Lifting Resugaring by Lazy Desugaring

ANONYMOUS AUTHOR(S)

With the rapid development of computer science, domain-specific language (DSL) is quite useful in our daily life, not only for programmers or computer scientists, but for people from all walks of life. Syntactic sugar is a good way to implement embedded DSLs, because it can make good use of existing general-purposed language's feature. However, the evaluation sequences became unrecognizable after the sugar expression desugared.

Resugaring is an method to solve the problem above. In this paper, we purposed a lightweight approach of resugaring based on reduction semantics—getting evaluation sequences without fully desugaring the whole syntactic sugar expression. We implement a tool based on our method using PLT Redex and test our approach on some applications. The results show that our lightweight approach can even deal with more syntactic sugar's feature.

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

1 INTRODUCTION

```
    let t = not(true) in
      if t then t else not(false)

--> let t = false in
      if t then t else not(false)

--> if false then false else not(false)

--> not(false)

--> true
```

Domain-specific language[Fowler 2011] is becoming useful for people's daily tasks. For example, the IFTTT app and IOS's shortcuts designed DSLs describing some tasks to make our lives more convenient. So the users of DSL are no longer limited to programmers, but people from all walks of life.(to be completed)

Syntactic sugar[Landin 1964], as a simple way of implementing DSL, has an obvious problem. DSL based on syntactic sugars contains many components of its host language. Then its interpretation will be outside the DSL itself. The evaluation sequences of syntactic sugar expressions will contain many terms of the host language, which may confuse the users of DSL.

There is an existing work—resugaring[Pombrio and Krishnamurthi 2014][Pombrio and Krishnamurthi 2015], which aimed to solve the problem upon. It converted the evaluation sequences of desugared expression (core language) into representative sugar's syntax (surface language). The evaluation sequences shown by resugaring will not contain components of host language (todo: another express?). But we found the existing resugaring approach using match and substitution is kind of redundant. The biggest deficiency of existing resugaring method is that the syntactic sugars in an expression have to fully desugar before evaluation. This limits the processing ability of the method. Moreover, it limits the complexity of getting the resugaring sequences. If we need to resugar a very huge expression, the match and substitution processes will cost so much. Also, processing of hygienic macros is a little bit complex due to the extra data structure. Finally, we found the existing approach only assumes a stepper for core language, when the semantics of core

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

languages can be got in some cases. We want to figure out how the semantics of core language will help.

In this paper, we propose an unidirectional resugaring approach by lazy dusugaring mixed with a dynamic approach and a static approach. The key idea of the whole approach is—syntactic sugar expressions only desugar at the point they have to desugar, which is what the word "lazy" means. It would be correct for resugaring if we can prove the whole sugar expressions will keep the properties.

The dynamic approach use the reduction semantics[**?**] of core language to decide whether desugaring the sugar. The static approach use the reduction semantics of core language to get reduction semantics of surface language based on sugars' syntax, then execute the syntactic sugar programs on the surface's semantics. The context rules of surface language decide which subexpression can be reduced, or desugaring is necessary because of the reduction rules.

Our main contribution is as follow:

- **A mixture approach of resugaring.** We introduce an mixture of two different resugaring approachs to combine the advances of following approaches. The lazy dusugaring is common feature of two approaches, which give each approach some good properties.
- **A lightweight but powerful dynamic approach.** The dynamic approach we proposed is based on core language's reduction semantics. It takes surface language and core language as a whole, then decided whether expanding 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 dynamic approach on many applications. The result shows that in addition to handle what existing work can handle, our dynamic approach can process recursive sugar easily, which makes it powerful. And the rewriting system based on reduction semantics makes it possible to write syntactic sugar easily.
- **An independent and efficient static approach.** The static approach we proposed also used core language's reduction semantics. But instead of executing at the level of core language, we turn the core language's semantics into automata. Then for each syntactic sugar, we would generate the surface language's semantics without depending on some rules in core language. (some meta-functions may be necessary.) Thus, it is efficient because many steps in core language can be omitted. todo: complete

In the rest of this paper, we present the technical details of our approach together with the proof of correctness. In details, the rest of our paper is organized as follow:

- An overview of our approach with mixed with dynamic and static approach.[sec 2]
- The technique of dynamic approach, with algorithm and evaluation.[sec 3]
- The technique of static approach, todo.[sec **??**]
- Relative work and discussions.[sec 5]
- Conclusion and feature work.[sec 6]

2 OVERVIEW

2.1 Defination of resugaring

This subsection is partially similiar to original defination in[Pombrio and Krishnamurthi 2014].

Defination 2.1 (Resugaring). *Given core language (named **CoreLang**) and its evalation rules, together with surface language based on syntactic sugars of CoreLang (named **Surflang**). For any expression of Surflang, getting the evaluation sequences of the expression in terms of SurfLang.*[1]

[1]It's not strict, because we could allow some expressions in CoreLang shown.

For correctness of the resugaring, the evaluation sequences should maintain the following three properties:

(1) **Emulation** The evaluation sequences reflect the actual execution process.
(2) **Abstraction** The resugaring sequences should only contains terms in SurfLang, and each term of SurfLang should originate from initial expression.
(3) **Coverage** No sequence is skipped during the process.

Given an example below.

For syntactic sugar **and** and **or**, the sugar rules are:

(and e1 e2) ⟶ (if e1 e2 #f)
(or e1 e2) ⟶ (if e1 #t e2)

which forms a simple SurfLang.

The evaluation rules of **if** is:

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

Then for SurfLang's expression and(or($\#f$, $\#t$), and($\#t$, $\#f$)) should get resugaring sequences as follow.

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

The reason we should get the sequences above is because (and (or $\#f$ $\#t$) (and $\#t$ $\#f$)) should desugar to (if (if $\#f$ $\#t$ $\#f$) (if $\#t$ $\#f$ $\#f$) $\#f$). Then in the CoreLang, the evaluation sequences will be as follow.

```
   (if (if #f #t #f) (if #t #f #f) #f)
⟶ (if #t (if #t #f #f) #f)
⟶ (if #t #f #f)
⟶ #f
```

The second item in the sequences can be desugared from (and $\#t$ (and $\#t$ $\#f$)), so resugars to it. So as the third item.

2.2 Mixture Approach Framework

We limit the language to s-expressions. Given an expression Exp = (Headid Exp∗), the process of mixture approach will as Fig 1.

Given an example here.

For syntactic sugar **and**, **or** and **map**, the sugar rules are:

(and e1 e2) ⟶ (if e1 e2 #f)
(or e1 e2) ⟶ (if e1 #t e2)
(f e lst) ⟶ (if (empty? lst) empty (cons (let x e (or x (and (first lst) x))) (f e (rest lst))))

which forms a simple SurfLang.

As we will see in Sec3, the dynamic approach could handle recursive syntactic sugar, and the static approach could not. If we execute (f #t (list #f #t)), the expression will Since the dynamic

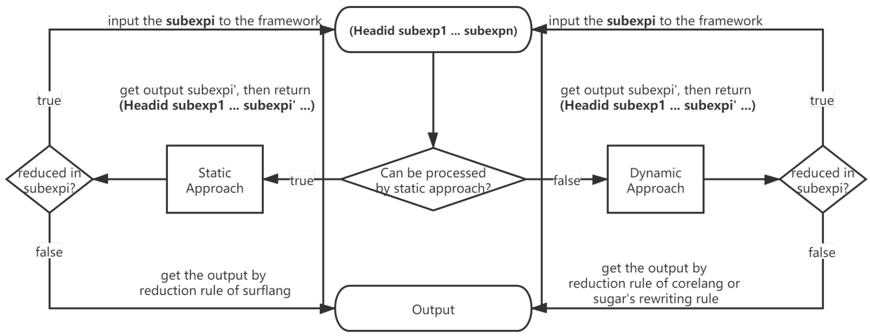


Fig. 1. One step in framework of mixture approach

approach is more powerful and the static approach is more efficent, we should judge whether the input expression can be processed by static approach. If so, then use the static approach; if not, then use the dynamic approach to get (cons (let x #t (or x (and (first (list #f #t)) x))) (f #t (rest (list #f #t)))). Then the expression with headid **cons** can be handled by static approach, which let the first subexpression reduced, so we get subsequences like (cons ... (f #t (rest (list #f #t)))), (cons #f (f #t (rest (list #f #t)))). At this time, the static approach will let the second subexpression reduced, so the expression with headid **f** will be processed by dynamic approach. (A recursive process it is.)

The key idea of our dynamic approach, is that, regarding surface language and core language as a whole under the strategy of lazy desugaring. We design a core algorithm to choose the right reduction rule for any expression during the execution. Take the example and(or($\#f, \#t$), and($\#t, \#f$)) again. We will get the sequence as ...

(and (or #f #t) (and #t #f))
step1
(and (if #f #t #t) (and #t #f))
step2
(and #t (and #t #f))
step3
(if #t (and #t #f) #f)
step4
(and #t #f)
step5
(if #t #f #f)
step6
#f

Fig. 2. core-algo example

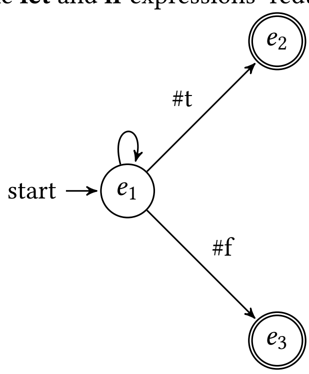
At step 1, we found the outermost *and* sugar don't have to expand, because its first sub-expression will reduce earlier. At step 2, the same as step 1. At step 3, the outermost *and* sugar have to expand, because no sub-expression will reduce after the whole expression desugar. At step 4, the inner *and* sugar don't have to expand either. At step 5, the sugar have to desugar to CoreLang. Finally at step 6, we get the final result.

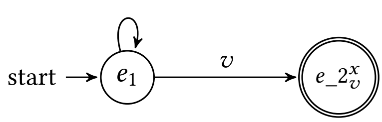
The key idea of our dynamic approach, is that, converting reduction semantics of core language into automata (called **IFA**), building IFA for syntactic sugar, converting the IFA of sugars into reduction semantics. It is an abstract of dynamic approach in a sence, we will discuss it in Sec6. Take another **or** sugar for example.

(or e1 e2) ⟶ (let x e1 (if x x e2))

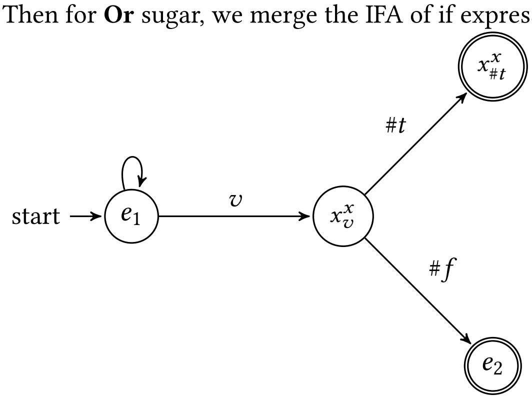
$(Or\ e_1\ e_2) \longrightarrow (let\ x\ e_1\ (if\ x\ x\ e_2))$

The **let** and **if** expressions' reduction semantics can be represented as the following automata.





Then for **Or** sugar, we merge the IFA of if expression into node e_2 of let's IFA.



From the IFA of or expression, we can get the following reduction semantics.

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

$$(\text{Or } \#t\ e2) \longrightarrow \#t$$

$$(\text{Or } \#f\ e2) \longrightarrow e_2$$

Then the resugaring sequences can be get by the reduction semantics.

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

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

Fig. 3. Core and Surface Expressions

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

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 3. It is a variable, a constant, or a (language) constructor expression. Here, cId stands for a language constructor such as IF and LET. To be concrete, we will use the core language defined in Figure 4 to demonstrate our approach.

3.1.2 Surface Language. Our surface language is defined by a set of syntactic sugars, together with some language constructs in the core language. So an expression of the surface language is some core constructor expressions with sugar expressions, as defined in Figure 3.

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

$$(\textsc{SurfHead}\ x_1\ x_2\ \ldots\ x_n) \Rightarrow \textsc{SurfExp}$$

where its LHS is a simple pattern (unnested) and its RHS is a surface expression. Note that if the pattern is nested, we can introduce a new syntactic sugar to flatten it. One may wonder why not restricting the RHS to be a core expression CE, which sounds more natural. We use SE to be able to allow definition of recursive syntactic sugars, which will be seen later.

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

3.1.3 Mixed Language. Our mixed language for resugaring combines the surface language and the core language. The difference between our core language (CoreLang) and our surface language (SurfLang) is identified by HeadId. But there are some terms in the core language should be displayed during evaluation, or we need some terms to help us getting better resugaring sequences. So we defined CommonExp, which origin from CoreLang, but can be displayed in resugaring

Syntax	Reduction rules
(if e e e)	(if #t e2 e3) $\longrightarrow$ e2 (if #f e2 e3) $\longrightarrow$ e3
((λ (x ...) e) e ...)	((λ (x0 x1 ...) e) v0 v1 ...) $\longrightarrow$ (let ((x0 v0) ((λ (x1 ...) e) v1 ...))
((λ_N (x ...) e) e ...)	((λ_N (x0 x1 ...) e) e0 e1 ...) $\longrightarrow$ (let ((x0 e0) ((λ_N (x1 ...) e) e1 ...))
(let ((x e) ...) e)	(let ((x0 e0) (x1 e1) ...) e) $\longrightarrow$ (let ((x1 e1) ...) (subst x0 e0 e)) (let () e) $\longrightarrow$ e (where subst is a meta function)
(first e)	(first (list v1 v2 ...)) $\longrightarrow$ v1
(rest e)	(rest (list v1 v2 ...)) $\longrightarrow$ (list v2 ...)
(empty e)	(empty (list)) $\longrightarrow$ #t (empty (list v1 ...)) $\longrightarrow$ #f
(cons e e)	(cons v1 (list v2 ...)) $\longrightarrow$ (list v1 v2 ...)
(op e e) op=+-*/><==	(op v1 v2) $\longrightarrow$ arithmetic result

Fig. 4. An Core Language Example

Exp	::=	DisplayableExp
	|	UndisplayableExp
DisplayableExp	::=	SurfExp
	|	CommonExp
UndisplayableExp	::=	CoreExp
	|	OtherSurfExp
	|	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. 5. Our Mixed Language

sequences. The CoreExp terms are terms with undisplayable CoreLang's HeadId. The SurfExp terms are terms with SurfLang's HeadId and all sub-expressions are displayable. The CommonExp terms are terms with displayable CoreLang's Headid, together with displayable sub-expressions. There exists some other expression during our resugaring process, which have HeadId which can be displayed, but one or more subexpressions cannot. They are UndisplayableExp.

Take some terms in the core language in Figure 4 as examples. We may assume **if**, **let**, λ_N, **empty**, **first**, **rest** as CoreExp's Headid, **Op**, λ, **cons** as CommonExp's Headid. Then we could show some useful intermediate steps.

3.2 Resugaring Algorithm

Our resugaring algorithm is based on a core algorithm f. For every expression during resugaring process, it may have one or more reduction rules. The core algorithm f chooses the one that satisfies three properties of resugaring, then applies it on the given expression. The core algorithm f is defined as 1.

Algorithm 1 Core-algorithm f

Input:

Any expression Exp=($Headid\ Subexp_1\ \dots\ Subexp_{\dots}$) which satisfies Language setting

Output:

Exp' reduced from Exp, s.t. the reduction satisfies three properties of resugaring

1: Let $ListofExp' = \{Exp'_1, Exp'_2\ \dots\}$
2: **if** Exp is Coreexp or Commonexp or OtherCommonexp **then**
3:   **if** Lengthof($ListofExp'$)==0 **then**
4:     **return** null; Case1
5:   **else if** Lengthof($ListofExp'$)==1 **then**
6:     **return** first($ListofExp'$); Case2
7:   **else**
8:     **return** $Exp'_i = (Headid\ Subexp_1\ \dots\ Subexp'_i\ \dots)$; //where i is the index of subexp which have to be reduced. Case3
9:   **end if**
10: **else**
11:   **if** Exp have to be desugared **then**
12:     **return** desugarsurf(Exp); Case4
13:   **else**
14:     Let $DesugarExp$ = desugarsurf(Exp)
15:     **if** $Subexp_i$ is reduced to $Subexp'_i$ during $f(DesugarExp)$ **then**
16:       **return** $Exp'_i = (Headid\ Subexp_1\ \dots\ Subexp'_i\ \dots)$; Case5
17:     **else**
18:       **return** $DesugarExp$; Case6
19:     **end if**
20:   **end if**
21: **end if**

We briefly describe the core algorithm f in words.

For Exp in language defined as last section, try all reduction rules in the language, get a list of possible expressions ListofExp'={Exp'_1,Exp'_2,...}.

Line 2-9 deal with the case when Exp has a CoreLang's Headid. When Exp is value or variable (line 3-4), ListofExp' won't have any element (not reducible). When Exp is of Coreexp or Commonexp (line 5-6), due to the context restriction of CoreLang, only one reduction rule can be applied. When Exp is OtherCommonexp (line 7-8), due to the context restriction of CoreLang, only one sub-expression can be reduced, then just apply core algorithm recursively on the sub-expression.

Line 10-21 deal with the case then Exp has a SurfLang's Headid. When Exp only has one reduction rule (line 11-12), the syntactic sugar has to desugar. If not, we should expand outermost sugar and find the sub-expression which should be reduced (line 14-16), or the sugar has to desugar (line 17-18), because it will never be resugared. The steps in line 14 to 16 are the critical part of our algorithm (call **one-step try**).

Then, our lightweight-resugaring algorithm is defined as 2.

Algorithm 2 Lightweight-resugaring

Input:

Surfexp *Exp*

Output:

Exp's evaluation sequences within DSL

```
while tmpExp = f(Exp) do
    if tmpExp is empty then
        return
    else if tmpExp is Surfexp or Commonexp then
        print tmpExp;
        Lightweight-resugaring(tmpExp);
    else
        Lightweight-resugaring(tmpExp);
    end if
end while
```

The whole process of the lightweight resugaring executes core algorithm f, and output sequences which is of Surfexp or Commonexp.

3.3 Proof of correctness

First of all, because the difference between our lightweight resugaring algorithm and the existing one is that we only desugar the syntactic sugar when needed, and in the existing approach, all syntactic sugar desugars firstly and then executes on CoreLang.

Second, to prove convenience, define some terms.

$Exp = (Headid\ Subexp_1\ Subexp_{\ldots} \ldots)$ is any reducible expression in our language.

If we use the reduction rule that desugar Exp's outermost syntactic sugar, then the reduction process is called **Outer Reduction**.

If the reduction rule we use reduce $Subexp_i$, where $Subexp_i$ is $(Headid_i\ Subexp_{i1}\ Subexp_{i\ldots} \ldots)$

- If the reduction process is Outer Reduction of $Subexp_i = (Headid_i\ Subexp_{i1}\ Subexp_{i\ldots} \ldots)$, then it is called **Surface Reduction**.
- If the reduction process reduces $Subexp_{ij}$, then it is called **Inner Reduction**.

Example:

(if $\#t$ Exp_1 Exp_2) $Exp1$	Outer Reduction
(if (And $\#t$ $\#f$) Exp_1 Exp_2) (if (if $\#t$ $\#f$ $\#f$) Exp_1 Exp_2)	Surface Reduction
(if (And (And $\#t$ $\#t$) $\#t$) $\#f$) Exp_1 Exp_2) (if (And $\#t$ $\#t$) Exp_1 Exp_2)	Inner Reduction

DEFINATION 3.1 (UPPER AND LOWER EXPRESSION). *For* Exp=$(Headid\ Subexp_1\ Subexp_{\ldots} \ldots)$, Exp *is called* ***upper expression***, $Subexp_i$ *is called* ***lower expression***.

We only need to prove that all the 6 cases of core algorithm f won't effect its properties. Case 1 and case 3 won't effect any properties, because it does what CoreLang should do.

Proof of Emulation.
For case 4 and 6, desugaring won't change Emulation property, because desugaring and resugaring are interconvertible.

For case 2 and 5, our core algorithm reduces the sub-expression which should be reduced. So if applying core algorithm f on the subexpression satisfies emulation property, then this two cases satisfy. A recursive proof it is. todo:case5 □

Proof of Abstraction.
It's true, because we only display the sequence which satisfies abstraction property. □

Lemma 3.1. *If no syntactic sugar desugared before it has to, then coverage property is satisfied.*

Proof of Lemma. Assume that no syntactic sugar not necessarily expanded desugars too early, existing an expression in CoreLang

$Exp = (Headid\ Subexp_1\ Subexp_{\ldots}\ \ldots)$ which can be resugared to

$ResugarExp' = (Surfid\ Subexp'_1\ Subexp'_{\ldots}\ \ldots)$, and $ResugarExp'$ is not displayed during lightweight-resugaring. Then

- Or existing
  $ResugarExp$=$(Surfid\ Subexp'_1\ \ldots\ Subexp_i\ Subexp'_{\ldots}\ \ldots)$ in resugaring sequences, such that the expression after $ResugarExp$ desugaring reduces to Exp, and the reduction reduces $ResugarExp$'s sub-expression $Subexp_i$. If so, outermost syntactic sugar of $ResugarExp$ is not expanded. So if $ResugarExp'$ is not displayed, then the sugar not necessarily expanded desugars too early, which is contrary to assumption.
- Or existing
  $ResugarExp$=$(Surfid'\ \ldots\ ResugarExp'\ \ldots)$ in resugaring sequences, such that the expression after $ResugarExp$ desugaring reduces to Exp, and Exp is desugared from $ResugarExp'$'s sub-expression. If $ResugarExp'$ is not displayed, then the outermost syntactic sugar is expanded early, which is contrary to assumption. todo

□

Proof of Coverage.
For case 4 and 6, the syntactic sugar has to desugar.

For case 2 and 5, the reduction occurs in sub-expression of Exp. So if applying core algorithm f on the subexpression doesn't desugar syntactic sugars not necessarily expanded, then this two cases don't. A recursive proof it is. □

3.4 Implementation

Our lightweight approach of resugaring is implemented using PLT Redex[Felleisen et al. 2009], which is an semantic engineering tool based one reduction semantics. The whole framework is as Fig6.

The grammar of the whole language contains Coreexp, Surfexp and Commonexp as the language setting in sec3. OtherSurfexp is of Surfexp and OtherCommonexp is of Commonexp. The identifier of any kind of expression is Headid of expression. If we need to add a syntactic sugar to the whole language, only three steps is needed.

(1) Add grammar of the syntactic sugar.
(2) Add context rules of the sugar, such that any sub-expressions can be reduced.
(3) Add desugar rules of the sugar to reduction rules of the whole language.

Then inputting an expression of the syntactic sugar to lightweight-resugaring will get the resugaring sequences.

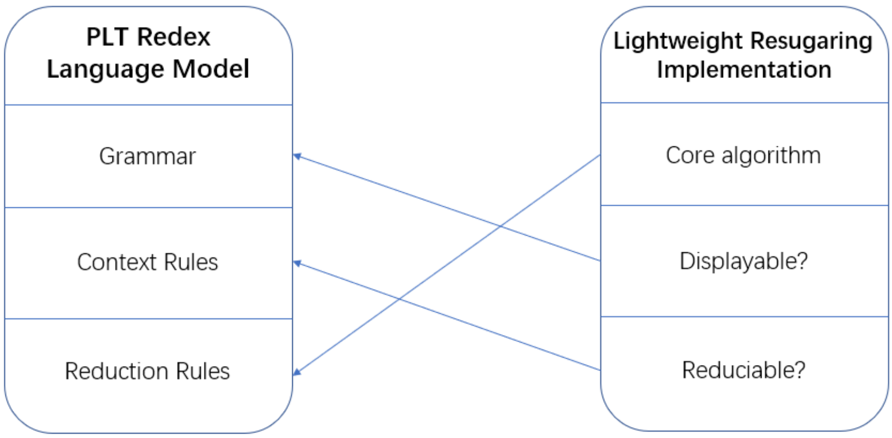


Fig. 6. framework of implementation

3.5 Evaluation

We test some applications on the tool implemented using PLT Redex. Note that we set CBV's 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 sugar 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 process all sugar features of their first work.

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

$S \longrightarrow (\lambda_N\ (x_1\ x_2\ x_3)\ (x_1\ x_2\ (x_1\ x_3)))$
$K \longrightarrow (\lambda_N\ (x_1\ x_2)\ x_1)$
$I \longrightarrow (\lambda_N\ (x)\ x)$

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 expression $(S\ (K\ (S\ I))\ K\ xx\ yy)$, we get the following resugaring sequences as following.

```
   (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 existing approach, the sugar expression should firstly desugar to

$((\lambda_N(x_1x_2x_3)(x_1x_3(x_2x_3)))((\lambda_N(x_1x_2)x_1)((\lambda_N(x_1x_2x_3)(x_1x_3(x_2x_3)))(\lambda_N(x)x)))(\lambda_N(x_1x_2)x_1)xxyy)$

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. We will omit more steps for a larger expression. So the unidirectional resugaring algorithm makes our approach lightweight.todo

3.5.2 hygienic macro. The second work[Pombrio and Krishnamurthi 2015] mainly processes hygienic macro compared to first work. We try a *Let* sugar , which is a common hygienic sugar example, on our tool. Our algorithm can easily process hygienic macro without special data structure. The *Let* sugar is define as follow

$(Let\ x\ v\ exp) \longrightarrow (Apply\ (\lambda\ (x)\ exp)\ v)$

Take $(Let\ x\ 1\ (+\ x\ (Let\ x\ 2\ (+\ x\ 1))))$ for an example. First, a temp expression

$(Apply\ (\lambda\ (x)\ (+\ x\ (Let\ x\ 2\ (+\ x\ 1))))\ 1)$

is needed. (case 5 or 6)Then one-step try on the temp expression, we will get

$(+\ 1\ (Let\ 1\ 2\ (+\ 1\ 1)))$ which is out of the whole language's grammar. In this case, it is not a good choice to desugar the outermost *Let* sugar. Then we just apply the core-algo f on the sub-expression where the error occurs ($(+\ x\ (Let\ x\ 2\ (+\ x\ 1)))$ in this example). So the right intermediate sequence $(Let\ x\ 1\ (+\ x\ 3))$ will be get.

In practical application, we think resugaring for a unhygienic rewriting system is not interesting at all, because hygienic macro can be easily processed by rewriting system. So in the finally implementation of our tool, we just use PLT Redex's binding forms to deal with hygienic macros. But we did try it on the version without hygienic rewriting system.

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

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

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

are typical recursive sugars. The previous works can process this kind of syntactic sugar 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 start resugaring as Fig7 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))

...

Fig. 7. Odd2's desugaring process

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 Fig 8 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 sugar. And for syntactic sugar's feature, it is of recursive sugar. Giving the following two higher-order syntactic sugar as examples.

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

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

(Odd 2)

(Even (- 2 1))

(Even 1)

(Odd (- 1 1))

(Odd 0)

#f

Fig. 8. Odd2's resugaring sequences

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

These two syntactic sugars use different sugar forms to implement. For *Map* sugar, we use if expression in CoreLang to constrain the boundary conditions. For *Filter* sugar, we use two different parameters' form, which is another easy way for constructing syntactic sugar. The testing results show as Fig9 10.

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

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

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

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

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

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

(cons 2 (list 3 4))

(list 2 3 4)

Fig. 9. Map's resugaring sequences

3.6 Compare to previous work

As mentioned many times before, the biggest difference between previous resugaring approach and our approach, is that our approach doesn't need to desugar the sugar expresssion totally. Thus, our approach has the following advantages compared to previous work.

- **Lightweight** As the example at sec3.5.1, the match and substitution process searchs all intermediate sequences many times. It will cause huge cost for a large program. So out approach—only expanding a syntactic sugar when necessarily, is a lightweight approach.

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

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

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

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

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

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

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

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

(cons 4 (cons 5 (list)))

(cons 4 (list 5))

(list 4 5)

Fig. 10. Filter's resugaring sequences

- **Friendly to hygienic macro** Previous hygienic resugaring approach use a new data structure—abstract syntax DAG, to process resugaring of hygienic macros. Our approach simply finds hygienic error after expansion, and gets the correct reduction instead.
- **More syntactic sugar features** The ability of processing recursive sugar is a superiority compared to previous work. The key point is that recursive syntactic sugar must handle boundary conditions. Our approach handle them easily by not necessarily desugaring all syntactic sugars. Higher-order functions, as an important feature of functional programming, was supported by many daily programming languages. So the ability on higher-order sugar is important.
- **Rewriting rules based on reduction semantics** Any syntactic sugar that can expressed by reduction semantics can be used in our approach. It will give more possible forms for constructing syntactic sugars. todo:example?

4 ZC

5 RELATED WORK

The series of resugaring[Pombrio and Krishnamurthi 2014, 2015, 2018; Pombrio et al. 2017] is the most related work. The first two are about resugaring evaluation sequences, the third one is about resugaring scope rules, and the last one is about resugaring type rules. The whole series is for better syntactic sugar. We have compared our approach with existing sequences resugaring method before. The type resugaring work indicates that it is possible to automatically construct surface language's semantics. But after trying to do this by unification as type resugaring does, we found it impossible because todo.

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 similiar idea on slicing expressions and processing on subexpressions. The dynamic approach is like the forward component, so the method to handle side effects in functional programs may be useful for a better resugaring with side effects.

Macros as Multi-Stage Computations[?] is an old research similar to lazy expansion for macros. Some other researches[?] about multi-stage programming[?] indicate that it is an useful idea for implementing domain-specific languages. Macro systems in some language (such as Racket[?]) have support lazy expansion. Our dynamic approach is a combination of existing resugaring and lazy expansion, which achieves a more powerful approach.

Addition to PLT Redex[Felleisen et al. 2009] we used to engineer the semantics, there are some other semantics engineering tools[??] which aim to test or verify the semantics of languages. The methods of these researches can be easily combined with our static approach.

5.1 Comments on resugaring

5.1.1 Side effects in resugaring. The previous resugaring approach used to tried a *Letrec* sugar and found no useful sequences shown. We explain the reason from the angle of side effects. We also used to try some syntactic sugars which contain side effect. 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.

5.1.2 Hygienic resugaring. As mentioned in Sec3.5.2, our approaches can deal with hygienic resugaring without much afford as the existing approach[Pombrio and Krishnamurthi 2015]. (Of course with the help of core language's semantics, see in next discussion) The dynamic approach uses a trivial, not beautiful tricky to handle the hygienic macros, so that we decide to make the rewriting system hygienic instead. ($\# : binding - forms$ keyword in PLT Redex) But the static approach handle the hygienic macro very easily, by adding a substitution's hash table. The dynamic approach can also use this method, but a hygienic rewriting system is enough.

5.1.3 Assumption on CoreLang's evaluator. As mentioned in Sec , the work "resugaring" originated from has weaker assumption on the core language—it just required a stepper of core languages' expression, when our approach needed the whole reduction semantics. Thus, the intent of our resugaring is not a tool for supporting resugaring for languages, but a tool for implementing DSL better. We will discuss this in feature work for details.

6 CONCLUSION

In this paper, we purpose a new approach (see Fig 1) or resugaring mixed with a dynamic apporach and static approach, which has some advances compared to existing approaches. The two approaches are seemingly similar in lazy desugaring. Essentially, we would see the static approach is the abstract(todo:another express?) of dynamic approach. In the dynamic approach, the most important part is **one-step try** (see in sec 3.2), which decides whether reducing the subexpression or desugaring the outermost sugar. Reducing subexpressions are just the same as context rules in static approach; desugaring the outermost sugar is similar to reduction rules in static approach. However, the reduction rules is more convinent and efficent than dynamic resugaring, because the static approach evolves a process like abstract interpretation[?], then reduces many steps executed in core language. Moreover, the semantics got by static approach 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. Actually, the rewriting based on reduction semantics makes the sugar represented in many ways.

REFERENCES

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

Martin Fowler. 2011. *Domain-Specific Languages.* Addison-Wesley. http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0321712943,00.html

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), 361371. https://doi.org/10.1145/2666356.2594319

Justin Pombrio and Shriram Krishnamurthi. 2015. Hygienic Resugaring of Compositional Desugaring. *SIGPLAN Not.* 50, 9 (Aug. 2015), 7587. 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), 812825. https://doi.org/10.1145/3296979.3192398

Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. 2017. Inferring Scope through Syntactic Sugar. *Proc. ACM Program. Lang.* 1, ICFP, Article 44 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110288

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

Christopher Schuster, Tim Disney, and Cormac Flanagan. 2016. Macrofication: Refactoring by Reverse Macro Expansion. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632.* Springer-Verlag, Berlin, Heidelberg, 644671.

A APPENDIX

Text of appendix …