Compositional Resugaring by Lazy Desugaring

Anonymous Author(s)

Abstract

Syntactic sugar, first coined by Peter J. Landin in 1964, has proved to be very useful for defining domain-specific languages and extending languages. Unfortunately, when syntactic sugar is eliminated by transformation, it obscures the relationship between the user's source program and the transformed program. Resugaring is a powerful technique to resolve this problem, which automatically converts the evaluation sequences of desugared expression in the core language into representative sugar's syntax in the surface language. However, the traditional approach relies on reverse application of desugaring rules to desugared core expression whenever possible. When a desugaring rule is complex and a desugared expression is large, such reverse desugaring becomes very complex and costive.

In this paper, we propose a novel approach to compositional resugaring by lazy desugaring, where reverse application of desugaring rules is unnecessary. We recognize a sufficient and necessary condition for a syntactic sugar to be desugared, and propose a reduction strategy, based on the evaluator of the core languages and the desugaring rules, which can produce all necessary resugared expressions on the surface language. We have implemented a system based on this new approach. Compared to the traditional approach, the new approach is not only more efficient, but also more powerful in that it cannot only deal with all cases (such as hygienic and simple recursive sugars) published so far, but can do more allowing more flexible recursive sugars.

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

1 Introduction

Syntactic sugar, first coined by Peter J. Landin in 1964 [12], was introduced to describe the surface syntax of a simple ALGOL-like programming language which was defined semantically in terms of the applicative expressions of the core lambda calculus. It has proved to be very useful for defining domain-specific languages (DSLs) and extending languages [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 [13, 14]. It can automatically convert the evaluation sequences of desugared expression 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 programs after the desugared with syntactic sugars' rules to reversely desugar the sugars—that why it was named "resugaring". Just as the example in Fig 1, here we use the sugar `Or` to see how existing resugaring approach works, with following desugaring rule.

$$\texttt{(Or x y)} \rightarrow_d \texttt{(let ((t x)) (if t t y))}$$

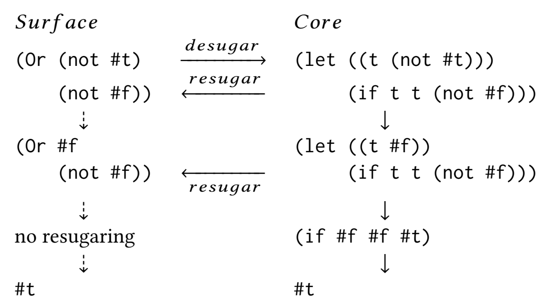

Figure 1. simple example

But it is not as easy as the example above. Sometimes the program in the core language contains a desugared form of a sugar, but the form may belong to original (not desugared) program. (See the example in Fig 2) In such cases, the unexpansion of sugar should be noticed.

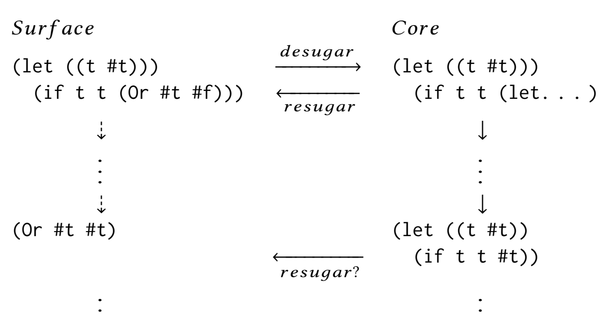

Figure 2. error example

Moreover, when meeting hygienic sugar, the simple match and substitution will not work as Fig 3 shows. Here the `Double` sugar has the following desugaring rule.

$$\texttt{(Double x)} \rightarrow_d \texttt{(let (t x) (* 2 t))}$$

If we use simple binder renaming for solving the simple case, some other information is needed (such as the permutation).

The existing resugaring approaches subtly solved the problems above by "tagging"[13] and "abstract syntax DAG"[14].

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

$$\begin{array}{lcl} \textit{Surface} & & \textit{Core} \\ \texttt{(let (t 1)} & \xrightarrow{desugar} & \texttt{(let (t 1)} \\ \quad\texttt{(Double t))} & \underset{resugar?}{\longleftarrow\!-\!-\!-\!-\!-} & \quad\texttt{(let (t' t) (* 2 t')))} \end{array}$$

Figure 3. hygiene example

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

Figure 4. A Typical Resugaring Example

While those techniques successfully make the resugaring method usable, there is a key point which makes the existing approaches not very practical—the unexpandsion of sugars needed to match the desugared programs 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 can be desugared to large sub-programs. Also, as the debugging for host language can be a good application of resugaring, the efficiency matters.

In this paper, we propose a novel approach to compositional 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, our resugaring approach is based on the evaluation rules (consist of the context rules and reduction rules) of the core language. To this end, we consider the surface language and the core language as one language. We regard the desugaring rules as the reduction rules of surface language and deriving the context rules of surface language to see how lazy the desugaring 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 reverse application of desugaring rules becomes unnecessary. We recognize a sufficient and necessary condition for a syntactic sugar to be desugared, and propose a reduction strategy, based on evaluator of the core languages and the desugaring rules, which is sufficient to produce all necessary resugared 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 traditional approach, because it completely avoids unnecessary complexity of the reverse desugaring. It is more powerful in that it cannot only deal with all cases (such as hygienic and simple recursive sugars) published so far [13, 14], but can do more allowing more 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 black-box stepper, how it can easily deal with hygiene, and how we deal with the trade-off of correctness.

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 the implementation and show some applications as case studies in Section 4. We discuss on our core insight in Section 5, on related work in Section 6, and conclude the paper in Section 7.

2 Overview

In this section, we give a brief overview of our approach, explaining its difference from the traditional approach and highlighting its new features. To be concrete, we will consider the following simple core language, defining boolean expressions based on `if` construct:

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

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

$$\begin{array}{lcl} C & := & (\texttt{if}\ [\cdot]\ e\ e) \\ & & \textit{hole} \end{array}$$

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

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

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

$$\begin{array}{rcl} (\texttt{And}\ e_1\ e_2) & \longrightarrow_d & (\texttt{if}\ e_1\ e_2\ \#f) \\ (\texttt{Or}\ e_1\ e_2) & \longrightarrow_d & (\texttt{if}\ e_1\ \#t\ e_2) \end{array}$$

Now let us demonstrate how to execute `(And (Or #t #f) (And #f #t))`, and get the resugaring sequences as Figure 4 by both the traditional approach and our new approach.

To solve the problem in the traditional approach, we propose a new resugaring approach by eliminating "reverse desugaring" via "lazy desugaring", where a syntactic sugar will be desugared only when it is necessary. While giving the evaluation rules of the core language, we can figure the following context rules of the surface language.

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

$$
\begin{array}{lcl}
\texttt{Mixedexp} & ::= & \texttt{Coreexp} \\
 & | & \texttt{Surfexp} \\
 & | & \texttt{Commonexp} \\
\texttt{Coreexp} & ::= & (\texttt{if}\ e\ e\ e) \\
\texttt{Surfexp} & ::= & (\texttt{And}\ e\ e) \\
 & | & (\texttt{Or}\ e\ e) \\
\texttt{Commonexp} & ::= & \#t \\
 & | & \#f
\end{array}
$$

(a) Syntax

$$
\begin{array}{lcl}
C & := & (\texttt{if}\ [\cdot]\ \mathrm{e}\ \mathrm{e}) \\
 & | & (\texttt{And}\ [\cdot]\ \mathrm{e}) \\
 & | & (\texttt{Or}\ [\cdot]\ \mathrm{e}) \\
 & | & \mathit{hole}
\end{array}
$$

(b) Context

$$
\begin{array}{rcl}
(\texttt{And}\ e_1\ e_2) & \longrightarrow_m & (\texttt{if}\ e_1\ e_2\ \#f) \\
(\texttt{Or}\ e_1\ e_2) & \longrightarrow_m & (\texttt{if}\ e_1\ \#t\ e_2) \\
(\texttt{if}\ \#t\ e_1\ e_2) & \longrightarrow_m & e_1 \\
(\texttt{if}\ \#f\ e_1\ e_2) & \longrightarrow_m & e_2
\end{array}
$$

(c) Reduction

Figure 5. Mixed Language Example

```
   (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
```

Figure 6. mix

Then we can just mix the surface language and the core language as follows.

Finally the program `(And (Or #t #f) (And #f #t))` will got the evaluation sequence in Fig 6 in the mixed language. We can just filter the intermediate sequences without `Coreexp` in any sub-expressions to get the resugaring sequence as Fig 4.

3 Resugaring by Lazy Desugaring

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

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

Figure 7. Core and Surface Expressions Todo:

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**, that is, for evaluation contexts E_1 and E_2, $E_1[E_2]$ is also a evaluation context.Todo: where should it appear??

3.1.1 Core Language. For our core language, its evaluator 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 condition; a counterexample for this assumption is in Fig 8.

$$
\begin{array}{lcl}
\mathrm{C} & ::= & (\texttt{notif}\ [\cdot]\ e_2\ e_3) \\
 & | & (\texttt{notif}\ v_1\ [\cdot]\ e_3), (\mathit{side-condition}\ (\mathit{equal?}\ v_1\ \#t)) \\
 & | & (\texttt{notif}\ v_1\ e_2\ [\cdot]), (\mathit{side-condition}\ (\mathit{equal?}\ v_1\ \#f))
\end{array}
$$

Figure 8. An Example of Context Rules with Conditions

An expression form of the core language is defined in Figure 7. It is a variable, a constant, or a (language) constructor expression. Here, `CoreHead` stands for a language constructor such as `if` and `let`. To be concrete, we will use a simplified core language defined in Figure 9 to demonstrate our approach. Here the [e/x] is a capture-avoiding substitution.

3.1.2 Surface Language. Our surface language is defined by a set of syntactic sugars, together with some basic elements in the core language, such as constant, variable. The surface language has expressions as given in Figure 7.

A syntactic sugar is defined by a desugaring rule in the following form (the desugaring here is just a simple textual rewriting):

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

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

(b) Context Rules

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

(c) Part of Reduction Rules

Figure 9. A Core Language Example

where its left-hand-side (LHS) is a unnested pattern and its RHS is an expression of the surface language or the core language, and any pattern variable (e.g., e_1) in LHS only appears once in RHS. For instance, we may define syntactic sugar And by

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

Note that if the pattern is nested, we can introduce a new syntactic sugar to flatten it. And if we need to use a pattern variable multiple times in RHS, a `let` binding may be used (a normal way in syntactic sugar). We take the following sugar as an example

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

If we execute `(Twice (+ 1 1))`, it will firstly be desugared to `(+ (+ 1 1) (+ 1 1))`, then reduced to `(+ 2 (+ 1 1))` by one step. The subexpression `(+ 1 1)` has been reduced but should not be resugared to the surface, because the other `(+ 1 1)` has not been reduced yet. So we just use a `let` binding to resolve this problem. 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.

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

As described above, we only assume the desugaring is a textural rewriting, 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. Todo: maybe an example needed

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

3.1.3 Mixed Language.

Our mixed language for resugaring combines the surface language and the core language, described in Figure 10. The differences between expressions in our core language and those in our surface language are identified by their `Head`. But there may be some expressions in the core language which are also used in the surface language for convenience, or we need some core language's expressions to help us getting better resugaring sequences. So we take `CommonHead` as a subset of the `CoreHead`, which can be displayed in resugaring sequences. Then if any sub-terms in

```
Exp              ::=  DisplayableExp
                  |   MixedExp

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

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

Figure 10. Our Mixed Language

a expression contains no `CoreHead` except for `CommonHead`, we should let them display during the evaluation process (named `DisplayableExp`). Otherwise, the expression should not be displayable. We just use a `MixedExp` term to present the expressions which is not necessarily displayed for concision.

We distinguish the displayable expressions for the *abstraction* property (discussed in Section 5).

As an example, for the core language in Figure 9, we may assume `arithop`, `lambda` (call-by-value lambda calculus), `cons` as `CommonHead`, `if`, `let`, `lambdaN` (call-by-name lambda calculus), `listop` as `CoreHead` but out of `CommonHead`. This will allow `arithop`, `lambda` and `cons` to appear in the resugaring sequences, and thus display more useful intermediate steps during resugaring.

Note that some expressions with `CoreHead` contain subexpressions with `SurfHead`, they are of `CoreExp` but not in the core language. In the mixed language, we process these expressions by the context rules of the core language, so that the reduction rules of core language and the desugaring rules of surface language can be mixed as a whole (the $\rightarrow_c$ in next section). For example, suppose we have the context rule of `if` expression[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` (actually, how the subexpression reduced does not matter, because it is just to mark the location where it should be reduced); if e_1 is also a `CoreExp` which has one or more non-core subexpressions, a recursive reduction by $\rightarrow_c$ is needed.

3.2 Resugaring Algorithm

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

- Calculating the context rules of syntactic sugar.
- Filtering `DisplayableExp` during the execution of the mixed language.

We describe the algorithm of calculating the context rules for a syntactic sugar as follows.

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`

```
Let ListofRule = {}, tmpLHS = currentLHS, InCal = append(currentInCal, SurfHead)
if ∄ contexts rules of Head then
  if ∃ Head in InCal then
    return error
  else
    ListofRule.append(
        calcontext(Head.LHS,Head.RHS,InCal))
  end if
end if
Let OrderList = {e'_i, e'_j, ...} //computation order got by context rules
for flag in OrderList do
  if ∃i, s.t.e_i = flag then
    ListofRule.append(tmpLHS[[•]/e_i])
  else
    Let recRule, recLHS = calcontext(
        tmpLHS, flag, Incal)
    tmpLHS = recLHS
    ListofRule = append(ListofRule, recRule)
    Todo: align?
  end if
end for
return ListofRule, tmpLHS
```

For sugar

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

we can just 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. See example in Fig 11

Or if the `Head` is a `SurfHead` with its context rules calculated, then we regard it as `CoreHead`. If it is without context

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

$$(\texttt{Sg1}\ e_1\ e_2\ e_3\ e_4) \longrightarrow_d (\texttt{if}\ (=\ (+\ e_4\ e_2)\ (*\ e_1\ e_3))\ \#t\ \#f)$$

$$\begin{aligned}
&OrderList = \{(=\ (+\ e_4\ e_2)\ (*\ e_1\ e_3))\} && \text{(depth1)}\\
&OrderList = \{(+\ e_4\ e_2),\ (*\ e_1\ e_3)\} && \text{(depth2)}\\
&OrderList = \{e_4,\ e_2\} && \text{(depth3.1, getting rules}\\
& && \{(\texttt{Sg1}\ e_1\ e_2\ e_3\ [\bullet]), (\texttt{Sg1}\ e_1\ [\bullet]\ e_3\ v_4)\})\\
&OrderList = \{e_1,\ e_3\} && \text{(depth3.2, getting rules}\\
& && \{(\texttt{Sg1}\ [\bullet]\ v_2\ e_3\ v_4), (\texttt{Sg1}\ v_1\ v_2\ [\bullet]\ v_4)\})
\end{aligned}$$

Figure 11. example1

rules we should calculate 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.

$$(\texttt{Odd}\ x) \longrightarrow_d (\texttt{Even}\ (-\ x\ 1)) \qquad (\texttt{Even}\ x) \longrightarrow_d (\texttt{Odd}\ (-\ x\ 1))$$

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 Sec 2.

Now, our resugaring algorithm can be easily defined based on evaluation rules of the mixed language. Let $\longrightarrow_m$ be one step reduction in the mixed language.

$$\begin{aligned}
\texttt{resugar}(e) \quad = \quad & \textbf{if}\ \texttt{isNormal}(e)\ \textbf{then}\ \textit{return}\\
& \textbf{else}\\
& \quad \textbf{let}\ e \longrightarrow_m e'\ \textbf{in}\\
& \quad \textbf{if}\ e' \in \texttt{DisplayableExp}\\
& \quad\quad \texttt{output}(e'),\ \texttt{resugar}(e')\\
& \quad \textbf{else}\ \texttt{resugar}(e')
\end{aligned}$$

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

3.3 Correctness

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

Defination 3.1 (fulldesugar)**.** *The function that recursively desugars any expressions of the mixed language is defined as Fig 12.*

```
fulldesugar(v) = v
fulldesugar(x) = x
fulldesugar((CoreHead e1 e2 ...)) =
  (CoreHead fulldesugar(e1) fulldesugar(e2) ...)
fulldesugar((SurfHead e1 e2 ...)) =
  (CoreHead fulldesugar(e1') fulldesugar(e2') ...)
where desugar((SurfHead e1 e2 ...)) = (CoreHead e1' e2' ...)
```

Figure 12. Defination of `fulldesugar`

A program `P` can be fulldesugared if `fulldesugar(P)` is terminable.

Theorem 3.1. *For a program of the mixed language* `P` *which can be fulldesugared, if a sugar expression* `S` *in the program* `P` *is desugared in one step of the mixed language's evaluation, and program* `P'` *is the* `P` *after* `fulldesugar` *to the core language, then one step of the core language's evaluation on* `P'` *will destroy the sugar* `S`*'s desugared form.*

Theorem 3.2. *For a program of the mixed language* `P` *which can be fulldesugared, if a core language's expression* `E` *in a program* `P` *of the mixed language is reduced by reduction rules, and program* `P'` *is the* `P` *after* `fulldesugar` *to the core language, then one step of the core language's evaluation on* `P'` *will reduce on the* `P'` *will also reduce on the correspond expression of* `E`*.*

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

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

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

and context rule $\texttt{C} = \texttt{SurfHead}.LHS[[\bullet]/e_i]$*, where* $[\bullet]$ *is at* e_i*'s location. Then desugar*$(\texttt{C}) = \texttt{SurfHead}.RHS[[\bullet]/e_i]$

Lemma 3.1. *The context rules of syntactic sugar are correct if computable.*

Or to say, for sugar `S`*, the context rules* $C_1, C_2 \ldots$*, if the context rules limit reduction order of S.LHS in* $\{e_i,\ \ldots,\ e_j\}$*, then the reduction order of S.RHS is also of this sequence.*

Proof. In algo 1, we recursively search the sugar's RHS to find $e_i \ldots$ in RHS's computation order, so the sugar's context rules are correct. □

Proof of Theorem 3.1. Consider a context `C`, where the sugar `S` in the hole. So the `P` is the program `C[S/hole]`.

If no sugar expression in `C`, then it is to prove the program `C[fulldesugar(S)/hole]` will reduce on `fulldesugar(S)`. It is obvious because that is where the hole at.

If there is any sugar in `C`, then it is to prove the program `fulldesugar(C[S/hole])` will reduce on `fulldesugar(S)`. According to the lemma, it is true because the hole will be at the same place after any sugar recursively desugared. □

Proof of Theorem 3.2. According to the lemma, any sugar has the correct context rules. So the hole is also correct for the mixed language. □

4 Case Studies

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 Figure 13. Note that we set call-by-value lambda calculus as terms in `CommonExp`, because we want to output some intermediate expressions including lambda expressions in some examples. It's easy if we want to skip them.

4.1 Simple Sugars

We construct some simple syntactic sugars and try it on our tool. Some sugars are inspired by the first work of resugaring [13]. The result shows that our approach can handle all sugar features of their first work. Take an SKI combinator syntactic sugar as an example. (We can regard `S` as an expression headed with S, without subexpression.) And for showing a concise result, we add the call-by-need lambda calculus in the core language for this example.

$$\begin{array}{l} \texttt{S} \rightarrow_d (\lambda_N\ \texttt{(x1 x2 x3) (x1 x2 (x1 x3)))} \\ \texttt{K} \rightarrow_d (\lambda_N\ \texttt{(x1 x2) x1)} \\ \texttt{I} \rightarrow_d (\lambda_N\ \texttt{(x) x)} \end{array}$$

Although SKI combinator calculus is a reduced version of lambda calculus, we can construct combinators' sugar based on call-by-need lambda calculus in our core language. For the sugar expression `(S (K (S I)) K xx yy)`, we get the resugaring sequences as Figure 13a. During the test, we find 33 intermediate steps (in our implementation) are needed after the totally desugaring of the input expression, but only 5 of them can be returned to the surface, so many attempts to reverse the desugaring would fail if using the traditional resugaring approach, in such a little expression. That's why lazy desugaring makes our approach efficient.

Then for expressions headed with `or`, we won't need the one-step try to figure out whether desugaring or processing on a subexpression, which makes our approach more concise. We also try some more complex sugars where context rules for multi-branches are different, then get the correct evaluation rules without difficulty. Overall, the unidirectional resugaring algorithm based on lazy desugaring makes our approach efficient, because no attempts for resugaring the expression are needed.

4.2 Hygienic Sugars

The second work [14] of traditional resugaring approach mainly processes hygienic sugar compared to first work. It uses a DAG to represent the expression. However, hygiene is not hard to be handled by our lazy desugaring strategy. Our algorithm can easily process hygienic sugar without a special data structure. A typical hygienic problem is as the following example.

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

For traditional resugaring approach, if we want to get sequences of `(let (x 2) (Hygienicadd 1 x))`, it will firstly desugar to `(let (x 2) (let x 1 (+ x x)))`, which is awful because the two x in `(+ x x)` should be bound to different values. So the traditional hygienic resugaring approach uses abstract syntax DAG to distinct different `x` in the desugared expression. But for our approach based on lazy desugaring, the `Hygienicadd` sugar does not have to desugar until necessary, thus, getting resugaring sequences as Figure 13b based on a non-hygienic rewriting system.

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

Overall, our results show lazy desugaring is a good way to handle hygienic sugars in any systems.

4.3 Recursive Sugars

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

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

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

Take `(Odd 2)` as an example. The previous work will firstly desugar the expression using the rewriting system. Then the rewriting system will never terminate as following shows.

```
   (Odd 2)
--> (if (> 2 0) (Even (- 2 1) #f))
--> (if (> (- 2 1) 0) (Odd (- (- 2 1) 1) #t))
--> (if (> (- (- 2 1) 1) 0) (Even (- (- (- 2 1) 1) 1) #f))
--> ...
```

Then the advantage of our approach is embodied. Our lightweight approach does not require a whole expanding of sugar expression, which gives the framework chances to judge boundary conditions in sugars themselves, and showing more intermediate sequences. We get the resugaring sequences as Figure 13c of the former example using our tool.

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

$$\begin{array}{l} \texttt{(Filter}\ e\ \texttt{(list}\ v_1\ v_2\ \texttt{...))} \rightarrow_d \\ \quad \texttt{(let f}\ e\ \texttt{(if (f}\ v_1\texttt{)} \\ \qquad\qquad \texttt{(cons}\ v_1\ \texttt{(Filter f (list}\ v_2\ \texttt{...)))} \\ \qquad\qquad \texttt{(Filter f (list}\ v_2\ \texttt{...))))} \\ \texttt{(Filter}\ e\ \texttt{(list))} \rightarrow_d \texttt{(list)} \end{array}$$

and getting resugaring sequences as Figure 13e. Here, although the sugar can be processed by traditional resugaring approach, it will be redundant. The reason is that, a Filter for a list of length n will match to find possible resugaring

```
   (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 13. Resugaring Examples

$n*(n-1)/2$ times. Thus, lazy desugaring is really important to reduce the resugaring complexity of recursive sugar.

Moreover, just like the *Odd* and *Even* sugar above, there are some simple rewriting systems which do not allow pattern-based rewriting. Or there are some sugars that need to be expressed by the expressions in core language as branching conditions. Take the example of another higher-order sugar `Map` as an example, and get resugaring sequences as Figure 13d.

```
(Map e1 e2) →d Todo :format?
  (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 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 expression. So it will be difficult to handle such inline boundary conditions for traditional rewriting systems. But our approach is easy to handle cases like this. So our resugaring approach by lazy desugaring is powerful.

5 Discussion

5.1 Model Assumption

As we mentioned in the introduction (Sec 1), our approach has a more specific assumption compared to the existing approach. Here is a small gap between the motivation of existing approach and ours—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 Sec 4.3 have shown how the lazy desugaring solve some problems in practice.

In addition, as what we need for the lazy desugaring is just the computation order of the syntactic sugar, we can make a extension for the resugaring algorithm to see how it is able to work with only a black-box core language stepper. The most important difference between black-box stepper and the evaluation rules is the computation order—while the same language will behave uniquely, the evaluation rules can show the computation 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 computation order of the core language as the following example 14 shows.

But that's not enough—the core language and the surface language cannot be mixed easily. We should do the same try

```
(if tmpe1 tmpe2 tmpe3)
        ⇓stepper
(if tmpe1' tmpe2 tmpe3)
        ⇓getnext
(if tmpv1 tmpe2 tmpe3)
        ⇓stepper
       tmpei
```

so getting one context rule for the `if` expression.

Figure 14. getting the order

during the evaluation to make the core language's stepper useful when meeting some surface language's term.

Defination 5.1 (dynamic mixed language's one step reduction $\rightarrow_m$)**.** *Defined in Fig 15.*

Putting them in simple words. For expression (`CoreHead` $e_1 \ \ldots \ e_n$) whose subexpressions contain `SurfExp`, replacing all `SurfExp` subexpressions not in core language with any reducible core language's term `tmpe`. Then getting a result after inputting the new expression e' to the original black-box stepper. If reduction appears at a subexpression at e_i or what the e_i replaced by, then the stepper with the extension should return (`CoreHead` $e_1 \ \ldots \ e_i' \ \ldots \ e_n$), where e_i' is e_i after the mixed language's one-step reduction ($\rightarrow_m$) or after core language's reduction with extension ($\rightarrow_e$) (rule `CoreExt1`, an example in Figure 16). Otherwise, the stepper should return e', with all the replaced subexpressions replacing back (rule `CoreExt2`, an example in Figure 17). The extension will not violate the properties of original core language's evaluator. It is obvious that the evaluator with the extension will reduce at the subexpression as it needs in core language, if the reduction appears in a subexpression. The stepper with extension behaves the same as mixing the evaluation rules of core language and desugaring rules of surface language.

But something goes wrong when substitution takes place during `CoreExt2`. For a expression 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 subexpression a substitution will occur. Then for these subexpressions, 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 subexpression of a 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 desugared, as example in Fig 18. (The bold `Head` means trying on this term.)

5.2 Correctness and Trade-off

The existing resugaring approach proposed three properties to define the correctness. Here we will show what are the similarities and differences between theirs and our properties.

> 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.

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: Actually, our reduction in the mixed language has some similarities to theirs. But since our framework has no execution for the fulldesugared program and no reverse desugaring, it will be some different in details.

Overall, our approach restricts the output by the `Head` of a program and its inner(or to say, 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 program contains some components desugared from the original program's components. Let's see the following example in Fig 19

Then for the program in 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))
```

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

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

$$\frac{\forall\, i.\ tmp_i = (e_i \in \mathtt{SurfExp}\ ?\ \mathtt{tmpe}\ :\ e_i) \qquad (\mathtt{CoreHead}\ tmp_1\ \ldots\ tmp_n) \longrightarrow_c e' \ \text{// not reduced in subexpressions}}{(\mathtt{CoreHead}\ e_1\ \ldots\ e_n) \longrightarrow_m e'[e_1/tmp_1\ \ldots\ e_n/tmp_n]} \quad (\textsc{CoreExt2})$$

where `tmpe` is any reducible `CoreExp` term

Figure 15. Dynamic reduction

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

Figure 16. `CoreExt1`'s Example

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

Figure 17. `CoreExt2`'s Example

```
resugaring                       onesteptry
(And (Or #t #f)                  (if (Or #t #f)
     (And #f #t))    --try-->         (And #f #t)
                                      #f)
      ⇣                                ↓
(And (Or #t #f)                  (And (if #t #t #f)
     (And #f #t))                     (And #f #t))
      ⇣                                ↓
(And #t              --try-->    (if #t
     (And #f #t))                     (And #f #t)
                                      #f)
      ⇣                                ↓
(And #f #t)          --try-->    (if #f #t #f)
      ⇣                                ↓
#f
```

Figure 18. simple example

$$\texttt{(Nor x y)} \longrightarrow_d \texttt{(And (not x) (not y))}$$
$$\texttt{(And x y)} \longrightarrow_d \texttt{(if x y \#f)}$$

Figure 19. Nor sugar

$\longrightarrow$ `(not #f)`

$\longrightarrow$ `#t`

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

5.3 Experiment

Todo: steps?

5.4 Hygiene

As an important property for sugar or macro system, the existing approach use a data structure (ASD) to handle hygiene 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 hygiene problem is solved easily.

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

- A sugar in binding's context.
- A sugar's subexpression containing bindings.

The Fig 20 and 21 shows how the simple cases of these two.

For the case1, the very basic and common hygiene problem, is not a problem, because our "lazy desugaring" setting won't let the sugar `Or` expanded before the x is substituted.

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

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

Figure 20. Case1's Example

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

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

Figure 21. Case1's Example

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 subexpression `f` is a free variable introduced by the sugar `Subst`, the program will firstly desugar the `Subst` (because the only hole at e_3 has been a value), then hygiene 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, specifying the scopes 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 rewriting system (e.g., by specific the binding scope).

6 Related Work

As discussed many times before, our work is much related to the pioneering work of *resugaring* in [13, 14]. 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 [14], our approach of "lazy desugaring" can achieve kind of natural hygiene without special treatment. It oughts to combine with existing hygienic resugaring approaches.

Macros as multi-stage computations [9] is a work related to our lazy expansion for sugars. Some other researches [16] about multi-stage programming [18] 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* [15], 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, 8] and the round-tripping between desugaring and resugaring in traditional approach. In contrast to these work, our resugaring approach is basically unidirectional, with a local bidirectional step for a one-step try in our lazy desugaring. It should be noted that Galois slicing may be useful to handle side effects in resugaring in the future (for example, slicing the part where side effects appear).

There is a long history to hygienic macro expansion[11], and a formal specific hygiene definition was given [10] by specific the binding scopes of macros. [1] is also another formal definition of hygienic macro.

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 [17, 19] 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. In our resugaring approach, the most important part is calculating context rules for the syntactic sugars (see in Section 3), which decides whether it should reduce the subexpression or desugar the outermost 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. In addition, we found it is possible to derivate the stand-alone evaluation rules for the surface language by means 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] 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.

[9] 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

[10] 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.

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

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

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

[19] 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