编写时源码优化工具的研究与实现

Research and Implementation of Source Code Optimization Tools at Writing-time.

弗来米斯特

掀垫子柯基大学 - 赛博工程学院

2023 年 8 月 13 日

西安電子科技大學
XIDIAN UNIVERSITY

目前编写时源码优化工具的局限性

目前程序逻辑简化往往应用于程序编译时或运行时的优化，鲜有在编写时面向程序员可读性的建议与分析。

```
1  def foo(a):
2      return b * 2 / 2
```

(a) 给出警告：b 未定义

```
1  def bar(a):
2      return a * 2 / 2
```

(b) 无建议

本毕业设计旨在研究和实现一个在代码编写时环境下一个代码静态分析工具，能交互式地在代码编写时提供程序优化指导。

“Writing-time” 的额外要求

ISO/IEC 25010:2011[1] 是目前使用的国际软件质量评价标准，描述了软件质量的 **8 个质量特性** 和 31 个子特性。

适合性 性能效率 兼容性 易用性 可靠性 安全性

维护性 **可移植性**

[1] ``ISO / IEC 25010 : 2011 Systems and software engineering —Systems and software Quality Requirements and Evaluation (SQuaRE) —System and software quality models''. In: 2013.

使用形式化方法进行源码优化

形式化方法（Formal Methods）是基于严格数学基础，对计算机软 (硬) 件系统进行形式规约、开发和验证的技术。

形式化方法中主要有以下两个方向：形式规约与形式验证。[2]

形式验证方向包括定理证明和模型检验两个子方向，主要关系到软件安全领域。这些都是是程序静态分析的重要方法。[3]

[2] 王戟 et al. ``形式化方法概貌''. In: 软件学报 30.1 (2019), p. 33. DOI: 10.13328/j.cnki.jos.005652.

[3] 梅宏 et al. ``软件分析技术进展''. In: 计算机学报 9 (2009), p. 14.

项重写

将一个表达式进行重写，但不改变其意义就是项重写（term rewriting）。类如编译器中的各种优化技术可以看作为项重写。[4]

直接地对表达式进行重写可能是破坏性的：会失去原表达式的信息。这会导致编译器优化中的一个常见问题：阶段顺序问题（Phase ordering），即如何规划施加不同重写规则的各阶段才能够得到最优的结果。

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。

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

E-Graph

等式饱和（equality saturation）[5] 更好地解决了这一问题，它是一种使用 e-graph 有效地进行此重写的技术。

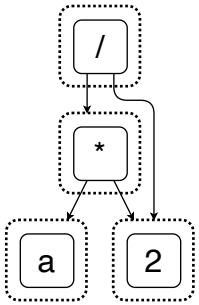
e 图（Equality graphs）扩展了并查集（union-find[6]），用来紧凑地表示表达式的等价类。

当应用任何重写后都不能向 e 图增加信息，我们就称 e 图就达到了 ``饱和'' 状态，即等式饱和。

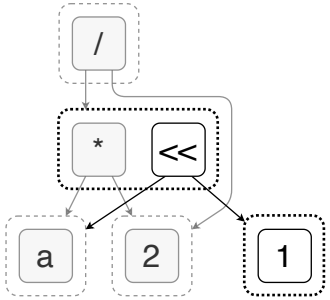
[5] Ross Tate et al. ``Equality Saturation: A New Approach to Optimization''. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. Savannah, GA, USA: ACM, 2009, pp. 264–276. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480915. URL: http://doi.acm.org/10.1145/1480881.1480915.

[6] Robert Endre Tarjan. ``Efficiency of a Good But Not Linear Set Union Algorithm''. In: *J. ACM* 22.2 (Apr. 1975), pp. 215–225. ISSN: 0004-5411. DOI: 10.1145/321879.321884. URL: https://doi-org.offcampus.lib.washington.edu/10.1145/321879.321884.

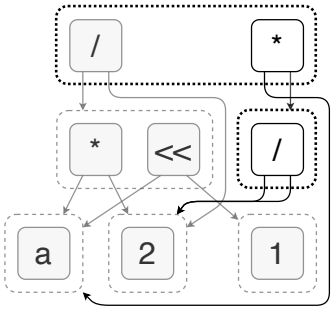
等式饱和：示例



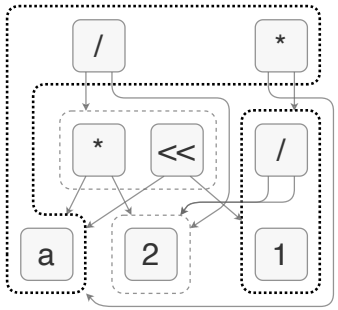
(a) 初始化包含 $(a \times 2)/2$ 的 e 图



(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 $1 \times x \to x$

Egg（egraphs good）

但是这种技术仍然不是通用的，等式饱和用户必须自己实现专门针对语言的 e 图，而且没办法直接实现纯语法重写无法支持的解释性推理。

Willsey 等人[7] 的工作 egg 旨在解决这些困难，使 e 图快速和方便扩展，使其专门用于等式饱和。egg 提供了两项技术：

- 重建：效率更高
- e 类分析：提供 e 图的更多拓展能力

[7] Max Willsey et al. ``egg: Fast and Extensible Equality Saturation''. In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.

一些使用 egg 的案例

Panchekha 等人[8] 研发的 Herbie 使用 egg 提取最小误差的表达式，可以自动提高浮点表达式的准确性。

Nandi 等人[9] 研发的 Szalinski 使用 egg 通过一套重写规则来生成程序变体，可以将 CAD 反编译为结构化程序。

缺乏泛用性，不能用在优化通用计算机程序语言上。

[8] Pavel Panchekha et al. ``Automatically Improving Accuracy for Floating Point Expressions''. In: *SIGPLAN Not.* 50.6 (June 2015), pp. 1–11. ISSN: 0362-1340. DOI: 10.1145/2813885.2737959. URL: https://doi.org/10.1145/2813885.2737959.

[9] Chandrakana Nandi et al. ``Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations''. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 31–44. ISBN: 9781450376136. DOI: 10.1145/3385412.3386012. URL: https://doi.org/10.1145/3385412.3386012.

egg 的中间表示 CommonLanguage

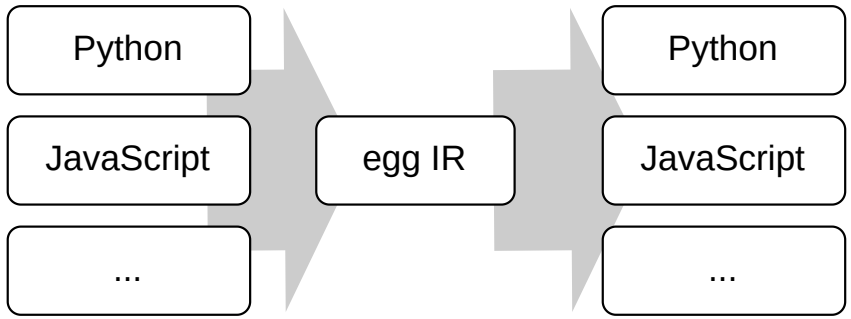
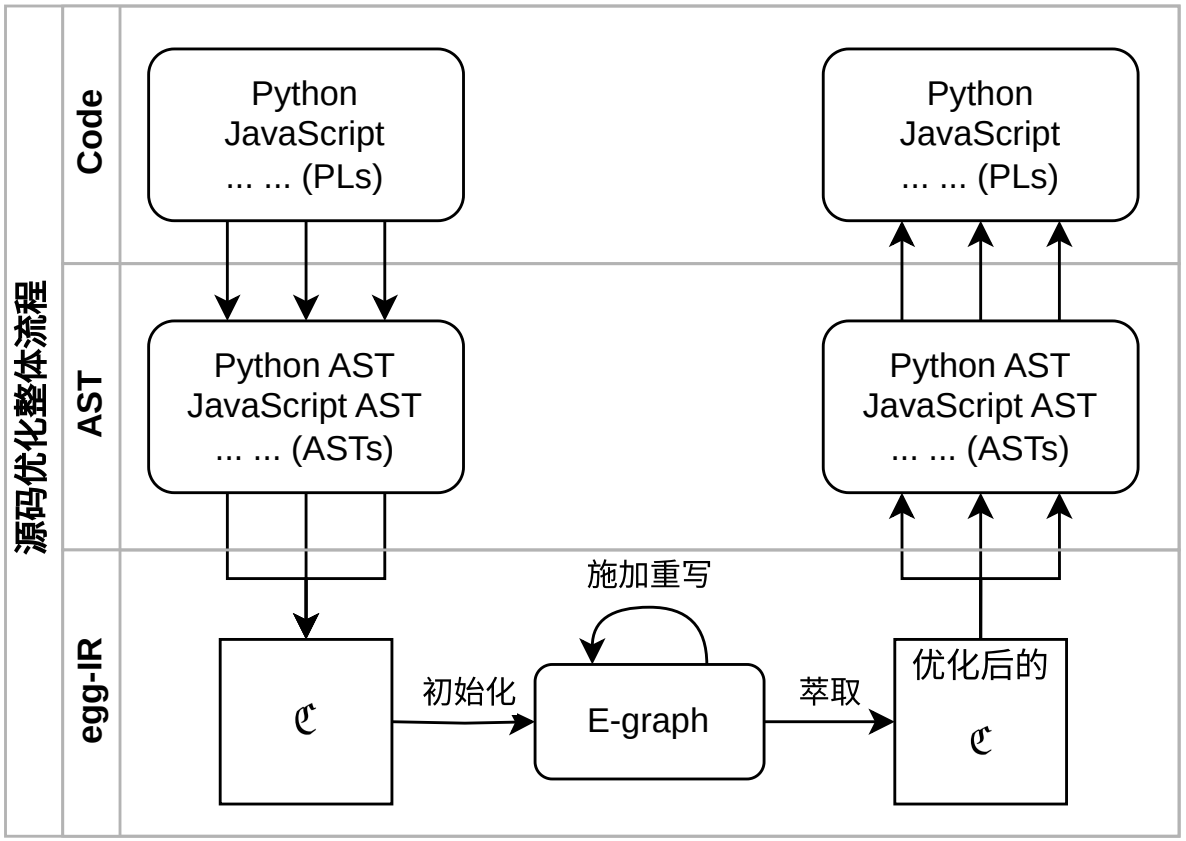


图 3: egg-IR 作为不同编程语言的共同中间表示

下面我们称之为中间语言（CommonLanguage，$\mathfrak{C}$）。

总体设计架构



1. Code → AST
2. AST → IR
3. IR ↔ IR
4. IR → AST
5. AST → Code

项与性质

定义 ($\mathfrak{C}$ 表达式)

BNF 文法定义 $\mathfrak{C}$ 的抽象语法：

<$\mathfrak{C}$> ::= <symbol> | $n \in \mathbb{N}$ | <bool> | <$\mathfrak{C}$> | (var <$\mathfrak{C}$>) | (= <$\mathfrak{C}$> <$\mathfrak{C}$>) | (app <$\mathfrak{C}$> <$\mathfrak{C}$>) | (lam <$\mathfrak{C}$> <$\mathfrak{C}$>) | (let <$\mathfrak{C}$> <$\mathfrak{C}$> <$\mathfrak{C}$>) | (fix <$\mathfrak{C}$> <$\mathfrak{C}$>) | (if <$\mathfrak{C}$> <$\mathfrak{C}$> <$\mathfrak{C}$>) | (+ <$\mathfrak{C}$> <$\mathfrak{C}$>) | (* <$\mathfrak{C}$> <$\mathfrak{C}$>) | (- <$\mathfrak{C}$> <$\mathfrak{C}$>) | (/ <$\mathfrak{C}$> <$\mathfrak{C}$>) | (pow <$\mathfrak{C}$> <$\mathfrak{C}$>) | (ln <$\mathfrak{C}$> <$\mathfrak{C}$>) | (sqrt <$\mathfrak{C}$> <$\mathfrak{C}$>) | (~ <$\mathfrak{C}$>) | (& <$\mathfrak{C}$> <$\mathfrak{C}$>) | (| <$\mathfrak{C}$> <$\mathfrak{C}$>) | (> <$\mathfrak{C}$> <$\mathfrak{C}$>) | (< <$\mathfrak{C}$> <$\mathfrak{C}$>) | (>= <$\mathfrak{C}$> <$\mathfrak{C}$>) | (<= <$\mathfrak{C}$> <$\mathfrak{C}$>) | (!= <$\mathfrak{C}$> <$\mathfrak{C}$>) | (cons <$\mathfrak{C}$> <$\mathfrak{C}$>) | nil | (laml <$\mathfrak{C}$> <$\mathfrak{C}$>) | (appl <$\mathfrak{C}$> <$\mathfrak{C}$>) | skip | (seq <$\mathfrak{C}$> <$\mathfrak{C}$>) | (seqlet <$\mathfrak{C}$> <$\mathfrak{C}$>)

<bool> ::= $\top$ | $\bot$

<Symbol> ::= // 其他未捕获的符号

$\mathfrak{C}$ 表达式示例

例 (一个 $\mathfrak{C}$ 表达式，添加缩进仅为易于阅读)

```
1  (let fib (fix fib (lam n
2  (if (= (var n) 0)
3      0
4  (if (= (var n) 1)
5      1
6  (+ (app (var fib)
7          (- (var n) 1))
8      (app (var fib)
9          (- (var n) 2))))))))
```

重写与替换

定义 ($\mathfrak{C}$ 项等价 $c_1 \equiv c_2$)

$$\forall \mathsf{st}, \mathsf{st}' \in \mathsf{state}, \forall c_1, c_2 \in \mathfrak{C}, \quad \left(\mathsf{st} \xRightarrow{[c_1]} \mathsf{st}'\right) \longleftrightarrow \left(\mathsf{st} \xRightarrow{[c_2]} \mathsf{st}'\right).$$

例 (定理 A.1 (comm-add))

$$\forall a, b, \quad (+\ a\ b) \equiv (+\ b\ a)$$

运用定理对 $\mathfrak{C}$ 进行重写和替换

例 (运用定理对 $\mathfrak{C}$ 进行重写和替换)

```
1  ;; 原始 𝔠
2  (lam x (+ 42
3      (app (lam y (var y))
4           24)))
5
6  ;; 运用 𝔠 定理进行重写
7  (lam x (+ 42 (app (lam y (var y)) 24)))
8  (lam x ([根据定理 comm-add 重写](+ (app (lam y (var y)) 24) 42)))
9  (lam x (+ ([根据定理 beta 重写] (let y 24 (var y))) 42))
10 (lam x (+ ([根据定理 let-var-same 重写] 24) 42))
11 (lam x ([常数折叠重写] 66))
12
13 ;; 优化后的 𝔠
14 (lam x 66)
```

其他要注意的、方案的其他部分

- 程序变换的可靠性
- 常量折叠替换
- 自由变量分析
- 成本函数
- 方案的其他部分：
  - 源码解析　粒度问题、变参问题（算法 3.1）
  - 伪码生成　逆波兰表达式的栈解析（算法 3.2）

篇幅有限，不再详述

总体实现流程

基于 egg 的源码优化主要分为以下过程的具体实现：

步骤 1. Code $\rightarrow$ AST：通过 Tree-sitter 将源码解析为 AST。

步骤 2. AST $\rightarrow$ IR：针对特定目标语言分别实现的 AST 转 $\mathfrak{C}$ 算法。

步骤 3. IR $\leftrightarrow$ IR: 由 $\mathfrak{C}$ 表示。通过 egg 进行 Rewrite。

步骤 4. IR $\rightarrow$ AST：CommonLanguage 自动派生方法。

步骤 5. AST $\rightarrow$ Code：针对特定目标语言分别实现的逆波兰表达式的栈解析。

使用 egg 来定义、化简语言的基本流程

1. **定义语言**：使用 `define_language!` 宏定义一个枚举类型来表示 $\mathfrak{C}$ 的元素。
2. **创建规则**：创建包含了所有需要应用的规则的 `Rewrite` 类型的向量。并且使用 `e` 类分析器用自定义的算法拓展重写规则。
3. **优化表达式**：首先解析表达式，然后使用 `Runner` 类创建一个 e-graph，并在其上运行给定的规则。最后，使用 `Extractor` 类从根 e-class 中选择最佳元素。

具体定义 CommonLanguage

```
Num(i32),
Bool(bool),
Symbol(Symbol),
Other(Symbol, Vec<Id>),
```

(a) 一些基本元素

```
"&" = And([Id; 2]),
"~" = Not(Id),
"|" = Or([Id; 2]),
```

(b) 布尔运算

```
"+" = Add([Id; 2]),
"*" = Mul([Id; 2]),
"-" = Sub([Id; 2]),
"/" = Div([Id; 2]),
"pow" = Pow([Id; 2]),
"ln" = Ln(Id),
"sqrt" = Sqrt(Id),
```

(c) 算数运算

```
">" = Gt([Id; 2]),
"<" = Lt([Id; 2]),
">=" = Ge([Id; 2]),
"<=" = Le([Id; 2]),
"!=" = Ne([Id; 2]),
```

(d) 关系运算

具体定义 CommonLanguage

```
"var" = Var(Id),
"=" = Eq([Id; 2]),
"app" = App([Id; 2]),
"lam" = Lambda([Id; 2]),
"let" = Let([Id; 3]),
"fix" = Fix([Id; 2]),
"if" = If([Id; 3]),
```

(a) 带变量运算与 λ 演算

```
"cons" = Cons([Id; 2]),
"nil" = Nil,
"laml" = LambdaL([Id; 2]),
"appl" = AppL([Id; 2]),
"skip" = Skip,
"seq" = Seq([Id; 2]),
"seqlet" = SeqLet([Id; 2]),
"while" = While([Id; 2]),
"for" = For([Id; 4]),
```

(b) 指令式语言的过程

创建规则

我们定义的 CommonLanguage 部分简单的重写性质的定理可以很容易地转换为 egg 重写规则。

例 (定理 A.20 (double-neg-flip) 与转换的重写规则)

$$\forall a, \quad \neg\neg a = a$$
$$\Rightarrow$$
$$\forall a, \quad (\sim(\sim a)) \equiv a$$
$$\Rightarrow$$

```
rewrite!("double-neg-flip"; "(~ (~ ?a))" => "?a"),
```

优化表达式总体流程

```rust
pub fn simplify(s: &str) -> Result<...> {
    let expr = s.parse();   // 解析 𝔈
    // 运行等式饱和
    let runner = Runner::default()
        .with_expr(&expr)
        .run(&make_rules());
    let root = runner.roots[0];
    // 使用提取器选择最佳元素
    let extractor = Extractor::new(&runner.egraph, 𝔈.
    CostFn);
    let (best_cost, best) = extractor.find_best(root);
    // 计算是否进行了显著的优化
    if best_cost < 𝔈.CostFn.cost_rec(&expr)
        Ok(Some(best))
}
```

语言插件整体结构

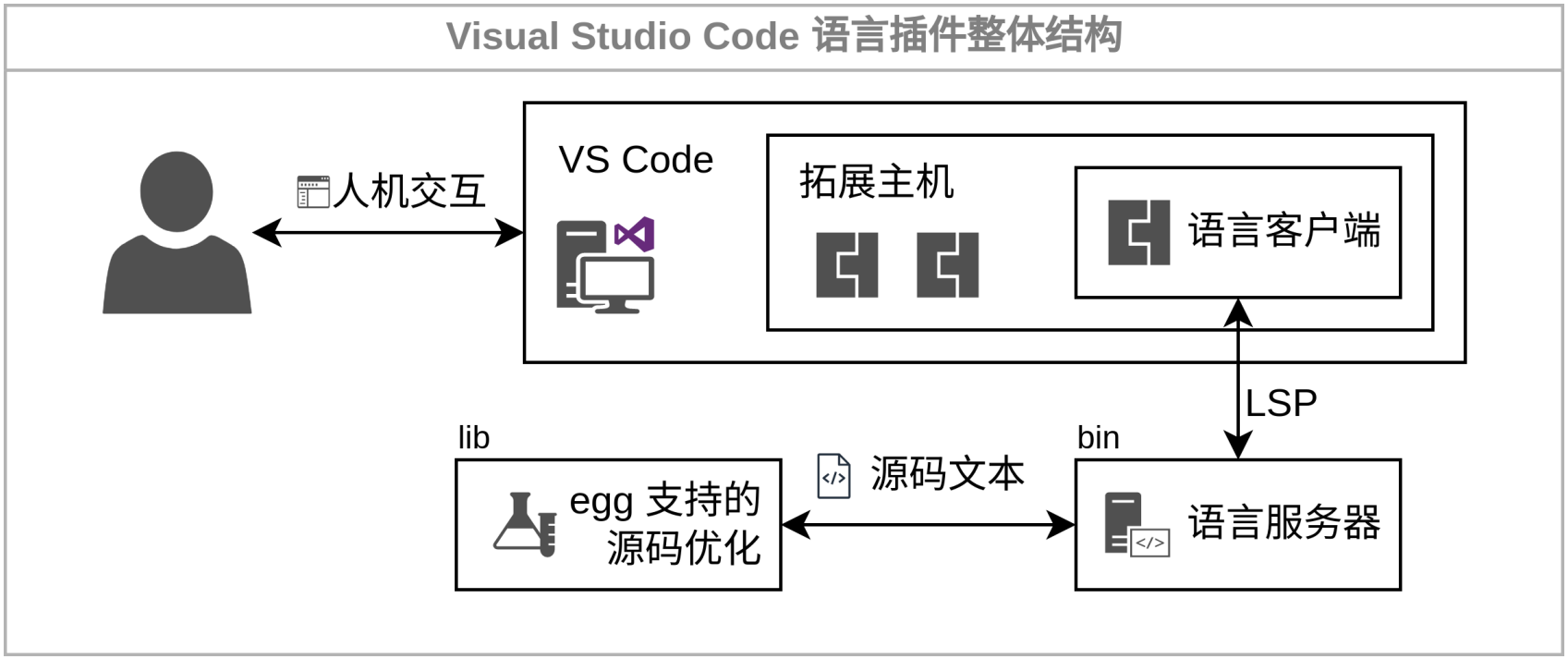


图 6: 语言插件的结构

语言服务器协议（Language Server Protocol）

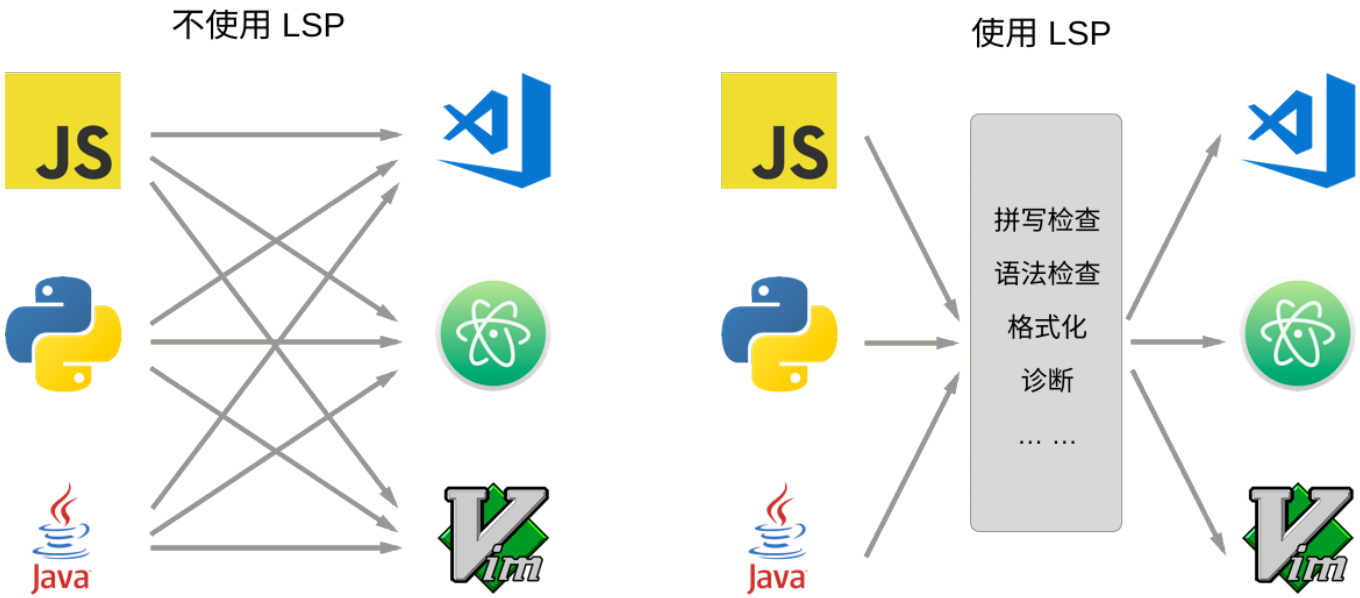


图 7: LSP[10] 的作用

[10] *LSP / LSIF*. 2022. URL: https://microsoft.github.io/language-server-protocol/.

源码解析【示例】

```python
def add1(x):
    x = x + 1
    return x
y = 1
add1(y)
```

(a) Python 源代码

```javascript
function add1(x) {
    return x + 1;
}
y = 1;
add1(y);
```

(b) JavaScript 源代码

```
(seq (seq (seqlet add1 (lam x (seq (seqlet x (+ (var x) 1)
    ) (var x)))) (seqlet y 1)) (app (var add1) (var y)))
```

(c) 解析出的 CommonLanguage

伪码生成【示例】

```
(seq (seqlet f (laml (cons x (cons y nil)) (laml (var x) (+ 42 (appl
     (laml (var y) (var y)) (cons 24 nil)))))) (appl (appl (var f)
    (cons 2 (cons 3 nil))) (cons 6 nil)))
```

(a) 被解析的 CommonLanguage

```
let f = (  x :: y :: nil:
         (  `x`:
                  (42 + (  `y`:
                           `y`)(24 :: nil))));
`f`(2 :: 3 :: nil)(6 :: nil)
```

(b) 逆向源码解析出的 debug 样式的伪代码

```
f = (lambda x, y: (lambda x:
     (42 + (lambda y: y)(24)))
     )
f(2, 3)(6)
```

(c) Python 样式的伪代码

```
let f = ((x, y) => ((x) => (42
      + ((y) => y)(24))));
f(2, 3)(6)
```

(d) JavaScript 样式的伪代码

寻找目标模式的时间

我们提供了很多已知的 $\mathfrak{C}$ 表达式与其最优的重写项作为测试，

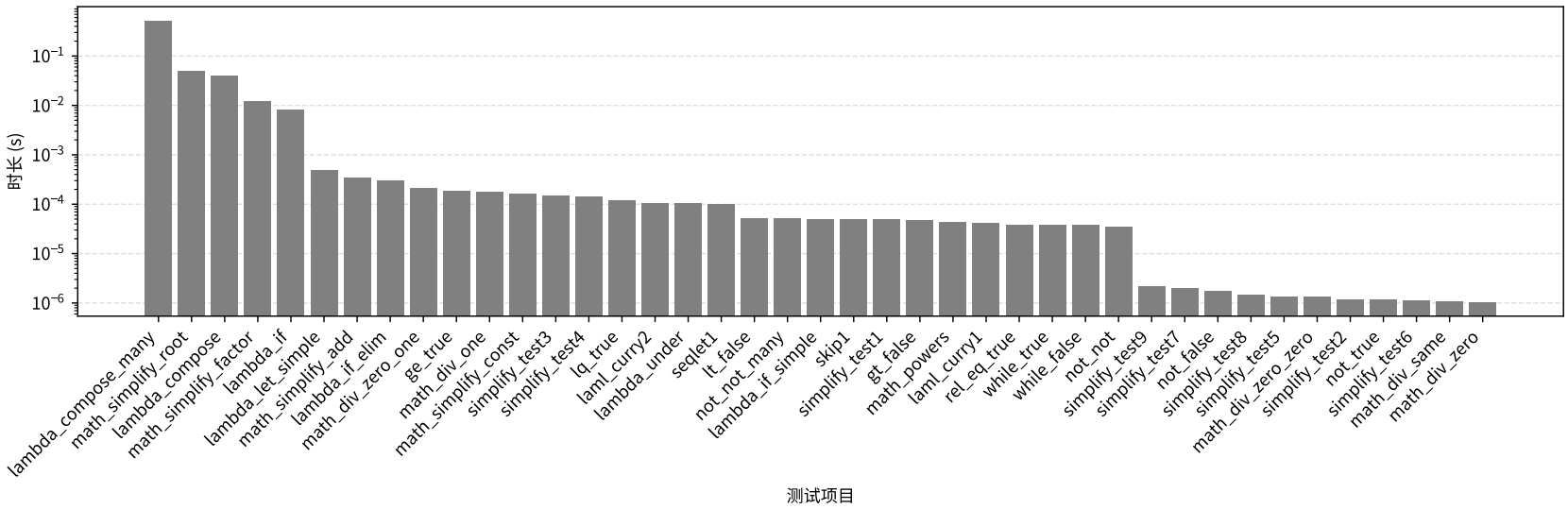


图 10: 所有的测试都在亚秒级时间内完成，绝大部分测试都在亚毫秒级时间内完成。

与基准相比的性能损耗

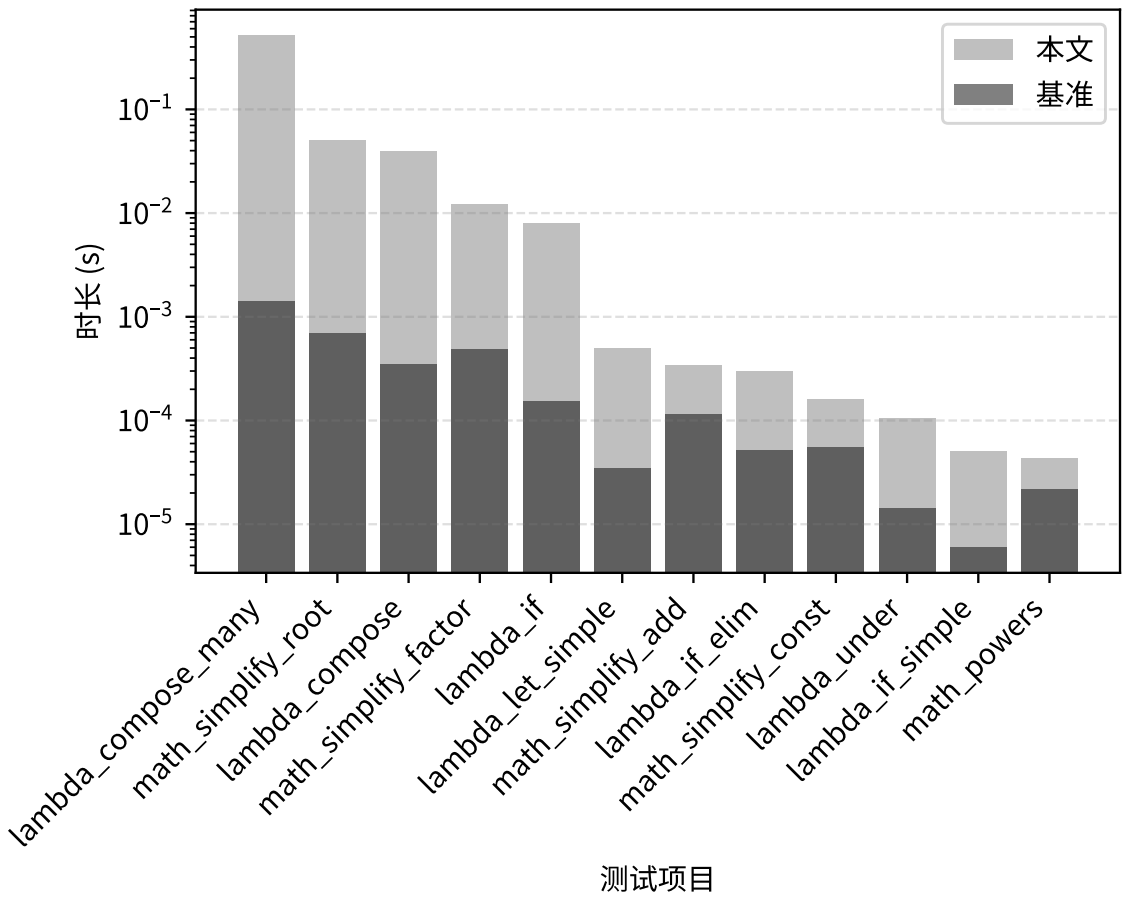
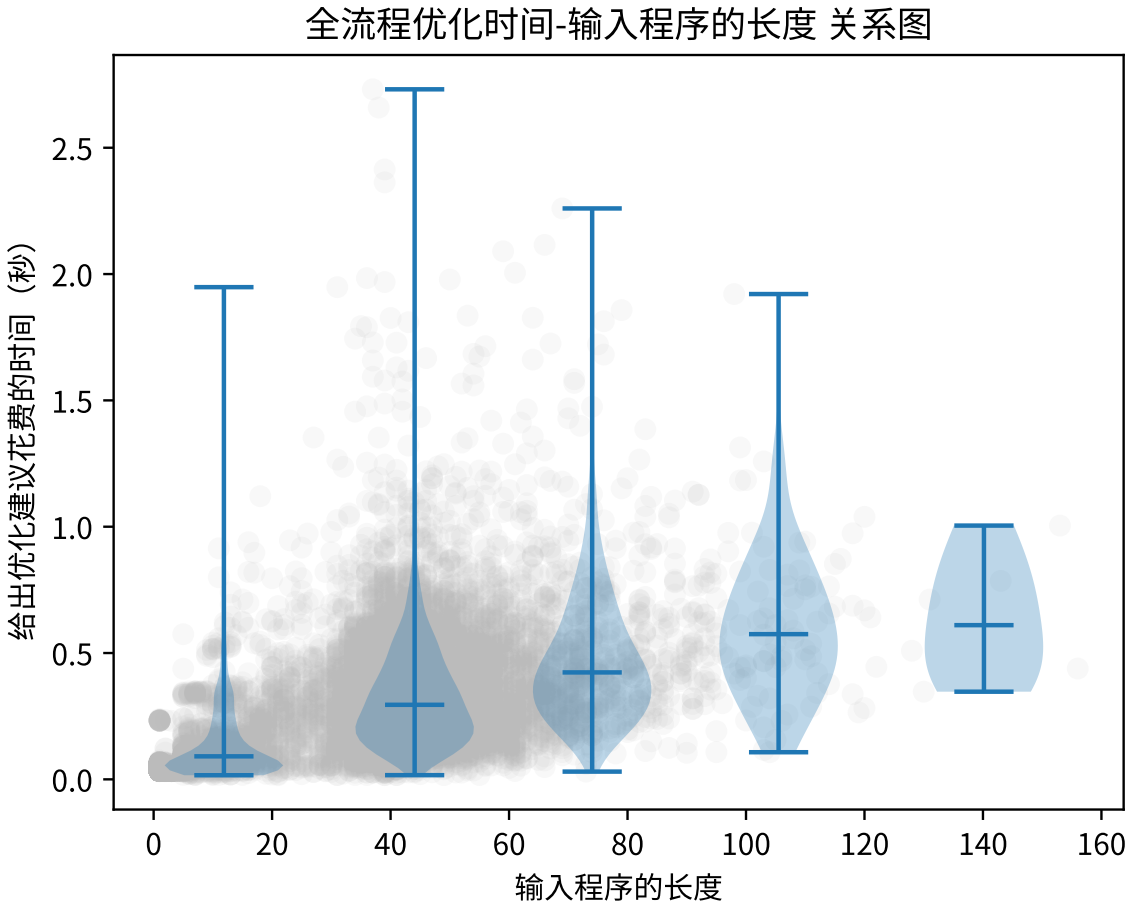


图 11: 对比测试

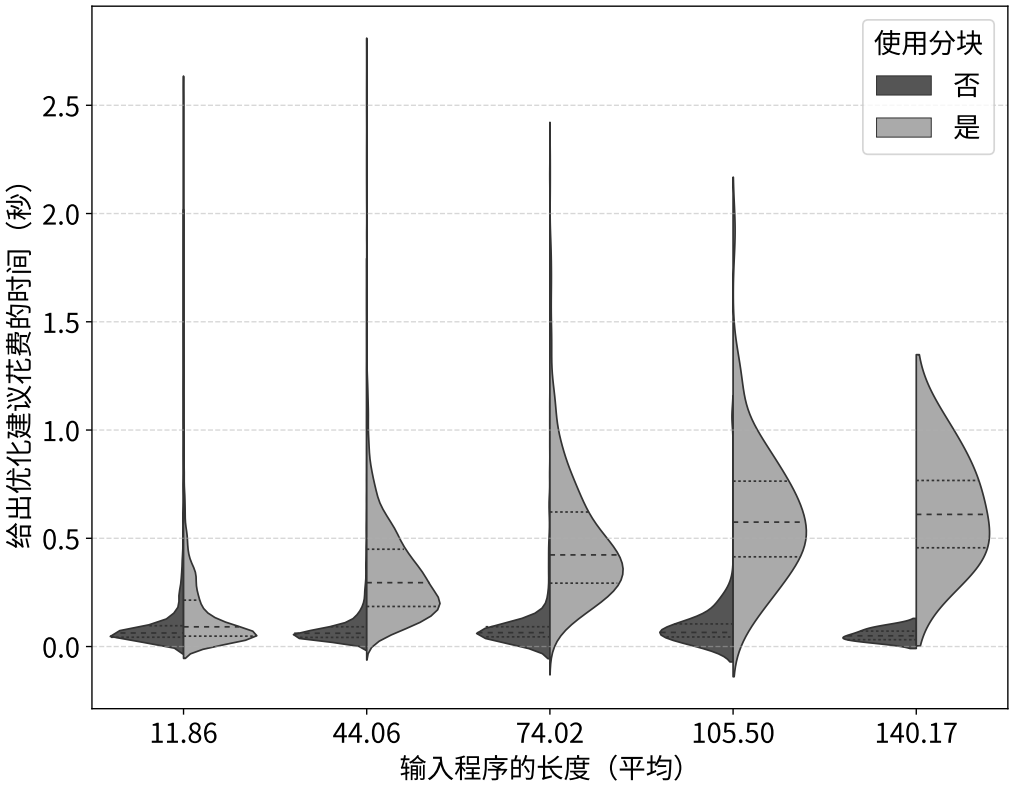
模糊测试与全流程优化性能



- 测试数据：20,000 组语法模糊[a]生成的 Python 程序
- 相关系数：0.438472

[a] Andreas Zeller et al. *The Fuzzing Book*. Retrieved 2023-01-07 14:37:57+01:00. CISPA Helmholtz Center for Information Security, 2023. URL: https://www.fuzzingbook.org/ (visited on 01/07/2023).

模糊测试 - 探究造成微弱相关性消耗的来源



测试数据 各 10,000 组

相关系数 0.014868

图 12: 有无分块优化算法的优化时间对比

实机演示

Demo

总结

本文研究了计算机程序的通用构造与对其优化的方法，实现了一个在代码编写时的代码静态分析与优化工具。

- 设计了一种中间语言 $\mathfrak{C}$，满足常见编程语言的通用性质
- 通过形式化手段提出证明了一系列等价指令作为重写规则
- 使用萃取器依据本文目标定制的损失函数提取最佳程序
- 设计实现了针对不同编程语言特性的解析器与生成器
- 完成了一个针对 VSCode 的语言插件作为一个示例实现

进一步工作

本文实现的编写时源码优化工具还有下列需求需要完善：

- 更完善的语言支持
- 更人性化的输出
- 更好的性能
- 更好的源码优化

谢谢大家