北京大學

本科生毕业论文

题目：一种利用 Redex 实现重组糖的轻量级方法

A Lightweight Resugaring Method using PLT Redex

姓　　名：杨子毅

学　　号：1600011063

院　　系：信息科学技术学院

本科专业：软件工程

指导老师：胡振江

二〇二〇　年　五　月

北京大学本科毕业论文导师评阅表

学生姓名	杨子毅	学生学号	1600011063	论文成绩	
学院 (系)	信息科学技术学院			学生所在专业	软件工程
导师姓名	胡振江	导师单位/ 所在研究所		导师职称	
论文题目 （中、英文）					

导师评语

（包含对论文的性质、难度、分量、综合训练等是否符合培养目标的目的等评价）

导师签名：

年　　月　　日

版权声明

摘要

随着计算机科学的普及和程序设计语言的发展，程序设计语言、特别是领域特定语言的应用越来越日常化。语法糖作为实现领域特定语言的一项重要技术在近年来发展火热，相关的研究以及为支持 DSL 设计特性的的程序设计语言(例如 Racket、Scala) 都有显著进展显著。

重组糖是一项关于语法糖的研究—将嵌入在内部语言的语法糖表达式执行序列在表面语言（语法糖）上提取出来，以得到在语法糖层面的执行序列（详见1）。其目的是为了解决语法糖解糖的单向性，让语法糖表达式的执行过程能表现在语法糖结构上。但现存方法很难处理递归糖、高阶糖等语法糖特性，且对于卫生宏处理很繁琐。

我们基于 PLT Redex 设计了一个轻量级重组糖算法，简单实现了一套工具并在一些例子上应用进行测试。结果显示我们的轻量级重组糖算法相较于现有重组糖算法可以多处理一些语法糖特性，也更容易处理卫生宏等特性。除此之外，我们的方法在表示语法糖的方式上更加灵活。

关键词: 领域特定语言、语法糖、解释器、重写系统

Abstract

With the popularization of computer science and the development of programming languages, the application of programming languages, especially domain-specific languages, is becoming more and more routine. Syntactic sugar, as an important technique for implementing a domain-specific language, has developed fiercely in recent years, and related research and programming languages designing features for DSL(such as Racket, Scala) are making significant progress.

Resugaring is a research on syntactic sugar–lifting the evaluation sequence of syntactic sugar expression embedded in the core language on the surface language (syntactic sugar) to get the sequence at the level of syntactic sugar. (see detail1) The purpose of resugaring is to solve the unidirectionality of desugaring the syntactic sugar expression, so that the evaluation sequence of syntactic sugar expression can be expressed in the structure of syntactic sugar. However, the existing methods are difficult to deal with the features of syntactic sugar such as recursive sugar and high-order sugar, and are very cumbersome for hygienic macro processing.

We designed a lightweight resugaring algorithm based on PLT Redex, simply implemented a set of tools and tested on some examples. The results show that our lightweight resugaring algorithm can handle more syntactic sugar features than existing resugaring algorithms, also handle features like hygienic macro more simply. In addition, our method is more flexible in the way of representing syntactic sugar.

Key Words: Domain-specific language, Syntactic sugar, Interpreter, Rewriting system

全文目录

第一章 绪论

1. 研究背景

领域特定语言[1]的研究及应用日益广泛，其中不乏正则表达式[2]、SQL[3]、XML[4]等应用场合广泛的领域特定语言。而随着计算机科学技术的发展，DSL的应用场景步入了日常生活中，如IFTTT应用、IOS快捷指令。而在一般场合中，语言设计者是计算机科学家，语言使用者却是领域内的专家。因此，当领域特定语言的使用出现一些问题时，领域内的专家（使用者）需要得到领域特定的信息来进行处理。

语法糖是近些年一种流行的领域特定语言实现方法，其方法源于Peter Landin对λ演算应用计算的替换[5]。函数式语言方面，经过Lisp的宏系统（Macro），Scheme语言的发展，再到Racket[6]语言的扩展[7]，如今形成了比较完善的宏系统来用语法糖构造DSL。而像如Scala[8]这种结合了面向对象和函数式语言的程序设计语言也对语法糖的构造很重视。语法糖相关的研究[9]也十分火热。

语法糖构造DSL的最主要优点就是简单高效，语言设计者只需要写一个简单的映射，就可以构造DSL。然而，语法糖的一项缺陷，导致其应用领域大多局限于计算机科学内部。我们先来看下面这个例子。

我们拟构造一个自动化饭店的DSL（如图1）为例。

```
打卤面(卤, 面, 量)→
(for (cnt: getsymbol(量))
        (let (tmpnoodle==getnoodle(面), tmpsauce=getsauce(卤))
            Stir(tmpnoodle, tmpsauce) ))

皮皮虾卤(大中小, 咸淡, 蔬菜丁)→
(begin stirfry(清蒸(luohualx, getsymbol(大中小)), getsymbol(咸淡), 蔬菜丁)
      加香菜)

煮面(种类) →
    makenoodle(getsymbol(种类))
.....
```

图 1 自动化饭店 DSL

其中箭头表示语法糖展开的形式。在这个 DSL 内，执行

打卤面 (皮皮虾卤 (大, 咸, 胡萝卜), 煮面 (毛细), 2)

因为其过程中会有很多中间过程，我们期望得到的执行过程是这样的2。

打卤面 (皮皮虾卤 (大, 咸, 胡萝卜), 煮面 (毛细), 2)
↓
打卤面 (皮皮虾卤 (大, 咸, 胡萝卜丁), 煮面 (毛细), 2)
↓
打卤面 (大份咸皮皮虾卤（胡萝卜丁）, 煮面 (毛细), 2)
↓
打卤面 (大份咸皮皮虾卤（胡萝卜丁）, 毛细抻面, 2)
↓
两份皮皮虾打卤面（大份、咸、胡萝卜丁、毛细）

图 2 期待的执行序列

而实际上，这个 DSL 的执行序列是这样的3。（先将语法糖展开成难懂的内部语言）

打卤面 (皮皮虾卤 (大, 咸, 胡萝卜), 煮面 (毛细), 2)
↓
(begin for. . . let(. . .) . . .)
↓
. . .
↓
两份皮皮虾打卤面（大份、咸、胡萝卜丁、毛细）

图 3 实际执行序列

将这个例子抽象到简单的例子 $and(or(\#f,\ \#t),\ and(\#t,\ \#f))$

（其中 and、or 是语法糖，展开成 if 的表达式）有如下执行序列4

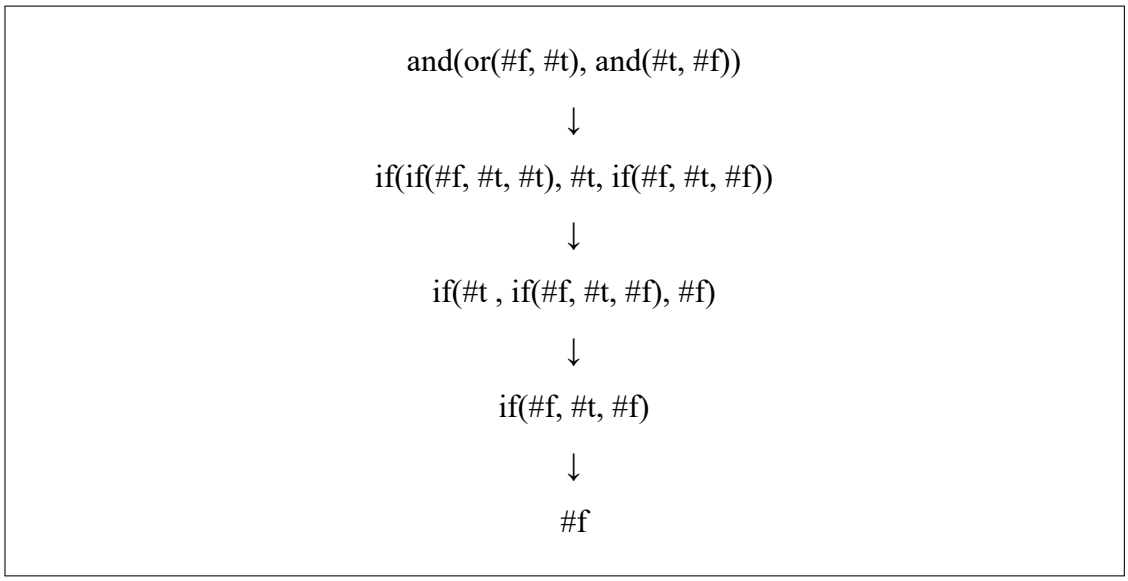

图 4 语法糖表达式执行示例

我们可以看到，在执行过程中，语法糖表达式被展开成通用语言表达式，在通用语言中继续执行，得到最终结果。但实际应用到特定领域时，我们很明显不希望得到这样复杂的执行过程，特别是对于对计算机内部语言不熟悉的领域专家来说，这种执行过程是没有意义的。

2. 语法糖的困境

我们将上述问题总结为语法糖解糖的单向性，对上面的例子4的第三行

$$\text{if}(\#t, \text{if}(\#f, \#t, \#f), \#f)$$

进行观察，我们可以看出，其存在等价的领域特定语言表示 $\text{and}(\#t, \text{and}(\#t, \#f))$，这正是初始表达式的第一个子表达式规约后的结果。如果语法糖的解糖有一个逆过程（重组糖），我们就可以找到语法糖表达式其对应的在语法糖层面的执行序列。对上面的 and、or 糖例子，我们希望得到的重组糖序列如下图5。

我们将在第二章对重组糖问题进行形式化定义。在后文中，将领域特定语言视为外部语言，通用语言视为内部语言。

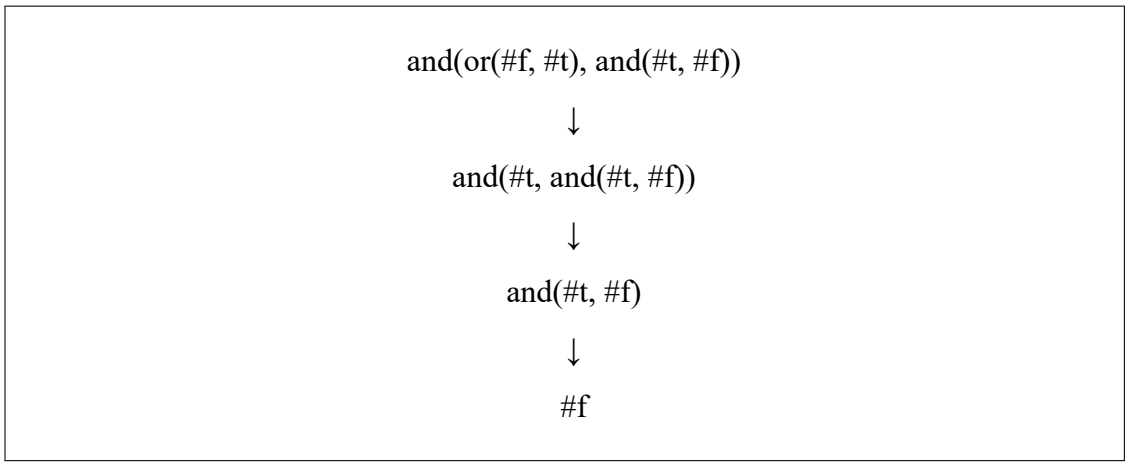

图 5 语法糖表达式重组糖序列实例

3. 现有工作及存在的问题

我们的方法主要借鉴和对比 Resugaring 系列[10][11][12][13] 的一些工作，其中前两篇和本文的工作紧密相关，我们将在这里简单讲述一下这两篇工作的方法及缺陷，并在相关工作部分3.1进一步讨论。

第一篇工作[10] 提出了重组糖问题的概念，并介绍了一个解决重组糖的方法，其基本思想是将内部语言的求值序列每一步加上标签，进行搜索（匹配和替代），试图得到其对应的在外部语言的表示。这也是重组糖名字的由来—将解糖的语法糖表达式重新组成到语法糖。

其算法希望具有如下三个性质（详见1）：

仿真性/抽象性/覆盖性（没有被证明）

第二篇工作[11] 在第一篇的基础上，新增了三个优点：

- 解决卫生宏的重组糖
- 拓展语法糖规则
- 覆盖性得到形式化证明

然而该工作仍然存在一些问题：

- 语法糖规则依然不够丰富
- 算法定义繁琐，通用性差

4. 基本思路

本文基于语义工程工具 PLT Redex[14]，设计了一个全新的语法糖——重组糖框架。其想法源于完全 β 规约的完备性，如下图6。

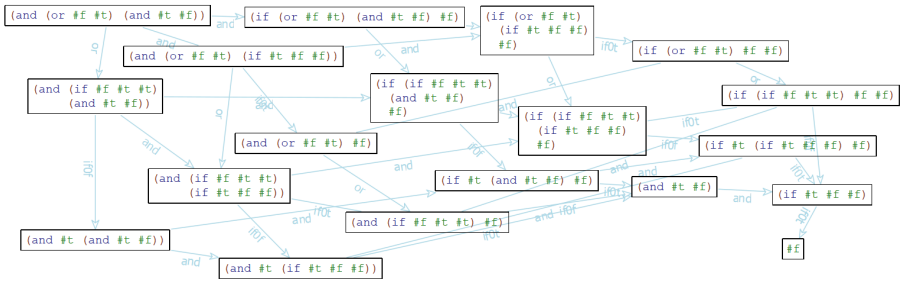

图 6 基本思路

该例子中，对一个结构化（类似 S 表达式，定义在第二章）的语言，定义其基础语义和规约规则。其中 and 和 or 是语法糖，语法糖展开内部语言的 if 表达式。在没有限制其上下文规则（求值顺序）的前提下，生成了其所以规约路径的流程图。我们可以看出，在该图中，既包含了将语法糖展开的规约，也包含了我们期待的 $\mathrm{and}(\#t,\ \mathrm{and}(\#t,\ \#f))$ 等等这些中间求值路径。因此我们希望使用基于规约语义[15] 的 PLT Redex，实现一个轻量级的重组糖，在这个完全图中提取出我们需要的重组糖序列。

在实现过程中，我们发现生成中国完全图并不是必须的，而我们可以在从最初的表达式开始**每次进行单步规约，在一条或多条规约规则中选择我们需要的那条规约规则** (是本工作的核心算法)，且该规则的多次执行保证重组糖的三个基本重要性质。则在这个核心算法迭代执行过程中，会留下一个对应的求值序列，在其中提取出符合输出规则的中间序列，则此序列即为重组糖的输出。

5. 本文主要贡献

1. 我们针对现有重组糖的方法法进行改进，得到新的轻量级重组糖方法。

   - 我们的方法不需要将所有语法糖都展开就可以得到重组糖序列，而现有方法需要展开后在内部语言执行，并基于 match 和 substitute 对可重组的语法糖进行搜索。

2. 我们基于轻量级重组糖算法，用 PLT Redex 实现了一套工具。

3. 基于我们实现的工具，测试得到我们的方法相对于现有重组糖方法，支持更多语法糖特性。

- 对递归糖，我们的方法可以很简单的处理。而现有方法只能用 letrec 处理一下递归绑定。
- 对高阶糖，我们的方法也可以很容易处理。而现有方法不能处理。
- 对卫生宏，我们的方法处理卫生宏很简单而自然，而现有工作处理卫生宏需要引入新的数据结构以及很多其他处理。

6. 全文结构

我们将在第二章讲述本文工作的一些背景知识及思考路线；第三章讲述工作的算法定义及正确性证明；第四章对利用 PLT Redex 的轻量级重组糖工具进行实现上的简单介绍；第五章讲述一些轻量级重组糖的应用，对一些语法糖例子进行讨论评估；第六章总结我们的工作，并对一些未来可能的方向进行简单探讨与展望。

第二章 背景知识

1. 重组糖形式化定义

对于给定求值规则的内部语言 CoreLang，和在 CoreLang 基础上用语法糖构造的表面语言 SurfLang；对于任意 SurfLang 的表达式，得到其在 SurfLang 上的求值序列，且该求值序列满足三个性质：

1. 仿真性: 求值序列需要和在 CoreLang 上的求值顺序相同，即存在 CoreLang 上的求值序列中的部分中间过程与该序列中的元素对应。该性质是重组糖有意义的前提。

2. 抽象性: 求值序列中只存在 SurfLang 中存在的术语，没有引入 CoreLang 中的术语。该性质是重组糖研究的目的。

3. 覆盖性: 在求值序列中没有跳过一些中间过程。该性质不是正确性的必要条件，却是在应用中极其重要的；加上前两条性质满足的正确性，构成了重组糖的全部重要性质。

例子：$\text{and}(\text{or}(\#f, \#t), \text{and}(\#t, \#f))$

语法糖规则 (Surflang)：

$$\text{and(e1, e2)} \rightarrow \text{if(e1, e2, \#f)}$$
$$\text{or(e1, e2)} \rightarrow \text{if(e1, \#t, e2)}$$

其中 if 的规则是在 coreLang 上规定的，其具体规约规则如下：

$$\text{if(\#t, e1, e2)} \rightarrow \text{e1}$$
$$\text{if(\#f, e1, e2)} \rightarrow \text{e2}$$

则我们期望得到的重组糖序列是如下1的序列:

全文将围绕这个例子展开初步的讲解。

and(or(#f, #t), and(#t, #f))

↓

and(#t, and(#t, #f))

↓

and(#t, #f)

↓

#f

图 1 重组糖示例

2. 完全 β 规约、归约语义和 PLT Redex

本节内容共同的理论基础是 Church-Rosser 定理[16]。

与 β 规约的概念不同，完全 β 规约是一种基于 β 规约的 λ 演算的求值策略。对于一个嵌套的 lambda 表达式，每个表达式都可能进行 β 规约，常规的 call-by-name 和 call-by-value 都是对规约顺序进行了约定，而完全 β 规约就是一种不定序的求值规则，每个可 β 规约的位置都有可能进行规约，因此得到的规约路径不是一条，而是一个图，且这个图的起点和终点只有一个。如图2所示的例子就是一个完全 β 规约的求值图。

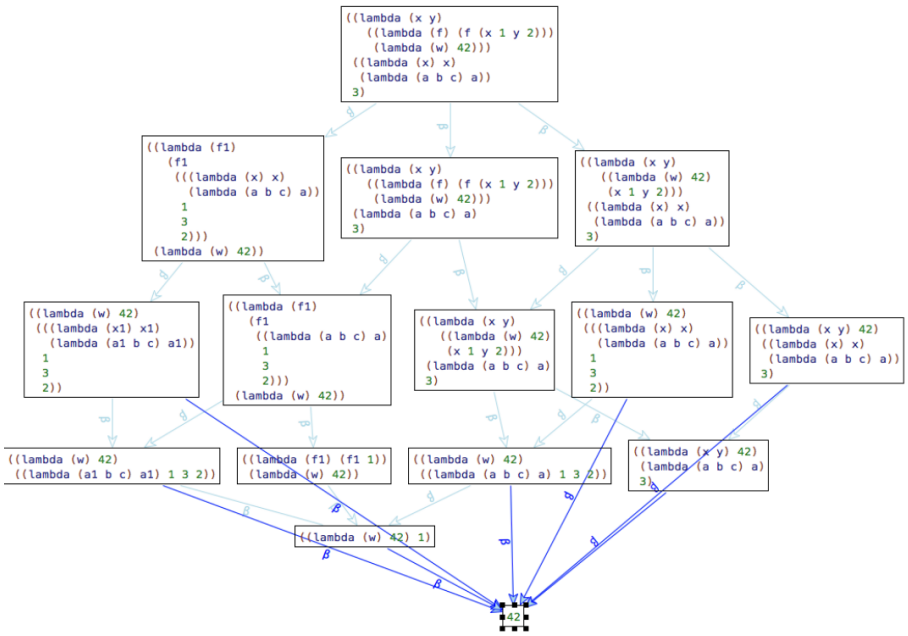

图 2 完全 β 规约

可以看出，在完全 β 规则中，对任何位置的可 β 规约的 lambda 表达式，都可以进行规约。因此，与 call-by-name 和 call-by-value 不同的是，这种求值规则是不定序的。

结构化操作语义[17] 之父 Gordon Plotkin 在 1975 年的文章[18] 中说明了抽象自动机和递归解释器的表达能力是相同的，而 λ 演算在一定约束下也具有相同的表达能力，Plotkin 的 λ 演算基于推理规则，如图3。

$$\frac{e \to e'}{(v\ e) \to (v\ e')}$$

$$\frac{e \to e'}{(e\ e'') \to (e'\ e'')}$$

图 3 Plotkin's lambda

而 1992 年 Felleisen 等人提出的规约语义[15] 则是将推理规则优化为了规约规则加上上下文规则，如图4的规约语义规则就是和上面两条推理规则等价的。

E = (E e) | (v E) | []

if e → e' then E[e] → E[e']

图 4 Reduction semantic

相较于传统的操作语义，规约语义的优点是更容易处理顺序控制和状态信息，本质上属于操作语义，是传统操作语义的一种修订和优化。

PLT Redex[14] 则是基于规约语义的语言建模工具（也称为语义工程工具），是基于 Racket 实现的。给定一个语言的文法、上下文规则和规约规则，PLT Redex 可以对这个语言的表达式进行规约，生成规约图，随机测试语言正确性等功能。本文运用 PLT Redex 主要是运用其优秀的语言建模功能。

本文工作的最初思想就是基于完全 beta 规约。我们在对规约语义不限制上下文环境的情况下，其规约路径也将成为类似完全 β 规约的图。还是基于上文的例子 and(or(#f, #t), and(#t, #f))，我们可以得到如下的完全规约图5。

在这个图中，我们可以看出，用红色标出的子序列是将语法糖直接展开后进行规约的求值序列，而这其中有许多中间表达式是可以重组成语法糖的，用蓝色标出。我们可以发现，在这个中间表达式中可重组的部分就是图1中的重组糖序列。而又可以在图中找到绿色标出的序列—可以惊喜而又自然的发现这一条规约规约路径包含了我们想要的重组糖序列。自然是因为我们对语法糖的规约

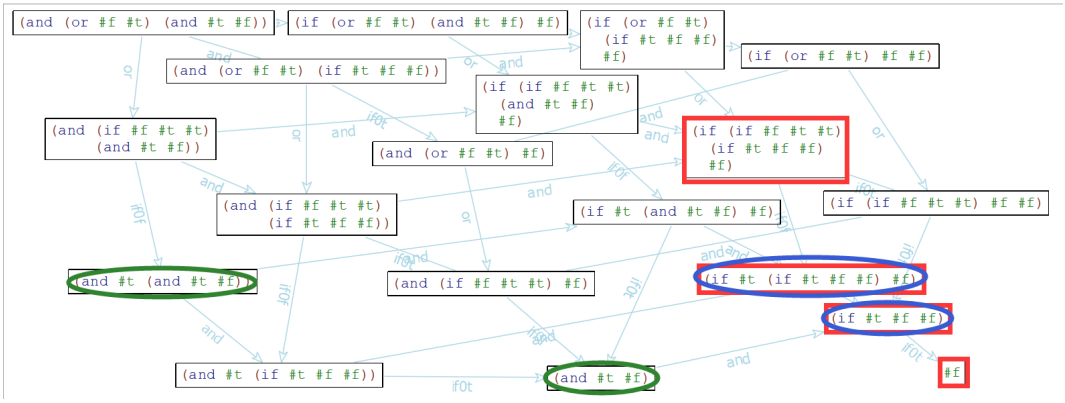

图 5 完全规约

做了类似完全 β 规约的处理，导致每个子表达式都有可能首先被规约处理，因此重组糖序列一定在我们的完全图中。

第三章　轻量级重组糖算法

1. 对语言的规定

1.1 文法限制

首先，我们需要将整个语言限定在基于树形表达式的结构化语言。

定义 (结构化). *对于每个表达式中的子表达式，其规约规则只和子表达式本身有关。*

此限制约束了 Corelang 的作用域，限制语言子表达式不能有对外的副作用。我们将在第五章详细讨论副作用的一些具体解决办法。

定义 (树形表达式). *此处我们使用类似 Lisp 的 S 表达式的递归树，基础定义如下*

$$\begin{aligned} \text{Exp} \quad ::= \quad & (\text{Headid Exp}*) \\ | \quad & \text{Value} \\ | \quad & \text{Variable} \end{aligned}$$

1.2 上下文规则限制

对每个子表达式都是 CoreLang 表达式的表达式 Exp，最多只能有一条规约路径（通过求值顺序约束）。这一点约束并不过分，为了保证每个程序只能有一条执行路径。

对 SurfLang，任意一个语法糖只能有一个 CoreLang 的表达式与之对应。这也是很自然的要求，因为同一个语法糖不应该有二义性。对于求值顺序，SurfLang 上的子表达式约定类似完全规约的规则，任何子表达式都可以首先进行规约。

1.3 语法糖形式限制

对于语法糖的形式限制为如下。

$$\boxed{(\text{Surfid } e_1\ e_2\ \ldots) \rightarrow (\text{Headid } \ldots)}$$

其中的主要约束是在语法糖这一侧不能出现形如 (Surfid ... $(e1\ e2)$...)) 这种形式。这对语法糖的表达能力并不会造成影响。[1]

1.4　文法描述

在 PLT Redex 中，我们将 CoreLang 和 SurfLang 视为同一个语言。则当我们定义了一个语言内部各种规约规则后，对于任意 Exp 都有其对应的一条或多条规约规则。根据对 CoreLang 的约定，有多条规约规则的表达式必然存在 SurfLang 的表达式。

为了区分 CoreLang 的语言和 SurfLang 的语言，我们将表达式的文法定义为如下

$$
\begin{array}{rcl}
\text{Exp} & ::= & \text{Coreexp} \\
& | & \text{Surfexp} \\
& | & \text{Commonexp} \\
& | & \text{OtherSurfexp} \\
& | & \text{OtherCommonexp} \\
\\
\text{Coreexp} & ::= & (\text{CoreHead Exp}*) \\
\\
\text{Surfexp} & ::= & (\text{SurfHead } (\text{Surfexp} \mid \text{Commonexp})*) \\
\\
\text{Commonexp} & ::= & (\text{CommonHead } (\text{Surfexp} \mid \text{Commonexp})*) \\
& | & \text{Value} \\
& | & \text{Variable} \\
\\
\text{OtherSurfexp} & ::= & (\text{SurfHead Exp} * \text{ Coreexp Exp}*) \\
\\
\text{OtherCommonexp} & ::= & (\text{CommonHead Exp} * \text{ Coreexp Exp}*)
\end{array}
$$

在这里，我们将 CoreLang 的表达式一部分提取处理作为公共表达式，是因为在重组糖序列中必定有一些表达式是属于 CoreLang 的，但需要在序列中输出

[1]此约束主要是为了算法描述和实现更简单。

（比如说数字，布尔表达式，以及一些可能的基础运算）。在这种情况下，对于 Commonexp 来说，满足 CoreLang 的约束，但是也可以作为重组糖的中间序列输出

可以看出，在我们的重组糖方法中，可以输出的表达式是 Surfexp 和 Commonexp，即不存在任何子表达式中存在 Coreexp。

2. 算法描述

本节讨论建立在上一节中符合约定的语言基础上。

2.1 核心算法

核心算法 f[2]定义如下1。

接下来对整个算法语言说明，分别解释每一个分支规则。对 Exp 尝试所有规约规则，得到多个可能的表达式 ListofExp'=$\{Exp'_1, Exp'_2, \ldots\}$

如果 Exp 是 Coreexp 或 Commonexp 或 OtherCommonexp，则其规约规则

- 或是已经无法被规约（ListofExp' 为空），此时返回的表达式为空。$Rule1.1$
- 或是将表达式规约到另一个表达式，此时只有一条规则，应用后输出 Exp'; $Rule1.2$
- 或是其规约不满足导致内部子表达式需要规约，此时因为 CoreLang 的定序性，只会有一个子表达式被规约（且此表达式为 Surfexp），此时对该子表达式 Subexp 递归调用核心算法 f 得到 Subexp'，则在 ListofExp' 中找到将此 Exp 中子表达式 Subexp 规约为 Subexp' 的表达式就是我们需要的表达式; $Rule1.3$

如果 Exp 是 Surfexp 或 OtherSurfexp

- 如果内部子表达式无可规约的，则必然会展开该语法糖，此时输出表达式为 Exp 解糖后的表达式; $Rule2.1$

[2]核心思想：对于每个表达式 Exp，我们将对它所有规约规则中选择一条符合 resugaring 的仿真性规则的规约，且尽可能不破坏任何语法糖。

Algorithm 1 核心算法 f

输入:

符合规定语言的表达式 Exp=($Headid\ Subexp_1\ \ldots\ Subexp_{\ldots}$)

输出:

Exp 的所有规约规则中选择合适的规则规约后的 Exp'，使得该过程满足重组糖性质

1: Let $ListofExp' = \{Exp'_1, Exp'_2\ \ldots\}$
2: **if** Exp is Coreexp or Commonexp or OtherCommonexp **then**
3: **if** Lengthof($ListofExp'$)==0 **then**
4: **return** null; // Rule1.1
5: **else if** Lengthof($ListofExp'$)==1 **then**
6: **return** first($ListofExp'$); // Rule1.2
7: **else**
8: **return** $Exp'_i = (Headid\ Subexp_1\ \ldots\ Subexp'_i\ \ldots)$; //where i is the index of subexp which have to be reduced. Rule1.3
9: **end if**
10: **else**
11: **if** Exp have to be desugared **then**
12: **return** desugarsurf(Exp); // Rule2.1
13: **else**
14: Let $DesugarExp'$ = desugarsurf(Exp)
15: **if** $Subexp_i$ is reduced to $Subexp'_i$ during $f(DesugarExp')$ **then**
16: **return** $Exp'_i = (Headid\ Subexp_1\ \ldots\ Subexp'_i\ \ldots)$; // Rule2.2.1
17: **else**
18: **return** desugarsurf(Exp); // Rule2.2.2
19: **end if**
20: **end if**
21: **end if**

- 如果存在可规约的子表达式对于每个子表达式，如果可规约，则根据我们的设定，存在一条关于此子表达式的规约规则。因此每个子表达式都可能被规约的前提下，我们需要对 Surfexp 或 OtherSurfexp 的外层语法糖进行展开为 DesugarExp’（此展开只有一种规约规则对应），之后对 Exp’ 调用核心算法 f（**单步尝试**）

  – 如果 f(DesugarExp’) 是对 DesugarExp’ 的子表达式进行规约，由于此子表达式一定由 Exp 的子表达式组成，需要检测是哪一个子表达式 $Subexp_i$ 先被规约为 $Subexp_i'$，则在 ListofExp’ 中将此子表达式规约的表达式就是所需要的。 $Rule2.2.1$

  – 如果 f(DesugarExp’) 不是对子表达式进行规约，则说明这个糖不会被重组（此糖被展开后必然会被继续破坏），因此输出不在 ListofExp’ 中，而是输出 DesugarExp’ $Rule2.2.2$

2.2 轻量级重组糖算法

整体算法 lightweight-resugaring 定义如下2。

Algorithm 2 轻量级重组糖算法 Lightweight-resugaring

输入:

给定 Surfexp 的表达式 Exp

输出:

Exp 的重组糖序列

```
1:  while tmpExp = f(Exp) do
2:    if tmpExp is empty then
3:      return
4:    else if tmpExp is Surfexp or Commonexp then
5:      print tmpExp;
6:      Lightweight-resugaring(tmpExp);
7:    else
8:      Lightweight-resugaring(tmpExp);
9:    end if
10: end while
```

3. 正确性证明

首先，由于我们的算法和先解糖后重组糖的区别就是我们只在需要的时候将语法糖解开，而传统意义上语法糖是先将语法糖全部展开然后再 CoreLang 上进行执行。

其次，为了证明方便，定义一些术语。

$(Headid\ Subexp_1\ Subexp_{...}\ldots)$ 为任意可规约表达式

对表达式运用 Headid 的对应规则进行规约，称为外部规约。

对其子表达式 $Subexp_i$ 规约。其中 $Subexp_i$ 为 $(Headid_i\ Subexp_{i1}\ Subexp_{i...}\ldots)$

- 如果对 $Subexp_i = (Headid_i\ Subexp_{i1}\ Subexp_{i...}\ldots)$ 进行外部规约，则称为表面规约。

- 如果对 $Subexp_{ij}$ 进行规约，则称为内部规约。

例：

(if $\#t\ Exp_1\ Exp_2$) $\rightarrow Exp1$　　外部规约

(if (And $\#t\ \#f$) $Exp_1\ Exp_2$) $\rightarrow$ (if (if $\#t\ \#f\ \#f$) $Exp_1\ Exp_2$)　　表面规约

(if (And (And $\#t\ \#t$) $\#t$) $\#f$) $Exp_1\ Exp_2$) $\rightarrow$ (if (And $\#t\ \#t$) $Exp_1\ Exp_2$)

内部规约

定义. 对 Exp=$(Headid\ Subexp_1\ Subexp_{...}\ldots)$，$Exp$ 称为**上层表达式**，$Subexp_i$ 称为**下层表达式**。

3.1 仿真性

仿真性：求值序列需要和在 CoreLang 上的求值顺序相同，即存在 CoreLang 上的求值序列中的部分中间过程与该序列中的元素对应。

定理 3.1 (仿真性)**.** 我们的轻量级重组糖具有仿真性。

仿真性证明. 对核心算法 f 逐条进行分析。

首先 $Rule1.1$ 和 $Rule1.3$ 不会影响仿真性，因为其本身就是在对 CoreLang 的表达式进行操作；

对 $Rule1.2$ 是对子表达式进行规约（用核心算法 f），因为 $Coreexp$ 内部的上下文规则是定的，因此即使将表达式所有糖都进行展开，也是该位置先进行规

约。而在递归调用 f 的过程中，由于表达式深度优先，如果对下层的表达式调用 f 满足仿真性，那么此处上层表达式也满足仿真性。

对 $Rule2.1$ 和 $Rule2.2.2$，解糖操作也不会破坏仿真性。

对 $Rule2.2.1$，由于仿真性的定义，重组糖的序列的每一步可以解糖到 CoreLang 上执行序列的步骤，因此证明仿真性只需要证明我们的序列在 CoreLang 上的序列有对应。我们在对 $(Surfid\ Subexp_1\ Subexp_{...}\ ...)$ 进行正常运算时，也会将其依据 Surfid 的规约规则进行展开；而在我们的核心算法中，我们也是这样进行单步展开，区别在于内部的糖没有规约。由于 Rule2.2 对 Exp 进行外部规约后得到 DesugarExp'，并调用了 f(DesugarExp')

- 如果 DesugarExp' 是 $Coreexp$ 或 $Commonexp$ 或 $OtherCommonexp$，且其子表达式规约，因此对于将 Exp 完全解糖后也是该子表达式的位置进行规约，其他子表达式的内容不变，因此对 Exp 的该子表达式规约后完全解糖和对 Exp 整体先解糖再规约，得到的是相同的表达式。如下图1所示。

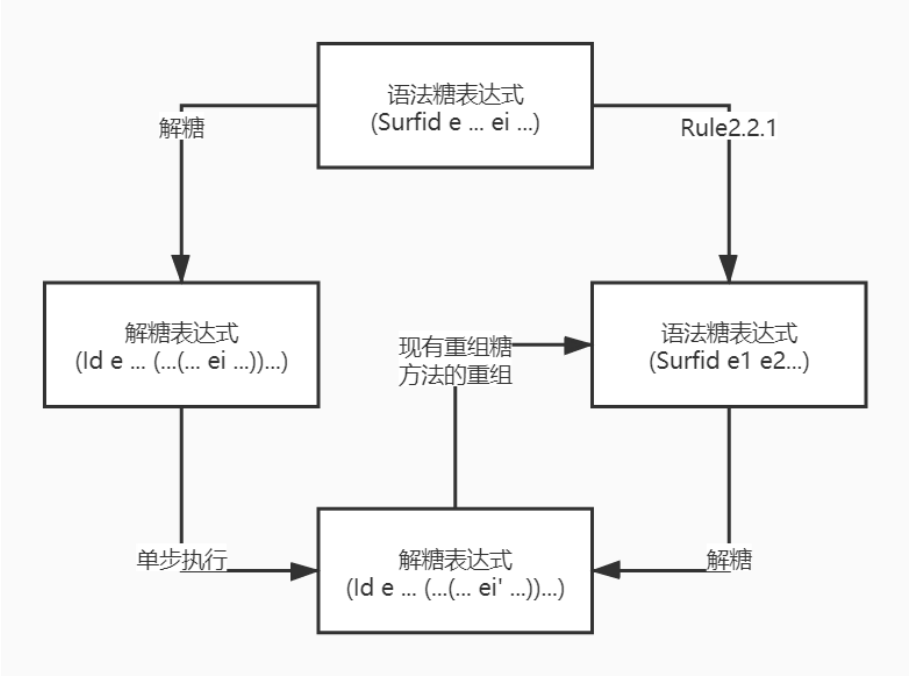

图 1 Rule2.2.1 仿真性图示

- 如果 DesugarExp' 是 $Surfexp$ 或 $OtherSurfexp$，则此处将递归调用 f，由于表达式深度有限，如果下层的表达式满足仿真性，则此处也满足仿真性.

证毕。

□

3.2 抽象性

抽象性：求值序列中只存在 SurfLang 中存在的术语，没有引入 CoreLang 中的术语。

定理 3.2 (仿真性)**.** 我们的轻量级重组糖具有仿真性。

抽象性的正确性是显然的，因为我们在每次输出都判断了输出的 Exp 是否是 $Surfexp$ 或 $Commonexp$。（在 lightweight-resugaring 算法中）

3.3 覆盖性

覆盖性：在求值序列中没有跳过一些中间过程。

引理 3.1. 如果在重组糖序列中语法糖没有在不必须破坏的时候被破坏，那么就没有跳过中间过程。

引理证明. 假设没有语法糖提前破坏的情况下，存在 CoreLang 序列中的某一项 $Exp = (Headid\ Subexp_1\ Subexp_{...}\ldots)$ 可被重组为 $ResugarExp' = (Surfid\ Subexp'_1\ Subexp'_{...}\ldots)$, 且没有在算法 lightweight-resugaring 中被表现。则说明

- 或是重组糖序列中存在

  $ResugarExp$=$(Surfid\ Subexp'_1\ \ldots\ Subexp_i\ Subexp'_{...}\ldots)$

  使得 $ResugarExp$ 解糖后得到的表达式单步规约得到 Exp，且此时该单步规约将 $ResugarExp$ 中子表达式 $Subexp_i$ 对应部分规约。我们发现此时 $ResugarExp$ 的语法糖结构并没有被破坏,因此如果不能展示 $ResugarExp'$ 则是提前破坏了不必要破坏的语法糖，与假设前提矛盾。

- 或是重组糖序列中存在

  $ResugarExp$=$(Surfid'\ \ldots\ ResugarExp'\ \ldots)$

  使得 $ResugarExp$ 解糖后得到的表达式单步规约得到 Exp，且该 Exp 是从 $ResugarExp$ 中的子表达式 $ResugarExp'$ 解糖得到，说明此步单步规约不涉及 $ResugarExp'$ 的规约。而如果不能展示 $ResugarExp'$，则说明该语法糖在执行前面的序列被提前破坏了，也与假设矛盾。

证毕。 □

定理 3.3 (覆盖性)**.** *我们的轻量级重组糖具有覆盖性。*

证明. 由引理可知，只需证明核心算法 f 得每一条都不会将语法糖在不必要破坏时被破坏。对 f 的每一条进行讨论。

对 Rule1.1 和 Rule1.3 显然没有破坏不必须破坏的语法糖。

对 Rule1.2，由于 CoreLang 的表达式需要进一步规约必须对特定子表达式进行规约，因此要对相应的下层表达式作用 f。同仿真性，由于表达式深度有限，下层表达式作用 f 满足覆盖性可递归证明上层表达式满足覆盖性。

对 Rule2.1，此时必须破坏语法糖，否则无法继续规约，因此页没有破坏不必须破坏的语法糖。

对 Rule2.2.1，与 Rule1.2 相同，递归的对下层表达式作用 f，递归证明覆盖性。

对 Rule2.2.2，因为单步尝试后发现语法糖解糖后，解糖后结构继续被破坏，因此无法还原回语法糖，也是必须破坏的语法糖。

因此求值序列没有破坏任何不必须破坏的语法糖，核心算法 f 满足覆盖性。

证毕。 □

第四章 实现

1. 程序框架

整体程序框架如图1。

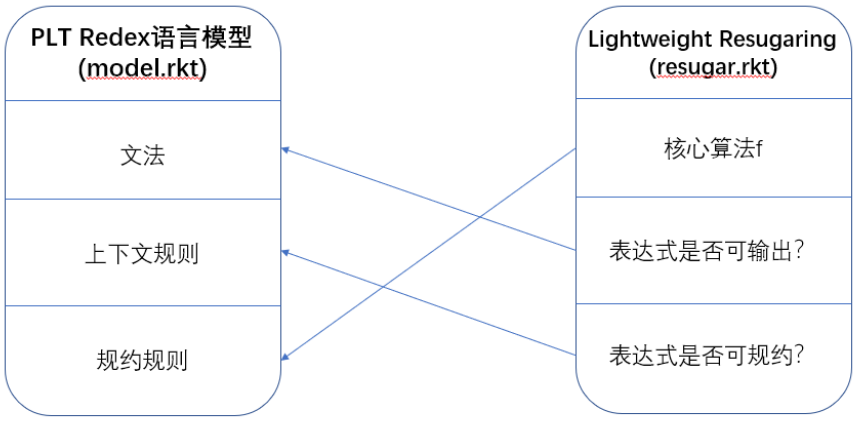

图 1 程序整体架构

我们将语言模型（内部语言和外部语言）的文法、上下文规则、规约规则放在 model.rkt 中，而算法部分放在 resugar.rkt 中。对于任意新的语法糖，需要进行的修改就是在 model.rkt 中

- 增加对应文法到 surfexp
- 增加对于上下文规则（让每个子表达式都能进行规约）
- 增加语法糖解糖规则到 reduction

之后在 resugar.rkt 中构造表达式进行测试即可输出对应重组糖序列。

2. 实现细节

在 PLT Redex 中，对一个语言的定义主要需要三个内容。

- 文法

- 上下文环境
- 规约规则

我们在实现中，对所有表达式按如下文法进行子类划分。

```
exp ::= coreexp | surfexp | commonexp
```

与 (第三章第 1 节) 的定义有所差别的是，这里的 surfexp 包括了第三章第 1 节的 Surfexp 和 OtherSurfexp，commonexp 包括了第三章第一节的 Commonexp 和 OtherCommonexp。

对于 coreexp，自然是包含了所有内部语言的术语。因为我们的定义是将 Exp 的形式限制在 $(Headid\ Subexp_1 \ldots)$，通常情况下可以根据 Headid 判断是否是 coreexp。

例：

```
coreexp ::= (if exp exp exp)
        |   (let ((x exp) ...) exp)
        |   (first exp)
        |   (rest exp)
        |   ...
```

对于 surfexp，就是 $Headid$ 是表示 DSL 的标识的表达式。

例：

```
surfexp ::= (and exp exp)
        |   (or exp exp)
        |   (map exp exp)
        |   (filter exp exp)
        |   ...
```

对于 commonexp，是为了让我们的重组糖序列中有一些包含在内部语言、但是可以输出的中间过程。例如对于高阶糖

$(Map\ f\ lst) \rightarrow (cons\ (f\ (first\ lst))\ (Map\ f\ (rest\ lst)))$

我们需要一些中间序列用 cons 表示，来输出有用的中间过程。我们主要需要的 commonexp 有所有值、所有基础运算（包括算数运算和列表运算）。

对于上下文规则，对 coreLang 为了限制每个表达式只有一条规则可以规约，需要仔细限制位置。而对于 surfLang，由于我们不知道哪个子表达式需要先规约，要把每一个位置都设为可规约的洞。

```
(E  (v ... E e ...)
(let ((x v) ... (x E) (x e) ...) e)
(if E e e)
(and E e)
(and e E)
(or E e)
(or e E)
...
hole)
```

规约规则没有特殊说明。具体可参见代码。[1]

具体算法实现中，比较复杂的是单步尝试的过程（对应核心算法 f 的 Rule2.2)。因为尽管“检测哪一个子表达式被规约”看起来很容易，实际需要考虑一些极端情况的特例。比如多个子表达式相同，我们不用匹配替代的方法（match/substitute），而是用子表达式替换后检测的方法，将其中一个子表达式替换为一个不可规约的表达式后，检测被规约的子表达式是否发生变化。还有从一个表达式到另一个表达式是否破坏了原表达式的结构，比如从 $(and\ \#t\ (and\ \#t\ \#t))$ 到 $(and\ \#t\ \#t)$ 究竟是破坏了外部语法糖还是对内部子表达式进行规约，都是需要在实现时需要注意的地方。

[1] https://github.com/yangdinglou/Lightweight-Resugaring-using-PLT-Redex

第五章 应用及讨论

1. 应用

我们就不同类型（特性）的语法糖分别进行尝试。因为本节中有一些高阶的例子需要将 λ 表达式作为 term 输出，因此本文的测试中将 λ 表达式视作 commonexp，因此可以输出。如果不需要的话（比如 Let 语法糖），将 λ 表达式对应内容放到 coreexp 中即可。

1.1 普通糖

我们首先就简单的

$(and\ e1\ e2) \rightarrow (if\ e1\ e2\ \#f)$

$(or\ e1\ e2) \rightarrow (if\ e1\ \#t\ e2)$ 糖进行测试。

因为我们的 CoreLang 上规定的求值顺序是，对 $(if\ e1\ e2\ e3)$ 的 $e1$ 求到值后就规约，因此我们应该看到的是对 and 和 or 糖，先将第一个值求到值，如何对糖进行破坏。

测试结果如图1、2。

(and (and #t #f) (or #f #t))

↓

(and #f (or #f #t))

↓

#f

图 1 and、or 糖测试 1

我们可以看到如上的两个例子皆按照我们预想的样子输出了重组糖序列。

另外尽管我们的重组糖序列不考虑有副作用的语言，但我们依然测试了一下考虑副作用时的糖的正确性。

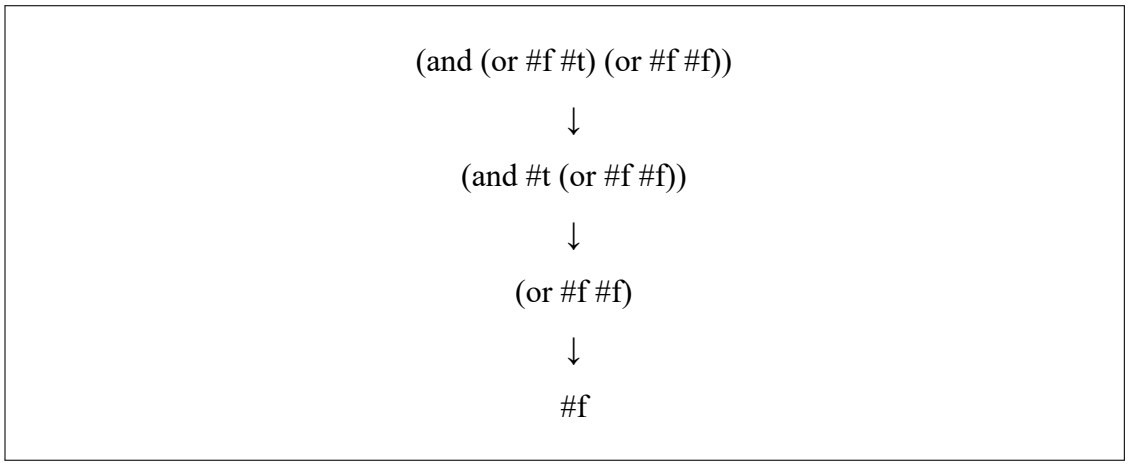

图 2 and、or 糖测试 2

对语法糖 $(Myor\ e1\ e2) \rightarrow (let\ ((tmp\ e1))\ (if\ tmp\ tmp\ e2))$ 其中没有用到任何其他糖，因此我们可以期待它的效果和上面的 or 糖一样。

测试结果如图3。

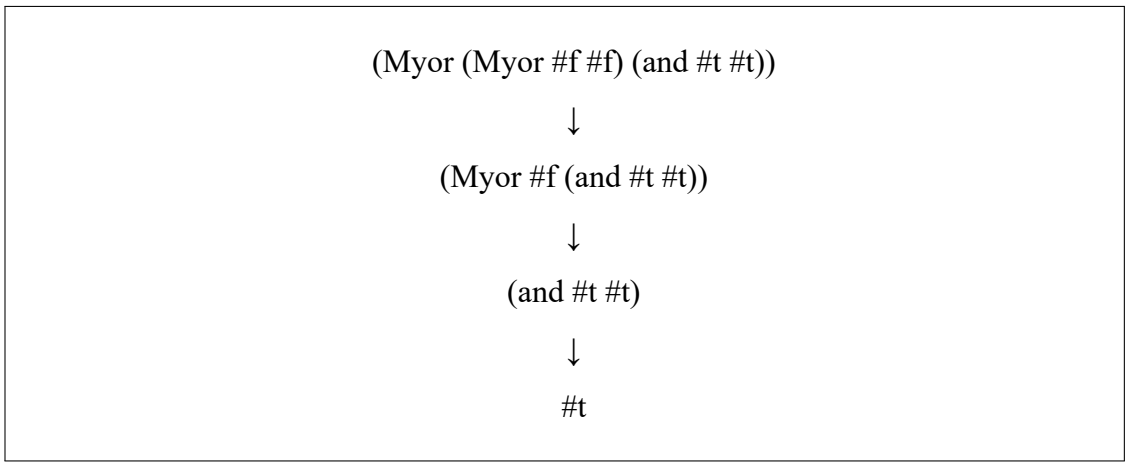

图 3 Myor 糖测试

结果显示我们的方法确实正确得到了该重组糖的序列。

1.2 复合糖

尽管和普通糖没有本质区别，但本节测试的糖在解糖后依然存在语法糖结构。我们构造如下 Sg 糖为例

$(Sg\ e1\ e2\ e3) \rightarrow (and\ (or\ e1\ e2)\ (not\ e3))$ 测试结果如图4。

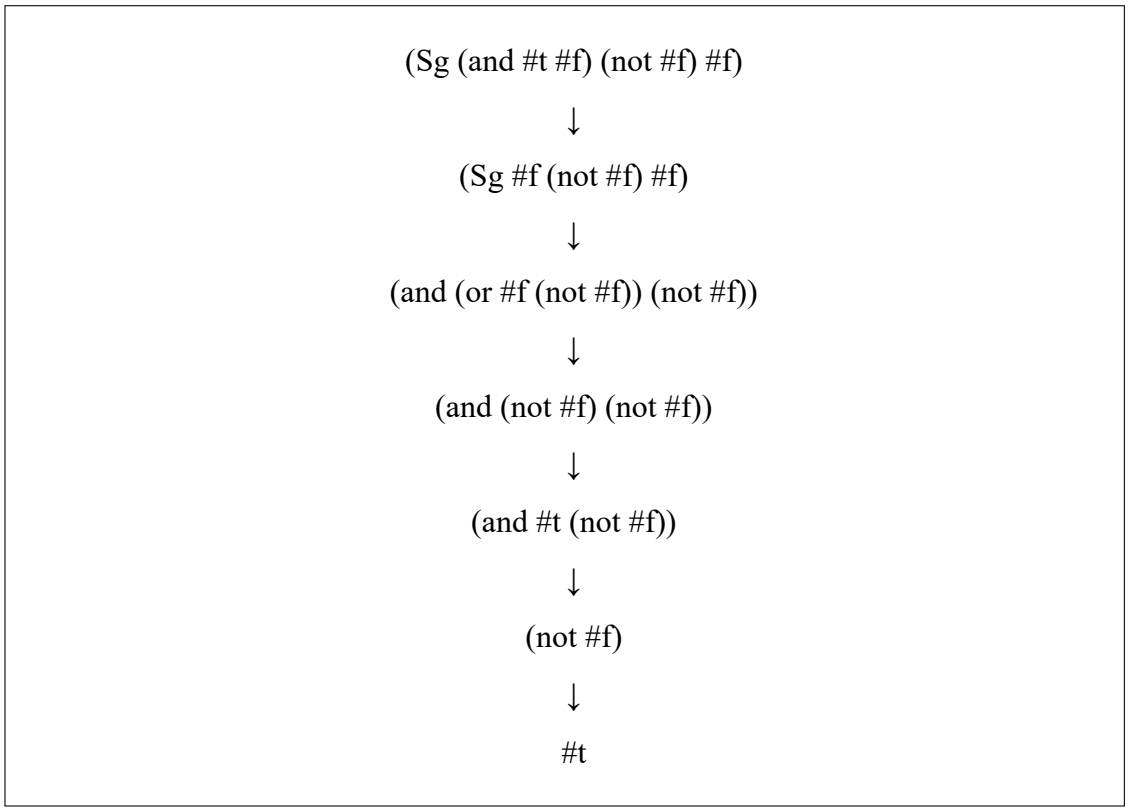

图 4 Sg 糖测试

通过简单的展开验证我们发现，对于该语法糖表达式我们的方法确实做到了在语法糖需要展开的时候展开。

1.3 卫生宏

卫生宏定义：展开宏时变量作用域不被改变的宏。

例：

对语法糖 $(Let\ x\ v\ exp) \rightarrow (Apply\ (\lambda\ (x)\ exp)\ v)$

表达式 $(Let\ x\ 1\ (Let\ x\ 2\ (+\ x\ 1)))$ 中的 (+ x 1) 将用值为 2 的 x，因为他在内部 Let 的作用域内。

实现卫生宏的方法有很多，但对于现有的重组糖方法，卫生宏的重组糖[11]并不是那么容易。而在我们的方法中，实现卫生宏的重组糖很简单。

以表达式 $(Let\ x\ 1\ (Let\ x\ 2\ (+\ x\ 1)))$ 为例。我们会先将它单步展开到 $(Apply\ (\lambda\ (x)\ (Let\ x\ 2\ (+\ x\ 1)))\ 1)$，接着用 Apply 的规则进行规约，发现将 $(Let\ x\ 2\ (+\ x\ 1)))$ 中的 x 置换为 1 后，表达式并不符合规则，因此这不是一条

有效的规约，不能对外层 Let 糖先展开。因此输出的第一个重组糖中间序列是 $(Let\ x\ 1\ (+\ 2\ 1))$

而在实现过程中，我们发现借助 PLT Redex 的 $\#:refers-to$ 命令，我们可以更简单的实现更漂亮的卫生宏重组糖。

测试结果如图5。

(Let x (+ 1 2) (+ x (Let x (+ 1 4) (+ x 1))))

↓

(Let x 3 (+ x (Let x (+ 1 4) (+ x 1))))

↓

((λ (x) (+ x (Let x (+ 1 4) (+ x 1)))) 3)

↓

(+ 3 (Let x (+ 1 4) (+ x 1)))

↓

(+ 3 (Let x 5 (+ x 1)))

↓

(+ 3 ((λ (x) (+ x 1)) 5))

↓

(+ 3 (+ 5 1))

↓

(+ 3 6)

↓

9

图 5 Let 糖测试

1.4 递归糖

递归糖是对于现有重组糖方法很难处理的一种语法糖，指的是多个语法糖相互调用。在本节我们主要用如下例子说明。

$(Odd\ e) \rightarrow (if\ (>\ e\ 0)\ (Even(-\ e\ 1)\ \#f))$

$(Even\ e) \rightarrow (if\ (>\ e\ 0)\ (Odd(-\ e\ 1)\ \#t))$

看似不复杂的两条语法糖规则，对于现有方法来说并不是一件容易的事情。具体原因我们将在下一高阶糖和下一节中仔细探讨。测试结果如图6。

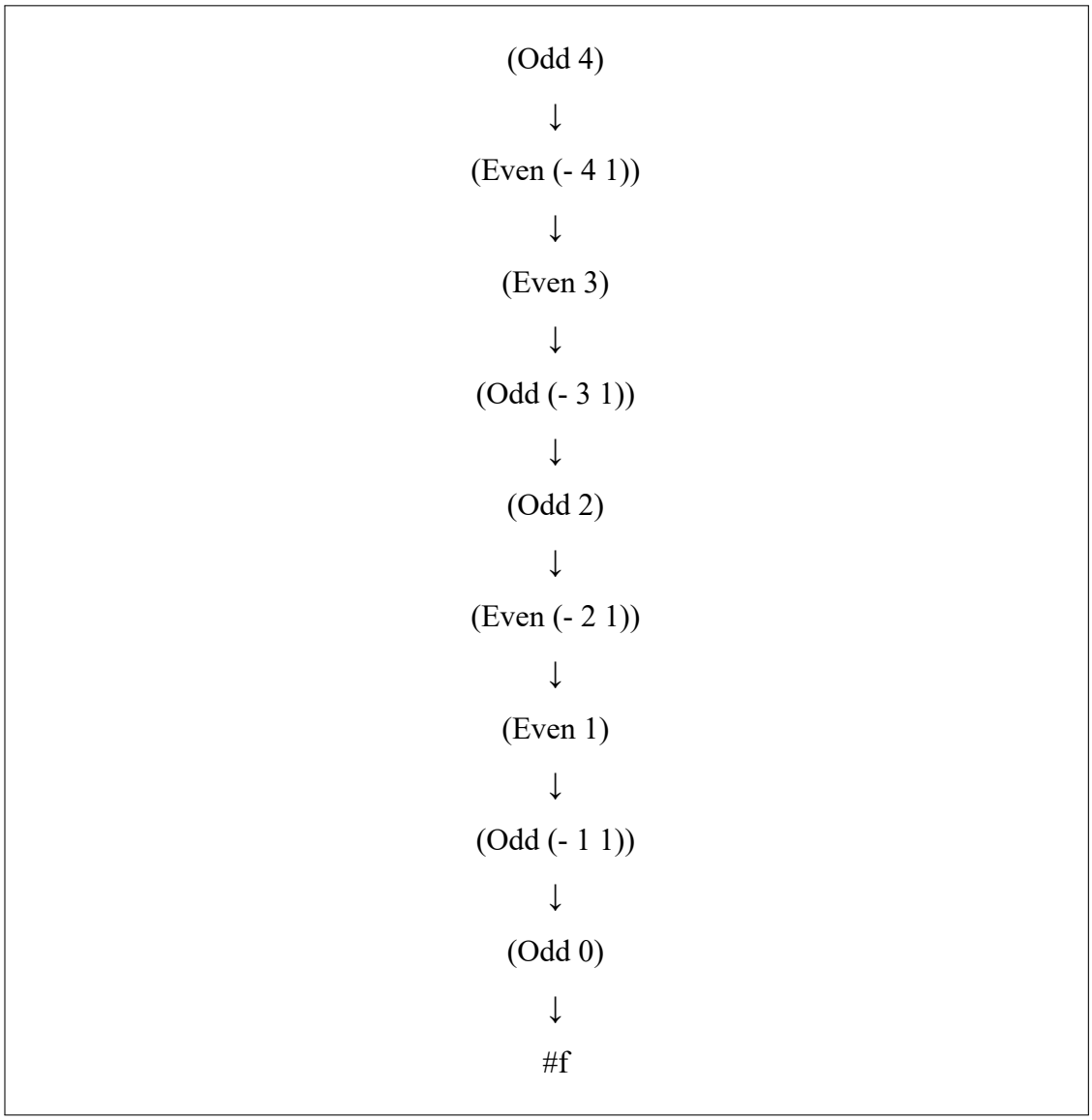

图 6 递归糖 Odd、Even 测试

1.5 高阶糖

高阶糖本质上和递归糖没有区别，但在实际应用中，对高阶糖的支持是很有意义的。

我们分别用两种不同的语法糖形式实现了 map 和 $filter$ 两个语法糖。

$(map\ e\ (list\ v_1 \ldots)) \rightarrow$

$(if\ (empty\ (list\ v_1 \ldots))\ (list)\ (cons\ (e\ (first\ (list\ v_1 \ldots)))\ (map\ e\ (rest\ (list\ v_1 \ldots)))))$

$(filter\ e\ (list\ v_1\ v_2 \ldots)) \rightarrow$

$(if\ (e\ v_1)\ (cons\ v_1\ (filter\ e\ (list\ v_2 \ldots)))\ (filter\ e\ (list\ v_2 \ldots)))$

$(filter\ e\ (list)) \rightarrow (list)$

可以看出，对 map 糖我们将边界条件用解糖后的 if 表达式进行了限制；而对 filter 糖，我们用两个糖的参数区分了不同的条件用来表示边界条件。在保证同名不同参数的糖没有二义性的前提下，我们允许一糖多参，让书写语法糖更加简单。

测试结果如图7、8。

(map (λ (x) (+ 1 x)) (list 1 2 3))

↓

(cons 2 (map (λ (x) (+ 1 x)) (list 2 3)))

↓

(cons 2 (cons 3 (map (λ (x) (+ 1 x)) (list 3))))

↓

(cons 2 (cons 3 (cons 4 (map (λ (x) (+ 1 x)) (list)))))

↓

(cons 2 (cons 3 (cons 4 (list))))

↓

(cons 2 (cons 3 (list 4)))

↓

(cons 2 (list 3 4))

↓

(list 2 3 4)

图 7 高阶 map 糖测试

(filter (λ (x) (and (> x 3) (< x 6))) (list 1 2 3 4 5 6 7))

↓

(filter (λ (x) (and (> x 3) (< x 6))) (list 2 3 4 5 6 7))

↓

(filter (λ (x) (and (> x 3) (< x 6))) (list 3 4 5 6 7))

↓

(filter (λ (x) (and (> x 3) (< x 6))) (list 4 5 6 7))

↓

(cons 4 (filter (λ (x) (and (> x 3) (< x 6))) (list 5 6 7)))

↓

(cons 4 (cons 5 (filter (λ (x) (and (> x 3) (< x 6))) (list 6 7))))

↓

(cons 4 (cons 5 (filter (λ (x) (and (> x 3) (< x 6))) (list 7))))

↓

(cons 4 (cons 5 (filter (λ (x) (and (> x 3) (< x 6))) (list))))

↓

(cons 4 (cons 5 (list)))

↓

(cons 4 (list 5))

↓

(list 4 5)

图 8 高阶 filter 糖测试

我们接下来讨论为什么我们的方法能处理递归语法糖。以

$(map\ (\lambda\ (x)\ (+\ 1\ x))\ (list\ 1\ 2\ 3))$ 为例。如果将该糖完全展开，将变成如下表达式

$(if\ (empty\ (list\ 1\ 2\ 3))$

$(list)$

$(cons\ ((\lambda\ (x)\ (+\ 1\ x))\ (first\ (list\ 1\ 2\ 3)))$ 接下页

$$
\begin{array}{l}
(if\ (empty\ (rest\ (list\ 1\ 2\ 3))) \\
\quad (list) \\
\quad (cons\ ((\lambda\ (x)\ (+\ 1\ x))\ (first\ (rest\ (list\ 1\ 2\ 3))))\ \ldots
\end{array}
$$

我们可以看出，由于糖的分支条件写在了语法糖内部，展开过程中不会判断 if 分支，因此语法糖第二个子表达式 $(list\ 1\ 2\ 3)$ 会在递归中不断变为

$$(rest\ (list\ 1\ 2\ 3))\ (rest\ (rest\ (list\ 1\ 2\ 3)))\ \ldots$$

而不会停止。而我们的重组糖方法则不需要将语法糖完全展开，因此不会发生这种事情。实际上，将语法糖展开条件写在语法糖内部是一个很自然的想法，而现有的语法糖基本上并不支持这么做，而更多的需要手动书写边界条件，这对于作为实现领域特定语言的工具的语法糖来说，是不太友好的。

1.6 其他例子

1.6.1 惰性求值

我们的工具设计之初以 call-by-value 的 λ 演算作为内部语言的基础。而实际应用时，惰性求值，也就是 call-by-need 也是比较常见的。我们试图构造一个惰性求值 λ 演算的语法糖。

$$
\begin{array}{l}
(shell\ (\lambda_N\ x\ \ldots\ e)) \rightarrow (\lambda\ (x\ \ldots)\ (shell\ e)) \\
(shell\ (e_1\ e_2)) \rightarrow ((shell\ e_1)\ (\lambda\ (x_{new})\ (shell\ e_2))) \\
(shell\ (\lambda\ (x\ \ldots)\ (shell\ e))) \rightarrow e
\end{array}
$$

并进行一个样例测试，结果如图9。

结果发现，我们构造的 shell 糖和 λN 糖虽然能够在我们的框架下执行得到求值序列，但并没有进行重组糖。经过简单思考，我们发现原因在于惰性求值本身就是不会对子表达式优先进行计算，因此根据核心算法的 $Rule2.2.2$，语法糖展开后不会重组，因此得到的序列虽然满足重组糖的性质，但看起来没有重组。而用 shell 糖来构造这种 call-by-need 到 call-by-value 的转化是一种自认为不错的想法，因此也记录在这里。

1.6.2 SKI 组合子

SKI 组合子演算[19] 是无类型 λ 演算的一种简化版，其运用 $S\ K\ I$ 三个组合子的组合使用，可以表达无类型 λ 演算的所有行为，是一个极小的图灵完备语

(shell (((λN x y x) xx) ((λN x x) yy)))

↓

((shell ((λN x y x) xx)) (λ (x_{new}) (shell ((λN x x) yy))))

↓

(((shell (λN x y x)) (λ (x_{new1}) (shell xx))) (λ (x_{new}) (shell ((λN x x) yy))))

↓

(((λ (x y) (shell x)) (λ (x_{new1}) (shell xx))) (λ (x_{new}) (shell ((λN x x) yy))))

↓

((λ (y) (shell (λ (x_{new1}) (shell xx)))) (λ (x_{new}) (shell ((λN x x) yy))))

↓

(shell (λ (x_{new1}) (shell xx)))

↓

xx

图 9 lazy λ 糖测试

言。[20] 我们在我们的工具上尝试构造这三个组合子，并尝试输出 SKI 组合子表达式的重组糖序列。语法糖定义如下[1]：

$S \rightarrow (\lambda\ (x_1\ x_2\ x_3)\ (x_1\ x_2\ (x_1\ x_3)))$

$K \rightarrow (\lambda\ (x_1\ x_2)\ x_1)$

$I \rightarrow (\lambda\ (x)\ x)$

并构造一个将两个参数对置的 SKI 演算，结果如图10。

而实际上，对 SKI 组合子，构建在惰性求值的 λ 演算是一种更合适的语法糖，于是我们构造了惰性求值的 λ_N 作为 coreLang 的一部分，并重新测试该表达式。结果如图11。

[1]此处实现时，为了方便，我们将 S 写做 (S *comb*)（满足 Racket 的 pair 形式即可）

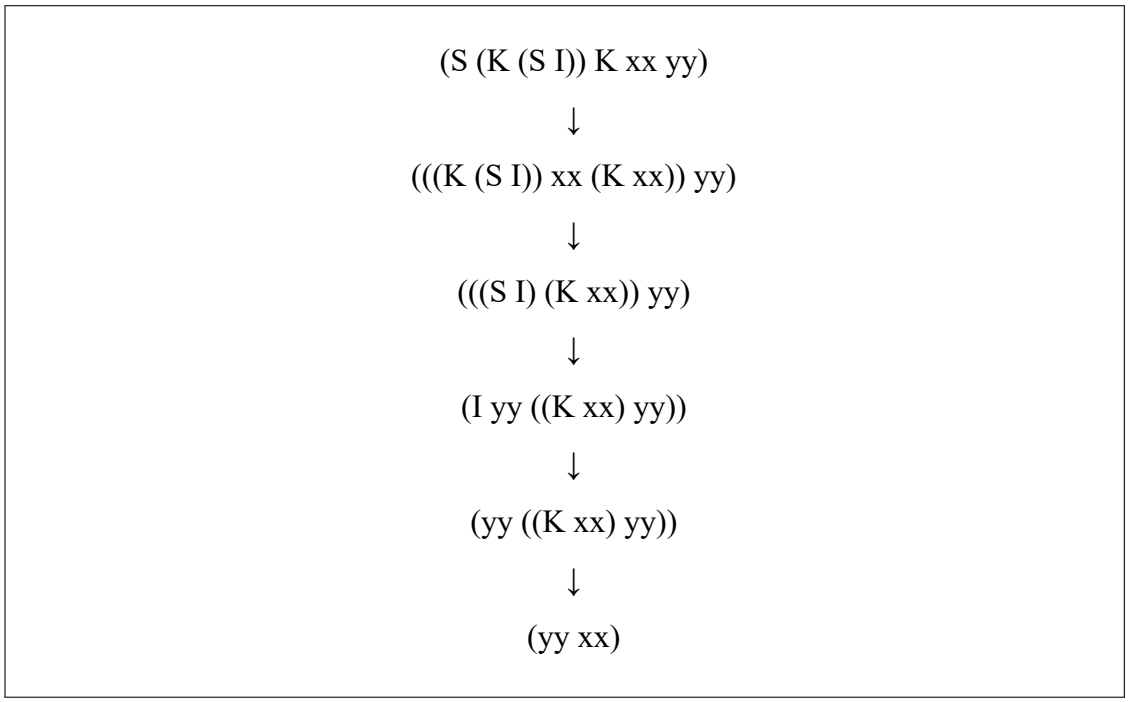

图 11 SKI 糖测试（基于 lazy λ 演算）

结果显示我们的重组糖工具漂亮的显示了该 SKI 表达式的序列。

2. 与现有方法对比

正如前文多次提到的，我们的方法与现有重组糖方法最大的不同是，不会完全将语法糖展开。我们的轻量级重组糖方法先比于现有工作有如下优点。

- 轻量级。不将语法糖完全展开，意味着不需要对语法糖表达式展开后的内部语言表达式的多步执行过程进行一次又一次的匹配和替代。因为对于一个相对比较大的程序来说，这样的匹配开销并不小，因此我们的做法—只在需要（不得不）展开语法糖时再将语法糖破坏的方法，理论上很大程度节省了重组糖的时间开销。

- 对卫生宏友好。重组糖系列工作第二篇用一个新的数据结构—抽象语法有向无环图，来处理卫生宏的重组糖问题；而正如上一小节所介绍1.2，我们的系统处理卫生宏和普通宏的区别只在于借用 PLT Redex 的重命名；并且即使不用这个功能，算法依然可以有效运行。[2]这条优点本身也属于一种

[2]需要对 Rule2.2 进行微小改动，代码中也有运行实例

“轻量级”的表现—算法本身简单以至于拓展性强，而现有工作则对算法进行重新设计。

- 更多语法糖特性（尤其递归语法糖）。正如前一节所介绍的，我们的方法处理递归糖和高阶糖的能力，是现有方法很难达到的。原因很简单：递归糖需要处理终止条件，而本身语法糖处理终止条件就不是一件容易的事情。而得益于规约语义的强大能力，加上算法本身的精心设计，让处理递归这件事成为可能，进而让高阶语法糖成为可能。我们可以发现，高阶函数作为函数式语言的一个非常重要的特性，在近年来被多种其他编程范式的语言所支持。因此支持递归的语法糖是本工作十分重要的优势。

- 更简单的书写语法糖。相比于 resugaring 系列工作[10]，我们的方法得益于规约语义的友好形式，对语法糖的书写并不必须是模式匹配为基础的。只要可以保证对同一个语法糖没有出现多种解糖方式，我们可以对语法糖的各种条件进行分类处理。比如说实现一个斐波那契数列语法糖，我们可以设计如下解糖规则。（对应写在规约规则中）

  $(f\ 0) \rightarrow 1$

  $(f\ 1) \rightarrow 1$

  $(f\ x) \rightarrow (+\ (f\ (-\ x\ 1))\ (f\ (-\ x\ 2)))$ side-condition (x>1)

  这种语法糖写法相对于简单的 pattern based 语法糖，更加友好。

在前文有提到，我们需要限制语法糖和内部语言的结构化1.1。该限制最大的影响就是语言不能有副作用，我们将在下一节对该问题进行讨论。

3. 一些其他讨论

3.1 副作用

表达式不能用对外副作用一定程度我们我们方法的便利性。在一定解决范围内，我可以基于 let（变量绑定）解决部分需求。

不能有副作用的原因，是因为：一旦语法糖的子表达式带有副作用，而当表达式并没有执行到带副作用语句前是可以重组糖的，执行后便不能重组。而对于嵌套表达式来说，判断是否执行过有副作用的语句是一件很困难的事情。可能的解决方案是设计一个检测副作用的算法来增强我们的系统。

如下的语法糖

$(Sg\ exp_1\ exp_2) \rightarrow (begin\ (if\ exp_1\ (set!\ x\ 0)\ (set!\ x\ exp_2))\ x)$

如果 exp_1 不包含副作用，那么显然可以不展开 Sg 糖，先对 exp_1 进行规约，再根据 exp_1 的值决定之后的重组糖序列。

但是一旦 exp_1 表达式包含了副作用，我们就不得不将 Sg 糖展开。此时该糖就不会有重组糖序列，只会得到最后 x 的值。

另一种可能的解决方法，是将副作用封装到内部语言的两个函数中，比如表达式 exp_1 中将 x 改为 x'，y 改为 y'，则 $f(exp_1)$ 为 $(list\ (cons\ x\ x')\ (cons\ y\ y'))$，如果别的表达式需要得到 exp_1 的副作用，就将 $f(exp_1)$ 保存到一个 let 绑定中，供别的表达式使用。

另一方面，我们可以发现的是，对于有副作用的语法糖来说，重组糖的是没有很大作用的。因为当一个语法糖表达式内部副作用语句被执行，该糖就不能重组了。这也是现有方法很难处理递归糖的原因之一—尽管不支持递归调用，现有工作实际上可能可以用 letrec 之类的内部语言表达式处理递归糖，但是由于 letrec 糖本质上是有副作用的，因此尽管可以写递归的糖，却不能输出任何中间序列。

3.2 相关工作

重组糖 (Resugaring) 系列工作[10][11][12][13] 是和我们工作最相关的工作。前两篇工作都是和本文类似的—对执行序列重组糖；第三篇是对语法糖的变量作用域实现自动推导，第四篇则是对语法糖的类型规则进行自动推导。这一系列工作与我们的目的很相似，尤其是前两篇工作。而受到重组糖系列工作的启发，我们希望将语法糖的求值规则也能进行自动推导，因为类型规则本身是求值规则的抽象。而本文是对语法糖求值规则自动推导的一个探索，尽管没有得到自动推导，但我们的核心思想—单步不完全展开既改进了现有对执行序列的重组糖，又对求值规则的自动推导有启发作用。

宏形成 (Macrofication)[21] 是一个对代码内部进行匹配，希望自动生成宏，用以进行代码重构的研究工作。而本文工作可以让利用宏形成得到重构的代码更加简洁—得到的宏看作领域特定语言，即可得到基于自动生成宏的执行序列。因为我们的实现设定十分简单，从得到自动生成的宏、到对生成的宏进行轻量级重组糖很容易做到。

基于伽罗瓦连接的函数式语言程序切片[22] 是为了让函数式语言的程序在执

行期间自动将执行过程进行自我阐述（比如程序在什么位置出现什么错误）。该工作和本文基本想法有相似之处—对结构化语言的子表达式进行递归运算。

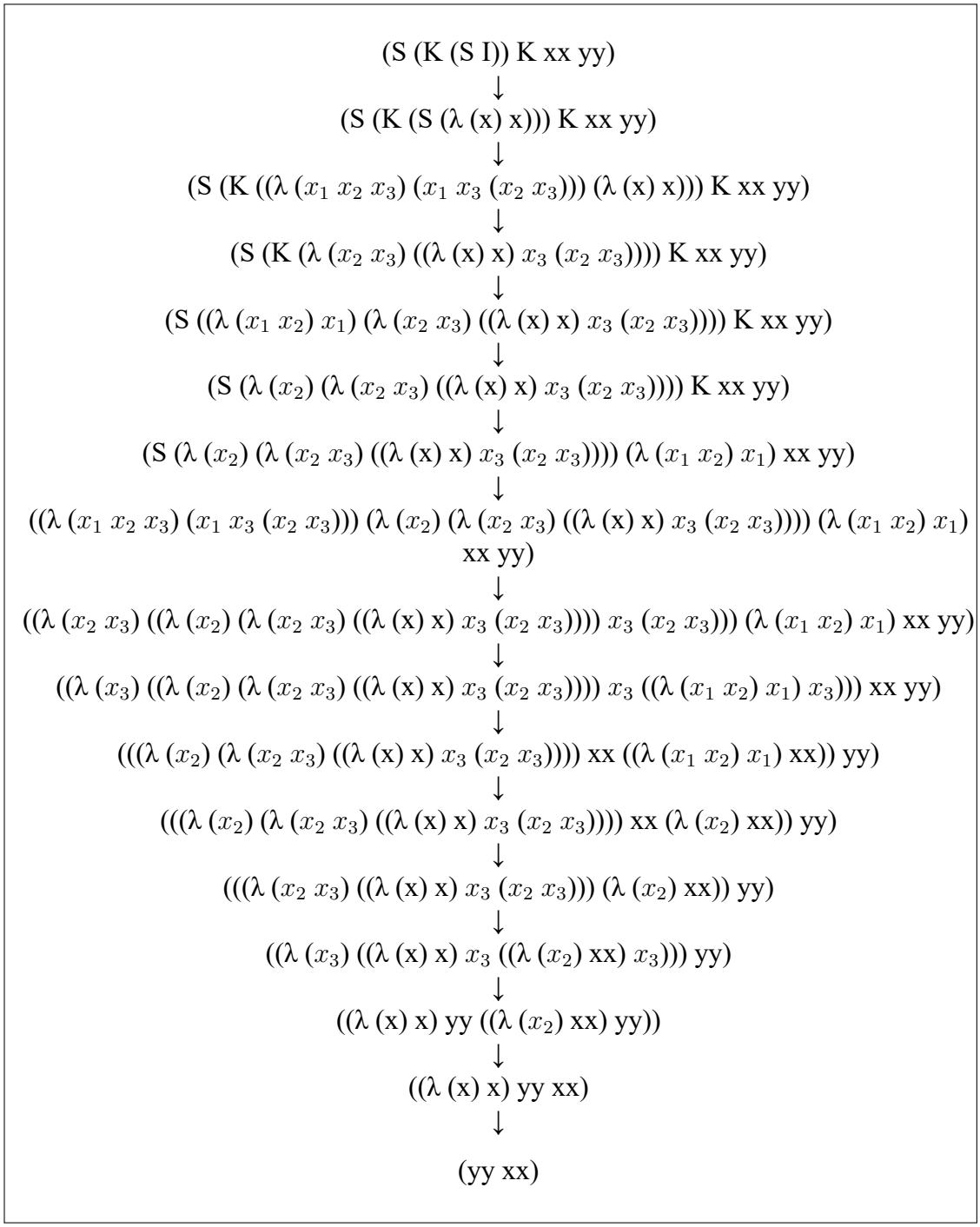

图 10 SKI 糖测试（基于 CBV λ 演算）

第六章 总结与展望

1. 结论

我们的重组糖方法相对于现有方法，是一种更轻量级的算法—用单步尝试加上不完全展开的基础思想，解决了现有工作不能处理递归糖、高阶糖的问题，并且让处理语法糖尤其是卫生宏的流程简单而自然。我们基于 PLT Redex 工具，实现了我们的轻量级重组糖框架，并在其基础上进行对多种特性的语法糖进行测试，并对特殊的应用进行尝试，结果显示我们的重组糖方法确实处理了更多的语法糖特性，且工具使用简单。我们在开头的**本文主要贡献**4小节总结了本文，此处不再赘述。

2. 未来可能的工作

2.1 副作用语法糖

正如前文3所提到的，目前该工作不能很好的处理有副作用的语法糖，尽管本身对带有副作用的语法糖进行重组糖没有什么意义，我们依然希望将副作用在语法糖中的确切难点弄清楚并试图解决，因为这对于下面的 DSL 解释器工作有相当的影响。

2.2 DSL 解释器

我们的工作初衷是为了解决如下的问题：

> 给定内部（通用语言）的解释器（或求值规则），加上外部语言（领域特定语言）的映射关系，用算法自动推导出外部语言（不依赖内部语言）的解释器（或求值规则）。

在对问题初步思考后，结合学习阅读 resugaring 系列工作，提出了本文主要工作的想法。而我们的最终目标依然没有变。

广义上讲，本文实现的工具是一个解释器—它将带有语法糖的表达式，在表面语言上一步一步的解释执行。但最终这个解释执行依然是包含算法对内部语言的一些尝试，没有脱离对内部语言的依赖。然而本工作核心算法的部分思想对此目标有借鉴意义，今后的工作可能会朝这方向发展，将该方法抽象到符号层面。

2.3 多步程序综合

程序综合研究在今年来发展火热，而目前大多数基于例子的程序综合都是给定一个输入和对应的输出，得到想要的表达式。针对多步执行序列为例子的程序综合目前还没有被深入研究，而该研究是具有意义的。我们的工作和 DSL 的解释器工作一定程度上就是一个对序列进行综合的问题；如果能够将 DSL 解释器研究清楚，可能对于多步程序综合的研究具有一定关联。

参考文献

[1] FOWLER M. Addison-wesley signature series (fowler): Domain-specific languages[M/OL]. Pearson Education, 2010. https://books.google.com.hk/books?id=ri1muolw_YwC.

[2] THOMPSON K. Programming techniques: Regular expression search algorithm[J/OL]. Commun. ACM, 1968, 11(6):419–422. https://doi.org/10.1145/363347.363387.

[3] DATE C J, DARWEN H. A guide to the sql standard (4th ed.): A user’s guide to the standard database language sql[M]. USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[4] TEXTUALITY T. Extensible markup language (xml) 1.0 (second edition)[J]. W3C Rec., 2000.

[5] LANDIN P J. The Mechanical Evaluation of Expressions[J/OL]. The Computer Journal, 1964, 6(4):308-320. https://doi.org/10.1093/comjnl/6.4.308.

[6] FLATT M. Creating languages in racket[J/OL]. Commun. ACM, 2012, 55(1):48–56. https://doi.org/10.1145/2063176.2063195.

[7] CULPEPPER R, FELLEISEN M, FLATT M, et al. From macros to dsls: The evolution of racket[C/OL]//LERNER B S, BODÍK R, KRISHNAMURTHI S. LIPIcs: volume 136 3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019: 5:1-5:19. https://doi.org/10.4230/LIPIcs.SNAPL.2019.5.

[8] WAMPLER D, PAYNE A. Programming scala: Scalability = functional programming + objects[M]. 2nd ed. [S.l.]: O’Reilly Media, Inc., 2014.

[9] ERDWEG S, RENDEL T, KÄSTNER C, et al. Sugarj: library-based syntactic language extensibility[C]//Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications. [S.l.: s.n.], 2011: 391-406.

[10] POMBRIO J, KRISHNAMURTHI S. Resugaring: Lifting evaluation sequences through syntactic sugar[J/OL]. SIGPLAN Not., 2014, 49(6):361–371. https://doi.org/10.1145/2666356.2594319.

[11] POMBRIO J, KRISHNAMURTHI S. Hygienic resugaring of compositional desugaring [J/OL]. SIGPLAN Not., 2015, 50(9):75–87. https://doi.org/10.1145/2858949.2784755.

[12] POMBRIO J, KRISHNAMURTHI S, WAND M. Inferring scope through syntactic sugar [J/OL]. Proc. ACM Program. Lang., 2017, 1(ICFP). https://doi.org/10.1145/3110288.

[13] POMBRIO J, KRISHNAMURTHI S. Inferring type rules for syntactic sugar[J/OL]. SIGPLAN Not., 2018, 53(4):812–825. https://doi.org/10.1145/3296979.3192398.

[14] FELLEISEN M, FINDLER R B, FLATT M. Semantics engineering with plt redex[M]. 1st ed. [S.l.]: The MIT Press, 2009.

[15] FELLEISEN M, HIEB R. The revised report on the syntactic theories of sequential control and state[J/OL]. Theor. Comput. Sci., 1992, 103(2):235–271. https://doi.org/10.1016/0304-3975(92)90014-7.

[16] CHURCH A, ROSSER J B. Some properties of conversion[J/OL]. Transactions of the American Mathematical Society, 1936, 39(3):472-482. http://www.jstor.org/stable/1989762.

[17] PLOTKIN G. A structural approach to operational semantics[J/OL]. J. Log. Algebr. Program., 2004, 60-61:17-139. DOI: 10.1016/j.jlap.2004.05.001.

[18] PLOTKIN G. Call-by-name, call-by-value and the λ-calculus[J/OL]. Theoretical Computer Science, 1975, 1(2):125 - 159. http://www.sciencedirect.com/science/article/pii/0304397575900171. DOI: https://doi.org/10.1016/0304-3975(75)90017-1.

[19] CURRY H B. Grundlagen der kombinatorischen logik[J/OL]. American Journal of Mathematics, 1930, 52(3):509-536. http://www.jstor.org/stable/2370619.

[20] SMULLYAN R. A beginner's further guide to mathematical logic: Principles and applications (with companion media pack)fourth edition of rapid prototyping fourth edition[M/OL]. World Scientific Publishing Company, 2016. https://books.google.com/books?id=tFEyDwAAQBAJ.

[21] SCHUSTER C, DISNEY T, FLANAGAN C. Macrofication: Refactoring by reverse macro expansion[C]//Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632. Berlin, Heidelberg: Springer-Verlag, 2016: 644–671.

[22] RICCIOTTI W, STOLAREK J, PERERA R, et al. Imperative functional programs that explain their work[J/OL]. Proc. ACM Program. Lang., 2017, 1(ICFP). https://doi.org/10.1145/3110258.

本科期间的主要工作和成果

本科期间参加的主要科研项目
本研基金

1. 校长基金. 校长基金（理）. 熊英飞. 1 年

各种科研项目

1. TV Backscatter. 课程项目
2. Static Analysis using Inductive Logic Programming. 本科生科研
3. DryadSynth Solver. 暑期科研

致谢

感谢胡振江老师对我毕设工作期间的指导。前期和胡老师的对 DSL 项目的多次探讨激发了本文的初步想法，并在胡老师的指引下对工作不断完善。胡老师对我研究方法、思考方式、写作等方面的指导都受益匪浅。尽管有时无心学习进度缓慢，胡老师依然对有限的进展予以肯定，让我不好意思继续拖延，最终能够让标志着本科结束的工作有一个令自己满意的结果。

感谢熊英飞老师对我本科生科研期间及之后的指导。熊老师是我正式步入科研的引路人，且对我很多学习习惯的培养、和我对工作生活态度的探讨对我帮助很大。并且毕设是熊老师推荐我和胡老师做的。

感谢 17 级本科生关智超同学。关智超同学在今年 3 月加入胡老师和我的项目讨论，并在之后对我论文写作方面提出宝贵建议。特别是本文中“Resugaring”一词的中文翻译—重组糖是关智超同学提出的。

感谢我的舍友们。尽管由于疫情原因，本文工作及写作大部分时间不是在宿舍进行的，但他们三个人努力学习，创造宿舍良好的学习环境，让不爱学习的我羞愧难当，没有完全不学习；并且宿舍的卫生条件、生活气氛等等都十分不错，让我在本科三年半的时间里生活愉快。

感谢女朋友周宇航同学。周宇航同学在我整个本科期间督促我学习、陪我玩、帮助我、支持我。毕设工作的代码和论文经常是在夜间进行，她和她的猫经常熬夜陪我。

感谢父母和其他家人。他们一直在支持我的各种选择，并且在物质上和精神上帮助我。父母对我要求极低，导致我有一个自认为不错的性格，以至于快乐的生活。

感谢本科期间遇见的所有老师和同学及其他人士、动植物和其他事物。这些人和事构成了我本科期间的无数的碎片化回忆。