Lifting Resugaring by Lazy Desugaring

ANONYMOUS AUTHOR(S)

Syntactic sugars provide an effective way to define and implement domain-specific languages. However, the programs after desugaring to a host language would be unrecognizable for people who are unfamiliar with the host language, which is bad because domain-specific languages are not always used by programmers.

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

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

1 INTRODUCTION

Syntactic sugar, first coined by Peter J. Landin in 1964 [Landin 1964], was introduced to describe the surface syntax of a simple ALGOL-like programming language which was defined semantically in terms of the applicative expressions of the core lambda calculus. It has proved to be very useful for defining domain specific languages (DSLs) and extending languages [Culpepper et al. 2019; Felleisen et al. 2018]. Unfortunately, when syntactic sugar is eliminated by transformation, it obscures the relationship between the user's source program and the transformed program.

Resugaring is a powerful technique to resolve this problem [Pombrio and Krishnamurthi 2014, 2015]. It can automatically convert the evaluation sequences of desugared expression in the core language into representative sugar's syntax in the surface language. As demonstrated in Section 2, the key idea in this resugaring is "tagging" and "reverse desugaring": it tags each desugared core term with the corresponding desugared rule, and follows the evaluation steps in the core language but keep applying the desuagring rules reversibly as much as possible to find surface-level representations of the tagged core terms.

While it is natural to do resugaring by reverse desugaring of tagged core terms, it introduces complexity and inefficiency.

- *Tricky to handle reursive sugar.* While tagging is used to remember the position of desugaring so that reverse desugaring can be done at correct position when desugared core expression is evaluated, it becomes very tricky and complex when recursive sugars are considered [Pombrio and Krishnamurthi 2014].Todo: pattern-based?
- *Complicated to handle hygienic sugar.* For reverse desugaring, we need to match part of the core expression on the RHS of the desugar rule and to get the surface term by substitution. This match-and-substitute turns out to be very complex if we consider local bindings (hygienic sugars) [Pombrio and Krishnamurthi 2015].
- *Inefficient in reverse desugaring.* It need to keep checking whether reverse desugaring is applicable during evaluation of desugared expression, which is very costive. Moreover, the match-and-substitute for reverse desugaring is costive particularly when the core term is big.

In this paper, we propose a novel approach to resugaring, which does not use tagging and reverse desugaring at all. The key idea is "lazy desugaring", in the sense that desuagring is delayed so

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

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

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

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

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

The rest of our paper is organized as follow. We start with an overview of our approach in Section 2. We then discuss the core of resuarging by lazy desugaring in Section 3, and automatic derivation of reduction rules for syntactic sugars in Section **??**. We discuss relative work in Section 5, and conclude the paper in Section 6.

2 OVERVIEW

In this section, we show a simple example processed by existing approach and our approach. Firstly, we define a simple core language with following syntax and semantic.

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

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

$$\texttt{(if \#t e1 e2)} \rightarrow \texttt{e1} \quad \text{(REDUCTION RULE OF IFTRUE)}$$

$$\texttt{(if \#f e1 e2)} \rightarrow \texttt{e2} \quad \text{(REDUCTION RULE OF IFFALSE)}$$

We define two syntactic sugars—*and* sugar and *or* sugar based on the core language.

$$\texttt{(and e1 e2)} \rightarrow_d \texttt{(if e1 e2 \#f)}$$
$$\texttt{(or e1 e2)} \rightarrow_d \texttt{(if e1 \#t e2)}$$

and execute `(and (or #t #f) (and #f #t))` as an example.

2.1 Existing resugaring approach

As we mentioned above, the existing approach use ”tagging” and ”reverse desugaring” to get resugaring sequences. Tags is to show where are terms from, and reverse dusugaring is what resugaring needs.

```
      (and (or #t #f) (and #f #t))
  ⇢   (if-andtag (if-ortag #t #t #f) (if-andtag #f #t #f) #f)  //desugar
   start evaluating
  ⟶   (if-andtag #t (if-andtag #f #t #f) #f) //check resugarable
   get (and #t (and #f #t)) by reverse desugaring
  ⟶   (if-andtag #f #t #f) //check resugarable
   get (and #f #t) by reverse desugaring
  ⟶   #f end of resugaring
```

We can find that the expression fully desugared before resugaring. The subexpression `(and #f #t)`, though desugars to `(if-andtag #f #t #f)`, doesn't be reduced in first two steps after desugaring. But the reverse desugaring trys on it in these two steps, which is redundant. Moreover, there will be some intermediate steps which can not be resugared by reverse desugaring during the evaluation in a more complex core language. Many useless resugarings on subexpressions will take place.

2.2 Resugaring by lazy desugaring

To solve the problem in existing resugaring approach, we try ”lazy desugaring”, which means that a syntactic sugar should only desugar when it has to desugar. We use a one-step try method to judge it.

```
      (resugar (and (or #t #f) (and #f #t)))
  ⇢   (tmp (if (or #t #f) (and #f #t) #f)) // should reduce on (or #t #f)
  ⟶   (and (resugar (or #t #f)) (and #f #t)) // no reduction
  ⇢   (and (tmp (if #t #t #f)) (and #f #t)) //  (or #t #f) has to desugar and reduce
  ⟶   (desugar (and #t (and #f #t)))
  ⇢   (tmp (if #t (and #f #t))) // the outermost and sugar has to desugar and reduce
  ⟶   (desugar (and #f #t))
  ⇢   (tmp (if #f #t #f)) // have to desugar and reduce
  ⟶   #f // end
```

For each step in the processes above we use one reduction or none. So 7 reduction steps is needed for the whole resugaring, while the existing approach needs 9 (3 in desugaring, 3 in evaluator, 3 reversed reduction in resugaring). Note that reversed reduction is more complex because of the match and substitution, the existing approach will be more reduntant on larger expressions than ours. Also, there is no need for tagging, which makes our approach lightweight.

Moreover, derivation of semantics for syntactic sugar will make our approach more efficent for part of sugars. Both *and* sugar and *or* sugar can be handled by the derivation. We can get the following semantics (consisted of reduction rules and context rules).

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

Then the resugaring sequences can be got not depending on core language.

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

Only 4 steps is needed by using the rules. Although the derivation of rules cannot work for all syntactic sugars which can be handled by our resugaring approach, it can work together with the resugaring. For example, we have another sugar named *hard* (regardless of its desugar rule) whose reduction rules cannot be derivated. Then if we execute `(and (hard (and #t #t) ...) ...)`, the *and*'s semantics will let the *hard* sugar execute recursively. Then the sugar *hard* will be processed by our basic resugaring approach because no semantic rules for it. Note that the resugaring process may also recursively execute on inner subexpression (e.g. `(and #t #t)`), the subexpression may use the semantic rules again.

In summary, our resugaring approach with lazy desugaring is lightweight and efficent. Following chapters will show that, though lightweight, the approach is more powerful than existing approach in some ways.

3 RESUGARING BY LAZY DESUGARING

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

3.1 Mixed Language for Resugaring

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

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

Fig. 1. Core and Surface Expressions

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

Fig. 2. An Core Language Example

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

An expression of the core language is defined in Figure 1. It is a variable, a constant, or a (language) constructor expression. Here, `CoreHead` stands for a language constructor such as `if` and `let`. To be concrete, we will use the core language defined in Figure 2 to demonstrate our approach.

3.1.2 Surface Language. Our surface language is defined by a set of syntactic sugars, together with some language constructs in the core language. So an expression of the surface language is some core constructor expressions with sugar expressions, as defined in Figure 1.

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

$$(\texttt{SurfHead}\ x_1\ x_2\ \ldots\ x_n)\ \rightarrow_d\ \texttt{Exp}$$

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

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

```
Exp               ::=  DisplayableExp
                   |   UndisplayableExp

DisplayableExp    ::=  SurfExp
                   |   CommonExp
UndisplayableExp  ::=  Core'Exp
                   |   OtherSurfExp
                   |   OtherCommonExp

CoreHead          ::=  CoreHead'
                   |   CommonHead

Core'Exp          ::=  (CoreHead' Exp∗)

SurfExp           ::=  (SurfHead DisplayableExp∗)

CommonExp         ::=  (CommonHead DisplayableExp∗)
                   |   c       // constant value
                   |   x       // variable

OtherSurfExp      ::=  (SurfHead Exp ∗ UndisplayableExp Exp∗)

OtherCommonExp    ::=  (CommonHead Exp ∗ UndisplayableExp Exp∗)
```

Fig. 3. Our Mixed Language

Note that if the pattern is nested, we can introduce a new syntactic sugar to flatten it. One may wonder why not restricting the RHS to be a core expression CoreExp, which sounds more natural. We use surfExp to be able to allow definition of recursive syntactic sugars, as seen in the following example.

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

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

3.1.3 Mixed Language. Our mixed language for resugaring combines the surface language and the core language. The difference between our core language (CoreLang) and our surface language (SurfLang) is identified by their Head. But there are some terms in the core language should be displayed during evaluation, or we need some terms to help us getting better resugaring sequences. So we defined CommonExp, which origin from CoreLang, but can be displayed in resugaring sequences. The Core'Exp terms are terms with undisplayable CoreHead (named CoreHead'. The SurfExp terms are terms with SurfHead and all sub-expressions are displayable. The CommonExp terms are terms with displayable CoreLang's Head (named CommonHead, together with displayable sub-expressions. There exists some other expression during our resugaring process, which have displayable Head, but one or more subexpressions cannot. They are UndisplayableExp.

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

Note that some expressions with `CoreHead` contains subexpressions with `SurfHead`, they are of `CoreExp` but not in core language, we need a tricky extension for the core language's evaluator. For expression (`CoreHead` e_1 ... e_n), replacing all subexpression not in core language with different reduciable core language's term. Then getting a result after inputting the new expression `exp'` to the original blackbox stepper. If reduction appears at subexpressions after e_i replaced by, then the stepper with the extension should return (`CoreHead` e_1 ... e_i' ... e_n), where e_i' is e_i after desugaring. (an example in Fig 4) Otherwise, stepper should return `exp'`, with all the replaced subexpressions replacing back. (an example in Fig 5) The extension will not vialate properties of original core language's evaluator. It is obvious that the evaluator with the extension will reduce at the subexpression as it needs in core language, if the reduction appears in s subexpression.

```
(if (and e1 e2) true false)
```
$\Downarrow_{replace}$
```
(if tmpe1 true false)
```
$\Downarrow_{blackbox}$
```
(if tmpe1' true false)
```
$\Downarrow_{desugar}$
```
(if (if e1 e2 false))
```

Fig. 4. e1

```
(if (if true ture false) (and ...) (or ...))
```
$\Downarrow_{replace}$
```
(if (if true ture false) tmpe2 tmpe3)
```
$\Downarrow_{blackbox}$
```
(if true tmpe2 tmpe3)
```
$\Downarrow_{reback}$
```
(if true (and ...) (or ...))
```

Fig. 5. e2

3.2 Resugaring Algorithm

Our resugaring algorithm works on our mixed language, based on the reduction rules of the core language and the desugaring rules for defining the surface language. Let $\rightarrow_c$ denote the one-step reduction of the core language (based on the blackbox stepper with extension, and $\rightarrow_d$ the one-step desugaring of outermost sugar. We define $\rightarrow_m$, the one-step reduction of our mixed language, as follows.

$$\frac{(\texttt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_c\ e'}{(\texttt{CoreHead}\ e_1\ \ldots\ e_n)\ \rightarrow_m\ e'} \quad (\textsc{CoreRed})$$

$$\frac{(\texttt{SurfHead}\; x_1\; \ldots\; x_i\; \ldots\; x_n) \longrightarrow_d e,\; e_i \longrightarrow_m e_i'' \qquad \exists i.\, e[e_1/x, \ldots, e_i/x_i, \ldots, e_n/x_n] \longrightarrow_m e[e_1/x, \ldots, e_i'/x_i, \ldots, e_n/x_n]}{(\texttt{SurfHead}\; e_1\; \ldots\; e_i\; \ldots\; e_n) \longrightarrow_m (\texttt{SurfHead}\; e_1\; \ldots\; e_i''\; \ldots\; e_n)} \quad (\textsc{SurfRed1})$$

$$\frac{(\texttt{SurfHead}\; x_1\; \ldots\; x_i\; \ldots\; x_n) \longrightarrow_d e \qquad \neg\exists i.\, e[e_1/x_1, \ldots, e_i/x_i, \ldots, e_n/x_n] \longrightarrow_m e[e_1/x_1, \ldots, e_i'/x_i, \ldots, e_n/x_n]}{(\texttt{SurfHead}\; e_1\; \ldots\; e_i\; \ldots\; e_n) \longrightarrow_m e[e_1/x_1, \ldots, e_i/x_i, \ldots, e_n/x_n]} \quad (\textsc{SurfRed2})$$

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

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

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

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

3.3 Correctness

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

Then, to prove convenience, define some terms.

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

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

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

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

Example:

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

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

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

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

Lemma 3.1. *For* `Exp=(SurfHead` $e_1\ \ldots\ e_i\ \ldots\ e_n$), *if inputting* `Desugar(Exp)` *to core language's evaluator reduces the term original from* e_i *in one step, then the* $\rightarrow_m$ *will reduce* `Exp` *at* e_i.

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

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

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

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

Proof of Emulation.

For `SurfRed1` rule, (`SurfHead` $e_1\ \ldots\ e_i\ \ldots\ e_n$) $\rightarrow_m$ (`SurfHead` $e_1\ \ldots\ e_i''\ \ldots\ e_n$), where $e_i\ \rightarrow_m\ e_i''$. If `Desugar(`$e_i$`)=Desugar(`$e_i''$`)`, then `Desugar(Exp)=Desugar(Exp')`. If not, what we need to prove is that, `Desugar(Exp)` $\rightarrow_c$ `Desugar(Exp')`. Note that the only difference between `Exp` and `Exp'` is the i-th subexpression, and we have proved the lemma that the subexpression is the one to be reduced after the expression desugared totally, it will be also a recursive proof on the subexpression e_i.

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

Proof of Abstraction.

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

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

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

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

$ResugarExp'$ = ($Surfid\ Subexp_1'\ Subexp_{\ldots}'\ \ldots$), and $ResugarExp'$ is not displayed during lightweight-resugaring process. Then

- Or existing $ResugarExp$=($Surfid\ Subexp_1'\ \ \ldots\ \ Subexp_i\ \ Subexp_{\ldots}'\ \ldots$) in resugaring sequences, such that the expression after $ResugarExp$ desugaring reduces to Exp, and the reduction reduces $ResugarExp$'s sub-expression $Subexp_i$. If so, outermost syntactic sugar of $ResugarExp$ is not expanded. So if $ResugarExp'$ is not displayed, then the sugar not necessarily expanded desugars too early, which is contrary to assumption.
- Or existing $ResugarExp$=($Surfid'\ \ldots\ ResugarExp'\ \ldots$) in resugaring sequences, such that the expression after $ResugarExp$ desugaring reduces to Exp, and Exp is desugared from $ResugarExp'$'s sub-expression. If $ResugarExp'$ is not displayed, then the outermost syntactic sugar is expanded early, which is contrary to assumption.
- Or though the Exp exists, it doesn't from $ResugarExp$.

□

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

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

3.4 Implementation

Our lightweight resugaring approach is implemented using PLT Redex[Felleisen et al. 2009], which is an semantic engineering tool based on reduction semantics[Felleisen and Hieb 1992]. The whole framework is as Fig6.

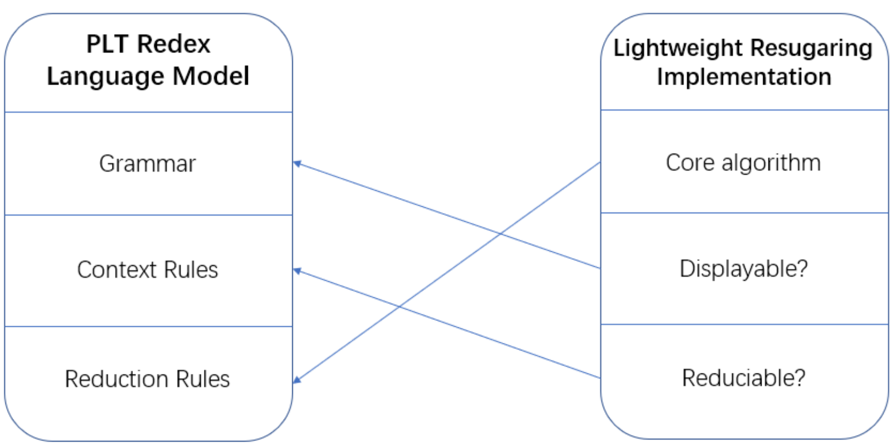

Fig. 6. framework of implementation

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

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

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

3.5 Application

We test some applications on the tool implemented using PLT Redex. Note that we set CBV's lambda calculus as terms in commonexp, because we need to output some intermediate sequences including lambda expressions in some examples. It's easy if we want to skip them.

3.5.1 simple sugar. We construct some simple syntactic sugar and try it on our tool. Some sugar is inspired by the first work of resugaring[Pombrio and Krishnamurthi 2014]. The result shows that our approach can handle all sugar features of their first work.

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

```
S ⇢ (lamN (x1 x2 x3) (x1 x2 (x1 x3)))
K ⇢ (lamN (x1 x2) x1)
I ⇢ (lamN (x) x)
```

Although SKI combinator calculus is a reduced version of lambda calculus, we can construct combinators' sugar based on call-by-need lambda calculus in our CoreLang. For expression $(S\ (K\ (S\ I))\ K\ xx\ yy)$, we get the following resugaring sequences as following.

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

For existing approach, the sugar expression should firstly desugar to

```
((lamN
   (x_1 x_2 x_3)
   (x_1 x_3 (x_2 x_3)))
  ((lamN (x_1 x_2) x_1)
   ((lamN
     (x_1 x_2 x_3)
     (x_1 x_3 (x_2 x_3)))
    (lamN (x) x)))
  (lamN (x_1 x_2) x_1)
  xx yy)
```

Then in our CoreLang, the execution of expanded expression will contain 33 steps. For each step, there will be many attempts to match and substitute the syntactic sugars. It will omit more steps for a larger expression.

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

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

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

```
(Hygienicadd e1 e2) ⇢ (let ((x e1)) (+ x e2))
```

For existing resugaring approach, if we want to get sequences of `(let ((x 2)) (Hygienicadd 1 x))`, it will firstly desugar to `(let ((x 2)) (let ((x 1)) (+ x x)))`, but it is awful because the two x in `(+ x x)` should be bind to different value. But for our lazy desugaring, the `hygienicadd` sugar does not have to desugar until necessary, so, getting following sequences.

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

The lazy desugaring is also comvinent for hygienic resugaring for non-hygienic core language. For example, `(let ((x 1)) (+ x (let ((x 2)) (+ x 1))))` may be reduced to `(+ 1 (let ((1 2)) (+ 1 1)))` by a simple core language whose `let` expression does not handle cases like that. But by writing a simple sugar Let,

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

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

```
      (Let x 1 (+ x (Let x 2 (+ x 1))))
⟶(Let x 1 (+ x (+ 2 1)))
⟶(Let x 1 (+ x 3))
⟶(+ 1 3)
⟶4
```

In practical application, we think hygiene can be easily processed by rewriting system. But our result shows lazy desugaring is really a good way to handle hygienic macro in any systems.

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

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

are typical recursive sugars. The existing resugaring approach can't process this kind of syntactic sugar easily, because boundary conditions are in the sugar itself.

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

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

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

```
      (Odd 2)
⟶ (Even (- 2 1))
```

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

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

```
   (filter e (list v1 v2 ...))
⇢ (if (e v1) (cons v1 (filter e (list v2 ...))) (filter e (list v2 ...)))
   (filter e (list)) ⇢ (list)
```

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

```
   (filter (lam (x) (and (> x 1) (< x 4))) (list 1 2 3 4))
⟶ (filter (lam (x) (and (> x 1) (< x 4))) (list 2 3 4))
⟶ (cons 2 (filter (lam (x) (and (> x 1) (< x 4))) (list 3 4)))
⟶ (cons 2 (cons 3 (filter (lam (x) (and (> x 1) (< x 4))) (list 4))))
⟶ (cons 2 (cons 3 (filter (lam (x) (and (> x 1) (< x 4))) (list))))
⟶ (cons 2 (cons 3 (list)))
⟶ (cons 2 (list 3))
⟶ (list 2 3)
```

Here, although the sugar can be processed by existing resugaring approach, it will be rebundant. The reason is that, a filter for a list of length n will match to find possible resugaring $n * (n-1)/2$ times. Thus, lazy desugaring is really important to reduce the resugaring complexity of recursive sugar.

Moreover, just like the *Odd and Even* sugar above, there are some simple rewriting systems which do not allow pattern-based rewriting. Or there are some sugars which need to be expressed by the terms in core language as conditions. Take the example of another higher-order sugar `map` as an example.

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

Get following resugaring sequences.

```
   (map (lam (x) (+ x 1)) (cons 1 (list 2)))
⟶ (map (lam (x) (+ x 1)) (list 1 2))
⟶ (cons 2 (map (lam (x) (+ 1 x)) (list 2)))
⟶ (cons 2 (cons 3 (map (lam (x) (+ 1 x)) (list))))
⟶ (cons 2 (cons 3 (list)))
⟶ (cons 2 (list 3))
⟶ (list 2 3)
```

Note that the `let` term is to limit the subexpression only appears once in RHS. In this example, we can find that the list `(cons 1 (list 2))`, though equal to `(list 1 2)`, is represented by core language's term. So it will be difficult to handle the inline boundary conditions by rewriting system. But our approach is easy to handle cases like this.

3.6 Compare to previous work

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

- *Lightweight* As the example at sec3.5.1, the match and substitution process searchs all intermediate sequences many times. It will cause huge cost for a large program. So out approach—only expanding a syntactic sugar when necessarily, is a lightweight approach.
- *Friendly to hygienic macro* Previous hygienic resugaring approach use a new data structure—abstract syntax DAG, to process resugaring of hygienic macros. Our approach simply finds hygienic error after expansion, and gets the correct reduction instead.
- *More syntactic sugar features* The ability of processing non-pattern-based (Todo: inline?) recursive sugar is a superiority compared to previous work. The key point is that recursive syntactic sugar must handle boundary conditions. Our approach handle them easily by not necessarily desugaring all syntactic sugars. Higher-order functions, as an important feature of functional programming, was supported by many daily programming languages. Lazy desugaring makes writing higher-order sugars easier.

4 STATIC APPROACH

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

4.1 Inference Automaton

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

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

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

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

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

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

4.1.3 *IFA of or.*

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

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

4.1.4 *Definition of IFA.*

Defination 4.1 (Inference Automaton). *An inference automaton (IFA) of syntactic structure* (`Headid` $e_1 \ldots e_n$) *is a 5-tuple,* $(Q, \Sigma, q_0, F, \delta)$*, consisting of*

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

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

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

In IFA, state transition does not depend on input. The only input IFA accepts is the term to be evaluated with this syntactic structure. The state transition is through pattern matching on the evaluation result of the term in the previous node. Note that IFA is associated with syntactic structure. At Each IFA only represents the current evaluation of a syntactic structure. The state indicates that some sub-expressions of the syntactic structure have been evaluated, and the rest have not.

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

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

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

Proof of Lemma. □

4.2 Convert inference rules to IFA

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

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

$$\frac{(\texttt{Headid}\ v_1 \dots v_p\ e_1 \dots e_i \dots e_q) \longrightarrow (\texttt{Headid}\ v_1 \dots v_p\ e_1 \dots e_i' \dots e_q)}{e_i \longrightarrow e_i'} \quad \text{(E-Head)}$$

$$(\texttt{Headid}\ v_1 \dots v_p\ e_1 \dots e_q) \longrightarrow e$$

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

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

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

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

Assumption 3. The rules satisfy the determinacy of one-step evaluation.

By assumption 3, we can get the following lemma, which points out the feasibility of using a node in IFA to represent the evaluation process of sub-expressions.

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

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

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

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

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

For a term like $(\texttt{Headid}\ e_1\ \dots\ e_n)$, considering that $e_1 \cdots e_n$ are not value, According to Assumption 3, we have the unique rule r of *Headid* for one-step evaluation. Let node q corresponding to r be q_0.

If r is a context rule for e_i, let the term of q_0 be build a new node q and add it into Q. The term of q is e_i. And the symbol table is set to empty. Assume that the evaluation of e_i results in v_i, we get term $(\texttt{Headid}\ e_1 \dots e_{i-1}\ v_i\ e_{i+1} \dots e_n)$. For each possible value of v_i, choose the rules that should be used.

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

□

4.3 Convert IFA to Inference Rules

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

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

4.4 Syntactic Sugar

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

DEFINATION 4.3. *Considering the following syntactic sugar*

$$(\texttt{SurfHead}\, x_1 \ \ldots \ x_n) \ \to_d \ e,$$

the IFA of `SurfHead` *is defined as the IFA of syntactic structure* `SurfHead'` *whose inference rule is*

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

5 RELATED WORK

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

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

The type resugaring work[Pombrio and Krishnamurthi 2018] indicates that it is possible to automatically construct surface language's semantics by unification. But after trying to do this as type resugaring does, we found it impossible because Todo: the reason

Galois slicing for Imperative Functional Programs[Ricciotti et al. 2017] is a work for dynamic analyzing functional programs during execution. The forward component of the Galois connection maps a partial input x to the greatest partial output y that can be computed from x; the backward component of the Galois connection maps a partial output y to the least partial input x from which we can compute y. Our approach used a similiar idea on slicing expressions and processing on subexpressions. The dynamic approach is like the forward component, so the method to handle side effects in functional programs may be useful for a better resugaring with side effects.

Macros as Multi-Stage Computations[Ganz et al. 2001] is a work similar to lazy expansion for macros. Some other researches[Rompf and Odersky 2010] about multi-stage programming[Taha 2003] indicate that it is an useful idea for implementing domain-specific languages. Our resugaring approach combines the idea of multi-stage programming with syntactic sugars, which achieves a better approach. Macro systems in some language (such as Racket[Flatt 2012]) have support lazy expansion. Our dynamic approach is a combination of existing resugaring and lazy expansion, which achieves a more powerful approach.

Origin tracking[Deursen et al. 1992] is about tracking the origins of terms in rewriting system, which is similar to existing resugaring approaches. Our approach, as an unidirectional resugaring, is quite suitable for domain-specific languages. The reason is that syntactic sugars used to be seen as an extension of host language, while our approach regards them as components of a new language.

Ziggurat[Fisher and Shivers 2006] is a semantic-extension framework. It allows defining new macros with semantics based on existing terms in a language. It is quite useful for static analysis on macros. Instead of semantics based on core language, the reduction rules of sugar got by our static approach is independent of core language, which may be more concise for static analysis.

Addition to PLT Redex[Felleisen et al. 2009] we used to engineer the semantics, there are some other semantics engineering tools[Rosu and Serbanuta 2010; Vergu et al. 2015] which aim to test or verify the semantics of languages. The methods of these researches can be easily combined with our static approach.

6 CONCLUSION

In this paper, we purpose a new approach (see Fig **??**) or resugaring mixed with a dynamic apporach and static approach, which has some advances compared to existing approaches. The two approaches are seemingly similar in lazy desugaring. Essentially, we would see the static approach is the abstract(todo:another express?) of dynamic approach. In the dynamic approach, the most important part is *reduction in mixed language* (see in sec **??**), which decides whether reducing the subexpression or desugaring the outermost sugar. Reducing subexpressions are just the same as context rules in static approach; desugaring the outermost sugar is similar to reduction rules in static approach. However, the reduction rules is more convinent and efficent than dynamic resugaring, because the static approach evolves a process like abstract interpretation[Cousot and Cousot 1977], then reduces many steps executed in core language. Moreover, the semantics got by static approach make it possible to do some optimization at the surface language level, which is important for implementing a DSL. In contrast, the dynamic approach is more powerful by supporting recursive sugars' resugaring. Besides, the rewriting based on reduction semantics makes the sugar represented in many ways.

As we mentioned before, the original intent of our research is finding a better method (or building a tool) for implementing DSL. We could see static approach is better for achieving the goal, because getting the semantics of DSL (based on syntactic sugar) will be very useful for applying any other techniques on the DSL. But it will be better if the defects of expressiveness in the static approach can be solved. So the first future work may be achieving a more powerful static approach as our dynamic approach. Then we will carefully design a core language for as the host language of our dream system and find a better type resugaring approach for the system. Finally, a general optimazation method for DSL in our system is needed.

REFERENCES

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

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.

A. Van Deursen, P. Klint, and F. Tip. 1992. Origin Tracking. *JOURNAL OF SYMBOLIC COMPUTATION* 15 (1992), 523–545.

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

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2018. A programmable programming language. *Commun. ACM* 61, 3 (2018), 62–71.

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (Sept. 1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

David Fisher and Olin Shivers. 2006. Static Analysis for Syntax Objects. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. Association for Computing Machinery, New York, NY, USA, 111–121. https://doi.org/10.1145/1159803.1159817

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

Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence, Italy) *(ICFP '01)*. Association for Computing Machinery, New York, NY, USA, 74–85. https://doi.org/10.1145/507635.507646

P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (01 1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308 arXiv:https://academic.oup.com/comjnl/article-pdf/6/4/308/1067901/6-4-308.pdf

Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: Lifting Evaluation Sequences through Syntactic Sugar. *SIGPLAN Not.* 49, 6 (June 2014), 361–371. https://doi.org/10.1145/2666356.2594319

Justin Pombrio and Shriram Krishnamurthi. 2015. Hygienic Resugaring of Compositional Desugaring. *SIGPLAN Not.* 50, 9 (Aug. 2015), 75–87. https://doi.org/10.1145/2858949.2784755

Justin Pombrio and Shriram Krishnamurthi. 2018. Inferring Type Rules for Syntactic Sugar. *SIGPLAN Not.* 53, 4 (June 2018), 812–825. https://doi.org/10.1145/3296979.3192398

Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. 2017. Imperative Functional Programs That Explain Their Work. *Proc. ACM Program. Lang.* 1, ICFP, Article 14 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110258

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *SIGPLAN Not.* 46, 2 (Oct. 2010), 127–136. https://doi.org/10.1145/1942788.1868314

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

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

Vlad Vergu, Pierre Neron, and Eelco Visser. 2015. DynSem: A DSL for Dynamic Semantics Specification. In *26th International Conference on Rewriting Techniques and Applications (RTA 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Maribel Fernández (Ed.), Vol. 36. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 365–378. https://doi.org/10.4230/LIPIcs.RTA.2015.365

A APPENDIX

Text of appendix ...