Lifting Resugaring by Lazy Desugaring

ANONYMOUS AUTHOR(S)

Syntactic sugar is a good way of implementing domain-specific languages. However, desugaring of sugar makes some information hidden. Then the programs after desugaring will be unrecognizable for people who is unfamiliar to the host language. It is not good because domain-specific languages are not always used by programmers.

Resugaring is an method to solve the problem above. In this paper, we purposed an approach of resugaring mixed with two approachesm, based on lazy desugaring—getting evaluation sequences without fully desugaring the whole syntactic sugar expression. The first approach is lightweight but powerful, which lazy desugaring the sugars in programs. The second approach is efficent, which gets inference rules of sugars, then runs the programs using new inferences rules.

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

1 INTRODUCTION

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 which should be not shown (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 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 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 by such lazy processes.

The dynamic approach uses the reduction semantics[Felleisen and Hieb 1992] of core language to decide whether desugaring the sugar. The static approach uses the reduction semantics of core

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

language to get reduction semantics of surface language based on sugars' syntax, then execute the syntactic sugar programs on the surface's semantics.

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 on resugaring.[sec 5]
- Conclusion and feature work.[sec 6]

2 OVERVIEW

In this section, we firstly show an example of traditional resugaring approach. Then we will describe the framework of our whole approach, with two different approaches which work together and their separate examples.

2.1 Existing resugaring method

This subsection is original from [Pombrio and Krishnamurthi 2014] and [Pombrio and Krishnamurthi 2015]. But their original idea is from the first one, and the second one is a optimized version on hygienic macros and rewriting system.

Defination 2.1 (Resugaring). *Given core language (named* ***CoreLang****) and its evaluator, together with surface language based on syntactic sugars of CoreLang (named* ***Surflang****). For any syntactic sugar, getting the evaluation sequences of the expression in SurfLang's syntax.It's not strict, so they use three properties for defining correctness.*

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

(1) *Emulation* Every surface term desugars to (a termisomorphic to) the core term it purports to represent.

(2) *Abstraction* If a term is shown in the reconstructedsurface evaluation sequence, then each non-atomic part of it orig-inated from the original program and has honest tags.
(3) *Coverage* A sugar with good coverage shows many steps in the reconstructed surface evaluation sequence.

It is a good summary for resugaring's properties, so we also use similar properties in our approach (using our domain).

Given an example to show how existing approach works. For syntactic sugar **and** and **or**, the sugar rules are:

$$(\textsc{And}\ e_1\ e_2)\ \rightarrow_d\ (\textsc{if}\ e_1\ e_2\ \#f)$$
$$(\textsc{Or}\ e_1\ e_2)\ \rightarrow_d\ (\textsc{if}\ e_1\ \#t\ e_2)$$

which forms a simple SurfLang.

The evaluation rules of **if** is:

$$(\textsc{if}\ \#t\ e_1\ e_2) \dashrightarrow e_1$$
$$(\textsc{if}\ \#t\ e_1\ e_2) \dashrightarrow e_2$$

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. In summary, the traditional approach firstly desugars the whole expression, then trys to transform the core sequences into surface sequences by match and substitution.

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 based on the former section. Besides sugar **and**, **or**, we add a recursive sugar **mapf** based on another new sugar **f**. The recursive sugar can be handled by the dynamic approach, but not for the static one. (Reasons in later sections)

```
(f e1 e2) ⇢ (let x e1 (or x (and e2 x)))
(mapf e lst) ⇢ (if (empty? lst) empty (cons (f e (first lst)) (mapf e (rest lst))))
```

In the mapf (map of f) sugar, we use both core language's term (such as **if, empty?, cons, let, first, rest**) and existing syntactic sugar (**and, or**). The semantics of core language is as common. But to show some exact steps, we set the term **cons** as a common expression (belonging to core language, but being displayed as surface language).

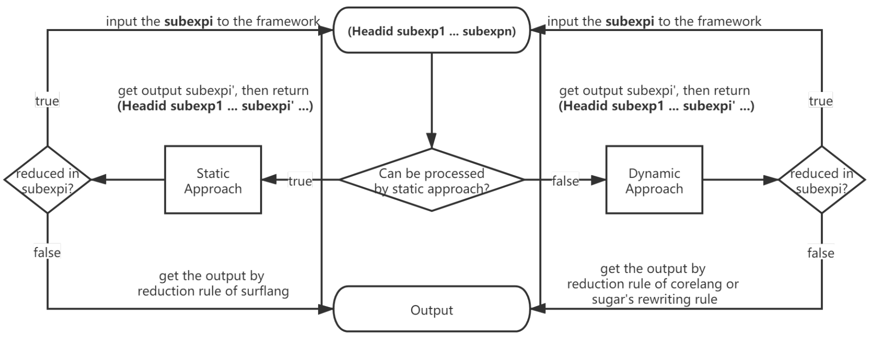
(Headid subexp1 ... subexpn)
input the **subexpi** to the framework
input the **subexpi** to the framework
get output subexpi', then return (Headid subexp1 ... subexpi' ...)
get output subexpi', then return (Headid subexp1 ... subexpi' ...)
true
true
reduced in subexpi?
Static Approach
true
Can be processed by static approach?
false
Dynamic Approach
reduced in subexpi?
false
false
get the output by reduction rule of surflang
get the output by reduction rule of corelang or sugar's rewriting rule
Output

Fig. 1. One step in framework of mixture approach

If we execute

```
(mapf #t (list #f #t))
```

the mixture approach will judge whether sugar mapf can be handle by the static approach. No, then we use the dynamic approach in one step and get the intermidiate expression.

```
    (mapf #t (list #f #t))
⟶ (cons (f #t (first (list #f #t))) (mapf #t (rest (list #f #t))))
```

Then according to semantics of **cons**, the first subexpression should be reduced. The subexpression can be handled by the static approach, so getting a subsequence.

```
    (cons (f #t (first (list #f #t))) (mapf #t (rest (list #f #t))))
⟶ (cons (f #t #f) (mapf #t (rest (list #f #t))))
⟶ (cons #t (mapf #t (rest (list #f #t))))
```

Then the second subexpression should be reduced, which is a recursive process. Finally, the subexpression (mapf #t (list)) will be processed by dynamic approach.

```
   (mapf #t (list))
⇢ (if (empty? (list)) empty ...)
⟶ empty
```

Note that there are some steps should not be displayed, we define the common expressions above in syntaxs to restrict which intermediate step should be displayed.

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 Fig2.

(and (or #f #t) (and #t #f)) $\dashrightarrow_{step1}$ (and (if #f #t #t) (and #t #f)) $\dashrightarrow_{step2}$ (and #t (and #t #f)) $\dashrightarrow_{step3}$ (if #t (and #t #f) #f) $\dashrightarrow_{step4}$ (and #t #f) $\dashrightarrow_{step5}$ (if #t #f #f) $\dashrightarrow_{step6}$ #f

Fig. 2. core-algo example

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

Fig. 3. Core and Surface Expressions

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. Note that there are some sequences which should not be shown, we use another function to filter them.

The key idea of our static 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.

$$(\textsc{or}\ e_1\ e_2) \ \longrightarrow_d\ (\textsc{let}\ x\ e_1\ (\textsc{if}\ x\ x\ e_2))$$

where the rules of *if* and *let* is the same following

$$\begin{array}{rcl}
(\textsc{if}\ \#t\ e_1\ e_2) & \longrightarrow_d & e_1 \\
(\textsc{if}\ \#t\ e_1\ e_2) & \longrightarrow_d & e_2
\end{array}$$

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 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

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

Fig. 4. An Core Language Example

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 Todo: need to be corrected. 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, CoreHead 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:

$$(\textsc{SurfHead}\ x_1\ x_2\ \ldots\ x_n)\ \longrightarrow_d\ \textsc{Exp}$$

where its LHS is a simple pattern (unnested) and its RHS is a surface expression. For instance, we may define syntactic sugar And by

$$(\textsc{And}\ x\ y)\ \longrightarrow_d\ (\textsc{if}\ x\ y\ \#f).$$

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 CoreExp, which sounds more natural. We use surfExp to be able to allow definition of recursive syntactic sugars, as seen in the following example.

$$\begin{array}{l}(\textsc{Odd}\ x)\ \longrightarrow_d\ \textsc{if}\ (>\ \ x\ 0)\ (\textsc{Even}\ (\ x\ 1))\ \#f)\\(\textsc{Odd}\ x)\ \longrightarrow_d\ \textsc{if}\ (>\ \ x\ 0)\ (\textsc{Odd}\ (\ x\ 1))\ \#t)\end{array}$$

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	::=	Core'Exp
	|	OtherSurfExp
	|	OtherCommonExp
CoreHead	::=	CoreHead'
	|	CommonHead
Core'Exp	::=	(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

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 their Head. 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 sequences. The Core'Exp terms are terms with undisplayable CoreHead (named CoreHead'. The SurfExp terms are terms with SurfHead and all sub-expressions are displayable. The CommonExp terms are terms with displayable CoreLang's Head (named CommonHead, together with displayable sub-expressions. There exists some other expression during our resugaring process, which have displayable Head, 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 (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. For expression (`CoreHead` e_1 ... e_n), replacing all subexpression not in core language with different reduciable core language's term. Then getting a result after inputting the new expression `exp'` to the original blackbox stepper. If reduction appears at subexpressions after e_i replaced by, then the stepper with the extension should return (`CoreHead` e_1 ... e_i' ... e_n), where e_i' is e_i after desugaring. (an example in Fig 6) Otherwise, stepper should return `exp'`, with all the replaced subexpressions replacing back. (an example in Fig 7) The extension will not vialate 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 s subexpression.

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

Fig. 6. e1

$$\begin{gathered}
\texttt{(if (if true ture false) (and ...) (or ...))} \\
\Downarrow_{replace} \\
\texttt{(if (if true ture false) tmpe2 tmpe3)} \\
\Downarrow_{blackbox} \\
\texttt{(if true tmpe2 tmpe3)} \\
\Downarrow_{reback} \\
\texttt{(if true (and ...) (or ...))}
\end{gathered}$$

Fig. 7. e2

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

$$\frac{(\textsc{CoreHead}\ e_1\ \ldots\ e_n)\ \to_c\ e'}{(\textsc{CoreHead}\ e_1\ \ldots\ e_n)\ \to_m\ e'} \qquad (\textsc{CoreRed})$$

$$\frac{\begin{gathered}(\textsc{SurfHead}\ x_1\ \ldots\ x_i\ \ldots\ x_n)\ \to_d\ e,\ e_i\ \to_m\ e_i'' \\ \exists i.\ e[e_1/x, \ldots, e_i/x_i, \ldots, e_n/x_n]\ \to_m\ e[e_1/x, \ldots, e_i'/x_i, \ldots, e_n/x_n]\end{gathered}}{(\textsc{SurfHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \to_m\ (\textsc{SurfHead}\ e_1\ \ldots\ e_i''\ \ldots\ e_n)} \qquad (\textsc{SurfRed1})$$

$$\frac{\begin{gathered}(\textsc{SurfHead}\ x_1\ \ldots\ x_i\ \ldots\ x_n)\ \to_d\ e \\ \neg\exists i.\ e[e_1/x_1, \ldots, e_i/x_i, \ldots, e_n/x_n]\ \to_m\ e[e_1/x_1, \ldots, e_i'/x_i, \ldots, e_n/x_n]\end{gathered}}{(\textsc{SurfHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \to_m\ e[e_1/x_1, \ldots, e_i/x_i, \ldots, e_n/x_n]} \qquad (\textsc{SurfRed2})$$

The CoreRed rule describes how our mixed language handle expressions with CoreHead—just leave it to the core language's evaluator. Then for the expression with SurfHead, we will firstly desugar the outermost sugar (identified by the SurfHead), then recursively executing $\to_m$. 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 $\rightarrow_m$.

```
desugar(e)  =  if isNormal(e) then return
               else
                   let e →_m e′ in
                   if e′ ∈ DisplayableExp
                       output(e′), desugar(e′)
                   else desugar(e′)
```

We use the DisplayableExp to restrict immediate sequences to be output or not. It is more explicit compared to existing approaches.

3.3 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.

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

Defination 3.1 (Upper and lower expression). *For Exp=(Headid $Subexp_1$ $Subexp_{\ldots}$ ...), Exp is called **upper expression**, $Subexp_i$ is called **lower expression**.*

Case 2, 4, 6 in the core algorithm are of outer reduction. And case 3 or 5 are of surface reduction if the reduced subexpression is processed by outer reduction, or they are of inner reduction. What we need to prove is that all the 6 cases of core algorithm core-algo satisfy the properties. Case 1 and case 2 won't effect any properties, because it does what CoreLang should do.

Defination 3.2 (Emulation). *For* Exp=(SurfHead $e_1\ \ldots\ e_i\ \ldots\ e_n$),
if Exp $\rightarrow_m$ Exp' *and* Desugar(Exp)≠Desugar(Exp'), *then* Desugar(Exp) $\rightarrow_c$ Desugar(Exp')

Lemma 3.1. *For* Exp=(SurfHead $e_1\ \ldots\ e_i\ \ldots\ e_n$),
if inputting Desugar(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. For (SurfHead $x_1\ \ldots\ x_i\ \ldots\ x_n$) $\rightarrow_d$ e

if e is of normal form, the Desugar(Exp) will not be reduced by core evaluator.

if e is headed with CoreHead, then according to the CoreRed rule, the $\rightarrow_c$ will execute on e, 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 Emulation.
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 Desugar(e_i)=Desugar(e_i''), then Desugar(Exp)=Desugar(Exp'). If not, what we need to prove is that, Desugar(Exp) $\rightarrow_c$ Desugar(Exp'). Note that the only difference between Exp and Exp' is the i-th subexpression, and we have proved the lemma 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 Desugar(Exp)=Desugar(Exp'). □

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

Lemma 3.2. *If no syntactic sugar desugared before necessary (if the sugar not desugared, the expression of mixed language cannot be reduced after other sugars desugared, then coverage property is satisfied.*

Proof of Lemma3.2. Assume that no syntactic sugar not necessarily expanded desugars too early, existing an expression in CoreLang

Exp = (Head e_1 $\ldots$ e_i $\ldots$ e_n) which can be resugared to

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

- Or existing
  $ResugarExp$=($Surf id$ $Subexp'_1$ $\ldots$ $Subexp_i$ $Subexp'_{\ldots}$ $\ldots$) 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
  $ResugarExp$=($Surf id'$ $\ldots$ $ResugarExp'$ $\ldots$) 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.
- Or though the Exp exists, it doesn't from $ResugarExp$.

□

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

For case 3 and 5, the reduction occurs in sub-expression of Exp. So if applying core algorithm core-algo on the subexpression doesn't desugar syntactic sugars not necessarily expanded, then this two cases don't. If the reduction is surface reduction, then the reduction of the subexpression is processed by case 2, 4 or 6, which don't desugar sugars not necessarily expanded; if the reduction is inner reduction, then it's another recursive proof as emulation. So in these two cases, the core-algo only desugar the sugar which has to be desugared. □

3.4 Implementation

Our lightweight 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 whole framework is as Fig8.

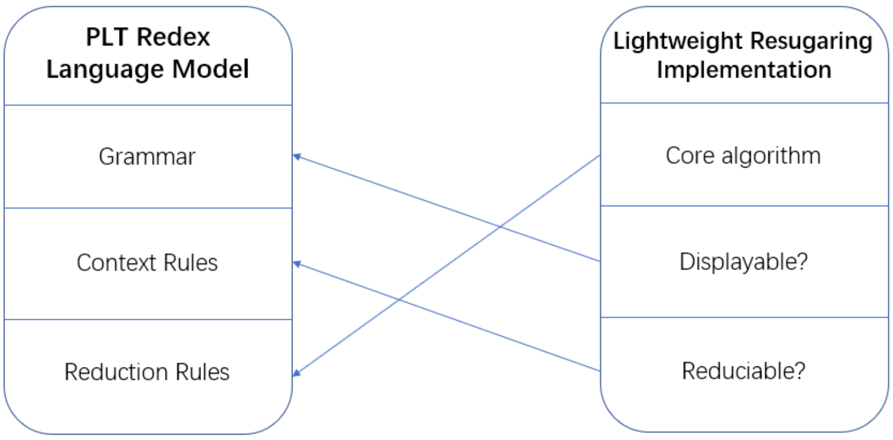

Fig. 8. framework of implementation

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.

3.5 Application

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 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 lightweight.

```
S --> (lamN (x1 x2 x3) (x1 x2 (x1 x3)))
K --> (lamN (x1 x2) x1)
I --> (lamN (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

```
((lamN
   (x_1 x_2 x_3)
   (x_1 x_3 (x_2 x_3)))
  ((lamN (x_1 x_2) x_1)
   ((lamN
     (x_1 x_2 x_3)
     (x_1 x_3 (x_2 x_3)))
    (lamN (x) x)))
  (lamN (x_1 x_2) x_1)
  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. It will omit more steps for a larger expression.

So the unidirectional resugaring algorithm makes our approach lightweight, because no attempts for resugaring the expression take place.

3.5.2 hygienic macro. The second work[Pombrio and Krishnamurthi 2015] mainly processes hygienic macro 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 macro without special data structure.

A typical hygienic example is as the example origined from Hygienic resugaring[Pombrio and Krishnamurthi 2015]. We simplify the example to the following one.

```
(Hygienicadd e1 e2) ⇢ (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)))`, but it is awful because the two x in `(+ x x)` should be bind to different value. But for our lazy desugaring, the HYGIENICADD sugar does not have to desugar until necessary, so, getting following sequences.

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

The lazy desugaring is also comvinent for hygienic resugaring for non-hygienic core language. 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,

$$(Let\ e_1\ e_2\ e_3)\ \longrightarrow_d\ (let\ ((e_1\ e_2))\ e_3)$$

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. But our result shows lazy desugaring is really a good way to handle hygienic macro 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,

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

are typical recursive sugars. The existing resugaring approach can't 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 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 sugar. And many higher-order sugars should be constructed by recursive defination. The first sugar is FILTER, implemented by pattern matching term rewriting.

```
    (filter e (list v1 v2 ...))
--> (if (e v1) (cons v1 (filter e (list v2 ...))) (filter e (list v2 ...)))
    (filter e (list)) --> (list)
```

and getting the following result. (by making `(lam ...)` COMMONEXP)

```
     (filter (lam (x) (and (> x 1) (< x 4))) (list 1 2 3 4))
⟶  (filter (lam (x) (and (> x 1) (< x 4))) (list 2 3 4))
⟶  (cons 2 (filter (lam (x) (and (> x 1) (< x 4))) (list 3 4)))
⟶  (cons 2 (cons 3 (filter (lam (x) (and (> x 1) (< x 4))) (list 4))))
⟶  (cons 2 (cons 3 (filter (lam (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 rebundant. 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 conditions. Take the example of another higher-order sugar MAP as an example.

```
     (map e1 e2)
--> (let ((x e2)) (if (empty? x) (list) (cons (e_1 (first x)) (map e_1 (rest x)))))
```

Get following resugaring sequences.

```
     (map (lam (x) (+ x 1)) (cons 1 (list 2)))
⟶  (map (lam (x) (+ x 1)) (list 1 2))
⟶  (cons 2 (map (lam (x) (+ 1 x)) (list 2)))
⟶  (cons 2 (cons 3 (map (lam (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 the inline boundary conditions by rewriting system. But our approach is easy to handle cases like this.

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.

- *L*ightweight 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.
- *F*riendly 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.
- *M*ore syntactic sugar features The ability of processing non-pattern-based (Todo: inline?) 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. Lazy desugaring makes writing higher-order sugars easier.

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[Ganz et al. 2001] is an old research 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. Macro systems in some language (such as Racket[Flatt 2012]) 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[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 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 **??**), 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[Cousot and Cousot 1977], 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. Besides, the rewriting based on reduction semantics makes the sugar represented in many ways.

As we mentioned before, the original intent of our research is finding a better method (or building a tool) for implementing DSL. We could see static approach 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 in the static approach can be solved. So the first future work may be achieving a more powerful static approach as our dynamic approach. Then we will carefully design a core language for as the host language of our dream system and find a better type resugaring approach for the system. Finally, a general optimazation method for DSL in our system is needed.

REFERENCES

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

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

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

Matthew Flatt. 2012. Creating Languages in Racket. *Commun. ACM* 55, 1 (Jan. 2012), 48–56. https://doi.org/10.1145/2063176.2063195

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

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

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

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 ...