract}(c_1), \text{extract}(c_2), ...)$。这不仅进一步证明了 e-class 分析器的普遍性，还提供了“即时”提取的能力；条件和动态重写可以根据分析数据来确定其行为。

萃取（无论是作为单独的过程还是作为 e-class 分析器）也可以从分析数据中受益。通常，局部成本函数只能查看 e-node n 的函数符号和 n 的子节点的成本。但是，当 e-class 分析器附加到 e-graph 时，成本函数可以观察与 n 的 e-class 关联的数据，以及与 n 的子节点关联的数据。这允许成本函数依赖于计算出的论据（facts）而不仅仅是纯语法信息。换句话说，运算符的成本可能因其输入而不同。Section 6.2 提供了一个激励性的案例研究（motivating case study ），其中 e-class 分析器将计算张量的大小和形状，这些大小信息会影响成本函数。

5 egg: 易用, 可拓展, 高效率的 E-GRAPHS

我们在 egg 中实现了重构和 e-class 分析技术，它是一个易于使用、可扩展、高效的 e-graph 库。据我们所知，egg 是第一个通用、可重用的 e-graph 实现。这使得我们可以专注于易用性和优化, 因为任何优点都将在不同的使用场景中体现, 而不是仅限于一个特定的情况。

本节详细说明了 egg 的实现和它为用户提供的各种优化和工具。我们使用了一个对于 lambda 演算的部分求值器的扩展示例 [8]，完整的源代码在 Figure 10 和 Figure 11 中提供（仅做了少量修改以方便阅读）。尽管示例稍显做作, 但它紧凑易懂, 并突出了 (1) egg 的使用方式和 (2) 其中一些新特性, 比如 e-class 分析和动态重写。它演示了如何借助 egg 的可扩展性，比如使用简单的显式替换方法来解决绑定问题——这是一个对于e-graphs 来说永恒的难题。Section 6 提供了更多真实世界的案例研究, 展示了依赖 egg 的已经发布的项目。

egg 由 ~5000 行包括代码、测试和文档的 Rust [9]实现。egg 是开源的、文档齐全的, 通过 Rust 的包管理系统发布。[10] egg 所有组件都是针对用户提供的语言、分析和成本函数的通用组件。

5.1 易用

egg 的易用性主要来自其作为库的设计。通过只定义一种语言和一些重写规则，用户可以快速开始开发合成或优化工具。作为一个 Rust 库来使用 egg ，用户可以使用 `define_language!` 宏定义语言。示例见 Figure 10 ，行 1-22。语言中的子节点为空的变量可以包含用户定义类型的数据，e-class 分析或动态重写可以审查这些数据。

用户可以提供重写规则，见 Figure 10，行 51-100 。每个重写规则都有一个名称，一个左部和一个右部。对于纯语法重写，右部仅仅是一个模式。更复杂的重写可以包含条件甚至是动态的右部，这些在 Section 5.2 和 Figure 11 中有解释。

无论应用领域如何，等式饱和工作流程通常具有相似的结构：向空的 e-graph 添加表达式，运行重写直到饱和或超时，并根据一些代价函数提取最佳等价表达式。这种“外循环（outer loop）”的等式饱和涉及大量的容易出错的繁文缛节：

- 饱和、超时和 e-graph 大小限制的检测。
- 协调读取阶段、写入阶段和重建系统（Figure 4）来加速 egg 。
- 在每次迭代中记录性能数据。

[8] E-graphs 没有任何“内置”的绑定支持; 例如，“同模（equality modulo）阿尔法重命名（alpha renaming）”不是无成本的。本节中提供的显式替换相当耗性能，未来的重要工作是更好的支持有绑定的语言。
[9] Rust 是一种高级系统编程语言。egg 已经被集成到其他编程语言编写的应用程序中，使用 C FFI 和序列化方法。
[10] 源码: https://github.com/mwillsey/egg. 文档: https://docs.rs/egg. 包: https://crates.io/crates/egg.

```
define_language! {
  enum Lambda {
    // 枚举变量具有数据或子元素 (eclass Ids)
    // [Id; N] 是 N 个 Id 的数组

    // 基础类型操作符
    "+" = Add([Id; 2]), "=" = Eq([Id; 2]),
    "if" = If([Id; 3]),

    // 函数和绑定
    "app" = App([Id; 2]), "lam" = Lambda([Id; 2]),
    "let" = Let([Id; 3]), "fix" = Fix([Id; 2]),

    //  (var x) 是使用`x`作为表达式
    "var" = Use(Id),
    //  (subst a x b) 在 b 中替换 (var x) 的 a
    "subst" = Subst([Id; 3]),

    // 基础类型没有子元素，只有数据
    Bool(bool), Num(i32), Symbol(String),
  }
}

// 示例项和它们简化为的内容
// 直接从 egg  测试套件中提取

test_fn! { lambda_under, rules(),
  "(lam x (+ 4 (app (lam y (var y)) 4)))"
  => "(lam x 8))",
}

test_fn! { lambda_compose_many, rules(),
  "(let compose (lam f (lam g (lam x
                (app (var f)
                     (app (var g) (var x))))))
    (let add1 (lam y (+ (var y) 1))
    (app (app (var compose) (var add1))
         (app (app (var compose) (var add1))
              (app (app (var compose) (var add1))
                   (app (app (var compose) (var add1))
                        (app (app (var compose) (var add1))
                             (var add1))))))))"
  => "(lam ?x (+ (var ?x) 5))"
}

test_fn! { lambda_if_elim, rules(),
  "(if (= (var a) (var b))
       (+ (var a) (var a))
       (+ (var a) (var b)))"
  => "(+ (var a) (var b))"
}

// 返回重写规则列表
fn rules() -> Vec<Rewrite<Lambda, LambdaAnalysis>> { vec![

 // open term rules 开放项规则
 rw!("if-true";  "(if  true ?then ?else)" => "?then"),
 rw!("if-false"; "(if false ?then ?else)" => "?else"),
 rw!("if-elim";  "(if (= (var ?x) ?e) ?then ?else)" => "?else"
     if ConditionEqual::parse("(let ?x ?e ?then)",
                              "(let ?x ?e ?else)")),
 rw!("add-comm";  "(+ ?a ?b)"        => "(+ ?b ?a)"),
 rw!("add-assoc"; "(+ (+ ?a ?b) ?c)" => "(+ ?a (+ ?b ?c))"),
 rw!("eq-comm";   "(= ?a ?b)"        => "(= ?b ?a)"),

 // substitution introduction 替换引入
 rw!("fix";     "(fix ?v ?e)" =>
                "(let ?v (fix ?v ?e) ?e)"),
 rw!("beta";    "(app (lam ?v ?body) ?e)" =>
                "(let ?v ?e ?body)"),

 // substitution propagation 替换传播
 rw!("let-app"; "(let ?v ?e (app ?a ?b))" =>
                "(app (let ?v ?e ?a) (let ?v ?e ?b))"),
 rw!("let-add"; "(let ?v ?e (+   ?a ?b))" =>
                "(+   (let ?v ?e ?a) (let ?v ?e ?b))"),
 rw!("let-eq";  "(let ?v ?e (=   ?a ?b))" =>
                "(=   (let ?v ?e ?a) (let ?v ?e ?b))"),
 rw!("let-if";  "(let ?v ?e (if ?cond ?then ?else))" =>
                "(if (let ?v ?e ?cond)
                     (let ?v ?e ?then)
                     (let ?v ?e ?else))"),

 // substitution elimination 替换消除
 rw!("let-const";    "(let ?v ?e ?c)" => "?c"
     if is_const(var("?c"))),
 rw!("let-var-same"; "(let ?v1 ?e (var ?v1))" => "?e"),
 rw!("let-var-diff"; "(let ?v1 ?e (var ?v2))" => "(var ?v2)"
     if is_not_same_var(var("?v1"), var("?v2"))),
 rw!("let-lam-same"; "(let ?v1 ?e (lam ?v1 ?body))" =>
                     "(lam ?v1 ?body)"),
 rw!("let-lam-diff"; "(let ?v1 ?e (lam ?v2 ?body))" =>
     { CaptureAvoid {
        fresh: var("?fresh"), v2: var("?v2"), e: var("?e"),
        if_not_free: "(lam ?v2 (let ?v1 ?e ?body))"
                     .parse().unwrap(),
        if_free: "(lam ?fresh (let ?v1 ?e
                              (let ?v2 (var ?fresh) ?body)))"
                 .parse().unwrap(),
     }
     if is_not_same_var(var("?v1"), var("?v2"))),
]}
```

Fig. 10. egg 是针对用户定义语言的通用框架；在这里，我们为 lambda 计算的部分求值器定义了语言和重写规则。提供的 `define_language!` 宏（行 1-22）允许简单地将语言定义为 Rust enum，可自动派生出解析器和产生漂亮的输出的打印器。Lambda 类型的值是一个 e-node，它保存用户可以检查的数据或一些 e-class 子节点（e-class Ids）。

也可以简洁地定义重写规则（行 51-100）。模式被解析为 s-expressions：从 `define_language!` 调用中解析的字符串（如: `fix`, `=`, `+`）和从变体（variant）中解析的数据（如: `false`, `1`）解析为运算符或项（term）；以 "?" 为前缀的名称解析为模式变量。

其中一些重写是条件重写，使用 "`left => right if cond`" 语法。在 57 行的 `if-elim` 重写使用 egg 提供的 `ConditionEqual` 作为条件，只有在 e-graph 可以证明两个参数模式等价时才应用右式。最终的重写 `let-lam-diff` 是动态的，用来支持捕获避免（capture avoidance）；右边是一个实现了 `Applier` 的 trait 而不是模式的 Rust 值。Figure 11 包含了这些重写的支持代码。

我们也展示了 egg's lambda 测试套件的一些测试（行 27-50）。测试的过程是在左边插入 term，运行 egg 的等式饱和，然后检查以确保右部的模式可以在与初始 term 相同的 e-class 中找到。

- 潜在地协调规则的执行，以便像结合律（associativity）这样的扩张性（expansive）规则不会在 e-graph 中占主导地位。
- 最后，根据用户定义的代价函数提取最佳表达式。

egg 通过其 `Runner` 和 `Extractor` 接口提供了这些功能。`Runners` 会自动检测饱和状态，并可以配置为在特定时间、e-graph 大小或迭代次数限制后停止。由 egg 提供的等式饱和循环会调用 `rebuild`，因此用户甚至不需要了解 egg 的延迟不变性（deferred invariant）维护。`Runners` 自动记录每次迭代的各种指标，用户可以钩入（hook）此过程以报告相关数据。`Extractors` 根据用户定义的局部代价函数从 e-graph 中选择最优项。[11] 这两者也可以结合起来；用户通常在每次迭代中萃取来记录“到目前为止最好”的表达。

Figure 10 也展示了 egg 的用于轻松创建测试的 `test_fn!` 宏（行 27-50）。这些测试使用给定表达式创建一个 e-graph，使用 `Runner` 运行等式饱和，并检查右部的模式是否能够在与初始表达式相同的 e-class 中找到。

5.2 可拓展性

对于简单的领域，定义语言和纯语法重写就足够了。但是，我们的部分评估器需要解释性推理，因此我们使用了一些 egg 更高级的功能，如 e-class 分析和动态重写。重要的是，egg 作为库支持这些可扩展性功能：用户无需修改 e-graph 或 egg 的内部。

Figure 11 展示了我们的 lambda 部分求值器的代码的剩余部分。它使用一个 e-class 分析（`LambdaAnalysis`）来跟踪与每个 e-class 相关联的自由变量和常量。e-class 分析的实现在第 11-50 行。e-class 分析不变量保证了分析数据包含来自该 e-class 表示的 term 的自由变量的上近似（或作“过近似”，Over-approximation）。该分析还进行了常量折叠（请参阅 `make` 和 `modify` 方法）。`let-lam-diff` 重写（Figure 10，第 90 行）使用 `CaptureAvoid`（Figure 11，第 81-100 行）的动态右部，根据自由变量的信息仅在必要时进行捕获避免地替换（capture-avoiding substitution）。Figure 10 的条件重写取决于条件 `is_not_same_var` 和 `is_var`（Figure 11，第 68-74 行）以确保正确的替换。

egg 在其他方面也是可扩展的。如上所述，`Extractor` 由用户提供的成本函数来参数化。`Runner` 也可以使用用户提供的规则调度程序进行扩展，以控制潜在有问题的重写的行为。在典型的等式饱和中，每次迭代都会搜索和应用每个重写。这可能导致某些重写（通常是结合率或分配率）占据其他重写的地位，使搜索空间变得不够高效。适量使用这些重写可以触发其他重写并找到更好的表达式，但它们也可能会使搜索变慢，因为它们会使 e-graph 的大小指数级增长。默认情况下，egg 使用内置的退避调度程序（backoff scheduler），该调度程序识别在指数级位置匹配的重写并暂时禁用它们。我们已经观察到，这在许多情况下大大减少了运行时间（产生相同的结果）。egg 也可以使用常规的每规则每次调度器（every-rule-every-time scheduler），或者用户自定义的调度器。

5.3 效率

egg 的新颖的重建算法 (Section 3) 与系统编程最佳实践相结合，使得 e-graphs ——尤其是等式饱和使用案例——比之前的工具更有效率。

egg 是用 Rust 实现的，这使得编译器可以自由地特化（specialize）和内联（inline）用户编写的代码。这非常重要，因为 egg 的通用性导致了库代码（例如搜索重写）和用户代码（例如比较运算符）之间的紧密交互。egg 从头开始设计，使用缓存友好的、带有最少间接层（indirection）的平面缓冲区（flat buffers），用于大多数内部数据结构。这与包含许多树和链表类似的数据结构的传统 e-graphs 表示形式 [Detlefs et al. 2005; Nelson 1980] 形成鲜明对比。与递归遍历模式的树状表示法相比，egg 另外还编译了模式，以便由一个小型虚拟机执行 [de Moura and Bjørner 2007]。

[11] 正如在 Section 4.3 中提到的，萃取可以作为 e-class 分析的一部分实现。由于人性化和性能原因，独立的 `Extractor` 功能仍然有用。

```
type EGraph = egg::EGraph<Lambda, LambdaAnalysis>;
struct LambdaAnalysis;
struct FC {
  free: HashSet<Id>,    // 我们的分析数据存储自由变量
  constant: Option<Lambda>, // 以及常量值（如果有）
}

// 帮助函数，用于制作模式元变量（pattern meta-variables）
fn var(s: &str) -> Var { s.parse().unwrap() }

impl Analysis<Lambda> for LambdaAnalysis {
  type Data = FC; // 将 FC 附加到每个 eclass
  // merge 通过合并到 “to” 实现半格联并（semilattice join）
  // 如果 “to” 数据被修改，则返回 true
  fn merge(&self, to: &mut FC, from: FC) -> bool {
    let before_len = to.free.len();
    // union the free variables 联并自由变量
    to.free.extend(from.free.iter().copied());
    if to.constant.is_none() && from.constant.is_some() {
      to.constant = from.constant;
      true
    } else {
      before_len != to.free.len()
    }
  }

  fn make(egraph: &EGraph, enode: &Lambda) -> FC {
    let f = |i: &Id| egraph[*i].data.free.iter().copied();
    let mut free = HashSet::default();
    match enode {
      Use(v) => { free.insert(*v); }
      Let([v, a, b]) => {
        free.extend(f(b)); free.remove(v); free.extend(f(a));
      }
      Lambda([v, b]) | Fix([v, b]) => {
        free.extend(f(b)); free.remove(v);
      }
      _ => enode.for_each_child(
            |c| free.extend(&egraph[c].data.free)),
    }
    FC { free: free, constant: eval(egraph, enode) }
  }

  fn modify(egraph: &mut EGraph, id: Id) {
    if let Some(c) = egraph[id].data.constant.clone() {
      let const_id = egraph.add(c);
      egraph.union(id, const_id);
    }
  }
}
```

```
// 如果子元素有常量，评估 enode
// Rust的 `?` 提取一个 Option，如果是 None，则提前返回。
fn eval(eg: &EGraph, enode: &Lambda) -> Option<Lambda> {
  let c = |i: &Id| eg[*i].data.constant.clone();
  match enode {
    Num(_) | Bool(_) => Some(enode.clone()),
    Add([x, y]) => Some(Num(c(x)? + c(y)?)),
    Eq([x, y]) => Some(Bool(c(x)? == c(y)?)),
    _ => None,
  }
}

// 这种类型的函数可以作为重写的条件
trait ConditionFn = Fn(&mut EGraph, Id, &Subst) -> bool;

// 以下两个函数返回正确签名的闭包，
// 它可用作 Figure 10 中的条件
fn is_not_same_var(v1: Var, v2: Var) -> impl ConditionFn {
    |eg, _, subst| eg.find(subst[v1]) != eg.find(subst[v2])
}
fn is_const(v: Var) -> impl ConditionFn {
     // check the LambdaAnalysis data
    |eg, _, subst| eg[subst[v]].data.constant.is_some()
}

struct CaptureAvoid {
  fresh: Var, v2: Var, e: Var,
  if_not_free: Pattern<Lambda>, if_free: Pattern<Lambda>,
}

impl Applier<Lambda, LambdaAnalysis> for CaptureAvoid {
  // 给定egraph、匹配的 eclass id 和匹配生成的替换，
  // 应用重写
  fn apply_one(&self, egraph: &mut EGraph,
               id: Id, subst: &Subst) -> Vec<Id>
  {
    let (v2, e) = (subst[self.v2], subst[self.e]);
    let v2_free_in_e = egraph[e].data.free.contains(&v2);
    if v2_free_in_e {
      let mut subst = subst.clone();
      // 使用eclass id制作新的符号 (fresh symbol)
      let sym = Lambda::Symbol(format!("_{}", id).into());
      subst.insert(self.fresh, egraph.add(sym));
      // 使用修改后的 subst 应用于给定的模式
      self.if_free.apply_one(egraph, id, &subst)
    } else {
      self.if_not_free.apply_one(egraph, id, &subst)
    }
  }
}
```

Fig. 11. 我们的部分评估器示例突出了 egg 提供扩展性的三个重要特性：e-class 分析、条件重写和动态重写。

LambdaAnalysis 类型实现了 Analysis trait，表示 e-class 分析。它的关联数据 (FC) 存储来自该 e-class 的常量项（如果有）和该 e-class 中项目使用的自由变量的上近似（over-approximation）。常量项用于进行常量折叠。merge 操作实现了半格联并（semilattice join），结合自由变量集并采用常量（如果存在）。在 make 中，分析基于 e-node 和它的子节点的自由变量集计算自由变量集；如果可能，eval 生成新的常量。Analysis 的 modify 钩子将常量添加到 e-graph 中。

Figure 10 中的一些条件重写取决于这里定义的条件。任何具有正确签名的函数都可以作为条件。

CaptureAvoid 类型实现了 Applier trait，允许它作为重写的右式。CaptureAvoid 接受两个模式和一些模式变量。它检查自由变量集来确定是否需要捕获避免的替换，如果需要，则应用 if_free 模式，否则应用 if_not_free 模式。

除了延迟重建外，egg 的等式饱和算法还带来了实现层面的性能增强。搜索重写匹配，这是运行时间的主要部分，可以通过阶段分离并行化。规则或 e-classes 可以并行搜索。此外，每次迭代重建的频率允许 egg 建立其他在只读搜索阶段期间保持的性能增强不变量，。此外，每次迭代一次（once-per-iteration）的重建频率允许 egg 建立其他增强性能的不变量，这些不变量在只读搜索阶段保持不变。例如，egg 在每个 e-class 内排序 e-nodes 以启用二分查找，并维护将函数符号映射到包含具有该函数符号的 e-nodes 的 e-classes 的缓存。

egg 的许多可扩展性功能也可用于提高性能。如上所述，规则调度可以在面对“扩展性”规则时带来巨大的性能改进，否则这些规则将主导搜索空间。`Runner` 接口也支持用户钩子，可以在任意条件后停止 equality saturation。当使用等式饱和来证明项目相等时，这是非常有用的; 一旦它们一致，就没有继续的必要了。egg 的 `Runner` 也支持批量简化，在运行等式饱和之前可以将多个 term 添加到初始 e-graph 。如果这些 term 显着相似，重写和任何 e-class 分析都将从 e-graph 的固有结构去重复中受益。Section 6.1 中的案例研究使用批量简化来实现简化相似表达式的大幅加速。

6 案例研究

本节介绍了三个独立开发的、来自不同领域的公开项目，它们将 egg 作为一种易用的、高性能的 e-graph 实现。在所有这三个案例中，开发者首先推出了他们自己的 e-graph 实现。由于 egg 的速度和灵活性，他们可以删除代码，获得性能，并在某些情况下极大地扩大了项目的范围。除了获得性能外，这三个项目都使用了 egg 的新颖的可扩展性特性，如 e-class 分析和动态/条件重写。

6.1 Herbie：提高浮点精度

Herbie 自动提高了浮点表达式的准确性，使用随机抽样来测量误差，使用一套重写规则来生成程序变体，并使用算法来修剪和组合程序变体以达到最小误差。Herbie 获得了 PLDI 2015 的杰出论文奖 [Panchekha et al. 2015]。从那时起，Herbie 一直在不断发展。在 Github 上获得了数以百计的 star 和下载量，在其在线版本上有成千上万的用户。Herbie 使用 e-graphs 对数学表达式进行代数简化，这对于避免由消解（cancellation）、函数反转（function inverses）和冗余计算（redundant computation）引入的浮点错误尤为重要。

在我们的案例研究之前，Herbie 使用了一个用 Racket（Herbie 的实现语言）编写的定制的 e-graph 实现，它紧跟传统的 e-graph 实现。在禁用超时的情况下，基于 e-graph 的简化工作消耗了 Herbie 的绝大部分运行时间。作为一种修正，Herbie 对简化过程进行了严格的限制，对 e-graph 本身设置了大小限制，对整个过程设置了时间限制。当超过时间限制时，简化过程就会完全失败。此外，Herbie 的作者还知道一些他们认为可以改善 Herbie 输出的功能，但却无法实现，因为它们需要更多的简化调用，因此会带来不可接受的减速。总的来说，缓慢的简化过程降低了 Herbie 的性能、完整性和有效性。

我们为 Herbie 实现了一个 egg 简化的后端。egg 后端比 Herbie 的初始简化器快 3000×，现在已被 Herbie 1.4 已经默认使用。Herbie 还将一些 egg 的功能，如批量简化和重建移植到它的 e-graph 实现中（它仍然可用，只是不是默认的），证明了 egg 的概念改进的可移植性。

6.1.1 **实现**. Herbie 用 Racket 实现，而 egg 用 Rust 实现；因此 egg 的简化后端被实现为一个 Rust 库，提供一个 C 级 API，供 Herbie 通过外部函数接口 (FFI) 访问。Rust 库定义了 Herbie 表达式语法（带有命名常量、数值常量、变量和操作），以及进行常量折叠所需的 e-class 分析。该库用不到 500 行的 Rust 代码实现。

Herbie 的重写规则集并不固定；用户可以使用命令行标志来选择使用哪些重写。Herbie 将重写规则序列化为字符串，而 EGG 后端则在 Rust 端解析并实例化它们。

Herbie 分离了精确和不精确的程序常量：对精确常量的精确操作（如两个有理数的加法）会被评估并添加到 e-graph 中，而对不精确常量或产生不精确输出的操作则不会。因此，我们将 Rust 端语法中的数字常量分为精确的有理数和不精确的常量，它们由一个不透明的

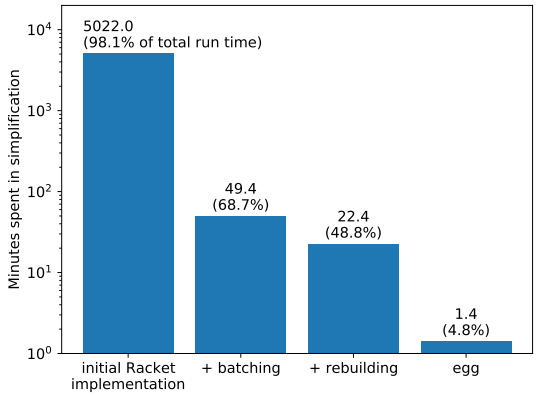


Fig. 12. Herbie **通过在基于** Racket **的** e-graph **实现中采用像批量简化和重建这样的** egg **启发式特性，来加速其表达式简化阶段。**Herbie **还支持使用** egg **本身来获得额外的加速。注意** y **轴是对数刻度。**

标识符（opaque identifier）描述，并将 Racket 端表达式转化为这种形式，然后将它们序列化并传递给 Rust 驱动。为了评估精确常量上的操作，我们使用了常量折叠的 e-class 分析来跟踪每个 e-class 的“精确值”。[12] 每当一个运算符 e-node 被添加到 egg e-graph 时，我们会检查该操作的所有参数是否具有精确值（使用分析的数据），如果是则进行有理数运算去计算它。e-class 分析比 Herbie 实现中相应的代码更干净，因为它是整个 e-graph 的内建过程（built-in pass）。

6.1.2 **评估**. egg 简化器后端是现有 Herbie 简化器的插入式替代品（drop-in replacement），使比较速度和结果变得容易。我们使用 Herbie 的标准测试套件，大约有 500 个基准测试，其中禁用了超时。Figure 12 显示了测试结果。egg 简化后端比 Herbie 的初始简化器快了 3000×。这种加速消除了 Herbie 的最大瓶颈：原始实现占 Herbie 总运行时间的 98.1%，将 egg 改进移植到 Herbie 中将其减少到总运行时间的一半，egg 简化器占总运行时间的不到 5% 。实际上，Herbie 初始实现的运行时间更短，因为超时会导致简化时间过长时测试失败。因此加速也提高了 Herbie 的完备性，因为简化过程现在永远不会超时，。

自从将 egg 纳入 Herbie 以来，Herbie 开发人员已经将一些 egg 的关键性能改进移植到了 Racket e-graph 实现中。首先，批量简化给出了很大的加速，因为 Herbie 简化了许多相似的表达式。在一个等式饱和的同时进行时，e-graph 的结构共享可以大量地减少重复工作。其次，推迟重建 (如 Section 3 中所述) 给出了进一步的 2.2× 加速。如 Figure 7 所示，重建提供了渐近加速，因此 Herbie 的改进实现（以及 egg 后端）将随着搜索规模的增长而更好地扩展（scale）。

6.2 Spores: 优化线性代数

Spores [Wang et al. 2020] 是一个机器学习程序的优化器。它将线性代数（LA）表达式翻译成关系代数（RA），进行重写，最后将结果再翻译成线性代数。每一次重写都是由关系代数中的简单特性建立起来的，比如连接的关联性。与教科书上的线性代数特性相比，这些关系特性表达了更精细的平等性，使得 Spores 能够发现传统优化器基于 LA 特性所不能发现的新型优化。Spores 进行整体优化（holistic optimization），考虑到诸如稀疏性、共同子表达式和可融合运算符等因素之间复杂的相互作用以及它们对执行时间的影响。

[12]Herbie 的重写规则保证了不同的精确值永远不能变得相等；半格 **加入**（semilattice join）在 Rust 端检查这个不变量。

$$A \oplus (B \oplus C) = \oplus(A, B, C) \qquad (\oplus \text{ 满足结合律 \& 交换律}) \tag{1}$$

$$A \otimes (B \otimes C) = \otimes(A, B, C) \qquad (\otimes \text{ 满足结合律 \& 交换律}) \tag{2}$$

$$A \otimes (B \oplus C) = A \otimes B \oplus A \otimes C \qquad (\otimes \text{ 对于 } \oplus \text{ 满足分配率}) \tag{3}$$

$$\sum_i (A \oplus B) = \sum_i A \oplus \sum_i B \tag{4}$$

$$\sum_i \sum_j A = \sum_{i,j} A \tag{5}$$

$$A \otimes \sum_i B = \sum_i (A \otimes B) \qquad (\text{要求 } i \notin A) \tag{6}$$

$$\sum_i A = A \otimes \text{dimension}(i) \qquad (\text{要求 } i \notin A) \tag{7}$$

Fig. 13. RA **等式规则** R_{EQ}.

6.2.1 **实现**. Spores 完全用 Rust 和 egg 实现。egg 赋予 Spores 优雅而轻松地编排上述复杂交互。Spores 工作三个步骤：首先，它将输入的 LA 表达式转换为 RA; 其次，它通过等式饱和优化 RA 表达式; 最后，它将优化后的 RA 表达式转换回 LA。由于 LA 和 RA 之间的转换是直接的，因此我们将讨论集中在 RA 中的等式饱和步骤上。Spores 将关系表示为从元组到实数的函数: $A : (a_1, a_2, ..., a_n) \to \mathbb{R}$。这类似于线性代数中的索引符号，其中矩阵 A 可以被视为函数 $\lambda i, j.A_{ij}$。元组与命名记录相同，例如 $(1, 2) = \{a_1 : 1, a_2 : 2\} = \{a_2 : 2, a_1 : 1\}$，因此元组中的顺序无关紧要。关系上只有三种操作: join, union 和 aggregate。Join (⊗) 接受两个关系并返回它们的自然 join，将连接元组的关联实数相乘：

$$A \otimes B = \lambda \bar{a} \cup \bar{b}.A(\bar{a}) \times B(\bar{b})$$

这里 $\bar{a}$ 是 A 中记录的字段名集合。在 RA 术语中，$\bar{a}$ 是 A 的 *schema*。Union (⊕) 是一种伪装的 join：它在其两个参数上也执行自然 join，但是添加而不是乘以关联实数：

$$A \oplus B = \lambda \bar{a} \cup \bar{b}.A(\bar{a}) + B(\bar{b})$$

最后，aggregate(Σ) 沿给定维度求和其参数。它与数学中的“sigma 符号”完全相同：

$$\sum_{a_i} A = \lambda \bar{a} - a_i. \sum_{a_i} A(\bar{a})$$

在 Figure 13 中呈现的 RA 指示器（identities）也是简单直观的。符号 $i \notin A$ 表示 i 不在 A 的模式中，$dim(i)$ 是维度 i 的大小（例如，矩阵中的行长度）。在Equation 6 中，当 $i \in A$ 时，我们首先将 i 重命名为 B 中的新变量 i'，这给我们：$A \otimes \sum_i B = \sum_{i'} (A \otimes B[i \to i'])$。除了这些等式之外，Spores 还支持用融合运算符替换表达式。例如，$(X - UV)^2$ 可以用 $sqloss(X, U, V)$ 替换，这将从 X, U, V 中流式传输值并在不创建中间矩阵的情况下计算结果。这些融合运算符中的每一个都在 egg 中以简单身份编码。

注意，在优化过程中，Equation 6 需要存储每个表达式的模式。Spores 使用 e-class 分析来为 e-classes 注释适当的模式，它还利用e-class 分析进行成本估算，使用过高估计的保守成本模型。因此，等价的表达式可能具有不同的成本估计。对分析数据的 `merge` 操作采用较低的成本，逐渐改善成本估计。最后，Spores 的 e-class 分析还执行常量折叠。总之，e-class 分析是三个较小分析的组合，类似于抽象解释中的格点（lattices）组合。

6.2.2 **评估**. Spores 已被集成到 Apache SystemML [Boehm 2019] 的原型中，在那里它能够推导出 84 个手写的规则（hand-written rules）和启发式算法来进行和积优化。它还发现了新

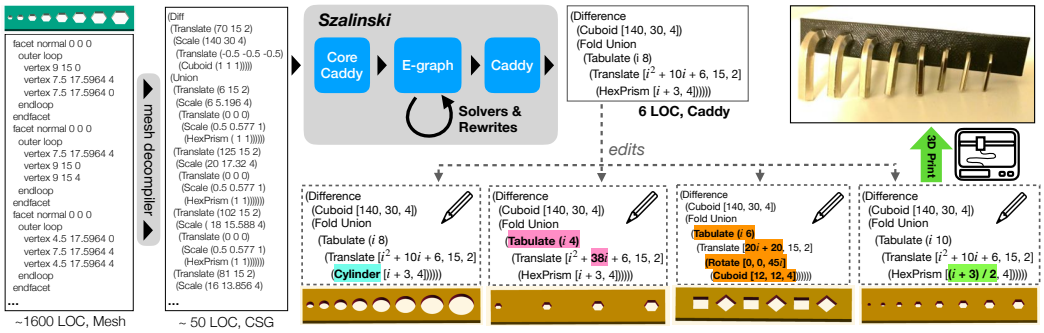


Fig. 14. （图源自 Nandi et. al. [Nandi et al. 2020]）现有的网格反编译器（mesh decompiler）将三角网格转换为平面的计算固体几何（CSG）表达式。Szalinski [Nandi et al. 2020] 在一种称为 Core Caddy 的格式中输入这些 CSG 表达式，基于 Caddy 语言综合（synthesizes）了更小的、结构化的程序，它具有函数式编程的特性。这可以通过简化编辑来简化定制：小的、主要是局部更改，来产生有用的不同模型。照片展示了定制孔尺寸后 3D 打印的六角扳手架。Szalinski 是基于 egg 高性能、e-class 分析和动态重写的可扩展等式饱和驱动的。

的重写，在端到端实验中贡献了 1.2× 到 5× 的加速。使用贪心提取（greedy extraction），所有编译能都在一秒钟内完成。

6.3 Szalinski: 将 CAD 反编译为结构化程序

目前已有几种工具可以从多边形网格和体素中逆向 CAD（高级计算机辅助设计，Computer Aided Design）模型 [Du et al. 2018; Ellis et al. 2018; Nandi et al. 2018; Sharma et al. 2017; Tian et al. 2019]。这些工具的输出是构造实体几何 (CSG) 程序。一个 CSG 程序由 3D 实体如立方体、球体、圆柱体、仿射变换（如缩放、平移、旋转）和二元运算符如并集、交集和差集组成，它们结合了 CSG 表达式。对于像齿轮这样的重复模型，CSG 程序可能太长，因此难以理解。一个近来的工具，Szalinski [Nandi et al. 2020]，通过自动推断映射（maps）和折叠（folds）（Figure 14），可以在网格反编译（mesh decompilation）工具的 CSG 输出中提取固有结构。Szalinski 通过 egg 的可扩展等式饱和系统实现了这一目标：

- 使用循环重投（loop rerolling）规则发现结构。这使得 Szalinski 能够从平面 CSG 输入中推断出 `Fold`, `Map2`, `Repeat` 和 `Tabulate` 等函数模式。
- 识别由网格解压器表示为不同表达式的 CAD 术语之间的等价性。Szalinski 通过使用 CAD id （CAD identities）来实现这一点。Szalinski 中的一个 CAD id 示例是 $e \leftrightarrow \text{rotate } [0\ 0\ 0]\ e$。这意味着任何 CAD 表达式 e 等价于对 e 应用 x, y, z 轴上零度旋转的 CAD 表达式。
- 使用外部求解器向 e-graph 中添加可能有利可图的表达式。网格解压器经常生成以非直观方式排序和/或分组列表元素的 CSG 表达式。为了从这样的表达式中恢复结构，Szalinski 这样的工具必须能够重新排序和重新分组列表，暴露任何潜在结构。

6.3.1 **实现**. 尽管 CAD 与编程语言技术针对的传统语言不同，但是 egg 以直接的方式支持 Szalinski 的 CAD 语言。Szalinski 使用纯语法重写来表示 CAD 身份和一些循环重绕规则（如从 CAD 表达式列表推断出 `Fold`）。然而关键的是，Szalinski 依赖于 egg 的动态重写和 e-class 分析来推断列表的函数。

考虑 Figure 15b 中的平面 CSG 程序。结构发现重写首先将平面的 `Union` 列表重写为:

```
(Fold Union (Map2 Translate [(0 0 0) (2 0 0) ...] (Repeat Cube 5)))
```

(a) **一条线中的五个立方体。**

```
(Union
  (Translate (0 0 0) Cube)
  (Translate (2 0 0) Cube)
  (Translate (4 0 0) Cube)
  (Translate (6 0 0) Cube)
  (Translate (8 0 0) Cube))
```

(b) Flat CSG **输入到** Szalinski 。

```
(Fold Union
  (Tabulate (i 5)
    (Translate
      ((* 2 i) 0 0)
      Cube)))
```

(c) **输出捕获了重复。**

Fig. 15. Szalinski **将求解器作为动态重写集成到** egg **的等式饱和中。支持求解器的重写可以将重复列表转换为捕获重复结构的** `Tabulate` **表达式。**

向量列表存储为 `Cons` 元素（上面简化了）。Szalinski 使用 e-class 分析来跟踪累积列表，类似于常量折叠。然后，动态重写使用算术求解器重写分析数据中的具体 3D 向量列表，以 `(Tabulate (i 5) (* 2 i))` 的形式。最后一组语法重写可以提升（hoist）`Tabulate`，得到 Figure 15 右侧的结果。由于一系列 CAD 语法重写的存在，即使面对 CAD id，这种结构发现也能正常工作。例如，原始程序可能省略无操作的 `Translate (0 0 0)`，即使它是观察重复结构所必需的。

在很多情况下，输入 CSG 表达式的重复结构进一步混淆，因为子表达式可能以任意顺序出现。对于这些输入，算术求解器必须首先重新排序表达式，以找到像 `Tabulate` 这样的封闭形式，如 Figure 15 所示。然而，重新排序列表不会保留等价性，因此将其添加到具体列表的 e-class 中将是不安全的。因此，Szalinski 引入了新技术 *反向变换*（*inverse transformations*），允许求解器推测重新排序和重新分组列表元素以找到封闭形式。求解器使用导致成功发现封闭形式的排列或分组注释可能有利可图的表达式。在重写过程的后期，语法重写在可能时删除反向变换（例如，在 `Fold Union` 下重新排序列表可以被删除）。egg 支持了这种新颖的技术，而无需修改。

6.3.2 **评估**. Szalinski 的初始原型使用 OCaml 编写的自定义 e-graph 。有趣的是，使用 egg 删除了大部分代码，消除了错误，促进了求解器支持的重写（solver-backed rewrites）和反向变换（inverse transformations）的关键贡献，并使工具快了约 1000×。egg 的性能允许从运行小型手工选择的示例转变为对来自 3D 模型共享论坛 [Nandi et al. 2020] 的 2000 多个真实世界模型的综合评估。

7 相关工作

Term 重写. Term 重写 [Dershowitz and Jouannaud 1990] 已被广泛用于促进程序优化 [Boyle et al. 1996; van den Brand et al. 2002; Visser et al. 1998] 的等式推理中。Term 重写系统将保留语义的重写或公理的数据库应用于输入表达式，以得到一个新的表达式，根据一些成本函数，这个表达式与输入式相比可能更有利。重写通常是符号化的，有一个左式和一个右式。为了对一个表达式进行重写，重写系统实现了模式匹配——如果重写规则的左侧与输入表达式相匹配，系统就会计算出一个替换，然后应用到重写规则的右侧。在应用重写规则时，重写系统通常会用新的表达式替换旧的表达式。这可能会导致 阶段排序（*phase ordering*）的问题——它使得在未来不可能对旧的表达式进行重写，而这可能会导致更优的结果。

E-graphs 和 E-matching. E-graph 最初是几十年前提出的一种高效的数据结构，用于维护同余闭包 [Kozen 1977; Nelson 1980; Nelson and Oppen 1980]。E-graphs 仍然是成功的 SMT 求解器中的关键组成部分，它们被用于通过共享等价关系结合可满足性理论 [De Moura and Bjørner 2008]。过去 e-graph 实现和 egg 的 e-graph 的主要区别在于我们新颖的重建算法——只在某些关键点维护不变性 (Section 3)。这使得 egg 更适合等价饱和的目的。egg 实现了 de Moura 等人提出的模式编译策略 [de Moura and Bjørner 2007]，该策略在最先进的定理证

明器 [De Moura and Bjørner 2008] 中使用。一些证明器 [De Moura and Bjørner 2008; Detlefs et al. 2005] 提出了像 mod-time，pattern-element 和 inverted-path-index 这样的优化，用于找到新的术语和相关模式进行匹配，避免重复匹配。到目前为止，我们发现 egg 比之前的几个 e-graph 实现更快，即使没有这些优化。然而，它们与 egg 的设计兼容，可以在未来进行探索。另一个关键差异是 egg 强大的 e-class 分析抽象和灵活的界面。它们使程序员能够轻松地利用 e-graph 解决涉及复杂语义推理的问题。

同余闭包. 我们的重建算法类似于 Downey et al. [1980] 提出的同余闭包（congruence closure）算法。重建的贡献不在于它如何恢复 e-graph 不变性，而在于它什么时候恢复；客户端有能力将不变性恢复专门化为特定工作负载，如等式饱和。它的算法也具有要进一步处理的合并工作列表，但它是离线的，即算法处理给定的等价关系并输出经过同构闭包的等价关系集。重建适用于在线 e-graph（和等式饱和）设置，在这种设置中重写经常检查当前的等价关系并断言新的关系。重建还传播 e-class 分析事实（Section 4.1）。尽管存在这些差异，但核心算法相似，离线性能特征 [Downey et al. 1980] 适用于两者。我们没有为在线设置提供重建的理论分析；它可能与工作负载高度相关。

超优化（Superoptimization）和等式饱和. Denali [Joshi et al. 2002] 超优化器首先展示了如何使用 e-graphs 进行优化代码生成，作为手动优化机器代码和先前的详尽方法 [Massalin 1987] 的替代方案，两者都不易扩展。Denali 的输入是类 C 语言中的程序，它会生成汇编程序。Denali 支持三种重写——算术、架构和特定于程序。在应用这些重写直到饱和之后，它使用硬件描述生成约束，使用 SAT 求解器解决这些约束以输出近似最优程序。虽然 Denali 的方法比先前的工作有了重大改进，但它只适用于直线代码，因此不适用于大型实际程序。

等式饱和 [Stepp et al. 2011; Tate et al. 2009] 开发了编译优化选择算法，适用于循环和条件等复杂语言结构。第一篇关于等价饱和的论文使用中间表示，称为程序表达图（PEGs）来编码循环和条件。PEGs 有专门的节点，可以表示无限序列，这样就可以表示循环。它使用全局可盈利性启发式来进行提取，使用伪布尔求解器实现。最近，[Premtoon et al. 2020] 使用 PEGs 进行代码搜索（code search）。egg 可以支持 PEGs 作为用户定义语言，因此它们的技术可以移植。

8 总结

我们提出了两种新技术：重建和 e-class 分析，使得等式饱和变快且足够可扩展，可用于一类新的应用程序。重建是一种新的方法，用于摊销维护 e-graph 数据结构不变性的成本，将 e-graph 针对等式饱和工作量来特化。e-class 分析是一个通用框架，可以提供超出纯语法重写所能提供的解释性推理。

我们在 egg 中实现了这两种技术，egg 是一个可重用、可扩展和高效的 e-graph 库。egg 是针对用户定义语言的通用库，这使得我们能够专注于优化和效率，同时避免了临时性的 e-graph 实现和操作的需要。我们的案例研究表明，等式饱和现在可以进一步拓展，并比以前更灵活地使用；egg 提供了新的功能和大幅性能加速。我们相信这些贡献将等式饱和定位为程序合成和优化的的强大工具箱。

致谢

感谢我们的匿名论文和工件审稿人的反馈。特别感谢我们的指导人 Simon Peyton Jones, Leonardo de Moura, 和 PLSE 小组的许多成员。这项工作得到了由 SRC 和 DARPA 共同赞助的 JUMP 中心的应用驱动体系结构（Applications Driving Architectures, ADA）研究中心的部分支持，以及 Grant Nos. 1813166 和 1749570 号国家科学基金会的支持。

REFERENCES

Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. 1999. Graph Transformation for Specification and Programming. *Sci. Comput. Program.*

34, 1 (April 1999), 1–54. https://doi.org/10.1016/S0167-6423(98)00023-9

Matthias Boehm. 2019. Apache SystemML. *Encyclopedia of Big Data Technologies* (2019), 81–86. https://doi.org/10.1007/978-3-319-77525-8_187

James M. Boyle, Terence J. Harmer, and Victor L. Winter. 1996. The TAMPR Program Transformation System: Simplifying the Development of Numerical Software. In *Modern Software Tools for Scientific Computing, SciTools 1996, Oslo, Norway, September 16-18, 1996*, Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen (Eds.). Birkhäuser, 353–372. https://doi.org/10.1007/978-1-4612-1986-6_17

Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (July 1960), 201–215. https://doi.org/10.1145/321033.321034

Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Nachum Dershowitz. 1993. *A taste of rewrite systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 199–228. https://doi.org/10.1007/3-540-56883-2_11

Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Rewrite Systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Jan van Leeuwen (Ed.). Elsevier and MIT Press, 243–320. https://doi.org/10.1016/b978-0-444-88074-1.50011-1

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. 1980. Variations on the Common Subexpression Problem. *J. ACM* 27, 4 (Oct. 1980), 758–771. https://doi.org/10.1145/322217.322228

Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: automatic conversion of 3D models to CSG trees. 1–16. https://doi.org/10.1145/3272127.3275006

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Neural Information Processing Systems (NIPS)*.

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 47–62.

Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-directed Superoptimizer. *SIGPLAN Not.* 37, 5 (May 2002), 304–314. https://doi.org/10.1145/543552.512566

Dexter Kozen. 1977. Complexity of Finitely Presented Algebras. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing* (Boulder, Colorado, USA) *(STOC '77)*. Association for Computing Machinery, New York, NY, USA, 164–177. https://doi.org/10.1145/800105.803406

Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems* (Palo Alto, California, USA) *(ASPLOS II)*. IEEE Computer Society Press, Washington, DC, USA, 122–126. https://doi.org/10.1145/36206.36194

Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional Programming for Compiling and Decompiling Computer-aided Design. *Proc. ACM Program. Lang.* 2, ICFP, Article 99 (July 2018), 31 pages. https://doi.org/10.1145/3236794

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

Charles Gregory Nelson. 1980. *Techniques for Program Verification*. Ph. D. Dissertation. Stanford, CA, USA. AAI8011683.

Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (April 1980), 356–364. https://doi.org/10.1145/322186.322198

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications* (Nara, Japan) *(RTA' 05)*. Springer-Verlag, Berlin, Heidelberg, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. *SIGPLAN Not.* 50, 6 (June 2015), 1–11. https://doi.org/10.1145/2813885.2737959

Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK)

(PLDI 2020). Association for Computing Machinery, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

Rust. [n. d.]. *Rust programming language*. https://www.rust-lang.org/

Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2017. CSGNet: Neural Shape Parser for Constructive Solid Geometry. *CoRR* abs/1712.08290 (2017). arXiv:1712.08290 http://arxiv.org/abs/1712.08290

Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 737–742.

Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. https://doi.org/10.1145/321879.321884

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rylNH20qFQ

Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. 2002. Compiling language definitions: the ASF+SDF compiler. *ACM Trans. Program. Lang. Syst.* 24, 4 (2002), 334–368. https://doi.org/10.1145/567097.567099

Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 13–26. https://doi.org/10.1145/289423.289425

Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proceedings of the VLDB Endowment* (2020).

Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry Compiler. *ACM Transactions on Graphics* 38, 6 (2019), Article No. 195. presented at SIGGRAPH Asia 2019.