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 terms 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 [Landin 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 [Culpepper et al. 2019; Felleisen et al. 2018]. 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 [Pombrio and Krishnamurthi 2014, 2015]. 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 [Pombrio and Krishnamurthi 2014].

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

- *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 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) [Pombrio and Krishnamurthi 2015].
- *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 terms 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 of the syntactic sugars based on the syntactic sugar definition and evaluation rules (consist of the context rules and reductions rules) of the core language.

Our main technical contributions can be summarized as follow.

- 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 terms 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 [Pombrio and Krishnamurthi 2014, 2015], 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 simple

```
    (and (or #t #f) (and #f #t))
⟶ (and #t (and #f #t))
⟶ (and #f #t)
⟶ #f
```

Fig. 1. A Typical Resugaring Example

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

```
e  ::=  (if e e e)   // if construct
    |   #t           // true value
    |   #f           // false value
```

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

$$\frac{e \;\rightarrow\; e'}{(\texttt{if}\ e\ e_1\ e_2) \;\rightarrow\; (\texttt{if}\ e'\ e_1\ e_2)}$$

and two reduction rules:

$$(\texttt{if}\ \#t\ e_1\ e_2) \rightarrow e_1 \qquad (\texttt{if}\ \#f\ e_1\ e_2) \rightarrow e_2$$

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

$$(\texttt{and}\ e_1\ e_2) \;\rightarrow_d\; (\texttt{if}\ e_1\ e_2\ \#f)$$
$$(\texttt{or}\ e_1\ e_2) \;\rightarrow_d\; (\texttt{if}\ e_1\ \#t\ e_2)$$

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.

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

-->   { fully desugaring and tagging }
    (if-andtag (if-ortag #t #t #f) (if-andtag #f #t #f) #f)

⟶   { context rule of if, reduction rule of if-false}
    (if-andtag #t (if-andtag #f #t #f) #f)

⟶   { emit (and #t (and #f #t)) by reverse desugaring, reduction rule of if-true}
    (if-andtag #f #t #f) //check resugarable

⟶   { emit (and #f #t) by reverse desugaring, reduction rule of it-false}
     #f
```

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)))

-->     { a one-step try on the outermost and }
      (try (if (or #t #f) (and #f #t) #f))

⟶     { should reduce on subexpression (or #t #f) of and, delay desugaring of and}
      (and (resugar (or #t #f)) (and #f #t)) // no reduction

-->     { a one-step try on or }
      (and (try (if #t #t #f)) (and #f #t))

⟶     { keep this try, finish inner resugaring, and return to the top }
      (resugar (and #t (and #f #t)))

-->     { a one-step try on the outermost and }
      (try (if #t (and #f #t)))

⟶     { keep this try, finish inner resugaring, and return to the top }
      (resugar (and #f #t))

-->     { a one-step try on the outermost and }
      (try (if #f #t #f)) // have to desugar and reduce

⟶ #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*.

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

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

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

CoreExp	::=	x	variable
	|	c	constant
	|	(CoreHead $\texttt{CoreExp}_1$ ... $\texttt{CoreExp}_n$)	constructor
SurfExp	::=	x	variable
	|	c	constant
	|	(SurfHead $\texttt{SurfExp}_1$ ... $\texttt{SurfExp}_n$)	sugar expression

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 5.2.3).
- *Lightweight.* Our approach can be cheaply implemented using the PLT Redex tool [Felleisen et al. 2009]. 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.

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

```
CoreExp  ::=  (CoreExp CoreExp ...) // apply
          |   (lambda (x ...) CoreExp) // call-by-value
          |   (lambdaN (x ...) CoreExp) // call-by-need
          |   (if CoreExp CoreExp CoreExp)
          |   (let x CoreExp CoreExp)
          |   (first CoreExp)
          |   (empty? CoreExp)
          |   (rest CoreExp)
          |   (cons CoreExp CoreExp)
          |   (arithop CoreExp CoreExp) // +, -, *, /, >, <, =
          |   x
          |   c // boolean, number and list
```

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:

$$(\texttt{SurfHead}\ e_1\ e_2\ \ldots\ e_n)\ \rightarrow_d\ \texttt{exp}$$

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 and by

$$(\texttt{and}\ e_1\ e_2)\ \rightarrow_d\ (\texttt{if}\ e_1\ e_2\ \#f).$$

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 `let` binding may be used (a normal way in syntactic sugar). We take the following sugar as an example

$$(\texttt{Twice}\ e_1)\ \rightarrow_d\ (+\ e_1\ e_1).$$

If we execute `(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 `let` binding to resolve this problem.

Note that in the desugaring rule, we do not restrict the RHS to be a `CoreExp`. We can use `SurfExp` (more precisely, we allow the mixture use of syntactic sugars and core expressions) to define recursive syntactic sugars, as seen in the following example.

$$(\texttt{Odd}\ e)\ \rightarrow_d\ (\texttt{if}\ (>\ e\ 0)\ (\texttt{Even}\ (-\ e\ 1))\ \#f)$$
$$(\texttt{Even}\ e)\ \rightarrow_d\ (\texttt{if}\ (>\ e\ 0)\ (\texttt{Odd}\ (-\ e\ 1))\ \#t)$$

```
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

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

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

then if e_1 is a 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.

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

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

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

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

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 $\rightarrow_m$.

```
resugar(e)  =  if isNormal(e) then return
               else
                    let e →m e' in
                    if e' ∈ DisplayableExp
                         output(e'), resugar(e')
                    else resugar(e')
```

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 5.1).

3.3 Correctness

Following the traditional resugaring approach [Pombrio and Krishnamurthi 2014, 2015], we show that our resugaring algorithm 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.

Lemma 3.1 (Emulation). *For* `exp` = (`SurfHead` $e_1 \ \ldots \ e_i \ \ldots \ e_n$) ∈ `SurfExp`, *if* `exp` $\rightarrow_m$ `exp'` *and* `fulldesugar(exp)` ≠ `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 the exact subexpression is to be reduced. 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 $e_i \rightarrow_m e_i''$. Because lemma 3.2 says 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) ≠ `fulldesugar`(e_i''), non-desugaring reduction occurs at the subexpression, then what we need to prove is that, `fulldesugar(exp)` $\rightarrow_c$ `fulldesugar(exp')`, 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, it is the emulation of $\rightarrow_m$ on the subexpression. 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, `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. This 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 define the abstraction by catalog 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 is easy.

One may ask what if we just want to display core language's expression originated from the input expression. 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 $(\texttt{not}\ e_1) \rightarrow_d (\texttt{if}\ e_1\ \texttt{\#f}\ \texttt{\#t})$. 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 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

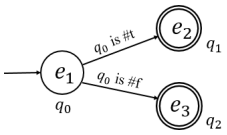


Fig. 5. IFA of `if`

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 a term `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 evalute depending on the value of e_1. The evaluation result of e_2 or e_3 will be the value of the term. 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 term needs to be evaluated, and the nodes before this 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 is the result of the term with this syntactic structure. In most cases, the transition condition is the evaluation result (an abstract value) of the previous node or a specific value. To simplify the representation of IFA in the figures of this paper, in the former case, we omit the condition on the transition edge; and in the latter case, we only mark the value on the transition edge.

When a term headed 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 (`#f`), the IFA selects the branch (e_3). Then the result of e_3 (`#t`) is the evaluation result of the term.

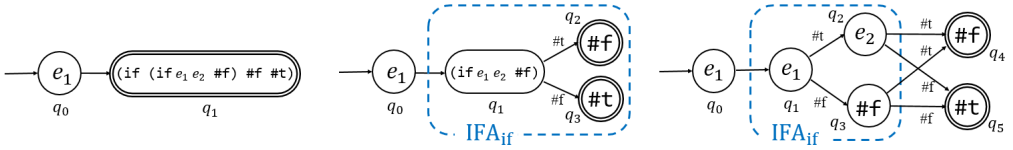
□

Example 4.2 (IFA of nand*).* Sometimes the rules may be more complex, such as being reduced into another syntactic structure, or a term containing other syntactic structures. For example, we can express nand's evaluation rules as follows. Based on the method discussed above, we can draw nand's IFA as Figure 6a.

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

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

When the automaton runs into the terminal node of Figure 6a, it derives the `if` term. In fact, we have known how `if` works through the IFA of `if`. Thus we can replace the last node with an $\text{IFA}_{\texttt{if}}$ and substitute e_1 to e_3 of $\text{IFA}_{\texttt{if}}$ with the parameters of the node. Use the termination nodes of $\text{IFA}_{\texttt{if}}$ as the termination nodes of new $\text{IFA}_{\mathsf{nand}}$. The results are shown in Figure 6b. Further decomposing



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

Fig. 6. IFA of nand

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

Fig. 7. IFA of and

the inner `if` node, connecting the terminating nodes of IFA_{nand} to the nodes pointed to by the original output edge, we get Figure 6c.

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

□

Example 4.3 (IFA of and*).* We represent the evaluation rule of and in a more complex way, as follows.

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

In this case, we use the let binding. So we need to record the term represented by each variable at each node called symbol table. The representation of IFA_{and} is shown in Figure 7a.

Further discuss the syntactic structure. We first evaluate e_1 and save the result in x. When evaluating the term of `if`, what is actually evaluated is $(\texttt{if}\ x\ e_2\ x)[e_1/x]$. We represent it in the form of Figure 7b, where node q_1 contains a symbol table for recording variables and the nodes referred to. It is similar to context, but since we mapped the variable to the node, we distinguished it. Mapping to nodes is to ensure that the variable data will not be lost when the IFA is transformed.

□

Defination 4.1 (Inference Automaton). *An inference automaton (IFA) of syntactic structure* `CoreHead` *is a 5-tuple,* $(Q, \Sigma, q_0, F, \delta)$*, consisting of*

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

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

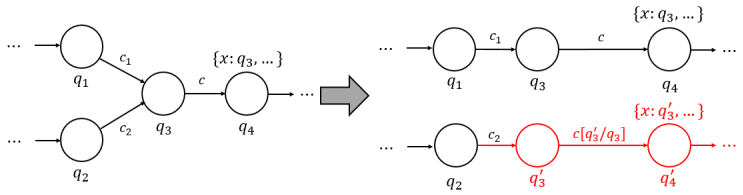


Fig. 8. Transform into a Tree

The last constraint requires that there be no circles in our IFA.

In IFA, state transition does not depend on input. The only input of IFA is the term to be evaluated with this syntactic structure. The state transition is through whether the term meets the condition. Note that IFA is associated with syntactic structure. At Each IFA only represents the current evaluation of a syntactic structure. The state indicates that some sub-expressions of the syntactic structure have been evaluated, and the rest have not.

4.2 Normal IFA

Although IFA can intuitively show the behavior of a syntactic structure for evaluation, IFA itself has a complicated form. For example, the IFA of a syntactic structure may contain other syntactic structures as shown in Figure 6a. We try to deform and simplify IFA to make it easier to analyze.

Defination 4.2 (Normal IFA). *If an IFA meets following constraints, we call it a normal IFA.*

- *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 there are no more composite syntactic structures in it. And it is a tree structure. We prove that 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.*

In order to prove this theorem, we give the following steps and prove their correctness.

4.2.1 Transform into a Tree. When evaluating a term, different branches do not influence each other. Therefore, a node with at least two sources can be cloned and applied to different branches as shown in Figure 8. Replace the node referenced by the branch with the cloned node in conditions and symbol tables. The correctness is obvious. In this way, we can transform an IFA into a tree so that it satisfies the second constraint of normal IFA.

4.2.2 Substitute Sub-Syntactic Structure. If the node q in the tree IFA contains a term of a syntactic structure 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. Connect all the termination nodes of H to the original output node q', and then 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 can guarantee correctness, for $(\mathsf{H}\ e_1\ \ldots\ e_n)[e/x] = (\mathsf{H}\ e_1[e/x]\ \ldots\ e_n[e/x])$.

In particular, if we guarantee that a term 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.

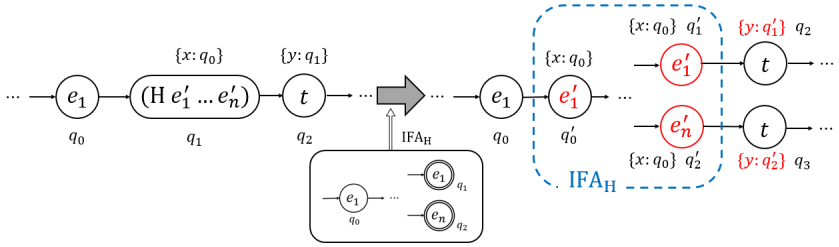


Fig. 9. Substitute Sub-Syntactic Structure

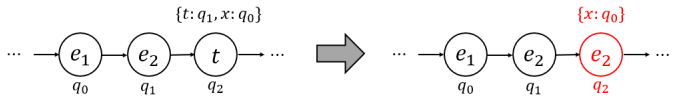


Fig. 10. Replace Variables in the Symbol Table

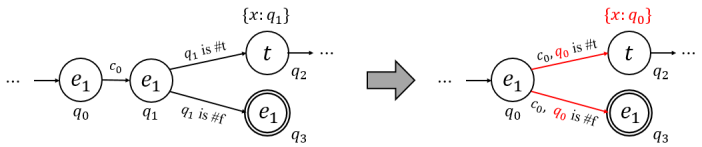


Fig. 11. Remove Evaluated Nodes and Merge Transition Conditions

For hygienic sugar, this step of substitution can also guarantee correctness. 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 syntactic structure.

4.2.3 Replace 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 Figure 10. Because $x[e_1/x, e_2/y] = e_1[e_2/y]$, it is correct.

4.2.4 Remove 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 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.

4.2.5 Remove 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.

4.2.6 Normalizability of IFA.

Proof of Theorem 4.1. By repeating the above steps continuously, we can always get normal IFA. And it is equivalent to the original IFA for the correctness of each step. □

4.3 Convert evaluation rules to IFA

In the examples at the beginning of this section, we construct IFAs based on their semantics. 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.

4.3.1 *Assumptions.*

Assumption 1. *A syntactic structure CoreHead only contains the following evaluation rules.*

$$\frac{e_i \longrightarrow e_i' \quad T}{(\texttt{CoreHead}\ e_1 \ldots e_i \ldots e_n) \longrightarrow (\texttt{CoreHead}\ e_1 \ldots e_i' \ldots e_n)}$$

$$\frac{T}{(\texttt{CoreHead}\ v_1 \ldots v_p\ e_1 \ldots e_q) \longrightarrow e}$$

where e can be any value, one of the parameters or a term of another syntactic structure. T is a set of constraints of the parameters containing e_j is a value or e_j is a specific value where $j \in 1, 2, \ldots, n$.

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 $(\texttt{if\ \#t}\ e_1\ e_2) \longrightarrow e_1$ can be seen as

$$\frac{e\ \textbf{is}\ \texttt{\#t}}{\texttt{if}\ e\ e_1\ e_2 \longrightarrow e_1.}$$

Assumption 2 (Orderliness of Syntactic Structure). *The syntactic structure in CoreLang is finite. Think of all syntactic structures as points in a directed graph. If one of CoreHead's evaluation rules can generate a term containing CoreHead′, then construct an edge that points from CoreHead to CoreHead′. 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 *CoreHead*, IFA of *CoreHead′* is known.

Assumption 3 (Determinacy of One-Step Evaluation). *The 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 a term* $(\texttt{CoreHead}\ e_1\ \ldots\ e_n)$ *does a one-step evaluation by rule (E-Head) of* CoreHead*, 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. □

4.3.2 *Algorithm.*

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

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 syntactic structure `CoreHead` 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 term e as the expression of the node. Symbol tables of these nodes are set to be empty. Next we connect these nodes.

For a term like (`CoreHead` e_1 ... e_n), considering that $e_1 \cdots e_n$ are not value, According to Lemma 4.1, we have the unique rule r of `CoreHead` for one-step evaluation. Let node q corresponding to r be q_0.

If r is a context rule for e_i, let the term of q be e_i. Assume that the evaluation of e_i results in v_i, we get term (`CoreHead` $e_1 \ldots e_{i-1}\ v_i\ e_{i+1} \ldots e_n$). 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 `CoreHead`. According to the lemma 4.1, we can also get a normal IFA of `CoreHead`.

4.3.3 Example: Construct 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.

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

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

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

$$(\texttt{xor}\ \#\texttt{f}\ \#\texttt{t}) \longrightarrow \#\texttt{t} \quad \text{(E-XorFalseTrue)}$$

$$(\texttt{xor}\ \#\texttt{f}\ \#\texttt{f}) \longrightarrow \#\texttt{f} \quad \text{(E-XorFalseFalse)}$$

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

Considering a term (`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 term will be (`xor #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.

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

Lemma 4.2. *If a term e of syntactic structure* `CoreHead` *uses rule r to derive, we can find a path from q_0 to q generated by r.*

Proof of Lemma 4.2. Suppose e = (`CoreHead` v_1 ... v_p e_1 ... e_q). We build a term e_0' = (`CoreHead` `id`(v_1) ... `id`(v_p) e_1 ... e_q), where `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' = (`CoreHead` v_1 `id`(v_2) ... `id`(v_p) e_1 ... e_q) after derived by r_1. Similarly, we find

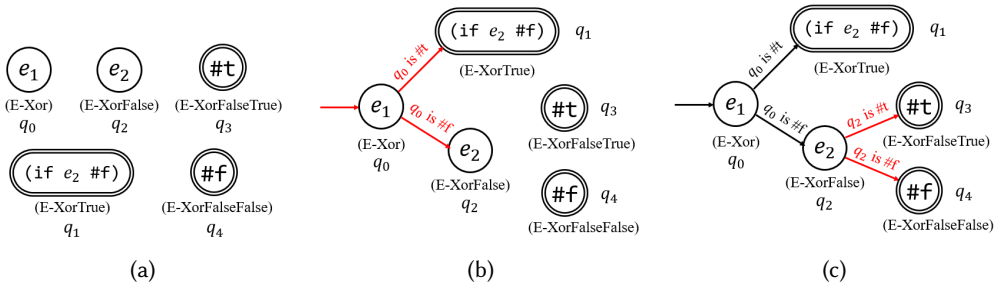


Fig. 12. IFA of xor: Constructed by evaluation rules

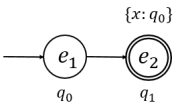


Fig. 13. IFA of `let`

a rule r_2 which is a context rule for e_2. The node q_2 generated by r_2 satisfies $\delta(q_1, (e_1 \textbf{ is } v_1)) = q_2$. By analogy, we can get $e'_p = (\texttt{CoreHead}\ v_1\ \ldots\ v_p\ e_1\ \ldots\ e_q) = e$ and a path $q_1(= q_0), q_2, \ldots, q_p$. At last, we use rule r to derive e and add q to the path. □

Lemma 4.3. *For any syntactic structure* `CoreHead`*, if its evaluation rules meets the above assumptions, the normal IFA got by the algorithm in Theorem 4.2 has the same semantics as the rules.*

In other words, for any term of syntactic structure `CoreHead`, evaluating the term by IFA and evaluating by rule get the same derivation sequence.

Proof of Lemma 4.3. We only need to discuss that in a one-step derivation, both get the same result.

Considering a term $e = (\texttt{CoreHead}\ e_1 \ldots e_n)$ use rule r to derive. Suppose r generate node q. By Lemma 4.2, we find a path from q_0 to q. The term e must meet the conditions from q_0 to q. Therefore, the one-step derivation of this term 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 of them are e'.

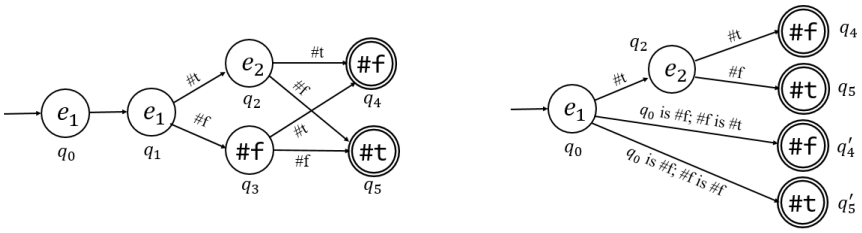
Similarly, if an term can be derived in one step in IFA, then it must be able to use the corresponding rule for one-step derivation. □

4.3.5 IFA of `let`. If a certain syntactic structure does not meet the above assumptions, it does not mean that this syntactic structure 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.

$$\frac{e_1 \longrightarrow e'_1}{\texttt{let}\ x\ e_1\ e_2 \longrightarrow \texttt{let}\ x\ e'_1\ e_2}$$

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

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 Convert IFA to Evaluation Rules

Next we try to convert IFA to evaluation rules. Because IFA can be converted into normal IFA 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 also need to have stricter constraints on IFA to ensure that evaluation rules can be 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 very strong assumption, which requires that only termination nodes could have substitution. Because it is difficult for us to generate a context rule for the term after 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 of Theorem 4.3. Suppose that the IFA stands for the syntactic structure H, then we build evaluation rules for H. First traverse all nodes to find the set of all terms in nodes, which is the parameters of the syntactic structure H like $(\mathsf{H}\ e_1 \dots e_n)$. 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 $\{x : q_x; y : q_y; \dots\}$. Let $e_x, e_y, \dots$ be the expressions of $q_x, q_y, \dots$. Add a reduction rule like

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

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

$$\frac{e_i \longrightarrow e_i' \quad T}{(\mathsf{H}\ e_1\ \dots\ e_i\ \dots\ e_n) \longrightarrow (\mathsf{H}\ e_1\ \dots\ e_i'\ \dots\ e_n)}$$

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. □

4.4.2 Example: 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

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

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.

$$\frac{e_2 \longrightarrow e_2' \quad e_1 \textbf{ is } \mathtt{\#t}}{(\mathsf{nand}\ e_1\ e_2) \longrightarrow (\mathsf{nand}\ e_1\ e_2')}$$

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.

$$\frac{e_1 \textbf{ is } \mathtt{\#t} \quad e_2 \textbf{ is } \mathtt{\#t}}{(\mathsf{nand}\ e_1\ e_2) \longrightarrow \mathtt{\#f}} \qquad \frac{e_1 \textbf{ is } \mathtt{\#t} \quad e_2 \textbf{ is } \mathtt{\#f}}{(\mathsf{nand}\ e_1\ e_2) \longrightarrow \mathtt{\#t}}$$

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.

$$\frac{e_1 \textbf{ is } \mathtt{\#f} \quad \mathtt{\#f} \textbf{ is } \mathtt{\#t}}{(\mathsf{nand}\ e_1\ e_2) \longrightarrow \mathtt{\#f}} \qquad \frac{e_1 \textbf{ is } \mathtt{\#f} \quad \mathtt{\#f} \textbf{ is } \mathtt{\#f}}{(\mathsf{nand}\ e_1\ e_2) \longrightarrow \mathtt{\#t}}$$

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.

$$\frac{e_1 \longrightarrow e_1'}{(\mathsf{nand}\ e_1\ e_2) \longrightarrow (\mathsf{nand}\ e_1'\ e_2)} \qquad \frac{e_2 \longrightarrow e_2'}{(\mathsf{nand}\ \mathtt{\#t}\ e_2) \longrightarrow (\mathsf{nand}\ \mathtt{\#t}\ e_2')}$$

$$(\mathsf{nand}\ \mathtt{\#t}\ \mathtt{\#t}) \longrightarrow \mathtt{\#f} \quad (\mathsf{nand}\ \mathtt{\#t}\ \mathtt{\#f}) \longrightarrow \mathtt{\#t} \quad (\mathsf{nand}\ \mathtt{\#f}\ e_2) \longrightarrow \mathtt{\#t}$$

4.4.3 *Correctness.*

Lemma 4.4. *For any syntactic structure* 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.*

In other words, for any term of syntactic structure H, evaluating the term by IFA and evaluating by rule get the same derivation sequence.

Proof of Lemma 4.4. We only need to discuss that in a one-step derivation, both get the same result.

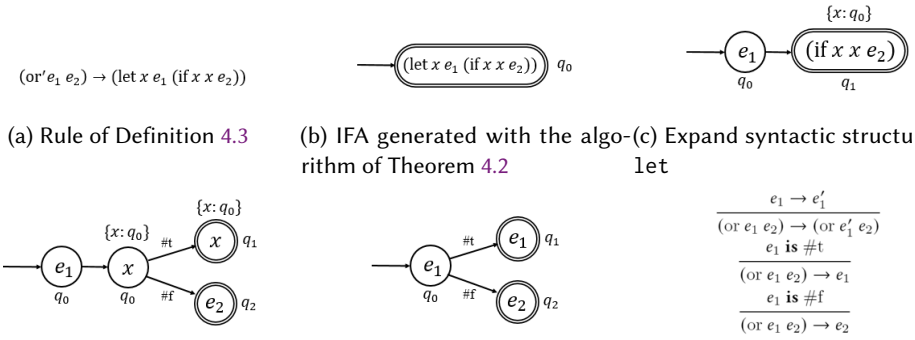
Considering a term $e = (\mathsf{H}\ e_1 \ldots e_n)$ use rule r to derive. The term e must meet the condition T of r such as some parameters must be value or a specific value. Suppose r is generated by node q. Then T is the set of all transition conditions from q_0 to q. Therefore, the one-step derivation of this term 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 term can be derived in one step in IFA, then it must be able to use the corresponding rule for one-step derivation. □

4.5 Syntactic Sugar

We can find that although rules of nand we used when constructing the IFA included the syntactic structure of `if`, the final result does not. For syntactic sugar, we also deal with it with a similar idea, that is, construct the IFA of syntactic sugar, simplify it and convert it into rules. With the IFA, we can easily get the evaluation rules for syntactic sugars.

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



(a) Rule of Definition 4.3 (b) IFA generated with the algorithm of Theorem 4.2 (c) Expand syntactic structure of `let`

(d) Expand syntactic structure 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`

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

Defination 4.3. *Considering the following syntactic sugar*

$$(\texttt{SurfHead}\ x_1\ \ldots\ x_n)\ \rightarrow_d\ e,$$

the IFA of `SurfHead` *is defined as the IFA of syntactic structure* `SurfHead'` *whose evaluation rule is*

$$(\texttt{SurfHead'}\ x_1\ \ldots\ x_n) \rightarrow e$$

4.5.1 An Example of Syntactic Sugar. Suppose that there are only `if` and `let` in our CoreLang, whose IFAs are known. Now we build rules for syntactic sugar `or` and `sg`.

$$(\texttt{or}\ e_1\ e_2)\ \rightarrow_d\ (\texttt{let}\ x\ e_1\ (\texttt{if}\ x\ x\ e_2))$$

`or` syntactic sugar only uses the syntax structure of CoreLang, which meets Assumption 5. Then we 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.

4.5.2 Correctness. 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 syntactic structure 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 syntactic structure is reduced to another syntactic structure. 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.

5 IMPLEMENTATION AND CASE STUDIES

5.1 Implementation

Our resugaring approach is implemented using PLT Redex[Felleisen et al. 2009], which is a semantic engineering tool based on reduction semantics[Felleisen and Hieb 1992]. The framework of the implementation is as Figure 16. In the language model, desugaring rules are written as reduction

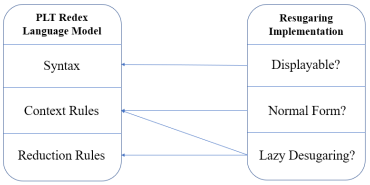


Fig. 16. Framework of Implementation

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

(a) Subsequences of Sugar2

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

(b) Optimized Resugaring of Sugar1

Fig. 17. Recursive Optimization's Example

rules of SurfExp. And context rules of SurfExp have no restriction (every subexpression is reducible as a hole). Then for each resugaring step, we should choose the exact reduction which satisfies the reduction of mixed language's reduction rule in Section 3.2.

Note that in SurfRed1 rule and CoreExt1 rule, there is a recursive call on $\rightarrow_m$. We can optimize the resugaring algorithm by recursively resugaring. 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 sequences 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.

5.2 Case Studies

We test some applications on the tool as case studies. 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 need to output some intermediate sequences including lambda expressions in some examples. It's easy if we want to skip them.

5.2.1 Simple sugar. We construct some simple syntactic sugars and try it on our tool. Some sugar is inspired by the first work of resugaring [Pombrio and Krishnamurthi 2014]. The result shows that our approach can handle all sugar features of their first work. Take an SKI combinator syntactic sugar as an example.

```
S →d (lambdaN (x1 x2 x3) (x1 x2 (x1 x3)))
K →d (lambdaN (x1 x2) x1)
I →d (lambdaN (x) x)
```

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

```
    (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

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?

$$\texttt{(or } e_1 \ e_2\texttt{)} \rightarrow_d \texttt{(let x } e_1 \texttt{ (if x x } e_2\texttt{))}$$

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

$$\frac{e_1 \ \longrightarrow \ e_1'}{(\text{or } e_1 \ e_2) \longrightarrow (\text{or } e_1' \ e_2)} \qquad (\text{or } \#t \ e2) \longrightarrow \#t \quad (\text{or } \#f \ e2) \longrightarrow e_2$$

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 makes our approach efficient, because no attempts for resugaring the expression are needed.

5.2.2 Hygienic sugar. The second work [Pombrio and Krishnamurthi 2015] 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.

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

For traditional resugaring approach, if we want to get sequences of `(let x 2 (Hygienicadd 1 x))`, it will firstly desugar to `(let x 2 (let x 1 (+ x x)))`, which is awful because the two x in `(+ x x)` should be bind to different value. 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 (does not rewrite variables' name during rewriting).

The lazy desugaring is also convenient for achieving hygiene for non-hygienic rewriting. For example, `(let x 1 (+ x (let x 2 (+ x 1))))` may be reduced to `(+ 1 (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,

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

and making some simple modifies (see in appendix) in the reduction of mixed language, we will get the resugaring sequences as Figure 18c in our tool.

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

$$\texttt{(Or } e_1\ e_2\texttt{)} \rightarrow_d \texttt{(let x } e_1\ \texttt{(if x x } e_2\texttt{))}$$
$$\texttt{(Hygienicor } e_1\ e_2\texttt{)} \rightarrow_d \texttt{(let x } e_1\ \texttt{(or } e_2\ \texttt{x))}$$

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 4.2.2, 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.

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

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

Overall, our result shows lazy desugaring is really a good way to handle hygienic sugar in any systems.

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

$$\texttt{(Odd e)} \rightarrow_d \texttt{(if (> } e\ \texttt{0) (Even (- } e\ \texttt{1)) \#f)}$$
$$\texttt{(Even e)} \rightarrow_d \texttt{(if (> } e\ \texttt{0) (Odd (- } e\ \texttt{1)) \#t)}$$

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.

```
(Filter e (list v1 v2 ...)) →d
    (if (e v1) (cons v1 (Filter e (list v2 ...))) (Filter e (list v2 ...)))
(Filter e (list)) →d (list)
```

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 terms 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.

```
(Map e1 e2) →d
(let x e2 (if (empty? x) (list) (cons (e1 (first x)) (Map e1 (rest x)))))
```

Note that the `let` term 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 term. 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 [Pombrio and Krishnamurthi 2014, 2015]. 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 [Pombrio and Krishnamurthi 2015], 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* [Pombrio and Krishnamurthi 2018], 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 inferring tree.

Macros as multi-stage computations [Ganz et al. 2001] is a work related to our lazy expansion for sugars. Some other researches [Rompf and Odersky 2010] about multi-stage programming [Taha 2003] 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* [Ricciotti et al. 2017], 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 [Czarnecki et al. 2009; Foster et al. 2007] 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* [Deursen et al. 1992], 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 [Felleisen et al. 2009], a semantics engineering tool, but it is possible to implement our approach on other semantics engineering tools such as those in [Rosu and Serbanuta 2010; Vergu et al. 2015] 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* [Fisher and Shivers 2006] 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. Essentially, we would see the derivation of evaluation rules is the abstract of the basic resugaring approach. In the basic resugaring approach, the most important part is *reduction in mixed language* (see in sec 3.2), which decides whether reducing the subexpression or desugaring the outermost sugar. Reducing subexpressions are just the same as derivate context rules; desugaring the outermost sugar is similar to derivate reduction rules. The derivate evaluation rules, if can be got, is more convenient and efficient than the basic resugaring, because the derivation evolves a process like abstract interpretation [Cousot and Cousot 1977], then reduces many steps executed in core language. Moreover, the semantics got by derivation make it possible to do some optimization at the surface language level, which is important for implementing a DSL. However, the derivation method has more restrictions than our basic resugaring approach, so not powerful enough to handle all cases that can be processed by the resugaring. So we use the derivate rules as a shortcut for our basic resugaring.

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 syntactic sugar, the sugar can not be 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 constraints of IFA in our setting limit the expressiveness. So we may achieve a more powerful derivation as future work.

REFERENCES

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Ryan Culpepper, Matthias Felleisen, Matthew Flatt, and Shriram Krishnamurthi. 2019. From Macros to DSLs: The Evolution of Racket. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 136. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:19.

Krzysztof Czarnecki, Nate Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective, Vol. 5563. 260–283. https://doi.org/10.1007/978-3-642-02408-5_19

A. Van Deursen, P. Klint, and F. Tip. 1992. Origin Tracking. *JOURNAL OF SYMBOLIC COMPUTATION* 15 (1992), 523–545.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex* (1st ed.). The MIT Press.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2018. A programmable programming language. *Commun. ACM* 61, 3 (2018), 62–71.

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (Sept. 1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

David Fisher and Olin Shivers. 2006. Static Analysis for Syntax Objects. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. Association for Computing Machinery, New York, NY, USA, 111–121. https://doi.org/10.1145/1159803.1159817

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (May 2007), 17–es.

Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence, Italy) *(ICFP '01)*. Association for Computing Machinery, New York, NY, USA, 74–85. https://doi.org/10.1145/507635.507646

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (01 1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308 arXiv:https://academic.oup.com/comjnl/article-pdf/6/4/308/1067901/6-4-308.pdf

Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: Lifting Evaluation Sequences through Syntactic Sugar. *SIGPLAN Not.* 49, 6 (June 2014), 361–371. https://doi.org/10.1145/2666356.2594319

Justin Pombrio and Shriram Krishnamurthi. 2015. Hygienic Resugaring of Compositional Desugaring. *SIGPLAN Not.* 50, 9 (Aug. 2015), 75–87. https://doi.org/10.1145/2858949.2784755

Justin Pombrio and Shriram Krishnamurthi. 2018. Inferring Type Rules for Syntactic Sugar. *SIGPLAN Not.* 53, 4 (June 2018), 812–825. https://doi.org/10.1145/3296979.3192398

Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. 2017. Imperative Functional Programs That Explain Their Work. *Proc. ACM Program. Lang.* 1, ICFP, Article 14 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110258

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *SIGPLAN Not.* 46, 2 (Oct. 2010), 127–136. https://doi.org/10.1145/1942788.1868314

Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.* 79 (2010), 397–434.

Walid Taha. 2003. A Gentle Introduction to Multi-stage Programming. 30–50.

Vlad Vergu, Pierre Neron, and Eelco Visser. 2015. DynSem: A DSL for Dynamic Semantics Specification. In *26th International Conference on Rewriting Techniques and Applications (RTA 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Maribel Fernández (Ed.), Vol. 36. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 365–378. https://doi.org/10.4230/LIPIcs.RTA.2015.365