Lifting Resugaring by Lazy Desugaring

ANONYMOUS AUTHOR(S)

Syntactic sugars provide an effective way to define domain-specific languages and extend languages. However, the programs after desugaring to a host language would be unrecognizable especially for people who are unfamiliar with the host language. Moreover, the programs lose correspondence with the surface language defined by sugars, which is a leaky of abstraction.

Resugaring is an method to solve the problem above. In this paper, we purposed an approach of resugaring based on lazy desugaring—getting evaluation sequences without fully desugaring the whole syntactic sugar expression. We purposed a automatic derivation method for part of sugar abstracted from the basic resugaring algorithm. They can work together on resugaring tasks. We implement our approach and test on some application. The result show that our resugaring approach is a efficient, powerful and lightweight approach.

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.

- *Tricky to handle recursive sugar.* While tagging is used to remember the position of desugaring so that reverse desugaring can be done at correct position when 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].
- *Complicated to handle hygienic sugar.* For reverse desugaring, we need to match part of the core expression on the RHS of the desugar rule and to get the surface term by substitution. But when a syntactic sugar introduce 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 need to keep checking whether reverse desugaring is applicable during evaluation of desugared expression, which is very costive. Moreover, the match-and-substitute for reverse desugaring is costive particularly when the core term is big.

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

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 become unnecessary. We consider the surface language and the core language as one language, and reduce expressions dynamically either by the reduction rules in the core language or by the desugaring rules for defining syntactic sugars. To gain more efficiency, we can make a shortcut of a sequence of core expression reduction to a one-step reduction of the surface language, by automatically deriving evaluation rules on the surface language from those on the core language.

Our main technical contributions can be summarized as follow. Todo: The following contributions will be revised later.

- *A lightweight, efficient and powerful resugaring approach by lazy desugaring.* The resugaring approach we proposed is based on reductions of a mixed language. It takes surface language and core language as a whole, then decided whether desugaring the sugars or reducing the subexpressions according to properties that make the resugaring correct. Thus, it is lightweight because many match and substitution processes can be omitted. We test the approach on many applications. The result shows that in addition to handle what existing work can handle, our dynamic approach can process non-pattern-based recursive sugar and hygienic sugar easily, which makes it powerful. And the rewriting system based on reduction semantics makes it possible to write syntactic sugar easily.
- *Automatic derivation of syntactic sugar's evaluation rule.* We abstract our resugaring approach to a derivation of syntactic sugar's evaluation rule. The derivation turns core language's evaluation rules into *Inference automaton* (Todo: see ...) . And for each syntactic sugar, we would generate the surface language's evaluation rules without depending on some rules in core language. (some meta-functions may be necessary.) The the resugaring sequences can be easily got from the evaluation rules. Since the derivation can work together with the basic resugaring, our resugaring approach becomes more efficient because many steps in core language can be omitted.
- *Correctness.*

We have implemented lazy desugaring and automatic derivation of reduction rules for syntactic sugars. All the example in this paper have passed the test of the system.

The rest of our paper is organized as follow. 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 reduction rules for syntactic sugars in Section 4. We discuss relative work in Section 5, and conclude the paper in Section 6.

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 core language, defining the `if` expressions:

$$\begin{array}{lll} \texttt{e} & ::= & \texttt{(if e e e)} \\ & | & \#t \\ & | & \#f \end{array}$$

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

$$\frac{\texttt{e} \rightarrow \texttt{e'}}{\texttt{(if e e1 e2)} \rightarrow \texttt{(if e' e1 e2)}} \quad \text{(CONTEXT RULE OF IF)}$$

`(if #t e1 e2)` $\rightarrow$ `e1` (REDUCTION RULE OF IFTRUE)

and one reduction rule to compute if:

`(if #f e1 e2)` $\rightarrow$ `e2` (REDUCTION RULE OF IFFALSE)

Assume that our surface language is defined by two syntactic sugars—*and* sugar and *or* sugar on the core language.

`(and e1 e2)` $\rightarrow_d$ `(if e1 e2 #f)`

`(or e1 e2)` $\rightarrow_d$ `(if e1 #t e2)`

Now let us demonstrate how to execute `(and (or #t #f) (and #f #t))`, and get the following resuaring sequence.

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

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 existing 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 language. Therefore many useless resugarings on subexpressions take place in the traditional approach. Moreover, reverse resugaring would 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 that how the "one-step try" can be cheaply done by derivation of reduction rules for syntactic sugars.

```
        (resugar (and (or #t #f) (and #f #t)))
  -->    { a one-step try on the outmost 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 make one reduction step and move resugaring focus if desugaring is unnecessary. So 7 reduction steps are needed 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 desugar 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 context rules and reductions rules for both *and* sugar and *or*.

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

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

Now with these rules, our resugaring will need only 4 steps.

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

$\longrightarrow$ `#f`

Two remarks are worh making here. First, we do not require a complete set of reduction/context rules for all 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 reduction rules cannot be derived. We can still do resugaring on `(and (hard (and #t #t) ...) ...)`, as we do early. 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, so there is no worry about desugaring an original core expression to a syntactic sugar. For instance, we can deal with `(and (if #t then (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 have ”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 3.5.3).
- *Lightweight.* Our approach can be cheaply implemented using the PLT Redex tool [Felleisen et al. 2009]. This is because our ”lazy” desugaring is much more simpler than ”reverse desugaring” that needs careful design of patterns, and 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 and the core languages, we combine them together as one mixed language, allowing users to freely use 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 necessarily to be resugared by the traditional approach can be correctly produced.

3.1 Mixed Language for Resugaring

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

Fig. 1. Core and Surface Expressions

We will define a mixed language for a given core language and a surface language defined over the core language. An expression in this language will be reduced step by step by the reduction rules for the core language and the desugaring rules for defining the syntactic sugars in the surface language.

3.1.1 Core Language. For our host language, we consider its evaluator as a blackbox but with two natural assumptions. First, there is a deterministic stepper in the evaluator which, given an expression in the host language, can deterministically reduce the expression to a new expression. Second, the evaluation of any sub-expression has no side-effect on other parts of the whole expression.

An expression of the core language is defined in Figure 1. 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 2 to demonstrate our approach. Todo: semantic needed?

```
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. 2. 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 and variable. So an expression of the surface language is some core constructor expressions with sugar expressions, as defined in Figure 1.

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

$$(\mathsf{SurfHead}\ e_1\ e_2\ \ldots\ e_n)\ \rightarrow_d\ \mathsf{Exp}$$

where its LHS is a simple pattern (unnested) and its RHS is an expression of surface language or core language, and any subterms (e.g. e_1) in LHS only appear once in RHS. For instance, we may define syntactic sugar And by

$$(\mathsf{And}\ e_1\ e_2)\ \rightarrow_d\ (\mathtt{if}\ e_1\ e_2\ \#f).$$

Note that if the pattern is nested, we can introduce a new syntactic sugar to flatten it. And if we need a subterm multi times in RHS, a `let` binding is needed (a normal way in syntactic sugar). One may wonder why Todo: don't understand.. not restricting the RHS to be a core expression `CoreExp`, which sounds more natural. We use `surfExp` to be able to allow definition of recursive syntactic sugars, as seen in the following example.

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

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

`Exp`	::=	`DisplayableExp`
	|	`UndisplayableExp`
`DisplayableExp`	::=	`SurfExp`
	|	`CommonExp`
`UndisplayableExp`	::=	`CoreExp'`
	|	`OtherSurfExp`
	|	`OtherCommonExp`
`CoreExp`	::=	`CoreExp'`
	|	`CommonExp`
`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. 3. Our Mixed Language

3.1.3 *Mixed Language.*

Our mixed language for resugaring combines the surface language and the core language. The differences between terms in our core language (CoreLang) and those in our surface language (SurfLang) are identified by their `Head`. But there may be some terms in the core language should be displayed during evaluation, or we need some core terms to help us getting better resugaring sequences . So we defined `CommonExp`, which origin from CoreLang, but can be displayed in resugaring sequences. The `Core'Exp` terms are terms with undisplayable `CoreHead` (named `CoreHead'`). The `SurfExp` terms are terms with `SurfHead` and all subexpressions are displayable. The `CommonExp` terms are terms with displayable CoreLang's `Head` (named `CommonHead`), together with displayable subexpressions. There exists some other expressions during our resugaring process, which have displayable `Head`, but one or more subexpressions should not display. They are `UndisplayableExp`. We distinct the two kinds of expression for *abstraction* property (discussed in Section 3.3.2).

Take some terms in the core language in Figure 2 as examples. We may assume `if`, `let`, λ_N (call-by-name lambda calculus), `empty`, `first`, `rest` as `CoreHead'`, op, λ, `cons` as `CommonHead`. Then we would show some useful intermediate steps.

Note that some expressions with `CoreHead` contains subexpressions with `SurfHead`, they are of `CoreExp` but not in core language, we need a tricky extension for the core language's evaluator. We use $\rightarrow_c$ to donate a reduction step of core language's expression, and $\rightarrow_e$ to donate a step in the extension evaluator for the mixed language. We may use $\rightarrow_m$ to donate one-step reduction in our mixed language, defined in the next section.

$$\frac{\begin{array}{c}\forall\ i.\ e_i \in \texttt{CoreExp} \\ (\texttt{CoreHead}\ e_1\ \ldots\ e_n) \rightarrow_c e'\end{array}}{(\texttt{CoreHead}\ e_1\ \ldots\ e_n) \rightarrow_e e'} \qquad (\textsc{CoreRed})$$

$$\frac{\begin{array}{c}\forall\ i.\ subst_i = (e_i \in \texttt{SurfExp}\ ?\ \texttt{tmpexp}\ :\ e_i),\ \textit{where}\ \texttt{tmpexp}\ \textit{is any reduciable}\ \texttt{CoreExp} \\ (\texttt{CoreHead}\ subst_1\ \ldots\ subst_i\ \ldots\ subst_n) \rightarrow_c (\texttt{CoreHead}\ subst_1\ \ldots\ subst'_i\ \ldots\ subst_n)\end{array}}{\begin{array}{c}(\texttt{CoreHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n) \rightarrow_e (\texttt{CoreHead}\ e_1\ \ldots\ e'_i\ \ldots\ e_n) \\ \textit{where}\ e_i \rightarrow_m e'_i\ \textit{if}\ e_i \in \texttt{SurfExp},\ \textit{else}\ e_i \rightarrow_c e'_i\end{array}} \qquad (\textsc{CoreExt1})$$

$$\frac{\begin{array}{c}\forall\ i.\ subst_i = (e_i \in \texttt{SurfExp}\ ?\ \texttt{tmpexp}\ :\ e_i),\ \textit{where}\ \texttt{tmpexp}\ \textit{is any reduciable}\ \texttt{CoreExp} \\ (\texttt{CoreHead}\ subst_1\ \ldots\ subst_n) \rightarrow_c e'\ \text{// not reduced in subexpressions}\end{array}}{(\texttt{CoreHead}\ e_1\ \ldots\ e_n) \rightarrow_e e'[e_1/subst_1\ \ldots\ e_n/subst_n]} \qquad (\textsc{CoreExt2})$$

For expression (`CoreHead` $e_1\ \ldots\ e_n$), replacing all subexpression not in core language with any reducible core language's term `tmpexp`. Then getting a result after inputting the new expression `e'` to the original blackbox stepper. If reduction appears at a subexpression at e_i or what the e_i replaced by, then the stepper with the extension should return (`CoreHead` $e_1\ \ldots\ e'_i\ \ldots\ e_n$), where e'_i is e_i after the mixed language's one-step reduction (*redm*) or after core language's reduction ($\rightarrow_c$) (the rule `CoreExt1`, an example in Figure 4). Otherwise, stepper should return `e'`, with all the replaced subexpressions replacing back. (the rule `CoreExt2`, an example in Figure 5) The extension will not violate properties of original core language's evaluator. It is obvious that the evaluator with the extension will reduce at the subexpression as it needs in core language, if the reduction appears in a subexpression. One may notice that the stepper with extension behaves the same as mixing the evaluation rules of core language and surface language. The extension is just to make it works when the evaluator of core language is a blackbox stepper. That's why the extension is tricky.

$$\begin{array}{c}\texttt{(if (and e1 e2) true false)} \\ \Downarrow_{replace} \\ \texttt{(if tmpe1 true false)} \\ \Downarrow_{blackbox} \\ \texttt{(if tmpe1' true false)} \\ \Downarrow_{desugar} \\ \texttt{(if (if e1 e2 false) true false)}\end{array}$$

Fig. 4. `CoreExt1`'s example

3.2 Resugaring Algorithm

Our resugaring algorithm works on our mixed language, based on the reduction rules of the core language and the desugaring rules for defining the surface language. Let $\rightarrow_e$ denote the one-step

```
(if (if true ture false) (and ...) (or ...))
                  ⇓replace
(if (if true ture false) tmpe2 tmpe3)
                  ⇓blackbox
(if true tmpe2 tmpe3)
                  ⇓replaceback
(if true (and ...) (or ...))
```

Fig. 5. CoreExt2's example

reduction of the core language (based on the blackbox stepper with extension), and $\rightarrow_d$ the one-step desugaring of outermost sugar. We define $\rightarrow_m$, the one-step reduction of our mixed language, as follows.

$$\frac{(\mathtt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_e\ e'}{(\mathtt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_m\ e'} \quad (\textsc{ExtRed})$$

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

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

The `CoreRed` rule describes how our mixed language handles expressions with `CoreHead`—just leave it to the core language's evaluator with the extension. Then for the expression with `SurfHead`, we will firstly desugar the outermost sugar (identified by the `SurfHead`), then recursively executing $\rightarrow_m$. In the recursive call, if one of original subexpression e_i is reduced (`SurfRed1`), then the original sugar is not necessarily desugared, we should only reduce the subexpression e_i; if not (`SurfRed2`), then the sugar have to desugar.

Then our desugaring algorithm is defined based on $\rightarrow_m$.

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

During the resugaring, we just call the mixed language's reduction ($\rightarrow_m$) on the input expression until the expression becomes a normal form. We use the `DisplayableExp` to restrict immediate sequences to be output or not. It is more explicit compared to existing approaches. And because $\rightarrow_m$ will be executed recursively on the subexpressions, it can be optimized. (see in 3.4, because the current description is more easier to understand.)

3.3 Correctness

Existing resugaring works[Pombrio and Krishnamurthi 2014, 2015] define three properties for correctness of resugaring. We think they are also reasonable to describe correctness of our approach. We describe the following properties in our mixed language's domain, then prove or discuss on them.

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.

Abstraction. Only displayable expressions defined in our mixed language appear in our resugaring sequences.

Coverage. No syntactic sugar is desugared before its sugar structure should be destroyed in core language.

3.3.1 Emulation. It is a basic property for correctness. Since desugaring won't change an expression after totally desugared, what we need to prove is that a non-desugaring reduction in the mixed language shows the exactly reduction which should appear after the expression totally desugared. We express it by following lemma. (`fulldesugar(exp)` returns the expression after `exp` totally desugared)

Lemma 3.1. *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')`

Defination 3.1 (Emulation). *If the mixed language satisfies Lemma 3.1, then the resugaring satisfies emulation property.*

Lemma 3.2. *For* `exp` = (`SurfHead` $e_1 \ \ldots \ e_i \ \ldots \ e_n$)*, if inputting* `fulldesugar(exp)` *to core language's evaluator reduces the term original from* e_i *in one step, then the* $\rightarrow_m$ *will reduce* `exp` *at* e_i.

Proof of Lemma 3.2. For (`SurfHead` $x_1 \ \ldots \ x_i \ \ldots \ x_n$) $\rightarrow_d \ e$

if e is of normal form, the `fulldesugar(exp)` will not be reduced by core evaluator.

if e is headed with `CoreHead`, then according to the `CoreRed` rule, the $\rightarrow_e$ will execute on e according to `ExtRed`, which will reduce the subexpression e_i according to the blackbox evaluator with extension. Then the `SurfRed2` rule will reduce e_i. Because of the extension of evaluator reduces the subexpression in correct location, so it is for $\rightarrow_m$.

if e is headed with `SurfHead`, then the *redm* will execute recursively on e. If the new one satisfies the lemma, then it is for the former. Because any sugar expression will finally be able to desugar to expression with `CoreHead`, it can be proved recursively. □

Proof of Lemma 3.1.

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''), then `fulldesugar(Exp)=fulldesugar(Exp')`. If not, what we need to prove is that, `fulldesugar(Exp)` $\rightarrow_c$ `fulldesugar(Exp')`. Note that the only difference between `Exp` and `Exp'` is the i-th subexpression, and we have proved the lemma 3.2 that the subexpression is the one to be reduced after the expression desugared totally, it will be also a recursive proof on the subexpression e_i.

For `SurfRed2` rule, `Exp'` is `Exp` after the outermost sugar resugared. So `fulldesugar(Exp)=`

`fulldesugar(Exp')`.

□

So our resugaring approach satisfies evaluation property.

3.3.2 Abstraction. Abstraction is not a restrict properties, because each expression has its meaning. Users may choose what they want to output during the process. Existing resugaring approaches use marks to determine whether to display a term generated by desugaring, or only changes on original terms will show.

We define the abstraction by catalog the expression in the mixed language, from the reason why we need resugaring—sugar expressions become unrecognizable after desugaring. So why cannot a recursive sugar's resugaring sequences show the sugars generated by itself? We think the users should be allowed to decide which terms are recognizable. Then during the resugaring process, if no unrecognizable term for the user appears in the whole expression, the expression should be shown as a step in resugaring sequences. 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.

3.3.3 Coverage. The coverage property is important, because resugaring sequences are useless if lose 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. *A syntactic sugar only desugars when* necessary*, that means after a reduction on the fully-desugared expression, the sugar's structure is destroyed.*

DEFINATION 3.2 (COVERAGE). *If the reduction of mixed language satisfies Lemma 3.3, then the resugaring satisfies coverage property.*

PROOF SKETCH OF LEMMA 3.3. From Lemma 3.2, we know the $\rightarrow_m$ recursively reduces a expression at correct subexpression. Or the $\rightarrow_m$ will destroy the outermost sugar (of the current expression) in rule `SurfRed2`. Note that it is the only rule to desugar sugars directly (other rules only desugar sugars when recursively call `SurfRed2`), we can prove the lemma recursively if `SurfRed1` is *lazy* enough.

In `SurfRed2` rule, we firstly expand the outermost sugar and get a temp expression with 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. □

3.4 Implementation

Our resugaring approach is implemented using PLT Redex[Felleisen et al. 2009], which is an semantic engineering tool based on reduction semantics[Felleisen and Hieb 1992]. The framework of the implementation is as Figure 6.

Instead of implementing a blackbox stepper of core language, we just used the core language's reduction semantics, because its behavior is same as the stepper with extension for mixed language. We have proved or discussed the correctness with the assumption that the core language's evaluator is a blackbox stepper. In the language model, desugaring rules are written as reduction rules of `SurfExp`. And context rules of `SurfExp` have no restrict (every subexpressions is reducible as a hole). Then for each resugaring step, we should choose the exact reduction which satisfies the reduction of mixed language's rule (see 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 e21 e22 ...) e11 e12 ...)` as the input, and find the first subexpression should be reduced. We can firstly get the resugaring sequences of `(Sugar2 e21 e22)`

```
(Sugar2 e21 e22 ...)
```

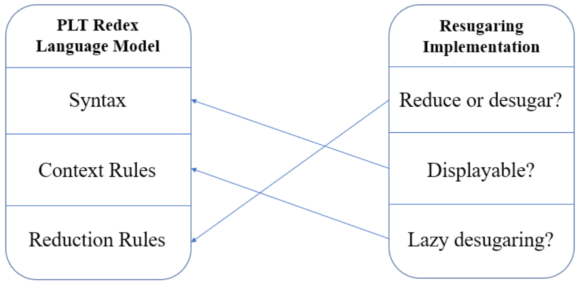

Fig. 6. framework of implementation

```
⟶ exp1
--> exp... // may be 0 or more steps
⟶ expn
```

Then a resugaring subsequence is got as

```
    (Sugar1 (Sugar2 e21 e22 ...) e11 e12 ...)
⟶ (Sugar1 exp1 e11 e12 ...)
--> (Sugar1 exp... e11 e12 ...) // may be 0 or more steps
⟶ (Sugar1 expn e11 e12 ...)
```

Thus, we will not need to try to expand the outermost sugar for each inner step (recursively resugaring for inner expression).

3.5 Application

We test some applications on the tool. 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.

3.5.1 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.

We take a SKI combinator syntactic sugar as an example. We will show why our approach is efficient.

$$\texttt{S} \longrightarrow_d \texttt{(lambdaN (x1 x2 x3) (x1 x2 (x1 x3)))}$$
$$\texttt{K} \longrightarrow_d \texttt{(lambdaN (x1 x2) x1)}$$
$$\texttt{I} \longrightarrow_d \texttt{(lambdaN (x) x)}$$

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 CoreLang. For sugar expression `(S (K (S I)) K xx yy)`, we get the following resugaring sequences.

```
    (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)
```

For the existing approach, the sugar expression should firstly desugar to

```
((lambdaN
   (x1 x2 x3)
   (x1 x3 (x2 x3)))
  ((lambdaN (x1 x2) x1)
   ((lambdaN
     (x1 x2 x3)
     (x1 x3 (x2 x3)))
    (lambdaN (x) x)))
  (lambdaN (x1 x2) x1)
  xx yy)
```

Then in our CoreLang, the execution of expanded expression will contain 33 steps. For each step, there will be many attempts to match and substitute the syntactic sugars to resugar the expression. It will omit more steps for a larger expression. So the unidirectional resugaring algorithm makes our approach efficient, because no attempts for resugaring the expression are needed.

3.5.2 hygienic sugar. The second work[Pombrio and Krishnamurthi 2015] of existing resugaring approach mainly processes hygienic sugar compared to first work. It use a DAG to represent the expression. However, hygiene is not hard to handle by our lazy desugaring strategy. Our algorithm can easily process hygienic sugar without special data structure.

A typical hygienic problem is as the following example.

$$\texttt{(Hygienicadd e1 e2)} \rightarrow_d \texttt{(let (x e1) (+ x e2))}$$

For existing 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 existing hygienic resugaring approach use 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, so, getting following sequences based on a hygienic rewriting system.

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

The lazy desugaring is also convenient for hygienic resugaring 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,

$$\texttt{(Let e1 e2 e3)} \rightarrow_d \texttt{(let ((e1 e2)) e3)}$$

and some simple modifies in the reduction of mixed language, we will get the following sequences in our system.

```
  (Let x 1 (+ x (Let x 2 (+ x 1))))
→ (Let x 1 (+ x (+ 2 1)))
```

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

In practical application, we think hygiene can be easily processed by rewriting system, so we just use a rewriting system which can rename variable automatically. But our result shows lazy desugaring is really a good way to handle hygienic sugar in any systems.

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

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

are common recursive sugars. The existing resugaring approach can't process syntactic sugar written as 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 of the former example using our tool.

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

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.

$$\texttt{(filter e (list v1 v2 ...))}$$
$$\longrightarrow_d \texttt{(if (e v1) (cons v1 (filter e (list v2 ...))) (filter e (list v2 ...)))}$$
$$\texttt{(filter e (list))} \longrightarrow_d \texttt{(list)}$$

and getting the following result. (by making `(lambda ...)` CommonExp)

```
  (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)
```

Here, although the sugar can be processed by existing 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 which 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.

```
(map e1 e2) ⟶d
(let ((x e2)) (if (empty x) (list) (cons (e1 (first x)) (map e1 (rest x)))))
```

Get following resugaring sequences.

```
  (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)
```

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 by rewriting system. But our approach is easy to handle cases like this.

4 STATIC APPROACH

In this section, we introduce a static approach, which is more efficient than the one discussed above.

4.1 Inference Automaton

Based on the idea of DFA (Deterministic Finite Automaton), we designed inference automaton (IFA). An IFA describes the inference rules of a certain syntactic structure. To help readers better understand it, first we give a few examples, then we give the formal definition of IFA and proofs of theorems.

4.1.1 IFA of if. The inference rules of `if` are shown as AAAR1. We can observe that an if term is first evaluated for e1, and is chosen to be evaluated for e2 and e3 depending on the value of it, then the result of the evaluation of e2 or e3 is the result of the evaluation of the term. Thus, we use AAAP1 to represent the inference rules of if.

The arrow from e1 to e2 indicate that this branch will be selected when the result of the e1 evaluation is #t. The arrows between e1 and e3 are the same. The double circles of e2 and e3 denotes that their evaluation result is the result of the syntactic structure. When a term with an if syntactic structure needs to be evaluated (for example if (if #t #t #f) #f #t)), first evaluating the e1 (if #t #t #f) part. Note that in this process, evaluating a subexpression requires running another automaton based on its syntax, while the outer automaton hold the state at e1. According to the result of e1 (#f), the IFA selects the branch (e3). Then the result of e3 (#t) will be the evaluation result of the term.

4.1.2 IFA of nand. Sometimes the rules may be more complex, such as being reduced into another syntactic structure, or the term contains other syntactic structures. For example, we can express nand's inference rule in the form of AAAR2. Based on the method discussed above, we can draw nand's IFA as AAAP2.

When the automaton runs to the last node, its evaluation rule is essentially an evaluation of the if syntax structure. Thus we can replace the last node with an IFA_if and use the IFA_if termination nodes as the termination nodes of the IFA_Nand. The results are shown in AAAP3. Further decomposing the intermediate nodes, connecting the terminating node of IFA_if to the node pointed to by the original output edge, we get AAAP4.

As can be seen, the nodes of IFA in AAAP4 have only the forms e_i, v_i and values, and no other composite syntactic structure. We call such an IFbA a *standard IFA*.

4.1.3 IFA of or. We represent the inference rule of or in a more complex way, as shown in AAAR3. In this case, we use the let binding, which expresses a class of rules containing substitution. At this point, we need to record the term represented by each variable at each node, denoted by Γ. The representation of IFA_or is shown in AAAP5.

More generally, the handling of substitution tables will be more complex. We will discuss this in more detail in AAAF1.

4.1.4 Definition of IFA.

DEFINATION 4.1 (INFERENCE AUTOMATON). *An inference automaton (IFA) of syntactic structure* (`Headid` $e_1 \dots e_n$) *is a 5-tuple,* $(Q, \Sigma, q_0, F, \delta)$*, consisting of*

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

and for each node q, there is no sequence of pattern $P = (p_1, p_2, \dots, p_n) \subseteq \Sigma^*$, which makes that after q transfers sequentially according to P, it returns q.

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

In IFA, state transition does not depend on input. The only input IFA accepts is the term to be evaluated with this syntactic structure. The state transition is through pattern matching on the evaluation result of the term in the previous node. 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.

Defination 4.2 (Standard IFA). *If the term of node in Q can only be e_i (where $i \in 1, \ldots, n$) or a value, we name the IFA standard IFA.*

If an IFA is standard, it means there are no more composite syntactic structures in it. In the above example, for the syntactic structure of `if`, we substituted the IFA of the `if` into `nand` and converted it into a standard IFA. Below we will prove that it is always feasible to convert IFA to standard IFA, and give the algorithm.

Lemma 4.1. *Considering an IFA of a syntactic structure, if the standard IFAs of all syntactic structures of terms contained in the IFA are known, then the IFA can be transformed into a standard IFA.*

Proof of Lemma. □

4.2 Convert inference rules to IFA

Considering the inference rules in CoreLang, which we have more strict limits on.

Assumption 1. A syntactic structure *Headid* only contains the following inference rules.

$$\frac{(\texttt{Headid}\ v_1 \ldots v_p\ e_1 \ldots e_i \ldots e_q) \rightarrow (\texttt{Headid}\ v_1 \ldots v_p\ e_1 \ldots e_i' \ldots e_q)}{e_i \rightarrow e_i'} \qquad \text{(E-Head)}$$

$$(\texttt{Headid}\ v_1 \ldots v_p\ e_1 \ldots e_q) \rightarrow e$$

$$(\texttt{Headid}\ v_1 \ldots v_p\ e_1 \ldots e_q) \rightarrow \textbf{let}\ x = Exp_1\ \textbf{in}\ Exp_2$$

This assumption specifies the form of the inference rules to ensure that IFAs can be generated. The first one is context rule, and the others are reduction rules.

Assumption 2. The syntactic structure in CoreLang is finite. Think of all syntactic structures as points in a directed graph. If one of *Headid*'s inference rules can generate a term containing *Headid'*, then construct an edge that points from *Headid* to *Headid'*. The directed graph generated from this method has no circles.

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

Assumption 3. 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 process of sub-expressions.

Lemma 4.2. *If a term* $(\texttt{Headid}\ e_1\ \ldots\ e_n)$ *does a one-step evaluation by rule (E-Head) of* `Headid`*, which is a one-step evaluation of e_i, then it will continue to use this rule until e_i becomes a value.*

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

Lemma 4.3. *If all syntactic structures in CoreLang satisfy Assumption 1 and Assumption 2, We can construct standard IFAs for all syntactic structures in CoreLang.*

Proof of Lemma. By Assumption 2, we get an order of syntactic structures. We generate the IFA for each structure in turn.

We generate a node for each rule and insert them into Q. If the rule is a reduction rule, add them into F as terminal nodes. Next we will connect these nodes.

For a term like $(\texttt{Headid}\ e_1\ \ldots\ e_n)$, considering that $e_1 \cdots e_n$ are not value, According to Assumption 3, we have the unique rule r of *Headid* 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_0 be build a new node q and add it into Q. The term of q is e_i. And the symbol table is set to empty. Assume that the evaluation of e_i results in v_i, we get term ($\texttt{Headid}\ e_1 \ldots e_{i-1}\ v_i\ e_{i+1} \ldots e_n$). For each possible value of v_i, choose the rules that should be used.

If r is a reduction rule, build a new node q and add it into Q and F. The term of q is e_i, and the symbol table is set to

□

4.3 Convert IFA to Inference Rules

Lemma 4.4. *For each IFA, it can be converted to inference rules.*

Proof of Lemma. Give an algorithm: convert IFA to inference rules. □

4.4 Syntactic Sugar

With the IFA, we can easily get the inference rules for syntactic sugars.

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 inference rule is*

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

5 RELATED WORK

Resugaring sequences [Pombrio and Krishnamurthi 2014, 2015] As we have discussed many times, the concept of resugaring is original from their work, by the main idea of "tagging" and "reverse desugaring". Our approach is more lightweight, powerful and efficient, as discussed before. In summary, we also find some common issues about resugaring.

- Side effects in resugaring. In the first paper of resugaring, they try a `letrec` sugar based on `set!` term in core language and get no intermediate steps. After trying some syntactic sugars that contain side effects, we would say a syntactic sugar including side-effect is bad for resugaring, because after a side effect takes effect, the desugared expression should never resugar to the sugar expression. Thus, we don't think resugaring is useful for syntactic sugars including side effects, though it can be done by marking any expressions which have a side effect.
- Hygienic resugaring. As we showed in both the basic approach and the automatic derivation, hygiene is easily and naturally resolved by lazy desugaring, because it may behave as what the sugar ought to express. The second paper of resugaring presents a DAG to solve the problem, which is a smart but not concise way.
- Assumption on core language. The traditional resugaring and the dynamic approach both use a blackbox evaluator of core language, while the dynamic approach use the semantics of core language. We found that if given the semantics of core language, the resugaring will be more convenient. The blackbox evaluator in our dynamic approach will not need the extension, while the rules getting by our static approach is more express.

Type resugaring [Pombrio and Krishnamurthi 2018] is a later work of sequence resugaring. It automatically derivation type rules of syntactic sugar by unification. It indicates that it is possible to automatically construct surface language's semantics by unification. But after trying to do this as type resugaring does, we found it hard because the method may assume only one rule for multi-branches of one syntactic sugar, which is not suitable for evaluation rules. We also find getting evaluation rules in big-step semantics is easier, but not works well for syntactic sugar. *Pretty-Big-Step Semantics* [Charguéraud 2013] is an improved version of big-step semantics, which may be useful to achieve more powerful automatic derivation of evaluation rules.

Macros as Multi-Stage Computations [Ganz et al. 2001] is a work similar to lazy expansion for macros. Some other researches[Rompf and Odersky 2010] about multi-stage programming[Taha 2003] indicate that it is an useful idea for implementing domain-specific languages. Overall, multi-stage programming is a metaprogramming method based on run-time code generation. In contrast, our resugaring approach gives abstraction by carefully design the reduction of mixed language. Moreover, the lazy desugaring gives us chance to derive evaluation rules of sugars, which is a new good point compared to multi-stage programming.

Galois slicing for Imperative Functional Programs [Ricciotti et al. 2017] is 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. Our approach used a similar idea on slicing expressions and processing on subexpressions. It's just like bidirectional transformation [Czarnecki et al. 2009] and the round-tripping between desugaring and resugaring in existing approach. In contrast, our resugaring approach is unidirectional, by an attempt on expanding sugar, just like the forward component of Galois connection. It should be noted that Galois slicing may be useful to handle side effects in resugaring (for example, slicing the part where side effects appear).

Origin tracking[Deursen et al. 1992] is about tracking the origins of terms in rewriting system, which does not involve the use of syntactic sugar. The key idea *lazy desugaring* makes our approach easy to solve the problems introduced by syntactic sugar. And the tracking in their notation can be easily done for sugar whose rules can be derived automatically.

Since the existing resugaring approach firstly desugars all sugar in programs, it is more similar to debugging process, where as our approach is likely engineering the semantics of language. *Ziggurat* [Fisher and Shivers 2006] is a semantic-extension framework. It allows defining new macros with semantics based on existing terms in a language. It is quite useful for static analysis on macros. Instead of semantics based on core language, the reduction rules of sugar derived by our approach is independent of core language, which may be more concise for static analysis. Addition to PLT Redex[Felleisen et al. 2009] which we used to engineer the semantics, there are some other semantics engineering tools [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 implement more general rule derivation.

6 CONCLUSION

In this paper, we purpose a lightweight, efficient and powerful resugaring approach by lazy desugaring. Essentially, we would see the derivation of rules is the abstract of the basic resugaring approach. In the basic 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 derivated context rules; desugaring the outermost sugar is similar to derivated reduction rules. However, the derivated evaluation rules is more convinent and efficent 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. In contrast, the dynamic approach is more powerful by supporting recursive sugars' resugaring. Besides, the rewriting based on reduction semantics makes the sugar represented in many ways.

The original intent of our research is finding a better method (or building a tool) for implementing DSL. We could see derivated evaluation rules is better for achieving the goal, because getting the semantics of DSL (based on syntactic sugar) will be very useful for applying any other techniques on the DSL. But it will be better if the defects of expressiveness of sugar which the derivation can handle are improved. So we may be achieving a more powerful derivation which can recursive recursive sugar as a future work.

REFERENCES

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3

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

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

A APPENDIX

Text of appendix ...