# Lifting Resugaring by Lazy Desugaring

ANONYMOUS AUTHOR(S)

Syntactic sugar, first coined by Peter J. Landin in 1964, has proved to be very useful for defining domain-specific languages and extending languages. Unfortunately, when syntactic sugar is eliminated by transformation, it obscures the relationship between the user's source program and the transformed program. Resugaring is a powerful technique to resolve this problem, which automatically converts the evaluation sequences of desugared expression in the core language into representative sugar's syntax in the surface language. However, the traditional approach relies on reverse application of desugaring rules to desugared core expression whenever possible. When a desugaring rule is complex and a desugared expression is large, such reverse desugaring becomes very complex and costive.

In this paper, we propose a novel approach to resugaring by lazy desugaring, where reverse application of desugaring rules is unnecessary. We recognize a sufficient and necessary condition for a syntactic sugar to be desugared, and propose a reduction strategy, based on evaluator of the core languages and the desugaring rules, which can produce all necessary resugared expressions on the surface language. Furthermore, we show that this approach can be made more efficient by automatic derivation of evaluation rules for syntactic sugars. We have implemented a system based on this new approach. Compared to the traditional approach, the new approach is not only more efficient, but also more powerful in that it cannot only deal with all cases (such as hygienic and simple recursive sugars) published so far, but can do more allowing more flexible recursive sugars.

Additional Key Words and Phrases: Resugaring, Syntactic Sugar, Interpreter, Domain-specific language, Reduction Semantics

## 1 INTRODUCTION

Syntactic sugar, first coined by Peter J. Landin in 1964 [?], was introduced to describe the surface syntax of a simple ALGOL-like programming language which was defined semantically in terms of the applicative expressions of the core lambda calculus. It has proved to be very useful for defining domain-specific languages (DSLs) and extending languages [??]. Unfortunately, when syntactic sugar is eliminated by transformation, it obscures the relationship between the user's source program and the transformed program.

Resugaring is a powerful technique to resolve this problem [??]. It can automatically convert the evaluation sequences of desugared expression in the core language into representative sugar's syntax in the surface language. As demonstrated in Section 2, the key idea in this resugaring is "tagging" and "reverse desugaring": it tags each desugared core term with the corresponding desugared rule, and follows the evaluation steps in the core language but keep applying the desugaring rules reversibly as much as possible to find surface-level representations of the tagged core terms.

While it is natural to do resugaring by reverse desugaring of tagged core terms, it introduces complexity and inefficiency.

- *Impractical to handle recursive sugar.* While tagging is used to remember the position of desugaring so that reverse desugaring can be done at the correct position when a desugared core expression is evaluated, it becomes very tricky and complex when recursive sugars are considered. Moreover, it can only handle the recursive sugar which can be written by pattern-based desugaring rules [?].
- *Complicated to handle hygienic sugar.* For reverse desugaring, we need to match part of the core expression on the RHS (which could be much larger than LHS) of the desugaring rule

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

and to get the surface term by substitution. But when a syntactic sugar introduces variable bindings, this match-and-substitute turns out to be very complex if we consider local bindings (hygienic sugars) [?].

- *Inefficient in reverse desugaring*. It needs to keep checking whether reverse desugaring is applicable during the evaluation of desugared core expressions, which is costive. Moreover, the match-and-substitute for reverse desugaring is expensive particularly when the core term is large.

In this paper, we propose a novel approach to resugaring, which does not use tagging and reverse desugaring at all. The key idea is "lazy desugaring", in the sense that desugaring is delayed so that the reverse application of desugaring rules becomes unnecessary. To this end, we consider the surface language and the core language as one language, and reduce expressions in such a smart way that all resugared expressions can be fully produced based on the reduction rules in the core language and the desugaring rules for defining syntactic sugars. To gain more efficiency, we propose an automatic method to compress a sequence of core expression reductions into a one-step reduction of the surface language, by automatically deriving evaluation rules (consist of the context rules and reductions rules) of the syntactic sugars based on the syntactic sugar definition and evaluation rules of the core language.

Our main technical contributions can be summarized as follows.

- We propose a novel approach to resugaring by lazy desugaring, where reverse application of desugaring rules becomes unnecessary. We recognize a sufficient and necessary condition for a syntactic sugar to be desugared, and propose a reduction strategy, based on evaluator of the core languages and the desugaring rules, which is sufficient to produce all necessary resugared expressions on the surface language. We prove the correctness of our approach.
- We show that evaluation rules for syntactic sugars can be fully derived for a wide class of desugaring rules and the evaluation rules of the core language. We design an inference automaton to capture symbolic execution of a syntactic sugar based on the evaluation and desugaring rules, and derive evaluation rules for new syntactic sugars from the inference automaton. These evaluation rules, if derivable, can be seamlessly used with lazy desugaring to make resugaring more efficient.
- We have implemented a system based on the new resugaring approach. It is much more efficient than the traditional approach, because it completely avoids unnecessary complexity of the reverse desugaring. It is more powerful in that it cannot only deal with all cases (such as hygienic and simple recursive sugars) published so far [??], but can do more allowing more flexible recursive sugars. All the examples in this paper have passed the test of the system.

The rest of our paper is organized as follows. We start with an overview of our approach in Section 2. We then discuss the core of resugaring by lazy desugaring in Section 3, and automatic derivation of evaluation rules for syntactic sugars in Section 4. We describe the implementation and show some applications as case studies in Section 5. We discuss related work in Section 6, and conclude the paper in Section 7.

## 2 OVERVIEW

In this section, we give a brief overview of our approach, explaining its difference from the traditional approach and highlighting its new features. To be concrete, we will consider the following

$$\begin{aligned} & \texttt{(And (Or \#t \#f) (And \#f \#t))} \\ \longrightarrow\ & \texttt{(And \#t (And \#f \#t))} \\ \longrightarrow\ & \texttt{(And \#f \#t)} \\ \longrightarrow\ & \texttt{\#f} \end{aligned}$$

Fig. 1. A Typical Resugaring Example

simple core language, defining boolean expressions based on if construct:

$$\begin{array}{rcll} e & ::= & (\text{if e e e}) & \text{// if construct} \\ & | & \#t & \text{// true value} \\ & | & \#f & \text{// false value} \end{array}$$

The semantics of the language is very simple, consisting of the following context rule defining the computation order:

$$\frac{e \ \longrightarrow\ e'}{(\text{if } e\ e_1\ e_2)\ \longrightarrow\ (\text{if } e'\ e_1\ e_2)}$$

and two reduction rules:

$$(\text{if } \#t\ e_1\ e_2) \longrightarrow e_1 \qquad (\text{if } \#f\ e_1\ e_2) \longrightarrow e_2$$

Assume that our surface language is defined by two syntactic sugars defined by:

$$\begin{array}{c} (\text{And } e_1\ e_2)\ \longrightarrow_d\ (\text{if } e_1\ e_2\ \#f) \\ (\text{Or } e_1\ e_2)\ \longrightarrow_d\ (\text{if } e_1\ \#t\ e_2) \end{array}$$

Now let us demonstrate how to execute (And (Or #t #f) (And #f #t)), And get the resugaring sequence as Figure 1 by both the traditional approach and our new approach.

### 2.1 Traditional Approach: Tagging and Reverse Desugaring

As we discussed in the introduction, the traditional approach uses ”tagging” and ”reverse desugaring” to get resugaring sequences; tagging is to mark where and how the core terms are from, and reverse desugaring is to resugar core terms back to surface terms. Putting it simply, the traditional resugaring process is as follows.

$$\begin{array}{ll} & \text{(And (Or \#t \#f) (And \#f \#t))} \\ \dashrightarrow & \{\text{ fully desugaring And tagging }\} \\ & \text{(if-Andtag (if-Ortag \#t \#t \#f) (if-Andtag \#f \#t \#f) \#f)} \\ \longrightarrow & \{\text{ context rule of if, reduction rule of if-false}\} \\ & \text{(if-Andtag \#t (if-Andtag \#f \#t \#f) \#f)} \\ \longrightarrow & \{\text{ emit (And \#t (And \#f \#t)) by reverse desugaring, reduction rule of if-true}\} \\ & \text{(if-Andtag \#f \#t \#f) //check resugarable} \\ \longrightarrow & \{\text{ emit (And \#f \#t) by reverse desugaring, reduction rule of it-false}\} \\ & \text{\#f} \end{array}$$

In the above, the surface expression is fully desugared before resugaring. It is worth noting that some desugared subexpressions (e.g., the if-Andtag subexpression) are not touched in the first two steps after desugaring, but each reverse desugaring tries on them, which is redundant and costive. This would be worse in practice, because we usually have lots of intermediate reduction steps which will be tried by reverse desugaring (but may not succeed) during the evaluation of a more complex core expression. Therefore many useless resugarings on subexpressions would take place in the traditional approach. Moreover, reverse resugaring may introduce complexity in the resugaring process, as discussed in the introduction.

### 2.2 Our Approach: Resugaring by Lazy Desugaring

To solve the problem in the traditional approach, we propose a new resugaring approach by eliminating "reverse desugaring" via "lazy desugaring", where a syntactic sugar will be desugared only when it is necessary. We test the necessity of desugaring by a one-step try. We shall first briefly explain our one-step try resugaring method, and then show how the "one-step try" can be cheaply done by derivation of evaluation rules for syntactic sugars.

(resugar (And (Or #t #f) (And #f #t)))

$\dashrightarrow$ { a one-step try on the outermost And }
(try (if (Or #t #f) (And #f #t) #f))

$\longrightarrow$ { should reduce on subexpression (Or #t #f) of And, delay desugaring of And}
(And (resugar (Or #t #f)) (And #f #t)) // no reduction

$\dashrightarrow$ { a one-step try on Or }
(And (try (if #t #t #f)) (And #f #t))

$\longrightarrow$ { keep this try, finish inner resugaring, then return to the top }
(resugar (And #t (And #f #t)))

$\dashrightarrow$ { a one-step try on the outermost And }
(try (if #t (And #f #t) #f))

$\longrightarrow$ { the reduction will destroy structure of the outermost And, so keep this try to desugar it }
(resugar (And #f #t))

$\dashrightarrow$ { a one-step try on the outermost And }
(try (if #f #t #f)) // have to desugar for reduction

$\longrightarrow$ #f

For each step in the above, we take at most one reduction step and move resugaring focus if desugaring is unnecessary. There are 7 reduction steps for our whole resugaring, while the traditional approach needs 9 steps (3 in desugaring, 3 in evaluation, 3 for reverse resugaring). Note that reverse desugaring is more complex and costive because of match and substitution. Note also that the traditional approach would be more redundant if it works on larger expressions.

We can go further to make our approach more efficient. As the purpose of a "one-step try" is to determine the computation order of the syntactic sugar, we should be able to derive this computation order through the desugaring rules and the computation orders of the core language, rather than just through runtime computation of the core expression as done in the above. As will be shown in Section 4, we can automatically derive the following evaluation rules for both *And* and *Or*.

$$\frac{e_1 \;\rightarrow\; e_1'}{(\text{And } e_1\ e_2) \rightarrow (\text{And } e_1'\ e_2)} \qquad (\text{And } \#t\ e_2) \rightarrow e_2 \quad (\text{And } \#f\ e_2) \rightarrow \#f$$

$$\frac{e_1 \;\rightarrow\; e_1'}{(\text{Or } e_1\ e_2) \rightarrow (\text{Or } e_1'\ e_2)} \qquad (\text{Or } \#t\ e_2) \rightarrow \#t \quad (\text{Or } \#f\ e_2) \rightarrow e_2$$

Now with these rules, our resugaring will need only 3 steps as in Figure 1.

$$
\begin{array}{llll}
\text{CoreExp} & ::= & x & \text{variable} \\
 & | & c & \text{constant} \\
 & | & (\text{CoreHead CoreExp}_1 \ \ldots \ \text{CoreExp}_n) & \text{constructor} \\
\\
\text{SurfExp} & ::= & x & \text{variable} \\
 & | & c & \text{constant} \\
 & | & (\text{SurfHead SurfExp}_1 \ \ldots \ \text{SurfExp}_n) & \text{sugar expression}
\end{array}
$$

Fig. 2. Core and Surface Expressions

Two remarks are worth making here. First, we do not require a complete set of reduction/context rules for syntactic sugars; if we have these rules, we can elaborate them to remove one-step try, and make a shortcut for a sequence of evaluation steps on core expression. For example, suppose that we have another syntactic sugar named *Hard* whose evaluation rules cannot be derived. We can still do resugaring on (And (Hard (And #t #t) ...) ...), as we do early, and the derivate rules of And sugar can be used for avoiding part of one-step tries in the basic approach. Second, our example does not show the case when a surface expression contains language constructs of the core expression. This does not introduce any difficulty in our method; as we have no reverse desugaring, there is no worry about desugaring an original core expression to a syntactic sugar. For instance, we can deal with (And (if #t (And #f #t) #f) #f), without resugaring it to (And (And #t (And #f #t)) #f).

## 2.3 New Features

As will be seen clearly in the rest of this paper, our approach has the following new features.

- *Efficient.* As we do not have "tagging" and costive and repetitive "reverse desugaring", our approach is much more efficient than the traditional approach. As discussed above, by deriving reduction/context rules for the syntactic sugars, we can gain more efficiency.
- *Powerful.* As we do not need "reverse desugaring", we can avoid complicated matching when we want to deal with local bindings (hygienic sugars) or more involved recursively defined sugars (see map in Section 5.1.3).
- *Lightweight.* Our approach can be cheaply implemented using the PLT Redex tool [?]. This is because our "lazy desugaring" is much simpler than the "reverse desugaring" which needs development of matching/substitution algorithms.

# 3 RESUGARING BY LAZY DESUGARING

In this section, we present our new approach to resugaring. Different from the traditional approach that clearly separates the surface from the core languages, we intentionally combine them as one mixed language, allowing free use of the language constructs in both languages. We will show that any expression in the mixed language can be evaluated in such a smart way that a sequence of all expressions that are necessary to be resugared by the traditional approach can be correctly produced.

## 3.1 Mixed Language for Resugaring

As a preparation for our resugaring algorithm, we define a mixed language that combines a core language with a surface language (defined by syntactic sugars over the core language). An expression in this language is reduced step by step by the evaluation rules for the core language and the desugaring rules for the syntactic sugars in the surface language.

*3.1.1 Core Language.* For our core language, its evaluator is driven by evaluation rules (context rules and reduction rules), with two natural assumptions. First, the evaluation is deterministic, in the sense that any expression in the core language will be reduced by a unique reduction sequence. Second, evaluation of a sub-expression has no side-effect on other parts of the expression. (We discuss how to trickily extend our approach with a black-box stepper as the evaluator in Appendix A.1)

An expression form of the core language is defined in Figure 2. It is a variable, a constant, or a (language) constructor expression. Here, $\mathrm{CoreHead}$ stands for a language constructor such as $\mathrm{if}$ and $\mathrm{let}$. To be concrete, we will use a simplified core language defined in Figure 3 to demonstrate our approach.

$$
\begin{array}{lcl}
\mathrm{CoreExp} & ::= & (\mathrm{CoreExp}\ \mathrm{CoreExp}\ ...)\ \text{// apply} \\
 & | & (\mathrm{lambda}\ (\mathrm{x}\ ...)\ \mathrm{CoreExp})\ \text{// call-by-value} \\
 & | & (\mathrm{lambdaN}\ (\mathrm{x}\ ...)\ \mathrm{CoreExp})\ \text{// call-by-need} \\
 & | & (\mathrm{if}\ \mathrm{CoreExp}\ \mathrm{CoreExp}\ \mathrm{CoreExp}) \\
 & | & (\mathrm{let}\ \mathrm{x}\ \mathrm{CoreExp}\ \mathrm{CoreExp}) \\
 & | & (\mathrm{first}\ \mathrm{CoreExp}) \\
 & | & (\mathrm{empty?}\ \mathrm{CoreExp}) \\
 & | & (\mathrm{rest}\ \mathrm{CoreExp}) \\
 & | & (\mathrm{cons}\ \mathrm{CoreExp}\ \mathrm{CoreExp}) \\
 & | & (\mathrm{arithop}\ \mathrm{CoreExp}\ \mathrm{CoreExp})\ \text{// +, -, *, /, >, <, =} \\
 & | & \mathrm{x} \\
 & | & \mathrm{c}\ \text{// boolean, number and list}
\end{array}
$$

Fig. 3. A Core Language Example

*3.1.2 Surface Language.* Our surface language is defined by a set of syntactic sugars, together with some basic elements in the core language, such as constant, variable. The surface language has expressions as given in Figure 2.

A syntactic sugar is defined by a desugaring rule in the following form:

$$(\mathrm{SurfHead}\ e_1\ e_2\ \ldots\ e_n)\ \to_d\ \mathrm{exp}$$

where its LHS is a simple pattern (unnested) and its RHS is an expression of the surface language or the core language, and any pattern variable (e.g., $e_1$) in LHS only appears once in RHS. For instance, we may define syntactic sugar $\mathrm{And}$ by

$$(\mathrm{And}\ e_1\ e_2)\ \to_d\ (\mathrm{if}\ e_1\ e_2\ \#f).$$

Note that if the pattern is nested, we can introduce a new syntactic sugar to flatten it. And if we need to use a pattern variable multiple times in RHS, a $\mathrm{let}$ binding may be used (a normal way in syntactic sugar). We take the following sugar as an example

$$(\mathrm{Twice}\ e_1)\ \to_d\ (+\ e_1\ e_1).$$

If we execute $(\mathrm{Twice}\ (+\ 1\ 1))$, it will firstly be desugared to $(+\ (+\ 1\ 1)\ (+\ 1\ 1))$, then reduced to $(+\ 2\ (+\ 1\ 1))$ by one step. The subexpression $(+\ 1\ 1)$ has been reduced but should not be resugared to the surface, because the other $(+\ 1\ 1)$ has not been reduced yet. So we just use a $\mathrm{let}$ binding to resolve this problem. The RHS should be $(\mathrm{let}\ \mathrm{x}\ e_1\ (+\ \mathrm{x}\ \mathrm{x}))$ in this case.

Note that in the desugaring rule, we do not restrict the RHS to be a $\mathrm{CoreExp}$. We can use $\mathrm{SurfExp}$ (more precisely, we allow the mixture use of syntactic sugars and core expressions) to

| | | |
|---|---|---|
| Exp | ::= | DisplayableExp |
| | \| | UndisplayableExp |
| DisplayableExp | ::= | SurfExp |
| | \| | CommonExp |
| UndisplayableExp | ::= | CoreExp' |
| | \| | OtherSurfExp |
| | \| | OtherCommonExp |
| CoreExp | ::= | CoreExp' |
| | \| | CommonExp |
| | \| | OtherCommonExp |
| CoreExp' | ::= | (CoreHead' Exp*) |
| SurfExp | ::= | (SurfHead DisplayableExp*) |
| CommonExp | ::= | (CommonHead DisplayableExp*) |
| | \| | $c$ // constant value |
| | \| | $x$ // variable |
| OtherSurfExp | ::= | (SurfHead Exp* UndisplayableExp Exp*) |
| OtherCommonExp | ::= | (CommonHead Exp* UndisplayableExp Exp*) |

Fig. 4. Our Mixed Language

define recursive syntactic sugars, as seen in the following example.

$$\begin{array}{l} (\text{Odd } e) \ \rightarrow_d \ (\text{if } (> \ e \ 0) \ (\text{Even } (- \ e \ 1)) \ \#f) \\ (\text{Even } e) \ \rightarrow_d \ (\text{if } (> \ e \ 0) \ (\text{Odd } (- \ e \ 1)) \ \#t) \end{array}$$

We assume that all desugaring rules are not overlapped in the sense that for a syntactic sugar expression, only one desugaring rule is applicable for a single sugar in the expression.

*3.1.3 Mixed Language.* Our mixed language for resugaring combines the surface language and the core language, described in Figure 4. The differences between expressions in our core language and those in our surface language are identified by their Head. But there may be some expressions in the core language which are also used in the surface language for convenience, or we need some core language's expressions to help us getting better resugaring sequences. So we take CommonHead out from the CoreHead, which can be displayed in resugaring sequences (the rest of CoreHead becomes CoreHead'). The CoreExp' denotes expressions with undisplayable CoreHead (named CoreHead'). The SurfExp denote expressions that have SurfHead and their subexpressions being displayable. The CommonExp denotes expressions with displayable Head (named CommonHead) in the core language, together with displayable subexpressions. There exist some other expressions during our resugaring process, which have displayable Head, but one or more of their subexpressions should not be displayed. They are of UndisplayableExp. We distinguish these two kinds of expressions for the *abstraction* property (discussed in Section 3.3.2).

As an example, for the core language in Figure 3, we may assume if, let, lambdaN (call-by-name lambda calculus), empty?, first, rest as CoreHead', op, lambda (call-by-value lambda calculus), cons as CommonHead. This will allow op, lambda and cons to appear in the surface language, and thus display more intermediate steps during resugaring.

Note that some expressions with CoreHead contain subexpressions with SurfHead, they are of CoreExp but not in the core language. In the mixed language, we process these expressions by the context rules of the core language, so that the reduction rules of core language and the desugaring rules of surface language can be mixed as a whole (the $\rightarrow_c$ in next section). For example, suppose we have the context rule of if expression

$$\frac{e_1 \rightarrow e_1'}{(\text{if } e_1\ e_2\ e_3) \rightarrow (\text{if } e_1'\ e_2\ e_3)}$$

then if $e_1$ is a reducible expression in the core language, it will be reduced by the reduction rule in the core language; if $e_1$ is a SurfExp, it will be reduced by the desugaring rule of $e_1$'s Head (how the subexpression reduced does not matter, because it is just to mark the location where it should be reduced); if $e_1$ is also a CoreExp with non-core subexpressions, a recursive reduction by $\rightarrow_c$ is needed.

### 3.2 Resugaring Algorithm

Our resugaring algorithm works on the mixed language, based on the evaluation rules of the core language and the desugaring rules for defining the surface language. Let $\rightarrow_c$ denote the one-step reduction of the CoreExp (described in previous section), and $\rightarrow_d$ the one-step desugaring of outermost sugar. We define $\rightarrow_m$, a one-step reduction of our mixed language, as follows.

$$\frac{\begin{array}{c}\exists i.\, (\text{CoreHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \rightarrow_c\ (\text{CoreHead}\ e_1\ \ldots\ e_i'\ \ldots\ e_n)\ \textit{and}\ e_i\ \in\ \text{SurfExp}\\ e_i\ \rightarrow_m\ e_i''\end{array}}{(\text{CoreHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \rightarrow_m\ (\text{CoreHead}\ e_1\ \ldots\ e_i''\ \ldots\ e_n)}\quad (\textsc{CoreRed1})$$

$$\frac{\begin{array}{c}\nexists i.\, (\text{CoreHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \rightarrow_c\ (\text{CoreHead}\ e_1\ \ldots\ e_i'\ \ldots\ e_n)\ \textit{and}\ e_i\ \in\ \text{SurfExp}\\ (\text{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_c\ e'\end{array}}{(\text{CoreHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \rightarrow_m\ e'}\quad (\textsc{CoreRed2})$$

$$\frac{\begin{array}{c}(\text{SurfHead}\ x_1\ \ldots\ x_i\ \ldots\ x_n)\ \rightarrow_d\ e\\ \exists i.\, e[e_1/x, \ldots, e_i/x_i, \ldots, e_n/x_n]\ \rightarrow_m\ e[e_1/x, \ldots, e_i'/x_i, \ldots, e_n/x_n]\\ e_i\ \rightarrow_m\ e_i''\end{array}}{(\text{SurfHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \rightarrow_m\ (\text{SurfHead}\ e_1\ \ldots\ e_i''\ \ldots\ e_n)}\quad (\textsc{SurfRed1})$$

$$\frac{\begin{array}{c}(\text{SurfHead}\ x_1\ \ldots\ x_i\ \ldots\ x_n)\ \rightarrow_d\ e\\ \nexists i.\, e[e_1/x_1, \ldots, e_i/x_i, \ldots, e_n/x_n]\ \rightarrow_m\ e[e_1/x_1, \ldots, e_i'/x_i, \ldots, e_n/x_n]\end{array}}{(\text{SurfHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \rightarrow_m\ e[e_1/x_1, \ldots, e_i/x_i, \ldots, e_n/x_n]}\quad (\textsc{SurfRed2})$$

Putting them in simple words, for an expression with CoreHead, we just use the evaluation rules of core language combined with desugaring rules as reduction rules. As described in the previous section, if a SurfExp subexpression is to be reduced (CoreRed1), we should mark the location and apply $\rightarrow_m$ on it recursively; otherwise (CoreRed2), the expression is just reduced by the $\rightarrow_c$ 's rule. For the expression with SurfHead, we will first expand the outermost sugar (identified by the SurfHead), then recursively apply $\rightarrow_m$ on the partly desugared expression. In the recursive call, if one of the original subexpression $e_i$ is reduced then the original sugar is not necessarily desugared, we should only reduce the subexpression $e_i$ (SurfRed1); otherwise, the sugar should be desugared (SurfRed2).

Now, our desugaring algorithm can be easily defined based on $\rightarrow_m$ .

$$
\begin{array}{rcl}
\text{resugar}(e) & = & \textbf{if } \text{isNormal}(e) \textbf{ then } \textit{return} \\
& & \textbf{else} \\
& & \quad \textbf{let } e \rightarrow_m e' \textbf{ in} \\
& & \quad \textbf{if } e' \in \text{DisplayableExp} \\
& & \quad\quad \text{output}(e'),\ \text{resugar}(e') \\
& & \quad \textbf{else } \text{resugar}(e')
\end{array}
$$

During the resugaring, we just apply the reduction ( $\rightarrow_m$ ) on the input expression step by step until it becomes a normal form, while outputting those intermediate expressions that belong to DisplayableExp. This algorithm is more explicit and simpler than the traditional one. Note that $\rightarrow_m$ is executed recursively on the subexpressions, which provides rooms for further optimization (see Section 5).

### 3.3 Correctness

Following the traditional resugaring approach [??], we show that our resugaring approach satisfied the following three properties: (1) *Emulation*: For each reduction of an expression in our mixed language, it should reflect on one-step reduction of the expression totally desugared in the core language, or one step desugaring on a syntactic sugar; (2) *Abstraction*: Only displayable expressions defined in our mixed language appear in our resugaring sequences; and (3) *Coverage*: No syntactic sugar is desugared before its sugar structure should be destroyed in core language.

*3.3.1 Emulation.* This is the basic property for a correct resugaring. Since our desugaring does not change an expression after it is totally desugared, what we need to show is that a non-desugaring reduction in the mixed language is exactly the same reduction which should appear after the expression is totally desugared. We use fulldesugar(exp) to represent the expression after exp totally desugared in this section.

Lemma 3.1 (Emulation). *For* exp = (SurfHead $e_1 \ \ldots \ e_i \ \ldots \ e_n$) $\in$ SurfExp, *if* exp $\rightarrow_m$ exp' *and* fulldesugar*(*exp*)* $\neq$ fulldesugar(exp')*, then* fulldesugar(exp) $\rightarrow_c$ fulldesugar(exp').

Lemma 3.2. *For* exp = (SurfHead $e_1 \ \ldots \ e_i \ \ldots \ e_n$)*, if* $\rightarrow_c$ *reduces* fulldesugar(exp) *at the expression original from $e_i$ in one step, then* $\rightarrow_m$ *will reduce* exp *at $e_i$.*

Proof of Lemma 3.2. Correctness of lemma 3.2 is obvious, because the $\rightarrow_m$ always expands the outermost sugar to check which exact subexpression is to be reduced by $\rightarrow_c$ . And if the expansion of outermost sugar is not enough to get the location of reduced subexpression, the $\rightarrow_m$ will be called recursively. □

Proof of Lemma 3.1. We need to consider two cases—reduction rules SurfRed1 and SurfRed2, because they are rules for SurfExp.

For SurfRed1 rule, (SurfHead $e_1 \ \ldots \ e_i \ \ldots \ e_n$) $\rightarrow_m$ (SurfHead $e_1 \ \ldots \ e_i'' \ldots \ e_n$), where $e_i \rightarrow_m e_i''$. If fulldesugar($e_i$) = fulldesugar($e_i''$), that means the i-th subexpression only involves desugaring, then fulldesugar(exp) = fulldesugar(exp'). If fulldesugar($e_i$) $\neq$ fulldesugar($e_i''$), non-desugaring reduction occurs at the subexpression, then what we need to prove is that, fulldesugar(exp) $\rightarrow_c$ fu that means the one-step reduction in $\rightarrow_m$ shows what exactly reduced in the totally desugared expression. Note that the only difference between exp and exp' is at the i-th subexpression. Because of lemma 3.2, the $\rightarrow_m$ reduces exp at the correct subexpression, what we need to prove is fulldesugar($e_i$) $\rightarrow_c$ fulldesugar($e_i''$)—that is the emulation of the subexpression $e_i$. So the equation can be proved recursively.

For SurfRed2 rule, exp' is exp after the outermost sugar desugared. So the expressions after totally desugared are equal, that is fulldesugar(exp) = fulldesugar(exp'). □

*3.3.2 Abstraction.* The correctness of abstraction is obvious, because the resugaring algorithm checks whether the intermediate step is of DisplayableExp before outputting. The abstraction property is related to why we need resugaring—sugar expressions become unrecognizable after desugaring. So different from the abstraction of the traditional approach which only shows the changes of original expression, our approach defines the abstraction by catalog of the expressions in the mixed language. Thus, users can decide which expressions are recognizable, so that resugaring sequences will be abstract enough. For example, recursive sugar expressions' resugaring sequences will show useful information during the execution. Lazy resugaring, as the key idea of our approach, makes any intermediate steps retain as many sugar structures as possible, so the abstraction property of our approach does achieve a good abstraction.

One may ask what if we just want to display core language's expression originated from the input expression (just as what traditional approach does). The solution is that we can write surface mirror term for core expressions. For example, if we want to show resugaring sequences of (and (if (and #t #f) ...) ...) without displaying if expression produced by and, we can write a IF term, with the same evaluation rules as if. Then just input (and (IF (and #t #f) ...) ...) to get the resugaring sequences. It is really flexible.

*3.3.3 Coverage.* The coverage property is important, because resugaring sequences are useless if losing intermediate steps. By lazy desugaring, it becomes obvious, because there is no chance to lose. In Lemma 3.3, we want to show that our reduction rules in the mixed language is *lazy* enough. Because it is obvious, we only give a proof sketch here.

LEMMA 3.3 (COVERAGE). *A syntactic sugar only desugars when* necessary *during the reduction of the mixed language, that means after a core reduction on the fully-desugared expression, the sugar's structure is destroyed.*

PROOF SKETCH OF LEMMA 3.3. From Lemma 3.2, we know the $\rightarrow_m$ recursively reduces an expression at correct subexpression, or the $\rightarrow_m$ will expand the outermost sugar (of the current expression) in rule SurfRed2. Note that SurfRed2 is the only reduction rule to desugar sugars directly (other rules only desugar sugars when recursively call SurfRed2), we can prove the lemma recursively if SurfRed2 is *lazy* enough.

In SurfRed2 rule, we firstly expand the outermost sugar and get a temp expression with the structure of the outermost sugar. Then when we recursively call $\rightarrow_m$, the reduction result shows the structure has been destroyed, so the outermost sugar has to be desugared. Since the recursive reduction of a terminable (some bad sugars may never stop which are pointless) sugar expression will finally terminate, the lemma can be proved recursively. □

## 4 DERIVATION OF EVALUATION RULES

So far, we have shown that our resugaring algorithm can use lazy desugaring to avoid costive reverse desugaring in the traditional approach. However, as shown in the reduction rules SURFRED1 and SURFRED2, we still need a one-step try to check whether a syntactic sugar is required or not. It would be more efficient if such one-step try could be avoided. In this section, we show that this is possible, by giving an automatic method to derive evaluation rules for the syntactic sugar through symbolic computation.

To see our idea, consider a simple syntactic sugar defined by (Not $e_1$) $\rightarrow_d$ (if $e_1$ #f #t). To derive reductions rules for not from those of if, we design *inference automaton* (IFA) that can be used to express and manipulate a set of evaluation rules for a language construct. Assuming that

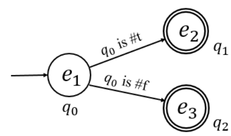
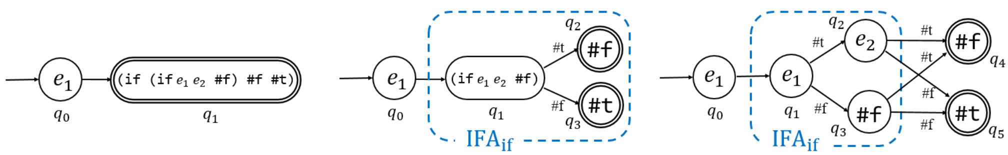



Fig. 5. IFA of if




(a) IFA according to the rules (b) After expanding the outer if (c) After expanding the inner if

Fig. 6. IFA of nand

we already have the IFAs of all language constructs of the core language, our method to construct the evaluation rules of a syntactic sugar is as follows: First, we construct an IFA for the syntactic sugar according to the desugaring rules, then we transform and simplify the IFA, and finally we generate evaluation rules for the syntactic sugar from the IFA.

In this following, we start with some examples of IFA, its formal definition and its normal form, before we proceed to give our algorithm for conversion between evaluation rules and IFA.

### 4.1 Inference Automaton

As mentioned above, IFA intuitively describes the evaluation rules of a certain language construct. To help readers better understand it, we start with some examples, then we give the formal definition of IFA.

*Example 4.1 (IFA of* if*).* Recall the three evaluation rules of if in the overview. Given an expression (if $e_1$ $e_2$ $e_3$), from these rules, we can see that $e_1$ should be evaluated first, then $e_2$ or $e_3$ will be chosen to evaluate depending on the value of $e_1$. The evaluation result of $e_2$ or $e_3$ will be the value of the expression. This evaluation procedure (or the three evaluation rules) for (if $e_1$ $e_2$ $e_3$) can be represented by the IFA in Figure 5.

A node of IFA indicates that the expression needs to be evaluated, and the nodes before it have been evaluated. The arrow from $q_0$ to $q_1$ indicates that this branch is selected when the evaluation result of $e_1$ is #t. The arrow between $e_1$ and $e_3$ is similar. The double circles of $e_2$ and $e_3$ denote that their evaluation result will be the result of the expression with this language construct. In most cases, the transition condition is the evaluation result (an abstract value) of the previous node or a specific value, so to simplify the representation of IFA, we will only mark the value on the transition edge when it is clear from the context.

When an expression with if needs to be evaluated (for example (if (if #t #t #f) #f #t)), first evaluating the $e_1$ ((if #t #t #f)). Note that in this process, evaluating a subexpression requires running another automaton based on its syntax, while the outer automaton hold the state at $q_0$. According to the result of $e_1$ (which is #f in this case), the IFA selects the branch ($e_3$). Then the result of $e_3$ (#t) is the evaluation result of the expression.

□


(a) IFA according to the rules (b) After expanding let

Fig. 7. IFA of and

*Example 4.2 (IFA of* nand*).* Sometimes the rules may be more complex, such as being reduced into another language construct, or an expression containing other syntactic structures. For example, for the following evaluation rules for nand:

$$\frac{e_1 \longrightarrow e_1'}{(\text{nand } e_1\ e_2) \longrightarrow (\text{nand } e_1'\ e_2)}$$

$$(\text{nand } v_1\ e_2) \longrightarrow (\text{if } (\text{if } v_1\ e_2\ \#\text{f})\ \#\text{f}\ \#\text{t})$$

we can draw nand's IFA as Figure 6a. When the automaton runs into the terminal node of Figure 6a, it goes to derive the if expression. In fact, we have known how if works through the IFA of if. Thus we can replace the last node with an $\text{IFA}_{\text{if}}$ and substitute $e_1$ to $e_3$ of $\text{IFA}_{\text{if}}$ with the parameters of the node. Use the termination nodes of $\text{IFA}_{\text{if}}$ as the termination nodes of new $\text{IFA}_{\text{nand}}$. The results are shown in Figure 6b. Further decomposing the inner if node, connecting the terminating nodes of $\text{IFA}_{\text{nand}}$ to the nodes pointed to by the original output edge, we get Figure 6c.

As the nodes of IFA in Figure 6c have no other composite syntactic structures, such an IFA completely expresses the semantics of a language construct, and no longer cares about the evaluation rules of other syntactic structures. We will do similar steps for syntactic sugars, which will be discussed later.

□

*Example 4.3 (IFA of* and*).* Suppose that the evaluation rule of and is defined in a more complex way as follows.

$$(\text{and } e_1\ e_2) \longrightarrow (\text{let } x\ e_1\ (\text{if } x\ e_2\ x))$$

In this case, we use the let binding, and we introduce a symbol table to record the bindings between variables and expressions (corresponding to nodes). The representation of $\text{IFA}_{\text{and}}$ is shown in Figure 7a.

Further discuss the language construct. We first evaluate $e_1$ and save the result in $x$. When evaluating the expression of if, what is actually evaluated is $(\text{if } x\ e_2\ x)[e_1/x]$. We represent it in the form of Figure 7b, where node $q_1$ contains a symbol table for recording the mapping from variables to nodes. Mapping to nodes is to ensure that the variable information will not be lost when the IFA is transformed.

□

Defination 4.1 (Inference Automaton). *An inference automaton (IFA) of language construct* H *is a 5-tuple,* $(Q, \Sigma, q_0, F, \delta)$*, consisting of*

- *A finite set of nodes* $Q$*, each node containing an expression and a symbol table which maps a variable to a node;*
- *A finite set of conditions* $\Sigma$*;*
- *A start node* $q_0 \in Q$*;*
- *A set of terminal nodes* $F \subseteq Q$*; and*
- *A transition function* $\delta : (Q - F) \times \Sigma' \longrightarrow Q$ *where* $\Sigma' \subseteq \Sigma$*.*

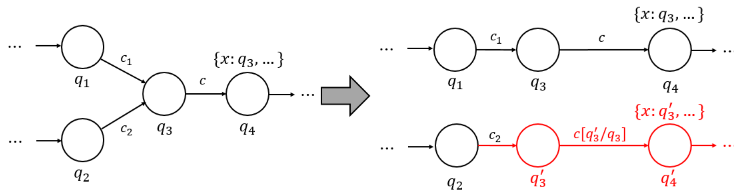



Fig. 8. Transform into a Tree

*and for each node $q$, there is no path to itself (i.e., there is no sequence of conditions $C = (c_1, c_2, \ldots, c_n) \subseteq \Sigma^*$, such that after $q$ transfers sequentially according to $P$, it returns $q$.)*

The state transition depends on whether the expression meets the condition. Note that each IFA is associated with language construct. Each state in IFA represents the current evaluation of a language construct, indicating that some subexpressions of the language construct have been evaluated at this state, and the rest have not.

### 4.2 Normal IFA

Although IFA can describe the evaluation behavior of a language construct, IFA itself may be in a complicated form. For example, the IFA of a language construct contains other syntactic structures in Figure 6a. We will simplify IFA to make it easier to analyze.

Defination 4.2 (Normal IFA). *An IFA is said to be normal if it satisfies the following conditions.*

- *The expression of node $q \in Q$ can only be a pattern variable $e_i$ or a local variable $x$. If it is a local variable, it cannot be in the symbol table of $q$.*
- *For any $q_1, q_2 \in Q$ and $c_1, c_2 \in \Sigma$, $\delta(q_1, c_1) \neq \delta(q_2, c_2)$.*
- *On each branch, each pattern variable $e_i$ can only be evaluated once.*

If an IFA is normal, it means that there are no more composite syntactic structures in it, and that it is of a tree structure. In fact, it is always feasible to convert IFA to normal IFA.

Theorem 4.1 (Normalizability of IFA). *An IFA can be transformed into a normal IFA, if the normal IFAs of all sub-syntactic structures in the IFA are known.*

We can prove this theorem by repeatedly applying the following correctness-preserving steps to normalize an IFA.

*Step 1: Transforming into a Tree.* As we have assumed that evaluation of a subexpression does not have side effect on the evaluation of other subexpressions, a node with at least two sources can be cloned and applied to different branches as shown in Figure 8; replacing the node referenced by the branch with the cloned node in conditions and symbol tables. In this way, we can transform an IFA into a tree so that it meet the second condition for a normal IFA.

*Step 2: Substituting Sub-Language Construct.* If node $q$ in the tree IFA contains an expression of a language construct H, we replace this node with the normal IFA of H, replace the parameters, and pass the symbol table of $q$ into the sub-IFA. Then, connect all the termination nodes of H to the original output node $q'$, and convert it into a tree IFA according to the method described in the previous step as shown in Figure 9. Replace the node referenced by the branch with the *new* one in conditions and symbol tables. This step preserve the semantics because of $(\mathrm{H}\ e_1\ \ldots\ e_n)[e/x] = (\mathrm{H}\ e_1[e/x]\ \ldots\ e_n[e/x])$.

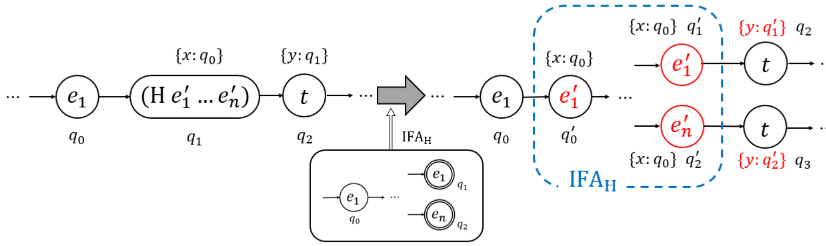



Fig. 9. Substitute Sub-Language Construct

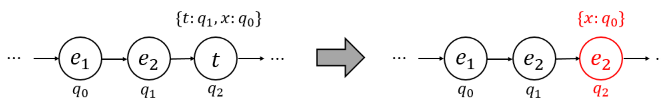



Fig. 10. Replace Variables in the Symbol Table

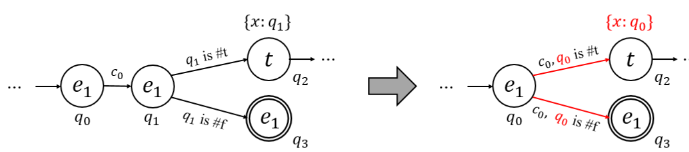



Fig. 11. Remove Evaluated Nodes and Merge Transition Conditions

In particular, if an expression $e$ does not contain a certain variable $x$, we can remove it from the symbol table. For example, in our problem, each pattern variable $e_i$ of syntactic sugar cannot contain unbound variables. So we can remove all symbol tables of nodes with $e_i$.

Note that even for hygienic sugar, this step of substitution is correct. Because what we require to substitute is a normal IFA, if the expression of a node $q$ in sub-IFA is a local variable $x$, it must be unknowable in the substructure ($x$ not in the symbol table of $q$). So the value of $x$ must be passed from the outer language construct.

*Step 3: Replacing Variables in the Symbol Table.* If the expression of a node $q$ is a local variable $x$ and $x$ is in the symbol table of $q$, we can replace $x$ with the expression of the node it points to, then remove $x$ from the symbol table as shown Figure 10. Because $x[e_1/x, e_2/y] = e_1[e_2/y]$, this step is correct.

*Step 4: Removing Evaluated Nodes and Merge Transition Conditions.* If an IFA is a tree, for each branch, remove the non-terminal nodes that have been evaluated with the same symbol table, and merge the conditions on the transition edge as in Figure 11. Replace the node referenced by the branch with the first-evaluated node in conditions and symbol tables. This step can make an IFA satisfy the third constraint of normal IFA.

Since on any branch, if $e_i$ is removed, it must have been evaluated. Therefore, when IFA runs to this node, there is no need to do any evaluation on $e_i$. At the same time, the transfer edge ensures that the conditions are not lacking. Correctness is guaranteed.

*Step 5: Removing Constant Value.* If the expression of a node is a constant value, remove the node and merge the conditions on the transition edge just like evaluated value. Because IFA does not do anything in this node, this does not affect the correctness of IFA.

## 4.3 Converting Evaluation rules to IFA

At the beginning of this section, we have shown several examples about correspondence between IFA and evaluation rules. Now, we give an algorithm that can automatically convert the evaluation rules to IFA and ensure its correctness. But at the same time, it has stricter requirements on the evaluation rules.

In this section, we only discuss core language. Syntactic sugar is similar to this, which we discuss it later.

Assumption 1. *A language construct $H$ only contains the evaluation rules in the following form:*

$$\frac{e_i \to e_i' \quad T}{(\mathrm{H}\ e_1 \ldots e_i \ldots e_n) \to (\mathrm{H}\ e_1 \ldots e_i' \ldots e_n)}$$

$$\frac{T}{(\mathrm{H}\ v_1 \ldots v_p\ e_1 \ldots e_q) \to e}$$

*where $T$ is a constraint over $e_j$ for $j \in 1, 2, \ldots, n$.*

This assumption specifies the form of the evaluation rules to ensure that IFAs can be generated. The first one is a context rule, and the other one is a reduction rule. Rule $(\text{if } \#\text{t } e_1\ e_2) \to e_1$ can be seen as

$$\frac{e \ \mathbf{is}\ \#\text{t}}{(\text{if } e\ e_1\ e_2) \to e_1.}$$

Assumption 2 (Orderliness of Language Construct). *The language construct in the core language is finite. Think of all syntactic structures as points in a directed graph. If one of $H$'s evaluation rules can generate an expression containing $H'$, then construct an edge that points from $H$ to $H'$. The directed graph generated in this method has no circles.*

IFAs are not able to construct syntactic structures that contain recursive rules. This assumption qualifies that we can find an order for all syntactic structures, and when we try to construct IFA for $H$, IFA of $H'$ is known.

Assumption 3 (Determinacy of One-Step Evaluation). *The evaluation rules satisfy the determinacy of one-step evaluation.*

By assumption 3, we can get the following lemma, which points out the feasibility of using a node in IFA to represent the evaluation of subexpressions.

Lemma 4.1. *If an expression $(\mathrm{H}\ e_1\ \ldots\ e_n)$ does a one-step evaluation by a context rule, which is a one-step evaluation of pattern variable $e_i$, then it continues to use this rule until $e_i$ becomes a value.*

Proof of Lemma 4.1. According to Assumption 3, this lemma is trivial. □

### 4.3.1 Converting Algorithm.

Theorem 4.2 (IFA Can Be Constructed by Evaluation Rules). *If all the syntactic structures in the core language satisfy these assumptions, we can construct IFAs for all syntactic structures in the core language.*

Proof of Theorem 4.2. We prove this theorem by giving an algorithm that converts evaluation rules to IFA. By Assumption 2, we get an order of syntactic structures. We construct the IFA for each structure in turn.

We generate a node for each rule of the language construct H and insert them into $Q$. If the rule is a context rule for a pattern varibale $e_i$, set $e_i$ as the expression of the node. If the rule is a reduction rule, add them into $F$ as terminal nodes and set the reduced expression $e$ as the expression of the node. Symbol tables of these nodes are set to be empty. Next we connect these nodes.

For an expression like $(\mathrm{H}\ e_1\ \dots\ e_n)$, suppose that $e_1, \dots, e_n$ are not value, According to Lemma 4.1, we have the unique rule $r$ of H for one-step evaluation. Let node $q$ corresponding to $r$ be $q_0$.

If $r$ is a context rule for $e_i$, let the expression of $q$ be $e_i$. Assume that the evaluation of $e_i$ results in $v_i$, we get expression $(\mathrm{H}\ e_1 \dots e_{i-1}\ v_i\ e_{i+1} \dots e_n)$. For each possible value of $v_i$, choose the rules $r'$ that should be used. The node is $q'$. Set a condition as $c = q$ **is** $v_i$ Let $\delta(q, c)$ be $q'$. For each branch, seem $r'$ as $r$ and keep doing this until $r$ is a reduction rule. □

In this way, we got an IFA of H. According to the lemma 4.1, we can also get a normal IFA of H.

*Example 4.4 (Constructing IFA of* xor *by Rules).* We give an example to show how to convert evaluation rules to IFA using the algorithm mentioned above. Since the symbol tables of all nodes are empty, we omit not to write.

$$\frac{e_1 \longrightarrow e_1'}{(\mathrm{xor}\ e_1\ e_2) \longrightarrow (\mathrm{xor}\ e_1'\ e_2)} \qquad (\text{E-Xor})$$

$$(\mathrm{xor}\ \#\mathrm{t}\ e_2) \longrightarrow (\mathrm{if}\ e_2\ \#\mathrm{f}\ \#\mathrm{t}) \qquad (\text{E-XorTrue})$$

$$\frac{e_2 \longrightarrow e_2'}{(\mathrm{xor}\ \#\mathrm{f}\ e_2) \longrightarrow (\mathrm{xor}\ \#\mathrm{f}\ e_2')} \qquad (\text{E-XorFalse})$$

$$(\mathrm{xor}\ \#\mathrm{f}\ \#\mathrm{t}) \longrightarrow \#\mathrm{t} \qquad (\text{E-XorFalseTrue})$$

$$(\mathrm{xor}\ \#\mathrm{f}\ \#\mathrm{f}) \longrightarrow \#\mathrm{f} \qquad (\text{E-XorFalseFalse})$$

Suppose that xor is a language construct in core language. There are five rules for it. Therefore, we construct five nodes for the rules and set the expression as Figure 12a.

Considering an expression $(\mathrm{xor}\ e_1\ e_2)$, where $e_1$ and $e_2$ are not values. It will be derived by rule (E-Xor). Therefore, set the node of the rule as the start node $q_0$. According to the rules of xor, the evaluation result of $e_1$ can be #t or #f. If the value is #t, the expression will be $(\mathrm{xor}\ \#\mathrm{t}\ e_2)$ and use rule (E-XorTrue) to derive. Then connect $q_0$ and $q_1$ with condition $q_0$ **is** #t. Connect $q_0$ and $q_2$ with condition $q_0$ **is** #f similarly as Figure 12b. Then connect $q_2$ to the last two nodes with conditions according to the value of $e_2$. The IFA of xor can be expressed as Figure 12c.

□

#### 4.3.2 Correctness.

Before proving the correctness of the algorithm, we first prove the following lemma.

Lemma 4.2. *If an expression $e$ of language construct* H *uses rule $r$ to derive, we can find a path from $q_0$ to $q$ generated by $r$.*

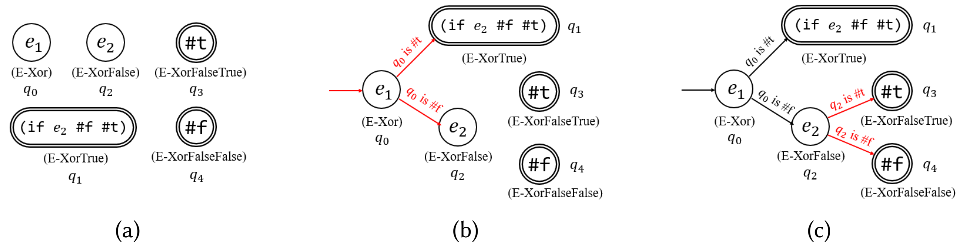



Fig. 12. IFA of xor: Constructed by evaluation rules

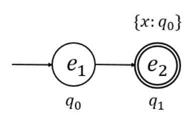



Fig. 13. IFA of let

Proof of Lemma 4.2. Suppose $e = (\mathrm{H}\, v_1\ \dots\ v_p\ e_1\ \dots\ e_q)$. We build an expression $e_0' = (\mathrm{H}\,\mathrm{id}(v_1)\ \dots\ \mathrm{id}(v_p)$ where $\mathrm{id}(x) = x$. $e_0'$ should have the same value as $e$. If $e_0'$ is derived by rule $r_1$, $r_1$ must be a context rule, or $e$ can also be reduced by $r_1$. Also, the node $q_1$ generated by $r_1$ will be the start node. Without loss of generality, assume that $r_1$ is a context rule for $e_1$. We get $e_1' = (\mathrm{H}\, v_1\ \mathrm{id}(v_2)\ \dots\ \mathrm{id}(v_p)\ e_1\ \dots\ e_q)$ after derived by $r_1$. Similarly, we find a rule $r_2$ which is a context rule for $e_2$. The node $q_2$ generated by $r_2$ satisfies $\delta(q_1, (e_1\ \mathbf{is}\ v_1)) = q_2$. By analogy, we can get $e_p' = (\mathrm{H}\ v_1\ \dots\ v_p\ e_1\ \dots\ e_q) = e$ and a path $q_1 (= q_0), q_2, \dots, q_p$. At last, we use rule $r$ to derive $e$ and add $q$ to the path. □

Lemma 4.3 (Correctness). *For any language construct* H*, under our assumption, the normal IFA got by the algorithm in Theorem 4.2 has the same semantics as the rules.*

Proof. We only need to discuss that in a one-step derivation, both get the same result.

Consider an expression $e = (\mathrm{H}\ e_1 \dots e_n)$ and a rule $r$ for derivation. Suppose $r$ generates node $q$. By Lemma 4.2, we find a path from $q_0$ to $q$. The expression $e$ must meet the conditions from $q_0$ to $q$. Therefore, the one-step derivation of this expression $e$ in IFA must be located at $q$. If $r$ is a context rule for a pattern variable $e_i$, the expression of $q$ is $e_i$ as well, and $e_i$ of $e$ is not a value. Thus both of the one-step derivation of $e$ are one-step derivation of $e_i$. If $r$ is a reduction rule to $e'$, both are $e'$.

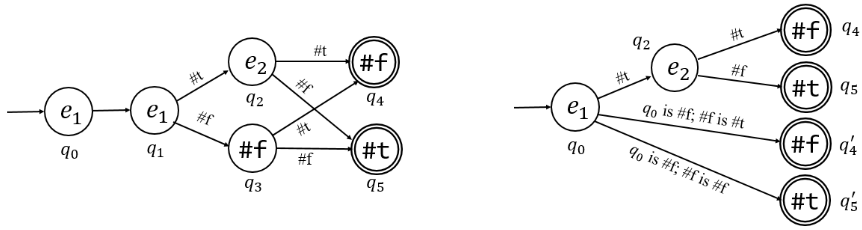
Similarly, if an expression can be derived in one step in IFA, then it must be able to use the corresponding rule for one-step derivation. □

It should be noted that if a certain language construct does not meet our assumptions, it does not mean that this language construct does not have an IFA. We can define its IFA according to its semantics. However, this method cannot be automated and requires users to ensure its correctness. For example, given the evaluation rules of let, we can specify the IFA of let as Figure 13.

$$\frac{e_1 \longrightarrow e_1'}{\mathrm{let}\ x\ e_1\ e_2 \longrightarrow \mathrm{let}\ x\ e_1'\ e_2}$$

$$(\mathrm{let}\ x\ e_1\ e_2) \longrightarrow e_2[e_1/x]$$

In the evaluation rules of let, there is a substitution. Therefore, in IFA of let, we need symbol table to express this. When $e_2$ is evaluated or expanded, it is necessary to replace $x$ with the value of node $q_0$ in $e_2$.




(a) Before Simplification (b) After Simplification

Fig. 14. IFA of nand

### 4.4 Converting IFA to Evaluation Rules

Next we show how to convert IFA back to evaluation rules. Because IFA can be normalized by Theorem 4.1, we only need to convert normal IFA into rules. Unfortunately, IFA can express some derivation methods whose evaluation rules are difficult to describe. Therefore, we have to impose more constraints on IFA to ensure that evaluation rules can be automatically generated.

Assumption 4. *In a normal IFA, if $q \notin F$, then the symbol table of $q$ is empty, and the expression of $q$ cannot be a local variable.*

In fact, this is a strong assumption, which requires that only terminal nodes could have substitution. This is because it is difficult to generate a context rule for an expression after its substitution.

*4.4.1 Algorithm.* Similarly, we first give its algorithm, and then prove its correctness.

Theorem 4.3 (Rules Can Be Constructed by Normal IFA). *For each normal IFA satisfy Assumption 4, it can be converted to evaluation rules.*

Proof. Suppose that the IFA stands for the language construct $H$, then we build evaluation rules for $H$. First traverse all nodes to find the set of all expressions in nodes, which is the parameters of the language construct $H$ like $(\mathrm{H}\ e_1 \ldots e_n)$. Then generate evaluation rule for each node.

Begin with $q_0$, traverse the IFA. Let $q$ be $q_0$. Record the conditions by a set $T$.

Suppose that $q$ is a terminal node, the expression of $q$ is $e$ and the symbol table of $q$ is like $\{x : q_x; y : q_y; \ldots\}$. Let $e_x, e_y, \ldots$ be the expressions of $q_x, q_y, \ldots$. Add a reduction rule like

$$\frac{T}{(\mathrm{H}\ e_1 \ldots e_n) \longrightarrow e[e_x/x][e_y/y]\ldots}$$

If $q$ is not a termination node, and the expression of $q$ is $e_i$, add a context rule like

$$\frac{e_i \longrightarrow e_i' \quad T}{(\mathrm{H}\ e_1\ \ldots\ e_i\ \ldots\ e_n) \longrightarrow (\mathrm{H}\ e_1\ \ldots\ e_i'\ \ldots\ e_n)}$$

For each condition $c$ and node $q'$ satisfying $\delta(q, c) = q'$, do the following steps separately. Replace the node in $c$ with its expression and add it to $T$. Let $q'$ be $q$. Keep doing this until $q$ is a terminal node. □

*Example 4.5 (Construct Rules of* nand *by IFA).* Figure 14a is the IFA of nand which have been constructed. We can simplify it and get a normal IFA as Figure 14b.

Start with $q_0$, we get a context rule as

$$\frac{e_1 \longrightarrow e_1'}{(\mathrm{nand}\ e_1\ e_2) \longrightarrow (\mathrm{nand}\ e_1'\ e_2)}$$

We first discuss the branch of $q_2$. Add $e_1$ **is** #t to $T$. Because $q_2$ is not a terminal node, add a new context rule for $q_2$.

$$\frac{e_2 \to e_2' \quad e_1 \textbf{ is } \#\text{t}}{(\text{nand } e_1\ e_2) \to (\text{nand } e_1\ e_2')}$$

Since $q_4$ is a reduction rule, append $e_2$ **is** #t to $T$ and add a new reduction rule for $q_4$. Reduction rule for $q_5$ is similar.

$$\frac{e_1 \textbf{ is } \#\text{t} \quad e_2 \textbf{ is } \#\text{t}}{(\text{nand } e_1\ e_2) \to \#\text{f}} \quad \frac{e_1 \textbf{ is } \#\text{t} \quad e_2 \textbf{ is } \#\text{f}}{(\text{nand } e_1\ e_2) \to \#\text{t}}$$

Back to $q_0$, for $q_4'$ and $q_5'$ are also terminal nodes, we can build reduction rules for $q_4'$ and $q_5'$ in the same way.

$$\frac{e_1 \textbf{ is } \#\text{f} \quad \#\text{f} \textbf{ is } \#\text{t}}{(\text{nand } e_1\ e_2) \to \#\text{f}} \quad \frac{e_1 \textbf{ is } \#\text{f} \quad \#\text{f} \textbf{ is } \#\text{f}}{(\text{nand } e_1\ e_2) \to \#\text{t}}$$

We can judge that #f is not #t, so we can remove the rule (NandFalse1) from the rules, for it contains a condition that is never met. At the same time, we rewrite the remaining rules into a more customary form.

$$\frac{e_1 \to e_1'}{(\text{nand } e_1\ e_2) \to (\text{nand } e_1'\ e_2)} \quad \frac{e_2 \to e_2'}{(\text{nand } \#\text{t}\ e_2) \to (\text{nand } \#\text{t}\ e_2')}$$

$$(\text{nand } \#\text{t}\ \#\text{t}) \to \#\text{f} \quad (\text{nand } \#\text{t}\ \#\text{f}) \to \#\text{t} \quad (\text{nand } \#\text{f}\ e_2) \to \#\text{t}$$

□

#### 4.4.2 *Correctness.*

Lemma 4.4. *For any language construct* H, *if its normal IFA meets the above assumptions, the evaluation rules obtained according to the algorithm in Theorem 4.3 have the same semantics as IFA.*

Proof. We only need to discuss that in a one-step derivation, both get the same result.

Considering an expression $e = (\text{H } e_1 \ldots e_n)$ use rule $r$ to derive. The expression $e$ must meet the condition $T$ of $r$ such as some parameters must be value or a specific value. Suppose $r$ is generated by node $q$. Then $T$ is the set of all transition conditions from $q_0$ to $q$. Therefore, the one-step derivation of this expression $e$ in IFA must be located at $q$. If $r$ is a context rule for a pattern variable $e_i$, the expression of $q$ is $e_i$ as well, and $e_i$ of $e$ is not a value. Thus both of the one-step derivation of $e$ are one-step derivation of $e_i$.

Similarly, if an expression can be derived in one step in IFA, then it must be able to use the corresponding rule for one-step derivation. □

### 4.5 Deriving Evaluation Rules for Syntactic Sugars

As we have seen in Example 4.5, although the rules of nand contain the use of the language construct of if, the final derived IFA does not. We can apply this procedure to derive reductions rules for each syntactic sugar: construct an IFA for the syntactic sugar, simplify it and convert it into rules.

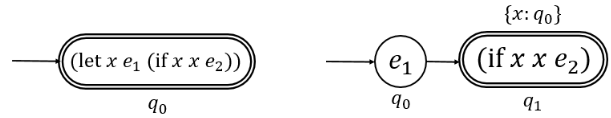
Similarly, we also need to add some constraints on syntactic sugar.

Assumption 5 (Orderliness of Syntactic Sugar). *The definition of each syntactic sugar can only use the language construct in core language and the syntactic sugar that has been defined.*

Defination 4.3. *Considering the following syntactic sugar*

$$(\text{SurfHead } x_1\ \ldots\ x_n)\ \to_d\ e,$$

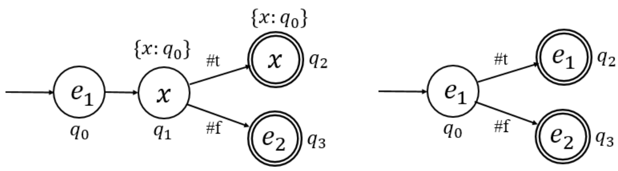
$(\mathrm{Or}' e_1\ e_2) \to (\mathrm{let}\ x\ e_1\ (\mathrm{if}\ x\ x\ e_2))$




(a) Rule of Definition 4.3

(b) IFA generated with the algorithm of Theorem 4.2

(c) Expand language construct of let




$$\frac{e_1 \to e_1'}{(\mathrm{Or}\ e_1\ e_2) \to (\mathrm{Or}\ e_1'\ e_2)}$$

$$\frac{e_1\ \mathbf{is}\ \#\mathrm{t}}{(\mathrm{Or}\ e_1\ e_2) \to e_1}$$

$$\frac{e_1\ \mathbf{is}\ \#\mathrm{f}}{(\mathrm{Or}\ e_1\ e_2) \to e_2}$$

(d) Expand language construct of if

(e) Replace $x$ with expression according to the symbol table

(f) Rules generated with the algorithm of Theorem 4.3

Fig. 15. Example: Syntactic Sugar of or

*the IFA of* SurfHead *is defined as the IFA of language construct* SurfHead' *whose evaluation rule is*

$$(\mathrm{SurfHead'}\ x_1\ \ldots\ x_n) \to e$$

*Example 4.6.* Suppose that there are only if and let in our core language, whose IFAs are known. Now we build rules for syntactic sugar Or.

$$(\mathrm{Or}\ e_1\ e_2)\ \to_d\ (\mathrm{let}\ x\ e_1\ (\mathrm{if}\ x\ x\ e_2))$$

The Or syntactic sugar only uses the syntax structure of the core language, which meets Assumption 5, so we can generate evaluation rules for Or. The IFA of Or is the same as the IFA of Or' whose rule is shown in Figure 15a. Therefore, we can generate an IFA according to the rule as shown in Figure 15b. Next we transform IFA to make it a normal IFA, as shown in Figure 15c to Figure 15e. Finally, according to the structure of IFA, generate or evaluation rules, as shown in Figure 15f.

□

As discussed in Section 3, our approach should satisfy the three properties: emulation, abstraction and coverage. The property of abstraction is obvious, for the rules we generate only contain the language construct itself, without any other structures in core language. However, our approach does not perfectly satisfy emulation and coverage. In the derivation, we lost the information that a language construct is reduced to another language construct. This makes the derivation sequence different. But through Theorem 4.1, Lemma 4.3 and Lemma 4.4, we can guarantee the correctness of the evaluation results.

Another issue is the expressiveness of IFA. As discussed above, we can well support general syntactic sugar and hygieneinderive sugar. But we require that when constructing IFA, the IFA of all its substructures is known. So at present we have difficulty dealing with recursive sugar. In addition, when we construct rules from IFA, if the deformation of IFA causes it to not satisfy the assumption 4, the rules cannot be generated. This would not happen in a core language with only if and let. But for a more complex language, this situation may exist.

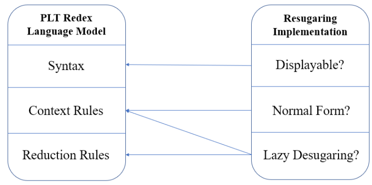



Fig. 16. Framework of Implementation

(a) Subsequences of Sugar2

```
    (Sugar2 ...)
⟶ exp1
⇢  expi //0 or more steps
⟶ expn
```

(b) Optimized Resugaring of Sugar1

```
    (Sugar1 (Sugar2 ...) ...)
⟶ (Sugar1 exp1 ...)
⇢  (Sugar1 expi ...) //0 or more steps
⟶ (Sugar1 expn ...)
```

Fig. 17. Recursive Optimization's Example

## 5 CASE STUDIES

We have implemented our resugaring approach using PLT Redex [?], which is a semantic engineering tool based on reduction semantics [?]. Figure 16 shows the framework of the implementation. In the language model, desugaring rules are written as reduction rules of SurfExp. And context rules of SurfExp have no restriction (every subexpression is reducible as a hole). Then for each resugaring step, we carefully choose the exact evaluation rule which satisfies condition as defined for the mixed language in Section 3.2.

Note that in Rules SurfRed1 and CoreRed1, there is a recursive call on $\rightarrow_m$ . We can optimize the resugaring algorithm in our actual implementation. For example, (Sugar1 (Sugar2 ...) ...) as the input, and find the first subexpression should be reduced. We can first get the resugaring sequences of (Sugar2 ...) as Figure 17a, then the resugaring sequence is got as 17b. Thus, we will not need to try to expand the outermost sugar for each inner step (recursively resugaring for inner expression).

As for the automatic derivation of evaluation rules, we implement a simple demo by writing some core language's IFAs (such as if, let) manually, because it is enough to do some case studies for the resugaring tasks.

### 5.1 Case Studies

We show several case studies to demonstrate the power of our approach. Some examples we will discuss in this section are in Figure 18. Note that we set call-by-value lambda calculus as terms in CommonExp, because we want to output some intermediate expressions including lambda expressions in some examples. It's easy if we want to skip them.

*5.1.1 Simple Sugars.* We construct some simple syntactic sugars and try it on our tool. Some sugar is inspired by the first work of resugaring [?]. The result shows that our approach can handle all sugar features of their first work. Take an SKI combinator syntactic sugar as an example. (We can

```
    (S (K (S I)) K xx yy)
⟶ (((K (S I)) xx (K xx)) yy)
⟶ (((S I) (K xx)) yy)
⟶ (I yy ((K xx) yy))
⟶ (yy ((K xx) yy))
⟶ (yy xx)
```

(a) Example of SKI

```
    (let x 2 (Hygienicadd 1 x))
⟶ (Hygienicadd 1 2)
⟶ (+ 1 2)
⟶ 3
```

(b) Example of Hygienicadd

```
    (Let x 1 (+ x (Let x 2 (+ x 1))))
⟶ (Let x 1 (+ x (+ 2 1)))
⟶ (Let x 1 (+ x 3))
⟶ (+ 1 3)
⟶ 4
```

(c) Example of Let

```
    (Odd 2)
⟶ (Even (- 2 1))
⟶ (Even 1)
⟶ (Odd (- 1 1))
⟶ (Odd 0)
⟶ #f
```

(d) Example of Odd and Even

```
    (Map (lambda (x) (+ x 1)) (cons 1 (list 2)))
⟶ (Map (lambda (x) (+ x 1)) (list 1 2))
⟶ (cons 2 (Map (lambda (x) (+ 1 x)) (list 2)))
⟶ (cons 2 (cons 3 (Map (lambda (x) (+ 1 x)) (list))))
⟶ (cons 2 (cons 3 (list)))
⟶ (cons 2 (list 3))
⟶ (list 2 3)
```

(e) Example of Map

```
    (Filter (lambda (x) (and (> x 1) (< x 4))) (list 1 2 3 4))
⟶ (Filter (lambda (x) (and (> x 1) (< x 4))) (list 2 3 4))
⟶ (cons 2 (Filter (lambda (x) (and (> x 1) (< x 4))) (list 3 4)))
⟶ (cons 2 (cons 3 (Filter (lambda (x) (and (> x 1) (< x 4))) (list 4))))
⟶ (cons 2 (cons 3 (Filter (lambda (x) (and (> x 1) (< x 4))) (list))))
⟶ (cons 2 (cons 3 (list)))
⟶ (cons 2 (list 3))
⟶ (list 2 3)
```

(f) Example of Filter

Fig. 18. Resugaring Examples

regard S as an expression headed with S, without subexpression.)

$$\begin{aligned} \mathrm{S} &\to_d (\mathrm{lambdaN}\ (\mathrm{x1\ x2\ x3})\ (\mathrm{x1\ x2}\ (\mathrm{x1\ x3}))) \\ \mathrm{K} &\to_d (\mathrm{lambdaN}\ (\mathrm{x1\ x2})\ \mathrm{x1}) \\ \mathrm{I} &\to_d (\mathrm{lambdaN}\ (\mathrm{x})\ \mathrm{x}) \end{aligned}$$

Although SKI combinator calculus is a reduced version of lambda calculus, we can construct combinators' sugar based on call-by-need lambda calculus in our core language. For sugar expression (S (K (S I)) K xx yy), we get the resugaring sequences as Figure 18a. During the test, we find 33 intermediate steps are needed after the totally desugaring of the input expression, but only 5 of them can be returned to the surface, so many attempts to reverse the desugaring would fail if using the traditional resugaring approach, in such a little expression. That's why lazy desugaring makes our approach efficient.

As for the derivation of syntactic sugar's evaluation rules, we have shown an example of and sugar and or sugar in overview. But what if the or sugar written as follows?

$$(\mathrm{Or}\ e_1\ e_2) \to_d (\mathrm{let\ x}\ e_1\ (\mathrm{if\ x\ x}\ e_2))$$

Of course, we got the same evaluation rules as the example in overview.

$$\frac{e_1 \to e_1'}{(\mathrm{Or}\ e_1\ e_2) \to (\mathrm{Or}\ e_1'\ e_2)} \qquad (\mathrm{Or}\ \#t\ e2) \to \#t \quad (\mathrm{Or}\ \#f\ e2) \to e_2$$

Then for expressions headed with or, we won't need the one-step try to figure out whether desugaring or processing on a subexpression, which makes our approach more concise. Overall, the unidirectional resugaring algorithm based on lazy desugaring makes our approach efficient, because no attempts for resugaring the expression are needed.

*5.1.2 Hygienic Sugars.* The second work [?] of traditional resugaring approach mainly processes hygienic sugar compared to first work. It uses a DAG to represent the expression. However, hygiene is not hard to be handled by our lazy desugaring strategy. Our algorithm can easily process hygienic sugar without a special data structure. A typical hygienic problem is as the following example.

$$(\mathrm{Hygienicadd}\ e_1\ e_2)\ \rightarrow_d\ (\mathrm{let\ x}\ e_1\ (+\ \mathrm{x}\ e_2))$$

For traditional resugaring approach, if we want to get sequences of $(\mathrm{let\ x\ 2\ (Hygienicadd\ 1\ x)})$, it will firstly desugar to $(\mathrm{let\ x\ 2\ (let\ x\ 1\ (+\ x\ x))})$, which is awful because the two $x$ in $(+\ \mathrm{x\ x})$ should be bind to different values. So the traditional hygienic resugaring approach uses abstract syntax DAG to distinct different x in the desugared expression. But for our approach based on lazy desugaring, the Hygienicadd sugar does not have to desugar until necessary, thus, getting resugaring sequences as Figure 18b based on a non-hygienic rewriting system.

The lazy desugaring is also convenient for achieving hygiene for non-hygienic rewriting. For example, $(\mathrm{let\ x\ 1\ (+\ x\ (let\ x\ 2\ (+\ x\ 1)))})$ may be reduced to $(+\ 1\ (\mathrm{let}\ 1\ 2\ (+\ 1\ 1)))$ by a simple core language whose let expression does not handle cases like that. But by writing a simple sugar Let

$$(\mathrm{Let}\ e_1\ e_2\ e_3)\ \rightarrow_d\ (\mathrm{let}\ e_1\ e_2\ e_3)$$

and making some simple modifies (rejecting one-step try if error occurs, see details in Appendix A.2) in the reduction of mixed language, we will get the resugaring sequences as Figure 18c in our tool. It is not resugaring in the usual sense for violating the emulation property, but can be useful for implementing lightweight hygiene.

In practical application, we think hygiene can be easily processed by rewriting systems, so we just use a rewriting system which can rename variables automatically.

And for the derivation method, there is no rewriting system at all, but the hygiene is handled more concisely. we build a hygienic sugar Hygienicor based on the Or sugar.

$$\begin{aligned}(\mathrm{Or}\ e_1\ e_2)\ &\rightarrow_d\ (\mathrm{let\ x}\ e_1\ (\mathrm{if\ x\ x}\ e_2))\\(\mathrm{Hygienicor}\ e_1\ e_2)\ &\rightarrow_d\ (\mathrm{let\ x}\ e_1\ (\mathrm{or}\ e_2\ \mathrm{x}))\end{aligned}$$

Though no need to write the sugar like that, something wrong may happen without hygienic rewriting system ((if x x x) appears). But with the step in normalization of IFA introduced in 4.2, we can easily get the following rules, which will behave as it should be in resugaring. The reason is that we have replaced x with expression when constructing normal IFA of or, the binding of x will not cause conflicts.

$$\frac{e_1\ \rightarrow\ e_1'}{(\mathrm{Hygienicor}\ e_1\ e_2) \rightarrow (\mathrm{Hygienicor}\ e_1'\ e_2)} \qquad \frac{e_2\ \rightarrow\ e_2'}{(\mathrm{Hygienicor}\ v_1\ e_2) \rightarrow (\mathrm{Hygienicor}\ v_1\ e_2')}$$

$$(\mathrm{Hygienicor}\ v_1\ \#t) \rightarrow \#t \qquad (\mathrm{Hygienicor}\ v_1\ \#f) \rightarrow v_1$$

Overall, our results show lazy desugaring is a good way to handle hygienic sugars in any systems.

#### 5.1.3 *Recursive Sugars.*

Recursive sugar is a kind of syntactic sugars where call itself or each other during the expanding. For example,

$$\begin{array}{ll}(\text{Odd e}) & \rightarrow_d \ (\text{if } (> e\ 0)\ (\text{Even } (\text{-}\ e\ 1))\ \#\text{f}) \\ (\text{Even e}) & \rightarrow_d \ (\text{if } (> e\ 0)\ (\text{Odd } (\text{-}\ e\ 1))\ \#\text{t})\end{array}$$

are common recursive sugars. The traditional resugaring approach can't process syntactic sugar written like this (non-pattern-based) easily, because boundary conditions are in the sugar itself.

Take (Odd 2) as an example. The previous work will firstly desugar the expression using the rewriting system. Then the rewriting system will never terminate as following shows.

```
  (Odd 2)
⇢ (if (> 2 0) (Even (- 2 1) #f))
⇢ (if (> (- 2 1) 0) (Odd (- (- 2 1) 1) #t))
⇢ (if (> (- (- 2 1) 1) 0) (Even (- (- (- 2 1) 1) 1) #f))
⇢ ...
```

Then the advantage of our approach is embodied. Our lightweight approach doesn't require a whole expanding of sugar expression, which gives the framework chances to judge boundary conditions in sugars themselves, and showing more intermediate sequences. We get the resugaring sequences as Figure 18d of the former example using our tool.

We also construct some higher-order syntactic sugars and test them. The higher-order feature is important for constructing practical syntactic sugars. And many higher-order sugars should be constructed by recursive definition. The first sugar is Filter, implemented by pattern matching term rewriting.

$$\begin{array}{l}(\text{Filter } e\ (\text{list } v_1\ v_2\ ...))\ \rightarrow_d \\ \quad (\text{let f } e\ (\text{if } (\text{f } v_1)\ (\text{cons } v_1\ (\text{Filter f } (\text{list } v_2\ ...)))\ (\text{Filter f } (\text{list } v_2\ ...)))) \\ (\text{Filter } e\ (\text{list}))\ \rightarrow_d\ (\text{list})\end{array}$$

and getting resugaring sequences as Figure 18f. Here, although the sugar can be processed by traditional resugaring approach, it will be redundant. The reason is that, a Filter for a list of length $n$ will match to find possible resugaring $n*(n-1)/2$ times. Thus, lazy desugaring is really important to reduce the resugaring complexity of recursive sugar.

Moreover, just like the *Odd and Even* sugar above, there are some simple rewriting systems which do not allow pattern-based rewriting. Or there are some sugars that need to be expressed by the expressions in core language as rewriting conditions. Take the example of another higher-order sugar Map as an example, and get resugaring sequences as Figure 18e.

$$\begin{array}{l}(\text{Map } e_1\ e_2)\ \rightarrow_d \\ \quad (\text{let f } e_1\ (\text{let x } e_2\ (\text{if } (\text{empty? x})\ (\text{list})\ (\text{cons } (\text{f } (\text{first x}))\ (\text{Map f } (\text{rest x}))))))\end{array}$$

Note that the let expression is to limit the subexpression only appears once in RHS. In this example, we can find that the list (cons 1 (list 2)), though equal to (list 1 2), is represented by core language's expression. So it will be difficult to handle such inline boundary conditions for traditional rewriting systems. But our approach is easy to handle cases like this. So our resugaring approach by lazy desugaring is powerful.

## 6 RELATED WORK

As discussed many times before, our work is much related to the pioneering work of *resugaring* in [**??**]. The idea of "tagging" and "reverse desugaring" is a clear explanation of "resugaring", but it becomes very complex when the RHS of the desugaring rule becomes complex. Our approach does not need to reverse desugaring, and is more lightweight, powerful, and efficient. For hygienic resugaring, compared with the approach of using DAG to solve the variable binding problem in

[?], our approach of ”lazy desugaring” is more natural, because it behaves as what the sugar ought to express.

Our work on evaluation rule derivation for the surface language was inspired by the work of *Type resugaring* [?], which shows that it is possible to automatically construct the typing rules for the surface language by statically analyzing type inference trees. But the method based on unification is hard to be applied for our evaluation rule derivation, because it has an assumption that syntactic sugar always has a unique type. But in our problem, syntactic sugar may evaluate different subexpressions under different conditions. This will make it difficult for us to express the rules by inference trees.

*Macros as multi-stage computations* [?] is a work related to our lazy expansion for sugars. Some other researches [?] about multi-stage programming [?] indicate that it is useful for implementing domain-specific languages. However, multi-stage programming is a metaprogramming method, which mainly works for run-time code generation and optimization. In contrast, our lazy resugaring approach treats sugars as part of a mixed language, rather than separate them by staging. Moreover, the lazy desugaring gives us chance to derive evaluation rules of sugars, which is a new good point compared to multi-stage programming.

Our work is related to the *Galois slicing for imperative functional programs* [?], a work for dynamic analyzing functional programs during execution. The forward component of the Galois connection maps a partial input $x$ to the greatest partial output $y$ that can be computed from $x$; the backward component of the Galois connection maps a partial output $y$ to the least partial input $x$ from which we can compute $y$. This can also be considered as a bidirectional transformation [??] and the round-tripping between desugaring and resugaring in traditional approach. In contrast to these work, our resugaring approach is basically unidirectional, with a local bidirectional step for a one-step try in our lazy desugaring. It should be noted that Galois slicing may be useful to handle side effects in resugaring in the future (for example, slicing the part where side effects appear).

Our method for deriving evaluation rule is influenced by work on *origin tracking* [?], which is to track the origins of terms in rewriting systems. For desugaring rules in a good form as given in Section 4, we can easily track the terms and make use of them for derivation of evaluation rules.

Our implementation is built upon the PLT Redex [?], a semantics engineering tool, but it is possible to implement our approach on other semantics engineering tools such as those in [??] which aim to test or verify the semantics of languages. The methods of these researches can be easily combined with our approach to implementing more general rule derivation. *Ziggurat* [?] is a semantic-extension framework, also allowing defining new macros with semantics based on existing terms in a language. It is should be useful for static analysis of macros.

## 7 CONCLUSION

In this paper, we purpose an efficient, powerful, and lightweight resugaring approach by lazy desugaring. In our basic resugaring algorithm, the most important part is the reduction definition for the mixed language (Section 3.2), which decides whether it should reduce the subexpression or desugar the outermost sugar. To gain more efficiency, we enhance the basic algorithm with automatic derivation of evaluation rules for syntactic sugars. In fact, reducing subexpressions corresponds to deriving context rules, while desugaring the outermost sugar corresponds to deriving reduction rules.

As for the future work, we found side effect is troublesome to handle in resugaring, because once a side effect is taken in RHS of a desugaring rule, the sugar cannot be easily resugared according to *emulation* property. We need to find a gentler way to handle sugars with side effects. In addition, we found the derivation of evaluation rules can be a general method to be used on other application (e.g., implementing DSL), but the assumptions of IFA in our setting limit its power. We are working on relaxing the assumption to get a more powerful derivation method.