International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Correct and Abstract Semantics Lifting for Domain-Specific Language Implementation

Hyacinthe Hamon

Hamon FZCO, Research And Development

Abstract

Implementing domain-specific languages (DSLs) through higher-order functions and macros often leads to abstraction leakage, diminishing their user-friendliness. This paper introduces a semantics-lifting framework to address this challenge. The proposed framework offers a general algorithm to derive DSL semantics independent of the host language based on host semantics and translation rules. It formulates cor- directness and abstraction properties, ensuring the semantics lifting process maintains these properties. Additionally, the paper explores the necessary assumptions for achieving correct and abstract lifted semantics, demonstrated through the implemented system Osazone. Case studies across various host languages, including functional and imperative types, confirm the flexibility and reliability of the framework, ensuring the lifted DSL semantics preserve correctness and abstraction integrity throughout.

KEYWORDS: Domain-Specific Languages (DSLs), Semantics Lifting, Abstraction Leakage, Higher-Order Functions, Correctness and Abstraction Properties, Host-Independent DSL Semantics

1. INTRODUCTION

Language-oriented programming (LOP) [14] involves solving a specific class of problems by creating new domain-specific languages (DSLs). To avoid re-implementing common language constructs such as loops and branches, developers usually implement DSLs on top of another general-purpose programming language, which is usually called the host language. Developers can define a DSL using language features supported by the host language, such that the DSL programs are specialized programs with domain-specific syntactic forms. Languages with features such as macros and higher-order functions are commonly used as host languages [2, 3]. These features provide methods to implement the translation from DSLs to the host language. By specifying the translation rules, developers can obtain DSL interpreters for free. DSLs implemented in this way are often called embedded DSLs (EDSLs).

EDSLs have the potential to reduce implementation efforts significantly. However, their convenience is hindered by the requirement for users to understand host-level language concepts and possible error messages. This is a type of abstraction leakage [13]: programs that the user writes must be translated into the host language before executing. Thus, the abstraction boundary is not preserved, and it is challenging to retrieve DSL-level information during the execution. Pombrio et al. have significantly contributed to preserving the abstraction boundary that a DSL aims to establish. Their approach considers DSLs are implemented using syntactic sugars (i.e., language constructs defined by translation to the host language) and achieve this objective by recovering evaluation traces in the core language to traces in the DSL through a process called sugaring [10]. The sugaring process selectively reorganizes the sequence of evaluations in the host language to the DSL according to the reverse translation rules, thereby maintaining the

abstraction boundary dynamically. It is noteworthy, however, that errors during the host language execution can still lead to abstraction leakage during the evaluation. Specifically, error messages may not be translatable back to the DSL level. Consequently, DSL users may observe that the evaluation is stuck but cannot pinpoint the location of the error.

One interesting step towards resolving this problem is type lifting [11], which aims to maintain abstraction boundaries during type checking. This technique statically infers typing rules for newly defined language constructs based on the typing rules of the host language and simple syntactic sugar definitions. The inferred typing rules of a DSL can be used for static analysis in Integrated Development Environments (IDE), ultimately facilitating the generation of high-quality compile-time error messages for DSLs.

Inspired by the idea of type lifting, I shall propose a new technique called semantics lifting, which aims to derive the semantics of a domain-specific language statically to overcome the limitations of desugaring. Using my technique, DSL developers only need to provide the translation rules from the DSL to the host language, and then they obtain self-contained evaluation rules of lifted semantics for the DSL for free! My primary desideratum is to preserve the abstraction boundary of the DSL, i.e., the evaluation of a program in DSL does not reveal details related to the host language to the user. This lifted semantics can assist users in diagnosing run-time errors in their DSL programs because all evaluation steps are defined over the DSL constructs only. Thus, users can identify the sub-expression that caused the error.

It is possible but not easy to adapt the core ideas of the type-lifting algorithm for semantics lifting. For instance, with a host language including lambda abstraction and application, I can define a new language construct by the following translation rule (sometimes called a desugaring rule):

$$let\ \mathrm{x} = \mathrm{e}_1\ in\ \mathrm{e}_2 \rightarrow_d (\lambda \mathrm{x}.\ \mathrm{e}_2)\ \mathrm{e}_1.$$

Following the lifting algorithm for types, I can obtain the derivation of the evaluation rule for let as shown in Fig. 1. First, a let expression is evaluated to a value v in DSL if the translated expression is evaluated to v in the host language (Step 1). Then, by the evaluation rule of application in the host, I expand the premise (Step 2). Finally, since a lambda abstraction is already a value with no premises (Step 3), I can, therefore obtain the following evaluation rule for let :

$$\frac{\mathrm{e}_1 \Downarrow \mathrm{v}_1 \quad [\mathrm{x} \rightarrow \mathrm{v}_1]\mathrm{e}_2 \Downarrow \mathrm{v}}{let\ \mathrm{x} = \mathrm{e}_1\ in\ \mathrm{e}_2 \Downarrow \mathrm{v}}$$

$$\frac{\dfrac{\lambda \mathrm{x}.\mathrm{e}_2 \Downarrow \lambda \mathrm{x}.\mathrm{e}_2 \quad \mathrm{e}_1 \Downarrow \mathrm{e}_1 \quad [\mathrm{x} \rightarrow \mathrm{v}_1]\mathrm{e}_2 \Downarrow \mathrm{v}}{(\lambda \mathrm{x}.\mathrm{e}_2)\ \mathrm{e}_1 \Downarrow \mathrm{v}}\ (\text{Step 2})}{\text{let } \mathrm{x} = \mathrm{e}_1 \text{ in } \mathrm{e}_2 \Downarrow \mathrm{v}}\ (\text{Step 1})$$

Figure 1: Evaluation Rule Derivation of *let*

It is important to note that this evaluation rule of let is described without any use of lambda abstraction and application, two host language constructs used for defining let .

However, not everything works well like this.

Consider the following translation rule

$$and_f \rightarrow_d \lambda \mathrm{x}.\ \lambda \mathrm{y}.\ \text{if x then y else false}$$

which defines a language construct and_f in DSL over the host language. Following similar steps as above,

I can construct a derivation tree and obtain the following evaluation rule for and_f :

$$and_f \overset{t}{\Downarrow} \lambda x.\ \lambda y.\ \textit{if}\ x\ \textit{then}\ y\ \textit{else}\ \textit{false}.$$

The rule still uses the host language construct if it leads to abstraction leakage because the evaluation of a DSL program may use the evaluation rules of the host language.

This example reveals an important difference between evaluation and typing rules. In typing rules, the type of an expression is usually determined by its sub-expression types. By applying the type rules recursively, the resulting typing rules of DSL constructs defined by the translation rules are eventually premised on the types of their sub-expressions. However, such a composition-quality property may not hold for evaluation rules. For example, it is not always possible to statically determine where a lambda abstraction is applied, Thus, it is impossible to evaluate the abstraction body statically. When a lambda abstraction is used in a translation rule, the evaluation of these host language constructs in the body is delayed until lambda abstraction is applied at run-time, which induces abstraction leakage.

In this paper, I present a general framework for semantics lifting, addressing the abovementioned difficulties abovementioned difficulties.

With this framework, developers define a DSL via translation rules from the DSL to the host language and automatically obtain evaluation rules for the DSL. These DSLs with the obtained evaluation rules become independent of the host language, such that users can be entirely agnostic of the host language. My main technical contributions are summarized as follows:

I propose a general semantics-lifting algorithm and provide sufficient conditions for guaranteeing the correctness of the algorithm.

I present a systematic theory that clarifies the assumptions about the host language, meta-functions, and translation rules. I prove that semantics lifting is correct and preserves abstraction boundaries under a few reasonable assumptions.

I have implemented the framework as a system called Osazone, which allows users to define DSL translation rules and automatically generates an interpreter for the DSL. I will present several non-trivial case studies to show the power of Osazone.

2. OVERVIEW

My overall goal is to generate DSL evaluation rules from user-defined translation rules of the DSL and the host language evaluation rules. The evaluation result of a DSL program should be equivalent to the evaluation result of the same program through translation and subsequent evaluation in the host language. I use D(e) to denote the translation from a DSL expression e to the host language. Then, I prove a standard correctness property:

Goal 1 (Correctness) *The correctness goal can be divided into the following two subgoals:*

– **Soundness:** If e ⇓ v, then D(e) ⇓ D(v)

– **Completeness:** If D(e) ⇓ v′, then there exists v, s.t. D(v) = v′ and e ⇓ v

where ⇓ (DSL) denotes the evaluation in DSL, and ⇓ (Host) denotes the evaluation in host languag

I use D(v) on the right-hand side in the soundness property. This is because the values of the two languages could be different even though I allow language constructs for values of the host language to be used in the DSL. To clarify this point, consider a DSL program that evaluates to λx. λy. and x y, where and is defined in Sec. 1. In the meantime, the corresponding host program should evaluate to λx. λy. If x then y else false. The completeness property follows a similar pattern.

My second goal concerns abstraction. Operationally, I expect that the evaluation sequence of a DSL program should be understandable to DSL users, meaning the language constructs used in the sequence should be values and DSL constructs. This concept is referred to as global abstraction, and it is reflected in the big-step operational semantics by the absence of host-language constructs in the derivation tree of the evaluation of a DSL program.

Goal 2 (Abstraction) The derivation tree of a DSL program evaluation can only mention language constructs of values and the DSL. ⊔⊓

In the rest of this section, I use a small functional language called STLC as the host language, whose syntax and semantics are illustrated in Fig. 2, to demonstrate how my framework works.

$$e ::= true \mid false \mid if\ e\ then\ e\ else\ e \mid x \mid \lambda x.\ e \mid e\ e \mid e + e \qquad v ::= true \mid false \mid \mid \lambda x.\ e$$

$$\frac{}{v \Downarrow v} \qquad \frac{e_1 \Downarrow true \quad e_2 \Downarrow v_2}{if\ e_1\ then\ e_2\ else\ e_3 \Downarrow v_2} \qquad \frac{e1 \Downarrow false \quad e_3 \Downarrow v_3}{if\ e_1\ then\ e_2\ else\ e_3 \Downarrow v_3}$$

$$\frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad [x' \rightarrow v_2]e \Downarrow v}{e1\ e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2 \quad i = plus(i_1, i_2)}{e_1 + e_2 \Downarrow v}$$

Figure 2: The Syntax and Semantics of STLC

$$not\ e \rightarrow_d if\ e\ then\ false\ else\ true$$
$$e_1\ and\ e_2 \rightarrow_d if\ e_1\ then\ e_2\ else\ false$$
$$e_1\ or\ e_2 \rightarrow_d if\ e_1\ then\ true\ else\ e_2$$
$$e_1\ nand\ e_2 \rightarrow_d not\ (e_1\ and\ e_2)$$
$$e_1\ nor\ e_2 \rightarrow_d not\ (e_1\ or\ e_2)$$
$$e_1\ xor\ e_2 \rightarrow_d (e_1\ and\ (not\ e_2))\ or\ ((not\ e_1)\ and\ e_2)$$

Figure 3: Translation Rules for BOOL

2.1 DERIVATION OF EVALUATION RULES

This section provides an example to illustrate the aforementioned goals further. Let us consider the scenario where developers wish to implement a DSL called Bool to carry out Boolean operations. Figure 3 showcases how Bool is defined through translation rules.

Each translation rule, represented by tr, includes a left-hand side (LHS) and a right-hand side (RHS) separated by $\rightarrow_d$. The LHS is a newly defined language construct with meta-variables as placeholders for expressions, while the RHS is composed of constructs in the host language and these meta-variables. Every translation rule defines a unique language construct in DSL.

In contrast to each language construct typically having only one typing rule, the evaluation of a language construct (such as if) is sometimes described by multiple rules for different cases. As a result, in the semantics-lifting process, I must derive an evaluation rule for each case. For instance, to derive the evaluation rules of and, I can construct the following inference tree:

$$\frac{\textit{if}\ e_1\ \textit{then}\ e_2\ \textit{else}\ \textit{false} \Downarrow v}{e_1\ \textit{and}\ e_2 \Downarrow v}$$

And then, the evaluation rule for *if* e_1 *then* e_2 *else* e_3 dependents on whether e_1 evaluates to *true* or *false*. Hence, my derivation tree will also be divided into
two parts:

$$\frac{\dfrac{e_1 \Downarrow \textit{true} \quad e_2 \Downarrow v}{\textit{if}\ e_1\ \textit{then}\ e_2\ \textit{else}\ \textit{false} \Downarrow v}}{e_1\ \textit{and}\ e_2 \Downarrow v} \qquad \frac{\dfrac{e_1 \Downarrow \textit{false} \quad \dfrac{}{\textit{false} \Downarrow \textit{false}}}{\textit{if}\ e_1\ \textit{then}\ e_2\ \textit{else}\ \textit{false} \Downarrow \textit{false}}}{e_1\ \textit{and}\ e_2 \Downarrow \textit{false}}$$

These two rules demonstrate that the outcome of an expression with outermost and solely depends on the evaluation results of e_1 and e_2. Thus, when I substitute specific DSL expressions for e_1 and e_2, the above derivation tem- plate necessarily becomes the root of the inference tree. As a result, there is no need to derive it for every expression constructed using and . By removing the intermediate derivation, I obtain the evaluation rules of and :

$$\frac{e_1 \Downarrow \textit{true} \quad e_2 \Downarrow v}{e_1\ \textit{and}\ e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow \textit{false}}{e_1\ \textit{and}\ e_2 \Downarrow \textit{false}}$$

In the above derivation, I have fully applied the evaluation rules of if statically, which guarantees that the premises of obtained evaluation rules no longer contain it. So, when a DSL program contains and is evaluated, the premise is to evaluate the two sub-expressions and if they will not be generated at this stage (goal 2).

Where e1 and e_2 are DSL expressions, through induction, I can demonstrate that the value v of evaluating e_2 in the DSL is equal to the value v′ of evaluating D(e_2) in the host language after translation. Therefore, I can conclude that goal 1 has been met.

Similarly, I can derive evaluation rules for the remaining language constructs in Bool. It is worth noting that in the translation rule for nand, I use DSL constructs, not and. This is allowed since I can consider these translation rules as being progressively incorporated into the host language, meaning that nand is conceptually defined based on a host language that already includes and does not. However, it should be noted that translation rules with cyclic dependencies, such as recursively defined translation rules, are not permissible.

2.2. ASSUMPTION FOR CORRECTNESS

In this section, I present a sufficient assumption to satisfy correctness. The assumption concerns meta-functions in the host language: Meta-function calls should be commutable with translation.

Meta-functions are functions in the meta-language for semantics definitions. In the semantics of Stlc (presented in Fig. 2), there are two meta-functions: substitution and. There is a distinction between them: the former is a syntactic substitution whose output is an expression used for further evaluation. At the same time, the latter describes a computation whose input and output are values.

In semantics lifting, meta-functions are often reserved in the derived evaluation rules.

Therefore, the correctness of a lifted DSL semantics depends on the definition of meta-functions used. Specifically, the substitution rule should satisfy:

$$D([x \mapsto e_1]e_2) = [x \mapsto D(e_1)]D(e_2)$$

For any other meta-function f, the following equation needs to be satisfied:

$$D(f(v_1, \cdots, v_n)) = f(D(v_1), \cdots, D(v_n))$$

Intuitively, this means that the meta-function f must be defined according to the meaning of its arguments rather than their syntactic structure. As a counterexample, I define a meta-function evil that does not satisfy this assumption:

$$evil(v) = \begin{cases} true & \text{if } v = \lambda x.\ \textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3 \\ false & otherwise \end{cases}$$

The meta-function evil is ill-behaved in semantics lifting because
evil(λx. if x then x else false) = true but evil(λx. x and x) = false.

2.3. ASSUMPTIONS FOR ABSTRACTION

This section describes the assumptions necessary for achieving the abstraction goal. These assumptions are:

2.3.1. Meta-functions: Each meta-function should be closed over a given set of language constructs.

2.3.2. Host language: The evaluation of a host-language expression should consist of evaluating its sub-expressions and meta-function calls.

2.3.3. Translation rules: In the RHS of a translation rule, host-language constructs cannot be used in any non-abstract component.

No need to panic! I will explain these assumptions one by one.

Meta-functions The assumption about meta-functions is crucial in reducing the global abstraction property of evaluation derivations to local abstraction criteria for each lifted evaluation rule. Intuitively, when I use expressions or values of the DSL as arguments to a meta-function call, its result should not contain any constructs from the host language. In other words, meta-functions should be closed under the DSL constructs. Hence, given a DSL program, if the evaluation of its sub-expressions does not generate host-language constructs or meta-function calls, abstraction can be guaranteed in evaluating the DSL program. Thus, I transform global abstraction into local abstraction: derived evaluation rules of DSL can only mention language constructs of values and DSL.

As an example, the following meta-function does not satisfy the above assumption:

$$f_{if}(v_1, v_2) = \lambda x.\ \mathit{if}\ x\ \mathit{then}\ v_1\ \mathit{else}\ v_2$$

Host Language Essentially, I require that all evaluation rules of the host language be premised on evaluations of its sub-expressions and meta-function calls. This means that no other host-language constructs are allowed to be introduced in the premises. Otherwise, these constructs may be used in the derived evaluation rules, which breaks the abstraction.

Translation Rules My assumptions about the translation rules can be justified with the evaluation-rule derivation of and given in Sec. 1. The evaluation rule of and breaks abstraction because it is defined via the lambda abstraction, and I use the host language constructs if in the body. Since the body of a lambda is not evaluated until it is invoked, the evaluation of its body cannot be derived statically and locally. I say e is a non-abstract component of the lambda abstraction λx. e. In general, in the evaluation rules of a language construct, a sub-expression is a non-abstract component of that language construct if it is used directly in the result or as an argument of a meta-function application. I assume that in the definition of translation rules, host-language constructs cannot be used in such components.

The assumptions on translation rules are less restrictive than one might first think, as I can introduce auxiliary language constructs to define abstraction-breaking components explicitly. For example, for the translation rule, I can introduce a new DSL construct ′ to express the semantics of the lambda abstraction's body and define and use this newly defined DSL construct.

$$\mathit{andf}'\ e_1\ e_2 \rightarrow_d \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ \mathit{false} \qquad \mathit{andf}$$
$$\rightarrow_d \lambda x.\ \lambda y.\ \mathit{andf}'\ x\ y$$

Since andf' is a DSL construct, the definition of andf satisfies the assumption above. And their evaluation rules can be derived as follows:

$$\frac{e_1 \Downarrow \mathit{true} \quad e_2 \Downarrow v_2}{\mathit{andf}'\ e_1\ e_2 \Downarrow v_2} \qquad \frac{e_1 \Downarrow \mathit{false}}{\mathit{andf}'\ e_1\ e_2 \Downarrow \mathit{false}} \qquad \frac{}{\mathit{andf} \Downarrow \lambda x.\ \lambda y.\ \mathit{andf}'\ x\ y}$$

Hence, the abstraction property of and is satisfied. For outermost lambda ab- stractions, the transformation is straightforward. Note that inner lambda abstractions need to be extracted recursively. For lambda

abstraction, I can automatically generate auxiliary constructs using lambda lifting [4]. After this re-processing of translation rules, the above assumptions are satisfied.

3. THEORY

In this section, I develop the theory of my semantics-lifting framework. In Sec. 3.1, I give a functional language with reference Func as the host language and formalize the non-abstract component of a language construct. In Sec. 3.2, I discuss the definition and requirements of translation rules. In Sec. 3.3, I formalize the semantics-lifting algorithm with several examples. Finally, in Sec. 3.4, I show the desired properties hold, i.e., correctness and abstraction.

3.1. HOST LANGUAGE

My approach supports a wide range of host languages. In particular, I support host languages with side effects that can be described using monads [8, 9]. Many practical language features can be defined with monads, such as environment, store, non-determinism, I/O, etc. In some instances, multiple effects can be combined by a monad transformer mechanism [7].

$$
\begin{aligned}
e ::= {} & x \mid \lambda x.\ e \mid (e\ e)V \mid (e\ e)_N \mid true \mid false \mid if\ e\ then\ e\ else\ e \\
& \mid\mid e + e \mid e = e \mid e < e \mid () \mid let\ x = e\ in\ e \mid e; e \\
& \mid (e, e) \mid e.1 \mid e.2 \mid inl\ e \mid inr\ e \mid case\ e\ of\ \{inl\ x \Rightarrow e \mid inr\ x \Rightarrow e\} \\
& \mid nil \mid cons\ e\ e \mid head\ e \mid tail\ e \mid isnil\ e \mid fix\ e \\
& \mid ref\ e \mid !e \mid e := e \mid l \qquad\qquad v \\
::= {} & \lambda x.\ e \mid true \mid false \mid \\
& \mid () \mid (v, v) \mid inl\ v \mid inr\ v \mid nil \mid cons\ v\ v \mid l
\end{aligned}
$$

Figure 4: Syntax of FUNC

I present an example host language, Func, which uses state monad to implement references. The syntax of this language is given in Fig. 4.

I introduce two types of applications in Func: call-by-value and call-by-name. I use $\Downarrow_S$ for evaluation with the state monad, where the state maps store locations to values.

The monad provides meta-functions like alloc for state management, and I use the notation $\Rightarrow_S$ to express the return value of meta-functions. Below are the forms of evaluation judgments and rules:

$$
\begin{aligned}
j \in \text{Judgement} ::= {} & e \Downarrow_S v \mid [x \rightarrow e_1, \ldots]e \Downarrow_S v \mid f(e_1, \cdots, e_n) \Rightarrow_S v \\
r \in \text{Rule} ::= {} & \frac{j1 \quad \cdots \quad jn}{c\ e_1 \cdots e_n \Downarrow_S v}
\end{aligned}
$$

There are two judgments meta-functions, similar to the setting I discussed in Sec. 2.2. One of them is the substitution at the syntactic level, whose output is an expression for evaluation. The other is monadic computation at the semantics level, ranging over by f. I require that the output of the latter kind of meta-functions be values. Some of the evaluation rules of Func are shown in Fig. 5. Because I pass state

implicitly through the state monad, the judgments in the premises must be seen as operations performed from left to right and cannot be exchanged with each other. For example, in Haskell, the evaluation of ref e can be defined as roughly follows:

$$eval\ (Ref\ e) = do\ \{v \leftarrow eval\ e; l \leftarrow alloc; () \leftarrow store\ l\ v; return\ l\}$$

I use R(c $e_1 \cdots e_n$) or R(c) to denote the rules of a language construct c.

$$\frac{e_1 \Downarrow_S v_1\ e_2 \Downarrow S\ v2\ plus(v_1, v_2) \Rightarrow_S v}{e_1 + e_2 \Downarrow_S v}$$

$$\frac{e_1 \Downarrow_S v_1\ [x \rightarrow v_1]e_2 \Downarrow_S v}{let\ x = e_1\ in\ e_2 \Downarrow_S v}$$

$$\frac{e \Downarrow_S \lambda x.\ e'[x \rightarrow fix\ (\lambda x.\ e')]e' \Downarrow_S v}{fix\ e \Downarrow_S v}$$

$$\frac{e \Downarrow_S v\ \ alloc()\ \Rightarrow_S l\ store(l, v) \Rightarrow_S ()}{ref\ e \Downarrow_S l}$$

$$\frac{e \Downarrow_S l\ \ fetch(l) \Rightarrow_S v}{!e \Downarrow_S v}$$

$$\frac{e_1 \Downarrow_S le_2\ \Downarrow_S v\ store(l, v) \Rightarrow_S ()}{e_1 := e_2\ \Downarrow_S ()}$$

Figure 5: Selected Evaluation Rules of FUNC

Extract Non-abstract Components In Sec. 2.3, I have demonstrated that the body of a lambda expression is non-abstract, as the body evaluation will be delayed. This delay in evaluation can lead to issues with the abstraction property if a host-language construct is used within the lambda expression's body. It will be reserved in the evaluation rules of the DSL construct, resulting in an undesirable violation of the abstraction property.

I call e in λx. e a non-abstract component. In the RHSs of translation rules, I cannot use host-language constructs in these non-abstract components. Roughly speaking, a sub-expression ei is a non-abstract component of a language construct c e1 · · · en if the sub-expression ei is:

used directly in the evaluated result or

passed as an argument to meta-functions in the evaluation rules of c.

In Func, most language constructs have no non-abstract components; the only exceptions are lambda abstraction and call-by-name application. In a call-by-name application $(e_1\ e_2)_N$, the expression e_2 is used as an argument of a meta-function (i.e., substitution). Hence, in the translation rules defined by the call-by-name application, e_2 will not be evaluated until the substitution occurs, i.e., the evaluation of e_2 is

delayed.

Formally, I use $\delta(c_h\, e_1 \cdots e_n)$ to extract all the non-abstract components of c_h, where c_h is a host-language construct and $e_1, \cdots, e_n$ are meta-variables. The formal definition of δ is shown in Fig. 6. Here are two examples:

$$\delta(\lambda x.\ e) = \{e\} \qquad \delta((e_1\ e_2)_N) = \{e_2\}$$

3.2. Translation Rules

A translation rule defines a new language construct by showing how to translate it into the host language. A rule for defining a new construct has the following

$$\delta(c_h\ e_1 \cdots e_n) = \bigcup_{r} \delta r\,(r)$$

$$\cdot\left(\frac{j1 \cdots jm}{\text{ch e1...en} \Downarrow_S \text{v}}\right) = \delta^{\{e_1 \cdots e_n\} i_{(j)}} \cup \delta^{\{e_1 \cdots e_n\}}(v)$$

$$\delta^{P}(e \Downarrow_S v) = \emptyset \qquad \text{if } e \in P$$

$$\delta^{P}([x_1 \text{'}\!\rightarrow e_1' \bullet\bullet\bullet x_p \text{'}\!\rightarrow e_p']e' \Downarrow S\ v) = [\,\delta^{P}(e') \cup \delta^{P}(e') \cup \delta^{P}(v)$$

$$\delta^{P}(f(e_1 \bullet\bullet\bullet e_q) \Rightarrow_S v) = [\,\delta^{P}(e_i) \cup \delta^{P}(v)$$

$$\delta P\,(c'_n\ e_1 \ldots e_n) = \delta^{P_c}(e_i)$$

$$\delta^{P}(e) \begin{cases} \{e\} & \text{if } e \in P \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 6: Non-abstract Components Extraction shape:

$c_n\ \alpha_1 \cdots \alpha_n \rightarrow_d e,$

where αi are meta-variables for expressions that may appear in the RHS. I call constructs defined by translation rules surface constructs.

TRANSLATION I use D(•) to denote the full translation from a DSL expression to the corresponding host expression. Intuitively, it recursively expands DSL constructs to their definitions and maintains the structure of host constructs. Formally:

$$D(c_h\ e_1 \cdots e_n) = c_h\ D(e_1) \cdots D(e_n) \qquad \text{if } c_h \text{ is a host language construct}$$
$$D(c_n\ e_1 \cdots e_n) = D(\sigma(e)) \qquad \text{if } c_n \text{ is defined by } c_n\ \alpha_1 \cdots \alpha_n \rightarrow_d e \text{ and } \sigma = \{\alpha_1 \text{'}\!\rightarrow e_1, \ldots, \alpha_n \text{'}\!\rightarrow e_n\}$$

where σ is a substitution. Translation can be extended into judgments as:

$D(e \Downarrow_S v) = D(e) \Downarrow_S D(v).$

REQUIREMENTS To ensure that translation rules are properly defined, I impose the following requirements:

Unique. A unique translation rule must define each surface construct.

Non-recursive. Language constructs in the RHS must belong to the host language or be surface constructs defined earlier.

Closed. The translation rule must be closed, meaning any RHS variable must be bound locally.

The first requirement ensures that the applicable rule for each surface construct is determined during translation. The second requirement states that translation rules cannot be defined recursively because recursive rules possibly lead to non-terminating translation.

The third requirement pertains to the scope of the variables. Infer-scope provides a method to infer scopes through translation rules, and I adopt their approach in my framework. By enforcing this requirement, scope safety is ensured. Specifically, the requirement prohibits capturing external variables and requires that locally bound variables be explicitly parameterized.

Example 1. Consider the following two translation rules and their sample programs:

leaked e $\rightarrow_d$ let x = 1 in e (✓) leaked (x + 1) (✗)

captured e $\rightarrow_d$ if x true e (✗) let x = true in captured false (✗)

The first leaked rule attempts to bind the constant value 1 to the variable x, whose scope is e. However, since x is bound locally in the translation rule and not declared in the DSL program, using x in e is not allowed. (But the translation rule is permitted.) Alternatively, I can use the following definition to comply with the requirement:

leaked ′ x e $\rightarrow_d$ let x = 1 in e,

Moreover, x can be used in e of leaked ′.

The second rule, captured, tries to obtain the value of x from the current environment, which may result in unbound identifiers after translation. The translation rule itself is not allowed.

Enforcing the requirement of closed translation rules ensures that my translation system is hygienic. Programming languages with non-hygienic macro systems may cause a hygiene problem, where variable bindings can potentially hidden by macros and vice versa [5]. For instance, suppose I define a translation rule or ′ via let :

e_1 or ′ e_2 $\rightarrow_d$ let x = e_1 in if x then x else e_2

Where x is a literal identifier, then the expression let x = false in (faithful or′ x) will be translated into let x = false in let x = valid in if x then x else x, causing an unintended behavior since the same variable x is used in both the inner and outer scopes. Closed translation rules ensure no unbound-identifier exceptions [12]. Therefore, I can modify the names of variables safely, just like α-equivalence. I treat literal variables bound in the RHS as exchangeable and always fresh. Thus, the translation is always hygienic.

3.3 SEMANTICS LIFTING

In this section, I formalize the semantics-lifting algorithm. As translation rules are assumed to be not mutually defined, I can process the translation rules one by one. Suppose that the translation rule tr has the shape cn $\alpha_1 \cdot \cdot \cdot \alpha n \rightarrow_d$ e. I aim to get the evaluation rules of c_n.

The core idea of deriving evaluation rules has been shown in Sec. 2: expand the evaluation of compound expressions recursively according to the host evaluation rules until all the evaluations are not expandable. Formally, I use d for one-step static (symbolic) derivation, which accepts a judgment and returns a set of lists of judgments. Intuitively, it applies each possible rule for the input judgment, and each of them

produces a list of judgments. The definition of d is given in Fig. 7. For example:

$$d(\text{if } \alpha \text{ then } \beta \text{ else } false \Downarrow v) = \begin{cases} [\alpha \Downarrow_S true, \beta \Downarrow_S v] \\ [\alpha \Downarrow_S false, false \Downarrow false, v = false] \end{cases}$$

$$d(\alpha \Downarrow_S v) = \{[\alpha \Downarrow_S v]\} \quad (1)$$

$$d([x' \rightarrow e]\alpha \Downarrow_S v) = \{[[x' \rightarrow e]\alpha \Downarrow_S v]\} \quad (2)$$

$$d([x' \rightarrow e_1, \ldots]e_2 \Downarrow_S v) = d(e \Downarrow_S v) \{ex1 \text{ if } [x' \rightarrow e_1, \ldots]e_2 = e_{em}\} \quad (3)$$

$$d(c_h\ e_1 \cdots e_m \Downarrow_S v) = |\sigma(j_1) \cdots \sigma(j_n)| \quad r \in R(c_h) \quad (4)$$

$$d(f(e_1 \bullet\bullet\bullet e_n) \Rightarrow_S v) = \{[f(e_1 \bullet\bullet\bullet e_n) \Rightarrow_S v]\} \quad (5)$$

Figure 7: One-step Static Derivation

I can define d* to represent a recursive derivation until all the judgments are of the form (1), (2) and (5). Formally, d* can be defined in Fig. 8.

According to the above definition, given a translation rule $c_n\ \alpha_1 \cdots \alpha_n \rightarrow_d e$, the evaluation rules of cn are

$$\frac{j_1 \cdots j_n}{c_n\ \alpha_1 \cdots \alpha_n \Downarrow_S v}$$

$$[j_i \ldots j_m] \in d^*(e \Downarrow_S V)$$

I now discuss some examples to illustrate the algorithm.

Example 2 defines nand by not and and , assuming that the evaluation rules of not and and have been derived already. Example 3 defines or ′ by a let -binding and an if expression.

Example 2. Let us consider the translation rule α *nand* $\beta \rightarrow_d$ *not* (α *and* β).

$$d^*(\alpha \Downarrow_S v) = \{[\alpha \Downarrow_S v]\} \quad (6)$$

$$d^*([x' \rightarrow e]\alpha \Downarrow_S v) = \{[[x' \rightarrow e]\alpha \Downarrow_S v]\} \quad (7)$$

$$d^*([x' \rightarrow e_1, \ldots]e_2 \Downarrow_S v) = d^*(e \Downarrow_S v) \text{ if } [x' \rightarrow e_1, \ldots]e_2 = e \quad (8)$$

$$d^*(c\ e \bullet\bullet\bullet e \Downarrow v) = (J ++ \bullet\bullet\bullet ++ J \quad [j_1 \bullet\bullet\bullet j_n] \in d(c_h\ e_1 \bullet\bullet\bullet e_m \Downarrow_S v)) \quad (9)$$
$$J_1 \in d^*(j_1), \ldots j_n \in d^*(j_n)$$

$$d^*(f(e_1 \bullet\bullet\bullet e_n) \Rightarrow_S v) = \{[f(e_1 \bullet\bullet\bullet e_n) \Rightarrow_S v]\} \quad (10)$$

Figure 8: Recursive Static Derivation

This example showcases how d* works recursively. According to the definition, I need to compute d*(not (α and β)) first.

$$\frac{?}{not\ (\alpha\ and\ \beta) \Downarrow_S v}$$

By the definition of d* and evaluation rules of not, I have

$$d^*(not\ (\alpha\ and\ \beta) \Downarrow_S v) = \{[\alpha\ and\ \beta \Downarrow_S true, v = false],\ [\alpha \text{ and } \beta \Downarrow_S false, v = true]\}$$

$$\frac{\dfrac{?}{\alpha\ and\ \beta \Downarrow_S true}}{not\ (\alpha\ and\ \beta) \Downarrow_S false} \qquad \frac{\dfrac{?}{\alpha\ and\ \beta \Downarrow_S false}}{not\ (\alpha\ and\ \beta) \Downarrow_S true}$$

I focus on the derivation of the first case, omitting the second one. Recursively, I have

$$d^*(\alpha\ and\ \beta \Downarrow true) = \{[\alpha \Downarrow_S true, \beta \Downarrow_S true], [\alpha \Downarrow false, false = true]\}$$

$$\frac{\dfrac{\alpha \Downarrow_S true \quad \beta \Downarrow_S true}{\alpha\ and\ \beta \Downarrow_S true}}{not\ (\alpha\ and\ \beta) \Downarrow_S false} \qquad \frac{\dfrac{\alpha \Downarrow_S false \quad false = true}{\alpha\ and\ \beta \Downarrow_S false}}{not\ (\alpha\ and\ \beta) \Downarrow_S true}$$

Since the above premises all fit the form of (6), I can summarize the following conclusions:

$$d^*(not\ (\alpha\ and\ \beta)_S\ v) = \{[\alpha \Downarrow_S false, false = true, v = false], [\alpha \Downarrow_S true, \beta \Downarrow_S false, v = true], [\alpha \Downarrow_S false, v = true]\}$$

By removing premises that cannot be satisfied, I can obtain the following evaluation rules:

$$\frac{e_1 \Downarrow_S true \quad \beta \Downarrow_S true}{e_1\ nand\ e_2 \Downarrow_S false} \qquad \frac{e_1 \Downarrow_S true \quad \beta \Downarrow_S false}{e_1\ nand\ e_2 \Downarrow_S true} \qquad \frac{}{e1\ nand\ e2 \Downarrow_S true}$$

Example 3. Let us consider the translation rule α or′ β →d let x = α in if x then x else β.
This example demonstrates the role of hygiene in semantics lifting. Note that
I assume that the translation is hygienic, i.e., here x is an arbitrary fresh variable, and x cannot be used in α or β. The evaluation rules of or′ are derived similarly to Example 2 as follows.

$$d(let\ x = \alpha\ in\ if\ x\ then\ x\ else\ \beta \Downarrow_S v) = \{[\alpha \Downarrow_S v_1, [x' \rightarrow v_1](if\ x\ then\ x\ else\ \beta) \Downarrow_S v]\}$$

$$\frac{\alpha \Downarrow_S v_1 \quad [x' \rightarrow v_1](if\ x\ then\ x\ else\ \beta) \Downarrow_S}{let\ x = \alpha\ in\ if\ x\ then\ x\ else\ \beta \Downarrow_S v}$$

The first judgment is already in the form (1), so I only need to consider the second one, which contains a substitution. I apply the substitution first and then get the premises according to the evaluation rules of if . I have

$$\begin{aligned} & d([x \text{ '}\rightarrow v_1](\textit{if } x \textit{ then } x \textit{ else } \beta) \Downarrow_S v) \\ = & \; d(\textit{if } v_1 \textit{ then } v_1 \textit{ else } [x \text{ '}\rightarrow v_1]\beta \Downarrow_S v) \\ & \left(\begin{array}{l} [v_1 = \textit{true}, v = \textit{true}], \\ [v_1 = \textit{false}, [x \text{ '}\rightarrow v_1]\beta \Downarrow_S v] \end{array} \right) \end{aligned}$$

$$\frac{v_1 = \textit{true}}{[x \text{ '}\rightarrow v_1](\textit{if } x \textit{ then } x \textit{ else } \beta) \Downarrow_S \textit{true}} \qquad \frac{[x \text{ '}\rightarrow v_1]\beta \Downarrow_S v}{[x \text{ '}\rightarrow v_1](\textit{if } x \textit{ then } x \textit{ else } \beta) \Downarrow_S v}$$

Therefore, I obtain the set of premises of evaluation rules for or′:

$$d^*(\textit{let } x = \alpha \textit{ in } \textit{if } x \textit{ then } x \textit{ else } \beta \Downarrow_S v) = \left(\begin{array}{l} [\alpha \Downarrow_S \textit{true}, v = \textit{true}], \\ [\alpha \Downarrow \textit{false}, [x \text{ '}\rightarrow \textit{false}]\beta \Downarrow v, x \text{ is fresh}] \end{array} \right)$$

The generated rule is correct. But because there must be no x in β, I can further simplify it to obtain the following evaluation rules:

$$\frac{e_1 \Downarrow_S \textit{true}}{e_1 \text{ or}' e_2 \Downarrow_S \textit{true}} \qquad \frac{e_1 \Downarrow_S \textit{false} \quad e_2 \Downarrow_S v}{\text{or}' e_2 \Downarrow_S v} \qquad e_1$$

3.4. ASSUMPTIONS AND PROPERTIES

In this section, I present the assumptions being used in my framework. These assumptions are sufficient because they can prove the correctness and abstraction property of semantics lifting. Review that correctness is to show the relationship between the evaluation result of e in DSL and the evaluation result of D(e) in the host language, and abstraction guarantees that only language constructs of values and DSL can be mentioned in the derivation tree of DSL program.

Correctness is stated the same as Goal 1 given in Sec. 2.

As for the abstraction property, I modify the statement given in Goal 2 slightly. I assume a set S of language constructs is allowed to appear in the evaluation derivation of the DSL. Intuitively, S represents language constructs that are understandable to the DSL user. Then, all the language constructs of S should be included in DSL. By default, S is composed of values and DSL constructs. Given the set of language constructs S, in the semantics lifting, if the judgment has the form e $\Downarrow_S$ v and all the language constructs in e are elements of S, I can let d(e $\Downarrow_S$ v) be {[e $\Downarrow_S$ v]} for such premise will not break the abstraction.

In turn, I formalize the assumptions for meta-functions, host language, and translation rules and then prove their correctness and abstraction their correctness and abstraction.

Meta-functions The two assumptions about meta-functions are, respectively, for ensuring correctness and abstraction. Starting with correctness, I have explained that translation and meta-functions invocation should be commutable in Sec. 2.2. In a language that includes side effects, this assumption can be formalized as follows:

Assumption 1 For each meta-function f, f satisfies:

It requires that the meta-functions be semantically driven and not influenced by the syntactic structure of the arguments. All the meta-functions I use in Func satisfy this property.

Another assumption is related to the set S of language constructs; that is, I can use local abstraction to justify global abstraction. If I invoke a meta-function

with expressions of DSL, the output should also be an expression of DSL. In other words, the meta-functions cannot generate any new language construct not in S.

Assumption 2 Given a set of language constructs S and expressions $e_1 \cdots en$, the language constructs of the return value $f(e_1 \cdots en)$ are either from some ei, or from set S.⊔⊓

Host Language In semantics lifting, the evaluation of compound expressions of the host language is expanded according to their evaluation rules. Therefore, I require that no host language constructs appear in the premises of the host language's evaluation rules. More strictly, I require that the permitted language constructs be elements of the set S that can appear in the DSL.

Because DSL constructs have yet to be introduced, I define a subset of S as S_h, which mainly consists of constructs for values.

Assumption 3 Given a set of language constructs Sh, all the language constructs used in the premises of evaluation rules of the host language should be in

S_h.

From another perspective, any language construct in the host language used in the premises of the evaluation rules must appear in S. For example, fix in Func uses fix itself in the evaluation rule, so the language construct fix must be an element of S_h. In Func, I define S_h as constructs of values and fixes. By design, it makes sense to include a language construct in S_h only if it is primitive and can be assumed to be understood by DSL users, like the fixed-point combinator. Extending S_h to encompass all language constructs would make the above assumption and abstractness property trivial; thus, I should avoid the problem of such Sh being too large and confusing due to a not well-designed host-language definition. In some cases, I can avoid this situation by introducing more meta-functions. For example, I can introduce the meta-function iteration

to expand the fixed-point combinator several times (i.e., the maximum number allowed of recursion). Then,the evaluation rule of the fixed-point combinator can be expressed as follows:

$$\frac{e \Downarrow_S \lambda x.\ e' \quad iteration(\lambda x.\ e') \Rightarrow_S v}{fix\ e \Downarrow_S v}$$

Translation Rules: My assumption about translation rules is also intended to ensure abstraction. In Sec. 2.3, I have shown that lambda abstraction containing host language constructs breaks abstraction in semantics lifting because of delayed evaluation. More generally, in the RHS of translation rules, any host-language construct that appears in the non-abstraction component of a host- language construct may lead

to abstraction leakage. Therefore, I propose the following assumption:

Assumption 4 Given a set of language constructs S, if a host language con- structs c is used in the RHS of definition, then the language constructs used in
the non-abstract components of c must be elements of S.

This is a sufficient but not necessary condition to ensure abstraction. For example, in the following translation rule, I use the host-language construct if in the non-abstract component of the lambda abstraction.

$$not' \ \alpha \rightarrow_d ((\lambda x.\ if\ x\ then\ false\ else\ true)\ \alpha)_V$$

However, since the lambda abstraction is used with the application, which is essentially equivalent to let , the body's evaluation is derived statically and, there- fore, does not break the abstraction. Similarly, abstraction leakage may not occur

for a translation rule that contains a call-by-name application. For example, in the translation rule

$$not'' \ \alpha \rightarrow_d ((\lambda x.\ x)\ (if\ \alpha\ then\ false\ else\ true))_N,$$

the if expression is used in the non-abstract component of call-by-name application. However, since the applied abstraction is an expression without any meta-variable, all substitutions can be determined statically.

For those translation rules that do not satisfy this assumption, I can generalize the approach of lambda lifting introduced in Sec. 2.3. The sub-expressions containing host-language constructs used in the non-abstract components of the translation rules are defined as new translation rules, and then the original sub-expressions are replaced by language constructs defined by these new translation rules. Re-processing translation rules ensures that the newly generated translation rules satisfy the above assumption. I do not discuss the transformation in detail here.

Main Theorem Given these assumptions, my semantics-lifting algorithm satisfies the following correctness and abstraction properties.

Theorem 5 (Correctness). If assumption 1 is satisfied, then

Soundness: If $e \Downarrow_S v$ *holds in DSL, then* $D(e) \Downarrow_S D(v)$ *holds in host language;* (11)

Completeness: If $D(e) \Downarrow_S v'$ *holds in host language, then exists* v, *s.t.* $D(v) = v'$ (12)

and $e \Downarrow_S v$. (13)

I prove the two parts by induction on the derivation of $e \Downarrow_S v$ and $D(e) \Downarrow_S v'$, respectively.

Theorem 6 (Abstraction): Given a set of language constructs S, if assumptions 2, 3 and 4 are satisfied,

then globally, all the language constructs used in the derivation tree of a DSL program evaluation should be in S, and locally, all the language constructs used in the derived evaluation rules are elements of S.

Let $\frac{j1 \cdots jn}{j}$ be a derived evaluation rule, I aim to show that $j_1 \ldots i_1$ do not mention language constructs of the host language. According to the termination condition of d^*, the derived judgments all have the form of (6), (7) and (10). So I only need to ensure that there are no such things in the substitution and arguments of meta-functions which break abstraction. If some meta-function call f uses an expression with host language construct c_h in a judgment, there are two possibilities. Consider the step of producing the meta- function call during the evaluation-rule derivation. If the applied host rule is like

$$\frac{\ldots \quad f(c_h \ldots) \Rightarrow_S v \ldots}{c \ldots \Downarrow_S v'}$$

then such c_h must be in S according to assumption

3. If the applied rule has shape $\frac{\ldots \quad f(e) \Rightarrow_S v \ldots}{c\ e \ldots \Downarrow_S \quad v'}$, and e is substituted as

an expression with ch in derivation, since e is a non-abstract component of c e ...,
ch must be an element of S according to assumption 4. Therefore, abstraction is guaranteed.

4. IMPLEMENTATION

I have implemented a prototype of my framework in Haskell named Osazone. To describe the semantics of a language, I design a meta-language. Osazone allows users to define host languages using this meta-language and design DSLs using some extensions and translation rules. I propose a general workflow to implement DSLs in Osazone. A developer should start by taking a language as the host language, introducing vocabularies by primitive extensions, and defining new language features by monad extensions. The developer can then define the DSL constructs by translation rules from the extended language.

Primitive extensions involve adding new language constructs with corresponding rules directly to the host language. When the expressiveness of the host language is insufficient, e.g., some data types are absent, it would be impossible to define a DSL by translation rules. In this case, primitive extensions can introduce new data types and operations. Also, monad extension is a technique for incorporating side effects in computation without changing the original semantics definition. For instance, monad extensions can extend the language Func from typed lambda calculus. In Sec. 5, I will provide examples of language engineering using primitive extensions, monad extensions, and translation rules.

Meta-language In big-step operational semantics, some language constructs have multiple evaluation rules, which leads to several branches when applying the evaluation rules in a derivation. Osazone introduces a meta-language for defining language semantics to overcome this issue, where each language construct has only one evaluation rule. My meta-language is based on Skeleton [1]. Using my meta-language, I can define the evaluation of if expressions using a single rule as follows:

if e_1 then e_2 else e_3 e_1 ; true ▷ e_2 false ▷ e_3.

Here, e1 represents the evaluation of sub-expression e1. The result of this evaluation is used to select a specific branch through pattern matching (after the colon), after which subsequent computations are processed (after the triangle). If e_1 evaluates to true, e_2 will be evaluated, and if e_1 evaluates to false, e_3 will be evaluated.

In my approach, all evaluation rules can be expressed through recursive com—computations, meta-operations, and branches. This method simplifies the semantics definition, improves the derivation process's clarity and coherence, and enhances my framework's overall usability.

Define languages in Osazone In Osazone, a language's definition includes evaluation and typing rules. When introducing a monad, the developer must also define the program entrance. For example, the entrance of evaluation in Func is runState • empty, where runState takes the initial state and returns the computation value and final state. Osazone then automatically derives an interpreter for this language by generating Haskell code for the interpreter. A program can be evaluated and typed using the interpreter.

To define a DSL, developers need to choose a host language and provide a file for primitive extensions (first step), a file for monad extensions (second step), and a file for translation rules (third step).

The definitions of primitive extensions are directly added to the language. In the monad extension, all the meta-functions are lifted through a monad transformer to obtain the definition of the extended language. Then, the translation rules are analyzed one by one. My implementation derives both the evaluation and type rules for each translation rule. These rules are added to the extended language. Users can choose certain language constructs from the mixed language as the constructs of the DSL, and Osazone will automatically select a minimal closed subset containing these language constructs as the output language. Finally, I got the definition of DSL, and an interpreter of DSL was obtained for free.

5. CASE STUDIES

In this section, I use functional and imperative languages as host languages to illustrate the flexibility and practicality of Osazone. In the first example, I aim to show how a functional language is progressively extended with richer features using primitive and monad extensions and specialized to DSLs using translation rules. In the second example, I aim to demonstrate the generality of my approach, presenting its applicability to imperative host languages.

5.1 DSLS ON FUNCTIONAL HOST LANGUAGES

The languages discussed in this section and their relationships are shown in Fig. 9 where horizontal arrows represent primitive extensions and monad extensions, and vertical arrows stand for DSL definition by translation rules.

I have shown the syntax and semantics of Stlc, Bool (in Sec. 2), and Func(in Sec. 3). I start with Stlc, a language with lambda calculus, integers, and Booleans. From Stlc, I define Bool using translation rules. With primitive extensions, I can extend Stlc to Stlcex by adding language constructs. Then, I introduce a state monad to support references, getting Func. I/O can also be supported in Func by adding a new

language construct print via monad extension. For brevity, I skipped the discussion about this extension.
Language Adder Consider that developers would like to develop a DSL based on the Bool language, named Adder, which supports half -adder and

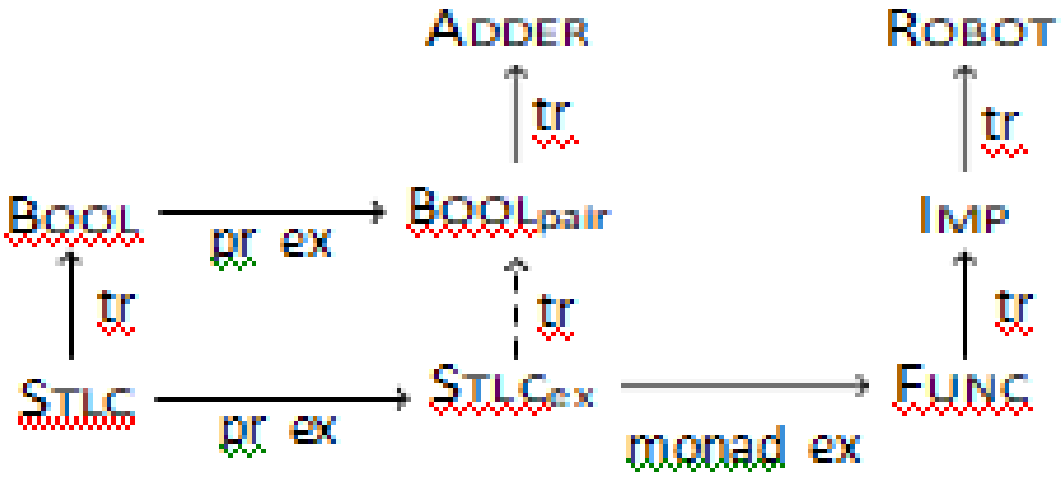

Figure 9: Languages Extended and Specialized from STLC

Full - Adderr to simulate a digital circuit. For example, in the language, I have

half Adderr true false ⇓ *(true, false).*

However, translation rules are not easily implemented directly because half-adder and full-adder will get the pair of sum and carry as the results. Pairs are now required to build compound data structures. Primitive extensions are used to support new data types in the host language. In this example, developers add pair and projection to the language and specify each newly-defined language construct's evaluation (and typing) rules. Then, developers get a new language, Boolpair, and support pairs and projection. With the following translation rules, I can implement the DSL Adder that simulates half-adders and full-adders.

$$half\text{-}adder\ e_1\ e_2 \rightarrow_d (e_1\ xor\ e_2, e_1\ and\ e_2)$$
$$full\text{-}adder\ e_1\ e_2\ e_3 \rightarrow_d (e_1\ xor\ e_2\ xor\ e_3, (e_1\ xor\ e_2)\ or\ (e_3\ and\ (e_1\ xor\ e_2)))$$

The evaluation rules of *half -adder* , for example, are derived as follows:

$$\frac{e_1 \Downarrow true \quad e_2 \Downarrow true}{half\text{-}adder\ e_1\ e_2 \Downarrow (false, true)} \qquad \frac{e_1 \Downarrow true \quad e_2 \Downarrow false}{half\text{-}adder\ e_1\ e_2 \Downarrow (true, false)}$$

$$\frac{e_1 \Downarrow false \quad e_2 \Downarrow true}{half\text{-}adder\ e_1\ e_2 \Downarrow (true, false)} \qquad \frac{e_1 \Downarrow false \quad e_2 \Downarrow false}{half\text{-}adder\ e_1\ e_2 \Downarrow (false, false)}$$

Language Imp Since I added a store to Func, I can now express programs with side effects. Taking Func as the host language, I want to implement a reference-based imperative language Imp. In Func, the declarations and initial values of variables are necessary.
For instance, a legal Func program is shown as follows:

$$let\ \mathrm{x} = ref\ 1\ in\ \mathrm{x} := !\mathrm{x} + 1;\ !\mathrm{x}$$

Ideally, I want to write the corresponding DSL program in a more traditional IMP-like syntax:

$$\mathrm{x} := 1; \mathrm{x} := \mathrm{x} + 1.$$

I discuss why it is impossible to make the syntax of Imp exactly the same as traditional IMP-like language. Moreover, I define some other common language constructs for Imp.
The declaration in the Imp language is a let-binding in Func, where what is assigned must reference some expression. In order to simulate the syntax of declaration in common imperative languages, I can define translation rule var as a syntactic sugar:

$$var\ \mathrm{x} = e_1; e_2 \rightarrow_d \mathrm{let\ x = ref}\ e_1\ \mathrm{in}\ e_2$$

This semicolon is used in the declaration syntax to ensure that x is valid in e_2.
At first glance, this is a perfect translation. However, in a usual assignment statement, the left-hand side is a variable that denotes a location in the store (known as the L-value in the C-world). However, any variable on the right side refers to its stored value (the R-value in the C-world). I cannot distinguish these different occurrences by using translation rules.
As a result, explicit dereferences are required in Imp programs.

$$var\ x = e_1; e_2 \rightarrow_d let\ x = ref\ e_1\ in\ e_2$$
$$when\ e_1\ e_2 \rightarrow_d if\ e_1\ e_2\ ()$$
$$e_1 \mathrel{+}= e_2 \rightarrow_d e_1 := !e_1 + e_2$$
$$while\ e_1\ do\ e_2\ end \rightarrow_d ((fix\ (\lambda f.\ \lambda x.\ \lambda y.\ if\ x\ then\ (y; (f\ x\ y)_N)\ else\ ()))\ e_1\ e_2)_N$$

Figure 10: Translation Rules for IMP

Other language constructs in Imp are defined in Fig. 10. I define them using the fixed construct because recursively defined translation rules are not allowed. Note that the translation rule does not satisfy assumption four because if and sequencing are used in the non-abstract component of lambda abstraction. Using lambda lifting, I get the following two translation rules:

$$\mathrm{body\ e}\ e_1\ e_2 \rightarrow_d \mathrm{if}\ e_1\ \mathrm{then}\ e_2; (\mathrm{e}\ e_1\ e_2)_N\ \mathrm{else}\ ()$$
$$\mathrm{while}\ e_1\ \mathrm{do}\ e_2\ \mathrm{end} \rightarrow_d ((\mathrm{fix}\ (\lambda f.\ \lambda x.\ \lambda y.\ \mathrm{body\ f\ x\ y}))\ e_1\ e_2)_N$$

Those derived evaluation rules satisfy abstraction property with body, fix ∈ S, shown in Fig. 11. Note

that I do not apply the rule of body recursively since all the constructs in the expression are elements of S. As an instance of Imp, the following program is implemented to calculate the sum of 1 to 10.

$$var\ \mathrm{n} = 1; var\ \mathrm{sum} = 0; while\ \mathrm{n} < 11\ do\ \mathrm{sum} \mathrel{+}= !\mathrm{n}; \mathrm{n} \mathrel{+}= 1;\ !\mathrm{sum}\ end$$

$$\frac{e_1 \Downarrow_S true \quad e_2 \Downarrow_S () \quad e \Downarrow_S \lambda y.\ e' \quad [x' \rightarrow e_1, y' \rightarrow e_2]e' \Downarrow_S v}{body\ e\ e_1\ e_2 \Downarrow_S v}$$

$$\frac{e_1 \Downarrow_S false}{body\ e\ e_1\ e_2 \Downarrow_S ()} \qquad \frac{body\ (fix\ (\lambda f.\ \lambda x.\ \lambda y.\ body\ f\ x\ y))\ e_1\ e_2 \Downarrow_S v}{while\ e_1\ do\ e_2\ end \Downarrow_S v}$$

Figure 11: Derived Evaluation Rule of *body* and *while*

Language Robot Based on Imp, I implemented a DSL named Robot. The language assumes that a robot is located at some initial coordinate, and users can control its movement or print out its position with commands.

A sample program of Robot is:

$$\text{robot 5 5 } \{up, up, right, where \text{ am I}, \text{left}, \text{whereAm}\}$$

Where robot 5 5 {...} declares a robot with an initial coordinate of (5, 5), and the braces contain a series of commands to be executed on the Robot, split by a comma. Commands up, right, and left control the Robot's movement, and commands are where AmI will print the current position.

As a natural idea, I might record the Robot's current position via the global variables x and y. Then, each command reads and manipulates global variables; the comma is defined as sequencing. These language constructs are defined as follows:

$$\begin{aligned} robot\ \mathrm{e}_1\ \mathrm{e}_2\ \{\mathrm{e}\} &\rightarrow_d let\ \mathrm{x} = ref\ \mathrm{e}_1, \mathrm{y} = ref\ \mathrm{e}_2\ in\ \mathrm{e} \qquad \mathrm{e}_1, \\ \mathrm{e}_2 &\rightarrow_d \mathrm{e}_1; \mathrm{e}_2 \\ left &\rightarrow_d \mathrm{x} := !\mathrm{x} - 1 \\ whereAmI &\rightarrow_d print\ !\mathrm{x}; print\ !\mathrm{y} \end{aligned}$$

Where x and y are literal identifiers. However, under such definitions, the requirement of closed translation rules is not satisfied. Because variables without local bindings cannot be used directly in translation rules, passing the variable's value as an argument to the language constructs or as an argument to a lambda abstraction is necessary. Hence, left should be expressed as:

$$\text{left} \rightarrow_d \lambda pos.\ (\text{fst pos}) \mathrel{+}= (-1); \text{pos},$$

where pos has type (Ref int) × (Ref int). Note that the left returns pos to keep passing on the "global" state. Then, the comma operator composes these functions. Some other selected translation rules for Robot are given in Fig. 12.

Do not forget that because they are defined via lambda abstraction, lambda lifting is also necessary.

$$robot\ e_1\ e_2\ \{e\} \rightarrow_d e\ (ref\ e_1, ref\ e_2) \qquad e_1,$$
$$e_2 \rightarrow_d \lambda pos.\ e_2\ (e_1\ pos)$$
$$left \rightarrow_d \lambda pos.\ (fst\ pos) \mathrel{+}= (-1); pos$$
$$whereAmI \rightarrow_d \lambda pos.\ print\ !(fst\ pos); print\ !(snd\ pos); pos$$

Figure 12: Translation Rules for ROBOT

5.2. DSLS ON IMPERATIVE HOST LANGUAGES

Consider that I would like to implement a DSL to simulate finite-state machines called Fsm. I simplify the setting by using integers to represent states and symbols. In this section, I will implement a language in which the programs look like this:

```
automata :
input 5 {r 0; r 1; r 0; r 1; r 0};
 exec        {0 0 2; 0 1 1; 1 0 2; 1 1 1; 2 0 2; 2 1 3; 3 0 1; 3 1 0; };
 end;
```

This machine has four states and 2 symbols in the alphabet. The initial state is 0 (implicitly), and transition rules are defined after the keyword exec. I use five symbols as input for this automaton, declared by input, and the program's evaluation should tell us the final state.

I chose a simplified version of CMINOR [6] as the host language to implement this. In my problem, I only need to define the primary function, ignoring language features such as the function call stack. In addition, types and memory models are not my focus. Therefore, I assume that only integers are stored in the heap, meaning that the offset of a pointer is fixed.

The DSL program will be translated into the execution of the primary function. In CMINOR, a function is defined by a list of parameters (null for the primary function), a list of local variables, and a body statement. Defining variables in the body is not allowed. Therefore, I need to identify the local variables used in Fsm to provide the translation rules for the DSL.

The translation rule of automata is defined as follows:

$$\text{automata : } stmt \rightarrow_d \text{main() \{ var } n, syms, i, st; stmt \}$$

Where stmt is the body of the primary function; n is the number of input symbols; syms is the pointer pointing to the first cell of input symbols; st is the current state; and i is an auxiliary variable for iteration. The language constructs input, and r records the input symbols. The input construct initializes local variables n and i and then allocates the requested size. After that, r is used to store the inputs. The translation rules of input and r are defined as:

$$\text{input } expr\ stmt \rightarrow_d n := expr;\ := 0;\ syms := \text{malloc}(n); \qquad stmt \quad r$$
$$expr \rightarrow_d [syms + i] := expr;\ := +\ 1;$$

In RHS, statements include assignment to local variables (with shape ident:= expr), memory stores (with shape [expr]:= expr), and sequencing. Expressions include reading local variables, constants, binary

operations, and function calls. In C minor, malloc is seen as an external function, while I use it as a primitive operation, which takes the size as an argument and returns the address of a newly allocated memory block with the requested size.

The FSM processes input through language construct exec. It repeatedly changes the state based on the current input until all input characters are consumed. The body of the exec consists of a list of transfer rules, which will be translated into an if statement. If one of the rules is matched, there should be a continuation. If no rule can be matched, the state will be set as −1, and the loop should exit through the break. In C minor, only infinite looping is introduced. Moreover, break and continue are implemented with blocks and exit statements. I use exit n to leave (n + 1) enclosing blocks. For example, in C minor, while e s is written as

block { loop { **if** !e then exit 0; block { s } }}.

Note that the braces here are used to enclose a group of statements connected by sequence, while a block is used with the block language construct. In s, if there are no nested blocks, continue can be written as exit zero, and break can be written as exit 1. Based on the above discussion, I can define the following translation rules:

exec stmt $\rightarrow_d$:= 0; block { loop { **if** (== n) exit 0;
block { stmt ; st := -1; exit 1; }}}

$\text{expr}_1 \longrightarrow \text{expr}_2 \rightarrow_d$ **if** (st == expr_1 && load (syms + i) == expr) {
st := expr_2; := + 1; exit 0; }

Finally, end is used to print the final state.

end $\rightarrow_d$ print(st); return 0;

Print is an external function in C minor, and the return statement finishes the function's execution.

Next, I will discuss the semantics of C minor. I use a monad to describe the changes in the global heap and local environment. Due to diverging computers caused by incorrect storage and retrieval and the potential use of undefined variables, I also need to introduce the maybe monad. I combine the above two monads with the I/O monad using a monad transformer and denote the combined monad as m.

The evaluation of an expression expr $\Downarrow_m$ val does not cause changes in the environment or memory, but it may read the value of variables or pointed-to values of pointers from the state. Statements evaluate to outcomes stmt $\Downarrow_m$ out, indicating how afterward execute. There are three types of outcomes: norm (for regular, continuing in sequence), ex n (for exit, terminating the n + 1 enclosing blocks), and ret val (for return). Some evaluation rules of expressions and statements are shown in Fig. 13. One thing to note is the loop construct. By using later, I transform the recursion in the evaluation rule of the loop construct into a meta-function call to satisfy assumption 3. The meta-function later(stmt) returns the evaluation result of stmt. Furthermore, because stmt is used as an argument in the meta-function, it is a non-abstract component of loop stmt. All these evaluation rules and meta-functions obey the assumptions.

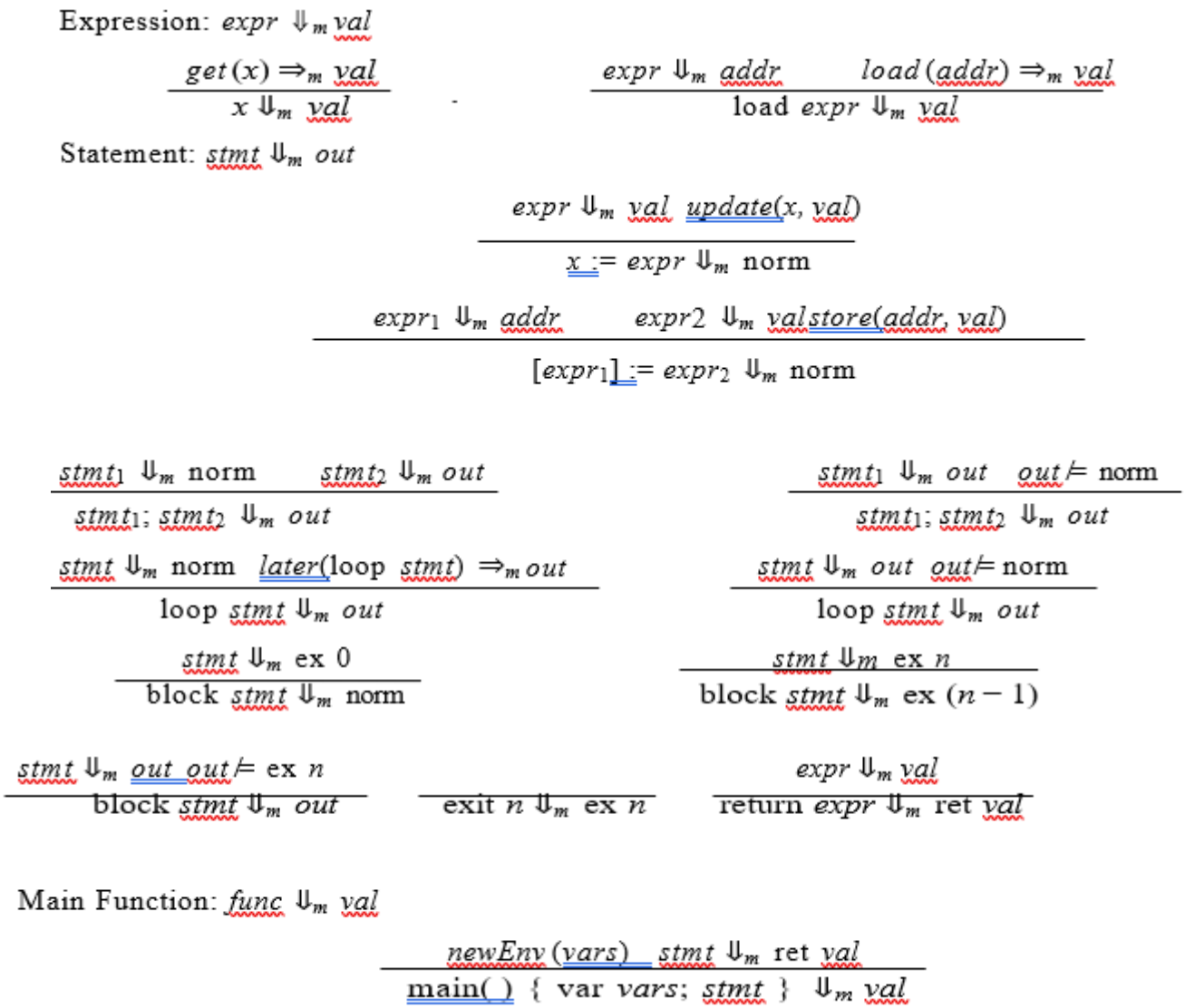

Figure 13: Selected Evaluation Rules of C MINOR

I must show that the translation rules satisfy my closed translation requirement. In Cminor, all variable assignments and reads are done through meta-functions, which means that undefined variables will fail at the run-time of the meta-function. In the derived evaluation rules, variables can be read dynamically. Hence, performing local binding for variables is unnecessary in the translation rules to achieve correct semantics lifting. However, hygiene can- not be guaranteed as a result. To satisfy assumption 4, exec needs to be split into two translation rules:

step stmt $\rightarrow_d$ if (== n) exit 0; block { stmt ; st := -1; exit 1; }

exec stmt $\rightarrow_d$:= 0; block { loop { step stmt }}

Their semantics can be statically derived and will not be elaborated here.

4 CONCLUSION

This paper proposes a systematic framework to lift semantics for domain-specific languages. I present reasonable assumptions to ensure that systematic lifting maintains correctness and satisfies abstraction. These assumptions are related to the meta-functions, host-language semantics, and translation rules. I have proved the correctness and abstraction of the semantics-lifting algorithm under such assumptions.

Following this idea, I implemented the tool Osazone, which is shown to be flexible, effective, and practical for developing DSLs. Osazone provides a meta-language to describe the evaluation rules of host languages. Based on a host language, users can extend the host language to support new vocabularies and language features and specify the DSL constructs by translation rules. As case studies, I take functional and imperative languages as host languages and implement DSLs based on these two languages through translation rules. I show how these languages are implemented and illustrate that I used the meta-functions, host languages, and translation rules to meet my assumptions. Eventually, I obtained correct, abstract semantics for DSLs.

REFERENCES

1. M., Gardner, P., Jensen, T., Schmitt, A.: Skeletal semantics and their inter- pretations. Proc. ACM Program. Lang. 3(POPL) (jan 2019). https://doi.org/ 10.1145/3290357, https://doi.org/10.1145/3290357
2. Culpepper, R., Felleisen, M., Flatt, M., Krishnamurthi, S.: From macros to dsls: The evolution of racket. In: Summit on Advances in Programming Languages (2019)
3. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S.: A programmable programming language. Commun. ACM 61(3), 62–71 (feb 2018). https://doi.org/10.1145/3127323, https://doi.org/ 10.1145/3127323
4. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Proc. of a Conference on Functional Programming Languages and Computer Ar- chitecture. p. 190–203. Springer-Verlag, Berlin, Heidelberg (1985)
5. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expan- sion. In: Proceedings of the 1986 ACM Conference on LISP and Functional Pro- gramming. p. 151–161. LFP '86, Association for Computing Machinery, New York, NY, USA (1986). https://doi.org/10.1145/319838.319859, https://doi.org/ 10.1145/319838.319859
6. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (jul 2009). https://doi.org/10.1145/1538788.1538814, https://doi. org/10.1145/1538788.1538814
7. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 333–343. POPL '95, Association for Computing Ma- chinery, New York, NY, USA (1995). https://doi.org/10.1145/199448.199528, https://doi.org/10.1145/199448.199528
8. Moggi, E.: Computational lambda-calculus and monads. In: [1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science. pp. 14–23 (1989). https://doi.org/10.1109/LICS.1989.39155
9. Moggi, E.: Notions of computation and monads. Information and Compu- tation 93(1), 55–92 (1991). https://doi.org/https://doi.org/10.1016/
10. 0890-5401(91)90052-4, https://www.sciencedirect.com/science/article/ pii/0890540191900524, selections from 1989 IEEE Symposium on Logic in Computer Science
11. Pombrio, J., Krishnamurthi, S.: Resugaring: Lifting evaluation sequences through syntactic sugar. In: Proceedings of the 35th ACM SIGPLAN Conference on Pro- gramming Language Design and Implementation. p. 361–371. PLDI '14, Associ- ation for Computing Machinery, New York, NY, USA (2014). https://doi.org/ 10.1145/2594291.2594319, https://doi.org/10.1145/2594291.2594319
12. Pombrio, J., Krishnamurthi, S.: Inferring type rules for syntactic sugar. SIGPLAN Not. 53(4), 812–

825 (jun 2018). https://doi.org/10.1145/3296979.3192398,
13. https://doi.org/10.1145/3296979.3192398
14. Pombrio, J., Krishnamurthi, S., Wand, M.: Inferring scope through syntactic sugar. Proc. ACM Program. Lang. 1(ICFP) (aug 2017). https://doi.org/10.1145/ 3110288, https://doi.org/10.1145/3110288
15. Spolsky, J.: The law of leaky abstractions. https://www.joelonsoftware.com/ 2002/11/11/the-law-of-leaky-abstractions/ (2002)
16. Ward, M.P.: Language-oriented programming. Softw. Concepts Tools 15, 147–161 (1994)