Semantic Lifting for Domain-Specific Languages

ZHICHAO GUAN and ZHENJIANG HU, School of Computer Science, Peking University, China

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

CCS Concepts: • **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Zhichao Guan and Zhenjiang Hu. 2018. Semantic Lifting for Domain-Specific Languages. In . ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Language-oriented programming (LOP)[5] is solving a class of problems by designing one or more new domain-specific languages (DSLs). To avoid building common languages constructs like loops and branches, developers may implement DSLs on top of another general-purpose programming language (called host language). Developers can extend the host language based on the language features, rendering programs with domain-specific forms. Languages with features like macros and higher-order functions are frequently employed. By specifying translation rules from the DSL to the host language, developers can obtain a DSL interpreter naturally. This type of DSL is called embedded DSL (eDSL).

Despite eDSL can significantly reduce implementation costs, DSL users find it inconvenient due to the inevitable need to learn the host language and understand error messages. This is called abstraction leakage, where the users writes a program in DSL and it is translated into the host language, causing the program executed formally differently from the original program. Much prior work has focused on how to translate information from the host language to the DSL users manually. Todo: More related work about generate information manually

Pombrio et al. [3] have made a great progress in maintenance of abstraction automatically. They proposed *resugaring*, by selectively reorganizing the sequence of evaluations on the host language to the DSL according to the reverse translation rules. But there are several practical problems in this approach. First, programs with a lot of syntactic sugars are pretty expensive to check whether each term in the evaluation sequences can be resugared. Second, it still contains a host language evaluator executing, so that users cannot intuit the functionality of language constructs. Yang et al. [6] have solved the first problem via lazy desugaring.

In general, embedded DSLs treat the host language as a black box. We take the semantics of the host language as a white box, written by the meta language, and derive the semantics for language constructs of DSL automatically.

Manuscript submitted to ACM

$$\text{(Step 1)}\ \dfrac{\text{(Step 2)}\ \dfrac{e_1 \Downarrow \mathit{true} \qquad e_2 \Downarrow v}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ \mathit{false} \Downarrow v}}{e_1\ \mathit{and}\ e_2 \Downarrow v} \qquad \text{(Step 1)}\ \dfrac{\text{(Step 2)}\ \dfrac{e_1 \Downarrow \mathit{false} \qquad \text{(Step 3)}\ \dfrac{}{\mathit{false} \Downarrow \mathit{false}}}{\mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ \mathit{false} \Downarrow \mathit{false}}}{e_1\ \mathit{and}\ e_2 \Downarrow \mathit{false}}$$

Fig. 1. Derivation of **and**

We follow the core ideas of inferring type rules for syntactic sugars [4] and extend it to the semantics. For example, in a host language with lambda abstraction and application, we can define *let* by translation rule:

$$e_1\ \mathit{and}\ e_2 \Rightarrow \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ \mathit{false}.$$

The derivation of *and* is given in Fig. 1. In DSL, an expression matching the pattern of left-hand side (LHS) evaluates to v, if and only if the right-hand side (RHS) evaluates to v (Step 1). Based on the rules of if-then-else, we can futher expand the evaluation of RHS. Since if-then-else is described in two rules according to the evaluation of e_1, our derivation tree have to do the same thing (Step 2). Similarly, the rule of *false* can be applied (Step 3). Hence, we can obtain the following conclutions:

$$\dfrac{e_1 \Downarrow \mathit{true} \qquad e_2 \Downarrow v}{e_1\ \mathit{and}\ e_2 \Downarrow v} \qquad \dfrac{e_1 \Downarrow \mathit{false}}{e_1\ \mathit{and}\ e_2 \Downarrow \mathit{false}}$$

It should be observed that without mentioning *if*, the rules directly describe the rules of *and*. That means, the abstraction we want to maintain — that the rules of *and* are independent of the host language — is true.

It is seemingly natural but we are facing three challenges: (1) Unlike type rules, which can typically be stated by one single rule, many evaluation rules, such as *if*, do not conform this property. This may lead to nondeterminacy when searching rules or exponential growth in the number of rules. (2) Most type rules are defined in a modular way, which means the type of an expression depends on the types of subexpressions, but the semantics may be not. In particular, when using lambda-calculus as the host language, lambda abstraction itself is a value. Translation rules defined by lambda abstraction, is a value itself. The evaluation rules of them may ruin the abstraction, like

$$\mathit{and}_f \Rightarrow \lambda x : \mathit{bool}, y : \mathit{bool}.\ \mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ \mathit{false}.$$

(3) Since the rules of application include substitution, we must specify the behavior of the expression containing substitution in semantic derivation.

To address challenge (1), we introduce a variant of Skeleton [1] as meta-language to describe semantics. Make sure that each language structure is evaluated according to unique rules, in order to guarantee the determinacy of semantic derivation. To address challenge (2), we propose lambda lifting, to reveal the semantics of lambda abstraction. To address challenge (3), we show that substitution can maintain the correctness and abstraction of semantic derivation, but we shall impose greater limitations on translation rules.

In this paper, we propose a new framework for DSL design. We use simply-typed lambda-calculus (STLC) as the host language for its elementariness and versatility. To increase the generality of the framework, we provide meta-extensions (to introduce new vocabularies) and monad-extensions (to introduce side-effects) on the host language. Then, users can specify DSL constructs by translation rules on the new extended host language. Hence, as the focus of this paper, the framework will derive the evaluation rules and type rules for these constructs automatically, to make the DSL standalone. All the semantics are described by the meta-language, and those of DSLs are generated. Finally

$$\begin{aligned} e \in \mathit{Exp} &::= \mathit{true} \mid \mathit{false} \mid \mathit{if}\ e_1\ e_2\ e_3 \\ &\mid x \mid \lambda x : t.e \mid e_1\ e_2 \\ t \in \mathit{Type} &::= \mathit{bool} \mid t_1 \longrightarrow t_2 \end{aligned}$$

Fig. 2. Syntax of STLC

and naturally, our framework will generate interpreters based on semantics. Our main technical contributions can be summarized as follows:

- Todo: host
- Todo: dsl
- We present an algorithm to derive semantics for DSL constructs defined by translation rules. For the translation rules defined with lambda abstraction, we will give the lambda-lifting method. We will prove the correctness and abstraction of the algorithm.
- We give an implementation of the framework.

2 OVERVIEW

In this section, we shall demonstrate how host language developers define a language with the meta-language, how DSL developers define a DSL by language extensions and translation rules, and how DSL become standalone by semantic derivation in our framework.

2.1 Define Host Language via Meta-Language

Skeletal semantics can be used to describe concrete and abstract semantics in a structural way. A skeleton body is defined by a sequence of operations. An operation is either recursive computations (hooks), meta functions (filters), different pathways (branches). Our meta-language is a variant of Skeleton. For example, consider the *if* construct with generic subexpressions denoted by e_1, e_2 and e_3, whose operational semantics are defined using two seperate rules.

$$\frac{e_1 \Downarrow \mathit{true} \qquad e_2 \Downarrow v_2}{\mathit{if}\ e_1\ e_2\ e_3 \Downarrow v_2} \qquad \frac{e_1 \Downarrow \mathit{false} \qquad e_3 \Downarrow v_3}{\mathit{if}\ e_1\ e_2\ e_3 \Downarrow v_3}$$

In our meta-language, the behaviour of *if* is given by

$$\mathcal{E}(\mathit{if}\ e_1\ e_2\ e_3) \coloneqq \mathcal{E}(e_1) : \begin{pmatrix} \mathit{true} \triangleright \mathcal{E}(e_2) \\ \mathit{false} \triangleright \mathcal{E}(e_3) \end{pmatrix}.$$

Here, $\mathcal{E}(e_1)$ identifies the evaluation of subexpression e_1, whose result is used to select a specific branch by pattern matching (after the colon). Then, subsequent computations are processed (after the triangle). We can also define the type rules of *if* in a similar way:

$$\mathcal{T}(\mathit{if}\ e_1\ e_2\ e_3) \coloneqq \textbf{let}\ \mathit{bool} = \mathcal{T}(e_1);\ \textbf{let}\ t_2 = \mathcal{T}(e_2);\ \mathcal{T}(e_3) : (t_3 \mid t_2 = t_3 \triangleright t_2)$$

In order to make the structure clear, we use *let* as a syntactic sugar to represent a pattern matching with unique branch. The type of e_1 is required to be *bool*. The matching of e_3 has a side condition (named *guard* in Haskell), which requires t_2 to equal t_3.

$$
\begin{aligned}
\mathcal{E}(true) &\coloneqq true \\
\mathcal{E}(false) &\coloneqq false \\
\mathcal{E}(if\ e_1\ e_2\ e_3) &\coloneqq \mathcal{E}(e_1) : \begin{pmatrix} true \triangleright \mathcal{E}(e_2) \\ false \triangleright \mathcal{E}(e_3) \end{pmatrix} \\
\mathcal{E}(\lambda x : t.e) &\coloneqq \lambda x : t.e \\
\mathcal{E}(e_1\ e_2) &\coloneqq \textbf{let}\ \lambda x : t.e = \mathcal{E}(e_1);\ \textbf{let}\ v_2 = \mathcal{E}(e_2);\ \mathcal{E}(e[v_2/x])
\end{aligned}
$$

Fig. 3. Semantics of STLC

$$
\begin{aligned}
\mathcal{T}(true) &\coloneqq bool \\
\mathcal{T}(false) &\coloneqq bool \\
\mathcal{T}(if\ e_1\ e_2\ e_3) &\coloneqq \textbf{let}\ bool = \mathcal{T}(e_1);\ \textbf{let}\ t_2 = \mathcal{T}(e_2);\ \mathcal{T}(e_3) : (t_3 \mid t_2 = t_3 \triangleright t_2) \\
\mathcal{T}(x) &\coloneqq searchCtx\ x \\
\mathcal{T}(\lambda x : t.e) &\coloneqq \textbf{let}\ \_ = updateCtx\ x\ t;\ \textbf{let}\ t' = \mathcal{T}(e); \\
&\quad\ \textbf{let}\ \_ = restoreCtx;\ t \to t' \\
\mathcal{T}(e_1\ e_2) &\coloneqq \textbf{let}\ \lambda x : t.e = \mathcal{E}(e_1);\ \textbf{let}\ v_2 = \mathcal{E}(e_2);\ \mathcal{E}(e[v_2/x])
\end{aligned}
$$

Fig. 4. Type Rules of STLC

The syntax of STLC is given in Fig. 2. And the semantics of STLC is given in Fig. 3. We can observe that (1) the evaluation of values gets themselves, which is consistent with the big-step operational semantics, (2) and we use substitution to evaluate application. In our meta-language, substitution is considered as a special program transformation. Hence, the substitution rules of language constructs should be specified, omitted from this paper.

We can think of evaluation $\mathcal{E}$ as a function of $Exp \to Exp$. Then the type $\mathcal{T}$ should be a function of $Exp \to Type$. But a context is necessary when typing lambda abstraction. Instead of changing $\mathcal{T}$ to $Exp \to Ctx \to Type$, we introduce state monad to describe context modification for modularity. So that $\mathcal{T}$ is a function of $Exp \to State\ CtxStack\ Type$. Todo: How to explain this monad more clearly?

$$\mathcal{T}(\lambda x : t.e) \coloneqq \textbf{let}\ \_ = updateCtx\ x\ t;\ \textbf{let}\ t' = \mathcal{T}(e);\ \textbf{let}\ \_ = restoreCtx;\ t \to t'$$

With the Introduction of monad, the type rule of *if* remains unchanged. We can also keep the original semantic description intact when subsequent DSL users extend the host language. The type rules of STLC is given in Fig. 4.

2.2 Define the DSL

Todo: Use textsc for language names Consider a task of implementing a DSL for boolean computations, half adder and full adder, named Bool. Bool has some expressions like

$$true\ and\ (false\ xor\ true) \equiv true$$

$$half\text{-}adder\ true\ false \equiv (false, true).$$

$$
\begin{aligned}
e \in \mathit{Exp} ::= & \cdots \mid \mathit{not}\ e \mid e\ \mathit{and}\ e \mid e\ \mathit{or}\ e \\
& \mid e\ \mathit{nand}\ e \mid e\ \mathit{nor}\ e \mid e\ \mathit{xor}\ e \\
& \mid \mathit{half\text{-}adder}\ e\ e \mid \mathit{full\text{-}adder}\ e\ e\ e \\
& \mid (e, e) \mid \mathit{fst}\ e \mid \mathit{snd}\ e \\
t \in \mathit{Type} ::= & \cdots \mid t \times t
\end{aligned}
$$

Fig. 5. Syntax of Bool

$$
\begin{aligned}
\mathcal{E}((e_1, e_2)) &\coloneqq \textbf{let } v_1 = \mathcal{E}(e_1);\ \textbf{let } v_2 = \mathcal{E}(e_2);\ (v_1, v_2) & \mathcal{T}((e_1, e_2)) &\coloneqq \textbf{let } t_1 = \mathcal{T}(e_1);\ \textbf{let } t_2 = \mathcal{T}(e_2);\ t_1 \times t_2 \\
\mathcal{E}(\mathit{fst}\ e) &\coloneqq \textbf{let } (v_1, \_) = \mathcal{E}(e);\ v_1 & \mathcal{T}(\mathit{fst}\ e) &\coloneqq \textbf{let } (t_1, \_) = \mathcal{T}(e);\ t_1 \\
\mathcal{E}(\mathit{snd}\ e) &\coloneqq \textbf{let } (\_, v_2) = \mathcal{E}(e);\ v_2 & \mathcal{T}(\mathit{snd}\ e) &\coloneqq \textbf{let } (\_, t_2) = \mathcal{T}(e);\ t_2
\end{aligned}
$$

Fig. 6. Meta-Extension Rules for Bool

$$
\begin{aligned}
\mathit{not}\ e &\Rightarrow \mathit{if}\ e\ \mathit{false}\ \mathit{true} & e_1\ \mathit{or}\ e_2 &\Rightarrow \mathit{if}\ e_1\ \mathit{true}\ e_2 \\
e_1\ \mathit{nand}\ e_2 &\Rightarrow \mathit{not}\ (e_1\ \mathit{and}\ e_2) & e_1\ \mathit{nor}\ e_2 &\Rightarrow \mathit{not}\ (e_1\ \mathit{or}\ e_2)
\end{aligned}
$$

$$
\begin{aligned}
e_1\ \mathit{xor}\ e_2 &\Rightarrow (e_1\ \mathit{and}\ \mathit{not}\ e_2)\ \mathit{or}\ (\mathit{not}\ e_1\ \mathit{and}\ e_2) \\
\mathit{half\text{-}adder}\ e_1\ e_2 &\Rightarrow (e_1\ \mathit{xor}\ e_2, e_1\ \mathit{and}\ e_2) \\
\mathit{full\text{-}adder}\ e_1\ e_2\ e_3 &\Rightarrow (e_1\ \mathit{xor}\ e_2\ \mathit{xor}\ e_3, (e_1\ \mathit{xor}\ e_2)\ \mathit{or}\ (e_3\ \mathit{and}\ (e_1\ \mathit{xor}\ e_2)))
\end{aligned}
$$

Fig. 7. Translation Rules for Bool

Since our language extends on the host language, all the syntax of STLC is also preserved in the DSL. The syntax of Bool is shown in Fig. 5 Remember that we are more concerned with semantics than syntax, and they are all treated as language constructs for which we need to define there evaluation and type rules.

Extend the host language with meta-extension. Half adder and full adder will get sum and carry as the results, so pairs can be used to build compound data structures. Naturally, projection *fst* and *snd* are involved as *Exp*, and product $t \times t$ is involved as *Type*. Developers need to indicate the evaluation and type rules (and substitution rules) for each newly defined language constructs, just like defining the host language. What the developers need to provide is shown in Fig. 6.

Specify the DSL with translation rules. Unlike pairs, boolean operations like *and* can be directly define through *if* etc. Each translation rule has a LHS and a RHS, which are expressions containing pattern variables. Translation rules define the language constructs in the DSL, and we require that each such language construct is determined by a unique translation rule. Each pattern variable must be of a specific class, e.g. *Exp*, *Type*. We use e_i for *Exp*, t_i for *Type*, x for identifiers.

$$e_1\ \mathit{and}\ e_2 \Rightarrow \mathit{if}\ e_1\ e_2\ \mathit{false}$$

Translation rules for the rest constructs in Bool are shown in Fig. 7. In this example, we can notice that we can use not only the language constructs of the host language, but also those defined by the meta-extension, as well as the former translation rules. We will discuss the recursive translation rules later, and it will cause some trouble.

2.3 Language Lifting

Now it is time for our framework. For meta-extension, just add them to the set of rules. For translation rules, we need to derive the evaluation and type rules for them.

Derive the semantics. Consider that for any e_1 and e_2, the evaluation of e_1 *and* e_2 should be the same as *if* e_1 e_2 *false*. Hence, according to the translation rule and evaluation rules of the host language, we have

$$
\begin{aligned}
\mathcal{E}(e_1\ \mathit{and}\ e_2) &= \mathcal{E}(\mathit{if}\ e_1\ e_2\ \mathit{false}) && \text{(Translation rule)} \\
&= \mathcal{E}(e_1) : \begin{pmatrix} \mathit{true} \triangleright \mathcal{E}(e_2) \\ \mathit{false} \triangleright \mathcal{E}(\mathit{false}) \end{pmatrix} && \text{(Evaluation rule of } \mathit{if}\text{)} \\
&= \mathcal{E}(e_1) : \begin{pmatrix} \mathit{true} \triangleright \mathcal{E}(e_2) \\ \mathit{false} \triangleright \mathit{false} \end{pmatrix} && \text{(Evaluation rule of } \mathit{false}\text{)}
\end{aligned}
$$

The evaluation rule of *and* has been derived. The following points are worth noting: (1) Abstraction: the semantics of *and* is made explicit, and no longer needs to be expressed by *if* from the host language. That is, even if we remove *if* from the language, the *and* works fine. (2) Uniqueness: due to our requirement of uniqueness of the rules, the rule unfolding in the derivation is always deterministic. Also thanks to uniqueness of translation rules, the rules we derive for these new language constructs stay unique.

Lambda Lifting. As mentioned above, a language construct can be defined as a lambda abstraction by translation rules like and_f. The above method will result in

$$
\begin{aligned}
\mathcal{E}(\mathit{and}_f) &= \mathcal{E}(\lambda x : \mathit{bool}.\ \lambda y : \mathit{bool}.\ \mathit{if}\ x\ y\ \mathit{false}) \\
&= \lambda x : \mathit{bool}.\ \lambda y : \mathit{bool}.\ \mathit{if}\ x\ y\ \mathit{false},
\end{aligned}
$$

which breaks abstraction. By converting and_f to *and*, the internal computation can be exposed. The approach of exposing parameters of the lambda abstraction in RHS of a translation rule r and transforming r into a set of new translation rules, is called lambda lifting. In the above example, and_f will be transformed into

$$
\begin{aligned}
\mathit{and}'_f\ e_1\ e_2 &\Rightarrow \mathit{if}\ e_1\ e_2\ \mathit{false} \\
\mathit{and}_f &\Rightarrow \lambda x : \mathit{bool}.\ \lambda y : \mathit{bool}.\ \mathit{and}'_f\ x\ y.
\end{aligned}
$$

Thinking of andf' as a language construct on the DSL, the abstraction property of and_f is satisfied. Todo: strong abstraction, week abstraction Lambda lifting is also valid for those translation rules for nested lambda abstractions.

Derive the type rules. The method of deriving evaluation rules can also be applied to type rules. For example, the type rule of *and* can be derived by

$$\begin{aligned}
\mathcal{T}(e_1 \textit{ and } e_2) &= \mathcal{T}(\textit{if } e_1\ e_2\ \textit{false}) && \text{(Translation rule)}\\
&= \textbf{let } \textit{bool} = \mathcal{T}(e_1);\ \textbf{let } t_2 = \mathcal{T}(e_2);\ \mathcal{T}(\textit{false}) : (t_3 \mid t_2 = t_3 \triangleright t_2) && \text{(Evaluation rule of } \textit{if}\text{)}\\
&= \textbf{let } \textit{bool} = \mathcal{T}(e_1);\ \textbf{let } t_2 = \mathcal{T}(e_2);\ \textit{bool} : (t_3 \mid t_2 = t_3 \triangleright t_2) && \text{(Evaluation rule of } \textit{false}\text{)}\\
&= \textbf{let } \textit{bool} = \mathcal{T}(e_1);\ \textbf{let } \textit{bool} = \mathcal{T}(e_2);\ \textit{bool}. && \text{(Simplification)}
\end{aligned}$$

Summary

After these processes, the DSL developers implement Bool language. Todo: talk about standalone

3 META-LANGUAGE AND HOST LANGUAGES

In this section, we will discuss the syntax and semantic of our meta-language.

3.1 Meta-Language

As mentioned above, our meta-language is inspired by Skeleton. But it differs in the problems and is therefore adapted. Skeleton focuses on illustrating the consistency of concrete and abstract interpretations. The meaning of filters (meta-operations in Skeleton) varies in different goals. We concentrate on evaluation and typing. As an interpretation of Skeleton, it is intuitively close to Haskell. For example, instead of branches, we use pattern matching; instead of defining the merging of the interpretation of branches, we will pick the first branch that successfully matches. In addition, we support monad at the meta-language level. We will show the syntax of the meta-language, and how the meta-language is translated to Haskell to illustrate its semantics.

3.1.1 Expressions of Meta-Language. Various *expressions* in the meta-language are used to encode the language constructs in the host language or the DSL, such as expressions and types. (We use L to refer to a language defined by meta-language like STLC. In the remainder of this section, we will indicate such language with an L to distinguish from the meta-language. Also, expressions ε in meta-language need to be sperated from expressions in language L.)

An expression ε is built by *base expressions*, like literals and identifiers, ranged by ε_b; *expression variables*, ranged by ε_x; and *compound expressions* by *constructors*, ranged by c. Each language construct in L corresponds to a constructor c in meta-language. To distinguish between various sorts of language constructs, we define S for a finite set of *sorts*, ranged over by s. A sort s can be either an *base sort* like *Int*, *Id*; or a *program sort* like *Exp*, *Type* in STLC. The sort of an expression ε, written as $sort(\varepsilon) = s$, illustrate its role in language L. A constructor c has a signature $sig(c)$, which is of the form $(s_1 \cdots s_n) \longrightarrow s$, where $s_1 \cdots s_n$ are appropriate sorts of parameters of c, and s is the sort of expressions constructed by c, briefly stated as sort of c. A compound expression, written as $c(\varepsilon_1, \cdots, \varepsilon_n)$ or $c\ \varepsilon_1 \cdots \varepsilon_n$, is legal, if $sort(\varepsilon_i) = s_i$ for every $i \in [1, n]$.

Example 3.1. $\lambda x : \textit{bool}.\ e$ in language L, is constructor **Abs** applied to "x", *bool* and expression variable e. "x" are base expressions, and $sort(\text{"}x\text{"}) = \textit{Id}$; *bool* is constructor **Bool** applied to nothing. The signature of **Abs** is $sig(\textbf{Abs}) = (\textit{Id}, \textit{Type}, \textit{Exp}) \longrightarrow \textit{Exp}$, and its sort is *Exp*. The signature and sort of **Bool** are both *Type*.

Language L		Meta-Language
$\lambda x : \textit{bool}.\ e$	$\leftrightarrow$	$\textbf{Abs}(\text{"}x\text{"}, \textbf{Bool}(), e)$ or $\textbf{Abs}\ \text{"}x\text{"}\ \textbf{Bool}\ e$

$$
\begin{aligned}
d \in \mathit{Body} &::= b \\
&\mid b : (br_1 \cdots br_n) && \text{(Pattern Matching)} \\
&\mid \textbf{let}\ pat = b_1;\ b_2 && \text{(Let Binding)} \\
b \in \mathit{Bone} &::= exp' && \text{(General Expression)} \\
&\mid h(exp') && \text{(Recursive Computation)} \\
&\mid f_m(exp'_1 \cdots exp'_n) && \text{(Monadic Meta-function)} \\
exp' \in \mathit{Exp}_G &::= exp && \text{(Expression)} \\
&\mid f_p(exp'_1 \cdots exp'_n) && \text{(Pure Meta-function)} \\
br \in \mathit{Branch} &::= pat \triangleright b \\
&\mid exp' \triangleright b \\
pat \in \mathit{Pattern} &::= exp_x && \text{(Expression Variable)} \\
&\mid c\ pat_1 \cdots pat_n && \text{(Compound Pattern)}
\end{aligned}
$$

Fig. 8. Body of Rules in Meta-Language

We use i for expressions with sort *Int*, x for *Id*, e for *Exp*, and t for *Type*. A *closed expression* $\widetilde{\varepsilon}$ is an expression with no expression variables, generally refering to a concrete component in language L. Let Σ be an *environment* mapping from expression variables to expressions. We write $\Sigma(\varepsilon)$ to replace all expression variables $\varepsilon_x \in dom(\Sigma)$ with $\Sigma(\varepsilon_x)$ in ε. An environment Σ is *closed* for expression ε, when $\Sigma(\varepsilon)$ is a closed expression.

3.1.2 Rules of Meta-Language. A rule of meta-language has the shape $h(c\ \varepsilon_{x_1} \cdots \varepsilon_{x_n}) := d$, where h is the name of interpretation like $\mathcal{E}$ and $\mathcal{T}$, c is a constructor, $\varepsilon_{x_1} \cdots \varepsilon_{x_n}$ are expression variables, and d is the rule body. An interpretation h has a signature $sig(h) = s_1 \to m\ s_2$, where s_1 states for which h computes, and s_2 states for the result of h, m is the monad for side-effects. When no side effects are introduced, m can be omitted. A body is composed of bones, shown in Fig. 8. A bone, which represents an operation, can be either a general expression, a recursive computation, or a meta-function.

General expressions are expressions ε or computation of expressions with f_p. Two types of meta-functions are mentioned here: monadic meta-functions, ranged over by f_m, pure meta-functions, ranged over by f_p. When monad is introduced, f_m is function containing side effects, whose signature is like $sig(f_m) = (s_1 \cdots s_n) \to m\ s$; while f_p is not, whose signature is of the form $sig(f_p) = (s_1 \cdots s_n) \to s$. Most of these functions are pre-defined. For example, $searchCtx$, $updateCtx$ and $restoreCtx$ are monadic meta-functions, equality checking ($=$) and substitution are pure meta-functions. Interestingly, as the result, the use of monad in the body of rules is implicit, where *return* or *bind* need not be written explicitly.

3.1.3 Language L Defined in Meta-Language. A *Language L* consists of a set of program sorts S_p; a set of language constructs C with signatures; a set of interpretations H with rules; a set of meta-functions F, seprerated by monadic ones F_m and pure ones F_p. In particular, in STLCand its extensions, $S_p = \{Exp, Type\}$. H can be seperated into a set of evaluation rules $R_{\mathcal{E}}$ and a set of typing rules $R_{\mathcal{T}}$, where $sig(\mathcal{E}) = Exp \to m_1\ \mathcal{E}$ and $sig(\mathcal{T}) = Exp \to m_2\ \mathcal{T}$. Substitution, as a pure meta-function, is divided from F_p and described as a set of substitution rules R_{subst}, whose definition is defined by develops.

$$L = \langle S_p, C, R_{\mathcal{E}}, R_{\mathcal{T}}, R_{subst}, F_p, F_m \rangle$$

3.1.4 Code Generation.

3.2 Host Languages

Our host language is designed to define DSL. For the purpose of semantic derivation, each language constructs of the host language should have a structural computation. the evaluation of an expression $c\ e_1 \cdots e_n : Exp$ should be an operation of the evaluation of each subexpression e_i if $e_i : Exp$. If some e_i is used without evaluation, or some expression being computed is not e_i, then the rule is not structural. The evaluation rules of *if* and *and* are structural. Structural evaluation rules are the key to ensure the abstraction in semantic derivation. In other words, any language construct with unstructural evaluation rule needs to be carefully accounted for in terms of how it will affect the semantic derivation.

In the case of STLC, the evaluation rules of lambda abstraction and application are not structural. The former uses the e_i not evaluated, while the latter evaluates an expression containing substitution. We rewrite these two rules according to the meta-language definition above as follows, where $subst \in F_p$:

$$\mathcal{E}(\textbf{Abs}\ x\ t\ e) \coloneqq \textbf{Abs}\ x\ t\ \boxed{e}$$

$$\mathcal{E}(\textbf{App}\ e_1\ e_2) \coloneqq \textbf{let Abs}\ x\ t\ e = \mathcal{E}(e_1);\ \textbf{let}\ v_2 = \mathcal{E}(e_2);\ \boxed{\mathcal{E}(subst\ e\ x\ v_2)}\ .$$

Todo: Safety? cannot be expressed by big-step semantics

4 DSL DEFINITION

In this section, language extensions and language specialization will be discussed in more detail.

4.1 Meta-Extensions

Meta-extensions are direct extensions to the host language by adding new constructors C_n and corresponding rules, and functions F_n. This expansion has the following requirements: for each $c \in C_n$, (1) $sort(c) \in S_p$, i.e. *Exp* or *Type*; (2) if $sort(c) = Exp$, then the evaluation, typing and substitution rules should be specified.

Example: integer extension. Consider integers are expected to be supported via literals. New constructors, **Int** as *Type*, **Lit**, **Plus**, **Lt** and **Eq** as *Exp*, will be added. Fig. 9 shows the formal definition of integer extension, where $F_n = \{(+), (<)\}$, both of them are pure meta-functions. Todo: function Substitution will be modified after adding new constructors

4.2 Monad Extensions

As it was mentioned above, monad extension is the technique for side effects in computation without changing the original semantic definition. By monad m, individual language features can be defined, such as environment, store, nondeterminism, I/O etc. To put these together in a modular way, we use a mechanism called *monad transformers*, ranged by m_t. In this section, we will show the introduction of reference to STLC. Both the change of pure functions to those including computational effects via monad (in evaluation) and the expansion of language features via monad transformer (in typing) will be explained.

4.2.1 Evaluation with Store. Monad extensions are often accompanied by meta-extensions. The basic language constructs for reference are allocation, dereferencing and assignment. For reading convenience, we use expression with syntax like $ref\ e$, $!e$ and $e_1 := e_2$. To carry through the modification on store, the signature of $\mathcal{E}$ changes from

$$
\begin{array}{lll}
\textbf{Int} : \textit{Type} & & \\
\textbf{Lit} : \textit{Int} \longrightarrow \textit{Exp} & \mathcal{E}(\textbf{Lit}\ i) \coloneqq \textbf{Lit}\ i & \mathcal{T}(\textbf{Lit}\ i) \coloneqq \textbf{Int} \\
 & (\textbf{Lit}\ i)[e/x] = \textbf{Lit}\ i & \\
\textbf{Plus} : (\textit{Exp}, \textit{Exp}) \longrightarrow \textit{Exp} & \multicolumn{2}{l}{\mathcal{E}(\textbf{Plus}\ e_1\ e_2) \coloneqq \textbf{let Lit}\ i_1 = \mathcal{E}(e_1);\ \textbf{let Lit}\ i_2 = \mathcal{E}(e_2);\ \textbf{Lit}\ (i_1 + i_2)} \\
 & \multicolumn{2}{l}{\mathcal{T}(\textbf{Plus}\ e_1\ e_2) \coloneqq \textbf{let Int} = \mathcal{T}(e_1);\ \textbf{let Int} = \mathcal{T}(e_2);\ \textbf{Int}} \\
 & \multicolumn{2}{l}{(\textbf{Plus}\ e_1\ e_2)[e/x] = \textbf{Plus}\ e_1[e/x]\ e_2[e/x]} \\
\textbf{Lt} : (\textit{Exp}, \textit{Exp}) \longrightarrow \textit{Exp} & \multicolumn{2}{l}{\mathcal{E}(\textbf{Lt}\ e_1\ e_2) \coloneqq \textbf{let Lit}\ i_1 = \mathcal{E}(e_1);\ \mathcal{E}(e_2) : \begin{pmatrix} \textbf{Lit}\ i_2 \mid i_1 < i_2 \triangleright \textbf{True} \\ \textbf{Lit}\ \_ \triangleright \textbf{False} \end{pmatrix}} \\
 & \multicolumn{2}{l}{\mathcal{T}(\textbf{Lt}\ e_1\ e_2) \coloneqq \textbf{let Int} = \mathcal{T}(e_1);\ \textbf{let Int} = \mathcal{T}(e_2);\ \textbf{Bool}} \\
 & \multicolumn{2}{l}{(\textbf{Lt}\ e_1\ e_2)[e/x] = \textbf{Lt}\ e_1[e/x]\ e_2[e/x]} \\
\textbf{Eq} : (\textit{Exp}, \textit{Exp}) \longrightarrow \textit{Exp} & \multicolumn{2}{l}{\mathcal{E}(\textbf{Eq}\ e_1\ e_2) \coloneqq \textbf{let Lit}\ i_1 = \mathcal{E}(e_1);\ \mathcal{E}(e_2) : \begin{pmatrix} \textbf{Lit}\ i_2 \mid i_1 = i_2 \triangleright \textbf{True} \\ \textbf{Lit}\ \_ \triangleright \textbf{False} \end{pmatrix}} \\
 & \multicolumn{2}{l}{\mathcal{T}(\textbf{Eq}\ e_1\ e_2) \coloneqq \textbf{let Int} = \mathcal{T}(e_1);\ \textbf{let Int} = \mathcal{T}(e_2);\ \textbf{Bool}} \\
 & \multicolumn{2}{l}{(\textbf{Eq}\ e_1\ e_2)[e/x] = \textbf{Eq}\ e_1[e/x]\ e_2[e/x]}
\end{array}
$$

Fig. 9. Formal Definition of Integer Extension

$$
\begin{aligned}
\mathcal{E}(\textbf{Loc}\ \textit{loc}) &\coloneqq \textbf{Loc}\ \textit{loc} \\
\mathcal{E}(\textit{ref}\ e) &\coloneqq \textbf{let}\ v = \mathcal{E}(e);\ \textbf{let}\ \textit{loc} = \textit{newLoc};\ \textbf{Loc}\ \textit{loc} \\
\mathcal{E}(!e) &\coloneqq \textbf{let Loc}\ \textit{loc} = \mathcal{E}(e);\ \textit{fetch loc} \\
\mathcal{E}(e_1 \coloneqq e_2) &\coloneqq \textbf{let Loc}\ \textit{loc} = \mathcal{E}(e);\ \textbf{let}\ v_2 = \mathcal{E}(e_2);\ \textbf{let}\ () = \textit{store}\ (\textit{loc}, v);\ ()
\end{aligned}
$$

Fig. 10. Evaluation Rules of Ref

Exp $\longrightarrow$ *Exp*, to *Exp* $\longrightarrow$ (*State Store*) *Exp*, abbreviated as M_e *Exp*, where *Store* maps locations to values.

$$\mathcal{E} : \textit{Exp} \longrightarrow \textit{Exp}\ \boxed{\textbf{monad}\ \textit{State Store}\ \textbf{as}\ M_e}$$

An abstract index of store is a location (used as a base sort *Loc*). A reference will be evaluated to a location expression, constructed by **Loc** : *Loc* $\longrightarrow$ *Exp*. New (monadic) meta-functions are also necessary for evaluation rules:

- *newLoc* : M_e *Loc* returns a newly allocated location;
- *store* : (*Loc*, *Exp*) $\longrightarrow$ M_e () updates the store to make the location contain the value;
- *fetch* : *Loc* $\longrightarrow$ M_e *Exp* gets the value at the location from the store.

Given these meta-functions, Fig. 10 shows the evaluation rules of these newly defined language constructs.

When introducing monad, pure meta-functions keep the original definition, while monadic meta-functions f_m : $(s_1 \cdots s_n) \longrightarrow s$ need to lift to $f_m^{\uparrow} : (s_1 \cdots s_n) \longrightarrow m\ s$ for type correctness, defined by:

$$f_m^{\uparrow}\ \textit{args} = \textit{return}\ (f_m\ \textit{args}).$$

4.2.2 Typing with Store. If the content of a location has type t, then the type of the location is $\textit{Ref}\ t$, introducing new constructor **Ref** : *Type* $\longrightarrow$ *Type*. To extend the store typing which records the association between location and their

$$\mathcal{T}(\textbf{Loc}\ loc) \coloneqq \textbf{let}\ t = ask\ loc;\ Ref\ t$$
$$\mathcal{T}(ref\ e) \coloneqq \textbf{let}\ t = \mathcal{T}(e);\ Ref\ t$$
$$\mathcal{T}(!e) \coloneqq \textbf{let}\ Ref\ t = \mathcal{T}(e);\ t$$
$$\mathcal{T}(e_1 := e_2) \coloneqq \textbf{let}\ Ref\ t = \mathcal{T}(e_1);\ \mathcal{T}(e_2) : (t_2 \mid t = t_2 \triangleright Unit)$$

Fig. 11. Typing Rules of Ref

types, the signature of $\mathcal{T}$ changes from $Exp \to State\ CtxStack\ Type$, to $Exp \to ReaderT\ Store_t\ (State\ CtxStack)\ Type$, where $Store_t$ maps locations to types, and $ReaderT\ Store_t$ is a monad transformer.

$$\mathcal{T} : Exp \to Type\ \textbf{monad}\ State\ CtxStack,\ \boxed{Reader\ Store_t}\ \textbf{as}\ M_t$$

Monad transformers allow us touch high-level language features, but skill keep the access to low-level details. Now, for example, we can use both the meta-functions associated with store typing like *ask* and the meta-functions provided by Context like *searchCtx*. Fig. 11 gives the typing rules of language constructs in Ref.

Likewise, pure meta-functions remains unchanged, and all the monadic meta-functions $f_m : (s_1 \cdots s_n) \to m\ s$ need to lift to $f_m^{\uparrow} : (s_1 \cdots s_n) \to (m_t\ m)\ s$ by *lift* operation[?], so that the type rule of lambda abstraction in Ref will be written as:

$$\mathcal{T}(\lambda x : t.e) \coloneqq \textbf{let}\ \_ = updateCtx^{\uparrow}\ x\ t;\ \textbf{let}\ t' = \mathcal{T}(e);$$
$$\textbf{let}\ \_ = restoreCtx^{\uparrow};\ t \to t'.$$

4.3 Translation Rules

After defining a fully featured host language, translation rules are designed to specialize the language constructs of the DSL. The difference between a translation rule and a macro is that it is not provided by the host language, but is an abstraction outside the language. The distinction between translation rules and syntactic sugar is that it not only simplifies the syntax, but also can be used to describe more operations.

A translation rule tr define a new constructor in meta-language, with shape

$$c_s\ exp_{x_1} \cdots exp_{x_n} \Rightarrow exp_r,$$

where expression variables in exp_r must appear in LHS. We use $LHS(tr)$ to express LHS, and $RHS(tr) = exp_r$ to express RHS. We call constructors defined by translation rules *surface constructor*. If a closed expression exp is constructed by c_s, then there exists an environment Σ satisfying $\Sigma(LHS(tr)) = exp$, and $\Sigma(RHS(tr))$ is called *one-step translation* (or desugaring), written as $\mathbb{D}_1(exp)$. *Total translation* is also pretty useful, which expands the rules recursively. But if a rule translates to a constructor defined by itself, i.e., a recursive translation rule, then total translation is not terminated. Therefore, we propose the following requirement.

Requirement 4.1. For a translation rule tr, constructors in $RHS(tr)$ must be those of host language, or surface constructors defined earlier.

Formally, a total translation of expression $\mathbb{D}(exp)$ is defined as:

$$\begin{aligned}
\mathbb{D}(exp_b) &= exp_b \\
\mathbb{D}(c_s\ exp_1 \cdots exp_n) &= \mathbb{D}(\mathbb{D}_1(c_s\ exp_1 \cdots exp_n)) && \text{if } c_s \text{ is a surface constructor} \\
\mathbb{D}(c_h\ exp_1 \cdots exp_n) &= c_h\ \mathbb{D}(exp_1) \cdots \mathbb{D}(exp_n) && \text{if } c_h \text{ is a host constructor}
\end{aligned}$$

For example, given an expression *true and* (*false or true*) of Bool, then:

$$\mathbb{D}_1(\mathit{true\ and}\ (\mathit{false\ or\ true})) = \mathit{if\ \ true}\ (\mathit{false\ or\ true})\ \mathit{false}$$

$$\mathbb{D}(\mathit{true\ and}\ (\mathit{false\ or\ true})) = \mathit{if\ \ true}\ (\mathit{if\ false\ true\ true})\ \mathit{false}$$

4.3.1 Variable Scope in Translation Rules. We have very strict requirements for the scope of variables in the translation rules. Variables defined in the rule cannot be leaked, and external variables may not be captured in the rule. These translation rules are not premitted:

$$\mathit{leaked}\ e \Rightarrow \mathit{let}\ x : \mathit{int} = 1\ \mathit{in}\ e$$

$$\mathit{captured}\ e \Rightarrow \mathit{if}\ \ x\ \mathit{true}\ e$$

The former tries to bind a constant in x for use in e, like *leaked* $(x + 1)$; and the latter attempts to get the value of x in the current environment, like *let* x = *true in captured false*. From the user's point of view, it is weird to use variables that are not explicitly defined and use variables but not explicitly stated. And in the semantic derivation, the variables that are not substituted will be stuck. Taking all these considerations into account, we give the following requirements.

Requirement 4.2. A translation rule must be *closed*: any variable used in RHS has a local binding.

4.3.2 Hygiene Problem. Programming languages with non-hygienic macro systems may cause the hygiene problem[2]: variable bindings are possible to be hidden by macros, which we also have to face. For example, we define a translation rule or' via *let*:

$$e_1\ or'\ e_2 \Rightarrow \mathit{let}\ \text{“}x\text{”} = e_1\ \mathit{in\ if}\ \text{“}x\text{”}\ \text{“}x\text{”}\ e_2$$

where "x" is a literal identifier. Then the expression *let* x = *false in* (*true* or' x) will be totally translated into *let* x = *false in let* x = *true in if* $x\ x\ x$, causing an error. Fortunately, thanks to the requirement 4.2, the variables in the RHS must be locally bound. Therefore, we can modify the names of variables safely. *We treat literals variables bound in the RHS as mutable and always fresh.* We use @ to denote fresh variables, and or' will be written as:

$$e_1\ or'\ e_2 \Rightarrow \mathit{let}\ @x = e_1\ \mathit{in\ if}\ \ @x\ @x\ e_2.$$

4.3.3 Substitution Rules.

5 LANGUAGE LIFTING

Language lifting is the method for generating a stondalone DSL from an extended host language and translation rules. Ignoring the modification of the syntax, we will present how to derive computation rules and type rules. In particular, we will discuss language lifting for those whose host language is STLC or extended STLC.

5.1 Preprocessing: Lambda Lifting

We have already exemplified how lambda abstractions break the abstraction property in semantic derivation. The solution to this problem is to expose the lambda abstractions in the translation rules and define them as new language constructs of the DSL. For the outermost lambda abstractions, the transformation is straightforward. The inner lambda abstractions, on the other hand, need to be extracted recursively. The challenge here is to ensure that the translation rules are closed. Formally, for the translation rule tr which defines c'_n:

$$\uparrow_\lambda((\lambda x : t.e_1)\ e_2) = (\lambda x : t.\ \uparrow_\lambda(e_1))\ \uparrow_\lambda(e_2) \tag{1}$$

$$\uparrow_\lambda(\lambda x : t.e) = \lambda x : t.(c'_n\ \text{exp}_1 \,..\, \text{exp}_p\ x_1..x_q) \tag{2}$$

$$\text{generating } c'_n\ \text{exp}_1 \,..\, \text{exp}_p\ e_1..e_q \Rightarrow \uparrow_\lambda(e[e_1/x_1..e_q/x_q]) \tag{3}$$

$$\text{for each expression variable } \text{exp}_i \text{ in } e \text{ and variable } x_j \text{ not bound in } e \tag{4}$$

$$\uparrow_\lambda(c\ exp_1 \cdots exp_n) = c\ \uparrow_\lambda(exp_1) \cdots \uparrow_\lambda(exp_n) \tag{5}$$

$$\uparrow_\lambda(exp_x) = exp_x \tag{6}$$

$$\uparrow_\lambda(exp_b) = exp_b \tag{7}$$

where $\uparrow_\lambda(\bullet)$ is lambda lifting transformation, with the generation of some translation rules. The first rule states that if a lambda abstraction is applied directly in the translation rule, then it does not need to be lifted because it does not break the abstraction (which can essentially be written as a *let*). The second is the main rule, which generates a new constructor c'_n to describe the semantics of a lambda abstraction. Any expression variables of $LHS(tr)$ used in e (i.e., parameters of c_n) need to be applied by c'_n. In addition, any variable x_j which have been bound and captured by e are also required to be included, by an new expression variable e_j, which ensures the closure of translation rules. Note that the newly generated c'_n needs to process lambda lifting recursively, and may generate some more translation rules.

Todo: examples

Lemma 5.1. *Lambda lifting preserves requirements 4.1 and 4.2.*

5.2 Semantic Derivation

In order for the DSL to support new language constructs defined by translation rules, it is necessary to provide their evaluation rules and type rules. In this section, a generic algorithm will be presented, to illustrate how to derive the rules for translation rules. For a DSL defined by multiple translation rules, we derive the evaluation and typing rules and add them to the language individually.

Suppose a host language based on STLC is defined as $L = \langle S_p, C, R_{\mathcal{E}}, R_{\mathcal{T}}, R_{subst}, F\rangle$, where $S_p = \{Exp, Type\}$, and a translation rule tr has shape $c_n\ exp_1 \cdots exp_n \Rightarrow e$, where $sort(c_n) = Exp$ and tr satisfies the requirements. Let $L' = \langle S_p, C', R'_{\mathcal{E}}, R'_{\mathcal{T}}, R'_{subst}, F\rangle$ be the new language that supports c_n (i.e., $C' = C \cup \{c_n\}$).

5.2.1 Algorithm. In overview, the core idea of the semantic derivation has been shown: expand expressions evaluation recursively according to the rules in L until all the evaluation are unexpandable. Formally, $\mathcal{E}(LHS(tr)) = C(\mathcal{E}(RSH(tr)))$, the definition of C is shown in Fig. 12.

To illustrate the algorithm, some examples will be presented. Example 5.2 defines *nand* by other translation rules *not* and *and*. Assume that the semantics of *not* and *and* have been derived earlier. We will observe that how C works recursively. For clarity, the parts that need to be applied by C are underlined.

$$\mathcal{E}(LHS(tr)) = C(\mathcal{E}(RSH(tr)))$$

Body:

$$C(b : (br_1 \cdots br_n)) = C(b) : (C(br_1) \cdots C(br_n)) \tag{8}$$

$$C(\textbf{let}\ pat = b_1;\ b_2) = \textbf{let}\ pat = C(b_1);\ C(b_2) \tag{9}$$

Bone:

$$C(\mathcal{E}(exp_x)) = \mathcal{E}(exp_x) \tag{10}$$

$$C(\mathcal{E}(exp_b)) = \mathcal{E}(exp_b) \tag{11}$$

$$C(\mathcal{E}(c\ exp_1 \cdots exp_m)) = C(\Sigma(d)) \qquad \text{if there exists } \Sigma \text{ and } \mathcal{E}(exp) := d, \text{ such that } \Sigma(exp) = c\ exp_1 \cdots exp_m \tag{12}$$

$$C(\mathcal{E}(f_p\ (exp'_1 \cdots exp'_n))) = f_p\ (exp'_1 \cdots exp'_n) \tag{13}$$

$$C(f_m(exp'_1 \cdots exp'_n)) = f_m(exp'_1 \cdots exp'_n) \tag{14}$$

General Expression:

$$C(exp') = exp' \tag{15}$$

Branch:

$$C(pat \triangleright b) = pat \triangleright C(b) \tag{16}$$

$$C(pat \mid exp' \triangleright b) = pat \mid exp' \triangleright C(b) \tag{17}$$

Fig. 12. Semantic Derivation

Example 5.2. $e_1\ nand\ e_2 \Rightarrow not\ (e_1\ and\ e_2)$:

$$\begin{aligned}
&\mathcal{E}(e_1\ nand\ e_2) \\
&= \underline{C(\mathcal{E}(not\ (e_1\ and\ e_2)))} \\
&= \underline{C(\mathcal{E}(e_1\ and\ e_2) : \begin{pmatrix} true \triangleright false \\ false \triangleright true \end{pmatrix})} \\
&= \underline{C(\mathcal{E}(e_1\ and\ e_2))} : \begin{pmatrix} true \triangleright \underline{C(false)} \\ false \triangleright \underline{C(true)} \end{pmatrix} \\
&= \underline{C(\mathcal{E}(e_1) : \begin{pmatrix} true \triangleright \mathcal{E}(e_2) \\ false \triangleright false \end{pmatrix})} : \begin{pmatrix} true \triangleright false \\ false \triangleright true \end{pmatrix} \\
&= \mathcal{E}(e_1) : \begin{pmatrix} true \triangleright \mathcal{E}(e_2) \\ false \triangleright false \end{pmatrix} : \begin{pmatrix} true \triangleright false \\ false \triangleright true \end{pmatrix}
\end{aligned}$$

The language construct *let* is a common syntactic sugar, defined by lambda abstraction and application. Considering *let* as a translation rule, example 5.3 presents the derivation of its evaluation rule. Note that the RHS of the translation rule satisfies 1, and no language constructs will be generated in lambda lifting. The last step is simplification, since that both left and right sides are constructed by **Abs** (i.e., λ in language). simplification is sometimes useful for abstraction property, for instance in this example, where the evaluation rule of *let* no longer depends on lambda abstraction (even if this is allowed). In addition, substituion, as a meta-function, appears in the following derivation, which does not affect the expansion. In Todo: , we will discuss substituion more deeply.

Example 5.3. $let\ x : t = e_1\ in\ e_2 \Rightarrow (\lambda x : t.e_2)\ e_1$:

$$
\begin{aligned}
&\mathcal{E}(let\ x : t = e_1\ in\ e_2)\\
=\ &\underline{C(\mathcal{E}((\lambda x : t.e_2)\ e_1))}\\
=\ &\underline{C(\textbf{let}\ \lambda x' : t'.e = \mathcal{E}(\lambda x : t.e_2);\ \textbf{let}\ v_1 = \mathcal{E}(e_1);\ \mathcal{E}(e[v_1/x']))}\\
=\ &\textbf{let}\ \lambda x' : t'.e = \underline{C(\mathcal{E}(\lambda x : t.e_2))};\ \underline{C(\textbf{let}\ v_1 = \mathcal{E}(e_1);\ \mathcal{E}(e[v_1/x']))}\\
=\ &\textbf{let}\ \lambda x' : t'.e = \lambda x : t.e_2;\ \textbf{let}\ v_1 = \underline{C(\mathcal{E}(e_1))};\ \underline{C(\mathcal{E}(e[v_1/x']))}\\
=\ &\textbf{let}\ \lambda x' : t'.e = \lambda x : t.e_2;\ \textbf{let}\ v_1 = \mathcal{E}(e_1);\ \mathcal{E}(e[v_1/x'])\\
=\ &\textbf{let}\ x' = x;\ \textbf{let}\ t' = t;\ \textbf{let}\ e = e_2;\ \textbf{let}\ v_1 = \mathcal{E}(e_1);\ \mathcal{E}(e[v_1/x'])
\end{aligned}
$$

Correctness is that, given a closed expression $e : Exp$ in L', the value by evaluation should be the same as the value, of fully translating first and then evaluating in L. We use subscripts to distinguish in which language the computation is performed. Then:

$$
\begin{aligned}
\mathcal{E}_{L'}(e) &= \mathcal{E}_L(\mathbb{D}(e))\\
\mathcal{T}_{L'}(e) &= \mathcal{T}_L(\mathbb{D}(e))
\end{aligned}
$$

If e is constructed by c_n, then further, the following equation holds:

$$
\begin{aligned}
\mathcal{E}_{L'}(e) &= \mathcal{E}_{L'}(\mathbb{D}_1(e))\\
\mathcal{T}_{L'}(e) &= \mathcal{T}_{L'}(\mathbb{D}_1(e))
\end{aligned}
$$

REFERENCES

[1] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. 2019. Skeletal Semantics and Their Interpretations. *Proc. ACM Program. Lang.* 3, POPL, Article 44 (jan 2019), 31 pages. https://doi.org/10.1145/3290357

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

[3] Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: Lifting Evaluation Sequences through Syntactic Sugar. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 361–371. https://doi.org/10.1145/2594291.2594319

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

[5] Martin P. Ward. 1994. Language-Oriented Programming. *Softw. Concepts Tools* 15 (1994), 147–161.

[6] Ziyi Yang, Yushuo Xiao, Zhichao Guan, and Zhenjiang Hu. 2022. A Lazy Desugaring System for Evaluating Programs With Sugars. In *Functional and Logic Programming: 16th International Symposium, FLOPS 2022, Kyoto, Japan, May 10–12, 2022, Proceedings* (Kyoto, Japan). Springer-Verlag, Berlin, Heidelberg, 243–261. https://doi.org/10.1007/978-3-030-99461-7_14

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009