Resugaring by Lazy Desugaring

Anonymous Author(s)

Abstract

We propose a novel approach to resugaring, which is a technique to get the surface evaluation sequences of syntactic sugars, by lazy desugaring. We recognize the condition for lazy desugaring, and propose a reduction strategy for the language mixed by the surface language and the core language, which can produce the correct resugaring evaluation sequence. We have implemented a system based on this new approach. The result shows that our new approach is more efficient compared to the existing approach, and the lazy desugaring also provides powerful expressiveness even for a simple desugaring so that it can also deal with the common hygienic sugars and flexible recursive sugars.

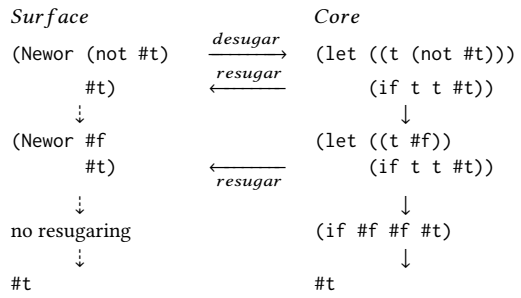
Keywords: Resugaring, Syntactic Sugar, Interpreter, Domain-Specific Language, Reduction Semantics

1 Introduction

Syntactic sugar, first coined by Peter J. Landin in 1964 [14], 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 been proved to be very useful for defining domain-specific languages (DSLs) and extending languages [2, 5]. Unfortunately, when syntactic sugar is eliminated by transformation, it obscures the relationship between the user's source program and the transformed program.

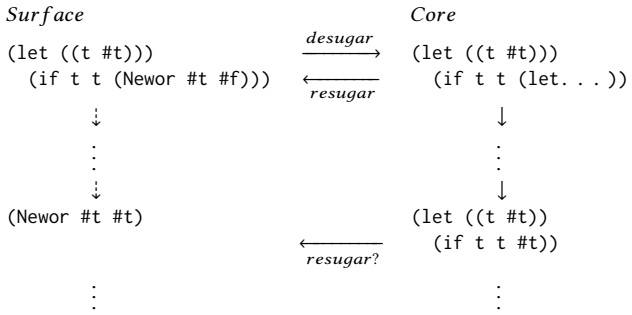
Resugaring is a powerful technique to resolve this problem [15, 16]. It can automatically convert the evaluation sequences of desugared program in the core language into representative sugar's syntax in the surface language. Just like the existing approach, it is natural to try matching the expressions after the desugared with syntactic sugars' dusugaring rules to reversely expand the sugars—that why it was named "resugaring". Here we use the sugar `Newor` to see how the existing resugaring approach works, with following desugaring rule. (Considering the `not` as a core language's constructor)

`(Newor` e_1 e_2`)` $\rightarrow_d$ `(let ((t` e_1`)) (if t t` e_2`))`



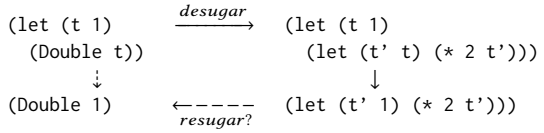
But it is not as easy as the example above. Sometimes the program in the core language contains a desugared form of a syntactic sugar, but the form may belong to the original (not desugared) program. In the following example, the reverse expansion of sugar should be noticed.

`(Newor` e_1 e_2`)` $\rightarrow_d$ `(let ((t` e_1`)) (if t t` e_2`))`



Moreover, when meeting hygienic sugar, the simple match and substitution will not work as the following example shows.

`(Double` e_1`)` $\rightarrow_d$ `(let (t` e_1`) (* 2 t))`



If we use binder renaming for solving the case as above, some other information is needed (such as the permutation `t` $\rightarrow$ `t'`).

The existing resugaring approaches subtly solved the problems above by "tagging"[15] and "abstract syntax DAG"[16]. While those techniques successfully make the resugaring method usable, there is a key point which makes the existing approaches not very practical—the reverse expansion of sugars needs to match the desugared expressions to see if it is able to resugar. It is quite a huge job when the program contains many syntactic sugars or some syntactic sugars

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

can be desugared to large sub-expressions. Also, as the debugging for surface languages can be a good application of resugaring, the efficiency matters.

In this paper, we propose a novel approach to resugaring, which does not use reverse desugaring at all. The key idea is "lazy desugaring", in the sense that desugaring is delayed so that the reverse application of desugaring rules becomes unnecessary. Rather than assuming a black-box stepper for the core language as the existing approach, our resugaring approach is based on the evaluation rules (consist of the context rules and reduction rules) of the core language. So our approach seems to be a meta-level language feature rather than a tool for existing languages. (Although it can also work with a black-box stepper, as we will discuss in Section 5.1) To this end, we consider the surface language and the core language as a whole language. We regard the desugaring rules as the reduction rules of surface language and calculate the context rules of surface language to see how lazy the expansion of sugar expressions can be. Then the intermediate evaluation steps of the mixed language will contain the resugaring evaluation sequences of a program.

Our main technical contributions can be summarized as follows.

- We propose a novel approach to resugaring by lazy desugaring, where the reverse application of desugaring rules becomes unnecessary. We recognize an algorithm to calculate the computational orders (limited by context rules) for syntactic sugars, and propose a reduction strategy, based on evaluator of the core languages and the desugaring rules together with the context rules of syntactic sugar, which is sufficient to produce all necessary resugared expressions on the surface language. We prove the correctness of our approach.
- We have implemented a system based on the new resugaring approach. It is much more efficient than the existing approach, because it completely avoids unnecessary complexity of the reverse desugaring. It also provides extra expressiveness for even the simple rewriting so that it can also deal with the common hygienic sugars and flexible recursive sugars. All the examples in this paper have passed the test of the system.
- We discuss how lazy desugaring makes sense, including how the approach can be extended to a model with a black-box stepper, how it can easily deal with hygiene, and how we deal with the trade-off of our approach's properties.

The rest of our paper is organized as follows. We start with an overview of our approach in Section 2. We then discuss the core of resugaring by lazy desugaring in Section 3. We describe our experiment and evaluation in Section 4. We discuss some other issues in Section 5, and discuss related work in Section 6, and conclude the paper in Section 7.

Note that there are two pairs of similar words in the whole paper.

1. *program* v.s. *expression*;
2. *desugar/desugaring* v.s. *expand/expansion.*

We use the word *program* if an expression is the input program of resugaring process, while using *expression* on else. We use the word *expand/expansion* when discussing desugaring on a syntactic sugar, while using *desugar/desugaring* when discussing desugaring on an expression or else.

2 Overview

In this section, we give a brief overview of our approach. To be concrete, we will consider the following simple core language, defining boolean expressions together with `if` construct:

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

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

$$
\begin{array}{lcl}
C & := & \texttt{(if C } e\ e\texttt{)} \\
& | & [\bullet]\text{//evaluation context's hole}
\end{array}
$$

and two reduction rules (the letter c means core):

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

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

$$
\begin{aligned}
\texttt{(And } e_1\ e_2\texttt{)} &\rightarrow_d \texttt{(if } e_1\ e_2\ \#f\texttt{)} \\
\texttt{(Or } e_1\ e_2\texttt{)} &\rightarrow_d \texttt{(if } e_1\ \#t\ e_2\texttt{)}
\end{aligned}
$$

Now let us demonstrate how to execute `(And (Or #t #f) (And #f #t))`, and get the resugaring sequences as follows by our approach.

```
  (And (Or #t #f) (And #f #t))
⟶ (And #t (And #f #t))
⟶ (And #f #t)
⟶ #f
```

We propose a new resugaring approach by eliminating "reverse desugaring" via "lazy desugaring", where a syntactic sugar will be expanded only when it is necessary. While giving the evaluation rules of the core language, we can figure out the following context rules of the surface language. From the context rules of `if`, we can find that the condition (e_1) is always evaluated first. Therefore, for expression `(And` e_1 e_2`)` defined by syntactic sugar, e_1 is also evaluated first, which is the context rule of `And`. Similarly, we can calculate the context rule of `Or`.

$$
\begin{array}{lcl}
C & := & \texttt{(And } C\ e\texttt{)} \\
& | & \texttt{(Or } C\ e\texttt{)} \\
& | & [\bullet]
\end{array}
$$

Then we mix the surface language and the core language as Fig. 1, where CommonExp means the expressions used in the surface language and the core language both, and $\rightarrow_m$ is a one-step evaluation in our mixed language.

e	::=	CoreExp
	|	SurfExp
	|	CommonExp
CoreExp	::=	(if e e e)
SurfExp	::=	(And e e)
	|	(Or e e)
CommonExp	::=	#t
	|	#f

(a) Syntax

C	:=	(if C e e)
	|	(And C e)
	|	(Or C e)
	|	[•]

(b) Context Rules

$$(\text{And } e_1\ e_2) \rightarrow_m (\text{if } e_1\ e_2\ \#f)$$
$$(\text{Or } e_1\ e_2) \rightarrow_m (\text{if } e_1\ \#t\ e_2)$$
$$(\text{if } \#t\ e_1\ e_2) \rightarrow_m e_1$$
$$(\text{if } \#f\ e_1\ e_2) \rightarrow_m e_2$$

(c) Reduction Rules

Figure 1. Mixed Language Example

Finally the program (And (Or #t #f) (And #f #t)) will get the evaluation sequence as follows in the mixed language. We can just filter the intermediate sequences without Coreexp in any sub-expressions to get the resugaring sequence above.

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

Note that the context rules should restrict the computational order of a sugar expression's sub-expressions, thus we should let the context rules be correct—reflecting what should be executed in the desugared expression. Also, as the goal of resugaring is to present the evaluation of sugar programs, we clearly defined which expression should be outputted. For the example in this section, of course, the sugar And and Or should be outputted, and also the boolean expressions should be. So we set boolean expressions as CommonExp, so that they can be displayed though they are of the core language. By clearly separating what should be displayed, we can always get the resugaring evaluation sequences we need. (Slightly different from the existing approach's setting, as we will discuss in Section 5.2.)

CoreExp	::=	x	variable
	|	c	constant
	|	(CoreHead CoreExp_1 ... CoreExp_n)	constructor
SurfExp	::=	x	variable
	|	c	constant
	|	(SurfHead SurfExp_1 ... SurfExp_n)	sugar constructor

Figure 2. Core and Surface Expressions

3 Resugaring by Lazy Desugaring

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

3.1 Mixed Language for Resugaring

As a preparation for our resugaring algorithm, we define a mixed language that combines a core language with a surface language (defined by syntactic sugars over the core language). An expression in this language is reduced step by step by the evaluation rules for the core language and the desugaring rules for the syntactic sugars in the surface language. Our approach assumes the evaluation is compositional (as the definition in [16]), that is, for evaluation contexts E_1 and E_2, $E_1[E_2]$ is also a evaluation context.

3.1.1 Core Language. The evaluator of our core language is driven by evaluation rules (context rules and reduction rules), with three natural assumptions. First, the evaluation is deterministic, in the sense that any expression in the core language will be reduced by a unique reduction sequence (restricted by context rules). Second, evaluation of a sub-expression has no side-effect on other parts of the expression. Third, the context rules have no conditions, which means the rules as following are not permitted.

C	::=	(notif [•] e_2 e_3)
	|	(notif v_1 [•] e_3), ($side-condition$ ($equal?$ v_1 #t))
	|	(notif v_1 e_2 [•]), ($side-condition$ ($equal?$ v_1 #f))

The expression form of the core language is defined in Fig. 2. It is a variable, a constant, or a (language) constructor expression. Here, CoreHead stands for a language constructor such as if and let. To be concrete, we will use a simplified core language defined in Fig. 3 to demonstrate our approach. Here the [e/x] is a capture-avoiding substitution.

```
CoreExp  ::=  (apply CoreExp CoreExp ...)
           |  (if CoreExp CoreExp CoreExp) // condition
           |  (let ((x CoreExp) ...) CoreExp) // binding
           |  (listop CoreExp) // first, rest, empty?
           |  (cons CoreExp CoreExp) // data structure of list
           |  (arithop CoreExp CoreExp) // +, -, *, /, >, <, =
           |  x // variable
           |  value
value    ::=  (λ (x ...) CoreExp) // call-by-value
           |  c // boolean, number and list
```

(a) Syntax

```
C  ::=  (apply value ... C CoreExp ...)
     |  (if C CoreExp CoreExp)
     |  (let ((x value) ... (x C) (x CoreExp) ...) CoreExp)
     |  (listop C)
     |  (cons C CoreExp)
     |  (cons value C)
     |  (arithop C CoreExp)
     |  (arithop value C)
     |  [•]
```

(b) Context Rules

$$
\begin{array}{lcl}
((\lambda\ (x_1\ x_2\ \ldots)\ \text{CoreExp})\ \text{value}_1\ \text{value}_2\ \ldots) & \rightarrow_c & ((\lambda\ (x_2\ \ldots)\ \text{CoreExp}[\text{value}_1/x_1])\ \text{value}_2\ \ldots) \\
(\text{if}\ \#t\ \text{CoreExp}_1\ \text{CoreExp}_2) & \rightarrow_c & \text{CoreExp}_1 \\
(\text{if}\ \#f\ \text{CoreExp}_1\ \text{CoreExp}_2) & \rightarrow_c & \text{CoreExp}_2 \\
(\text{let}\ ((x_1\ \text{value}_1)\ (x_2\ \text{value}_2)\ \ldots)\ \text{CoreExp}) & \rightarrow_c & (\text{let}\ ((x_2\ \text{value}_2)\ \ldots)\ \text{CoreExp}[\text{value}_1/x_1]) \\
\ldots & &
\end{array}
$$

(c) Part of Reduction Rules

Figure 3. A Core Language's 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 constants and variables. The expression forms of surface language is given in Fig. 2. To separate surface language's Head from core language's, we capitalize the first letter of surface language's Head.

Here we just assume a simple desugaring system for a syntactic sugar expression. We will show how this approach can be combined with other complex desugaring. A syntactic sugar is defined by a desugaring rule in the following form,

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

where its left-hand-side (LHS) is a pattern and its left-hand-side (RHS) is an expression of the surface language or the core language. The LHS can be nested or not, that means, we can write sugars like (SurfHead (e_1 (e_2 e_3)) ... e_n). And any pattern variable (e.g., e_1) in LHS only appears once in RHS. For instance, we may define syntactic sugar And by

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

And if we need to use a pattern variable multiple times in RHS, a let binding may be used (a normal way in syntactic sugar). We take the following sugar as an example

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

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

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

```
Exp             ::=  DisplayableExp
                  |  MixedExp

DisplayableExp  ::=  (SurfHead DisplayableExp ...)
                  |  (CommonHead DisplayableExp ...)
                  |  c
                  |  x

MixedExp        ::=  (SurfHead MixedExp ...)
                  |  (CoreHead MixedExp ...)
                  |  c
                  |  x
```

Figure 4. Our Mixed Language

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

As described above, we only assume the desugaring is a transformer without other helper function, thus there will be many kinds of ill-formed syntactic sugar which cannot desugared well (just as the Odd, Even sugars above, although can be processed by our lazy desugaring), or the semantics of the sugar cannot be defined clearly.

We assume that all desugaring rules are not overlapped in the sense that for any syntactic sugar in an 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, described in Fig. 4. The differences between expressions in

our core language and those in our surface language are identified by their Head. But there may be some expressions in the core language which are also used in the surface language for convenience, or to say, we need some core language's expressions to help us get better resugaring sequences. So we take CommonHead as a subset of the CoreHead, which can be displayed in resugaring sequences (just as the CommonExp in our Section 2). Then if any sub-expressions in an expression contain no CoreHead except for CommonHead, we should let them display during the evaluation process (named DisplayableExp). Otherwise, the expression should not be displayed. We just use a MixedExp expression to present the expressions which are not necessarily displayed for concision. We will discuss more on DisplayableExp in Section 5.

As an example, for the core language in Fig. 3, we may assume arithop, λ (call-by-value lambda calculus), cons as CommonHead, if, let, listop as CoreHead but out of CommonHead. This will allow arithop, λ and cons to appear in the resugaring sequences, and thus display more useful intermediate steps during resugaring.

Note that some expressions with CoreHead contain sub-expressions with SurfHead, they are of CoreExp but not in the core language. In the mixed language, we process these expressions by the context rules of the core language, so that the reduction rules of the core language and the desugaring rules of surface language can be mixed as a whole (the $\rightarrow_c$ in former section). For example, suppose we have the context rule of if expression[1]

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

then if e_1 is a reducible expression in the core language, it will be reduced by the reduction rule in the core language: if e_1 is a SurfExp, it will be reduced by the desugaring rule of e_1's Head; if e_1 is also a CoreExp which has one or more non-core sub-expressions, a recursive reduction by $\rightarrow_c$ is needed.

3.2 Resugaring Algorithm

Our resugaring algorithm works on the mixed language, based on the evaluation rules of the core language and the desugaring rules for defining the surface language. The process of getting the resugaring sequence contains two separate parts.

1. Calculating the context rules of syntactic sugars.
2. Filtering DisplayableExp during the execution of the mixed language.

The algorithm for the first part, calcontext, is described as Algorithm 1. It works as follows. For sugar

$$(\texttt{SurfHead}\ e_1\ e_2\ \ldots\ e_n) \rightarrow_d (\texttt{Head}\ \ldots\ e_1'\ e_2'\ \ldots\ e_m')$$

[1]It is another presentation of (if [•] e e), we use this form here for convenience.

Algorithm 1 calcontext

Input:
currentLHS = (SurfHead t_1 t_2 ... t_n)
//where t_i is e or v(value).
currentContext = (Head ... e_1' e_2' ... e_m')
//where e_i' can be at any depth of sub-expressions.
currentIncal = {...} //list of contexts in calculation.
Output:
ListofRule

```
1:  Let ListofRule = {}, tmpLHS = currentLHS, InCal
    = append(currentIncal, SurfHead)
2:  if ∄ contexts rules of Head then
3:    if ∃ Head in InCal then
4:      return error
5:    else
6:      ListofRule = append(ListofRule,
7:          calcontext(Head.LHS,Head.RHS,InCal))
8:    end if
9:  end if
10: Let OrderList = {e'_i, e'_j, ...}
    //RHS's computational order got by context rules
11: for subExp in OrderList do
12:   if ∃i, s.t.e_i = subExp then
13:     ListofRule= append(ListofRule,
14:         tmpLHS[[•]/e_i])
15:   else
16:     Let recRule, recLHS = calcontext(
17:         tmpLHS, subExp, Incal)
18:     tmpLHS = recLHS
19:     ListofRule = append(ListofRule, recRule)
20:     break//means the RHS has to be destroyed.
21:   end if
22: end for
23: return ListofRule, tmpLHS
```

we should run calcontext(LHS, RHS, {}), and add the context rules to the mixed language.

If Head is a CoreHead, for each context rule of the Head in order, we should just recursively make context rules for each hole, until a whole sub-expression is iterated. See examples in Fig. 5 and 6. For Sg1, the sugar does not have to be expanded until all four sub-expressions are reduced to value. But for Sg2, once the sub-expressions e_1 and e_2 are reduced to value, the sugar has to be expanded, because if being desugared to the core language, the sub-expression (+ v_3 v_4) will be reduced, then the sugar form of RHS is destroyed.

Or if the Head is a SurfHead with its context rules calculated, then we regard it as CoreHead. If it is without context rules, we will try calculating its context rules first. However, if the recursive process has tried calculating it, it will be an ill-formed recursive sugar, as the following example shows.

$$(\texttt{Odd}\ e) \rightarrow_d (\texttt{Even}\ (-\ e\ 1))$$

$$(\texttt{Sg1}\ e_1\ e_2\ e_3\ e_4) \rightarrow_d (+\ e_1\ (+\ e_2\ (+\ e_3\ e_4)))$$

$OrderList = \{e_1, (+\ e_2\ (+\ e_3\ e_4))\}$ (depth1, getting a rule $(\texttt{Sg1}\ [\bullet]\ e_2\ e_3\ e_4)$)

$OrderList = \{e_2, (+\ e_3\ e_4)\}$ (depth2, getting a rule $(\texttt{Sg1}\ v_1\ [\bullet]\ e_3\ e_4)$)

$OrderList = \{e_3, e_4\}$ (depth3, getting rules $(\texttt{Sg1}\ v_1\ v_2\ [\bullet]\ e_4), (\texttt{Sg1}\ v_1\ v_2\ v_3\ [\bullet])$)

Figure 5. Example1 of algorithm `calcontext`

$$(\texttt{Sg2}\ e_1\ e_2\ e_3\ e_4) \rightarrow_d (+\ (+\ (+\ e_1\ e_2)\ e_3)\ e_4)$$

$OrderList = \{(+\ (+\ e_1\ e_2)\ e_3), e_4\}$ (depth1)

$OrderList = \{(+\ e_1\ e_2), e_3\}$ (depth2)

$OrderList = \{e_1, e_2\}$ (depth3, getting rules $(\texttt{Sg2}\ [\bullet]\ e_2\ e_3\ e_4), (\texttt{Sg2}\ v_1\ [\bullet]\ e_3\ e_4)$)

Figure 6. Example2 of algorithm `calcontext`

$$(\texttt{Even}\ e) \rightarrow_d (\texttt{Odd}\ (-\ e\ 1))$$

The reason we can calculate the context rules of a syntactic sugar is: for a sugar's *RHS* like $(\texttt{Head}\ \ldots\ e'_1\ e'_2\ \ldots\ e'_m)$, if any reduction happens, the reduction will be on e_i of the sugar's *LHS* or the reduction will destroy the *RHS*'s form. So we can trace the computational order until the sugar form has to be destroyed.

After calculating all context rules, we can add them to the mixed language's context rule; we can also add the desugaring rules to the mixed language's reduction rule as what we showed in Section 2.

As the second part of the whole process, our resugaring algorithm can be defined based on evaluation rules of the mixed language. Let $\rightarrow_m$ be one-step reduction in the mixed language.

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

During the resugaring, we just apply the reduction ($\rightarrow_m$) on the input program step by step until no reduction can be applied (`isNormal`), while outputting the intermediate expressions that belong to `DisplayableExp`.

3.3 Correctness

We give following properties to describe the correctness of our resugaring approach.

Defination 3.1 (Fully desugar). *The function that recursively desugars any expressions of the mixed language is defined as Fig. 7.*

```
D(value) = value
D(x) = x
D((CoreHead e1 e2 ...)) =
  (CoreHead D(e'1) D(e'2) ...)
D((SurfHead e1 e2 ...)) =
  (CoreHead D(e'1) D(e'2) ...)
where (SurfHead e1 e2 ...) →d (CoreHead e'1 e'2 ...)
```

Figure 7. Defination of "fully desugar"

An expression `P` can be fully desugared if `D(P)` is terminable. We use `E[P]` to present a evaluation context `E`'s hole filled with `P`. And as the evaluation rules of the mixed language defined, there are only two kinds of reduction—desugaring on an expression headed with `SurfHead`; reduction on an expression headed with `CoreHead`. The fully desugaring of the evaluation context is also the same form, by following desugar rules of evaluation context.

Defination 3.2 (Desugaring rule of evaluation context). *For syntactic sugar S*

$$(\texttt{SurfHead}\ e_1\ e_2\ \ldots\ e_n) \rightarrow_d (\texttt{Head}\ \ldots\ e'_1\ e'_2\ \ldots\ e'_m)$$

and evaluation context $\texttt{C} = \texttt{S}.LHS[[\bullet]/e_i]$, *where* $[\bullet]$ *is at* e_i*'s location, then*

$$\texttt{C} \rightarrow_d \texttt{S}.RHS[[\bullet]/e_i]$$

Property 3.1. *For a program* `P=E[S]` *of the mixed language which can be fully desugared, where* `P'=D(P)=E'[C]`, *and* `S=(SurfHead e1 ... en)` *in the program* `P` *together with* `E'=D(E)` *(then of course* `C=D(S)`*); if* `E[S]` $\rightarrow_m$ `E[S']` *and* `S` $\rightarrow_d$ `S'`, *then* `E'[C]` $\rightarrow_c$ `E'[C']` *together with destroying the sugar's RHS form of* `S` *by* `C` $\rightarrow_c$ `C'`. *An example in Fig. 8.*

P/P'	E/E'	S/C
(And (And #t #f) #f)	(And [•] #f)	(And #t #f)
(if (if #t #f #f) #f #f)	(if [•] #f #f)	(if #t #f #f)

(And (And #t #f) #f) $\rightarrow_m$ (And (if #t #f #f) #f),
(if (if #t #f #f) #f #f) $\rightarrow_c$ (if #f #f #f),
so S'=(if #t #f #f), C'=#f; thus the $\rightarrow_c$ destroyed the sugar form of (And #t #f).

Figure 8. Example of property 3.1

Property 3.2. *For a program* `P=E[CC]` *of the mixed language which can be fully desugared, where* `P'=D(P)=E'[C]`, *and* `CC=(CoreHead e1 ... en)` *in the program* `P` *together with* `E'=D(E)` *(then of course* `C=D(CC)`*); if* `E[CC]` $\rightarrow_m$ `E[CC']` *reduced by the* `CoreHead`*'s reduction rule on* `CC`, *then for* `E'[C]` $\rightarrow_m$ `E'[C']`, *it also reduced by the* `CoreHead`*'s reduction rule on* `C`. *An example in Fig. 9.*

P/P'	E/E'	C/CC
(if (if #t (And #t #f) #t) #f #f)	(if [•] #f #f)	(if #t (And #t #f) #t)
(if (if #t (if #t #f #f) #t) #f #f)	(if [•] #f #f)	(if #t (if #t #f #t) #f)

(if (if #t (And #t #f) #t) #f #f) $\rightarrow_m$ (if (And #t #f) #f #f),
(if (if #t (if #t #f #f) #t) #f #f) $\rightarrow_c$ ((if (if #t #f #f) #f #f),
so C'=(And #t #f), CC'=(if #t #f #f); thus C and CC are both reduced by if's reduction rule.

Figure 9. Example of property 3.1

The properties limit the laziness of our mixed language—the resugaring sequences should behave as the sequences after desugared to the core language.

Proof sketch in Appendix.

3.4 Combining with Other Desugaring

We describe a resugaring approach based on a simple transformer. What if we need a more complex transformer system to describe more complex semantics for syntactic sugar? For example, we may need a type system for checking; we may specify the binding of syntactic sugar for more general hygiene (We solved the hygiene in our core language without specific the binding, as described in Section 5.3); we may use some other functions to help the desugaring. All of these extensions are possible as long as following some conditions. We summarize the conditions as follows.

1. *Compositional*: Generally speaking, the desugaring order and should not affect the semantics of a sugar expression. Otherwise, the lazy desugaring will not be correct.
2. *Unique Computational Order*: For any rules of syntactic sugar, the context rules should limit an expression to have only one computational order. Otherwise, the algorithm `calcontext` will not be deterministic.
3. *Clear Semantic*: If a syntactic sugar's desugaring rule is ambiguous or wrong, the algorithm `calcontext` may go wrong.

We find a possible problem that if we need a desugaring rule like follows,

$$\texttt{(Sugar } e_1\ e_2\ \ldots\ e_n\texttt{)} \rightarrow_d \texttt{(if (Helper } e_1\ e_2\texttt{) ...)}$$

where `Helper` is an external function, that means we don't have the evaluation rules of `Helper`. In this case, we have to force the expansion of sugar expressions headed by `Sugar`. We describe how to force the desugaring in Section 4.1.1.

4 Experiment

In this section, we present several examples to show different aspects of our approach.

4.1 Case Study on Expressiveness

We have implemented our resugaring approach using PLT Redex [4], which is a semantic engineering tool based on reduction semantics [6]. We show several case studies to demonstrate the power of our approach. Some examples we will discuss in this section are in Fig. 10. Note that we set call-by-value lambda calculus as terms in `CommonExp`, because we want to output some intermediate expressions including lambda calculus in some examples. It's easy if we want to skip them.

4.1.1 Simple Sugars.

We construct some simple syntactic sugars and try it on our tool. Some sugars are inspired by the first work of resugaring [15]. Take an SKI combinator syntactic sugar as an example. We can regard `S` as an expression headed with S, without sub-expression. And for showing a concise result, we add the call-by-need lambda calculus in the core language for this example.

$$\begin{aligned}
&\texttt{S} \rightarrow_d (\lambda_N\ (x_1\ x_2\ x_3)\ (x_1\ x_2\ (x_1\ x_3)))\\
&\texttt{K} \rightarrow_d (\lambda_N\ (x_1\ x_2)\ x_1)\\
&\texttt{I} \rightarrow_d (\lambda_N\ (x)\ x)
\end{aligned}$$

Although SKI combinator calculus is a reduced version of lambda calculus, we can construct combinators' sugars based on call-by-need lambda calculus in our core language. For the sugar program `(S (K (S I)) K xx yy)`, we get the resugaring sequences as Fig. 10a. Here the sugars contain no sub-expression, then the sugar should just desugar to the core expression. It is interesting that the sugar without sub-expressions (written by lambda calculus) and the sugar with sub-expressions will behave differently. For example, in this case, we can write the sugar as `(S e e e)` and so on, then the sugar may not have to be expanded. Moreover, we can use this difference to force a syntactic sugar desugared using call-by-need lambda calculus. See if we want a sugar `ForceAnd` which does not want to use the context rules of `if` to getting the resugaring sequence, we can just write the following sugar.

$$\texttt{ForceAnd} \rightarrow_d (\lambda_N\ (x_1\ x_2)\ \texttt{(if } x_1\ x_2\ \texttt{\#f))}$$

4.1.2 Hygienic Sugars.

The second work [16] of existing resugaring approach mainly processes hygienic sugar compared to the first work. It uses a DAG to represent the expression. However, hygiene is not hard to be handled by our lazy desugaring strategy. Our algorithm can easily process hygienic sugar without a special data structure. A typical hygienic problem is as the following example.

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

For the 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 bound to different values. So the existing hygienic resugaring approach uses abstract syntax DAG to distinct different `x` in the desugared expression. But for our approach based on lazy desugaring, the `Hygienicadd` sugar does not have to expand until necessary, thus, getting resugaring sequences as Fig. 10b based on

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

(a) Example of SKI

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

(b) Example of `Hygienicadd`

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

(c) Example of `Odd` and `Even`

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

(d) Example of `Map`

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

(e) Example of `Filter`

Figure 10. Resugaring Examples

a non-hygienic transformer system. We will discuss hygienic problem in Section 5.3.

In practical application, we think hygiene can be easily processed by more complex transformer systems (such as [12]). Overall, our results show lazy desugaring is a good way to handle hygienic sugars in any system.

4.1.3 Recursive Sugars.

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

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

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

Take `(Odd 2)` as an example. The previous work will firstly desugar the program using the rules. Then the desugaring will never terminate as the following shows.

```
  (Odd 2)
⇢ (let ((x 2)) (if (> x 0) (Even (- x 1)) #f))
⇢ (let ((x 2)) (if (> x 0)
     (let ((x1 (- x 1))) (if (> x1 0) (Odd (- x1 1)) #t))
     #f))
⇢ ...
```

Then the advantage of our approach is embodied. Our approach does not require a whole expansion 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. 10c of the former example using our tool.

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

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

and getting resugaring sequences as Fig. 10e. Here, although the sugar can be processed by the existing resugaring approach, it will be redundant. The reason is that a `Filter` program 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 desugaring systems which do not allow pattern-based desugaring. Or there are some sugars that need to be expressed by the expressions in core language as branching conditions. Take the example of another higher-order sugar `Map` as an example, and get resugaring sequences as Fig. 10d.

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

Note that the `let` expression is to limit the sub-expression only appears once in RHS. In this example, we can find that the list `(cons 1 (list 2))`, though equal to `(list 1 2)`, is represented by core language's expression. So it will be difficult to naturally handle such inline boundary conditions for existing desugaring systems. (The case can be specific by some setting, such as local-expansion[8] in Racket language's macro.) But our approach is easy to handle cases like this without specifying the expansion.

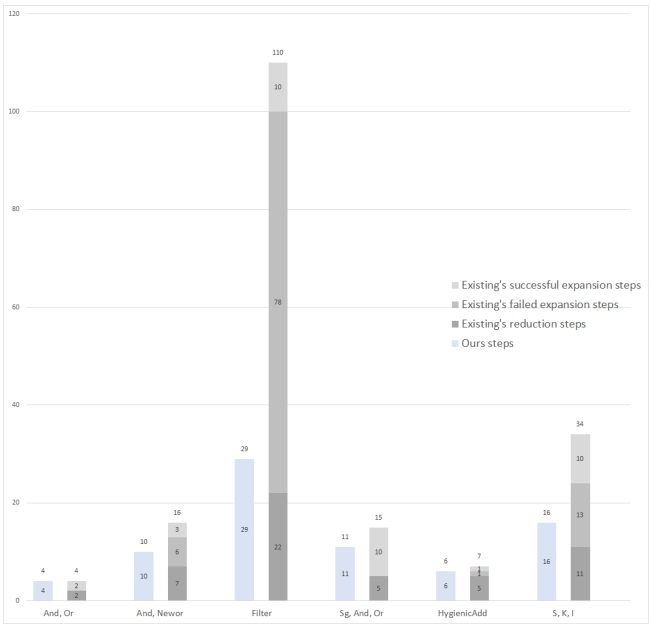


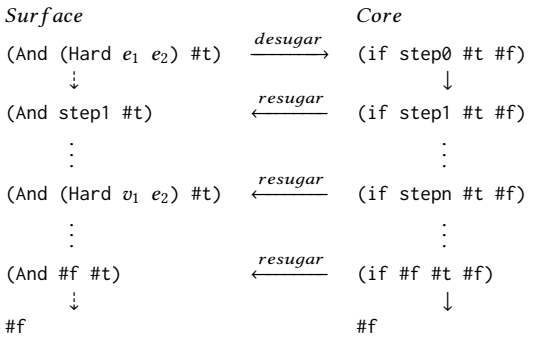
Figure 11. Comparison on Reduction Steps

4.1.4 Limitation on Presentation.

One may note that the context rules of our sugar setting limit the presentation of syntactic sugar. For example, it is difficult to present a sugar with ellipses, because the form of its context rule may vary. It is still possible if we add some restriction (so that the algorithm 1 will work), or we can just make it using the list operation just as what the sugar `Map, Filter` work. Overall, the presentation of our sugar system is not so flexible, but it won't affect the expressiveness.

4.2 Efficiency

To show the efficiency of our approach, we use the reduction steps comparing to the existing approach as the metrics. The following Fig. 11 shows the difference between the two approaches. Notice that both approaches have pre-processions—for the existing one, it is to desugar the programs to the core language together with some tags; for ours, it is to calculate the context rules of syntactic sugars. We do not consider the steps during these pre-processes. Besides, we derive the reduction steps of the existing approach into three different kinds—the reductions in the core language, the reverse expansion with failed resugaring, the reverse expansion with successful resugaring. Use the following example to see the difference. Consider a sugar named `Hard` with two arguments, which has many reduction steps after desugared. Assuming for specific e_1 and e_2, the `(Hard` e_1 e_2`)` after fully desugared has 100 reduction steps (finally to #f, for example), and only 1 intermediate step can be resugared (to `(Hard` v_1 e_2`)`, for example). Then for `(And (Hard` e_1 e_2`) #t)`, although all the 100 steps in the core language try the reverse desugaring, only `(if stepn #t #f)` (2 steps on `And`, `Hard`) and `(if #f #t #f)` (1 step on `And`) are successful. Other 98 attempts will be failed together with 98 steps on `And`.



The general regularity is—the more complex the sugar is, the more steps will our approach save. Note that if the RHS of a syntactic sugar is huge, one-step reduction of the reverse desugaring will also be more complex, because the huge sugars will contain many failed attempts to resugar. So avoiding reverse expansion of syntactic sugar can improve the efficiency for practical use because there are not always small programs like the demos.

5 Other Discussion

5.1 Model Assumption

As we mentioned in the introduction (Section 1), our approach has a more specific assumption compared to the existing approach. Here is a small gap between the motivation of the existing approach and ours—the existing approach focused mainly on a tool for existing language, while our approach considered more on a basic feature for language implementation. The examples in Section 4.1.3 have shown how the lazy desugaring solves some problems in practice.

In addition, as what we need for the lazy desugaring is just the computational order of the syntactic sugar, we can make an extension for the resugaring algorithm to work with only a black-box core language stepper. The most important difference between the black-box stepper and the evaluation rules is the computational order—while the same language behaves uniquely, the evaluation rules can show the computational order statically (without running the program). So when meeting the black-box stepper for the core language, we can just use some simple program to "get" the computational order of the core language as the following example shows: we simply let the sub-expressions of a `Head` be some reducible expressions and test the computational order.

`(if tmpe1 tmpe2 tmpe3)`
$\Downarrow_{stepper}$
`(if tmpe1' tmpe2 tmpe3)`//getting a context rule
$\Downarrow_{getnext}$
`(if tmpv1 tmpe2 tmpe3)`
$\Downarrow_{stepper}$
`tmpei`//no more rules

But that's not enough—the core language and the surface language cannot be mixed easily. We should do the same try during the evaluation to make the core language's stepper useful when meeting some surface language's expression.

Defination 5.1 (Dynamic mixed language's one step reduction $\rightarrow_m$)**.** *Defined in Fig. 12.*

Note that here we only define the reduction for unnested syntactic sugar for convenience. It is easy to extend to nested sugar (but so huge to express). Putting them in words. For expression (`CoreHead` $e_1 \ \ldots \ e_n$) whose sub-expressions contain `SurfExp`, replacing all `SurfExp` sub-expressions not in core language with any reducible core language's expression `tmpe`. Then getting a result after inputting the new expression e' to the original black-box stepper. If reduction appears at a sub-expression 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 ($\rightarrow_m$) or after core language's reduction with extension ($\rightarrow_e$) (rule `CoreExt1`, an example in Fig. 13). Otherwise, the stepper should return e', with all the replaced sub-expressions replacing back (rule `CoreExt2`, an example in Fig. 14). The extension will not violate the properties of the original core language's evaluator. It is obvious that the evaluator with the extension will reduce at the sub-expression as it needs in the core language, if the reduction appears in a sub-expression. The stepper with extension behaves the same as mixing the evaluation rules of the core language and desugaring rules of surface language.

But something goes wrong when substitution takes place during `CoreExt2`. For a program like `(let x 2 (Sugar x y))` as an example, it should reduce to `(Sugar 2 y)` by the `CoreRed2` rule, but got `(Sugar x y)` by the `CoreExt2` rule. So when using the extension of black-box stepper's rule (`ExtRed2`), we need some other information about in which sub-expression a substitution will occur. Then for these sub-expressions, we need to do the same substitution before replacing back. The substitution can be got by a similar idea as the dynamic reduction in our simple core language's setting. For example, we know the third sub-expression of an expression headed with `let` is to be substituted. we should first try `(let x 2 x)`, `(let x 2 y)` in one-step reduction to get the substitution [2/x], then, getting `(Sugar 2 y)`.

Then for any sugar expression, we can process them dynamically by "one-step try"—trying which hole is to be reduced after the outermost sugar expanded, like the example in Fig. 15. (The bold `Head` means trying on this expression.)

5.2 Properties and Trade-off

The existing resugaring approach proposed the following three properties to define the correctness.

> Emulation: Each term in the generated surface evaluation sequence desugars into the core term which it is meant to represent.
> Abstraction: Code introduced by desugaring is never revealed in the surface evaluation sequence, and code originating from the original input program is never hidden by resugaring.
> Coverage: Resugaring is attempted on every core step, and as few core steps are skipped as possible.

Here we will show what are the similarities and differences between theirs and our properties.

Emulation: The properties in Section 3 is just the same as the emulation property. We should admit that this property is the most basic one.

Abstraction and Coverage: Our reduction in the mixed language has some similarities to theirs. But since our framework has no execution for the fully desugared program and no reverse desugaring, it will be some differences in details.

Overall, our approach restricts the output by the `Head` of an expression and its sub-expressions. It is quite natural since the motivation of the resugaring is to show useful intermediate sequences, we think it will be better than restricting the output by judging whether the intermediate expressions contain some components desugared from the original program's components. Let's see the following sugar definition.

$$\texttt{(Nor x y)} \rightarrow_d \texttt{(And (not x) (not y))}$$

$$\texttt{(And x y)} \rightarrow_d \texttt{(if x y \#f)}$$

Then for a logic domain, what should be a resugaring sequence of the program `(not (And (Nor #f #t) #t))` ?

In our opinion, if the outer `not`, `And` can be displayed, so they should be after desugared. The existing approach will produce the sequences as follows.

```
    (not (And (Nor #f #t) #t))
⟶ (not (And #f #t))
⟶ (not #f)
⟶ #t
```

while ours will produce the following sequences.

```
    (not (And (Nor #f #t) #t))
⟶ (not (And (And (not #f) (not #t)) #t))
⟶ (not (And (And #t (not #t)) #t))
⟶ (not (And (not #t) #t))
⟶ (not (And #f #t))
⟶ (not #f)
⟶ #t
```

Also, if we want to display the core language's expression only when it is originated from the input program, we can just make a mirror for it as a `CommonHead`. For example, when we want to show resugaring sequences of `(And (if (And`

$$\frac{\forall\, i.\ e_i \in \texttt{CoreExp} \qquad (\texttt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_c\ e'}{(\texttt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_m\ e'} \quad (\textsc{CoreRed})$$

$$\frac{\forall\, i.\ tmp_i = (e_i \in \texttt{SurfExp}\ ?\ \texttt{tmpe}\ :\ e_i) \qquad (\texttt{CoreHead}\ tmp_1\ \ldots\ tmp_i\ \ldots\ tmp_n)\ \rightarrow_c\ (\texttt{CoreHead}\ tmp_1\ \ldots\ tmp_i'\ \ldots\ tmp_n)}{(\texttt{CoreHead}\ e_1\ \ldots\ e_i\ \ldots\ e_n)\ \rightarrow_m\ (\texttt{CoreHead}\ e_1\ \ldots\ e_i'\ \ldots\ e_n)\ \ where\ e_i\ \rightarrow_m\ e_i'} \quad (\textsc{CoreExt1})$$

$$\frac{\forall\, i.\ tmp_i = (e_i \in \texttt{SurfExp}\ ?\ \texttt{tmpe}\ :\ e_i) \qquad (\texttt{CoreHead}\ tmp_1\ \ldots\ tmp_n)\ \rightarrow_c\ e'\ \text{// not reduced in sub-expressions}}{(\texttt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_m\ e'[e_1/tmp_1\ \ldots\ e_n/tmp_n]} \quad (\textsc{CoreExt2})$$

where tmpe is any reduciable CoreExp expression

Figure 12. Dynamic Reduction

```
(if (and e1 e2) true false)
        ⇓replace
(if tmpe1 true false)
        ⇓blackbox
(if tmpe1' true false)
        ⇓desugar
(if (if e1 e2 false) true false)
```

Figure 13. CoreExt1's Example

```
(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 ...))
```

Figure 14. CoreExt2's Example

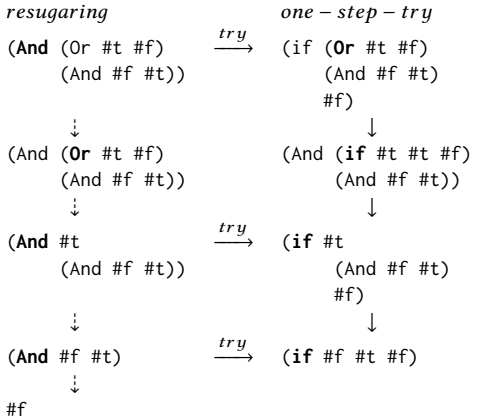


Figure 15. Example of One-step-try

`#t #f) ...) ...)` without showing the `if` expression expanded from And, we only need to set `If` as `CommonHead` together with its evaluation rules same as `if`.

In summary, our approach chooses a slightly different way for the *abstraction* for better *coverage* in the real application, which the authors of the existing approach also mentioned (but by different processing).

5.3 Hygiene

As an important property for sugar or macro system, the existing approach uses a data structure (ASD) to handle hygiene problems in resugaring, while in our approach the hygiene can be handled easily. Previous research[1] has discussed why some simple processing such as renaming is insufficient in some cases. But in our system, the hygienic problem is solved easily.

In our approach, the sugar can contain some bindings, written by the core language's `let`. The hygienic problem only happens when binders of an expanded sugar conflict with other binders. We file them into two cases—

- A sugar in binding's context.
- A sugar's sub-expression containing bindings.

Two simple examples are as follows.

`(let (x #t) (Or #f x))` case1 where

`(Or e1 e2)` $\rightarrow_d$ `(let (x e1) (if x x e2))`

`(Subst (+ f (let (f 1) f)) f 5)` case2 where

`(Subst` $e_1\ e_2\ e_3$`)` $\rightarrow_d$ `(let (e2 e3) e1)`

For the case1, the very basic and common hygienic problem, is not a problem, because our "lazy desugaring" setting won't let the sugar `Or` expanded before the x is substituted. Because the bound variables in sugar expressions are only introduced by let-binding, all of them can "delay" the expansion of the syntactic sugar.

For the case2, where the sub-expression `f` is a free variable introduced by the sugar `Subst`, the program will firstly expand the `Subst` (because the only hole at e_3 has been a value), then hygienic problem is just about the core language—which can be easily solved by capture-avoiding substitution.

Because of the definition of desugaring in our approach, we can not achieve hygiene by proving the $\alpha - equivalence$. Here what we want to show is that, even without complex things like macro systems, scope specification and so on, the lazy desugaring itself will solve the common hygienic problem with carefully-designed core language. And of course the lazy desugaring will also work together with a hygienic desugaring system (e.g., by specific the binding scope [12]). Also, the scope inference of syntactic sugar[17] provides a good perspective for solving the hygienic problem.

6 Related Work

As discussed many times before, our work is much related to the pioneering work of *resugaring* in [15, 16]. The idea of "tagging" and "reverse desugaring" is a clear explanation of "resugaring", but it becomes very complex when the RHS of the desugaring rule becomes complex. Our approach does not need to reverse desugaring, and is more lightweight, powerful, and efficient. For hygienic resugaring, compared with the approach of using DAG to solve the variable binding problem in [16], our approach of "lazy desugaring" can achieve kind of natural hygiene within our core language.

Macros as multi-stage computations [11] is a work related to our lazy expansion for sugars. Some other researches [19] about multi-stage programming [21] indicate that it is useful for implementing domain-specific languages. However, multi-stage programming is a metaprogramming method, which mainly works for run-time code generation and optimization. In contrast, our lazy resugaring approach treats sugars as part of a mixed language, rather than separate them by staging. Moreover, the lazy desugaring gives us a chance to derive evaluation rules of sugars, which is a new good point compared to multi-stage programming.

Our work is related to the *Galois slicing for imperative functional programs* [18], a work for dynamic analyzing functional programs during execution. The forward component of the Galois connection maps a partial input x to the greatest partial output y that can be computed from x; the backward component of the Galois connection maps a partial output y to the least partial input x from which we can compute y. This can also be considered as a bidirectional transformation [3, 9] and the round-tripping between desugaring and resugaring in the existing approach. In contrast to these works, our resugaring approach is basically unidirectional, with a local bidirectional step for a one-step try in our lazy desugaring. It should be noted that Galois slicing may be useful to handle side effects in resugaring in the future (for example, slicing the part where side effects appear).

There is a long history of hygienic macro expansion[13], and a formal specific hygiene definition was given [12] by specific the binding scopes of macros. another formal definition of the hygienic macro[1] is based on nominal logic[10].

Our implementation is built upon the PLT Redex [4], a semantics engineering tool, but it is possible to implement our approach on other semantics engineering tools such as those in [20, 22] which aim to test or verify the semantics of languages. The methods of these researches can be easily combined with our approach to implementing more general rule derivation. *Ziggurat* [7] is a semantic-extension framework, also allowing defining new macros with semantics based on existing terms in a language. It is should be useful for static analysis of macros.

7 Conclusion

In this paper, we purpose a novel resugaring approach by lazy desugaring. Overall, rather than a tool for existing programming languages, our approach is more likely to be a feature of meta-language. We design the approach based on a small core language, with a simple desugaring system. In our resugaring approach, the most important insight is delaying the expansion of syntactic sugars by calculating context rules(see in Section 3), which decide whether the mixed language should reduce the sub-expression or expand the sugar. The lazy desugaring gives our approach chances to achieve better efficiency and expressiveness.

As for the future work, we found side effects are troublesome to handle in resugaring, because once a side effect is taken in RHS of a desugaring rule, the sugar cannot be easily resugared according to *emulation* property. We need to find a gentler way to handle sugars with side effects. We also want to extend the core language and the desugaring system with things like type systems, analyzers. In addition, we found it is possible to derivate the stand-alone evaluation rules for the surface language by means the same as calculating the context rules. Maybe there is a more gentle way for developing domain-specific languages.

References

[1] Michael D. Adams. 2015. Towards the Essence of Hygiene. *SIGPLAN Not.* 50, 1 (Jan. 2015), 457–469. https://doi.org/10.1145/2775051.2677013

[2] 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.

[3] 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

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

[5] 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.

[6] 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

[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

[8] Matthew Flatt, Ryan Culpepper, David Darais, and Robert bruce Findler. 2012. Macros That Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. *J. Funct. Program.* 22, 2 (March 2012), 181–216. https://doi.org/10.1017/S0956796812000093

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

[10] Murdoch J. Gabbay and Andrew M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *Form. Asp. Comput.* 13, 3–5 (July 2002), 341–363. https://doi.org/10.1007/s001650200016

[11] 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

[12] David Herman and Mitchell Wand. 2008. A Theory of Hygienic Macros. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems* (Budapest, Hungary) *(ESOP'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 48–62.

[13] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 151–161. https://doi.org/10.1145/319838.319859

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

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

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

[22] 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