Metadata of the chapter that will be visualized in SpringerLink

Book Title	Functional and Logic Programming	
Series Title		
Chapter Title	A Lazy Desugaring System for Evaluating Programs with Sugars	
Copyright Year	2022	
Copyright HolderName	Springer Nature Switzerland AG	
Author	Family Name	**Yang**
	Particle	
	Given Name	**Ziyi**
	Prefix	
	Suffix	
	Role	
	Division	School of Computing
	Organization	National University of Singapore
	Address	Singapore, Singapore
	Email	yangziyi@u.nus.edu
Author	Family Name	**Xiao**
	Particle	
	Given Name	**Yushuo**
	Prefix	
	Suffix	
	Role	
	Division	Key Lab of High Confidence Software Technologies
	Organization	Ministry of Education
	Address	Beijing, China
	Division	School of Computer Science
	Organization	Peking University
	Address	Beijing, China
	Email	xiaoyushuo@pku.edu.cn
Author	Family Name	**Guan**
	Particle	
	Given Name	**Zhichao**
	Prefix	
	Suffix	
	Role	
	Division	Key Lab of High Confidence Software Technologies
	Organization	Ministry of Education
	Address	Beijing, China
	Division	School of Computer Science
	Organization	Peking University
	Address	Beijing, China

	Email	guanzhichao@pku.edu.cn
Corresponding Author	Family Name	**Hu**
	Particle	
	Given Name	**Zhenjiang**
	Prefix	
	Suffix	
	Role	
	Division	Key Lab of High Confidence Software Technologies
	Organization	Ministry of Education
	Address	Beijing, China
	Division	School of Computer Science
	Organization	Peking University
	Address	Beijing, China
	Email	huzj@pku.edu.cn

Abstract

Extending a programming language with syntactic sugars is common practice in language design. Given a core language, one can define a surface language on top of it with sugars. We propose a lazy desugaring system, which can generate the evaluation sequences of sugar programs in the syntax of the surface language. Specifically, we define an evaluation strategy on a mixed language which combines syntactic sugars with the core language. We formulate two properties, *emulation* and *laziness*, and prove that the evaluation strategy produces correct evaluation sequences. Besides, we have implemented a system based on this novel method and demonstrate its usefulness with several examples.

A Lazy Desugaring System for Evaluating Programs with Sugars

Ziyi Yang[1], Yushuo Xiao[2,3], Zhichao Guan[2,3], and Zhenjiang Hu[2,3(✉)]

[1] School of Computing, National University of Singapore, Singapore, Singapore
yangziyi@u.nus.edu
[2] Key Lab of High Confidence Software Technologies, Ministry of Education, Beijing, China
[3] School of Computer Science, Peking University, Beijing, China
{xiaoyushuo,guanzhichao,huzj}@pku.edu.cn

Abstract. Extending a programming language with syntactic sugars is common practice in language design. Given a core language, one can define a surface language on top of it with sugars. We propose a lazy desugaring system, which can generate the evaluation sequences of sugar programs in the syntax of the surface language. Specifically, we define an evaluation strategy on a mixed language which combines syntactic sugars with the core language. We formulate two properties, *emulation* and *laziness*, and prove that the evaluation strategy produces correct evaluation sequences. Besides, we have implemented a system based on this novel method and demonstrate its usefulness with several examples.

AQ1

1 Introduction

Syntactic sugar, first coined by Landin [13] in 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 been proved to be very useful for defining domain-specific languages (DSLs) and extending existing languages [4,6]. Unfortunately, when syntactic sugar is eliminated by transformation, it obscures the relationship between the user's source program and the transformed program. As a result, a programmer who only knows the surface language cannot understand the execution of programs in the core language, which makes the debugging of programs with sugars hard.

Resugaring [14,15] is a powerful existing method to resolve this problem. It reverses the application of the desugaring transformation. As a typical example of resugaring, consider the sugars `Or1` and `Not`, defined by the following desugaring rules[1].

[1] Throughout the paper, we use `#t` and `#f` to represent the Boolean constants true and false, respectively.

Z. Yang and Y. Xiao—Co-first authors, contributing equally to this work.

This work was partly supported by the National Key Research and Development Program of China (No. 2021ZD0110202).

M. Hanus and A. Igarashi (Eds.): FLOPS 2022, LNCS 13215, pp. 1–19, 2022.
https://doi.org/10.1007/978-3-030-99461-7_14

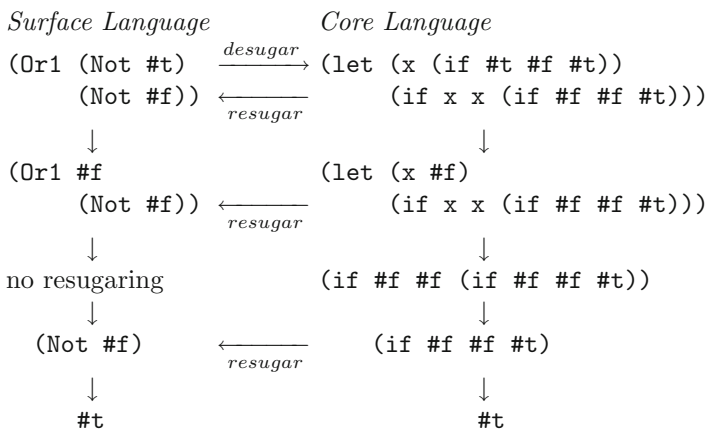

Fig. 1. A resugaring example.

$$\texttt{(Or1 } \boldsymbol{t}_1 \ \boldsymbol{t}_2\texttt{)} \stackrel{def}{=\!=} \texttt{(let (x } \boldsymbol{t}_1\texttt{) (if x x } \boldsymbol{t}_2\texttt{))}$$
$$\texttt{(Not } \boldsymbol{t}_1\texttt{)} \stackrel{def}{=\!=} \texttt{(if } \boldsymbol{t}_1 \texttt{ \#f \#t)}$$

The resugaring process for

```
(Or1 (Not #t) (Not #f))
```

is shown in Fig. 1. The sequence of terms on the left shows the evaluation steps in the surface language, which is obtained from the evaluation sequence of the desugared program (in the core language) on the right by repeated attempts of reverse expansion of each sugar.

While this approach is natural, there are two practical problems. First, as the reverse expansion of sugars needs to match the desugared terms against the desugaring rules to check whether they can be resugared, it would be very expensive if the surface program uses a large number of syntactic sugars, or some syntactic sugars are desugared to complex core terms. Second, in the resugaring process, many core programs cannot be reverted to surface programs, which means that many attempts at reverse application of desugaring rules fail and introduce lots of useless work.

In this paper, we propose a lazy desugaring system, which *produces the evaluation sequence in the surface language without reverse desugaring*. Our key observation is that if we consider desugaring rules as reduction rules like those in the core language, then the evaluation sequence of a surface program should exist in the reduction sequence by these reduction rules. To see this, recalling the example in Fig. 1, we can see from Fig. 2 that the reduction sequence can

be generated by the given desugaring rules and the reduction rules of the core language; the underlined part is the same as the resugaring sequence on the left of Fig. 1.

```
          (Or1 (Not #t) (Not #f))
desugar-> (Or1 (if #t #f #t) (Not #f))
core->    (Or1 #f (Not #f))
desugar-> (let (x #f) (if x x (Not #f)))
core->    (if #f #f (Not #f))
core->    (Not #f)
desugar-> (if #f #f #t)
core->    #t
```

Fig. 2. Proper desugaring for resugaring.

Attention should be paid here. There could be many possible reduction sequences if we do not restrict how to apply desugaring rules. For instance, Fig. 3 gives another possible evaluation sequence. Here, from this sequence, we could not extract the sequence we want, because the term `(Or1 #f (Not #f))` is lost. How can we make sure that a sequence which contains all the evaluation steps wanted is produced?

```
          (Or1 (Not #t) (Not #f))
desugar-> (let (x (Not #t)) (if x x (Not #f)))
desugar-> (let (x (if #t #f #t)) (if x x (Not #f)))
core->    (let (x #f) (if x x (Not #f)))
core->    (if #f #f (Not #f))
core->    (Not #f)
desugar-> (if #f #f #t)
core->    #t
```

Fig. 3. Improper desugaring.

The key insight of our approach is that we can delay the application of desugaring rules (sugar expansion) until it becomes necessary so that later reverse expansion in the original resugaring becomes unnecessary. To do so, we treat the surface language and the core language as one mixed language, regard desugaring rules as reduction rules of the mixed language, and derive the context rules of the mixed language to indicate when desugaring should take place. Our main technical contributions can be summarized as follows.

- We propose a lazy desugaring method, which evaluates sugar programs on the surface level. It guarantees that the evaluation sequence of a program in the mixed language is *correct* in the sense that it corresponds to the evaluation sequence of the fully desugared program in the core language, and that it is *sufficient* (*complete*) in the sense that it contains all evaluation steps we want in the surface language.
- We present a novel algorithm to calculate the context rules and the reduction rules for syntactic sugars to achieve lazy desugaring. Using the algorithm, we can get a new reduction strategy for the mixed language, based on which the evaluation sequence in the syntax of the surface language can be obtained.
- We have implemented a system based on this approach, and tested it with many non-trivial examples, which shows the promise of the system.

The rest of our paper is organized as follows. We start with an overview of our approach in Sect. 2. We give the template for language definition in Sect. 3, which makes clear what languages are supported. We then present the algorithm of lazy desugaring with its properties in Sect. 4. We briefly discuss the implementation of the system, and give some examples in Sect. 5. We discuss related work in Sect. 6 and conclude the paper in Sect. 7.

2 Overview

In this section, we give a brief overview of our approach. Given a very tiny core language and a surface language defined by a set of syntactic sugars, we shall demonstrate how we can obtain the evaluation sequence of a program with sugars in the syntax of the surface language by lazy desugaring.

Consider the following simple core language, which contains Boolean expressions using the `if` construct.

$$\begin{aligned} t ::= & \texttt{(if } t\ t\ t\texttt{)} \\ \mid & \texttt{\#t} \\ \mid & \texttt{\#f} \end{aligned}$$

The semantics of the language is given by reduction semantics: we have two reduction rules:

$$\begin{aligned} \texttt{(if \#t } t_1\ t_2\texttt{)} &\rightarrow t_1 \\ \texttt{(if \#f } t_1\ t_2\texttt{)} &\rightarrow t_2 \end{aligned}$$

together with the following context rules specifying the reduction order.

$$\begin{aligned} C ::= & \texttt{(if } C\ t\ t\texttt{)} \\ \mid & \bullet \qquad \triangleright \ \textit{context hole} \end{aligned}$$

The surface language is defined by two syntactic sugars:

$$\begin{aligned} \texttt{(And } \boldsymbol{t}_1\ \boldsymbol{t}_2\texttt{)} &\stackrel{def}{=\!=} \texttt{(if } \boldsymbol{t}_1\ \boldsymbol{t}_2\ \texttt{\#f)} \\ \texttt{(Or } \boldsymbol{t}_1\ \boldsymbol{t}_2\texttt{)} &\stackrel{def}{=\!=} \texttt{(if } \boldsymbol{t}_1\ \texttt{\#t}\ \boldsymbol{t}_2\texttt{)} \end{aligned}$$

Now let us demonstrate how to evaluate `(And (Or #t #f) (And #f #t))` by lazy desugaring to obtain the evaluation sequence on the surface level as follows.

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

Step 1: Calculating Context Rules and Reduction Rules for Sugars

In lazy desugaring, we first decide when a sugar should be desugared. To this end, from the context rules of the core language, we automatically derive the following context rules for the sugars.

$$\begin{array}{rcl} C & ::= & (\texttt{And}\ C\ t_2) \\ & | & (\texttt{Or}\ C\ t_2) \\ & | & \bullet \end{array}$$

The idea of derivation will be discussed in Sect. 4. Intuitively, from the context rules of `(if` $t_1\ t_2$ `#f)`, we can see that the condition, t_1, is always evaluated first, so `(And` $t_1\ t_2$`)` should also have t_1 evaluated first. This is indicated by the context rule of `And`. Similarly, we can calculate the context rule for `Or`.

Step 2: Forming Mixed Language with Mixed Reduction Rules

To treat desugaring rules and the reduction rules (of the core language) in one reduction system, we mix the surface language with the core language as in Fig. 4, and define $\rightarrow_m$, a one-step reduction for the mixed language (the letter m stands for "mixed"). It is derived from the reduction rules of the core language and the desugaring rules of the surface language. Note that the desugaring rules are a bit different from the initial definition. For instance, the desugaring rule `(And` $\boldsymbol{t}_1\ \boldsymbol{t}_2$`)` $\stackrel{def}{=\!=}$ `(if` $\boldsymbol{t}_1\ \boldsymbol{t}_2$ `#f)` has been changed to the reduction rule `(And` $v_1\ t_2$`)` $\rightarrow$ `(if` $v_1\ t_2$ `#f)` in Fig. 4, indicating that this reduction rule can be applied only when the first argument of `Add` has been reduced to a value. This change in fact follows from the context rules ontained at step 1.

Now, by using $\rightarrow_m$, we can get the evaluation sequence in the mixed language for the program

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

based on the computation order determined by the context rules obtained at step 1. The evaluation sequence in the mixed language is shown below.

```
    (And (Or #t #f) (And #f #t))
→m (And (if #t #t #f) (And #f #t))
→m (And #t (And #f #t))
→m (if #t (And #f #t) #f)
→m (And #f #t)
→m (if #f #t #f)
→m #f
```

$$
\begin{array}{l}
t ::= \texttt{(if } t_1\ t_2\ t_3\texttt{)} \\
\quad \mid \texttt{(And } t_1\ t_2\texttt{)} \\
\quad \mid \texttt{(Or } t_1\ t_2\texttt{)} \\
\quad \mid \texttt{\#t} \\
\quad \mid \texttt{\#f}
\end{array}
$$

(a) Syntax.

$$
\begin{array}{l}
C ::= \texttt{(if } C\ t_2\ t_3\texttt{)} \\
\quad \mid \texttt{(And } C\ t_2\texttt{)} \\
\quad \mid \texttt{(Or } C\ t_2\texttt{)} \\
\quad \mid \bullet
\end{array}
$$

(b) Context Rules.

$$
\begin{array}{r}
\texttt{(And } v_1\ t_2\texttt{)} \rightarrow \texttt{(if } v_1\ t_2\ \texttt{\#f)} \\
\texttt{(Or } v_1\ t_2\texttt{)} \rightarrow \texttt{(if } v_1\ \texttt{\#t } t_2\texttt{)} \\
\texttt{(if \#t } t_1\ t_2\texttt{)} \rightarrow t_1 \\
\texttt{(if \#f } t_1\ t_2\texttt{)} \rightarrow t_2
\end{array}
$$

(c) Reduction Rules.

Fig. 4. A small mixed language.

Step 3: Removing Unnecessary Terms

As seen above, our evaluation sequence in the mixed language may contain constructs in the core language (e.g. `(if #t (And #f #t) #f)`). Since our goal is to show the evaluation sequence of sugar programs, we give a flexible method to clearly specify a *filter* showing which terms should be displayed. (A default filter can be generated automatically.) For example, we may define the following subset of the mixed language as a filter for displaying:

$$
\begin{array}{l}
dt ::= \texttt{(And } dt\ dt\texttt{)} \\
\quad \mid \texttt{(Or } dt\ dt\texttt{)} \\
\quad \mid \texttt{\#t} \\
\quad \mid \texttt{\#f}
\end{array}
$$

With this filter, the sugars `And` and `Or`, together with Boolean constants `#t` and `#f`, will be displayed. Notice that we make Boolean values displayable even if they are in the core language. By clearly specifying what should be displayed, we can always get the evaluation sequences we need. This practical step is not essential to our system, so we will not go into detail in the rest of the paper.

In short, given a core language and a surface language defined by syntactic sugars, the major effort to build an evaluator of the surface language is to derive context rules and reduction rules from sugar definitions, which can be done automatically by our method and will be explained in detail in the rest of this paper.

3 Defining Languages and Sugars

As seen in previous sections, by introducing sugars we construct a language hierarchy, which contains a "core language", and "surface languages" extended by sugars. Since our approach needs to inspect and manipulate context rules and reduction rules explicitly, we give a language template for the definition of the core language and sugars, which stipulates what languages are allowed in our setting. These definitions and notations will be used when discussing the algorithm and its properties in Sect. 4.

Syntax	
$t \ ::= (\underline{\texttt{head}} \ t_1 \ \cdots \ t_n)$	▷ language constructs
$v \ ::= (\underline{\texttt{head}} \ t_1/v_1 \ \cdots \ t_n/v_n)$	▷ values
Context	
$C ::= (\underline{\texttt{head}} \ t/v \ \cdots \ C \ t/v \ \cdots)*$	▷ zero or more context rules
$\mid \bullet$	▷ context hole
Notion of reduction	
$\mathcal{R}((\underline{\texttt{head}} \ t/v \ \cdots))$	▷ one-step reduction without context

Fig. 5. Basic template for core language definition.

3.1 Core Language

We follow the syntax convention of Lisp for a core language, using S-expressions to represent programs, and we use reduction semantics to formalize the semantics of the core language.

The basic *template* for defining syntax, contexts, and reduction rules is shown in Fig. 5. This is merely a template for language definitions, with which we can encode more complex languages. In the template, all elements that are underlined will be replaced with some language-specific constructs. For example, $\underline{\texttt{head}}$ should be chosen from a set of language constructs specified by the user, such as `lambda`, `if`, `let`, etc. So the production of t may become $t ::= (\texttt{if}\ t_1\ t_2\ t_3)$. In addition, a language also needs to specify a set of *values*, which is the set of terms that can not be further reduced. Values must be defined in the format of v in Fig. 5. Here t/v means either t or v, and whether it is t or v is fixed in a given language. To encode the reduction semantics of the language, the user also needs to specify a set of context rules, in the form of the right-hand side of the production of C. Finally, the *notion of reduction* (as described in the literature [5]) needs to be defined, which is the one-step reduction without context. For simplicity, we define it using a partial function, $\mathcal{R}$, which yields the reduced term if the input is reducible, and $\perp$ if the input is not reducible. The notion of reduction should be *extensible*, in the sense that it can be extended to any mixed language. The exact meaning of "extensible" will be made clear in Sect. 3.2.

The languages defined using this template, while having a restricted form of syntax, can be arbitrarily complex, since the notion of reduction, $\mathcal{R}$, can be any partial function as long as it is extensible. We encode the language in the running example, the Boolean language, using our template, in Fig. 6. Note that constants can be encoded as language constructs with zero arguments, such as `(true)`. And `#t` is merely a shorthand of `(true)`. In Sect. 4, we use C_c to denote contexts generated by the non-terminal C of any given core language. In this example, one valid context is

```
(if (if • #t #f) #f #t).
```

Syntax

t ::= `(if` t_1 t_2 t_3`)` ▷ language constructs
| `(true)` | `(false)` ▷ constants
v ::= `(true)` | `(false)` ▷ constants are values

Context

C ::= `(if` C t_1 t_2`)` ▷ evaluating the condition first
| $\bullet$ ▷ context hole

Notion of reduction

$\mathcal{R}$(`(if (true)` t_2 t_3`)`) $= t_2$ ▷ one-step reduction without context (`true` branch)
$\mathcal{R}$(`(if (false)` t_2 t_3`)`) $= t_3$ (`false` branch)

Fig. 6. Core language example: boolean expressions.

Sugar definition

(Sg $\boldsymbol{t}_1/\boldsymbol{v}_1 \ \cdots \ \boldsymbol{t}_n/\boldsymbol{v}_n$) $\stackrel{def}{=\!=} t$

Fig. 7. Sugar definition.

Given a language definition, we can describe a small-step reduction of the core language with a partial function as follows.

$$\mathcal{R}_c(t) = \begin{cases} C_c[t'] & \triangleright \text{ if } t = C_c[t_0] \text{ and } \mathcal{R}(t_0) = t' \\ \bot & \triangleright \text{ otherwise} \end{cases} \tag{1}$$

And we naturally require that the reduction of any term t is deterministic, i.e., there does not exist more than one term t' such that t can be reduced to.

3.2 Mixed Language

Given a core language, we can define syntactic sugars on top of it. In the theoretical discussion, we assume for simplicity that only one sugar is defined based on the given language. The sugar definition follows a strict pattern, as illustrated in Fig. 7. We use Sg to denote a new sugar name, and use $\boldsymbol{t}_i$'s and $\boldsymbol{v}_i$'s as metavariables (for terms and values), which appear on the right-hand side (RHS). To distinguish the constructs from the core language and the names of sugars, we make the head of the core language constructs in lowercase and the first letter of sugar names in uppercase. Below is an example of a sugar definition based on pre-defined sugars `And`, `Or` and core language's construct `not`.

$$\texttt{(Sg}_1 \ \boldsymbol{t}_1 \ \boldsymbol{t}_2 \ mt_3\texttt{)} \stackrel{def}{=\!=} \texttt{(And (Or } \boldsymbol{t}_1 \ \boldsymbol{t}_3\texttt{) (not } \boldsymbol{t}_2\texttt{))}$$

To simplify later discussion of our algorithm, we assume that any metavariable on the left-hand side (LHS) of a sugar definition appears only once on the

Syntax	
t ::= ...	▷ terms in the core language form
| (Sg $t_1 \cdots t_n$)	▷ sugars
v ::= ...	▷ values in the core language
Context	
C ::= ...	▷ context rules of the core language and hole
| (Sg $t/v \cdots C\ t/v \cdots$)$*$	▷ zero or more Sg's context rules
Notion of reduction	
(head $t/v \cdots$) $\rightarrow_{m_1} t$	▷ remained reduction rules of core language
(Sg $t/v \cdots$) $\rightarrow_{m_1} t$	▷ reduction rule derived from desugaring rules

Fig. 8. Template for the mixed language.

RHS (*linear expansion*). (The restriction can be lifted with a simple extension.) Given a sugar definition, we now define the mixed language. The mixed language simply allows the sugar to appear as any part of a term. Formally, the syntax of the mixed language is defined in Fig. 8. Notice that the arguments of a core language's `head` can also be a sugar now, like in

```
(if (And #t #f) #f #t).
```

We use $\mathcal{D}$ to denote the outermost desugaring function induced by the sugar definition. For example, we have

$$\mathcal{D}(\texttt{(And (And \#t \#f) \#t)}) = \texttt{(if (And \#t \#f) \#t \#f)}$$

for the `And` sugar above. Then we can naturally define the fully desugaring function, $\mathcal{D}_F$, which works as traditional desugaring, recursively expanding all sugars of the input term. Formally, $\mathcal{D}_F$ is defined as follows.

$$\mathcal{D}_F(\texttt{(Sg } t_1 \cdots t_n\texttt{)}) = \mathcal{D}_F(\mathcal{D}(\texttt{(Sg } t_1 \cdots t_n\texttt{)}))$$
$$\mathcal{D}_F(\texttt{(head } t_1 \cdots t_n\texttt{)}) = \texttt{(head } \mathcal{D}_F(t_1) \cdots \mathcal{D}_F(t_n)\texttt{)}$$

With the definition of $\mathcal{D}_F$, we can now make the meaning of the extensible property of the core language clear.

Definition 1 (Extensible)**.** *The notion of reduction $\mathcal{R}$ of a core language is extensible, if for any possible sugar, and for any term t in the mixed language,*

$$\mathcal{D}_F(\mathcal{R}(t)) = \mathcal{R}(\mathcal{D}_F(t)),\ \textit{if } \mathcal{R}(t) \neq \bot.$$

It is saying that, for any term t that is reducible by the notion of reduction, reducing it first and then fully desugaring it will be the same as fully desugaring it first and then reducing it. One can easily check that reductions of our familiar language constructs are usually extensible. For example, the reduction rules of `if` is extensible, since it treats t_2 and t_3 as a whole, and thus the order of the reduction and the desugaring does not matter.

4 Lazy Desugaring Algorithm

As shown in Sect. 2, given *a core language*, *sugar definition* and *any term with sugars*, we can get the *evaluation sequence* of the term as the output. To obtain the sequence, the first step is to generate the reduction semantics for the mixed language, which is non-trivial. Then the sequence can be obtained easily by recursively applying the one-step reduction on the term in the mixed language. In this section, we start by describing a procedure that generates the reduction semantics of the mixed language in Sect. 4.1, and then show that the semantics gives the correct evaluation sequence in Sect. 4.2.

4.1 Algorithm Desciption

Supposing that we have a core language, we use the following function `getRules` to generate the reduction semantics for a given desugaring rule (i.e., a sugar definition):

$$D_{\text{LHS}} \stackrel{def}{=\!=} t_{\text{RHS}}, \quad \text{where } D_{\text{LHS}} = (\texttt{Sg}\ \ \boldsymbol{t}_1/\boldsymbol{v}_1\ \cdots\ \boldsymbol{t}_n/\boldsymbol{v}_n),$$

and the generated semantics consist of zero or more *context rules* of the sugar, and exactly one *reduction rule* corresponding to the original desugaring rule. These rules, together with the context rules and reduction rules of the core language, form the semantics of the mixed language. The following function `getRules` calculates the context rules and the reduction rule of `Sg`, which are put in a set.

$$\texttt{getRules}(D_{\text{LHS}} \stackrel{def}{=\!=} t_{\text{RHS}}) = \begin{cases} \{[\boldsymbol{t}_i := C]\, D_{\text{LHS}}\} \cup \\ \texttt{getRules}\left([\boldsymbol{t}_i := \boldsymbol{v}_i]\left(D_{\text{LHS}} \stackrel{def}{=\!=} t_{\text{RHS}}\right)\right) \\ \qquad \triangleright \text{ if } \exists\ i, C_c, \text{ s.t. } t_{\text{RHS}} = C_c[\boldsymbol{t}_i] \\ \{D_{\text{LHS}} \rightarrow_{m_1} t_{\text{RHS}}\} \\ \qquad \triangleright \text{ otherwise} \end{cases}$$

The substitutions are on the metalanguage level. For example, $[\boldsymbol{t}_i := C]$ means substituting a context hole C for metavariable $\boldsymbol{t}_i$. Metavariables $\boldsymbol{t}_i$ and $\boldsymbol{v}_i$ are implicitly replaced with symbols t_i and v_i, respectively (because our sugar definition and context rules use different notations). The substitution produces a new *rule*. Rules need to be interpreted properly to represent actual contexts. Intuitively, the function tries to match the RHS of sugar definition with context rules of the core language, and calculates the context rules of the sugar accordingly. When the expansion cannot be matched with any core context rules, we acquire the last item of the returned list, $D_{\text{LHS}} \rightarrow_{m_1} t_{\text{RHS}}$, which is the reduction rule (notion of reduction) of the sugar in the mixed language.

To demonstrate how `getRules` runs, we explain how the following invocation executes.

$$\texttt{getRules}((\texttt{And}\ \boldsymbol{t}_1\ \boldsymbol{t}_2) \stackrel{def}{=\!=} (\texttt{if}\ \boldsymbol{t}_1\ \boldsymbol{t}_2\ \texttt{\#f}))$$

In the first step, t_{RHS} is $C_c[\boldsymbol{t}_1]$, where C_c is `(if` $\bullet$ $\boldsymbol{t}_2$ `#f)`. The metavariable $\boldsymbol{t}_1$ is matched with the hole, so the first rule to be output is the context rule `(And` C $\boldsymbol{t}_2$`)`, indicating that the first operand of `And` sugar should be evaluated first. Then the algorithm runs recursively, calling

$$\texttt{getRules((And } \boldsymbol{v}_1 \ \boldsymbol{t}_2\texttt{)} \stackrel{def}{=\!=} \texttt{(if } \boldsymbol{v}_1 \ \boldsymbol{t}_2 \ \texttt{\#f))}.$$

This time there does not exist i, C_c, such that $t_{\text{RHS}} = C_c[\boldsymbol{t}_i]$, so the second rule output is the reduction rule

$$\texttt{(And } \boldsymbol{v}_1 \ \boldsymbol{t}_2\texttt{)} \rightarrow_{m_1} \texttt{(if } \boldsymbol{v}_1 \ \boldsymbol{t}_2 \ \texttt{\#f)}.$$

Finally, with the context rule and the reduction rule, the reduction semantics of the mixed language can be formed (as partly seen in Fig. 4), following the template in Fig. 8.

Based on the setting in the previous section, we can generate the semantics of the mixed language by the rules of the core language and the calculated rules. Following the definition of C_c and $\mathcal{R}_c$ in Sect. 3.1, we define the contexts of the mixed language, C_m, and the partial reduction function, $\mathcal{R}_m$, based on the mixed language's semantics as well. If there are more than one sugar definition, we calculate their rules and add them to the mixed language one by one. With the first sugar's rules calculated, the language mixed by the core and the first sugar becomes the new core language of the second sugar, and so on. If one desugaring rule's RHS depends on another syntactic sugar, the previous one's rules should be obtained first. Therefore, the context rules of sugars derived by the algorithm must not be cyclically dependent for mutual recursive sugars.[2] Finally, given any term in the mixed language, we can evaluate it by the mixed semantics.

4.2 Properties

What will the semantics of the mixed language do? It is important to answer the question because the evaluation sequences produced by lazy desugaring should be meaningful enough to have a practical use. In this section, we state and prove two important properties about the mixed semantics: *emulation* and *laziness*.

Emulation. The first property, *emulation*, adapted from the original resugaring work by [14], is described as follows (a diagram illustrating the property graphically is shown in Fig. 9). It says that, a one-step reduction of any t in the mixed language either (1) corresponds to a reduction of the desugared program in the core language, or (2) corresponds to a single-step expansion of the sugar.

Property 4.1 (Emulation)**.** For any term $t = \texttt{(}H \ t_1 \ \cdots \ t_n\texttt{)}$ where H is a core construct `head` or a sugar name `Sg`, either $\mathcal{R}_m(t)$ is not defined, or one of

[2] The sugars with cyclic dependence on evaluation contexts are ill-formed for general desugaring.

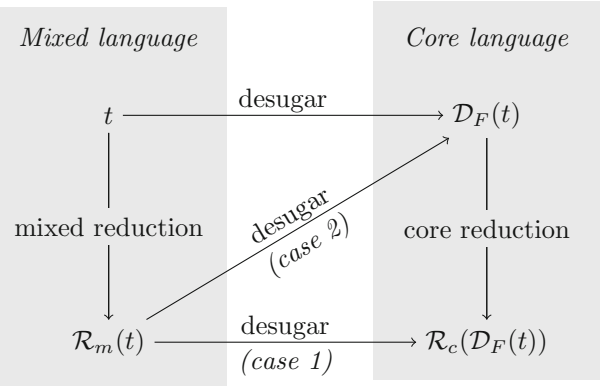

Fig. 9. Illustration of emulation.

the following statements holds: (Case 1) $\mathcal{D}_F(\mathcal{R}_m(t)) = \mathcal{R}_c(\mathcal{D}_F(t))$; and (Case 2) $\mathcal{D}_F(\mathcal{R}_m(t)) = \mathcal{D}_F(t)$.

Proof. By structural induction on the term t.

- Base case. If t is a normal form, $\mathcal{R}_m(t) = \bot$, which clearly satisfies the property.
- Induction hypothesis. Every sub-term of t, namely t_i, follows the emulation property.
- Induction step. If t is not a normal form, we conduct the proof with a case analysis.
  1. $H = \texttt{Sg}$, and $\mathcal{R}_m(t)$ expands the outermost `Sg` in t. Thus, we have
  $$\mathcal{D}_F(\mathcal{R}_m(t)) = \mathcal{D}_F(t).$$
  It turns out that the equation of *case 2* holds.
  2. $H = \texttt{Sg}$, and $\mathcal{R}_m(t)$ reduces t_i, i.e.,
  $$\mathcal{R}_m(t) = (\texttt{Sg}\ t_1\ \cdots\ \mathcal{R}_m(t_i)\ \cdots\ t_n).$$
  In this case,
  $$\mathcal{D}_F(\mathcal{R}_m(t)) = \mathcal{D}((\texttt{Sg}\ \mathcal{D}_F(t_1)\ \cdots\ \mathcal{D}_F(\mathcal{R}_m(t_i))\ \cdots\ \mathcal{D}_F(t_n))).$$
  On the other hand,
  $$\mathcal{D}_F(t) = \mathcal{D}((\texttt{Sg}\ \mathcal{D}_F(t_1)\ \cdots\ \mathcal{D}_F(t_i)\ \cdots\ \mathcal{D}_F(t_n))).$$
  If $\mathcal{D}_F(\mathcal{R}_m(t_i)) = \mathcal{D}_F(t_i)$, *case 2* holds. Otherwise, if $\mathcal{D}_F(\mathcal{R}_m(t_i)) = \mathcal{R}_c(\mathcal{D}_F(t_i))$, with Lemma 4.1 (which we state and prove later), we have
  $$\mathcal{R}_c(\mathcal{D}_F(t)) = \mathcal{D}((\texttt{Sg}\ \mathcal{D}_F(t_1)\ \cdots\ \mathcal{R}_c(\mathcal{D}_F(t_i))\ \cdots\ \mathcal{D}_F(t_n))).$$
  Thus, we conclude that either $\mathcal{D}_F(\mathcal{R}_m(t)) = \mathcal{D}_F(t)$ or $\mathcal{D}_F(\mathcal{R}_m(t)) = \mathcal{R}_c(\mathcal{D}_F(t))$ holds.

3. $H = \texttt{head}$, and $\mathcal{R}_m(t)$ is an application of the notion of reduction to t itself (the outermost `head`). In this case, we are going to prove that the equation of *case 1* holds. That is,

$$\mathcal{D}_F(\mathcal{R}(\texttt{(head } t_1 \ \cdots \ t_n\texttt{)})) = \mathcal{R}(\texttt{(head } \mathcal{D}_F(t_1) \ \cdots \ \mathcal{D}_F(t_n)\texttt{)}).$$

   This is exactly the definition of the extensible requirement of the notion of reduction in the core language.
4. $H = \texttt{head}$, and $\mathcal{R}_m(t)$ reduces t_i. First, $\mathcal{R}_m$ and $\mathcal{R}_c$ will reduce a term with the same index in

$$\texttt{(head } t_1 \ \cdots \ t_n\texttt{)}$$

   and

$$\texttt{(head } \mathcal{D}_F(t_1) \ \cdots \ \mathcal{D}_F(t_n)\texttt{)},$$

   respectively. The left-hand-side

$$\mathcal{D}_F(\mathcal{R}_c(t)) = \texttt{(head } \mathcal{D}_F(t_1) \ \cdots \ \mathcal{D}_F(\mathcal{R}_m(t_i)) \ \cdots \ \mathcal{D}_F(t_n)\texttt{)}.$$

   As for the RHS of *case 1* and *2*, we have

$$\mathcal{D}_F(t) = \texttt{(head } \mathcal{D}_F(t_1) \ \cdots \ \mathcal{D}_F(t_i) \ \cdots \ \mathcal{D}_F(t_n)\texttt{)}$$

   and

$$\mathcal{R}_c(\mathcal{D}_F(t)) = \texttt{(head } \mathcal{D}_F(t_1) \ \cdots \ \mathcal{R}_c(\mathcal{D}_F(t_i)) \ \cdots \ \mathcal{D}_F(t_n)\texttt{)}.$$

   By induction hypothesis, we can conclude that either $\mathcal{D}_F(\mathcal{R}_m(t)) = \mathcal{D}_F(t)$ or $\mathcal{D}_F(\mathcal{R}_m(t)) = \mathcal{R}_c(\mathcal{D}_F(t))$ holds.

□

In the above proof, we use Lemma 4.1 (for the second case at the induction step), which is stated and proved as follows.

Lemma 4.1. *For a term*

$$t = \texttt{(}\mathit{Sg}\ t_1 \ \cdots \ t_n\texttt{)},$$

if

$$\mathcal{R}_m(t) = \texttt{(}\mathit{Sg}\ t_1 \ \cdots \ \mathcal{R}_m(t_i) \ \cdots \ t_n\texttt{)}, \tag{2}$$

and $\mathcal{D}_F(\mathcal{R}_m(t_i)) \neq \mathcal{D}_F(t_i)$, *then*

$$\mathcal{R}_c(\mathcal{D}_F(t)) = \mathcal{D}(\texttt{(}\mathit{Sg}\ \mathcal{D}_F(t_1) \ \cdots \ \mathcal{R}_c(\mathcal{D}_F(t_i)) \ \cdots \ \mathcal{D}_F(t_n)\texttt{)}).$$

Proof. Equation (2) suggests that the sugar `Sg` is given a context rule like

$$\texttt{(Sg } \underline{t/v} \ \cdots \ C_i \ \cdots \ \underline{t/v}\texttt{)}$$

in the mixed semantics by the `getRules` algorithm. According to the algorithm, $\mathcal{D}(t)$ will also reduce at t_i's location for core context rule. Because $\mathcal{D}_F(\mathcal{R}_m(t_i)) \neq \mathcal{D}_F(t_i)$, $\mathcal{R}_m(t_i)$ reduces t_i by core language's reduction rule (as opposed to desugaring), so $\mathcal{D}_F(t_i)$ can be reduced by core language's reduction. Thus, the lemma holds. □

Laziness. Another property is *laziness*, which guarantees that desugaring acts as "lazy" as possible. In other words, the algorithm exposes as many terms in the surface level as possible. This property is crucial to the usefulness of lazy desugaring.

With the desugaring rule of a sugar `Sg`, we can define a $\mathbf{term}^n \to \mathbf{term}$ function $\mathcal{DSG}$ (the terms can be natually extended to metavariables) such that

$$\mathcal{DSG}(t_1, \cdots, t_n) = \mathcal{D}(\texttt{(Sg } t_1 \cdots t_n\texttt{)}).$$

Then it is obvious that

$$\mathcal{D}_F(\texttt{(Sg } t_1 \cdots t_n\texttt{)}) = \mathcal{DSG}(\mathcal{D}_F(t_1), \cdots, \mathcal{D}_F(t_n)). \tag{3}$$

Property 4.2 (Laziness)**.** For any term $t = \texttt{(Sg } t_1 \cdots t_n\texttt{)}$, if

$$\mathcal{R}_c(\mathcal{D}_F(t)) = \mathcal{DSG}(\mathcal{D}_F(t_1), \cdots, \mathcal{R}_c(\mathcal{D}_F(t_i)), \cdots, \mathcal{D}_F(t_n)) \tag{4}$$

then there exists j, such that

$$\mathcal{R}_m(t) = \texttt{(Sg } t_1 \cdots \mathcal{R}_m(t_j) \cdots t_n\texttt{)}.$$

That is to say, the sugar `Sg` will not be expanded in the mixed language, if the reduction occurs at one of the expanded $\mathcal{D}_F(t_i)$ in $\mathcal{D}_F(t)$.

Proof. Equations 3 and 4 imply

$$\mathcal{DSG}(\boldsymbol{t}_1/\boldsymbol{v}_1, \cdots, \boldsymbol{t}_n/\boldsymbol{v}_n) = C_c[\boldsymbol{t}_i]$$

where the j-th sub-metavariable is $\boldsymbol{v}_k$ when $\mathcal{D}_F(t_k)$ is a value, or $\boldsymbol{t}_k$ otherwise.

Then according to the first branch of function `getRules`, the context rule $[\boldsymbol{t}_i := C]\, D_{\text{LHS}}$ will be obtained, where $D_{\text{LHS}} = \texttt{(Sg } \boldsymbol{t}_1/\boldsymbol{v}_1 \, cdots \, \boldsymbol{t}_n/\boldsymbol{v}_n\texttt{)}$. Then for any k, such that $\mathcal{D}_F(t_k)$ is a value,

- if all t_k are also values, then based on the context rule above, equation $j = i$ holds;
- otherwise, simply assume that t_h is the only sub-term which is not a value, when $\mathcal{D}_F(t_k)$ is. Based on the function `getRules`, one of the context rule before computing $[\boldsymbol{t}_i := C]\, D_{\text{LHS}}$ will make $\boldsymbol{t}_h$ be a context hole, equation $j = h$ holds. (If t_h is not the only one, there must be one of t_h corresponding to the former context hole.) □

5 Case Studies

We have implemented our lazy desugaring system in PLT Redex [5], a semantic engineering tool based on reduction semantics [7]. It provides a useful environment for combining the core language's semantics with rules from our algorithm.

We have successfully tested a bunch of syntactic sugars with our system. In this paper, for the lack of space, we only describe the core algorithm of our method in Sect. 4, but other features like hygienic, (mutual) recursive, pattern based can be handled by simple extensions of our basic algorithm.

5.1 Simple Examples

We have seen several simple sugars in our running example, and we will give other examples to demonstrate some interesting observations.

It is not hard to convert from SKI combinator to call-by-need lambda calculus. Consider the S combinator as an example, which can be defined as a sugar below.

$$\texttt{S} \stackrel{def}{=\!=} (\lambda_N\ (\texttt{x}_1\ \texttt{x}_2\ \texttt{x}_3)\ (\texttt{x}_1\ \texttt{x}_3\ (\texttt{x}_2\ \texttt{x}_3)))$$

The interesting point is that we can use the call-by-need lambda calculus to force an expansion of a sugar in case we need it. For example, we may consider defining `S` in another form:

$$(\texttt{S}\ \boldsymbol{t}_1\ \boldsymbol{t}_2\ \boldsymbol{t}_3) \stackrel{def}{=\!=} (\texttt{let}\ (\texttt{x}\ \boldsymbol{t}_3)\ (\boldsymbol{t}_1\ \texttt{x}\ (\boldsymbol{t}_2\ \texttt{x}))).$$

In this case, the expansion of `S` will not happen until enough sub-terms have been normal-formed, which is different from the original combinator.

Similarly, recall the *And* sugar defined before. We may redefine it with call-by-need lambda calculus as follows.

$$\texttt{ForceAnd} \stackrel{def}{=\!=} (\lambda_N\ (\texttt{x}_1\ \texttt{x}_2)\ (\texttt{if}\ \texttt{x}_1\ \texttt{x}_2\ \texttt{\#f}))$$

Given any program with $(\texttt{ForceAnd}\ t_1\ t_2)$ as its sub-term, when $(\texttt{ForceAnd}\ t_1\ t_2)$ should be reduced, the evaluation sequence will look like this.

$$\begin{aligned} & (\ldots\ (\texttt{ForceAnd}\ t_1\ t_2)\ \ldots) \\ \longrightarrow\ & (\ldots\ ((\lambda_N\ (\texttt{x}_1\ \texttt{x}_2)\ (\texttt{if}\ \texttt{x}_1\ \texttt{x}_2\ \texttt{\#f}))\ t_1\ t_2)\ \ldots) \\ \longrightarrow\ & (\ldots\ (\texttt{if}\ t_1\ t_2\ \texttt{\#f})\ \ldots) \\ \longrightarrow\ & \ldots \end{aligned}$$

5.2 More Examples

Since the essential idea of our approach is not complex, it is possible to extend the basic algorithm to handle many kinds of complex sugar features. In this section, we give two examples of hygienic sugar and higher-order sugar.

Given a typical hygienic sugar

$$(\texttt{HygienicAdd}\ \boldsymbol{t}_1\ \boldsymbol{t}_2) \stackrel{def}{=\!=} (\texttt{let}\ (\texttt{x}\ \boldsymbol{t}_1)\ (\texttt{+}\ \texttt{x}\ \boldsymbol{t}_2)),$$

for the program

$$(\texttt{let}\ (\texttt{x}\ 2)\ (\texttt{HygienicAdd}\ 1\ \texttt{x})),$$

the existing resugaring approach [15] uses an abstract syntax DAG to distinguish different variables `x` in the desugared term

$$(\texttt{let}\ (\texttt{x}\ 2)\ (\texttt{let}\ (\texttt{x}\ 1)\ (\texttt{+}\ \texttt{x}\ \texttt{x}))).$$

But in our lazy desugaring setting, the `HygienicAdd` sugar is not expanded until necessary. The sequence will be as follows.

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

Higher-order functions from the functional language are introduced to many other programming languages as important features. We attempt to process the higher-order sugar with our method, for example, with the sugar[3]

```
(Filter t (list v1 v2 ...)) =def= (let (f t)
                                    (if (f v1)
                                        (cons v1 (Filter f (list v2 ...)))
                                        (Filter f (list v2 ...))))
        (Filter t (list)) =def= (list)
```

and we obtain the following sequence with the example.

```
    (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)
```

6 Related Work

As we discussed before, our work is closely related to the pioneering work of *resugaring* [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 reverse desugaring, which is both more powerful and efficient. For hygienic sugar, compared with the approach of using DAG to solve the variable binding problem [15], our approach of "lazy desugaring" can achieve natural hygiene with a hygienic expansion.

Macros as multi-stage computations [10] is a work related to our lazy expansion of sugars. Some other work [16] on multi-stage programming [18] indicates that it is useful for implementing domain-specific languages. However, multi-stage programming is a meta-programming method, which mainly aims for run-time code generation and optimization. In contrast, our lazy desugaring method treats sugars as part of a mixed language, rather than separating them by staging. Moreover, lazy desugaring gives us a chance to derive evaluation rules of sugars, which is an advantage over multi-stage programming.

[3] The expression '`t ...`' means zero or more `t` as a pattern.

The lazy desugaring used to be explored [3]. They model the expansion with explicit substitutions [1] and delay the expansion by subtle rules. They also declare the benefit to avoid unnecessary expansions. While their main contribution is a formal semantics of macro expansion, the macros in a program do not preserve their original formats. In contrast, our lazy desugaring can preserve the sugars as long as they do not have to be expanded.

There is a long history of hygienic macro expansion [12], and a formal specific hygiene definition was given by specifying the binding scopes of macros [11]. Another formal definition of the hygienic macro [2] is based on nominal logic [9]. Instead of designing something special for the hygienic sugar as by [15], our method can be easily combined with the existing hygienic method, because the reverse desugaring is not needed.

Our implementation is built upon PLT Redex [5], a semantics engineering tool, but it is possible to implement our approach with other semantics engineering tools [17,19] which aim to test or verify the semantics of languages. Their methods can be easily combined with our approach to implementing more general rule derivation. *Ziggurat* [8] is a semantics extension framework, also allowing defining new macros with semantics based on existing terms of a language. It should be useful for static analysis of the mixed language in our approach.

7 Conclusion

In this paper, we propose a novel lazy desugaring method which smartly evaluates the programs with sugar. Our algorithm automatically generates the mixed language semantics from the core language and the sugar definition, and achieves "resugaring" by outputting the evaluation sequence of a program with sugars based on the mixed language semantics. In our method, the most important point is delaying the expansion of syntactic sugars by deriving suitable context rules, which decide whether the mixed language should reduce the sub-term by reduction rules of the core language or expand a sugar term. Our approach is flexible for more extensions.

There are some interesting future works. One is to extend the framework from evaluation to other language components such as type system, analyzer, and optimizer. Also, we find it possible to derive stand-alone evaluation rules for the surface language by means similar to how we calculate context rules. This would make it more convenient to develop domain-specific languages. The usefulness of lazy desugaring's expressiveness is also worth exploring, since some ill-formed sugar definitions[4] for the general desugaring can be handled by lazy desugaring.

[4] For example, `(Odd t)` $\stackrel{def}{=}$ `(let (x t) (if (> x 0) (not (Odd (- x 1))) #f))` is ill-formed because of expansion without termination, but can be avoided by lazy desugaring.

References

1. Abadi, M., Cardelli, L., Curien, P.L., Levy, J.J.: Explicit substitutions. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1990, pp. 31–46. Association for Computing Machinery, New York (1989). https://doi.org/10.1145/96709.96712
2. Adams, M.D.: Towards the essence of hygiene. SIGPLAN Not. **50**(1), 457–469 (2015). https://doi.org/10.1145/2775051.2677013
3. Bove, A., Arbilla, L.: A confluent calculus of macro expansion and evaluation. In: Proceedings of the 1992 ACM Conference on LISP and Functional Programming, LFP 1992, pp. 278–287. Association for Computing Machinery, New York (1992). https://doi.org/10.1145/141471.141562
4. Culpepper, R., Felleisen, M., Flatt, M., Krishnamurthi, S.: From Macros to DSLs: the evolution of racket. In: Lerner, B.S., Bodík, R., Krishnamurthi, S. (eds.) 3rd Summit on Advances in Programming Languages, SNAPL 2019, Providence, RI, USA, 16–17 May 2019. LIPIcs, vol. 136, pp. 5:1–5:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
5. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex, 1st edn. The MIT Press (2009)
6. Felleisen, M., et al.: A programmable programming language. Commun. ACM **61**(3), 62–71 (2018)
7. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. **103**(2), 235–271 (1992). https://doi.org/10.1016/0304-3975(92)90014-7
8. Fisher, D., Shivers, O.: Static analysis for syntax objects. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, pp. 111–121. Association for Computing Machinery, New York (2006). https://doi.org/10.1145/1159803.1159817
9. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Form. Asp. Comput. **13**(3–5), 341–363 (2002). https://doi.org/10.1007/s001650200016
10. Ganz, S.E., Sabry, A., Taha, W.: Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming, ICFP 2001, pp. 74–85. Association for Computing Machinery, New York (2001). https://doi.org/10.1145/507635.507646
11. Herman, D., Wand, M.: A theory of hygienic macros. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 48–62. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78739-6_4
12. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, pp. 151–161. Association for Computing Machinery, New York (1986). https://doi.org/10.1145/319838.319859
13. Landin, P.J.: The mechanical evaluation of expressions. Comput. J. **6**(4), 308–320 (1964). https://doi.org/10.1093/comjnl/6.4.308
14. Pombrio, J., Krishnamurthi, S.: Resugaring: lifting evaluation sequences through syntactic sugar. SIGPLAN Not. **49**(6), 361–371 (2014). https://doi.org/10.1145/2666356.2594319
15. Pombrio, J., Krishnamurthi, S.: Hygienic resugaring of compositional desugaring. IGPLAN Not. **50**(9), 75–87 (2015). https://doi.org/10.1145/2858949.2784755

16. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. SIGPLAN Not. **46**(2), 127–136 (2010). https://doi.org/10.1145/1942788.1868314
17. Roşu, G., Şerbănută, T.F.: An overview of the K semantic framework. J. Logic Algebraic Program. **79**(6), 397–434 (2010)
18. Taha, W.: A Gentle Introduction to Multi-stage Programming, pp. 30–50, January 2003
19. Vergu, V., Neron, P., Visser, E.: DynSem: a DSL for dynamic semantics specification. In: Fernández, M. (ed.) 26th International Conference on Rewriting Techniques and Applications (RTA 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 36, pp. 365–378. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2015). https://doi.org/10.4230/LIPIcs.RTA.2015.365. http://drops.dagstuhl.de/opus/volltexte/2015/5208

Author Queries

Chapter 14

Query Refs.	Details Required	Author's response
AQ1	Per Springer style, both city and country names must be present in the affiliations. Accordingly, we have inserted the city and country names in affiliations "2". Please check and confirm if the inserted city and country names are correct. If not, please provide us with the correct city and country names.	