Dynamic Language Updating

Albert Shaqiri
Id. Number: R10589

Scuola di Dottorato in Informatica
PhD in Computer Science

PhD School Headmaster: **Prof. Paolo Boldi**

Advisor: **Prof. Walter Cazzola**

UNIVERSITÀ DEGLI STUDI DI MILANO
Computer Science Department
ADAPT-Lab

Ciclo XXIX
INF/01 Informatica
Academic Year 2016–2017

Contents

1. Introduction **1**

2. Background **7**
2.1. Programming Languages and Interpreters 7
2.1.1. Defining the Syntax: Context-free Grammars 7
2.1.2. Defining the Semantics: Syntax-Directed Definitions 9
2.1.3. Interpretation 10
2.1.4. Development of Programming Language Interpreters 10
2.2. Modular Development of Programming Languages 11
2.2.1. Language Decomposition 12
2.2.2. Language Composition 12
2.2.3. Composition Soundness 14
2.2.4. Neverlang 15
2.3. The Structure of Programming Language Interpreters 24

3. Open Interpreters **27**
3.1. Implications and Requirements for Open Interpreters 28
3.2. A Model for Open Interpreters 29
3.2.1. Reflection on Language Specification 29
3.2.2. Reflection on Language Feature Instances 31
3.2.3. Reflection on Non-Grammatical Language Feature Instances . . . 33
3.3. Intercession Operations 33
3.4. Semantics Adaptation Implications 36

4. Open Interpreters in Neverlang **39**
4.1. Definition Mapping 39
4.2. The Architecture 40
4.3. Reflection on Open Interpreters 41
4.3.1. Reflection on Language Specification and Non-grammatical Components 42
4.3.2. Reflection on Linguistic Component Occurrences 43
4.3.3. Reflection API 47
4.4. μDA: a Platform DSL for Open Interpreters 49
4.5. Microlanguages 54
4.6. Discussion 56

5. Applicability of Open Interpreters **59**
5.1. Backward Compatibility 59
5.2. Dynamic Software Updating 63
5.3. Context-Aware Variability 68
5.3.1. Accessibility 68

5.3.2. Resource Usage Optimization 71
5.4. Interpreter Optimization 80
5.5. Aspect-Oriented Programming 83
5.6. Debugging 84
5.7. Security 86
5.8. Discussion 87

6. Related Work **89**
6.1. Language Extensions 89
6.2. Metaobject Protocols 90
6.3. Runtime Software and Interpreter Adaptation 92
6.4. Interpreter Composition 93
6.5. Quantification 94

7. Conclusions **97**

A. Composition Soundness **99**
A.1. Syntax Formalization 99
A.2. Operational Semantics 101
A.3. Type System 104
A.4. Type Inference 107

1 Introduction

Software systems are subject to continuous evolution which is motivated by unforeseen requirements, evolution of the surrounding context, maintenance and the desire to keep the pace with the never-sleeping competitors. As software systems, programming languages are equally subject to evolution more or less for the same reasons. However, as programming languages underly other systems their evolution can have a tremendous impact on the existing software. This is one of the reasons why even a small language evolution is meticulously considered and tested before it is applied. In general, language engineers struggle to maintain the language backward compatible. However, experience shows that, sooner or later, programming languages need to evolve at the cost of loosing the full backward compatibility[1]. Furthermore, many language implementations are monolithic and their evolution could not be otherwise; they come in the form of monolithic updates with no possibility to partially apply the evolution of those parts of the language implementation that would not break existing applications [121]. Sometimes, language developers provide translation tools to encourage and assist application developers in the migration process. For example, Python provides the `2to3` translation tool which applies a series of fixers to transform the Python 2.x source code into valid Python 3 code. It even allows developers to define their own fixers to guide the translation process. Despite the provided migration tools and the fact that it has been around since December 2008, Python 3 did not see a mass adoption for various reasons. The `2to3` tool does not perform well for anything beyond small-scale programs. Converting hundreds of thousands of lines of Python 2 code to make it compliant with Python 3 costs a lot of engineering time, often without delivering any real benefit to the business. The latter is aggravated by the priority mismatch between various stakeholders. The management is reluctant to fund something that does not bring evident economic benefits. Furthermore, if existing software is not going to be in service past the end-of-life of the current language being used, then there is no point in putting the tremendous effort to migrate to the updated language. Consequently, application developers often prefer to stick with the old version of the language, while a minority adopts the new language and rolls up the sleeves to rewrite applications in order to maintain their original behavior. Whatever the decision they make, backward incompatibility of programming languages is a serious obstacle in software development.

[1] E.g., see `https://web.archive.org/web/20170625204029/http://www.oracle.com/technetwork/java/javase/compatibility-417013.html`

For the reasons discussed above, language developers prefer to adopt the "slow-but-sure" strategy. However, the slowness of evolution is not always motivated by the desire to reduce the negative impact on the existing software systems, but is rather due to the complexity of the language implementation itself. A possible evolution should positively contribute to (or at least maintain) language consistency, conciseness, reliability and predictability. In general, the success of the evolution heavily depends on whether the software development was guided by software design principles that contribute to clean, understandable and maintainable source code. Concepts like separation of concern, single responsibility principle, "don't repeat yourself", etc., are highly appreciated principles that enabled the extraordinary growth of complex software systems we witnessed in the last decades. The more these principles are respected during software development the easier it will be to keep the pace with the inevitable evolution of the application domain and context. One of the design principles that contribute the most to the evolution is information hiding. According to this principle, a software component should disclose its services through an interface, but hide its implementation details that are likely to change in the future [95, 94]. Components interact with each other through interfaces and no component should rely on the internal implementation of another component. This helps preserve the interoperability between components, their reusability and portability and greatly contributes to the maintenance of the system in its long-term evolution. Information hiding is typically implemented through the encapsulation mechanism in the form of modules[2]. As module interoperability depends solely on their interfaces, it becomes easy for one to switch modules while preserving the integrity of the whole application. This is especially useful when evolving systems that provide services which cannot be shut down. Despite these facts, implementations of programming languages often overlook many design principles, especially those related to modularity [121]. As a result, many programming language interpreters are built as monolithic software systems which are hard to evolve. The evolution becomes even harder when the language design itself is poor and the language definition is inconsistent.

The situation is even worse if a language should evolve while its interpreter is running a critical application that cannot be shutdown. Examples are applications in devices that regulate some vital human body activity, air traffic control systems, nuclear power plant monitors, etc. Updating the interpreter might be necessary if it is found to have a security flaw or a bug. However, the interruption of system services might lead to a non-negligible economic loss, an ecological disaster or even to tragic consequences like death. On the other hand, the same aftereffects can occur if the security flaw or a bug is not promptly removed. As discussed above, modularization, backed by information hiding, is the key to smoother evolution as it allows one to update a system by replacing its modules. However, experience shows that modularity is rarely taken in consideration in programming language implementations. Therefore,

[2]Depending on the modularization granularity, encapsulation may concern concepts like methods, classes, modules, etc. In this context, we use the term "module" in a broader sense to refer to a modularization unit that interacts with other units through an interface.

most interpreters are not crafted for being dynamically updated.

The ability of a software system to adapt to the context is an increasingly appreciated feature, especially in the area of mobile and ubiquitous (pervasive) computing. Consequently, a lot was done to assist developers in designing and creating context-aware applications[3]. However, despite the fact that programming language interpreters are software systems themselves, they are rarely considered as candidates for adaptation to the surrounding execution context. Some programming languages come with reflection support that allows one to modify the behavior of the underlying interpreter. However, such reflection support, if present at all, is often limited to specific linguistic features. Even if reflection is richly supported, it is language-specific, therefore usable only by applications written in that language. Interpreters lack a general mechanism that would allow one to modify any linguistic feature and/or component of a language interpreter.

Software systems often evolve in response to new requirements and the domain evolution. Typically, this requires that the existing code is rewritten to achieve the desired goals. However, there are situations when a language evolution would be a better and a more natural solution over the application evolution. This is especially the case if the evolution is implicit in a language construct. Think of domain-specific languages in which domain concepts are aligned with language constructs [122, 87, 49]. Any evolution in the domain can, thus, naturally be reflected in the evolution of the concerning constructs. This strong link between the domain and the language could be used to dynamically update systems that cannot be shutdown. In [19] we showed that even applications written in general-purpose languages can benefit from language evolution if the latter is implicit in a language construct. However, poor language design and the lack of modularity support often prevent this kind of evolution.

Sometimes developers resort to reflection in order to tailor the language to the application needs. However, in many mainstream languages the reflection support is quite restricted and often limited only to introspection. Furthermore, the reflection support often comes in the form of libraries or language constructs that developers can use from within the same applications they are trying to evolve through language evolution. Consequently, reflection and application code are mixed which might obscure the original application logic.

As briefly discussed above, the evolution of a language often requires that the overlying software be rewritten. This is often a timely and economically expensive process, especially in large-scale projects. Sometimes the software *cannot* follow the evolution of the language and the underlying platform due to economic reasons, but it *must* evolve in order to satisfy the customers which would otherwise seek for competitive solutions. A technically sustainable platform would allow one to adopt and exploit the new features of the platform language and at the same time it would preserve the original application semantics by leaving the sources untouched.

The discussion so far elicited the need for a smoother approach to language evolution which would enable developers to

– alleviate the migration from an older to a newer version of a programming

[3] E.g., see `https://developers.google.com/awareness`.

language by allowing one to partially apply the evolution;

- dynamically evolve the language in order to catch up with the emerging requirements and domain evolution;
- evolve the language without modifying the original application code;
- separate application evolution from that of the language.

To this purpose, we defined the concept of *open programming language interpreters* (from here after *open interpreters*) which enable language extensions and evolution through reflection. A peculiarity of this approach is that the language extension code is completely separated from the rest of the language implementation as well as from the application code. Furthermore, under specific conditions, the extensions can be shared across different language implementations. With this approach it is possible to evolve the language both statically and/or dynamically. We illustrate how the approach can be effectively used to extend languages and the overlying applications with cross-cutting concerns (profiling, logging, etc.), to build linguistic tooling and instrumentation (e.g., debuggers), to adapt an application to better fit the running context, to support backward compatibility, to remove security flaws, etc. Open interpreters use some aspect-oriented programming (AOP) concepts which draws our approach closer to those familiar with AOP.

Contribution. The contribution of this work is summarized in the following points:

- we contribute a clear definition for the novel concept of open programming language interpreters,
- we define the scope of application of open interpreters;
- we define a framework-level API to support introspection and intercession of open interpreters,
- we define a domain-specific language for user-friendly introspection and intercession of open interpreters,
- we integrate the support for open interpreters in Neverlang, hence, all Neverlang-based interpreters become automatically open,
- we define a type and inference system to ensure the correctness of composition and dynamic adaptation.

Organization. Chapter 2 introduces concepts that underly and help the reader better understand the novel concept of open interpreters. Among others, it presents Neverlang, our core framework for modular development of programming languages. We also present the Neverlang's type and inference system for preserving the composition soundness during and after the language evolution process. Chapter 3 defines the

concept of open interpreters and describes how tree-based interpreters can become open according to this definition. Chapter 4 describes the architecture needed to support open interpreters, the integration of such support in Neverlang and it introduces a DSL for reflection operations on interpreters. In Chapter 5 we discuss different application domains of open interpreters. Chapter 6 discusses the related work and Chapter 7 draws conclusions on the topic. Additionally, Appendix A provides a formal definition of language composition which is crucial for the correct operation of the interpreter and, especially, for preserving the integrity of language interpreters after their (dynamic) evolution.

2 Background

In this chapter we present the necessary concepts to understand the idea of open interpreters. We explain what a programming language is and how we define one. We briefly describe how an interpreter is developed and how it works. This information should provide the reader with the necessary background to understand the novel concept of open interpreters described in Section 3. Then, we introduce the modular development of programming languages with Neverlang as a representative of this model. Readers with background in programming languages can skip a large part of this chapter and read just Section 2.3. Also, we advise reading Section 2.2.4 that describes the Neverlang framework which is used in Chapter 5 to illustrate the applicability of open interpreters.

2.1. Programming Languages and Interpreters

Just as natural languages, programming languages are a communication means. They are used by developers to instruct the machine to perform some task. Differently from human brains, computers are bad at handling informality, vagueness and ambiguity which are typically present in natural languages. For this reason, the communication between the developer and a machine requires strict formal rules that leave no room for ambiguity. Several formal methods were designed to define programming languages. In the following, we briefly discuss context-free grammars and syntax-directed definitions. We provide only those details that are necessary to follow the discussion on open interpreters in Section 3.

2.1.1. Defining the Syntax: Context-free Grammars

According to the Merriam-Webster's dictionary, syntax is defined as "*the way in which linguistic elements (such as words) are put together to form constituents (such as phrases or clauses)*". In the context of programming languages, syntax is the way in which linguistic constructs (such as numbers, arithmetic operators, function calls, etc.) are put together to form valid sentences and programs. Valid sentences are typically determined by a language grammar which describes the hierarchical structure of language constructs. One possible and a generally-known notation for specifying a language syntax is context-free grammar.

Definition 1. *A context-free grammar $\mathcal{G}$ is defined as a 4-tuple $\mathcal{G} = (\Sigma, N, S, \Pi)$, where:*

```
S ⇾ Expr
Expr ⇾ Term | Expr "+" Term
Term ⇾ Factor | Term "*" Factor
Factor ⇾ Integer
Integer ⇾ Digit | Digit Integer
Digit ⇾ "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

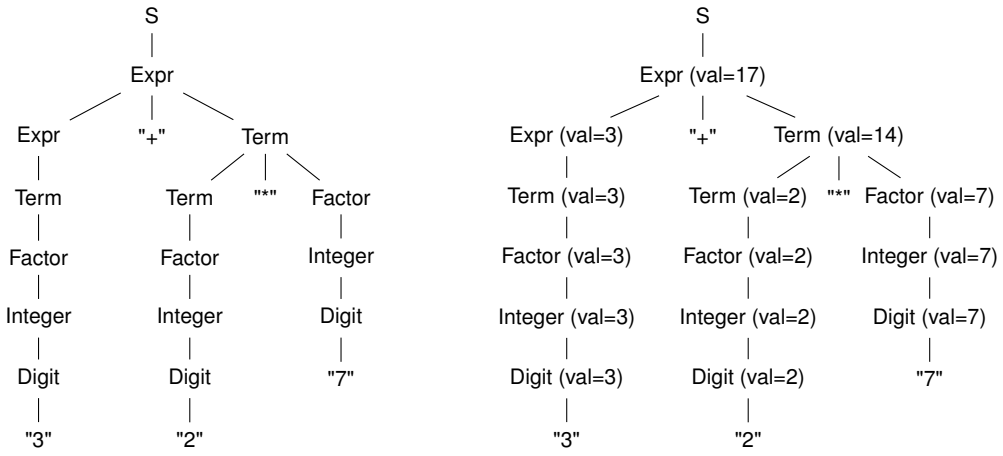
Listing 2.1: *Example grammar.*

- N *is a set of nonterminal symbols or syntactic categories,*
- Σ *is a set of terminal (or elementary) symbols of the language defined by the grammar and* $N \cap \Sigma = \varnothing$,
- Π *is a set of production rules of the form* $P : N \to (N \cup \Sigma)^*$,
- S *is the start symbol (or start nonterminal), where* $S \in N$

A context-free grammar is given by listing the production rules. Listing 2.1 shows a context-free grammar for the arithmetic operations of multiplication and addition. By convention, nonterminals are capitalized and terminals are quoted. Notice that the vertical bar "`|`" denotes an alternative derivation. For example, `Digit ⇾ "1"| "2"` denotes two production rules, namely `Digit ⇾ "1"` and `Digit ⇾ "2"`. From the grammar definition we see that

- $N = \{$`S,Expr,Term,Factor,Integer,Digit`$\}$ is a set of nonterminal symbols,
- $\Sigma = \{0,1,2,3,4,5,6,7,8,9,+,*\}$ is a set of terminal string symbols,
- $S =$ `S` is the starting symbol,
- and the set of production rules Π is given by the rules listed in Listing 2.1.

From a language grammar definition we can derive a parse tree of the source program code. A parse tree is a hierarchical syntactic structure of the input source code. The root of the tree is the start symbol of the grammar, the leaves correspond to the terminal symbols of the derived sentence, while the internal nodes correspond to the grammar nonterminals. The tree can be derived either top-down or bottom-up. In the top-down derivation we start with the start symbol and then apply the production rules until we derive the sentence. The application of rules means substituting the left-hand side nonterminal with those in the production body. On the other hand, in the bottom-up derivation we start from the sentence and apply the production rules in reverse order (by replacing the right-hand side symbols with the head nonterminal) until we derive the start symbol. If there exists no derivation given the grammar rules then the input program is not valid. Fig. 2.1(a) shows the parse tree derived by the grammar in Listing 2.1 for the expression `3+2*7`. The tree structure is closely related to the grammar definition and the leaf nodes, if read from left to right, correspond exactly to the original expression.



(a) parse tree (b) annotated parse tree

Figure 2.1.: *Parse tree for the expression 3+2*7.*

No.	Production	Semantic Rule
1	`S` ⇾ `Expr`	`print(Expr.val)`
2	`Expr` ⇾ `Term`	`Expr.val = Term.val`
3	`Expr` ⇾ $\texttt{Expr}_1$`"+"Term`	`Expr.val =` $\texttt{Expr}_1$`.val + Term.val`
4	`Term` ⇾ `Factor`	`Term.val = Factor.val`
5	`Term` ⇾ $\texttt{Term}_1$`"*"Factor`	`Term.val =` $\texttt{Term}_1$`.val * Factor.val`
6	`Factor` ⇾ `Integer`	`Factor.val = Integer.val`
7	`Integer` ⇾ `Digit`	`Integer.val = Digit.val`
8	`Integer` ⇾ `Digit` $\texttt{Integer}_1$	`Integer.val = concatInt(Digit.val,` $\texttt{Integer}_1$`.val)`
9	`Digit` ⇾ **`"0"`**	`Digit.val = 0`
10	`Digit` ⇾ **`"1"`**	`Digit.val = 1`
...		
11	`Digit` ⇾ **`"9"`**	`Digit.val = 9`

Table 2.1.: *Syntax-directed definition example.*

2.1.2. Defining the Semantics: Syntax-Directed Definitions

Syntax-directed definition (SDD) [1] is a formalism for defining both the syntax and the semantics of a programming language. It extends a context free grammar with attributes that are associated with grammar symbols. Attribute values are calculated by semantic rules that are associated with grammar productions. If semantic rules have no side-effects, SDDs are called attribute grammars [75].

For example, in Table 2.1 we define a list of SDDs by associating a semantic rule with each production from Listing 2.1. For each grammar symbol we define the attribute `val` which is calculated by semantic rules associated with grammar productions. For example, the last row associates with the production `Digit` ⇾ **`"9"`** the semantic rule which sets the value of the attribute `val` to 9.

Conceptually, a language construct can be defined by several SDDs, depending on

how one structures the grammar. A language construct which is fully-defined by its syntax and semantics represents the minimal distinguishable meaningful concept of a language and is called a language feature [121].

2.1.3. Interpretation

The interpretation process is more easily understood if we first build a parse tree from the input source code, although a parser can perfectly evaluate a program without building its tree representation. At each node, the parser will execute the semantic rule associated with the grammar production that was used to generate the subtree rooted at the current node. Conceptually, the link between nodes and their semantics is syntax-driven. The order in which the nodes are visited is determined by attribute dependency. For example, an attribute might depend on attributes defined in the node's children, hence these should be visited before the current node is evaluated. A badly defined SSD might lead to a situation where there is no suitable order for attribute evaluation.

Consider the expression $3 + 2 * 7$ interpreted by the language defined in Table 2.1. Figure 2.1(b) shows the annotated parse tree obtained by the interpretation. The attribute dependency would make the interpreter first go all the way down to the tree leaves. Then it would proceed up the tree to the root node. For example, when the `Digit` node in the left-most branch is visited, the interpreter executes the semantic rule no. 10 from Table 2.1, which attaches to the tree node the `val` attribute whose value is 1. The visit would proceed on the parent node (`Integer`) associated with semantic rule 7 which simply copies the `val` to the current node. The `val` attribute is thus propagated up the tree. When all nodes are visited and all rules executed, the tree is decorated as in Figure 2.1(b) and the `val` attribute of the topmost `Expr` node contains the final value of the expression $3 + 2 * 7$.

2.1.4. Development of Programming Language Interpreters

There are many ways to implement a programming language interpreter and each presents its own benefits and drawbacks. The developer is usually put in front of a trade off between simplicity and performance. For example, tree-based interpreters are simple to implement, but usually have performance issues. On the other hand, bytecode interpreters are faster at the cost of a more complex implementation. In this discussion we will focus on tree-based interpreters. We motivate our choice because 1) tree-based evaluators are considered to be one of the simplest way to implement interpreters [128] and thus facilitates the discussion, 2) many interpreter development frameworks are tree-based [96, 67, 128, 86, 58] which makes the idea of open interpreters, as described in this paper, highly portable and 3) recently it was shown that, despite their simplicity, tree-based interpreters can be very efficient [128, 11, 27].

The development approaches for tree-based interpreters can be divided in two categories: manual and semi-automatic development. The first approach consists in manually building a hierarchy of language features. In an object-oriented setting, this

```
class Node {
}
abstract class Expr extends Node {
  abstract Object execute(Environment env);
}
abstract class Stmt extends Node {
  abstract void execute(Environment env);
}
class IntegerNode extends Expr {
  @Override
  Integer execute(Environment env) {
    // implementation of integer behaviour goes here
  }
}
```

Listing 2.2: *Manual implementation of the AST node class hierarchy.*

is typically done by defining the hierarchy of AST node classes through the inheritance and polymorphism mechanisms. Listing 2.2 shows a Java snippet with a partial implementation of such a hierarchy. This approach can contribute to the interpreter's efficiency as its implementation can be tailored to a specific language, as opposed to framework-generated code that unavoidably contains some overhead code due to automatization.

A more common approach is to use a formalism, e.g., syntax-directed definitions, to describe a language specification and then use a compiler/interpreter generator to automatically produce an interpreter. Typical examples of tools that support this kind of development are Yacc [64, 80] and ANTLR [97, 96]. In the next section, we show this approach in the Neverlang framework.

2.2. Modular Development of Programming Languages

The complexity of software systems is traditionally faced with modularization which favors the separation of concerns, independent development, maintainability and reuse. Programming language implementations are complex systems, thus, several tools provide modularization support for developing programming languages [86, 58, 123, 67, 121]. Although the principles of modularization apply to language interpreters and compilers, we will focus on interpreters which are the topic of this dissertation. Later in this chapter we introduce the Neverlang framework for modular development of programming languages. This should help the reader to better grasp some abstract concepts and to understand the examples that in Chapter 5 are discussed as a proof of concept.

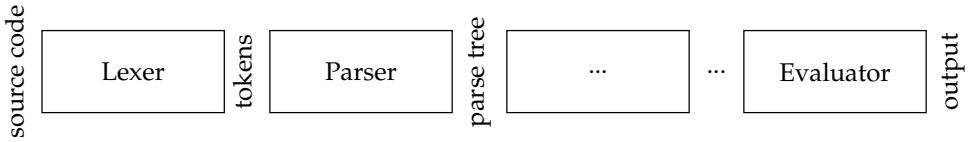


Figure 2.2.: *Vertical decomposition of an interpreter implementation.*

2.2.1. Language Decomposition

In traditional language development, the language complexity is commonly faced by decomposing the interpreter's implementation functionally. This type of decomposition is sometimes called a vertical decomposition and typically results in a lexer, parser, and code evaluator, with possible additional intermediate phases for optimization. As illustrated in Fig. 2.2, the output of each phase becomes the input of the next one and eventually the interpreter produces the desired output. Thanks to the information hiding principle, existing phases can be replaced or new ones can be interposed. Vertical (functional) decomposition is very common among language implementations. On the other hand, interpreters are rarely structurally decomposed. A structural (sometimes called horizontal) decomposition consists in developing language-oriented concepts as loosely-coupled reusable linguistic components that can be combined to form a complete language. In this context, a linguistic component can be any language feature, such as a variable declaration or the addition operation. Such components might even be precompiled [30] and shared across different language implementations. Thus, a language can potentially be extended by plugging in new components.

This fine-grained modularization highly eases the (dynamic) evolution of programming language interpreters as will be discussed in Chapter 3. In the following, we discuss this issue in the context of the Neverlang framework for modular language development. This will help the reader to better grasp the above concepts and will serve as the basis for the discussion on open interpreters (Chapter 3) and on the integration of their support in Neverlang (Chapter 4).

2.2.2. Language Composition

In general, the composition of components is possible due to the already highlighted information hiding principle according to which components interact with each other trough interfaces. Therefore, in order to compose linguistic components they must match on the interface. The question arises as to what makes the interface of a linguistic component? To answer this question, consider the following SDDs, provided as two separate components, that define, respectively, the addition and the multiplication operations.

```
Component 1:   Expr ⇐ Expr1 "+" Expr2   { Expr.val = Expr1.val + Expr2.val; }
Component 2:   Mul ⇐ Expr1 "*" Expr2   { Mul.value = Expr1.value * Expr2.value; }
```

To be fully compatible and, thus the two SDDs must match on nonterminals and the attributes they define. The compatibility on nonterminals determines if it is possible to build a parse tree for a mathematical expression with both the addition and the multiplication operations (e.g., $3 + 2 * 7$). The two components fail to match on this aspect of the interface. Consequently, there would be no way to build, for example, the parse tree for the expression $3 + 2 * 7$ by composing the two grammar productions. In fact, the production for the addition operation "accepts" only the `Expr` nonterminals, while the multiplication production is defined as `Mul`. To make the two SDDs compatible we have to rename the nonterminals as follows (the modified part is written in bold face).

```
Component 1:   Expr ⇐ Expr1 "+" Expr2   { Expr.val = Expr1.val + Expr2.val; }
Component 2:   Expr ⇐ Expr1 "*" Expr2   { Expr.value = Expr1.value * Expr2.value; }
```

However, although it is now possible to build a parse tree for the expression $3 + 2 * 7$, the SDDs would still be incompatible due to the wrong naming of attributes. Indeed, the semantic action for the addition operation expects that the nonterminals $\texttt{Expr}_1$ and $\texttt{Expr}_2$ provide the `val` attribute, but the multiplication semantic action defines the `value` attribute. Therefore, to make the two SDDs fully compatible, we would have to rename the attributes as follows.

```
Component 1:   Expr ⇐ Expr1 "+" Expr2   { Expr.val = Expr1.val + Expr2.val; }
Component 2:   Expr ⇐ Expr1 "*" Expr2   { Expr.val = Expr1.val * Expr2.val; }
```

Nonterminals in the body of a production also contribute to the component's interface. Indeed, these are nonterminals that must be provided by other components in order for the language definition to be complete. We say that two components are partially compliant when they provide some, but not all nonterminals that are required by each other. They are combinable but together they do not provide a complete language definition. Consider the following SDDs.

```
Component 1:   Term ⇐ Term "*" Factor    { ... }
Component 2:   Term ⇐ Factor    { ... }
```

These components can be combined because `Component 1` defines `Term` and requires `Term` and `Factor`. `Component 2` defines `Term` and requires `Factor`. Therefore, `Component 2` partially provides what `Component 1` needs. However, now they together require that another component provide a definition for `Factor`. To summarize, the interface of linguistic components is given by the required and provided nonterminals and the attributes they define. This is analogous to function definitions whose signature (interface) is given by the type of input parameters (required data) and the type of the returned value.

In Figure 2.2 we illustrated a vertical decomposition that transforms the input source code to the program output. Functional phases also need to match on an interface. Indeed, each phase elaborates its input and produces an output to feed the

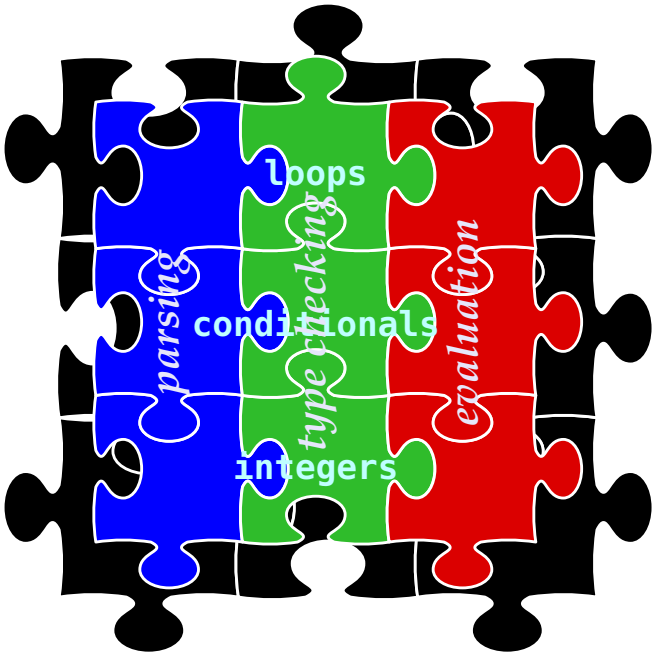


Figure 2.3.: *Language structural and functional composition.*

next phase, with the exception of the last phase that produces the final output. The input/output data therefore have to match on a predefined interface. When we combine the component definitions in both dimensions, i.e., functionally and structurally, we get the final language. This composition is illustrated in Figure 2.3. Each puzzle piece has rounded tabs and "holes" that represent their interface. Pieces that match can be combined. Each horizontal line represents a language component and each piece in a line represents its implementation in a given phase. Vertical lines represent semantic phases. For more information about component interfaces and their composition the reader is referred to the Appendix A.

2.2.3. Composition Soundness

As discussed above, component composition depends on whether the grammar is well defined. The problem of ensuring that a grammar is well defined has been addressed since 1968 when Knuth for the first time introduced attribute grammars. Several techniques were since then elaborated, but mainly they rely on the closure and non-circularity properties.

The *closure property* requires that for each attribute there exist a semantic rule that defines its value. This is fairly easy to ensure in the context of pure attribute grammars in which an attribute is associated with the result of a function without side-effects; by definition of a function, which always returns a value, this guarantees that attributes do not remain uninitialized. Therefore, to verify the closure property it suffices to check

that each attribute is associated with a function. Unfortunately, many tools do not adopt full-fledged attribute grammars and although a semantic rule exists it might not set the attribute. Among such tools we find Yacc, ANTLR, Silver [123], Neverlang [121] and Lisa [86].

The *non-circularity property* states that an attribute value must not depend on the attribute itself. In a monolithic setting, Knuth [75] presented an algorithm for testing a grammar for both closure and non-circularity. Vogt *et al.* [124] extended Knuth's algorithm in the context of higher-order attribute grammars. Backhouse [4] presented a definedness test that embodies both closure and circularity checks.

Although far from being trivial, ensuring well-definedness in a monolithic setting is simpler when compared to the same task carried on a modular model where information hiding, a key principle that assists composition, might also introduce interesting challenges. In a modular setting, Kaminski *et al.* [66] presented a well-definedness analysis for attribute grammars applied to a modular system. The analysis checks that the composition of a host language with its extension results in a complete grammar definition with no circular dependencies in attribute equations. The proposed solution is applied to Silver, an attribute grammar system supporting extensions through forwarding. Later in this chapter we present the principles behind the Neverlang's type and inference system that ensures composition soundness in modular setting where a challenge is present due to the dynamic nature of grammar attributes. A full description and a formal definition of this type system is provided in Appendix A.

2.2.4. Neverlang

Neverlang [18, 30, 121] is a framework for modular development of programming languages built in Java. It supports both functional and structural decomposition of a language implementation.

Composition Model. Language development in Neverlang consists in defining a variant of syntax-directed definitions as reusable modular units. This is best explained by an example, therefore, we show how to partially implement a small language for arithmetic expressions. Listing 2.3 illustrates the basic Neverlang concepts. In Neverlang, the smallest unit of modularity is a **module**. Each module is uniquely identified by a Java-style canonical name (e.g., `mylang.AddSyntax` in line 1). A module can define a syntax and/or semantics of a language construct. Listing 2.3 shows two modules that define, respectively, the syntax and the semantics of the addition operation. In Neverlang, the syntax is defined as a set of grammar productions where, by convention, nonterminals are capitalized, while terminals are quoted. Each production is optionally labeled for easier reference. In our example, we have one grammar production labeled `Add` (line 3). Notice that the `Term` nonterminal is defined elsewhere, i.e., in another module (for simplicity not showed here). This is possible due to the Neverlang's composition system that will be explained later.

Module `mylang.AddSemantics` defines how the addition operation behaves. In Neverlang, semantics is expressed in terms of semantic actions, where each action is written

```
1  module mylang.AddSyntax {
2    reference syntax {
3      Add: Expr <- Expr "+" Expr;
4    }
5  }
6  module mylang.AddSemantics {
7    imports { mylang.Math; }
8    role ( evaluation ) {
9      Add: .{ $Add.val = Math.add($Add[1].val, $Add[2].val); }.
10   }
11 }
```

Listing 2.3: *Neverlang basic concepts.*

between the ".{" and "}." symbols. The actions are written in pure Java code extended with a domain-specific language that provides means to access and attach attributes to tree nodes as mandated by the SDD formalism. Notice that there is no declaration of attributes which, indeed, are dynamically defined at runtime when actions are executed. This dynamic nature of attributes presents an interesting challenge in composition which is the reason we designed a type system described in Section 2.2.3 and formally defined in Appendix A. Semantic actions can optionally be labeled. Semantic actions are grouped in **role**s which represent semantic phases, such as evaluation, type-checking, etc. In our example, we define a role called `evaluation` (line 8), which defines one single semantic action labeled `Add` (line 9). In semantic actions, nonterminals are referenced to by the action label and the offset. Head nonterminal has offset 0. In our example, `Add[0]` or simply `Add` refers to the head nonterminal `Expr`, while `Add[1]` and `Add[2]` refer to the two `Expr` nonterminals in the production body[1]. Therefore, `$Add[2].val` used in line 9 refers to the `val` attribute attached to the tree node represented by the `Term` nonterminal of the production labeled `Add`. Modules can import arbitrary data structures as shown in line 7.

To this point the two modules discussed so far are unrelated. The Neverlang's **slice** construct can be used to combine two or more modules to form a component as shown in lines 1-4 in Listing 2.4. The association between productions and semantic actions is guided by labels. Hence, the grammar production labeled `Add` (in module `mylang.AddSyntax`) is tied up with the semantic action labeled `Add` (in module `mylang.AddSemantics`). Modules and slices can be shared and reused across different language implementations. If no explicit slice is defined, Neverlang implicitly creates a slice for each module.

Neverlang enables one to define objects that are globally accessible from within any semantic action. This is typically used to define data structures like symbol tables that store information about symbols in the input program, e.g., variable declarations, types, bindings, etc. Such objects are defined as endemic slices. In Listing 2.4, lines 5-7, the **endemic slice** named `mylang.MathEndemic` declares a globally accessible variable

[1]Given their constant nature, terminals need not be referenced, hence, they are not taken into account in the offset.

```
1  slice mylang.Addition {
2    concrete syntax from mylang.AddSyntax
3    module mylang.AddSemantics with role evaluation
4  }
5  endemic slice mylang.SymbolTableEndemic {
6    declare { SymbolTable : mylang.utils.SymbolTable; }
7  }
8  language mylang.Arith {
9    slices
10     mylang.Addition
11     mylang.Integer
12   endemic slices
13     mylang.SymbolTableEndemic
14   roles syntax < evaluation
15 }
```

Listing 2.4: *The composition of modules into a slice.*

called `Math` which is an instance of the user-defined class `mylang.Math` that implements mathematical operators. Endemic slices are accessed through the double dollar symbol ($$) and are accessible in any semantic action. In Listing 2.3, line 9, we use the declared `Math` variable to perform the addition operation.

Once all slices are defined, they can be composed to form a language. To this purpose Neverlang provides the **`language`** keyword whose usage is illustrated in lines 8-15 in Listing 2.4. First, one must assign a language a unique canonical name identifier (e.g., `mylang.Arith`). Then, she must list all the slices that form the language (lines 9-11). Next, any optional endemic slices are listed (lines 12 and 13). Finally, one must define the execution order of roles or semantic phases (line 14). Neverlang has a built-in role called `syntax` which is in charge of parsing and builds the parse tree that is used in subsequent phases.

Development, Compilation and Execution. The process of compiling, loading and running an interpreter in Neverlang is shown in Fig. 2.4. First, the Neverlang sources, such as those in Listings 2.3 and 2.4 are compiled to Java code using the Neverlang compiler (`nlgc`). The `nlgc` compiler takes in input optional parameters (e.g., class path, destination folder) and one or more source files. The usage of `nlgc` is shown in Listing 2.5(a). Each module and slice is compiled as a separate Java class and is thus

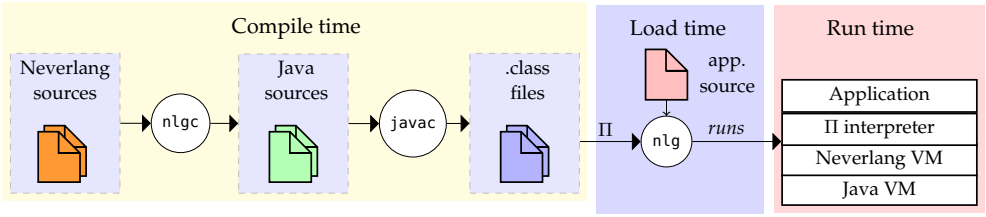


Figure 2.4.: *Neverlang's compilation process: from code to a running interpreter.*

a) `nlgc` usage.

```
albert@MSX-1914:~$ nlgc --help
Usage: nlgc <options> <source files>
Option                                Description
------                                -----------
-?, -h, --help                        Show this help message
--classpath, --cp <File: classpath>   A ':' delimited list of directories,
                                        JAR archives and ZIP archives used
                                        to look for a class file
-o, --open                            Generates an open interprter
-s <dest dir>                         Specify where to place generated
                                        source files
-v, --verbose                         Enable verbose output
--version                             Print version number
```

b) Compilation of a Neverlang module.

```
albert@MSX-1914:~/workspace/neverlang3/examples/Arith$ nlgc Addition.nl -s classes
---------------------------------------
Using Neverlang RSD Compiler 0.8.0 (compiled on: Tue Sep 26 10:41:56 CEST 2017)
Addition.nl
mylang/Addition$role$syntax.java
mylang/Addition$role$evaluation$0.java
mylang/Addition.java
albert@MSX-1914:~/workspace/neverlang3/examples/Arith$
```

c) Compilation of the generated Java sources to Java `.class` files.

```
albert@MSX-1914:~/workspace/neverlang3/examples/Arith$ javac -d build -cp
    $NEVERLANG_HOME/Neverlang.jar classes/mylang/*.java
albert@MSX-1914:~/workspace/neverlang3/examples/Arith$
```

Listing 2.5: *`nlgc` usage and the compilation process from Neverlang modules to Java `.class` files.*

reusable. Listing 2.5(b) shows the compilation of the modules for the addition operation (see Listing 2.3): `nlgc` compiles all modules defined in `Addition.nl` into Java files which are stored in the `classes` folder as specified by the `-s` flag. The `nlgc` compiler follows the Java's policy of creating a folder structure according to packages. Therefore, the generated Java files are stored in `classes/mylang` folder. The generated Java sources are then compiled to `.class` files using the Java compiler as shown in Listing 2.5(c). Finally, the `nlg` tool is used to load and run the interpreter. It takes in input the interpreter's canonical name (abbreviated with Π in Fig. 2.4) and the application source code written in the interpreted language. If the source file is not provided, `nlg` will spawn an interactive shell with a read-evaluate-print loop for the developed interpreter as shown in Listing 2.6. The `nlg` spawns the Neverlang Virtual Machine and runs the interpreter. The parser builds a tree representation of the application code which is then traversed according to the interpretation model as described in the following.

Interpretation Model The Neverlang's interpretation model is based on parse trees. For a given input program, Neverlang will derive the parse tree from the language grammar definition. The semantic phases, like type-checking, optimization, evaluation,

```
albert@MSX-1914:~/workspace/neverlang3/examples/Arith$ nlg -cp build mylang.Arith
---------------------------------------
NLGi. Neverlang Interactive REPL.

Language mylang.Arith

Available Commands:
:help      :h   Get this screen
:quit      :q   Leave Repl
:reload    :r   Reload the language implementation from disk
:tree      :t   Dump last parse tree as a Graphviz source file
:parser    :p   Dump parser to disk
:endemic   :e   Dump Endemic Slices
:grammar   :g   Dump language grammar
:multiline :m   Toggle multiline input

> 1+1
2
>
```

Listing 2.6: *Neverlang's read-evaluate-print loop for the developed interpreter.*

etc., are implemented as a visit of the parse tree during which the defined semantic actions are executed. Neverlang supports post-, preorder and a custom tree visit. The latter is supported by providing the developer with a special keyword for guiding the tree visit. If not specified otherwise, the visit is done in postorder. During the traversals, the parse tree is decorated with arbitrary attributes dynamically attached to tree nodes.

The Neverlang's interpretation model is shown in Fig. 2.5. Since a language grammar can be specified modularly, different portions of the parse tree can be related with different slices. For example, the tree in Fig. 2.5 was build from grammar productions defined in two different slices, a fact that we emphasize with the blue and pink boxes. The model's heart is given by the *inverted index* that maps from grammar productions to slices they were defined in. When a node is visited, Neverlang first checks which production was used to generate the subtree rooted at the visited node. For simplicity we refer to this production as current production and to the generated subtree as current subtree. For example, in Fig. 2.5 the visited node is `Stm` and corresponds to the head nonterminal of the print statement production (see line 4 in Listing 2.7) that generated the subtree bounded by a blue box. Neverlang asks the inverted index for the slice which defines the current production and what action is associated with it in the current role (semantic phase). The associated action is then applied to the current subtree.

```
1 module mylang.Print {
2   import mylang.Types;
3   reference syntax {
4     Print: Stm <- "print" Exp;
5   }
6 }
```

Listing 2.7: *Module implementing the* `print` *statement.*

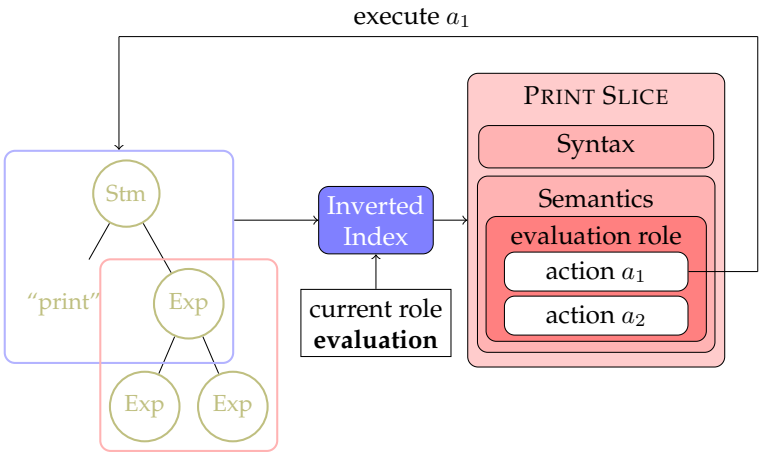


Figure 2.5.: *Neverlang's interpretation model based on the inverted index.*

Mutliple Semantic Action Dispacther. Neverlang allows one to associate many actions to the same grammar production in the same role. In Figure 2.5, the grammar production for the `print` statement is associated with two actions (a_1, a_2) in the `evaluation` role. The execution of actions is guided by condition guards, i.e., dynamic constraints that determine if an action should be applied. This allows one to specify runtime conditions that can take into account valuable context information that is not available at the development- or load time.

The developer can associate guards with a priority to sort the alternative actions by relevance. Priorities are integer numbers and a higher value means higher priority. Guards with the same priority are sorted in the order in which they were written (first come, first served). Actions without guards are implicitly assigned the lowest priority and always come after the actions with guards. When a tree node is reached, the dispatcher will verify guards one by one according to their order determined by the sorting rules. The dispatcher supports two modes: lazy and scrupulous. In the *lazy mode*, the dispatcher will trigger the first action whose guard evaluates to true and it will skip other actions. In the *scrupulous mode*, the dispatcher will execute *all* actions whose guards evaluate to true. The mode is set by a flag from command line when the interpreter is executed.

Listing 2.8 shows an example module in which three actions (lines 10, 14 and 17) are associated with the same production (they use the same label). According to the sorting rules, the guards will be verified in this order: first, the guard in line 13 (explicit priority of 10); then, the guard in line 9 (no priority, but an explicit guard); and, finally, the implicit guard for the last action which always evaluates to true. Due to the condition guards in lines 9 and 13, the first two semantic actions will trigger only when the type of the expression to be printed is, respectively, `Types.Integer` or `Types.String`.

Neverlang classifies guards in two categories, namely, soft and hard guards. Irrespectively of the type, whenever a guard evaluates to true, the associated action is executed.

```
1  module mylang.Print {
2    import mylang.Types;
3    reference syntax {
4      Print: Stm ← "print" Exp;
5    }
6  }
7  module mylang.PrintSemantics {
8    role (evaluation) {
9      (% Print[1].type == Types.Integer %)
10     Print: .{
11       // if expression type is Integer do something
12     }.
13     (% Print[1].type == Types.String %)[10]
14     Print: .{
15       // if expression type is Integer do something
16     }.
17     Print: .{
18       // otherwise do something else
19     }.
20   }
21 }
```

Listing 2.8: *Multiple semantic actions per production.*

However, the type of guard determines what happens after the action is executed. If a guard is hard, the system will proceed with normal execution, i.e., by visiting the next tree node according to the interpretation model. The next time the same node is visited, the system will recheck the same guard. The adjective "hard" conveys the idea that a hard guard should always be checked, i.e., the constraint it expresses is strong. On the other hand, if the executed action is soft guarded, it will become a specialized action for the associated tree node. The next time the system will visit this same node, it will skip the guard checking process and directly execute specialized action. The adjective "soft" suggests that, if the guard evaluates to true, its constraint is checked only once, whereas the tree node is specialized forever or until the node is despecialized (reset). Action specialization is useful for interpreter optimization as was explained in [27].

Composition Soundness Modularization is the key approach to breaking down the complexity of a system. However, it introduces the problem of ensuring the composition soundness when components are to be combined. As discussed in Section 2.2.2, components can be combined when they match on the interface they define. A component's interface is given by the required and the provided nonterminals and attributes. However, in Neverlang, attributes are defined dynamically, i.e., at runtime. Since there is no static declaration of attributes, the module's static interface is given only by the nonterminals it defines and uses. The components do not have a static interface. Consequently, it becomes more difficult to verify the composition soundness. Neverlang has a type and inference system for ensuring the correctness of composition. The inference system traces attribute definitions and their types and ensures that the

a) if-then-else implementation.

```
1  module IfThenElse {
2   reference syntax {
3    IF: Exp ⇐ "if" Exp "then" Exp "else" Exp;
4   }
5   role ( evaluation ) {
6    IF: .{
7     eval $IF[1];
8     if (toBool($IF[1].val)) {
9      eval $IF[2];
10     $IF.val = $IF[2].val;
11    } else {
12     eval $IF[3];
13     $IF.val = $IF[3].val;
14    }
15   }.
16  }
17 }
```

b) Numbers implementation.

```
18 module Numbers {
19  reference syntax {
20   INT: Exp ⇐ /\d+/
21   DBL: Exp ⇐ /\d+\.\d+/
22  }
23  role ( evaluation ) {
24   INT: .{
25    $INT.val = new Integer(#0.text);
26   }.
27   DBL: .{
28    $DBL.value = new Double(#0.text);
29   }.
30  }
31 }
```

Listing 2.9: *Composition soundness issue.*

composed interpreter is well defined, i.e., no attributes are missing and their types match. We believe that ensuring the composition soundness is essential for our system to be adopted and to support user-friendly development and adaptation of interpreters. In this section, we simplify the discussion by explaining the idea through an example. A curious reader wishful of formal details is referred to Appendix A and to [21].

In Sect 2.2.2 we highlighted that information hiding is the key principle that enables composition of linguistic components. However, as mentioned in Section 2.2.3, the same principle might introduce interesting challenges. This is especially the case when the composition interface, given by nonterminals and their attributes, is not fully statically defined, as happens in Neverlang. To better understand this, let us consider an example. Listing 2.9(a) shows a module implementing a functional version of the if-then-else construct. In line 7, the semantic action, first, traverses the subtree representing the condition part of the if-then-else construct. Indeed, the **`eval`** `$IF[1]` instruction forces the tree visit to proceed through the node identified by `$IF[1]` which corresponds to the condition part of the if-then-else language feature. This traversal may define new attributes in the node represented by `$IF[1]`. These attributes may come from other modules implementing the `Exp` nonterminal. Depending on the value of the `$IF[1].val` attribute, the control flow will proceed by traversing one of the branches of the if-then-else construct (**`eval`** `$IF[2]` or **`eval`** `$IF[3]` in lines 9 and 12, respectively). In both branches, we copy the `val` attribute of the branch to the current node (lines 10 and 13).

Since the nature of attributes in Neverlang is dynamic, an attribute might not be defined when needed for two reasons. First, an attribute might be defined only when a specific computational path is followed. For example, let us suppose that the else

branch in Listing 2.9(a) does not include the `$IF.val = $IF[3].val;` statement (line 13). In this case, we would have no guarantee that, after the semantic action execution, the attribute `$IF.val` will be defined. This would depend on whether the condition of the if-then-else statement is true. The second reason why attributes might no be available when needed lies in their naming. Indeed, attributes that have the same purpose could have different names. Modules can be developed by different developers who follows different naming conventions. Additionally, developers could name attributes differently by error or by distraction. Let us consider Listing 2.9(b) where both integer and floating point numbers are defined. The two rules define regular expressions to match integers and doubles respectively (lines 20 and 21). So when the parser matches a number it will build a subtree rooted at `Exp` and with a child holding the matched value. The semantic actions simply extract the matched number (stored by the lexer in `#0.text`) and put its value in an attribute. The semantic action for double—labeled by `DBL`—defines an attribute named `value` instead of `val`. Thus, the language could not guarantee that the `Exp` nonterminal will *always* have the `val` or the `value` attribute. This last issue becomes particularly common when composing programming features whose implementation has been developed by different teams.

To identify this kind of errors, we provided a formalization of Neverlang that describes all the relevant entities involved in the framework and their formal semantics. This formalization decorates semantic actions and nonterminals with types specifying their definition and use of attributes. The decorations are used to assess the result that: *if the code of the semantic actions is well-typed with respect to these decorations then computation on the syntax-tree of any string of the language correctly proceeds*. The result assumes that we have a complete language implementation, however, since development is compositional we specify type-checking incrementally, by associating with a slice the information about the defined/used attributes of the nonterminals occurring in it. To type-check the composition of slices, we use this information, i.e., the code of the semantic actions of the slice is not needed.

Returning to our example, the slice of Listing 2.9(a)[2] is well-typed and the type associated with this slice says that nonterminal `Exp` requires that the attribute `val` be defined after the evaluation of any semantic action associated with a production for `Exp` (since after **`eval`** `$IF[1]` the attribute `val` is required by the condition). Moreover, all the semantic actions of this slice (in this case there is a single one) define the attribute `val` of `Exp`. If this was not the case, e.g., one of the two branches of the conditional does not define the attribute `val` for the head nonterminal, this slice would not be well-typed. Also the slice of Listing 2.9(b) is well-typed, and the type associated with this slice specifies that nonterminal `Exp` does not require the definition of any attribute, and that the semantic action of the slice does not define any attribute for `Exp`. However, the composition of the two slices is not correct, since slice of Listing 2.9(b) does not define the attribute `val` for `Exp`, which is required by the type of the other slice. If we substitute `value` with `val` in the semantic action `DBL`, then the type for the slice of

[2]Recall that Neverlang implicitly creates a slice for each module. Therefore, although in Listing 2.9 we define modules, in the text we refer to them as slices.

Listing 2.9(b) would specify that the attribute `val` is defined for `Exp` and the composition would be correct. In this case, the type of the composition would be equal to the type of Listing 2.9(a).

The type decoration needed for type-checking can be inferred. In Appendix A we outline an algorithm that, given a Neverlang slice, analyzes the code of its semantic actions and produces the information about the definition/use of attributes for the nonterminals associated with the slice. The algorithm fails in case the slice cannot be decorated in such a way that type-checking succeeds. Moreover, if the algorithm succeeds from the information produced we can derive all possible decorations for the slice. Type inference of composition of slices relies on this information, making inference compositional.

2.3. The Structure of Programming Language Interpreters

From the application developer's point of view, an interpreter can be seen as a black box that takes in input the application source code and produces an output. However, if we open up this black box and consider things from the language engineer's perspective, we can identify many distinguishable parts that are combined to form the interpreter. Roughly, we can say that an interpreter is made of:

- grammar rules for identifying valid statements;
- semantic actions that assign behavior to language constructs;
- auxiliary data structures for storing various information (e.g., symbol table).

Grammar rules and semantic actions are typically tied up to form linguistic components. However, we will use the terms "component" and "language feature" in a broader sense to denote any component that contributes to language definition. Hence, language features include linguistic constructs (e.g., variable declaration, inheritance, etc.) and auxiliary data structures (e.g., symbol tables).

In the following we provide formal definitions of these elements which will later allow us to define reflection operations that should be supported by open interpreters.

Definition 2.

- *A grammar production p is defined as $p : X \to \overline{Y}$ where X is the head nonterminal and $\overline{Y}$ is the body of the production.*
- *A sequence of grammar productions is defined as $P = p_1, ..., p_n$ where p_k is a single production as per previous point.*
- *A set of semantic actions defined in a semantic phase (role) r is denoted as $\mathcal{A}_r = \{a_1, ..., a_n\}$.*
- *Given a sequence of productions P, a component $\mathcal{S}_P$ is a sequence of semantic phase (role) definitions $\mathcal{A}_1, \ldots, \mathcal{A}_n$ where productions in P and actions in each $\mathcal{A}_r$ are related positionally, i.e., p_k and a_k (from any $\mathcal{A}_r$ defined in component $\mathcal{S}_P$) are tied up*[3].

[3]Notice that in Neverlang productions and actions are coupled by labels. However, labels are syntactic sugar built upon this number-based composition mechanism.

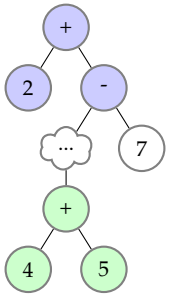


Figure 2.6.: *Two instances of the addition operation.*

- *For simplicity, components that do not contribute to language grammar (e.g., symbol tables) are also denoted with $\mathcal{S}_P$, where P is an empty sequence.*
- *The structure of the interpreter $\Upsilon_{\mathcal{L}}$ for the programming language $\mathcal{L}$ is defined as $\Upsilon_{\mathcal{L}} = \{\mathcal{S}_P, \ldots, \mathcal{S}_{P'}\}$*

Language features whose syntactic component is empty (e.g., garbage collectors, symbol tables, auxiliary data structures, helper classes, etc.) will from here after be called non-grammatical language features. On the other hand, we will use the term linguistic component or linguistic construct to refer to those language features that have a syntactic meaning.

The constituent parts identified so far all have a static and a dynamic aspect. The static aspect of an element is given by its definition or specification and is implementation-dependent. The dynamic aspect is given by its runtime instances. In the object-oriented terminology, the static aspect is given by the component's class and its dynamic aspect is given by the class instance objects. Consider, for example, a symbol table in a Java-based interpreter. The static aspect of a symbol table is given by its class definition which determines its behavior. The dynamic aspect is given by the state of its runtime instance (e.g., current variable bindings). Or, consider the Neverlang-based language construct for the addition operation defined in Listing 2.3. The static aspect of the construct is given by the modules that define its grammar and semantics. The dynamic aspect is given by its runtime instances which correspond to subtrees of the parse tree. Consider a simplified tree in Figure 2.6 where for clarity the nodes were labeled with numbers and arithmetic symbols (+ and −) instead of with their respective terminals and nonterminals. The blue- and green-colored nodes represent two instances of the addition operation construct.

3
Open Interpreters

We explain the concept of open interpreters top-down, starting from the general notion of open systems and then by restricting the definition to programming language interpreters. The concept of open implementations is a more general notion of reflection applicable to all sorts of adaptable systems, i.e., not necessarily to programming languages. Rao [102] gives the following definition:

> "*A system with an open implementation, besides providing a familiar interface to its functionality called a base level interface, reveals aspects of its implementation through a metalevel interface. The metalevel interface defines points in the implementation that can be tailored by the user.*"

Let us restrict this definition to programming language interpreters. In order to make the interpreter do something, the developer must use linguistic constructs to write a program and then execute it. In other words, the base-level interface to the functionality of a language interpreter is given by the language itself, i.e., by its constructs. Indeed, when language constructs are executed they put the interpreter "in motion". Many programming languages also expose the meta-level interface through a metaobject protocol, which "strips away a layer of abstraction", as suggested in [59], unveiling the implementation details of language constructs (base-level interface) which can potentially be customized. Reflectional features allow one to introspect and modify the base level functionality provided by the default interpreter implementation. Hence, one is able to tailor the behavior of the interpreter on the task to be solved.

Many software systems are critical and cannot be shut down. Any change and evolution of such systems require the adoption of dynamic software updating techniques. Furthermore, developers might not have access to the system's source code. Indeed, the system can be made of third-party modules which are precompiled. Therefore, there is a need to reflect upon such systems without modifying the original source code.

From what was said, we define open interpreters as follows:

Definition 3. *The interpreter $\Upsilon_{\mathcal{L}}$ of the programming language $\mathcal{L}$ is said to be open if it provides a meta-level interface which enables one to access and alter the static and dynamic aspects of its constituent parts as per Definition 2 in a controlled way through an interface without modifying the source code of the interpreted application.*

In the following we will discuss the implications of Definition 3 as well as the requirements that an interpreter must satisfy in order to be open. Then, we describe one possible model that complies with the definition and the evinced requirements.

3.1. Implications and Requirements for Open Interpreters

The definition of open interpreters is general enough to be implemented in a variety of ways making it applicable to many existing language development frameworks. However, frameworks differ from each other in the development model (monolithic or modular), the intermediate code representation (parse tree, bytecode), etc., which makes a broad discussion of open interpreters' applicability very difficult. There are just to many variations. But, from the definition we can derive general principles which imply high-level requirements that can guide framework developers to integrate the support for open interpreters. We discuss these implications and requirements and design a possible tree-based model for open interpreters.

The definition of open interpreters requires that the developer must be able to introspect and modify[1] all the interpreter's constituent parts as defined in Definition 2. This implies that all language features ($\mathcal{S}_P$) that form a language be distinguishable. Although a monolithic language implementation can provide a mechanism to distinguish language components, a modular implementation has the advantage that its components are by definition distinguishable modular units. This eases the reasoning about open interpreters and the reflection operations on its constituent parts. However, it is not mandatory to have a modular implementation as long as it provides a mechanism to distinguish single language components.

Similarly, Definition 3 states that an open interpreter must enable one to reflect upon single runtime instances of language features. This implies that component instances must be identifiable. Furthermore, any modification to the state or the specification of an instance must not affect other instances of the same type. To understand this, consider an implementation in an object-oriented setting. A class represents object specifications. Objects are instances of a class. Changing a class modifies the behavior of all its instances. Therefore, one must be able to modify single objects without modifying its class.

However, the definition of open interpreters says nothing about how exactly should interpreter's constituent parts be made available for reflection. Typically, reflective operations are provided through API or special constructs by the language itself. In this case, the reflection and the application code are mixed. However, we believe that interpreter adaptation, as a separate concern, should be coded separately to maintain the application sources clean and understandable. Furthermore, as explained above, this is sometimes inevitable, especially when sources are unavailable or reflection should be performed on critical systems that cannot be shutdown. Therefore, open interpreters require that reflection and application code be separated.

To summarize, we can derive the following requirements that an interpreter must satisfy in order to be open.

1. An open interpreter must enable one to distinguish and act upon the specification

[1] The ability of a program to modify its own execution state or its own interpretation or meaning is often called *intercession*. Therefore, in this dissertation we use terms "modify" and "intercede" interchangeably.

of its constituent parts which are defined in Definition 2.

2. An open interpreter must enable the developer to distinguish component occurrences and to introspect and intercede them in isolation.
3. The reflective code should be separated from the interpreter and application code.

The requirements are deliberately general and minimalistic in order to allow for a broad range of implementations. In the following we describe one possible model that complies with these requirements.

3.2. A Model for Open Interpreters

In this section we describe a minimalistic tree-based model for open interpreters that satisfy the above requirements. We focus on tree-based interpreters for following reasons: 1) tree-based evaluators are considered to be one of the simplest way to implement interpreters [128], 2) many frameworks for interpreter development are tree-based [96, 67, 128, 86, 58] which makes the idea of open interpreters, as described in this dissertation, highly portable and 3) it was shown that, despite their simplicity, tree-based interpreters can be very efficient [128, 11, 27]. Variations of the presented model are possible and the use of tree-based evaluators should in no way limit the portability of the idea to, e.g., bytecode interpreters, as long as the above definitions hold. Tree-based interpreters were widely discussed in Chapter 2. In the following we explain how an interpreter that reflects the structure defined in Definition 2 can be made open according to Definition 3.

3.2.1. Reflection on Language Specification

Reflection on language specification comprises reflection on language features ($\mathcal{S}_P$), which include language constructs (e.g., variable declaration, for loop, etc.) and non-grammatical components (e.g., symbol tables, etc.), as well as on the language configuration itself ($Y_{\mathcal{L}}$). The question of how to provide reflection on the language specification does not depend as much on the specific model but rather on the reflection capabilities of the language or the tool used to build the interpreter and its runtime system. Let us suppose, for example, that one decides to build an interpreter in Java by manually defining the class hierarchy of language constructs as illustrated in Listing. 2.2. The interpretation model is syntax-driven and tree-based. But that does not say anything about whether the developer is able to modify the language at runtime. That rather depends on Java's reflection support to introspect and modify classes and their instances.

Reflection on language specification requires a more powerful reflection mechanism than the one provided by Java[2]. Indeed, in a Java-based interpreter, the language specification would be given by class methods and fields which should be made

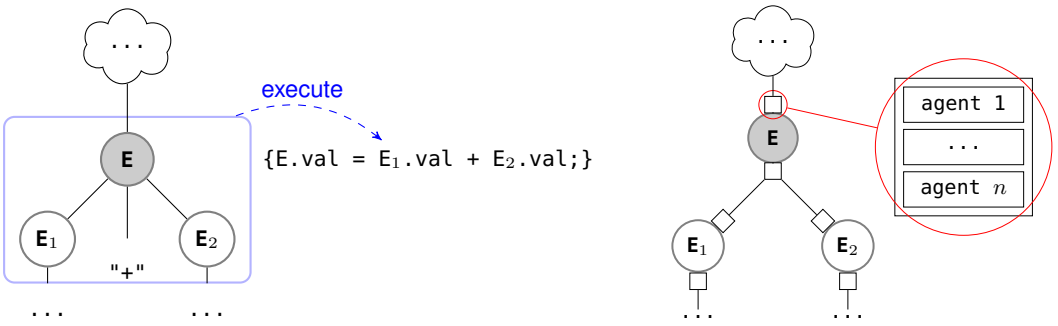
[2]The discussion that follows is not confined to Java which we use only as a representative of many programming languages that have a poor intercession support. Hence, the elicited principles apply to many programming languages.

accessible outside the interpreter. Java's native reflection support would allow one to "read" the language specification, but it has serious limitations in providing ways to modify the behavior of objects. Although one can use dynamic class loading to change the language specification, such changes would affect the behavior of all instances of the modified class. While in some cases this might be a desirable effect, it is in contrast with the second requirement defined in Section 3.1. In fact, Java provides no elegant way of modifying the behavior of single component instances without modifying the behavior of other instances of the same component. For example, consider changing the semantics of a linguistic component by replacing one class definition with another through dynamic class loading. The component replacement operation is formally defined as follows:

$$replace(\mathrm{Y}_{\mathcal{L}}, {\mathcal{S}_P}^{\mathrm{old}}, {\mathcal{S}_P}^{\mathrm{new}}) = (\mathrm{Y}_{\mathcal{L}} \setminus \{{\mathcal{S}_P}^{\mathrm{old}}\}) \cup \{{\mathcal{S}_P}^{\mathrm{new}}\} \quad \text{where } {\mathcal{S}_P}^{\mathrm{old}} \in \mathrm{Y}_{\mathcal{L}}$$

To understand how this change would affect all occurrences of that linguistic component, consider the interpretation of the expression `1+2+3` with the "program counter" just before the expression. If at that point of execution we change the specification of the addition operation, for example, by associating the subtraction semantics to the plus operator, the change would affect both occurrences of the addition operation, i.e., the result of the expression would be `-4`, instead of `6`. But the second requirement in Section 3.1 states that one must be able to affect single components in isolation without modifying other components of the same type. This is something hardly achievable with Java's native reflection support without recurring to intricate workarounds. The same is true for non-grammatical language features (e.g., symbol table) which, in addition to linguistic constructs, are part of the language specification.

Bottom line is that reflection on language specification depends on the reflection capabilities of the underlying language and its runtime system. Therefore, the model we present does not impose any specific reflection technique, but simply requires that its implementation must provide a way to introspect and intercede the static definition (the behavior) of language features. If reflection support is limited and there is no way of changing the behavior of components, the interpreter cannot be defined as open according to Definition 3. Notice, however, that the lack of reflection support by the underlying language does not necessarily mean that an interpreter cannot be open. For example, Neverlang-based interpreters target Java whose reflection support is limited mainly to introspection. However, the Neverlang framework provides a runtime system, which is an additional abstraction layer between Neverlang-based interpreters and the Java Virtual Machine (see Fig. 2.4). This runtime system provides additional reflectional features that rely upon dynamic class loading and enable one to fully introspect and modify all language elements described in Definition 2. The details will be provided in Chapter 4.



(a) Common tree-based interpretation model (b) Model extended with hooks and agents

Figure 3.1.: *A common and an extended interpretation models.*

3.2.2. Reflection on Language Feature Instances

As per Definition 3, an open interpreter must allow one to introspect and intercede single language feature instances which include occurrences of linguistic components (e.g., variable declaration, etc.) and instances of non-grammatical language features (e.g., symbol table, garbage collector, etc.). Let us now describe how the proposed model supports this kind of reflection.

Reflection on Linguistic Component Occurrences. Differently from providing reflection on language specification, enabling the introspection of component occurrences in a running application heavily depends on the model or, rather, from the way the interpreter represents the source code. In a tree-based interpreter, the source code is represented by a parse tree built from the language grammar (given by Ps of all $\mathcal{S}_P$ that form the language interpreter). An occurrence of a component $\mathcal{S}_P$ corresponds to a subtree built by P. For example, in Figure 3.1(a) the subtree bounded by the blue box corresponds to an occurrence of the component $\mathcal{S}_P$ defined as:

$\mathcal{S}_P$ = $\mathcal{A}$
P = { Add: E ⇽ E1 "+" E2 }
$\mathcal{A}$ = { Add: { E.val = E1.val + E2.val; }

When the gray node, which corresponds to the head nonterminal of the production in P, is visited, the associated action in $\mathcal{A}$ is executed.

To support reflection on single occurrences, the interpreter must enable one to 1) identify and select the desired occurrences (represented by tree nodes) and 2) to introspect/intercede the identified occurrences. The interpreter must, thus, expose to the developer the parse tree. Single nodes or subtrees could be tracked down by using pattern matching in trees [61]. However, the model does not impose any specific method for occurrence identification.

Regardless of how occurrences (tree nodes) are identified, one must be able to introspect/intercede them. To this purpose, we extend the common execution model illustrated in Figure 3.1(a) as shown in Figure 3.1(b). We borrow some concepts from aspect-oriented programming [74] in order to enable one to execute arbitrary code before and/or after a node is visited. To this purpose, we introduce the concept of hooks which are points in the execution flow where one can attach an arbitrary piece of code to perform reflective operations. In our model, hooks are positioned *before* and *after* each tree node as illustrated by small squares in Fig. 3.1(b). Depending on their position with respect to a node hooks are respectively called *before* and *after* hooks. A piece of code attached at a hook is generically termed as *agent*. It is a self-contained software entity, completely separated from the application source code represented by the tree. This complies with the third requirement for open interpreters as discussed in Section 3.1. Reflection upon construct components and their occurrences can be done through an API provided to agents by the interpreter.

The model does not impose any specific method for injecting agents into hooks. As illustrated in Figure 3.1(b), one can inject more than one agent in a hook. Before a node is visited, the interpreter checks if any agent is present in the *before* hook. If there is none, the node is regularly visited. Otherwise, the agents are executed in the order they were injected into the hook. Their execution is synchronous and the node is visited only after the last injected agent terminates its execution. After a node is visited, the interpreter applies the same execution pattern to run all agents injected in the node's *after* hook. Notice that agents can be injected into hooks at load time, i.e., before the interpreter starts executing any of the semantic phases. The only requirement is that the parse tree is already built.

Agents must also be selectively removable. As with injection, the model does not specify how agent removal should be implemented. However, the model being presented requires that 1) agent injection and removal can occur at any moment of the application execution (i.e., it is asynchronous) 2) agent injection/removal must be atomic operations. The latter means that when the insertion/removal operation begins the interpretation should pause. Then, an agent is either inserted to/removed from all of the desired hooks or the operation fails and no hook is affected. Only after the agent insertion/removal is completed, the interpretation can continue. This gives the interpreter adapters more control over the adaptation process and avoids unexpected behavior due to incomplete agent insertion/removal.

Conceptually, agent notification is event-driven where events are announced when hooks are reached. Agents are event handlers that react to announcements of events to which they previously registered.

With respect to traditional AOP, the concept of hooks is similar to joinpoints, tree patterns are analogous to pointcuts and the agents corresponds to AOP advices. However, our approach targets interpreter-level concepts, while the traditional AOP operates at the application level.

3.2.3. Reflection on Non-Grammatical Language Feature Instances

Recall that non-grammatical language features are components that do not contribute to the language grammar but are, however, necessary for the interpreter to do its job. For example, a symbol table is a non-grammatical language feature that helps the interpreter keep track of various symbols used in the input source code and perform operations on them. A typical use of a symbol table is for storing bindings from variables to their values. Such a data structure has nothing to do with the grammar, however, it is an indispensable component of an interpreter. Non-grammatical components can also be helper classes that implements some functionality of the interpreter. Such classes can also be static, i.e., with no runtime instance. In this section, however, we focus on the reflection of non-grammatical language features that can be instantiated.

When an interpreter is running, non-grammatical components are instantiated and have a dynamic state. For example, the state of a symbol table would be given by the current variable bindings. This state should be made available for introspection and modification. In the previous subsections we introduced the concept of agents which can use a specifically designed API to introspect/intercede the construct component instances. The same technique can be used to reflect upon non-grammatical components and their instances. However, due to the nature of non-grammatical components, it is useful to have a way to execute agents without injecting them into specific hooks. Suppose that we want to change the way the symbol table stores variables. For example, we want it to store variable bindings in a `TreeMap` instead of a `HashMap`. We can code an agent that uses the provided API to replace the current symbol table with a new one. However, there is no point in registering the agent in some specific hook in the parse tree. That hook might even never be reached and the change would never be deployed. Instead, there is a need for an asynchronous one-time execution of the agent that deploys the change. Again, this must be an atomic operation that pauses the interpreter, executes the agent and resumes with the interpreter execution.

The model described so far is purposely rather abstract. It does not impose any specific implementation method, except some constraints on how agent injection/removal and their execution should occur. Also, it is restricted to tree-based interpreters, although it should be fairly straightforward to port the idea on interpreter that use some different code representation as long as it is able to identify language construct occurrences and place hooks before and after them.

3.3. Intercession Operations

In this section we define a minimal set of intercession operations that an open interpreter must provide. For convenience and with a slight abuse of notation we introduce the following auxiliary definitions that we believe can simplify the exposition of intercession operations definition.

– $\mathcal{S}_P(r) = \mathcal{A}_r$, i.e., $\mathcal{S}_P(r)$ denotes the actions defined for P in semantic phase (role) r.

- $\Upsilon_{\mathcal{L}}(p_k) = \mathcal{S}_P$, where $\mathcal{S}_P \in \Upsilon_{\mathcal{L}}$, and $p_k \in P$. In other words, $\Upsilon_{\mathcal{L}}(p_k)$ denotes the component that defines the production p_k.
- $\Upsilon_{\mathcal{L}}(p_k, r) = a_k$, where $\mathcal{S}_P = \Upsilon_{\mathcal{L}}(p_k)$, $\mathcal{S}_P(r) = \mathcal{A}_r$ and $a_k \in \mathcal{A}_r$. In other words, $\Upsilon_{\mathcal{L}}(p_k, r)$ denotes the action defined for p_k in semantic phase (role) r.
- The node currently visited by the interpreter is denoted as η_{p_k}, where p_k is the production whose head nonterminal represents the current node.

In the following we provide a semi-formal definition of the execution model of an open interpreter. This is necessary to define reflection operations on specific occurrences of language components. A slightly simplified, but fully formal definition of the operational semantics is provided in Appendix A.

- By default, when node η_{p_k} is visited in semantic phase (role) r, the action $\Upsilon_{\mathcal{L}}(p_k, r) = a_k$ is executed.
- A node can have a specialized semantic action which overrides the predefined action given by $\Upsilon_{\mathcal{L}}(p_k, r) = a_k$. Let $\nu(\eta_{p_k}, r) = a$ be a mapping from nodes to specialized semantic actions where a is either an action or $null$.
- The semantic action to be executed when node η_{p_k} is visited in role r is therefore given by:
  - $\nu(\eta_{p_k}, r) = a$ if $a \neq null$ or
  - $\Upsilon_{\mathcal{L}}(p_k, r) = a$ otherwise.

We can now define the intercession operations. We divide the discussion in two parts according to the scope of the effects that the operations have on the application execution. System-wide operations globally affect the behavior of running applications. On the other hand, selective modifications alter the behavior of single component instances while leaving the other occurrences of the same type untouched.

System-wide Modification. System-wide modifications update the language or some component specifications. They have the effect of changing the behavior of *all* occurrences of the modified language constructs. Intercession operations that fall into this category are:

- **Component replacement** that is defined as:

$$replace(\Upsilon_{\mathcal{L}}, {\mathcal{S}_P}^{\text{old}}, {\mathcal{S}_P}^{\text{new}}) = (\Upsilon_{\mathcal{L}} \setminus \{{\mathcal{S}_P}^{\text{old}}\}) \cup \{{\mathcal{S}_P}^{\text{new}}\}$$

  Where:

  ${\mathcal{S}_P}^{\text{old}} \in \Upsilon_{\mathcal{L}}$.

Notice that grammar productions denoted with P remain the same in both ${\mathcal{S}_P}^{\text{old}}$ and ${\mathcal{S}_P}^{\text{new}}$. We do not allow the grammar to be changed as that would require one to modify the original source code which would violate the requirements from Section 3.1.

– **Action addition** that is defined as:

$$add(\Upsilon_{\mathcal{L}}, p, r, a) = (\Upsilon_{\mathcal{L}} \setminus {\mathcal{S}_P}^{\text{old}}) \cup {\mathcal{S}_P}^{\text{new}}$$

Where:

$$\Upsilon_{\mathcal{L}}(p) = {\mathcal{S}_P}^{\text{old}}$$
$${\mathcal{S}_P}^{\text{old}}(r) = \mathcal{A}_r \text{ and } a \notin \mathcal{A}_r$$
$${\mathcal{S}_P}^{\text{new}}(r) = \mathcal{A}_r \cup a$$

– **Action removal** that is defined as:

$$remove(\Upsilon_{\mathcal{L}}, p, r, a) = (\Upsilon_{\mathcal{L}} \setminus {\mathcal{S}_P}^{\text{old}}) \cup {\mathcal{S}_P}^{\text{new}}$$

Where:

$$\Upsilon_{\mathcal{L}}(p) = {\mathcal{S}_P}^{\text{old}}$$
$${\mathcal{S}_P}^{\text{old}}(r) = \mathcal{A}_r \text{ and } a \in \mathcal{A}_r$$
$${\mathcal{S}_P}^{\text{new}}(r) = \mathcal{A}_r \setminus a$$

– **Action replacement** can be defined in terms of action removal and action addition.

We do not consider the addition and removal of linguistic components since that would impact the language syntax. Consequently, the original application code would have to be modified which would violate the requirements from Section 3.1.

Notice that the defined operations have no effect if their arguments are inconsistent. For example, if one tries to replace a component that does not exist anymore, nothing will happen (notice the ${\mathcal{S}_P}^{\text{old}} \in \Upsilon_{\mathcal{L}}$ constraint in slice replacement definition). Also, a type system, such as one defined by Neverlang, can prevent unsound modifications of the interpreter.

Selective Modification. This type of adaptation is achieved by defining localized behavior on specific nodes. It is used when one wants to change the semantics of specific occurrences of language components in the program. The operations are defined as follows:

– **Set specialized action**:

$$specialized(\nu, \eta_{p_k}, r, a) = \nu[\eta_{p_k} \mapsto a]$$

Where:

ν is a mapping from nodes to specialized actions and $\eta_{p_k} \mapsto a$ updates the mapping from η_{p_k} to a,
η_{p_k} is the node to be affected,
r is the semantic phase (role) for which the node behavior should be specialized,
a is the specialized semantic action.

- **Action removal**:

$$remove(\nu, \eta_{p_k}, r, a) = \nu[\eta_{p_k} \mapsto a_{empty}]$$

Where:

ν is a mapping from nodes to specialized actions,
η_{p_k} is the node to be affected,
r is the semantic phase (role) for which the node behavior should be specialized,
a_{empty} is an empty action.

In other words, a local removal of an action is equivalent to associated a node with an empty specialized action.

- **Reset node**:

$$reset(\nu, \eta_{p_k}, r) = \nu[\eta_{p_k} \mapsto null]$$

Where:

ν is a mapping from nodes to specialized actions,
η_{p_k} is the node to be affected,
r is the semantic phase (role) for which the node behavior should be reset (unspecialized).

The defined operations represent a minimal set of intercession operations that an open interpreter should implement. The set is rich enough to allow one to tailor the interpreter to her needs for a variety of applications as will be shown in Chapter 5.

3.4. Semantics Adaptation Implications

Changing the semantics at runtime can have serious impact on interpreter execution, especially on its correctness. For example, the new semantics might lead to a broken interpreter, since components might have to provide other components with information needed for a correct execution. For example, if a semantic action is removed (see action removal operation defined in Section 3.3) the parse tree may be missing some attributes that are needed for correct execution. To prevent erroneous usage of intercession operations one can impose and check constraints before changing the semantics. In attribute grammars, one can use one of the many formalisms for ensuring the well-definedness of attribute grammar definitions [75, 124, 4, 66, 21]. This will guarantee that

new semantics correctly defines the required attributes. Alternatively, in an interpreter based on AST class hierarchies the constraints would be given by method signatures (i.e., the interface) which would have to be respected when replacing methods. From here after we will assume that intercession operations respect such constraints.

The feature interaction problem might occur when a dependency between language features is broken due to a change in the language specification. Think of replacing the semantics for the variable declaration with an empty semantic action. Will a `for` loop construct, which depends on the variable declaration to introduce the loop control variable, still work as expected? There would be no syntactic issues since at that point of execution the parse tree is already built and the syntax analysis phase already did its job. However, the interpreter would be broken since the loop control variable would be undefined. The model implementation should, thus, address this issue to prevent erroneous adaptations of the language specification.

Agent interaction issue concerns situations where an agent alters the interpreter in a way that "interfers" with another agent. A possible solution is to notify agents when the interpreter is modified. This way, agents can check whether conditions still hold for them to be active and act accordingly (e.g., remove themselves from hooks, etc.). Similar problems are present in aspect-oriented programming where aspects may interfere with each other or even prevent the execution of other aspects. To prevent these kind of errors, researchers introduced decoupling contracts that specify design rules to which aspects must conform. This allows one to independently change the aspects modulo their conformance to the related contracts [113]. The simples form of decoupling contracts are black-box contracts that specify for each advice's method the relationship between its inputs and outputs while hiding the method's implementation details. Although such constraints contribute to the system integrity, black-box contracts cannot fully prevent aspect interference. For example, if an aspect does not proceed (e.g., by calling `proceed` in AspectJ), none of the aspects that come after would be executed. To solve this issue, Bagherzadeh *et al.* [5] introduced translucid contracts which are based on gray-box specifications [15]. Translucid contracts provide an abstract description of the aspect behavior and thus allow one to better reason about aspect interference and to express stronger assertions about aspect behavior. Open interpreters could similarly introduce decoupling contracts to which agents should comply.

These are important implications and the possible implied issues need to be addressed to favor the adoption of open interpreters. The solutions discussed above are minimalistic and there is certainly room for further research to make open interpreters more useful and user-friendly.

4
Open Interpreters in Neverlang

In this Chapter we describe the main components of the Neverlang's architecture aimed at supporting open interpreters according to the model presented in Section 3.2. We further present a framework-level API provided to support introspection and intercession of an interpreter. We also present a domain-specific language for agent development with special emphasis on user-friendly hook selection. Finally, we discuss the strengths and drawbacks of the presented approach.

4.1. Definition Mapping

Let us first map Definitions 2 and 3 on the Neverlang framework introduced in Chapter 2.2.4. The mapping is summarized in Table 4.1. This will help us identify the Neverlang's concepts that are relevant to open interpreters.

Consider first the Definition 2 concerning the implementation of an interpreter. Conceptually, language components $_\circ k = (syn_k, sem_k)$ correspond to Neverlang slices where syn_k and sem_k correspond to modules that, respectively, define the syntax and the semantics of a language component. Non-grammatical components (i.e., $_\circ k$ where syn_k is empty) correspond to Neverlang's endemic slices which expose globally accessible data structures to semantic actions. Neverlang interpreters are tree-based, hence an occurrence of the linguistic component $_\circ k = (syn_k, syn_k)$ corresponds to a portion of the parse tree that was generated by syn_k. Notice that many subtrees can be generated by syn_k. When the root node of a subtree generated by syn_k is visited, the semantics in sem_k is applied to it.

Now, consider the above elements in the light of Definition 3 concerning open interpreters. A Neverlang interpreter is said to be open if it enables one to introspect

Open Interpreters	**Neverlang**
language feature $_\circ k$	slice
syntax syn_k	module defining the syntax
semantics sem_k	module defining the semantics
$_\circ k$ in which syn_k is empty	endemic slice
linguistic component occurrence of $_\circ k$	subtree generated by syn_k
non-grammatical language feature occurrence	endemic slice instance

Table 4.1.: *Mapping of open interpreter concepts to the Neverlang framework.*

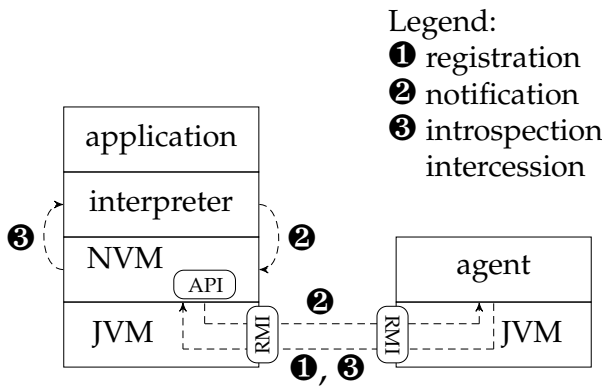


Figure 4.1.: *Neverlang's architecture for open interpreters.*

and modify the specification of modules, slices, endemic slices and of the language definition itself. Furthermore, it must enable one to introspect the parse tree and modify the behavior associated with subtrees (construct occurrences) without affecting the general behavior of the concerned linguistic component. It should also allow one to reflect upon the runtime state of endemic slices.

With this mapping in mind and the goal we want to achieve, let us now consider the Neverlang's architecture that fully supports the requirements for open interpreters.

4.2. The Architecture

We explain the Neverlang's architecture top-down in reference to Figure 4.1 which shows an abstract representation of the main architecture components and their interaction. The stack on the left-hand side corresponds to the stack in Figure 2.4 which depicts the Neverlang's compilation, loading and the execution process. An application written in the interpreted language is executed by a Neverlang-based interpreter which runs on top of the the Neverlang Virtual Machine (NVM). In turn, NVM runs on top of the Java Virtual Machine (JVM). The running interpreter can be introspected and/or modified by an agent, which is a Java object that implements the interface in Listing 4.1. An agent is a self-contained object and runs on a separate JVM as illustrated on the right-hand side of Figure 4.1. The reflection code is thus completely separated from the

```
import java.rmi.Remote;
import java.io.Serializable;
public interface IAgent extends Remote, Serializable {
  public void before(IPatternMatch __pmCtx) throws RemoteException;
  public void after(IPatternMatch __pmCtx) throws RemoteException;
}
```

Listing 4.1: *Agent's interface.*

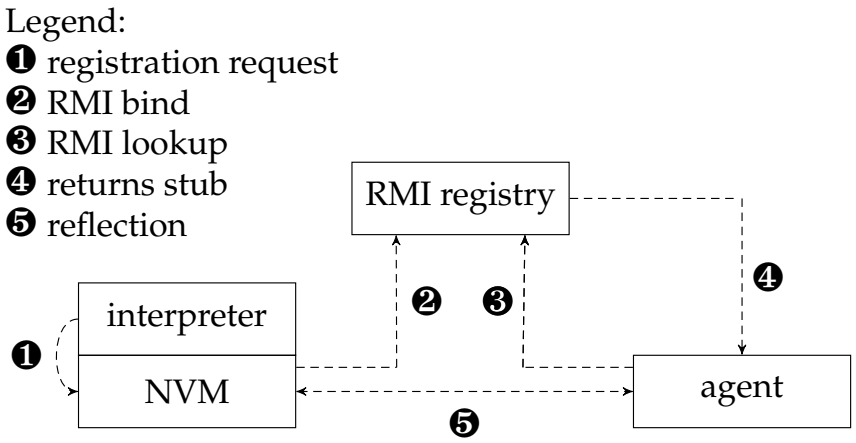


Figure 4.2.: *RMI registration and lookup process.*

application code, which satisfies the third requirement from Section 3.1. Agent are able to communicates with the interpreter through the API exposed by the NVM. There is no restriction to the number of agents that can communicate with an interpreter. For simplicity, Figure 4.1 shows just one agent.

The communication between agents and the interpreter is based on Remote Method Invocation (RMI). Notice that the model in Section 3.2 does not impose any particular communication method. The adoption of RMI is an implementational choice motivated by the fact that RMI is well-known, robust, secure and allows the interacting entities to reside either on the same machine or in a network. At the moment of execution, the NVM registers an open interpreter in the RMI registry (`rmiregistry`) which is a naming service that clients can use to look up for remote objects and call their remote methods. In our case, it is used by agents to find running open interpreters and to interact with them. The method of interaction between an agent and an interpreter depends on the target object upon which the agent will perform reflective operations. In the following section we discuss how reflection can be performed on constituent parts of an interpreter as defined in Definition 2. Figure 4.2 shows the RMI registration and the lookup process. First, the interpreter asks the NVM to be registered in the RMI registry (❶). NVM uses the Java RMI library to perform the RMI binding (❷). The agent can then lookup in the registry for an interpreter instance (❸). As a lookup result, the RMI registry returns a stub of the running interpreter (❹). The stub is an interface that defines which methods can be remotely invoked by the agent to reflect upon the interpreter (❺).

4.3. Reflection on Open Interpreters

Before an agent can perform any reflection operations, it must obtain a reference to the target interpreter instance. To simplify this process, Neverlang provides the developer

```
1  public abstract class Agent implements IAgent {
2    protected OpenNeverlang interpreter;
3    protected static OpenNeverlang init(String[] args) throws RemoteException,
        NotBoundException {
4      if (args.length < 1) {
5        System.out.println("Please provide the name of the interpreter's instance.");
6        System.exit(1);
7      }
8      Registry registry = LocateRegistry.getRegistry();
9      return (OpenNeverlang) registry.lookup(args[0]);
10   }
11 }
```

Listing 4.2: *`Agent` class provided by the Neverlang support library.*

with a basic agent class implementation shown in Listing 4.2. In lines 8 and 9, the `init` method retrieves from the RMI registry a reference to the previously registered interpreter instance bound to the name stored in `args[0]` (lines 8 and 9). This abstract class can be extended with customized agent behavior as illustrated in the following.

4.3.1. Reflection on Language Specification and Non-grammatical Components

The Neverlang VM exposes an API that allows agents to introspect and modify the language specification as illustrated in Listing 4.3. The agent called `BeforeAfter` extends the `Agent` abstract class defined in Listing 4.2 and uses the inherited `init` method to obtain a reference to the interpreter's instance (line 5). The developer can then use the `interpreter` variable to access the Neverlang's reflection API (❸ in Figure 4.1), which will be described in detail later in this chapter. In our example, the agent invokes `interpreter.getSlices()` to obtain the list of slices that form the interpreted language (line 11). When a method of the reflection API is invoked (e.g., `interpreter.getSlices()`), the NVM will pause the interpreter execution until the invoked method returns the control flow to the agent.

In RMI, objects that are shipped between the two communicating entities are serialized. Consequently, objects are sent by value and not by reference. While this would be fine for introspection, it would present a serious obstacle for intercession. Indeed, any agent-side change on the shipped object would not be reflected in the interpreter. Fortunately, RMI allows one to define stubs that act as client's local representatives for remote objects. This is exactly the same mechanism we use in the first place to make the agents-interpreter communication possible. In practice, this means that the objects that should be made available for intercessions must implement the `java.rmi.Remote` interface. However, we avoid declaring Neverlang's core data structures as `Remote` since that would expose them to uncontrolled reflection. As a matter of fact, the definition of open interpreters (Definition 3) requires that reflection be done *"in a controlled way through an interface"*. Therefore, we define a set of wrapper classes that act as stubs that agents can use to modify the interpreter. These classes provide an interface with a reduced

```
1  package reflection.test;
2  public class LangSpecAgent extends Agent {
3    int customVariable = 1;
4    public static void main(String[] args) throws RemoteException, NotBoundException {
5      OpenNeverlang interpreter = Agent.init(args);
6      if (interpreter == null) {
7        System.out.println("Error connecting to the interpreter instance.");
8        System.exit(1);
9      }
10     // use Neverlang's API to introspect/intercede the interpreter
11     SliceInfo[] slices = interpreter.getSlices();
12     ...
13   }
14   @Override
15   public void before(IPatternMatch __pmCtx) throws RemoteException {
16     // use Neverlang's API to introspect/intercede the interpreter
17   }
18   @Override
19   public void after(IPatternMatch __pmCtx) throws RemoteException {
20     // use Neverlang's API to introspect/intercede the interpreter
21   }
22 }
```

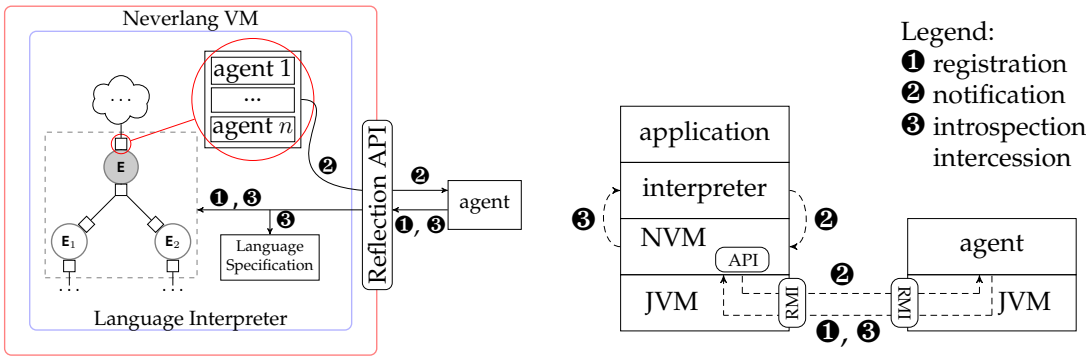
Listing 4.3: *Agent performing reflection operations on language specification.*

set of methods to prevent the agent from arbitrarily modifying the interpreter. These classes follow the `-Info` naming convention (e.g., `SliceInfo` in Listing 4.3), where the `Info` suffix conveys the idea of something non-modifiable. Indeed, these objects cannot be modified, but instead can only trigger requests for modification. The same approach is used to introspect and modify non-grammatical components which in Neverlang correspond to endemic slices. For example, `getSlice(String name)` returns an object of type `SliceInfo` which provides additional methods for introspection/intercession.

4.3.2. Reflection on Linguistic Component Occurrences

Conceptually, reflection on occurrences of a linguistic component is a two-step process: 1) the agent identifies the desired occurrences and 2) it performs reflection operations on them. As per model described in Section 3.2, the second step is further divided into agent injection into hooks and agent execution when these hooks are reached. We now explain how this process is implemented in Neverlang. We begin from a more abstract description of the agent injection process, which in Neverlang is called agent registration. Later, we dig into details by explaining how occurrences are identified.

Agent Registration and Notification. The agent registration process is illustrated in Figure 4.3(a), beside which we put the architecture representation from Figure 4.1. The numbers in both figures correspond to the same steps. This should help the reader grasp the whole picture of the architecture. The reflection on linguistic component



(a) Runtime representation of open interpreter's execution

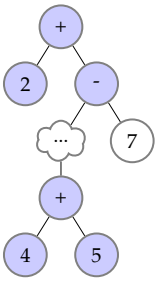
(b) Component interaction in open interpreters

Figure 4.3.: *Neverlang's architecture for open interpreters.*

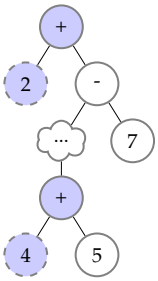
occurrences is based on the listener-notifier schema. As was illustrated in Listing 4.3, the agent must first acquire a reference to the interpreter. Then, the agent can register itself at the desired hooks (❶), represented by small squares in Figure 4.3(a), in order to be notified when such hooks are reached during the tree visit. The registration is asynchronous, i.e., it can be done at any moment of the interpreter execution. However, once the process of agent registration starts, the interpreter execution stops until the registration is finished. In other words, the registration is an atomic operation which avoids unexpected behavior that could have arisen from uncontrolled or incomplete registration. With respect to the model in Section 3.2, agent registration corresponds to agent injection and agent notification matches the agent execution. Depending on the hook position with respect to a node, the notification (❷) corresponds to calling through RMI either the `before` or `after` method of the `IAgent` interface shown in Listing 4.1. This call implicitly triggers the shift-up operation as it transfers the execution from the interpreter to the agent, i.e., from base- to meta level. At this point, the agents can use the NVM's reflection API to introspect and/or modify (❸) the language specification, component instances and the execution state of the interpreter. The shift-down operations is implicitly performed when the agent returns from the `before` or `after` method.

Neverlang enables one to inject agents into hooks at applications load time, i.e., when the interpreter loads an application, but before it is actually interpreted. All Neverlang-based interpreters accept an optional parameter to list agents that are to be injected after the parse tree is built. The user can also specify the semantic phase before which the agent injection must take place.

Occurrence Identification. As per model described in Section 3.2, occurrences of a linguistic component correspond to portions of a parse tree. These subtrees are built from grammar rules defined in the specification of linguistic components. Therefore, to identify subtrees of interest we can use tree patterns expressed in terms of grammar



(a) Nodes matched by the pattern in line 3 in Listings 4.4.



(b) Nodes matched by the qualifier **`after`** in Listing 4.9.

Figure 4.4.: *Node selection with pattern matching. To simplify node identification, nodes are labeled with arithmetic symbols and numbers, instead of nonterminals of their original grammar production.*

productions and/or nonterminals. Neverlang provides developers with a library for selecting nodes with tree patterns.

We illustrate the approach on the parse tree in Figure 4.4(a), parts of which were built by the grammar rule `E ← E "+" E`. Please notice that, in Figure 4.4, we labeled nodes with arithmetic symbols (+ and −) and numbers, instead of with the nonterminals of their original grammar productions. This notation should simplify node identification. With the provided reflection API the developer ask the Neverlang VM to identify subtrees that were built by the grammar rule for the addition operation (`E ← E "+" E`). The NVM would identify nodes that in Figure 4.4(a) are colored blue, where "+" nodes corresponds to the head nonterminal of the production and children nodes correspond to the two `E` nonterminals in the production body. Listing 4.4 illustrates how an agent can express tree patterns. Line 3 expresses a simple pattern in terms of the production labeled `Add` which is defined in module `mylang.AddSyntax` in Listing 2.3. Once passed to the NVM for node identification, this pattern would match the blue nodes in Figure 4.4(a), as described above. Neverlang provides binary operators to support complex patterns. For example, in lines 4-6 we express a complex pattern that would match portions of the tree where the addition operation precedes the subtraction operation. Writing patterns in terms of Java objects organized into hierarchies can quickly become cumbersome and error-prone. For this reason, we developed a platform DSL, called μDA, which allows one to express complex patterns in a more user friendly way. The DSL is explained in Section 4.4.

Hook selection Before an agent can interact with the interpreter, it has to register itself for hook notifications. We explain the process of hook selection and of the subsequent agent registration with reference to the example we introduced in Listing 4.4. The agent can request to be registered by invoking the `register` method as in line 8. The `register` method takes in input four parameters. The first is a reference to the agent itself. The NVM will use this reference to invoke the `before` and/or `after` RMI methods on the

```
1  public class MyAgent implements IAgent {
2   public void register() {
3    ITreePattern simplePattern = new ProductionPattern("Add", "mylang.AddSyntax");
4    ITreePattern complexPattern = new PrecedesPattern(
5          new ProductionPattern("Add", "mylang.AddSyntax"),
6          new ProductionPattern("Sub", "mylang.SubSyntax")
7    );
8    interpreter.register(this, simplePattern, Hook.BEFORE, "evaluation");
9   }
10  @Override
11  public void before(IPatternMatch __pmCtx) throws RemoteException {
12   // use Neverlang API here
13  }
14  @Override
15  public void after(IPatternMatch __pmCtx) throws RemoteException {
16    // use Neverlang API here
17  }
18 }
```

Listing 4.4: *Sample agent class.*

agent object (agent notification). The second parameter is a tree pattern which will be used by the NVM to match the nodes of interest. The third parameter specifies the hook at which the agent should be registered. The final parameter specifies the role (semantic phase) of interest, for example, type-checking, evaluation, etc. Indeed, agent registration and notification is done per role, i.e., an agent will be notified when selected hooks are reached only during the role for which it registered. By invoking the `register` method in line 8, the agent would register itself at the *before* hook of all nodes involved in the addition operation. The `register` method proceeds in two steps: 1) it selects all nodes that match the pattern and 2) it registers the agent at the specified hook of the nodes that were selected in step 1. Consider the already discussed tree in Figure 4.4(a). The pattern in line 3 would match the six colored nodes. The NVM would then register the agent at the *before* hook of these nodes. Hence, before any of the selected nodes is visited, the NVM would invoke the `before` method of the registered agent.

Dynamic Constraints. Neverlang additionally supports dynamic constraints that allow one to specify optional conditions that must be satisfied in order for an agent to be notified. This mechanism is useful for developing context-aware interpreters, debuggers, etc., as we will illustrate in Chapter 5.

Dynamic constraints are an optional parameter of tree patterns, although they do not influence the selection of hooks. Instead, they affect the notification of agents when these hooks are reached, i.e., if provided, a constraint determines whether the registered agent should be notified. If the constraint is not provided or it is satisfied, the agent is notified, otherwise, the notification for that agent is skipped.

A dynamic constraint is a class that implements the interface in Listing 4.5. The

```
1 import java.rmi.Remote;
2 import java.io.Serializable;
3
4 public interface IDynamicConstraint extends Remote {
5   public boolean isSatisfied(IPatternMatch __pm) throws RemoteException;
6 }
```

Listing 4.5: *`IDynamicConstraint` interface.*

```
1  import java.rmi.Remote;
2  import java.io.Serializable;
3
4  public class ConstraintsExample implements IDynamicConstraint {
5    public boolean isSatisfied(IPatternMatch __pm) throws RemoteException {
6      NodeInfo currentNode = __pm.getCurrentNode();
7      AttributeInfo leftChildAttr = currentNode.getChild(0).getAttribute("value");
8      return (int)leftChildAttr.value == 2;
9    }
10 }
```

Listing 4.6: *Example of using dynamic constraints.*

`isSatisfied` method must implement the constraint logic which returns a boolean value depending on whether the constraint holds or not. This is an RMI method that the interpreter will invoke to verify if the agent should be notified about the reached hook. The method takes in input an object holding the information about the nodes that match the specified pattern. This information can be used to introspect the tree structure and its state, in addition to other reflection methods that are provided by the reflection API.

Since dynamic constraints are specified in tree patterns, one can express complex conditions on a tree pattern to trigger agent notification. Listing 4.6 shows a constraint that will be evaluate to true only when the left operand of the addition operation has the `value` attribute set to 2. With reference to Figure 4.4(a), the constraint would trigger the agent notification only on the root node and its immediate children. The bottom three nodes would fail to satisfy the constraint since the left operand is 4, instead of 2 as required by the constraint.

4.3.3. Reflection API

Depending on the object it targets, the reflection API can be used in two ways. If we target the language specification or non-grammatical components (endemic slices), we can simply obtain a reference to the interpreter (see Listings 4.2 and 4.3) and invoke the desired methods on the interpreter object. On the other hand, if the agent targets linguistic component occurrences it must first register at the desired hooks and, once those hooks are reached, the agent is notified and obtains the execution control. At this point, it can use the framework-level API to introspect and intercede the interpreter.

A subset of the API methods is shown in Tab. 4.2. We only show the most interesting methods, especially those that are relevant for the examples discussed in

Introspection
Execution State
`NodeInfo getTree() throws RemoteException` *Returns the tree representation of the entire input program.*
`NodeInfo getCurrentNode() throws RemoteException` *Returns the tree representation of the subtree rooted at the current node.*
`SemanticActionInfo getAction() throws RemoteException` *Returns information on the action which, depending on the hook position, will be or was executed.*
`ProductionInfo getProduction() throws RemoteException` *Return the information on the production use to build the subtree rooted at the current node.*
`RoleInfo getRole() throws RemoteException` *Returns information on the current semantic phase.*
Interpreter Implementation
`ProductionInfo[] getGrammar() throws RemoteException` *Return all productions of the language grammar.*
`RoleInfo[] getRoles() throws RemoteException` *Returns the information on roles, i.e., semantic phases.*
`SliceInfo[] getSlices() throws RemoteException` *Returns the information on language slices.*
`EndemicSliceInfo[] getEndemicSlices() throws RemoteException` *Returns the information on language endemic slices.*
Intercession
`void setSpecializedAction(NodeInfo node, SemanticActionInfo action, String role) throws RemoteException` *For a given node sets the provided action as a specialized action in a given role. For a given node, the specialized action overrides the default action in the language specification.*
`void resetNode(NodeInfo node, String role) throws RemoteException` *A given node is reset to its original state, i.e., any specialized action is removed and a possibly removed action is restored.*
`void redoRole(String role) throws RemoteException` *Re-executes the specified role from the root node.*
`void replaceSlice(String oldSlice, String newSlice) throws RemoteException` *Replaces `oldSlice` with `newSlice`. Slice are provided with their canonical name.*
Various
`ITreePattern compilePattern(String mdaPatern)` *Accepts a μDA pattern as a string and transforms it to an instance of `ITreePattern`.*

Table 4.2.: *A subset of the Neverlang's API interface.*

Chapter 5. The reflection API provides the "`get-`" methods for all constituent parts (as per Definition 2) of a Neverlang-based interpreter. These methods return `-Info` objects (introduced in Section 4.3.1) with meta information about the queried object. For example, `getCurrentNode` returns a `NodeInfo` object with information about the node that is currently being visited by the interpreter. The API exposed by the NVM satisfies the first requirement in Section 3.2.

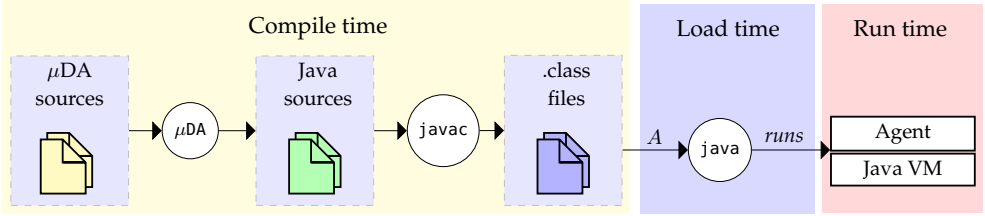


Figure 4.5.: *Agent compilation process: from μDA code to a running agent.*

4.4. μDA: a Platform DSL for Open Interpreters

The Neverlang's built-in library for tree patterns is quite powerful and enables one to write complex patterns, that may include dynamic constraints, however, expressing patterns in terms of Java objects organized into hierarchies can quickly become cumbersome and error-prone. Hence, on top of the Neverlang's API we developed a special DSL, called μDA[1], for expressing patterns in a user-friendly and intuitive way.

Listing 4.7(a) shows an example of an agent written in μDA. The code translates to the already discussed Java code shown in Listing 4.3 which for easier reference is put below the example μDA code. The process of building an agent through μDA is illustrated in Figure 4.5. A μDA file is first compiled by the μDA compiler to Java code similar to the one in Listing 4.3. Then the standard process of Java compilation and execution follows. The running agent can then communicate with the interpreter as illustrated and shown in Figure 4.3.

The DSL, summarized in Tab. 4.4, is best introduced by an example. With reference to listing Listing 4.7, we see that an agent is given a name and an optional namespace or Java package. In our example, the agent `LangSpecAgent` belongs to the `reflection.test` package. μDA agents can declare custom variables (line 2) which are translated to class fields (line 11). The agent body can define three types of blocks: **`before`**, **`after`** and **`system-wide`** (lines 3-7). **`before`** and **`after`** will be explained later when discussing Listing 4.8. These blocks can contain arbitrary Java code which can use the reflection API provided by the Neverlang framework. The `interpreter` variable (line 5) is reserved and always references the interpreter with which the agent is communicating through RMI. The **`system-wide`** qualifier is used for the code that is not meant to be executed before or after specific tree nodes but, instead, it is meant to be executed once, i.e., when the agent is instantiated and obtains a reference to an interpreter. Typically, this is used for (but not limited to) reflection on the language specification or the runtime state of the interpreter and its non-grammatical components. As illustrated in Figure 4.5, the μDA compiler will translate the example μDA code into Java code shown in Listing 4.7(b). Notice that any code defined in the system-wide code block is simply put in the `main` function of the generated agent and will be executed after the agent successfully obtains a reference to a running interpreter. The `before` and `after` methods are automatically generated to implement the `IAgent` interface. The comments

[1]The name μDA comes from "microlanguage dynamic adaptation". Microlanguages are discussed in Section 4.5.

a) Example μDA code.

```
agent reflection.test.LangSpecAgent {
  int customVariable = 1;
  system-wide {
    // use Neverlang's API to introspect/intercede the interpreter
    SliceInfo[] slices = interpreter.getSlices();
    ...
  }
}
```

b) Corresponding auto-generated Java code.

```
package reflection.test;
public class LangSpecAgent extends Agent {
  int customVariable = 1;
  public static void main(String[] args) throws RemoteException, NotBoundException {
    OpenNeverlang interpreter = Agent.init(args);
    if (interpreter == null) {
      System.out.println("Error connecting to the interpreter instance.");
      System.exit(1);
    }
    // use Neverlang's API to introspect/intercede the interpreter
    SliceInfo[] slices = interpreter.getSlices();
    ...
  }
  @Override
  public void before(IPatternMatch __pmCtx) throws RemoteException {
    // use Neverlang's API to introspect/intercede the interpreter
  }
  @Override
  public void after(IPatternMatch __pmCtx) throws RemoteException {
    // use Neverlang's API to introspect/intercede the interpreter
  }
}
```

Listing 4.7: *μDA agent translation to Java.*

in the code are manually added to the listings as documentation for the reader.

After an agent is compiled to a `.class` object file, it can be executed to perform reflection on a running interpreter. Notice that neither the μDA nor the Java code contain any name of a particular interpreter instance. The target interpreter will be determined at agent load time based on the command line arguments passed to the agent (stored in `args`). Furthermore, in the specific example, neither the μDA nor the Java code are specifically related to any target language developed with Neverlang. Indeed, the reflection code targets the Neverlang's framework-level concepts, instead of concepts of the language being interpreted. Consequently, an agent can potentially be shared across different language implementations. In combination with the Neverlang's ability to share slices across different language implementations, this feature presents a powerful approach to language engineering. Think, for example, of an endemic slice that represents a symbol table. If the behavior of a symbol table is the same for languages Π_1 and Π_2, we can develop one endemic slice and use it in both language

*a) µDA code for an agent with **before** and **after** qualifiers.*

```
// import custom libraries
import mylang.HelperMethods;
// define bindings
production addition : Add from module mylang.AddSyntax;
nt additionHead,left,_ : Add from module mylang.AddSyntax;
agent reflection.test.BeforeAfter {
  before addition {
    // can use custom imported libraries
    System.out.println("Before: " + addition);
  }
  after additionHead {
    // can use custom imported libraries
    System.out.println("After: " + additionHead);
  }
}
```

b) Corresponding auto-generated Java code.

```
package reflection.test;
import mylang.HelperMethods;
public class BeforeAfter extends Agent {
  public static void main(String[] args) throws RemoteException, NotBoundException {
    // interpreter initialization code omitted
    ...
    String semanticRole = args[1]; // e.g., evaluation, type-checking
    ITreePattern pattern1 = new ProductionPattern("Add", "mylang.AddSyntax");
    ITreePattern pattern2 = new NonterminalPattern("additionHead", "mylang.AddSyntax",
       "Add", 0);
    interpreter.register(this, pattern1, Hook.BEFORE, semanticRole);
    interpreter.register(this, pattern2, Hook.AFTER, semanticRole);
  }
  @Override
  public void before(IPatternMatch __pmCtx) throws RemoteException {
    // can use custom imported libraries
    NodeInfo[] addition = __pmCtx.getNodesInfo("addition");
    System.out.println("Before: " + addition);
  }
  @Override
  public void after(IPatternMatch __pmCtx) throws RemoteException {
    // can use custom imported libraries
    NodeInfo additionHead = __pmCtx.getNodeInfo("additionHead");
    System.out.println("After: " + additionHead);
  }
```

Listing 4.8: *Translation of tree patterns from µDA to Java.*

implementations. Suppose that someone develops a more efficient symbol table for Π_1. To perform a runtime replacement of the old symbol table with a new one, the developer can write an agent. Since, the two languages Π_1 and Π_2 share the same implementation of the original symbol table and supposed that the more efficient implementation is also compatible with Π_2, one can use the same agent to deploy the

change to the interpreter for Π_2.

The μDA DSL provides constructs for expressing tree patterns in a more user-friendly way with respect to writing them directly in Java. Listing 4.8(a) shows an μDA agent whose translation results in the Java code shown in Listing 4.8(b). The μDA code presents a few features that are worth to be emphasized. An μDA script can import custom Java classes that can be used the agent code (e.g., `mylang.HelperMethods` in line 2 in Listing 4.8(a)). After the "import" section we can bind identifiers to language concepts, like grammar productions or nonterminals. The binding will be explained in reference to the grammar production labeled `Add` in the `mylang.AddSyntax` module from Listing 2.3. The statement in line 4 would bind the identifier `addition` to the *whole production* labeled `Add` in the `mylang.AddSyntax` module. The statement in line 5 would instead "unpack" single *nonterminals* and bind them to the respective identifiers. For example, `head` would be bound to the head nonterminal `Expr` of the addition production, while `left` would be bound to the first `Expr` nonterminal in the production body. The underscore has the accustomed meaning of "ignore". The bound identifiers are then used to express patterns to identify nodes at which to perform reflective operations. For example, **`before`** `addition` (line 7) would match all "before" hooks on all nodes involved in the addition operation. With reference to Figure 4.4(a), the pattern would match the blue-colored nodes. Instead, **`after`** `additionHead` (line 11) would match only the nodes that correspond to the head nonterminal of the addition operations.

Listing 4.8(b) shows the generated Java code. The code that in Listing 4.8(a) is enclosed by the **`before`** and **`after`** qualifiers goes into the respective methods of the `IAgent` interface. The identifiers used in the pattern can be used in the code as they are automatically bound to the relative `NodeInfo` descriptors of matched nodes (lines 30 and 36). For example, the `additionHead` identifier used by the **`after`** qualifier (line 11) is translated into a `NodeInfo` object whose value is obtained from the pattern match context variable `__pmCtx` (line 36). The `addition` identifier (line 7), instead, is bound to a production which is made of many nonterminals and, hence, it matches several nodes. Therefore, in line 30, it is bound to an array of `NodeInfo` objects. The patterns are translated into suitable classes that implement the `ITreePattern` interface, namely to `ProductionPattern` for the **`before`** qualifier (line 22) and to `NonterminalPattern` for the **`after`** qualifier (line 23). The patterns are then use to register the agent at the *before* and *after* hooks of the matched nodes (lines 24 and 25). When these hooks are reached, the agent will get the execution control at either the `before` or `after` method,

```
nts head, left, _ : Add from module mylang.AddSyntax;
agent reflection.test.ComplexPatterns {
  after head < left[val==4] | head {
    ...
  }
}
```

Listing 4.9: *μDA code with complex patterns that include dynamic constraints.*

depending on the hook position with respect to the node.

So far it seems that μDA does not bring much benefit over writing agents directly in Java. The strength of μDA becomes evident when it is necessary to express complex patterns which potentially include dynamic constraints. Writing such patterns in Java is cumbersome and error-prone. Consider, for example, the μDA code in Listing 4.9. The pattern expression is composed of a tree pattern (`after head < left[val==4] | head`), a dynamic constraint (`left[val==4]`) and of a filter (`| head`). The first uses the < matching operator (see Tab. 4.4 for the description) and would match all colored nodes in Figure 4.4(b). However, the filter would filter out the dashed nodes and retain only those matched by `head`. Hence, the agent would be registered at *after* hooks of all "+" nodes. However, during the tree visit, the NVM would notify the agent only when the dynamic constraint is satisfied. For example, at the *after* hook of the root ("+") node, the NVM would check if the node referred to by the identifier `left` satisfies the constraint. Since the `val` attribute of the left node equals to 2 the constraint would not be satisfied and the agent would not be notified. The execution would proceed until the second "+" node is reached. This time, the NVM will notify the agent since the constraint `[val==4]` on the node's `left` child is satisfied. Expressing such patterns in Java is time-consuming and can quickly become human-unreadable and in, the long term, unmaintainable.

The `before` and `after` qualifiers can list more than one pattern to register the same agent at nodes of different type. The comma (",") operator is used to separate patterns from each other. Listing 4.10 shows an example in which the same agent is registered before the addition and the subtraction nodes (line 6). The listing shows how the agent can verify at which node it was notified (line 8). The `current` node is checked against the `add` identifier which, as explained above, is bound to the `NodeInfo` object with the same name. Therefore, if `add` and `current` point to the same object, the agent

```
1  import neverlangJS.utils.StackTrace;
2  nt addition,_,_ : Add from module mylang.AddSyntax;
3  nt subtraction,_,_ : Sub from module mylang.SubSyntax;
4  agent adapt.AddSub {
5    CustomData data = new CustomData();
6    before add, sub {
7      NodeInfo current = interpreter.getCurrentNode();
8      if (current == add) {
9        // addition specific behavior
10           customData.setValue("someValue", 1);
11     } else {
12       // subtraction specific behavior
13           int someValue = customData.getValue("someValue");
14     }
15     // common behavior
16   }
17 }
```

Listing 4.10: *The comma "," operator.*

was notified at the node identified by `add`. This kind of patterns are useful when an agent must gather some information at one type of node (e.g., as in line 10) and use it at another type of node (e.g., as in line 13). We will illustrate this by an example in Section 5.5.

As we will illustrate in Section 5.1, sometimes it is useful to generate tree patterns dynamically. To this purpose, open interpreters define the `compilePattern` API method (see Table 4.2). The method takes in input an μDA pattern as a string and dynamically generates an object that implements the `ITreePattern`. This is especially useful when we need to identify tree nodes without registering at a particular hook. For example, we can use this method in the `system-wide` block to obtain a reference to linguistic component occurrences on which we want to perform some reflection operations.

4.5. Microlanguages

The usage of open interpreters is somehow hindered by the challenge to identify components of the interpreter that should be changed in order to achieve the desired behavior. This is especially difficult since a behavior to be modified is in most cases implemented in terms of many linguistic constructs. To face this challenge, Chitchyan *et al.* [36] introduced the concept of microlanguages. Later, Cazzola *et al.* [19] presented the architecture of a support framework to use microlanguages in a user friendly way.

Recall from Section 2.1.2 that a language feature represents the minimal distinguishable meaningful concept of a language (e.g., variable declaration, method invocation, etc.). Similarly, from the application user's point of view, an application feature presents the minimal distinguishable meaningful concept. Czarnecki and Eisenecker [39] define an application feature as "a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept." In simple terms, application features describe the various functionality that are meaningful for the application user. The challenge is expressed in the question: what language features must be changed in order to modify one or more application features so that the application as whole have the desired behavior? This is where microlanguages come into play.

A microlanguage is a logical concept that associates an application feature with the language features (or constructs) that are used to implement it. In simple terms, a microlanguage identifies the set of linguistic constructs that were used by the developer to develop the portion of the application one wants to modify. Listing 4.11 shows a Java class that implements a bank account. We can identify several application features which for simplicity in this example coincide with class methods. Table 4.3 show how application features are related to language features. For example, the application feature for opening a bank account coincides with the `BankAccount` constructor. The constructor uses two language features of the Java language: object field assignment (e.g., `this.balance = ...`) and reference (or variable) access (e.g, `balance` on the right-hand side of the field assignment in `this.balance = balance`). Changing the semantics of any of these language features will affect the behavior of the associated applica-

```
public class BankAccount
{
  private double balance;
  private double interest;
  public BankAccount(double balance, double interest)
  {
    this.balance = balance;
    this.interest = interest;
  }
  public void deposit(double amount)
  {
    balance = balance + amount;
  }
  public void withdraw(double amount)
  {
    balance = balance - amount;
  }
  public void addInterest()
  {
    balance = balance + balance * interest;
  }
}
```

Listing 4.11: *Bank account class.*

No.	Application Feature	Language Features
1	Open account	field assignment, reference access
2	Deposit money	field assignment, reference access, addition
3	Withdraw money	field assignment, reference access, subtraction
4	Add interest	field assignment, reference access, addition, multiplication

Table 4.3.: *Microlanguages for the* `BankAccount` *class.*

tion feature. Notice that application features might not coincide with class methods. They are implementation-dependent, although ideally they will correspond to some modularization unit of the programming language used to develop the application.

As explained by Cazzola *et al.* [19], microlanguages have a two-fold purpose: 1) they identify language features used to implement an application feature and 2) with a proper framework support they help the developer to confine the effects of the language feature modification to a specific application feature. In simple terms, they allow one to modify languages in a controlled way to achieve the desired application-level behavior. In [19] we introduced a framework for dynamic software updating through microlanguages whose architecture is shown in Figure 4.6. The foundations of the architecture are represented by the framework for modular language development (❶) which provides the necessary modularization support for microlanguages. This framework is used to build an interpreter (❷) which runs an application (❻) whose

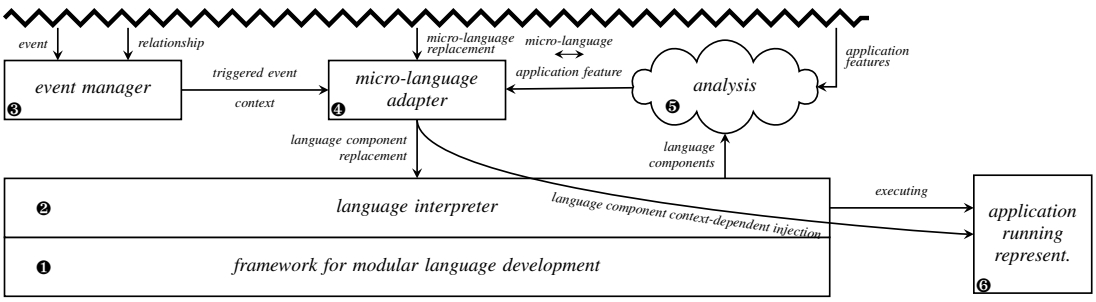


Figure 4.6.: *Architecture for Micro-Language Based Adaptation of a Running Application.*

application features one wants to modify. The analysis activity (❺) is responsible for identifying microlanguages, i.e., for associating language- and application features. At the current state, the analysis activity is manually done by the developer itself, although an automatic mechanism is currently under investigation. Once the microlanguages are defined, the application can be modified by changing the appropriate language features. This task is entrusted to the microlanguage adapter (❹). The adaptation process is triggered by the event manager (❸).

Microlanguages greatly assist the developer in using open interpreters as they bridge the gap between the application- and language features. For further details the reader is referred to [19].

4.6. Discussion

Open interpreters, as implemented in Neverlang, bring several advantages over traditional reflection. However, as with any approach, advantages come at a cost and often have limits. The main advantage of the approach is that the reflection capabilities are provided for free to any Neverlang-based interpreter. This is possible since the reflection API targets framework-level concepts (tree nodes, semantic actions, grammar productions, slices, etc.) which are unaware of (or non-specific to) the target language they implement. As separate software entities, agents are reusable and can be potentially shared across different language implementations. For example, if languages Π_1 and Π_2 share the same language component c_k and an agent is written to act upon c_k in Π_1, then it can also be used to introspect/intercede c_k in Π_2. Agents can also be reused even when there is no shared component among the two languages. In Listing 4.3 we showed an agent that can introspect slices independently of the developed language. However, these advantages comes at a cost: in order to introspect and/or modify the running interpreter, the developer should be a language engineer familiar with Neverlang. This drawback is somehow mitigated by the fact that developers that use reflection usually do and must have some knowledge about the language they try to intercede. Furthermore, once familiar with Neverlang, the developer can introspect/intercede *any* Neverlang-based language interpreter, instead of learning reflection for

each one of them, as done with traditional reflection approaches. The microlanguages framework further mitigates this drawback as it assists the developer in the interpreter adaptation process, which becomes more transparent and smooth.

Another drawback is that developers should have to adopt, in the specific case, Neverlang, which as many young academic projects lacks community support. Tooling support (IDE, debugger) are under development to make Neverlang more attractive for language developers.

The major drawback which is not Neverlang-specific, but related to open interpreters, is the feature interaction problem. Indeed, an agent might modify the interpreter in such a way that another agent, registered at the same hook, will not work as expected. To alleviate this problem, an agent can ask to be notified when the interpreter is modified by another agent. That allows agents to check if conditions still hold for them to be registered and act accordingly (unregister, change hooks, etc.). Nevertheless, there is a need for further studies to alleviate the feature interaction problem.

Also, dynamic modifications of the language implementation could lead to a broken interpreter. For example, a semantic action might be replaced with an action that does not define all the needed grammar attributes. Fortunately, Neverlang's type- and inference system is able to capture missing or wrong attribute definitions. Thus, to some degree it is able to prevent modifications that would lead to an incomplete interpreter implementation. However, it is currently unable to capture dynamic aspects, like a missing symbol table update, a problem scheduled to be solved in the future.

Counter-intuitive adaptation can harden the understanding and maintenance of the running application. Furthermore, in the long-term, continuous runtime language evolution might augment the gap between the original- and the modified application semantics. There is no technical solution to this problem but, instead, it is expected that the language engineer make sane modifications that maintain the expected and intuitive semantics. In Section 5.3, we show an example in which we dynamically change a `for` loop from sequential to parallel execution, which has clear performance benefits, while maintaining the expected results and semantics.

Context Definition
`[endemic] slice «id1» [, «id2», ...] : «slc» ;` *To bind the (endemic) slice «slc» to a name «id1»; if multiple names are provided they are all aliases for the same (endemic) slice.*
`production «id1» [, «id2», ...] : «rule» from module «mod» ;` *To bind a production «rule» from a slice/module «mod» to a name «id1»; if multiple names are provided they all refer to the same production.*
`nt «id1» [, «id2», ...] : «rule» from module «mod» ;` *To unpack into «id1», «id2», . . . , «idn» the first n nonterminals in «rule» from the slice/module «mod».*
`action «id» : «nonterminal» from module «mod» role «name» ;` *To bind the action associated to the «nonterminal» from the slice/module «mod» to the name «id».*
Matching Operations
`«id»[[«cond1(attr1)» [, «cond2(attr2)», ...]]]` *Matches the AST node identified by «id» whose attributes verify the condition; «attri» is the name of an attribute of the node and «condi()» is a relational operator that compares the current value of the attribute against a constant.*
`«id1»[[«cond(attr)»]] < «id2»[[«cond(attr)»]]]` *Matches the AST node identified by «id1» when one of its children is identified by «id2»; it is possible to express conditions on the node attributes as in the above kind of match.*
`«id1»[[«cond(attr)»]] << «id2»[[«cond(attr)»]]]` *Matches the AST node identified by «id1» when the node «id2» can be reached from it ; it is possible to express conditions on the node attributes as in the other kind of matches.*
Behavior Specification
`before | after «matching-operations» { «code» }` *Registers the agent code enclosed between the "{" and "}" symbols at either the **before** or **after** hook of the nodes matched by the matching operations.*
`system-wide { «code» }` *The agent will execute the code enclosed between the "{" and "}" once, irrespectively of the execution control flow. This is typically used for changes that have system-wide effects, like slice replacement, etc.*
Agent Definition
`agent «canonical-name» { «behavior-specifications» }` *Defines an agent named «canonical-name» that has the behavior specified by «behavior-specifications», i.e., it defines code that is attached **before** or **after** specific tree nodes or code that has a **system-wide** effect.*

Table 4.4.: *Summary of the µDA DSL.*

5 Applicability of Open Interpreters

In this Chapter, we illustrate how open interpreters can be used to support backward compatibility, dynamic software updating and adaptation, context-aware variability, interpreter optimization, debugging and sandboxing. Some examples were developed in Neverlang.JS [22]: a Neverlang implementation of the Javascript language.

5.1. Backward Compatibility

As discussed in Chapter 1, backward compatibility is an important factor that heavily influences the adoption rate of evolved languages. A typical example is that of Python 3 which was not readily adopted by masses, although it has been released as far back as in 2008. There were many incompatibilities between Python 2.x and Python 3. In this section we show how open interpreters can be used to mitigate the migration issues due to backward incompatibility. The example we use is rather simple and by itself would not present a real migration challenge. However, the simplicity of the example helps us better grasp the *principles* behind our approach.

Consider the Python 2.x code in Listing 5.1(a) which in line 2 has an occurrence of the division operator. The latter is backward incompatible in Python 3; in Python 2 it performs division on integers, while in Python 3 it divides floats. In the considered example, this incompatibility would create issues with `pivot_index` function which in Python 3 would yield a floating point number. But, Python does not accept floats as list index and would, thus, yield the following error:

```
Traceback (most recent call last):
  File "python", line 5, in <module>
TypeError: list indices must be integers or slices, not float
```

There are, of course, several ways how one could rewrite the program to circumvent

```
1 def pivot_index(length):
2  return length / 2
3 lst = [4, 1, 3, 10, 4]
4 index = pivot_index(len(lst))
5 print(lst[index])
```

Listing 5.1: *Valid Python 2.x code for concatenating two lists.*

```
1 slice intDiv : Div from module neverlangPy2.Div;
2 slice floatDiv : Div from module neverlangPy3.Div;
3 agent adapt.IntegerDivision {
4   system-wide {
5     interpreter.replaceSlice(floatDiv, intDiv);
6   }
7 }
```

Listing 5.2: *Agent that replaces the new division semantics with the old one to preserve the original application semantics.*

the incompatibility. But, we focus on the principle behind the issue which is not easily remedied in more complex applications. Suppose we want to exploit all the Python 3 enhancements, except for the division operator which would break our application. With an open Python interpreter, we are able to modify the semantics of the division operator to maintain its original integer division semantics. There are several ways one can achieve this, depending on the moment when the modification is applied and on the scope of the change. For example, one might want to apply the change at load time, i.e., before the application is executed. Or, the developer might discover the incompatibility only when the application is already being executed and cannot shut it down; in this case, the update must be performed at runtime. Also, one might want to change the semantics of all occurrences of the division operation. Or, she might prefer to modify only specific occurrences that cause issues.

Listing 5.2 shows an agent that replaces the new (Python 3) division slice with the old (Python 2) slice. The effects of such a change would be global and all occurrences of the division operation would perform the integer division. However, open interpreters allow one to selectively modify the problematic occurrence of the division operator,

a) `def` construct semantics

```
1 module neverlangPy2.Def {
2   reference syntax {
3     Def: Stmt <- "def" Id "(" ExprList ")" ":" Body;
4   }
5 }
```

b) Agent for replacing float- with integer division

```
6  nt def,id,_,body : Def from module neverlangPy3.Def;
7  nt div,_,_ : Div from module neverlangPy3.Div;
8  action intDiv : Div from module neverlangPy2.Div;
9  agent adapt.IntegerDivision {
10   before def < id[name=="pivot_index"] && body << div | div {
11     div.setSpecializedAction(division, intDiv, "evaluation");
12     interpreter.unregister(this);
13   }
14 }
```

Listing 5.3: *`def` syntax definition and the µDA agent for changing the division operator semantics.*

without affecting other non-problematic occurrences. This would allow one to partially migrate from Python 2 to Python 3. There are two solutions to this problem, however, one would work only if the agent is executed at the interpreter load time, i.e., before the application run. To understand both solutions, let us first analyze the `def` construct. Listing 5.3(a) shows the syntax definition of the `def` construct in Neverlang. The `Id` nonterminal holds the function name, `ExprList` will hold a list of function arguments and `Body` represents the function body. Figure 5.1 shows the tree structure of the `pivot_index` function declaration. The nodes are labeled with abbreviated nonterminals (`S` stands for Stmt, `EL` for `ExprList`, `B` for `Body`, `E` for `Expr` and the division symbol "/" represents the division expression, therefore replaces the `Expr` nonterminal). Beside the `Id` node we put a list of attributes it defines. We would like to identify the gray node that represents the division operator and change its semantics to integer division.

Listing 5.3(b) shows a solution that would work only if the agent is executed at the interpreter load time. The solution consists in registering the agent in the parse tree before the declaration of `pivot_index`. In line 10, we use a pattern that identifies the desired node. The pattern is best explained with reference to Figure 5.1. Keep in mind that `x < y` matches nodes where `x` is the immediate parent of `y`, while `x << y` matches nodes where `x` is an ancestor of `y` and `y` is not necessarily the immediate child of `x`. Therefore, `def < id[name=="pivot_index"]` will match the green nodes in Figure 5.1, i.e., nodes that represent the function definition where function name is "pivot_index". Once we identify the subtree that defines our function, we need to identify the division operation in its body. The `<<` operator comes in handy; `body << div` would match the node `B` and the (gray) division node in Figure 5.1. Once these nodes are selected, we filter out all but the node matched by `div`. On the remaining node we set as a specialized action the action that we previously bound to the `intDiv` identifier (line 8). This action is defined in `neverlangPy2` module, therefore implements the old integer division semantics. As already emphasized, the solution would only work if the agent is registered in the parse tree at load time. Indeed, if the agent is executed when the application is already running, the function declaration might have already taken place and the hook where the agent is registered might never be reached again. Therefore, the modification would never be applied.

To overcome the limitations of the above solution, we define our agent as in Listing 5.4. Instead of registering the agent on a particular tree node and applying the modification once the node is notified, we modify the desired node without prior registration. Therefore, we put the reflection code in the `system-wide` code block. Once the agent is executed, we identify the division node by using the same pattern as in Listing 5.3(b), just that now we dynamically generate it by using `interpreter.compilePattern(pattern)` (line 6). The pattern that is generated by `compilePattern` (line 7) is then passed to `interpreter.getNodes()` which will return all the nodes that match the pattern. In our case, it should match one single node, i.e., the gray node in Figure 5.1. Indeed, there is just on function named `pivot_index` with one single division operation. On the obtained node we set a specialized action (line 9) that we previously bound to the `intDiv` name (line 3). This solution would work irrespectively of the moment in which the agent was executed.

```
1  nt def,id,_,body : Def from module neverlangPy3.Def;
2  nt div,_,_ : Div from module neverlangPy3.Div;
3  action intDiv : Div from module neverlangPy2.Div;
4  agent adapt.IntegerDivision {
5    system-wide {
6      ITreePattern pattern = interpreter.compilePattern("def < id[name==\"pivot_index\"]
           && body << div | div");
7      NodeInfo[] nodes = interpreter.getNodes(pattern);
8      NodeInfo pivotIndex = nodes[0];
9      pivotIndex.setSpecializedAction(division, intDiv, "evaluation");
10   }
11 }
```

Listing 5.4: *Agent for replacing float- with integer division.*

The illustrated example shows that open interpreters can effectively be used to mitigate the issues that arise from backward incompatibility. In particular, they enable one to partially migrate from the old to the new version of the language. Indeed, one is able to choose which legacy constructs she wants to keep. Sometimes, an updated construct might depend on another updated construct, in which case one should either keep or migrate both of them. Kühn *et al.* [78] describe a tool that traces language feature dependencies and can potentially assist the developer in language migration.

An alternative solution, which does not rely on open interpreters, would be to directly tailor the updated interpreter. This would imply the modification of the interpreter's source code and its subsequent recompilation. However, it is unreasonable to believe that the developer posses this kind of knowledge. Also, many language

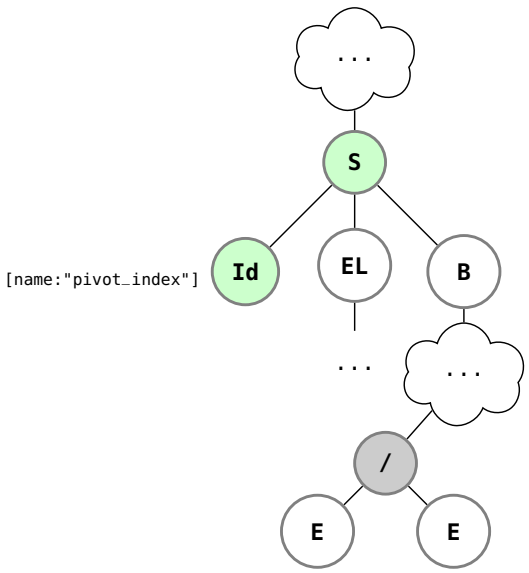


Figure 5.1.: *Tree structure of a function definition in Python.*

implementations are monolithic [121] which makes their evolution and updating harder. Alternatively, in the specific case of Python, one could use the special (also called "magic") methods[1] to overload the existing operators. However, this solutions strongly depends on the overloading support from the underlying language and is hardly applicable to a broader set of problems. Open interpreters, instead, provide a general and a fine-grained mechanism that can be applied to a wide range of problems, not limited to simple operators.

5.2. Dynamic Software Updating

Many software systems must provide continues, uninterrupted services, e.g., telephone switches, air traffic control systems, nuclear power plant monitors, human body activity regulators, etc. The interruption of such services might lead to a non-negligible economic loss, an ecological disaster or even to tragic consequences like death. Such problems led to the development of a variety of techniques for dynamic software updating, e.g., code injection, class reloading and dynamic/multiple method tables. Although progress was made in the field of DSU, the existing approaches still suffer from long-term performance decay (e.g., JavAdaptor [99, 100] and DUSC [93]) or impose limitations on what can or cannot be updated (e.g., JRebel [65]) These deficiencies urge researches to continuously seek for better solutions. Yet, little was done to support DSU through language evolution. Recently, we showed that software can be dynamically adapted through runtime language evolution [26, 29, 19, 28]. The key concept behind DSU through language evolution is the *semantic propagation principle* which relies on the following facts. The behavior of application modules is governed by the semantics of the language constructs used to write the application. Therefore, changing the semantics of a language construct affects the way a module behaves. With an appropriate framework support and under specific conditions, the semantic propagation principle can be used to evolve the software by changing how the underlying language behaves. Neverlang's support for open interpreters paves the way for dynamic software updating through interpreter adaptation. There is room for improvement, especially in making interpreter adaptation more user-friendly. Yet, we provide a proof of concept: the behavior of an application can be updated through interpreter adaptation.

Consider the example, originally discussed by Chiba [34], whose code is shown in Listing 5.5(a). The problem consists in adding a posteriori the ability to store objects of type `Node` into a persistent storage. Chiba's presents a compile-time metaobject protocol which obviously does not pretend to and cannot be a DSU approach, however, examining his solution helps us better understand how open interpreters can be used to address this problem. Therefore, for simplicity, we first consider the case in which

[1] `http://web.archive.org/web/20171108074402/https://www.python-course.eu/python3_magic_methods.php`
`http://web.archive.org/save/http://www.diveintopython3.net/special-method-names.html`

```
class Node {
public:
  Node* next;
  double value;
};
Node get_next_of_next(Node* p) {
  Node* q = p->next;
  return q->next;
}
```

a) Original code

```
class Node : public PersistenObject {
public:
  Node* next;
  double value;
};
Node get_next_of_next(Node* p) {
  Node* q = (p->Load(), p->next);
  return (q->Load(), q->next);
}
```

b) Manually modified code to support object persistence

Listing 5.5: *Manually implementing object persistence in C++.*

the already deployed application *can* be shut down. The most obvious way to introduce object persistence would be to directly update the `Node` class. However, that would also be the most invasive way and software design principles suggest that inheritance could be used to achieve the same goal with less changes to the original code. Listing 5.5(b) shows a possible solution where class `Node` inherits from `PersistentObject` which implements persistence through the `Load` method responsible for retrieving object data from a persistent storage (the actual implementation is omitted as it is irrelevant). However, as discussed by Chiba, the inheritance mechanism by itself does not suffice to smoothly introduce object persistence. In fact, the developer must further ensure that the usage of `Node` objects is correct across the entire application code. This implies that all occurrences of object access must be preceded by a call to the `Load` method as illustrated in the `get_next_of_next` method in Listing 5.5(b). If we better analyze the above problem, we see that the object persistence is closely related to the language features for object field (read and write) access. Indeed, if the "read object field" feature (implemented by the `->` operator in C++) would implicitly load the object from a persistence storage, the original application could remain untouched. Similarly, the field assignment operator (`o->f = v` in C++) would have to implicitly store the object state after it changes. Chiba then proposes a solution with OpenC++, a compile-time metaobject protocol, to extend the application with object persistence through compile-time code injection. The solution consists in marking `Node` as a subclass of a metaclass which automatically calls `Load` before an object field is accessed. Similarly, the metaclass will automatically save the object state when it changes. The OpenC++ compiler will then generate pure C++ code where `Node` will have the extension for object persistence. In other words, OpenC++ frees the developer from having to edit all source code, although it still requires to make small annotations on class definitions in order to instruct the OpenC++ compiler on how to perform the translation.

Conceptually, Chiba's solution injects persistence behavior *before* a field access and *after* a field assignment, which is exactly what we can do with open interpreters either at interpreter load- or runtime. Listing 5.6(a) shows how the Javascript's `new` construct

a) Neverlang modules implementing the new *construct syntax and prototype-based object instantiation*

```
1  module neverlangJS.NewSyntax {
2    reference syntax {
3      New: Expr ⇐ "new" PrototypeName "(" ArgList ")";
4    }
5  }
6  module neverlangJS.New {
7    imports { neverlangJS.util.ObjectInstance; }
8    role (evaluation) {
9      New: .{
10       $New.val = new ObjectInstance($New[1].name, $New[2].args);
11     }.
12   }
13 }
```

b) Sample Javascript code with prototype instantiation

```
14 function Node(next, value) {
15     this.next = next;
16     this.value = value;
17 }
18 var node1 = new Node(null, 7);
```

Listing 5.6: *Neverlang implementation of the Javascript's prototype-based instantiation and a Javascript instantiation example.*

is implemented. The `new` construct is an expression that returns a reference to the instantiated object (line 10). It's syntax is defined as the keyword `new` followed by a prototype name and a list of constructor arguments (line 3). Thanks to Neverlang's modularity support, nonterminals `Expr`, `PrototypeName` and `ArgList` can be defined elsewhere. Their specific implementation is irrelevant. We only have to know that `PrototypeName` defines the `name` attribute which stores the prototype name, and `ArgList` has the `args` attribute which stores a list of expressions. Module `neverlangJS.New` defines the semantics. The `ObjectInstance` class, imported in line 7, is a Java class that is internally used by the interpreter to represent Javascript objects. It stores relevant information, such as object identity, object prototype and object fields.

Listing 5.6(b) shows an example of Javascript prototype-based object instantiation. When Neverlang.JS encounters "`new Node(null, 7)`" (line 18) it will execute the semantic action defined in module `neverlangJS.New` (lines 9-11). The `$New[1].name` would store `"Node"`, i.e., the prototype name of the object being instantiated, and `$New[2].args` would store a Java `List` object with elements `null` and 7.

To persistently store objects of type `Node`, we need to identify tree nodes that represent objects instantiation of `Nodes` and modify their behavior to persistently store their state. However, depending on the moment in which we trigger the interpreter adaptation, some objects might have already been instantiated with the standard non-persistent behavior. Figure 5.2 illustrates this on a simplified parse tree in which colored nodes (leaves) represent object instantiation of `Node` objects. Suppose that the left subtree of

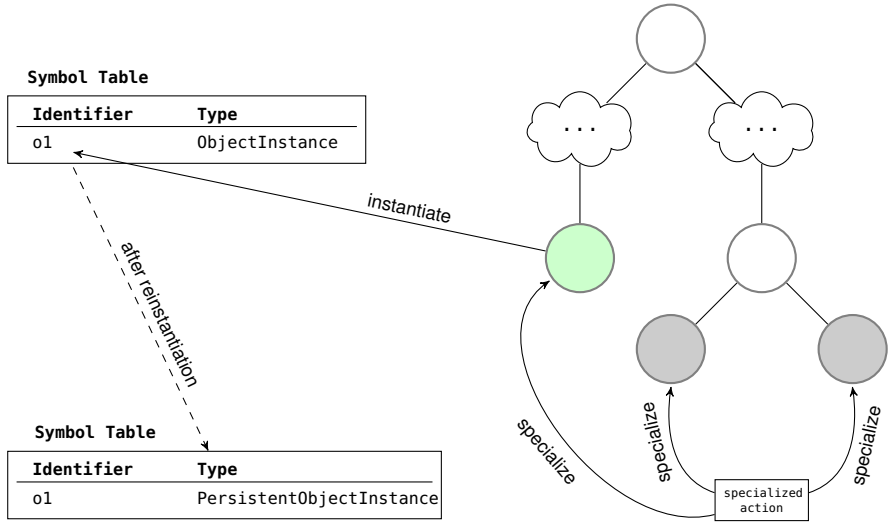


Figure 5.2.: *Transformation of* `Node` *instances into persistent objects.*

the root node was already visited. Therefore, the node instantiation represented by the green node was performed and the object was instantiated as a standard non-persistent object (as showed in the symbol table in the upper left corner[2]). On the other hand, the gray nodes in the right subtree were not visited yet, therefore no instantiation took place. The two situations require different modifications of the interpreter and its state as we discuss in the following.

Let us first consider the necessary modifications to make all future instantiations produce persistent objects. To this purpose, we first define a new framework-level class, `PersistentObjectInstance`, that will be used to represent application-level objects which are able to persist their state. This class ensures that before each access to object members the object state is loaded from a persistent store. Similarly, after an object member is modified the object's state must be stored in a persistent store. The actual implementation of `PersistentObjectInstance` is irrelevant. Then, we define a new module that implements the semantic action which uses the new class and hence implements the new, persistent behavior (see Listing 5.7(a)). Next, we have to trace all instantiation occurrences (tree nodes that represent instantiation) of `Node` and update them to use the new behavior defined in Listing 5.7(a). The agent is shown in Listing 5.7(b). In lines 9-11 we bind some variables to language concepts that we will use in the agent for the adaptation purpose. In line 24 we use a tree pattern that will identify all object instantiations of the prototype named `Node`. With `| newExpr` we filter out unnecessary nodes. On the collected nodes we set a specialized action with the

[2]Notice that the instantiation by itself does not insert the object in the symbol table. Indeed, this is done by the assignment operator which binds the instantiated object to a name. In Figure 5.2 we assume that the binding is done by some node hidden in the cloud node, which represents a portion of the tree irrelevant to the discussion.

a) instantiation construct for persistent objects

```
1  module neverlangJS.PersistentNew {
2    imports { neverlangJS.util.PersistentObjectInstance; }
3    role (evaluation) {
4      New: .{
5        $New.val = new PersistentObjectInstance($New[1].name, $New[2].args);
6      }.
7    }
8  }
```

b) µDA code to selectively introduce object persistence

```
9  nts newExpr, protoName, _ : New from module neverlangJS.NewSyntax;
10 action persistenNew : New from module neverlangJS.PersistentNew;
11 endemic slice st : SymbolTable from module neverlangJS.SymTable;
12 agent adapt.Persistent {
13   system-wide {
14     // transform any Node that was instantiated as non-persistent
15     for (String id : st.getVariables()) {
16       ObjectInstance oldObj = (ObjectInstance)st.get(id);
17       if (oldObj.getName("Node") && !(oldObj instanceof PersistentObjectInstance)) {
18         // PersistentObjectInstance constructor copies application-level identity
              properties}
19         ObjectInstance newObj = new PersistentObjectInstance(oldObj);
20         // update the symbol table
21         st.replaceAll((var,obj)-> obj.equals(oldObj) ? newObj : oldObj);
22     }
23   }
24   before newExpr < protoName[name == "Node"] | newExpr {
25     // change "new Node" to use persistent objects
26     interpreter.setSpecializedAction(newExpr, persistentNew);
27     interpreter.unregister(this);
28   }
29 }
```

Listing 5.7: *Persistent objects.*

new behavior defined in module `neverlangJS.PersistentNew` (see Figure 5.2). This specialized action will override the one defined in the original language implementation as was explained in Section 2.2.4. Once the change is made, the agent unregisters itself from the node hook. This change affects all future `Node` instantiations. Indeed, after the modification, whenever the interpreter will encounter the Javascript expression "`new Node`", it will run the specialized action which instantiates persistent objects. Notice that we also modify the behavior of the green node (left-most leaf) that was already visited. This is because we do not know if the same node will be revisited in the future and therefore would instantiate a new object of type `Node` for which we want it to be persistent.

To transform the already instantiated non-persistent objects we need to reinstantiate them with the `PersistentObjectInstance`. Lines 13-23 in Listing 5.7(b) shows the code snippet that performs the necessary reinstantiation. The code traverses the symbol

table to find all referable objects and for each one of them it checks if its type (prototype) is `Node` and whether it was instantiated with `PersistentObjectInstance` class (line 17). If the object is not persistent, we reinstantiate it with `PersistentObjectInstance` (line 19). Please notice that reinstantiating objects will change their JVM object identity. However, we preserve their *application-level* identity (as seen from Javascript code) by copying the identity-related properties to newly instantiated objects. In other words, application-level object identity is something handled by the interpreter not by the underlying JVM. The constructor of `PersistentObjectInstance` is responsible for the identity preservation. Next, in line 21 we update the symbol table so that all references to the old non-persistent object now point to the new one. The symbol table is a subclass of a Java `HashMap` therefore we use the `HashMap`'s `replaceAll` method for the purpose. In the bottom left corner of Figure 5.2 we show the state of the symbol table after reinstantiation.

The presented example illustrates how open interpreters can be used to dynamically adapt the behavior of a system by changing the behavior of the underlying interpreter. The approach is especially appropriate when the application behavior that one wants to modify is well-aligned with one or more language constructs. In the presented example, object persistence is perfectly aligned with the instantiation construct and the field "read" and "write" constructs. The greater the misalignment between application-level concepts and the underlying language features the more difficult it becomes to use open interpreters for this purpose. The concept of microlanguages discussed in Section 4.5 can greatly assist the developer in the process of interpreter adaptation. Intuitively, this approach is especially appropriate for dynamically updating domain-specific languages in which, by definition, domain concepts are aligned with language constructs [122, 87, 49]. One of the main strengths of this approach is that, if feasible, the solution can be shared among different applications and, in some cases, even to different language implementations.

5.3. Context-Aware Variability

Dynamic adaptation to the execution context is a desirable feature in software that operates in an evolving environment. In [19] we described how open interpreters can be used to support context-aware variability to provide accessibility support. In [28] we extended the Neverlang's model to support the concept of layers borrowed from context-oriented programming (COP). In the following we describe two examples of context-aware variability provided through open interpreters.

5.3.1. Accessibility

In [28] we used open interpreters to provide accessibility support in an HTML viewer whose visualization behavior is adapted according to the user's eyesight conditions[3].

[3] A demo application, based on a proof-of-concept prototype implementation of open interpreters, was demonstrated at the Modularity'16 demo track [26] and is illustrated in the video at

Language feature	Description
`print` Expr	prints `Expr` to screen
`set font size` Expr	sets the font size to `Expr` points
`set font color` Expr	sets the font color to `Expr`

Table 5.1.: *The linguistic constructs added to Neverlang.JS.*

```
function view(element) {
  var parsed = parse(element);
  set font color parsed.color;
  set font size parsed.size;
  print parsed.text;
}
```

Listing 5.8: *JavaScript snippet of the view application feature.*

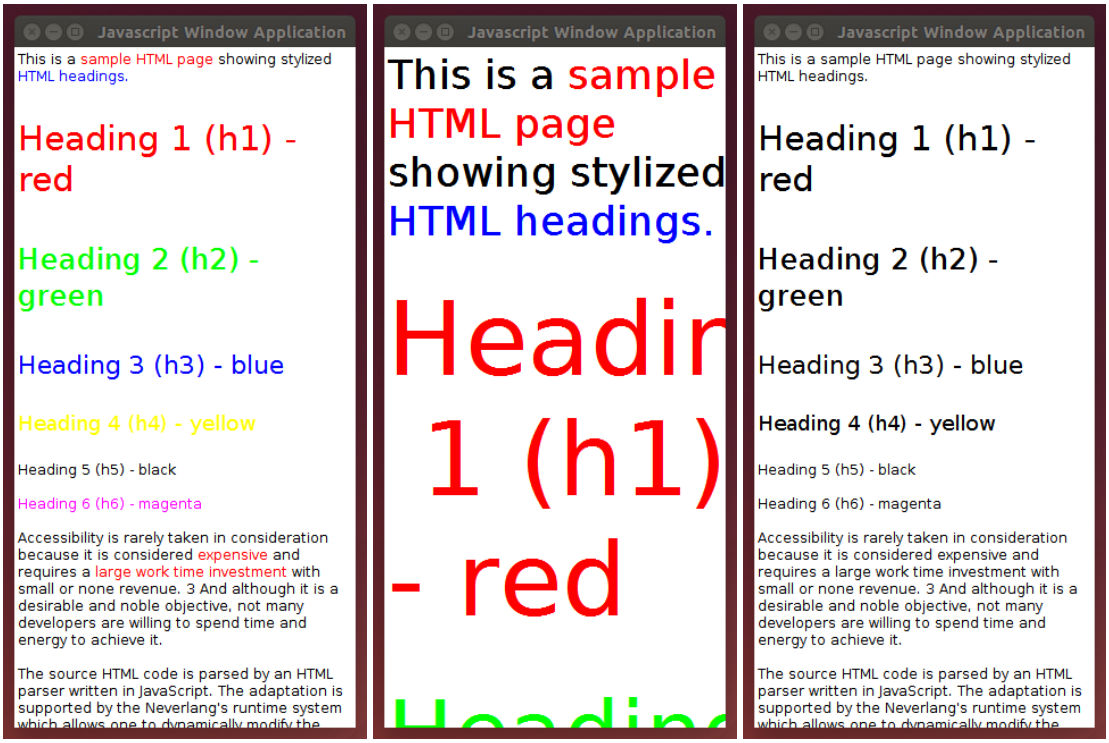
The viewer supports four different modes of visualization depending on whether the user has or does not have eyesight problems. For non-impaired users, the viewer renders the page as specified by the underlying HTML code. For users suffering from color vision deficiency (color blindness), any text color is substituted with black. For users suffering from long-sightedness (hyperopia), the viewer uses a larger font than the one specified in the HTML code. Finally, for blind users, the displayed text is read aloud by a text-to-speech engine. This simple example demonstrates how dynamic software updating through interpreter adaptation can facilitate development for accessibility, which is normally a very resource- and time-intensive task[4].

The HTML viewer was developed in Neverlang.JS which, for the purpose of concept illustration, was extended with a small domain-specific language for text visualization. This extension better aligns the evolving domain concepts (user eyesight condition) with the concerned language concepts. Consequently, this simplification facilitates the discussion and the proof of concept, while in no way limits the general application of the approach to dynamic software updating. Table 5.1 shows the three linguistic constructs that affect the page visualization. Listing 5.8 shows their usage in the HTML viewer. First, an HTML element (e.g., `<a>`, `<p>`, `etc.`) is parsed by the `parse` function. Then, given the attributes of the parsed element, `set font size` and `set font color` set the font color and size which are then used by the `print` statement. By changing the behavior of `print`, we can change how the page is visualized without modifying the

http://cazzola.di.unimi.it/μ-dsu/μ-dsu-demo.mp4.

[4]See discussions on cost of accessibility at

- `https://web.archive.org/web/20170925081419/https://www.viget.com/articles/an-uncomfortable-missing-part-of-the-accessibility-discussion`
- `http://web.archive.org/web/20160825194414/http://accessites.org/site/2007/11/does-accessibility-cost-more/`



(a) normal (b) long-sighted (c) color-blind

Figure 5.3.: *A sample page displayed according to 3 of the possible profiles.*

source code of the HTML viewer. Moreover, new visualization behavior can be added a posteriori according to the emerging needs (e.g., users with new eyesight conditions).

Figure 5.3 shows how a sample page is visualized to normal, long-sighted and color-blind users. The page rendering differs, although the underlying HTML code is the same for all three page visualizations. This is achieved by providing different behaviors of the language construct responsible for visualization. Depending on the user profile, the interpreter is updated to use the most suitable semantics for the linguistic construct that is responsible for visualization (`print`). Listing 5.9 shows four different implementations of the `print` statement. Basically, the "healthy" print simply prints the expression (line 8); the "blind" print statement both prints (line 17) and reads (line 18) the text aloud (the `speak` method uses a text-to-speech engine); the "colorblind" print prints the text with the black color (line 28); the "longsighted" print increases the font size (line 39) before printing the text (line 40).

Listing 5.10(a) shows the μDA code of the agent that replaces the `-Print` slices according to the user profile. Remember that the code enclosed by the `system-wide` qualifier goes in the `main` method of the class that extends the `Agent` abstract class. Therefore, in the `system-wide` code we can use the `args` variable provided by the `main` method. In line 3 we retrieve a reference to the already loaded slice for printing, whose

```
1  module Print {
2   reference syntax {
3    Print: Stm ⇐ "print" Exp;
4   }
5   role (evaluation) {
6    Print: @{
7     JSType t = (JSType)$Print[1].value;
8     view.print(t.stringValue());
9    }.
10  }
11 }
```

a) Print variation for healthy users

```
12 module BlindPrint {
13  reference syntax from Print
14  role (evaluation) {
15   Print: @{
16    JSType t = (JSType)$Print[1].value;
17    view.print(t.stringValue());
18    view.speak(t.stringValue());
19   }.
20  }
21 }
22
```

b) Print variation for blind users

```
23 module ColorBlindPrint {
24  reference syntax from Print
25  role (evaluation) {
26   Print: @{
27    JSType t = (JSType)$Print[1].value;
28    view.print(t.stringValue(),
         Color.BLACK);
29   }.
30  }
31 }
```

c) Print variation for color-blind users

```
34 module HyperopicPrint {
35  reference syntax from Print
36  role (evaluation) {
37   Print: @{
38    JSType t = (JSType)$Print[1].value;
39    int size = view.getCurrentSize()*3;
40    view.print(t.stringValue(), size);
41   }.
42  }
43 }
```

d) Print variation for longsighted users

Listing 5.9: *Four variations of the* `print` *language feature.*

name we pass to the agent from command line. Instead, in line 4 we *load* a slice whose canonical name is stored in `args[3]` because the new slice is not yet loaded by the interpreter, hence we cannot simply use `getSlice`. Once we have references to both the old and the new slice, we call `replaceSlice` which intuitively replaces the old slice with the new one. To reload the page we invoke `redoRole` which forces the tree to be revisited by the "evaluation" role stored in `args[1]`. The agent execution is governed by a simple bash script shown in Listing 5.10(b). Whenever a profile change is detected, the script runs the agent defined in Listing 5.10(a) which modifies the interpreter instance named `jsHTML` by replacing the slices according to the provided arguments. Notice that the original application code never changes, i.e., the application remains unaware of the supported adaptation, and new behaviors for the `print` statement can be added a posteriori.

5.3.2. Resource Usage Optimization

Software evolution is often encouraged by the desire to better exploit the available resources. For example, network routers can update their routing algorithms to increase the network throughput; a web server can update its caching policy to reduce its response time; a smartphone app can be evolved to reduce battery consumption; etc. We advocate that in certain situations resource usage can be more easily optimized if done through language- instead of application evolution. If the evolution is implicit in a language construct, it can achieve the same, if not better optimization rate, while leaving the input source code untouched, what contributes to preserving the original logic

a) µDA code for replacing `-Print` slices.

```
agent adapt.ReplacePrint {
  system-wide {
    SliceInfo old = interpreter.getSlice(args[2]);
    SliceInfo new = interpreter.loadSlice(args[3]);
    interpreter.replaceSlice(old, new);
    interpreter.redoRole(args[1]);
  }
}

```

b) `bash` script that governs the interpreter adaptation.

```
current="Healthy"
while true; do
   profile=getprofile()
   if ["$profile" != "$current"]; then
     java -cp $NEVERLANG_ROOT/gen-src:$NEVERLANG_HOME/Neverlang.jar jsHTML evaluation
        ${current}Print ${profile}Print
      current="$profile"
   fi
   sleep 1
done
```

Listing 5.10: *Code to modify the interpreter that is running the HTML viewer.*

and to maintenance. Let us consider two example of resource optimization through language evolution.

Class Instantiation. The following example, originally discussed by Tanter [115], shows that developers might want to evolve the language to optimize the usage of available resources (CPU, memory, etc.). Furthermore, the example emphasizes that typical language implementations do not take into account valuable runtime or user-possessed information (context). Consider the problem of how to efficiently instantiate objects in a class-based programming language. Listing 5.11(a) shows two classes, for simplicity written in Java. `Point` has two fields, namely `x` and `y`, whose values store the point coordinates in a plane. `Person` has potentially hundreds of fields describing a person. The two classes differ in the fact that a `Point` object will always have the two coordinates, so the class fields will always be used, while a `Person` object might have hundreds of empty fields. Depending on how the underlying runtime system stores object fields and on the actual data with which the application is fed, the application will either use too much memory or it will perform slower than it could. For example, if the language implementation stores object fields in an array-like data structure, the `Point` objects will perform optimally both with respect to the memory usage and the execution speed. On the other hand, if fields are stored in a map-like data structure, the field retrieval will be slower and the application will perform worse. In the `Person` class example, the ideal solution to reduce memory usage is to use a map-like data structure which would store only the fields that are actually used. But again, maps

a) `Point` *and* `Person` *classes*

```
class Point {
  // point coordinates in a plane
  private int x;
  private int y;
}
class Person {
  private String name;
  private String lastName;
  private int age;
  private String address;
  ... // hundreds of fields omitted
  private Color hairColor;
}
```

b) A possible solution to the problem of unused fields

```
class Person {
  private String name;
  private String lastName;
  private int age;
  private OptionalPersonInfo optional = null;
}
```

Listing 5.11: *Class field implementation problem.*

have undesired performance issues. On the other hand, storing fields in an array-like data structure will perform better with respect to the execution speed, but will waste more memory.

With no language-level support for differentiating instantiation strategies, the developer would have no other choice but to rewrite the code as, for example, in Listing 5.11(b). The solution consist in making the `Person` class store the least used fields in an optional field instantiated on demand. However, this solution introduces the problem (not present in the default implementation) of finding the optimal set of fields to be stored in `OptionalPersonInfo` class. It is also invasive as it clutters the application code to overcome a language weakness. Even if the language supported to different constructs for instantiation, this would unnecessarily obscure the application logic. In fact, instantiation optimization is a crosscutting concern which should not interfere with application logic.

Let us consider the same issue in Neverlang.JS, where the instantiation problem is inverted because Javascript objects, by default, store fields in a Java `HashMap`. Hence, `Person` object would already use the ideal strategy, but the `Point` object would suffer from performance issues. Being a Neverlang-based interpreter, Neverlang.JS is open in the sense of Definition 3. Therefore, as was illustrated in Section 5.2 when discussing the problem of object persistence, Neverlang.JS can be modified either at load- or at runtime to optimize class instantiation. For the reader's convenience, in Listing 5.12(a) we show again the implementation of object instantiation in Neverlang.JS. Class `ObjectInstance`, which is used at the interpreter-level to represent application-level Javascript objects, by

a) Neverlang modules implementing the `new` *construct syntax and prototype-based object instantiation.*

```
1  module neverlangJS.NewSyntax {
2    reference syntax {
3      New: Expr ←- "new" PrototypeName "(" ArgList ")";
4    }
5  }
6  module neverlangJS.New {
7    imports { neverlangJS.util.ObjectInstance; }
8    role (evaluation) {
9      New: .{
10       $New.val = new ObjectInstance($New[1].name, $New[2].args);
11     }.
12   }
13 }
```

b) Interpreter-level class for representing application-level Javascript objects.

```
14 class ObjectInstance {
15   HashMap<String,Object> fields;
16   String prototype;
17   ...
18   public ObjectInstance(String prototype, HashMap<String,Object> fields) {
19     this.prototype = prototype;
20     this.fields = fields;
21   }
22   ...
23 }
```

c) Neverlang module for array-based class instantiation.

```
24 module neverlangJS.ArrayLikeNew {
25   imports { neverlangJS.util.ArrayLikeInstance; }
26   role (evaluation) {
27     New: .{ $New.val = new ArrayLikeInstance($New[1].name, $New[2].args); }.
28   }
29 }
```

Listing 5.12: *Neverlang implementation of the Javascript's prototype-based instantiation and a Javascript instantiation example.*

default stores fields in a Java `HashMap` (see Listing 5.12(b)). Therefore, we need to change that behavior so that object fields are stored in a Java array. To this purpose, we define a new interpreter-level class for representing Java objects. The actual implementation is irrelevant. Next, we have to define a new Neverlang module that uses the new class. Listing 5.12(c) shows a module called `neverlangJS.ArrayLikeNew` which, in line 17, instantiates Javascript objects by using the `ArrayLikeInstance` class. The next step is to replace the semantics of the existing hashmap-based Javascript construct for instantiation with the array-based one.

As with object persistence, discussed in Section 5.2, we have to perform two modifications. First, we have to modify all future instantiations of `Point` objects. Listing 5.13

```
1  nts newExpr, protoName, _ : New from module neverlangJS.NewSyntax;
2  action arrayLikeNew : New from module neverlangJS.ArrayLikeNew;
3  endemic slice st : SymbolTable from module neverlangJS.SymTable;
4  agent adapt.ArrayLikeInstantiation {
5    system-wide {
6      // transform any Point that was instantiated with ObjectInstance
7      for (String id : st.getVariables()) {
8        ObjectInstance oldObj = (ObjectInstance)st.get(id);
9        if (oldObj.getName("Point") && !(oldObj instanceof ArrayLikeInstance)) {
10        // ArrayLikeInstance constructor copies application-level identity properties}
11        ObjectInstance newObj = new ArrayLikeInstance(oldObj);
12        // update the symbol table
13        st.replaceAll((var,obj)-> obj.equals(oldObj) ? newObj : oldObj);
14     }
15   }
16   before newExpr < protoName[name == "Point"] | newExpr {
17     // change "new Point" to use instantiation defined in module
          neverlangJS.ArrayLikeNew
18     interpreter.setSpecializedAction(newExpr, arrayLikeNew);
19     interpreter.unregister(this);
20   }
21 }
```

Listing 5.13: *µDA agent for class instantiation optimization.*

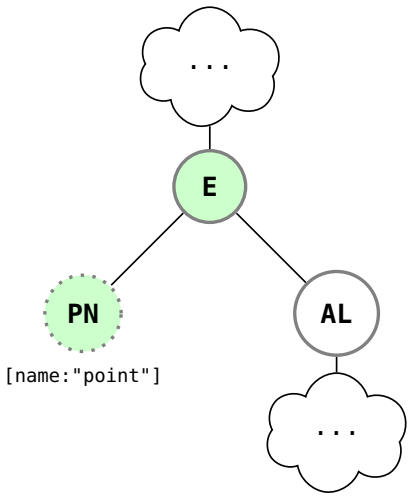


Figure 5.4.: *Tree representation of the instantiation construct* `new`*. Nonterminal names are abbreviated:* `E`*,* `PN` *and* `AL` *stand, respectively, for* `Expr`*,* `PrototypeName` *and* `ArgList` *(see Listing 5.12(a)).*

shows the necessary code. The pattern in line 16 will match all nodes that represent the Javascript construct for instantiation (`new`) of `Point`. Figure 5.4 shows the matched node. The `newExpr < protoName[name == "Point"]` part of the pattern matches the two green-colored nodes, but the filter removes the dotted `PN` node. On the remaining node (`E`) the agent sets a new specialized action which, as was explained in Section 2.2.4,

will override the original action. Consequently, the original (hashmap-based) instantiation strategy will be overridden by a new (array-based) strategy. However, by the time we apply this change, some objects of type `Point` might have already be instantiated with the original hashmap-based strategy of storing fields. We need to change the behavior of this objects too. Listing 5.13, lines 5 to 15, shows the necessary code. Basically, we iterate over all referable objects which are stored in the symbol table `st`. For each object we check whether its prototype is `Point` and whether it was instantiated with the original instantiation strategy (line 9). If yes, we reinstantiate the object (line 11) and we make sure that the application-level identity is preserved by copying all identity-related properties of the old object to the new one. The constructor of `ArrayLikeInstance` is responsible for the preservation of object identity. Finally, in line 13 we update all references to the old object to point to the new object.

Parallelization. The support for context variability is becoming an increasingly demanded feature especially in the area of ubiquitous and mobile computing. For example, think of a software handling an electric car that switches to power saving mode while the car is waiting at a traffic light; an app that turns off some of its functionality if the smartphone is low on battery, etc. All these represent a family of applications that have to adapt their behavior according to some context information with the aim of a smarter resource usage. The traditional approach to software development has a quite limited support for behavioral variability that mainly consists in having all the possible variants mixed in the base code each guarded by the check for the context change that should activate it. This clearly obstacles code extensibility, reuse and maintenance and together with the emerging requirement for context variability support led to the development of several approaches to support context variability; all based on separating the context dependent behavior from the base behavior and on introducing an activation logic that permits to (de)activate the variants according to the current context information (context awareness). In [28] we illustrated how open interpreters can be used to optimize resource usage without modifying the original source code. The approach was presented in contrast to the traditional context-oriented programming approaches which mostly do require one to change the application sources to code the variability. As such, COP languages are a valid approach when developing software from scratch. Instead, if we want to introduce context-aware variability to an existing application written in a non-COP language, traditional COP approaches require invasive and error-prone modifications. This is especially true when variability is implicit in a language construct.

Think of the example shown in Listing 5.14 which shows a Javascript implementation of the escape time algorithm for calculating the Mandelbrot set. A Mandelbrot set is a collection of complex numbers for which a function of the form $f(x) = x2 + c$ does not diverge. If we plot the obtained set to a plane we get a fractal. The algorithm details are not important; we focus only on the two loops in lines 4 and 5 which are responsible for the complete set calculation. These loops account for most of the execution time which increases with the size of the calculated set. The algorithm is known to be

```
1  var MAX_ITER=50, ZOOM=450, HEIGHT=300, WIDTH=300;
2  var I = create2DArray(WIDTH,HEIGHT);
3  var zx=0, zy=0, cX=0, cY=0, tmp=0, iter=0;
4  for(var y=0; y < HEIGHT; ++y) {
5      for(var x=0; x < WIDTH; ++x) {
6          zx=0; zy=0;
7          cX = (x - 400) / ZOOM;
8          cY=(y-300) / ZOOM;
9          iter=MAX_ITER;
10         while ((((zx * zx) + (zy * zy)) < 4) && (iter > 0)) {
11             var tmp=(zx * zx) - (zy * zy) + cX;
12             zy = (2.0 * (zx * zy)) + cY; zx=tmp; iter=iter - 1;
13         };
14         I[x][y]=iter | (iter << 8);
15     }
16 }
```

Listing 5.14: *A Javascript implementation of the escape-time algorithm.*

parallelizable [53, 20, 31] since each independent stage of a loop can be run in parallel on a different core. A parallel execution is fast, but consumes more energy and could drain the laptop's battery. A sequential execution is slower, but consumes less energy. On this trade off, we would like to switch between a sequential and a parallel execution depending on whether the laptop is plugged into the mains or it is running on battery. The example deliberately omits algorithmic details to avoid obscuring basic ideas. At the same time, it is representative of a wide range of context-aware applications whose execution depends on the resource-saving context as those listed before.

The traditional COP approach would require us to heavily modify our code. For example, if the original code is written in a language for which there is no COP extension, we would have to adopt a new, COP language and to (largely) rewrite the original code. Then, we would have to reorganize the code to explicitly separate behavioral variations and add the activation logic to switch from sequential to parallel mode (or vice versa) according to the context information. While in the specific case this might not present a serious problem, generally this is a really invasive and potentially error-prone change that someone could prefer to avoid. Indeed, we might achieve the desired behavior without changing a single line of code. If we carefully analyze the code and the parallelization problem, we see that the two behavioral variations (i.e., sequential and parallel execution) are closely aligned with the `for` loop language constructs in lines 4 and 5. In other words, the behavior of the application wrt. the desired goal depends on whether the `for` loop constructs run in sequential or in parallel mode. Thus, if we could change how the loops in lines 4 and 5 behave, we would affect the application behavior without modifying the original application code. We could even not adopt a COP language at all.

In traditional COP, behavioral variations are provided as partial methods grouped into layers. When no layer is active, a call to method `m` triggers the standard behavior variation defined in the body of `m`. If a particular layer is active, the variation of method

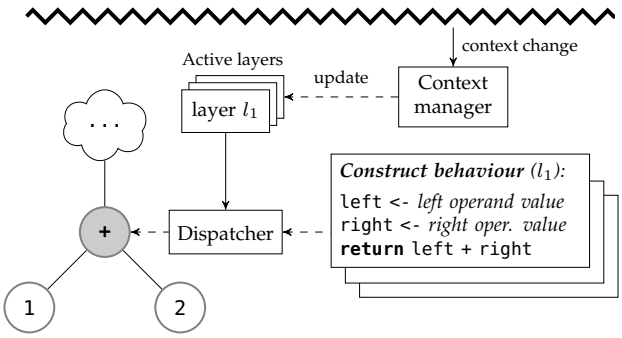


Figure 5.5.: *Multiple behavioral variations dispatched according to the current context.*

`m` defined in that layer is executed. In a sense, the method named `m` is associated with many behaviors defined in different method bodies and the right variation is determined by active layers (context).

Our idea is to *associate behavioral variations with a language feature* and dispatch on them according to some context information. In this view, a language feature and the associated semantic action correspond respectively to COP's partial method's name and body. Indeed, when a language feature is used in the code, its "name" is used to identify the associated semantic action ("partial method") and its "body" is executed. In this approach, variations are implicit in a language feature and need not be explicitly implemented in the application. The approach is illustrated in Fig. 5.5. The addition operation (+ node) is associated with many actions and a dispatcher executes the right action according to the context as follows. Each semantic action has a "layer" label which determines to which layer the action belongs. Layers are implemented on top of dynamic constraints explained in Section 4.3.2. For example, the front action in Fig. 5.5 belongs to the layer l_1. When a node is visited, the dispatcher checks which layers are active and executes the associated actions that belong to the currently active layers.

In contrast to the traditional COP, in our approach layers are not activated explicitly in the application code. Instead, an external context manager notifies the interpreter about the context change by (de)activating layers (see Figure 5.5). In this approach, the interpreter handles a global list of active layers whose order determines the order in which semantic actions are executed. Layer (de)activation is asynchronous with respect to the application execution, i.e., it can occur at any moment of the application interpretation. When the context changes, the interpreter pauses the interpretation, updates the active layers according to the information provided by the context manager and proceeds with the interpretation.

Listings 5.15(a) and 5.15(b) show the implementation of the sequential and parallel for loop variations for the Neverlang.JS interpreter. The implementation details do not matter and the code omits many irrelevant details. The main difference consist in that the variation in Listing 5.15(a) visits the loop body sequentially (line 10-12), while the variation in Listing 5.15(b) spawns a thread for each loop run (lines 24-28). Notice that semantic actions are assigned to layers. The action in line 21 is assigned to the

a) Neverlang module that implements the sequential variation of the for loop language feature

```
module neverlangJS.ForLoop {
 reference syntax {
   For:  Stm <- "for" "(" ForDecl ";" ForCond ";" ForStep ")" Stm;
 }
 role (evaluation) {
  For: .{ // Action pertains to the standard layer
   int start = (int)$For[1].Value; // loop start
   int end = (int)$For[2].Value;   // loop end
   int step = (int)$For[3].Value;  // loop step
   for (int i=start; i < end; i=i+step) {
     eval $For[4]; // evaluate loop body sequntially
   }
  }.
 }
}
```

b) Neverlang module that implements the parallel variation of the for loop language feature

```
module neverlangJS.ParForLoop {
 imports { java.util.concurrent.*; }
 /* reference syntax from the sequential for loop module */
 reference syntax from neverlangJS.ForLoop;
 role (evaluation) {
  For (performance): .{ // The action is grouped into the "performance" layer
   ...
   ExecutorService exec = Executors.newFixedThreadPool(4);
   for (int i=start; i <= end; i=i+step) {
     // ThreadVisit class implements Runnable
     ThreadVisit t = new ThreadVisit($For[4]);
     exec.submit(thread); // Submit the thread for execution
   }
  }.
 }
}
```

c) Event manager script responsible for power resource handling.

```
[[$(acpi -a)==*"off"*]]; plugged=$?
while true; do
  [[$(acpi -a)==*"off"*]]; now=$?
   if ["$plugged"!="$now"]; then
     if ["$now"==true]; then
       # plugged->battery
       mda Mandelbrot deactivate performance
       mda Mandelbrot activate standard
     else
       # battery->plugged
       mda Mandelbrot deactivate standard
       mda Mandelbrot activate performance
     fi
     plugged="$now"
   fi
  sleep 1
done
```

Listing 5.15: *Variants of for loop implementation in Neverlang and the event handler.*

"performance" layer. On the other hand, the action in line 6 does not explicitly declare a layer and is thus implicitly assigned to the "standard" layer. Only actions pertaining to active layers are executed. Layers are activated and deactivated from command line through the `mda` tool which uses RMI to update the layers' state. Listing 5.15(c) shows a context manager script that uses the `acpi` tool to verify whether the laptop is running on battery or it is plugged into the mains (line 4). When a change happens (e.g., the laptop is unplugged from the mains) the layers' state is updated. `mda` takes in input the interpreter instance name (e.g., Mandelbrot), the action to be performed (activate, deactivate) and the layer name. For example, the command in line 8 would deactivate the "performance" layer in the interpreter whose instance is bound to "Mandelbrot" in the RMI registry.

The examples discussed in this section illustrate how open interpreters can be used to dynamically adapt the application to the surrounding context by modifying the underlying interpreter. As discussed in the previous section, the feasibility of adaptation heavily depends on how well are the application-level concepts aligned with language constructs. Similarly, the obtained solutions can be shared among different applications and sometimes among different language implementations. The ability to consider the context at the level of programming language interpreters could be used to provide variations of language constructs that best fit in different situations. For example, an interpreter could be shipped with construct variations that, depending on the context, vary in energy consumption. This could be used in mobile devices to seamlessly modify the behavior of applications without the need to explicitly program such adaptation by the developers of mobile apps.

Alternatively, one could use traditional context-oriented programming approaches to implement the above examples. However, as was briefly discussed in the parallelization example, in some cases the traditional COP would unnecessarily require one to modify the original source code. Furthermore, the behavioral variations would have to be provided a priori and adding new behavior at runtime would imply using alternative tools for dynamic software updating to inject new code. Also, the original application code must explicitly code the variation triggering which can obscure the basic application logic. Instead, open interpreters allow one to separate variations and the triggering logic from the original application code. In addition, one can inject new variations at runtime without modifying a single line of the original code.

5.4. Interpreter Optimization

As discussed in Section 2.2.1, an interpreter is functionally decomposed in many phases, like parsing, type checking, etc. Many interpreters implement an optimization phase which is responsible for optimizing the input source code. Typical optimization techniques include constant folding, dead code elimination, function inlining, etc. Although valuable, these techniques are unable to take into account information that is only available at runtime. Therefore, researchers developed several runtime optimization techniques that include runtime tree rewriting [27, 128], partial evaluation [128], meta-

a) Runtime dispatching on the operands' types

```
1  package neverlangJS;
2  class MathHelper {
3    ...
4    public static Object add(Object l, Object r) {
5     if (l instanceof Integer && r instanceof Integer)
6       return (Integer)l + (Integer)r;
7     else if (l instanceof Integer && r instanceof String)
8       return ...;
9     else ...
10   }
11   ...
12 }
```

b) Neverlang.JS module for the addition operation

```
13 module neverlangJS.AddSyntax {
14   reference syntax {
15     Add: Expr <- Expr "+" Expr;
16   }
17 }
18 module neverlangJS.AddSemantics {
19   import neverlangJS.MathHelper;
20   role ( evaluation ) {
21     Add: .{ $Add.value = MathHelper.add($Add[1].value, $Add[2].value); }.
22   }
23 }
24 slice neverlangJS.Addition {
25   concrete syntax from neverlangJS.AddSyntax
26   module neverlangJS.AddSemantics with role evaluation
27 }
```

Listing 5.16: *Implementation of the Javascript addition operation.*

trace just-in-time (JIT) compiling [11], polymorphic inline caching [62], etc. Although these techniques are highly efficient, they are limited to a specific objective, namely performance, and often need to be pre-integrated in the interpreter. Instead, open interpreters provide a general mechanism for interpreter adaptation which might include application-tailored optimization.

Consider the linguistic component for the addition operation in Javascript, which is a dynamically typed language. Listing 5.16(b) shows the Neverlang.JS implementation. The semantic action simply invokes the helper static method `add` defined in Listing 5.16(a) (line 19) to which it passes the operand values. The `add` method tries to cover all possible operand types through a series of if-then-else statement. The more types are covered the easier it is for developers to write compact code. For example, the developer can simply write `1+"a"` and the plus operator will "do the magic" by concatenating a number and a string ("1a"). If these types were not covered, the Javascript developer would have to define her own method for the desired operand types. However, the vast operand type coverage is traded for a computational over-

a) Addition operation optimized for floating point numbers

```
module neverlangJS.AddFloatSemantics {
  role ( evaluation ) {
    Add: .{ $Add.value = (float)$Add[1].value + (float)$Add[2].value; }.
  }
}
slice neverlangJS.FloatAddition {
  concrete syntax from neverlangJS.AddSyntax
  module neverlangJS.AddFloatSemantics with role evaluation
}
```

b) µDA agent for optimizing the interpreter on floating point numbers

```
slice Add : neverlangJS.Addition;
slice FloatAdd : neverlangJS.FloatAddition;
agent adapt.FloatAdd {
  system-wide {
    interpreter.replaceSlice(Add, FloatAdd);
  }
}
```

Listing 5.17: *Optimization on floating point numbers.*

head [5] as the interpreter might often have to perform long conditional checks before hitting the right operand combination. Suppose we have a computation-intensive algorithm that performs repeated calculations *only* on floating point numbers. The proposed implementation in Listing 5.16(a) has at least three problems. First, it will perform type checking on types that are not present in the algorithm being interpreted. Second, the order of type-checking might not be optimal. For example, if the operand combination for floating point numbers is placed at the end, the interpreter will waste a lot of time by checking other combinations which are known to be absent in the algorithm. Third, even if the check for floating point operands were positioned in the very beginning of the `add` method, it would still perform type checks and casts, even though we know that the algorithm works only with floating point numbers. The already mentioned techniques (runtime tree rewriting, partial evaluation, meta-tracing with JIT, polymorphic inline caching) are able to solve this issue. However, such optimization must be pre-integrated in the interpreter. But, it is unreasonable to believe that a language developer will foresee every possible situation and optimize the language implementation accordingly. Instead, with open interpreters we can plug in an optimized semantic action `a posteriori`, i.e., after the interpreter has been deployed and is already running. This is especially useful for unforeseen situations that were not taken into account when the interpreter was built. Also, it allows one to exploit the valuable application-specific knowledge owned by application developers.

Listing 5.17(a) shows a Neverlang module for the addition operation which is optimized on floating point operands. The semantic action directly adds the operands which

[5] The illustrative implementation is (intentionally) quite naïve, however its simplicity points out well the need for an open implementation (see [128] for a similar choice of simplicity).

are previously casted to `float`. The cast is necessary due to Java's type system. The new slice is made of the original addition syntax define in module `neverlangJS.AddSyntax` and the new semantics defined in `neverlangJS.AddFloatSyntax`. Listing 5.17(b) shows an μDA agent which replaces the original addition slice with the new one that is optimized for floats.

The discussed example is rather simple and the existing optimization techniques would most probably outperform our solution. However, open interpreters allow one to apply such optimization a posteriori and to systems that cannot be shut down. Furthermore, under specific conditions, optimization modules can be shared among different language implementations. Combined with context-awareness discussed in Section 5.3, open interpreters can be used to conditionally optimize the interpreter execution depending on some context information.

5.5. Aspect-Oriented Programming

Open interpreters can be used to provide aspect-oriented programming support to any Neverlang-based interpreter. Indeed, open interpreters borrow some AOP concepts. Hooks are similar to joinpoints, tree patterns are analogous to pointcuts and the agents corresponds to AOP advices. The difference is that in traditional AOP, these concepts are applied at the target language level. Instead, in open interpreters, they are applied to framework-level concepts. This has important implications concerning the applicability. Traditional AOP is language-specific (e.g., AspectJ for Java, PostSharp for C# and VB), while in open interpreters AOP can be applied to any language whose interpreter is open according to Definition 3. In Neverlang, all interpreters are open and can benefit from the framework-level AOP. Furthermore, AOP languages provide fixed joinpoints at the target language concepts like method calls, method executions, object instantiations, constructor executions, field references and handler executions. While in open interpreters, hooks (joinpoints) are fixed (before and after each tree node), they do not target any specific target language level concept. Indeed, tree nodes can represent *any* target language concept and a single node represents a very fine-grained element of the running application. Therefore, from the target-language perspective, open interpreters do not impose fixed joinpoints. The developer can attach extra behavior at whatever target-language concept and implement crosscutting concerns.

Suppose that the interpreter's native stack tracing lacks information valuable for the user. With a simple agent, one could attach the desired behavior *before* and *after* a node that represents a method call is reached. Notice, however, that we cannot simply push extra information on the existing interpreter-side stack trace since it was not programmed to store that kind of information (i.e., we would have a type mismatch). Therefore, we implement an agent-side stack trace as illustrated in Listing 5.18. We first import a custom data structure `StackTrace` which implements a complementary stack trace (line 1). In line 8 we initialize the stack trace. Then we register the agent before and after each method call (lines 10 and 16). In addition, we register the agent after the `catch` statement. Before each call we push the necessary information on the agent-side

```
1  import neverlangJS.utils.StackTrace;
2  import neverlangJS.utils.StackRecord;
3  nt call,_,_ : Call from module neverlangJS.Call;
4  nt catch,_ : Catch from module neverlangJS.TryCatch;
5  agent adapt.ExtendedStackTrace {
6    StackTrace stack;
7    system-wide {
8      stack = new StackTrace();
9    }
10   before call {
11     int pos = call.getSourceLine();
12     String methodName = call.getAttribute("name");
13     // extract other information
14     stack.push(name, pos, /* other info */);
15   }
16   after call, catch {
17     NodeInfo current = interpreter.getCurrentNode();
18     StackRecord rec = stack.pop();
19     if (current == catch) {
20       // do something with rec
21     }
22   }
23 }
```

Listing 5.18: *µDA code for stack tracing.*

stack trace (lines 11-14). In the after hooks of the concerned nodes, we first pop a record off the stack trace (line 18). Then, if the agent was notified at the `catch` node, we perform some operations on the popped stack trace record (the code is omitted as irrelevant).

With respect to traditional AOP, open interpreters have the advantage in that they support a more fine-grained manipulation of construct behavior. Indeed, with open interpreters one is able to attach new behavior before and after every single node. There is potentially no limit in how a developer can extend the application behavior by attaching agents to tree nodes. With a rich set of introspection operations, the technique can be used for debugging as discussed in the next section.

5.6. Debugging

Open interpreters can be used to implement target language debuggers. For example, hooks can be used as breakpoints at which the debugger agent obtains control and prompts the developer with several option on how to proceed. At that point, the developer can use the reflection API to introspect the interpreter state and change it at will. Target language debuggers will register an agent only at nodes that represent relevant concepts for the interpreted language. For example, a Javascript debugger will most probably allow the developer to put a breakpoint only at Javascript expressions and statements (i.e., at nodes represented by nonterminals `Stmt` and `Expr`). Currently,

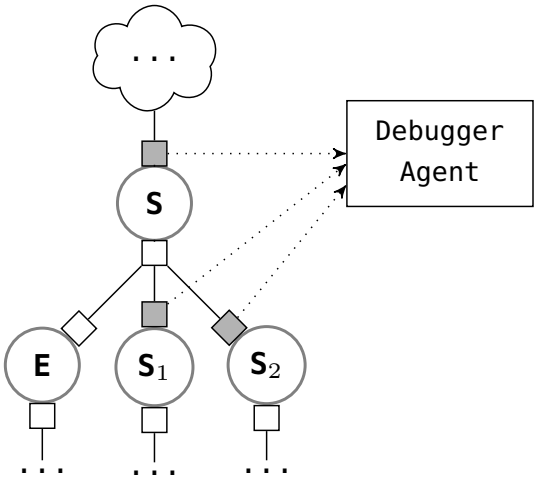


Figure 5.6.: *Debugger.*

we are designing and developing a model that would allow language developers to tag constructs that are meaningful from the target language perspective. These tags would then be used to automatically generate debuggers for target languages.

Listing 5.19 shows a simple debugger for the Neverlang.JS implementation that would break the execution before a statement is executed. This is illustrated in Figure 5.6 where we show a tree representation of the if-then-else statement. The tree has three statements: the if-then-else statement itself (represented by the `S` nonterminal) and the "then" (S_1) and "else" (S_2) branches. When a hook is reached *before* any node represented by the `S` nonterminal, the execution flow is transfered to the debugger agent. The debugger then prompts the user for a debugging command (see Listing 5.19). The example code is deliberately simple and omits irrelevant details, but instead focuses on

```
import java.util.Scanner;
production Stmt : Stmt from module neverlangJS.Statement;
agent debugger.GenericJS {
  before Stmt {
    System.out.println("What would you like to do? (type \"help\" for help)");
    Scanner scan = new Scanner(System.in);
    String input = scan.nextLine();
    switch(input) {
      case ...: ...;
      ...
    }
  }
}
```

Listing 5.19: *Simple debugger that would break the execution before each statement.*

```
action write : Write from module mydsl.Write;
agent adapt.Sandbox {
  system-wide {
    interpreter.replaceAction(write, interpreter.NOPAction);
  }
}
```

Listing 5.20: *Agent that implements "sandboxing" through semantic action neutralization.*

the idea of using hooks as breakpoints.

We used the same approach to build a generic debugger for the Neverlang framework. This debugger can be used to debug any interpreter by targeting framework-level concepts. It allows one to put a breakpoint at any tree node. When a breakpoint is reached, the agent presents the user with a command prompt where commands in a special DSL are accepted. The DSL commands are similar to those of gdb [6], except that they are expressed in terms of the Neverlang framework concepts. The developer can read node attributes, get the next node to be visited, etc.

The developer can write custom ad-hoc debugging scripts for specific purposes. Such scripts are reusable and can even be shared across different language implementation, depending on the concepts they target.

5.7. Security

Open interpreters can be used to implement security features similar to sandboxing and input validation and sanitization. Sandboxing is a security mechanism for isolating running programs from resources that should be protected. Typically, it is used for testing purposes or to execute software that originates from unverified and untrusted sources. Such a mechanism can be implemented through open interpreters by controlling and limiting the effects of linguistic constructs that could potentially harm the protected resources. For example, a construct for accessing files could be neutralized by replacing its semantic action with an empty one. To this purpose, Neverlang provides a "no operation" (NOP) semantic action. Listing 5.20 shows a μDA code snippet that neutralizes the "write" semantic action by replacing it with a NOP action.

Input validation and sanitization are distinct, but normally combined mechanisms to verify and guarantee that the input data will not break the program execution or, worse, harm the system. A classical example is given by the SQL injection hacking technique which consist in placing malicious code in SQL statements, typically through a web page input. To prevent SQL injection, the input is validated and sanitized before the SQL statement is finally executed. Validation consist in checking whether the input data meets specific criteria, e.g., that the input string does not contain standalone single quotation marks. If the input validation fails, sanitization tries to fix it, e.g., by inserting the matching single quotation marks.

[6] `https://www.gnu.org/s/gdb/`

a) SQL statement subject to code injection hack

```
userId = getRequestString("userId");
sqlStatement = "SELECT * FROM Customer WHERE Id = " + userId;
```

b) Agent that performs validation and sanitization.

```
nt call,_,_ : Call from module neverlangJS.Call;
agent adapt.ValidateAndSanitize {
  after call [methodName == "getRequestString"] {
    // validate and sanitize the input
    // which is stored in the return value of the method call
  }
}
```

Listing 5.21: *Input validation and sanitization problem and solution.*

Consider the code in Listing 5.21(a) that prepares an SQL statement to be executed on an SQL engine. The `getRequestString` method returns the value of the specified HTTP Request variable (e.g., `userId`). If `getRequestString` does not perform validation and sanitization, the SQL statement will be subject to code injection. If a hacker inserts as user id the input "`99 OR 1=1`" (without quotes), the `WHERE` condition will always be true, since `1=1` is always true. Therefore, the SQL statement will return the data of all customers.

An agent can extend and open interpreter in order to provide its applications with input validation and sanitization for free. Listing 5.21(b) shows an agent that requests to be registered after the `getRequestMethod` is executed. After it acquires the execution control, the agent perform input validation and sanitization (the actual code is omitted as it is irrelevant). By being application-independent, an agent can be reused in different applications written in the same language.

5.8. Discussion

The examples illustrated and discussed in this chapter show that open interpreters have real-world applicability that go beyond toy examples. Each example emphasized a strong point in favor of open interpreters and we concluded each section with a few comments that draw attention to the strengths of our approach which can be summarized as follows:

- a support for fine-grained behavior extensions that allow one to attach new behavior on single tree nodes;
- the ability to modify a posteriori the language and application behavior, even without stopping the application or interpreter execution;
- the ability to modify application behavior without changing a single line of the application code;

- the support for partial adoption of language evolution that mitigate backward incompatibility issues;
- the ability to share agents among different language and/or application implementations;
- the ability to take into account valuable context information and consequently adapt the interpreter behavior.

As with any approach, open interpreters have their drawbacks which were already discussed in Section 4.6. Here we especially emphasize the need for a support framework to assist the developers in the interpreter updating process. It is clear that users of this approach must be skilled language developers. In order to draw open interpreters closer to a wider user basin, there is a great need for a powerful integrated development environment (IDE). Currently, Neverlang is able to automatically generate editors for target languages. Furthermore, we have a generic debugger for Neverlang-based interpreters. We also have a graphical tool for language updating with a rich set of operations that enable one to adapt a running interpreter simply by using a mouse. On top of these, we are adding more support to assist developers in language updating. Also, there is a work in progress to provide a set of language components for mainstream languages that could be used as black box by developers with a weak background in language developers.

6
Related Work

6.1. Language Extensions

Ever since the birth of the first high-level programming languages, researchers sought for ways to introduce language extensions to better fit the application needs [33]. Mixins are a mechanism that allow one to define shareable specialized behavior to be applied to a variety of components. Although Bracha and Cook [14] are generally attributed to be the first to have written a scientific paper on mixins, the concept itself dates as far back as in the 1960s when Warren Teitelman introduced the mixins extension to Lisp [120, 50][1]. The Teitelman's extension provided the functionality of what would later be called *before*, *after*, and *primary* methods. The mechanism allowed one to specify that a behavior should be executed before or after a primary method. Later, Cannon introduced the Flavors system [16, 126, 88] with similar functionality. Flavor's model heavily influenced the development of Common Lisp Object System (CLOS) which provides an almost identical programming pattern called *standard method combination* [40, 71]. Later, Kiczales and his colleagues at Xerox PARC developed the concept of aspect-oriented programming that provides a mechanism to add behavioral extensions before and after the standard behavior in a modular way [74].

Many mainstream programming languages provide some reflection support, although it is often quite limited. For example, ever since its early days, Java provides a reach reflection library for introspection, but is rather tightfisted in providing ways to introspect the language. With the second version, Java enriched the support for class loaders which allow one to load classes at runtime. This enables one to introduce new behavior which is unknown at development- and load time. To overcome the Java's reflection limitations, developers sometimes use dynamic code generation and compilation [35]. Similar reasoning can be done for other mainstream programming languages.

Scala provides a rather rich set of introspection and intercession operations with

[1] Notice that Bracha and Cook never claimed to be the first authors on mixins. This wrong attribution is evident by the fact that, when citing the literature on mixins, most scientific papers point to Bracha's and Cook's paper entitled "Mixin-based inheritance" [14] and rarely mention the Teitelman's paper entitled "PILOT: A Step Toward Man-Computer Symbiosis". This is reflected on the number of citations which, at the moment of writing this dissertation, is 1095 for Bracha's paper and 44 for Teitlman's. Richard Gabriel's paper "The Structure of a Programming Language Revolution" [50] presents an interesting reading on this topic.

which one can reflect on the whole AST of the input program[2]. Developers can, thus, modify the behavior of the running application. However, differently from open interpreters, reflection in Scala is invasive as it requires the developer to modify the source code. If Scala's reflection API is changed due to language evolution, the original source code would not work anymore. On the other hand, any change to the open interpreters' reflection API would simply imply that reflection cannot be done without updating the agents. But, the original interpreter and the applications running on top of it will continue to work as before.

Few programming languages, such as `Racket` [47] and Scala, with implicits and embedded DSLs, provide mechanisms to support their own extension deriving a new dialect. This approach has the advantage that the extension is done inside the language so you do not need any new skill to develop it, but i) it is limited by the language itself ii) it only supports language extensions and adaptations not the removal of a language feature and iii) the extension is often part of the program that is going to use it and the business logic gets confused in the extension logic (e.g., look at the `Racket` definition for the textual adventure in [47]). Neverlang with its reflection API clearly separates the interpreter and its adaptation from both the program running the adaptation (the μDA programs) and the program executed on the adapted interpreter; these adaptations can be done during the program execution and the removal of language features are supported as well. In the same category of Neverlang we could enlist the language workbenches and frameworks such as Spoofax [67], Lisa [86], JastAdd [58], MPS [125], LTS [37], Xtext [9] and Melange [41]. To some extent all of them support modular development of general-purpose and domain-specific programming languages as Neverlang does. Therefore they have the potential architecture to support open interpreters and their dynamic adaptation but, as far as we know, none of them implements a mechanism to support the dynamic modification of a running interpreter nor do they provide the developer with a reflective API to drive the adaptation as we describe in this dissertation.

Instead of providing language extensions, one can extend the VM to introduce reflectional features. This is what was done by metaXa [54], Guaraná [91] and Iguana/J [103] for the JVM. The main disadvantage of this approach is that it sacrifices the portability and if the maintenance of non-standard VM-based MOPs is discontinued, the applications relying upon them will have to be modified. Instead, in open interpreters application and interpreter code are completely separated from the agent adaptation code. Therefore, any discontinuity in the maintenance of the Neverlang's VM does not break the original application, even if without the extra behavior.

6.2. Metaobject Protocols

A lot of research was done to overcome overcome the limited support for intercession in mainstream programming languages. Chiba [34] developed OpenC++, a compile-time MOP that allows one to extend the behavior of a program written in C++. Following

[2] http://docs.scala-lang.org/overviews/reflection/symbols-trees-types.html

Chiba's idea, Tatsubori *et al.* [119] developed a compile-time MOP for Java, called OpenJava. Both solutions share the same idea and principles: on per-class basis metaobjects instruct the meta-compiler on how to translate language components (classes, methods, etc.) before the application is finally compiled to either byte- or machine code. These compile-time translations can inject extension code to add new behavior to existing language constructs. Both OpenC++ and OpenJava require modifications to the source code in form of special comments to instruct the meta-compiler about which metaobjects to use for the translation. Differently, our approach requires no modification of the source code and provides reflection for free to *all* interpreters built with Neverlang. Furthermore, open interpreters support runtime interpreter intercession and can, thus, take advantage of valuable runtime information not available at compile-time. Also, our approach does not require recompilation and the reflection code can be reusable under specific conditions.

Jinline [118] is a load-time MOP, integrated in the integrated to the Javassist [35] framework, for altering Java semantics through bytecode manipulation. There are many similarities between Jinline and open interpreters. For example, jinlers, which correspond to our agents, have to register for notifications about language mechanism occurrences (e.g., message send, cast, etc.). When notified, a jinler can inline a method before, after or instead of the language mechanism. Inlined methods can also be provided with dynamic information. The main difference between Jinline and open interpreters is that Jinline is defined only for Java, while open interpreters target framework-level concepts and thus provide reflection support to every language built on top of that framework (e.g., on top of Neverlang).

Kava [127] and Reflex [116] are runtime MOPs that allow one to change the behavior of Java classes. With open interpreters they share the idea of hooks to transfer the execution from base- to meta-level. Again, open interpreters provide reflection support on framework-level concepts and thus provide reflection for free to every interpreter build on top of that framework. As stated by Tanter *et al.* [116], a reflective control over method invocation is all what is needed in a large range of applications. However, Kiczales *et al.* [73] outline the 90/10 principle according to which a there is always a small group of off-the-charts developers (10%) which seek for non-standard behavior which, applied to this context, would mean that they might need control over other events, like object creation, arithmetic operations or others. Open interpreters provide a fine-grained reflection support for every language feature.

GEPPETTO [105] supports the adaptation of applications at runtime through an unanticipated partial behavioral reflection. In many aspects, it is similar to open interpreters. It's a runtime MOP with the concept of hooks, support for fine-grained selection of language concerns and dynamic predicates. GEPPETTO's spatial and temporal patterns correspond, respectively, to our tree patterns and dynamic constraints. In GEPPETTO, hooks are installed dynamically, which is faster but requires the BYTESURGEON bytecode manipulator. On the other hand, the proposed model of open interpreters have fixedly positioned hooks which have to be checked, this is slower, but does not require external tools. However, the definition of open interpreters does not impose how extended behavior should be provided. GEPPETTO is designed for Smalltalk,

while open interpreters target framework-level concepts and, hence, work on every language built on top of the that framework.

One of the main disadvantages of runtime MOPs is the runtime overhead. To overcome this issue, researches proposed several solutions which include partial evaluation [106, 83, 112], partial behavioral reflection [117], trace-based compilation [6, 51], inlining, dispatch chains [82, 32] and others. A multi-stage technique as the one presented in MetaOCaML [3] and applied to the `Black` reflective programming language could be used to drive out some complexity and performance penalties from the Neverlang runtime in the future. The main difference (that applies also to the other techniques) is that Asai [3] applies its optimization technique to a language (`Black`) that provides reflective facilities to its programs whereas in our case, the open interpreter could implement a language without any support to reflection: to be clear, the reflective tower is separate from the execution model of the application (NVM→interpreter→application).

6.3. Runtime Software and Interpreter Adaptation

Some language development frameworks provide support for adaptation, usually restricted to specific objectives. For example, Truffle uses runtime tree rewriting to specialize tree nodes with the aim of optimizing execution performance [128]. The developer writes a priori the possible specializations based on the types the interpreter will support. The best specialization is then automatically chosen by the runtime according to manually specified coercion rules. Specialized tree is then further compiled by a JIT. To our knowledge, Truffle does not support (without recompilation and re-execution) the addition of specializations once an interpreter is deployed and running. Open interpreters, instead, allow a posteriori runtime interpreter manipulation. Also, reflection is defined in terms of framework-level concepts which can be shared across different language implementations.

Kollár and Forgáč [76] investigated on the possibility of adapting a programming language interpreter during its execution. The proposed approach is based on abstract-syntax-tree rewriting—as implemented by Truffle [128]—and code injection at the parse tree nodes. Except for working on parse trees, their approach differs from ours in that their solution uses code injection, while we use self-contained agents that are dynamically hooked either before or after the parse tree node. Moreover, open interpreters are able to confine the effects of the language evolution to a specific application features thanks to microlanguages. In addition, our μDA DSL provides a more user-friendly way to specify the language evolution whereas the approach proposed by Kollár and Forgáč does not foresee any facility to deal with these aspects.

The research in the area of adaptive systems produced solutions that can be categorized as either *architectural* [92, 77] or *linguistic* [109, 52], or a combination of both [107]. With architectural approaches one can adapt an application by either adding, removing, or substituting one of its components. Traditional dynamic software updating approaches (such as JavAdaptor [100, 99], DUSC [93], Rubah [98] and JRebel [65]) are

a variant of architectural approaches in which evolution does not simply reconfigure the system, but also supports changes in the code. However, these approaches suffer from performance decay (due to indirections introduced by table forwarding and object proxies) [100, 93], limited program adaptation (no class re-positioning [65], either limited—UpgradeJ [10]—or no support for schema changes—HotSwap [44]), misalignment between design and executable code [89, 24] and a general difficulty in maintaining the evolved code [43, 45].

Linguistic approaches to software adaptation provide means to change both the application code and its behavior. Supporting adaptation (or evolution) through ad hoc linguistic constructs was first introduced by Mens *et al.* [85]. This approach focuses on the introduction of non-functional features. We can roughly classify linguistic approaches to software adaptation in three categories: *aspect-oriented programming* [74], *reflection/meta-programming* [81] and *context-oriented programming* [60]. Both aspect-oriented programming and reflection provide means to inject new behavior into an application while keeping the new and the original code separated. In aspect-oriented programming the new code can be completely decoupled from the rest of the application code. Very few aspect-oriented languages—e.g., CaesarJ [2] and AspectJ's load-time weaving [72]—have any (or limited) support for dynamic weaving to update a running system. Despite the poor support for dynamic updating, a few dynamic software updating approaches were developed on top of aspect-oriented programing [129, 56, 130]. Open interpreters can be used as a mechanism to implement dynamic aspect-oriented weaving at the language-level [29] as was already illustrated in this dissertation.

Context-oriented programming allows one to specify behavioral variations through specific language-level abstractions. In context-oriented programming, context is a first-class construct of a programming language [68, 60, 55]. The system dynamically selects the best behavior or a combination of behaviors based on contextual information and selection conditions. Context-oriented programming enables one to separate context-dependent crosscutting concerns. A drawback of this approach is that the computation and coordination aspects are often interleaved and behavioral variations must often be explicitly activated [108]. Moreover, the adaptation is implemented on a per-application basis. Mixins [14, 48] and traits [110] are other linguistic approaches to software adaptation and evolution. These approaches enable one to extend a class with extra methods and override/enrich the existing methods. Matriona [111] is a framework that uses mixins to support dynamic adaptation.

Meta-programming approaches rely on the reflective features support of the programming language and its runtime system. Reflection is used to observe and adapt the underlying program [90]. Example frameworks that use Java's reflection facilities to support software adaptation are Chisel [69], PKUAS [63] and mChaRM [17].

6.4. Interpreter Composition

Interpreter composition enables language interoperability through cross-language API. The feasibility and the difficulty of composition depend on several factors. Consider

composing interpreters $Y_{\mathcal{L}1}$ and $Y_{\mathcal{L}2}$. If the interpreters are written in the same language then, e.g., $Y_{\mathcal{L}1}$ can import $Y_{\mathcal{L}2}$ and use the functionality exposed by $Y_{\mathcal{L}2}$. In the best case scenario, the two interpreters will share the same data structures. In a more realistic scenario, some glue code will be necessary to ensure correct data type conversion. If interpreters are not written in the same language, glue code is necessary to enable the execution control to flow from one interpreter to the other. Often, the interpreters must invasively be modified to achieve composition. Examples of interpreter composition are: Python and Prolog (Unipycation) [8]; Java and tuProlog [42]; Icon and Prolog [79]; Lisp and Prolog (LOGLISP) [104]; Smalltalk and SOUL [57].

Due to implementation misalignments, gluing two or more interpreters can negatively impact the execution performance. Techniques like partial evaluation [106, 83, 112, 128] and trace-based just-in-time compilation [12, 11, 13] can greatly alleviate the performance decay.

Although the main objective of open interpreters is not the composition of different languages, open interpreters are enablers for runtime composition. Agents could act as glue code and perform control flow pass from one interpreter to the other, convert data types, etc. The same performance and feasibility concerns discussed above are valid in this case. Ideally, both interpreters will be written in the same language or framework (e.g., Neverlang). However, the primary aim of open interpreters is not the composition of different interpreters, but rather to enable dynamic adaptation of existing interpreters without affecting the application code.

6.5. Quantification

The problem of quantification in the context of aspect-oriented programming was extensively discussed by Filman and Friedman [46]. In AOP, the developer uses a sort of regular expressions (pointcut descriptions or PCDs) to declaratively register handlers (advices) with a set of events. Events are expressed in terms of joinpoints. In open interpreters terminology, PCDs correspond to μDA tree patterns, advices correspond to agents and joinpoints correspond to hooks. Due to its fine-grained event model, with hooks positioned before and after *every single* node in the parse tree, open interpreters are not subject to quantification failure [114, 101] although, at the current state, the expressiveness of μDA might limit somehow the range of hook selection. Rajan and Leavens [101] introduce quantified, typed events that enable a programmer to add event announcement for an arbitrary statement in the base module, although such events must be explicitly triggered.

The problem of querying and mining graph-structured data, in our case the program parse tree, is well-known to be challenging. Several approaches can be found in the literature, to cite the closest to our μDA DSL we have: `Blueprint` [23], `CARMA` [70]. The aspect-oriented Blueprint language [23] to capture fine-grained definition of join points exploits parsing over graph-grammars to match an incomplete graph-pattern—the description of where a join point should be—on the application control flow graph [25]. `CARMA` [70] exploits *intensional views* [84] to describe some structural properties of a

program and the logic metaprogramming language Soul to gather all the points of a program call graph satisfying the provided intensional view (Soul behavior does not differ much from the way Prolog and Datalog calculate their knowledge base). Fortunately, the path queries that we have to express in μDA are much easier than those supported by Blueprint and CARMA or other graph query languages as G [38, 7] and it can rely on the Neverlang VM architecture that provides several hooks that ease the matching task. Anyway, we are planning to extend μDA with a richer matching language closer to the one used in CARMA.

7 Conclusions

This dissertation presented the concept of open interpreters and a possible tree-based model that can be integrated in development frameworks for building tree-based interpreters. Our prototype implementation in Neverlang shows that the idea is feasible and has real-world applications that go beyond toy examples, as shown in Chapter 5. The prototype fully supports introspection and intercession of interpreter components. Open interpreters allow one to modify and extend language and application behavior in a fine-grained manner through adaptation agents that are software entities completely separated from the interpreter or the application code. Therefore, open interpreters enable one to modify the application behavior without modifying a single line of its code. Modifications can be applied a posteriori, i.e., after the interpreter is deployed, and there is no need to stop the interpreter execution. With open interpreters, one can selectively modify the behavior of specific occurrences of language constructs in the application. Also, the adaptation of open interpreters can be guided by valuable context information. By targeting framework-level concepts, the reflection can be used on any language whose interpreter is open according to the definition provided in this dissertation. The illustrated examples showed that language evolution through open interpreters can be successfully used in different domains. We illustrated their usage on preserving backward compatibility, on building linguistic tooling (e.g., debuggers), on optimizing resource usage, on providing accessibility support, etc, all these without modifying the original application code. Indeed, language evolution through open interpreters is completely non-invasive for the application. Consequently, if the development and maintenance of the reflection API should one day be interrupted, that would not affect existing interpreters and applications running on top of them. Since the reflection code is separated from the interpreter or the application code, they would continue to work, although without the extra behavior. In Appendix A we showed the formal foundations that to some degree guarantee that the reflection operations will not break the interpreter.

A
Composition Soundness

Since modularization fosters development in isolation, grammar attributes could be undefined or used inconsistently due to the lack of coordination. In this Appendix we present the 1) operational semantics for tree-based interpreters, 2) a type system that permits to trace attributes and statically validate the composition against attributes lack or misuse and 3) a correct and complete type inference algorithm for this type system. The proofs are provided in [21].

A.1. Syntax Formalization

In this section, we provide a formalization of the syntax, similar to the one introduced in Section 2.1.1, except that we provide some extra definitions for easier discussion on composition soundness. For simplicity, we consider as the minimal unit of modularity a component which defines both syntax and semantics[1]. Furthermore, components may define only one semantic phase (e.g., type checking, execution, etc.). A component, thus, defines a portion of a grammar as a set of productions and a the corresponding semantic actions. In the following, we refer to the language we are defining as *target language*. In this Appendix, we will use the terms "slice" and "component" interchangeably.

Productions and grammars. As discussed in Section 2.1.1, a grammar is a quadruple (Σ, N, S, Π) where Σ is the set of terminals, N the set of nonterminals, S the start nonterminal, and Π the set of productions. To the purpose of language composition, terminals are of no importance, therefore we will not include them in the component formalization. A *production* is a pair of a nonterminal and a sequence of nonterminals, denoted by $X_0 \to X_1 \cdots X_q$, where $q \geq 0$. The empty sequence is denoted by ϵ. We use the metavariable X with subscripts and superscripts to range over nonterminals. Moreover, with P we denote *a subset of the productions of the grammar* (not necessarily all). Productions are uniquely identified by *labels*, p, with subscript or superscript if needed.

Definition 4. *Let p be the label for $X_0 \to X_1 \cdots X_q$,*

1. *$p[i]$ with $i = 0, \ldots, q$ refers to X_i where i represents the nonterminal position in the production p, $p[0]$ refers to the left-side nonterminal of the production*
2. *$|p| = q$*

[1] In Neverlang, this would correspond to slices, although Neverlang goes a little further in decomposition by allowing both syntax and semantics to be separately defined and as reusable modular units.

3. $NT(p,i) = X_i$, *for* $i = 0,\ldots,q$ *and*
4. $NT(p) = \cup_{0\le i\le q}\{NT(p,i)\}$.

Definition 5. *Given a sequence of productions* $P = p_1 \ldots p_m$,

1. $\mathcal{L}_P$ the set of labels of the production in P,
2. $NT(P) = \cup_{1\le k\le m} NT(p_k)$ *is* the set of nonterminals in P,
3. $Def(P) = \cup_{1\le k\le m}\{NT(p_k,0)\}$ *is* the set of nonterminals defined in P *and*
4. $P\restriction X = \{p_k \mid NT(p_k,0) = X\}$ *is the subset of* P *whose productions have* X *as the left-side nonterminal.*

In the following we give a definition of a grammar, which is slightly more restrictive, of the standard one.

Definition 6 (Grammar)**.** *A sequence of productions* P is a grammar, *if* $NT(P) = Def(P)$ *and there is a* start nonterminal *which occurs only on the left-side of a production, that we call the* start production*. We denote grammars with* $\mathcal{G}$.

In other words, a sequence of productions P is considered a grammar only if all it's nonterminals are defined, i.e., they all appear at least once on the left-hand side in one of the productions.

Components and Language for Semantic Actions. As mentioned above, components are formalized with a single semantic phase and one action per production. Actions are defined in terms of statements of the language defined by the following grammar.

$$
\begin{array}{lcll}
s & ::= & \mathtt{u}unit \mid \mathtt{if}\ e\ \mathtt{t}then\ s\ \mathtt{e}lse\ s \mid s;s \mid p[i].a = e \mid \mathtt{e}val\ p[i] \\
e & ::= & v \mid p[i].a \mid op(e_1,\ldots,e_n) & \\
v & ::= & \mathtt{t}r \mid \mathtt{f}ls \mid n & v_e ::= \mathtt{u}nit \mid v
\end{array}
$$

A statement can be the null statement unit, a conditional, a sequence of statements, an expression, an attribute update and/or definition, $p[i].a = e$, or the execution of a semantic action, e$val\ p[i]$ where $p[i]$ specifies the nonterminal at position i in the production labeled p in P. Expressions can be integer or boolean constants, the value of an attributes of instances of nonterminals ($p[i].a$), or the application of some operators to expressions. In the examples, we will use operators such as + and ==. Values are the results of the evaluation of expressions and can be assigned to attributes. Extended values v_e include unit, which is the value resulting from the execution of a statement and therefore also of an action.

Definition 7 (Components and Component Composition)**.**

- *Given a sequence of productions* P, *a* component $\mathcal{S}_P$ *on* P *is a set of* semantic actions *labeled by the productions in* P *denoted by* $\{p : .\{s\}. \mid p \in P\}$.
- *Let the labels of productions in* P *and* P' *be disjoint. The composition of components* $\mathcal{S}_P$ *and* $\mathcal{S}_{P'}$, $\mathcal{S}_P \circ \mathcal{S}_{P'}$, *denotes the component,* $\mathcal{S}_P \cup \mathcal{S}_{P'}$ *on* $P\,P'$.

Notice that this is a refinement of the definitions provided in Section 2.3 where semantic actions were simply denoted with a_k, while here we define them in terms of the language introduced above. In other words, $a_k = .\{s\}$.

In the component definition we require that each production is associated with a semantic action. Notice, however, that we can always associate a production with the null statement u*nit* to cover situations where a production requires no action. Composition of components is associative and commutative and since, in a component, productions and the corresponding semantic actions can be relabeled, we can always define the composition of two components.

A.2. Operational Semantics

We provide the small-step semantics for semantic actions by describing how the execution of its statements affects the attributes associated with a syntax tree for a given source code. The evaluation of semantic actions does not affect the tree structure. Attributes are conceptually separated from the syntax-tree.

Syntax-tree and Attributes. We begin by formalizing the syntax tree which is a data structure that represents the input source code as a result of parsing. Any subtree of a syntax tree is associated with a production p of $\mathcal{G}$ and contains subtrees for strings generated by the nonterminals on the right-hand-side of p. If p has an empty sequence of nonterminals on the right-hand-side the node is a leaf. Subtrees are assigned a unique identifier.

Let I be a denumerable set of identifiers with *id* being a metavariable ranging on elements of I.

Definition 8 (Syntax-tree)**.** *Let $\mathcal{G}$ be a grammar.*

- $\eta \equiv id : (p, \eta_1 \cdots \eta_q)$ is a syntax-tree for the production p *if* $p : X_0 \to X_1 \cdots X_q \in \mathcal{G}$ *and* $\forall i\ 1 \le i \le q\ \exists p' : X_i \to \cdots \in \mathcal{G}$ *such that* η_i *is a syntax-tree for* p'*.*
- η is a syntax-tree for a string of $\mathcal{G}$*, written* $\mathcal{G} \vDash \eta$*, if* η *is a syntax-tree for the start production of* $\mathcal{G}$ *and all the ids in* η *are distinct.*
- *Given a syntax tree* η*,*
  - $\eta(id) = \eta'$ *if* $\exists p, \eta_1, \ldots, \eta_q$ *such that* $\eta' = id{:}(p, \eta_1 \cdots \eta_q)$ *occurs in* η*. If id does not occur in* η*,* $\eta(id)$ *is undefined.*
  - *The* domain of η*,* $dom(\eta) = \{id \mid \eta(id)$ *is defined*$\}$*,*

Example 1. Consider the productions `IF`, `INT` and `DBL` in Listing 2.9 along with the start production `S: S <- Exp;`, where `S` is the start symbol, which is added to transform the productions into a grammar according to Definition 6. This production is associated with the action:

```
S: .{ eval $S[1]; $S[0].val = $S[1].val; }.
```

The input string `"if 1 then 2 else 3"` would generate the following tree η:

$$id_1 : (\mathtt{S}, id_2{:}(\mathtt{I}F, id_3{:}(\mathtt{I}NT, \epsilon)\ id_4{:}(\mathtt{I}NT, \epsilon)\ id_5{:}(\mathtt{I}NT, \epsilon)))$$

Therefore, $dom(\eta) = \{id_i \mid 1 \leq i \leq 5\}$, $\eta(id_1) = \eta$ and $\eta(id_3) = id_3{:}(\mathtt{I}NT, \epsilon)$.

Mappings are used to associate attributes with nodes of syntax trees. A *mapping*, m, from the set B to C is a partial function from B to C with finite domain. We write $m = [b_1 \mapsto c_1, \ldots, b_n \mapsto c_n]$ and $m(b_i) = c_i$. The empty map is denoted by $[\ ]$. If $m = [b_1 \mapsto c_1, \ldots b_n \mapsto c_n]$ the *domain of* m, $dom(m) = \{b_1, \ldots, b_n\}$. The mapping $m[b' \mapsto c']$ is such that $m[b' \mapsto c'](b') = c'$ and $m[b' \mapsto c'](b') = m(b)$ for $b \neq b'$.

Definition 9 (Attribute store)**.** *Given a syntax-tree η, to represent the values of the attributes associated with nodes of η, we define* attribute stores*, denoted by μ, which are mappings from I to mappings from $\mathcal{A}$ to values, such that $dom(\eta) = dom(\mu)$.*

Consider the syntax tree η of Example 1. The attribute store $[id_1 \mapsto [\mathtt{val} \mapsto 2], id_2 \mapsto [\mathtt{val} \mapsto 2], id_3 \mapsto [\mathtt{val} \mapsto 1], id_4 \mapsto [\mathtt{val} \mapsto 2], id_5 \mapsto [\ ]]$ says that the node associated with the condition part of the if-then-else construct (id_3) has the attribute v*al* with value 1. For the nodes associated with the start symbol (id_1), the if construct (id_2) and the then condition (id_4) the attribute v*al* is defined and has value 2. And, finally, there no attribute is defined for the node id_5 associated with the else condition. This attribute store is the result of the evaluation of the action associated with the production `S` starting with an attribute store in which all nodes have no defined attributes. Note that, for id_5 the attribute `val` is undefined because `toBool(1)` is true and consequently the node id_5 is never evaluated.

Run-time Terms and Configurations. To define the small-step execution of the language for semantic actions, we need to refer to:

- a (generic) syntax tree η,
- the attribute store associated with η which stores the attributes and their values that are currently defined for η,
- the term t (which can either be a statement or an expression), that is currently evaluated.

One-step of evaluation produces a new term and may modify the attribute store μ. We define the judgment of the reduction relation as follows:

$$\eta \vDash t \,|\, \mu \to t' \,|\, \mu'$$

The syntax-tree η is put on the left of $\vDash$ because it never changes during evaluation.

Run-time configurations are pairs of terms and attribute store denoted by $t \,|\, \mu$, but in order to understand μ we also need to refer to the specific η.

In the language for semantic actions, the nodes of syntax-trees are referenced to by labels of nonterminal instances in productions ($p[i]$). In the run-time configuration, these labels are substituted by the identifiers of the node they denote (given the node on which the current action is executed). The *run-time terms*, i.e., the terms in the run-time configuration, are defined by rewriting the language for semantic actions in which we substitute:

$$\eta \models op(\overline{v}) \,|\, \mu \to v \,|\, \mu \quad \text{if} \quad \widetilde{op}(\widetilde{\overline{v}}) = \tilde{v} \quad \text{(E-Op)} \qquad \eta \models \texttt{unit}; s \,|\, \mu \to s \,|\, \mu \quad \text{(E-Seq)}$$

$$\eta \models \texttt{if tr then } s \texttt{ else } s' \,|\, \mu \to s \,|\, \mu \quad \text{(E-IfTrue)} \qquad \eta \models \texttt{if fls then } s \texttt{ else } s' \,|\, \mu \to s' \,|\, \mu \quad \text{(E-IfFalse)}$$

$$\eta \models id.a \,|\, \mu \to \mu(id)(a) \,|\, \mu \quad \text{(E-GetA)} \qquad \eta \models id.a{=}v \,|\, \mu \to \texttt{unit} \,|\, \mu[id{\mapsto}\mu(id)[a{\mapsto}v]] \quad \text{(E-SetA)}$$

$$\frac{\eta(id) = id{:}(p, id_1{:}(\cdots)\cdots id_{|p|}{:}(\cdots)) \quad p : .\{s\}. \in \mathcal{S}_\mathcal{G} \qquad s' = (s[p[0] := id])[p[i] := id_i]_{1\le i\le |p|}}{\eta \models \texttt{eval } id \,|\, \mu \to s' \,|\, \mu} \text{ (E-Eval)}$$

$$\frac{\eta \models e \,|\, \mu \to e' \,|\, \mu'}{\eta \models op(\overline{v}, e, \overline{e}) \,|\, \mu \to op(\overline{v}, e', \overline{e}) \,|\, \mu'} \text{ (EC-Op)} \qquad \frac{\eta \models s_1 \,|\, \mu \to s_1' \,|\, \mu'}{\eta \models s_1; s_2 \,|\, \mu \to s_1'; s_2 \,|\, \mu'} \text{ (EC-Seq)}$$

$$\frac{\eta \models e \,|\, \mu \to e' \,|\, \mu'}{\eta \models \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \,|\, \mu \to \texttt{if } e' \texttt{ then } s_1 \texttt{ else } s_2 \,|\, \mu'} \text{ (EC-If)} \qquad \frac{\eta \models e \,|\, \mu \to e' \,|\, \mu'}{\eta \models id.a = e \,|\, \mu \to id.a = e' \,|\, \mu'} \text{ (EC-SetA)}$$

Figure A.1.: *Rules of operational semantics.*

- $p[i].a$ with $id.a$,
- $p[i].a = e$ with $id.a = e$ and
- e$val\ p[i]$ with e$val\ id$.

Operational Semantics Rules. Operational semantics rules, shown in Fig. A.1, define how the execution of a language construct uses/modifies a run-time configuration. In the rule (E-Op) with $\tilde{v}$ we mean that the integer or boolean value corresponds to the numerals or tr and fls tokens of the language respectively, and similarly $\widetilde{op}$ denotes the function that corresponds to the symbol op of the language. The interesting rules are those dealing with attributes. Rule (E-GetA) returns the value of the attribute a of id. The term is "stuck" if a is not defined for id. Rule (E-SetA) modifies the attribute store μ by defining (or overriding the value of) the attribute a to v. The evaluation, being a statement, returns unit. Finally, rule (E-Eval) replaces e$val\ id$ with the action associated with the production, p, generating the id node. In the action, instances of nonterminals $p[i]$ are substituted by the identifiers corresponding to the child node i, and $p[0]$ is substituted by id. This starts the visit of the node corresponding to id. The last four rules specify the evaluation order, which is the standard evaluation of imperative/functional programming languages.

Let id_r be the root node of η and $\eta(id_r) = (p_r, id_1 \cdots id_{|p_r|})$. The *initial configuration* of the evaluation of η in $\mathcal{S}_\mathcal{G}$ is $s_{in} \,|\, \mu_{in}$ where:

$$s_{in} = (s_r[p_r[0] := id])[p_r[i] := id_i]_{1\le i\le |p_r|} \quad \text{and} \quad \mu_{in} = [id_j \mapsto [\ ]]_{1\le j\le n}$$

For instance, for the Ex. 1, let $\mu_{in} = [id_j \mapsto [\ ]]_{1\le j\le 5}$ the initial configuration is:

$$\texttt{eval}\ id_2; id_1.val = id_2.val \mid \mu_{in}$$

Applying the rule (EC-Seq) with the application of (E-Eval) over the line we get:

$$\eta \models \texttt{eval}\ id_2; id_1.val = id_2.val \,|\, \mu_{in} \to s'; id_1.val = id_2.val \,|\, \mu_{in}$$

where s' is the action associated with the production labeled by `IF` in Listing 2.9 with `IF[`i`]` replaced by id_{i+2} for $i = 1, \ldots 3$.

A.3. Type System

We now introduce a type system that traces attribute definitions and prevents their erroneous usage. The tracing of definitions is done compositionally by type-decorating semantic actions which, in turn, are used to decorate components. Given this type information, we are able to type check and decorate the composition of two or more components.

We define $\mathcal{A} = \{\overline{a}\}$ as a set of attributes with a fixed type. Fixing a type allows us to focus on the "definedness" of attributes rather than on their effective type, which is an orthogonal problem with a wide range of solutions. T_a denotes the type of the attribute a.

Definition 10. *A* typed component, $\mathcal{TS}_P$, *is a set of* decorated actions *which are labeled by the productions in P denoted by $\{p : (R, D).\{s\}. \mid p \in P \wedge R, D \subseteq \mathcal{A}\}$. Given a $p : (R, D).\{s\}. \in \mathcal{TS}_P$*

- *R, called the* required set of attributes*, is a set of attributes of the nonterminal $p[0]$ that ensure the correct execution of s, and*
- *D, called the* defined set of attributes*, is the set of attributes that are surely defined for $p[0]$ by the execution of s.*

To type check semantic actions, we trace attribute definitions through attribute contexts which are defined as follows.

Definition 11 (Attribute context)**.** *An* attribute context Ψ for p *is a subset of the pairs of nonterminals in p and their attributes. That is, $\Psi \subseteq \{(p[i], a) \mid 0 \leq i \leq |p| \ \wedge \ a \in \mathcal{A}\}$.*
Given Ψ, we define $\Psi(p[i]) = \{a \mid (p[i], a) \in \Psi\}$. We say that Ψ refers to *p if Ψ is an attribute context for p.*

To collect information about attributes that are required and/or defined by the execution of semantic actions we define the nonterminal environment as follows.

Definition 12 (Nonterminal Environment)**.** A nonterminal environment Γ for a set of production P *is a set*

$$\{X_1{:}(R_1, D_1), \ldots, X_n{:}(R_n, D_n) \mid X_i \in NT(P) \wedge R_i, D_i \subseteq \mathcal{A}(1 \leq i \leq n)\}.$$

We assume that all nonterminals X_i are distinct. If $X{:}(R, D) \in \Gamma$, then the successful execution of any semantic action associated with a production defining X depends on the definedness for the node associated with X of some of the attributes in R. On the other hand, the execution of any of these semantic actions guarantees that, at the end, at least the attributes in D will be defined. Given a set of nonterminals M: $\Gamma{-}M = \{X{:}(R, D) \mid X{:}(R, D) \in \Gamma \wedge X \notin M\}$ and $\Gamma{\restriction}M = \{X{:}(R, D) \mid X{:}(R, D) \in \Gamma \wedge X \in M\}$.

For simplicity, and given the language definition for semantic actions defined in Section A.1, we consider just the following small set of primitive types:

$$T = \texttt{U}nit \mid \texttt{I}nt \mid \texttt{B}ool$$

where U*nit* is the type of statements, whereas I*nt* and B*ool* are the types for expressions.

The type judgment for terms t that represent a semantic action is

$$\Gamma; \Psi \vdash_p t : T; \Psi'$$

where Γ is a nonterminal environment, Ψ and Ψ' are attribute contexts and T is a type. The judgment should be read as: in the nonterminal environment Γ and attribute context Ψ, the term t has type T and its evaluation defines the attributes for the occurrences of the nonterminals of p conforming to Ψ'. The judgment is relative to a production p, since we have to check the correctness of instances of nonterminals. For uniformity, we use the same judgment for statement and expressions, even though expressions will always have $\Psi' = \varnothing$, since their evaluation cannot define attributes.

The type rules for the judgment $\Gamma; \Psi \vdash_p t : T; \Psi'$ are given in Fig. A.2. Rule (T-Sub) is a standard weakening of both required and defined attributes. It says that, if from an attribute context Ψ_1 we derive that t is correct, then we can derive the result also assuming a bigger attribute context. On the other hand, we derive that, if the execution of t defines the attributes in the attribute context Ψ'_1, its execution also defines a subset of Ψ'_1. The rules for expressions, excluding access to attributes, are obvious. Rule (T-Seq) says that, for a sequence of statements $s_1; s_2$, the attributes defined by the execution of s_1 are available during the execution of s_2. Since both s_1 and s_2 must be statements their type must be U*nit*. For a conditional statement, rule (T-If), the condition is a boolean expression, both branches are statements, so they must have type U*nit* and they must define the same set of attributes. This is not a restriction because using the rule (T-Sub) we can weaken the attribute contexts and make them equal. For an access to an attribute, a, of a nonterminal instance $p[i]$ to be correct, rule (T-GetAtt), the attribute context must contain the pair $(p[i], a)$. This could be for $i \neq 0$ only when the execution of the statements of the action associated with p preceding the evaluation of the current expression has defined a for $p[i]$. When $i = 0$, the attribute could have been in the required set of attributes of the action associated with p, i.e., it has been defined for $p[0]$ before the execution of the action. In rule (T-GetAtt), the type of the expression has to be equal to the type of the attribute to which it is assigned. Since these are statements their type is U*nit* and they define the attribute a of $p[i]$. Finally, to check e*val* $p[i]$ we have to refer to the nonterminal environment Γ. Let $X = NT(p[i])$ and $\Gamma(X) = (R, D)$, the attribute in R must be defined before the execution of a semantic action associated with a production defining X. Since e*val* $p[i]$ will cause the execution of one of such actions, the attributes in R must be defined for $p[i]$. The attributes in D are defined for the head nonterminal by the execution of a semantic action associated with a production defining X, therefore after the execution of e*val* $p[i]$ the attributes in D will be defined for $p[i]$.

$$\frac{\Psi' \subseteq \Psi'_1 \quad \Psi_1 \subseteq \Psi \quad \Gamma; \Psi_1 \vdash_p t : T; \Psi'_1}{\Gamma; \Psi \vdash_p t : T; \Psi'} \text{ (T-Sub)}$$

$$\Gamma; \Psi \vdash_p \texttt{tr/fls} : \texttt{Bool}; \emptyset \text{ (T-Tr/Fls)} \qquad \Gamma; \Psi \vdash_p \texttt{unit} : \texttt{Unit}; \emptyset \text{ (T-Unit)} \qquad \Gamma; \Psi \vdash_p n : \texttt{Int}; \emptyset \text{ (T-Int)}$$

$$\frac{\Gamma; \Psi \vdash_p \overline{e} : \overline{T}; \emptyset \quad \texttt{typeOf}(op) = (\overline{T}, T)}{\Gamma; \Psi \vdash_p op(\overline{e}) : T; \emptyset} \text{ (T-Op)} \qquad \frac{\Gamma; \Psi \vdash_p s : \texttt{Unit}; \Psi_1 \quad \Gamma; \Psi \cup \Psi_1 \vdash_p s' : \texttt{Unit}; \Psi_2}{\Gamma; \Psi \vdash_p s; s' : \texttt{Unit}; \Psi_1 \cup \Psi_2} \text{ (T-Seq)}$$

$$\frac{\Gamma; \Psi \vdash_p e : \texttt{Bool}; \emptyset \quad \Gamma; \Psi \vdash_p s : \texttt{Unit}; \Psi' \quad \Gamma; \Psi \vdash_p s' : \texttt{Unit}; \Psi'}{\Gamma; \Psi \vdash_p \texttt{if } e \texttt{ then } s \texttt{ else } s' : \texttt{Unit}; \Psi'} \text{ (T-If)} \qquad \frac{(p[i], a) \in \Psi \quad 0 \leq i \leq |p|}{\Gamma; \Psi \vdash_p p[i].a : T_a; \emptyset} \text{ (T-GetAtt)}$$

$$\frac{\Gamma; \Psi \vdash_p e : T_a; \Psi \quad 0 \leq i \leq |p|}{\Gamma; \Psi \vdash_p p[i].a = e : \texttt{Unit}; \{(p[i], a)\}} \text{ (T-SetAtt)} \qquad \frac{NT(p[i]){:}(R, D) \in \Gamma \quad R \subseteq \Psi(p[i])}{\Gamma; \Psi \vdash_p \texttt{eval } p[i] : \texttt{Unit}; \{(p[i], a) \mid a \in D\}} \text{ (T-Eval)}$$

Figure A.2.: *Rules of the type system for terms.*

$$\frac{\Gamma; \{(p\,[0], a) \mid a \in R\} \vdash_p s : \texttt{Unit}; \Psi}{\Gamma \vdash_p s : (R, \Psi(p[0]))} \text{ (T-Act)}$$

$$\frac{\Gamma_R \cup \Gamma_D \vdash_{p_k} s_k : (R_k, D_k) \quad (1 \leq k \leq m) \quad dom(\Gamma_R) \cap dom(\Gamma_D) = \emptyset \quad \Gamma_D = \{X{:}(\cup_{p_k \in (P \restriction X)} R_k, \cap_{p_k \in (P \restriction X)} D_k) \mid X \in Def(P)\}}{\Gamma_R \vdash \mathcal{TS}_P = \{p_1 : (R_1, D_1).\{s_1\}., \ldots, p_m : (R_m, D_m).\{s_m\}.\} : \Gamma_D} \text{ (T-Component)}$$

$$\frac{(\Gamma_R \cup \Gamma_D) - Def(P) \vdash \mathcal{TS}_P : \Gamma_D \restriction Def(P) \quad (\Gamma_R \cup \Gamma_D) - Def(P') \vdash \mathcal{TS}_{P'} : \Gamma_D \restriction Def(P')}{\Gamma_R \vdash \mathcal{TS}_P \circ \mathcal{TS}_{P'} : \Gamma_D} \text{ (T-Comp)}$$

Figure A.3.: *Well typed semantic actions, components and component composition..*

Figure A.3 shows the typing for semantic actions and typed components. Rule (T-Act) says that an action has the correct decoration (R, D) in the nonterminal environment Γ if from Γ and the attribute context in which the nonterminal instance $p[0]$ has all the attributes in R, the execution of the action defines for $p[0]$ all the attributes defined for $p[0]$ in the final attribute context Ψ. In typing a component, rule (T-Component) we distinguish two disjoint sets of nonterminals, the *defined nonterminals*, $X \in Def(P)$ and the *required nonterminals*, $X \in NT(P) - Def(P)$. The slice $\mathcal{TS}_P$ has type Γ_D from Γ_R if the domain of Γ_R does not contain assumptions for nonterminals defined in P and all the actions in the slice have the correct decoration in the nonterminal environment Γ_R, Γ_D, where Γ_D associates nonterminals $X \in Def(P)$ with the set of attributes compatible with all the semantic actions associated with productions defining X in the slice. That is, it requires the union of the set of attributes required by any action and ensures the intersection of the set of attributes defined by an action. Rule (T-Comp) says that the composition of slices $\mathcal{TS}_P$ and $\mathcal{TS}_{P'}$ has type Γ_D from the nonterminal environment Γ_R if $\mathcal{TS}_P$ can be derived from the restriction of Γ_D to the nonterminals defined in P from the nonterminal environment Γ_R extended with the assumptions on the nonterminals in Γ_D which are not defined in P. Similarly for $\mathcal{TS}_{P'}$. This ensures that the assumptions

on nonterminals in typing the actions of $\mathcal{TS}_P$ and $\mathcal{TS}_{P'}$ are consistent. Requiring exactly the same assumptions is not a restriction, since we have subtyping on the typing of actions. We can show that, if P'' is the sequence of productions $P\,P'$, then $\Gamma_R \vdash \mathcal{TS}_P \circ \mathcal{TS}_{P'} : \Gamma_D$ if and only if $\Gamma_R \vdash \mathcal{TS}_{P''} : \Gamma_D$ where $\mathcal{TS}_{P''} = \mathcal{TS}_P \cup \mathcal{TS}_{P'}$. Note that, from definition of composition, Def. 7, the labels of productions in P and P' are disjoint.

Finally we say that the composition of slices, $\mathcal{TS}_1 \circ \cdots \circ \mathcal{TS}_n$, where $\mathcal{TS}_i$ $(1 \leq i \leq n)$ is the slice associated with the productions P_i, is a *well-typed language implementation* when $P_1 \ldots P_n$ is a grammar with p_r as start production and, for some Γ and D, we have that $\vdash \mathcal{TS}_1 \circ \cdots \circ \mathcal{TS}_n : \Gamma$ and $\Gamma(NT(p_r[0]))=(\varnothing, D)$.

Soundness. Consider a grammar $\mathcal{G} = P_1 \ldots P_n$ and a well-typed language implementation $\mathcal{TS}_1 \circ \cdots \circ \mathcal{TS}_n$. We know that the slice $\mathcal{TS}_\mathcal{G} = \cup_{1\leq i\leq n}\mathcal{TS}_i$ is also a well-typed language implementation. Let η be a syntax tree derived from the grammar $\mathcal{G}$, i.e., $\mathcal{G} \vDash \eta$. Soundness is stated by Theo. 1 where $s_{in} \mid \mu_{in}$ is the initial configuration as defined in Section A.2 for η in $\mathcal{TS}_\mathcal{G}$.

Theorem 1 (Soundness). *If $\eta \vDash s_{in} \mid \mu_{in} \rightarrow^* s \mid \mu$, then either $s = unit$ or $\eta \vDash s \mid \mu \rightarrow s' \mid \mu'$ for some s' and μ'.*

Moreover, we can prove that the syntax-tree is correctly decorated with attributes in accordance with the type system.

A.4. Type Inference

In this section we give an informal definition of the type inference function for slices, T_S, describing the constraints returned by this function, and showing how constraints are checked for consistency and combined. Then, we state the results of correctness and completeness of type inference w.r.t. the type system of Section A.3.

Type inference is defined by a partial function T_S from slices, $\mathcal{S}_P$, to the requirements on nonterminals that are used but not defined in the slice, $NT(P) - Def(P)$, and the properties of the nonterminals defined in P derived by the analysis of the associated semantic actions of the slice. The function T_S is defined in terms of a partial function T_a that does the analysis of the actions associated with the productions of the slice.

The *type inference function for slices* T_S if defined is such that $T_S(\mathcal{S}_P) = \gamma, \Gamma$ where:

- γ is a set containing the constraints on the nonterminals X, such that $X \in NT(P) - Def(P)$, derived by the actions of the slice. In particular, γ is a set of associations between nonterminals and triples, written $X{:}(A_1, A_2, A_\perp)$, whose first two components are sets of attributes and the third is either a set of attributes or $\perp$ meaning that the set is undefined. The attributes in A_1 and A_2 are requirements on slices in which these nonterminals are defined. Namely,
  - attributes in A_1 must be in the *required set* of the actions associated with productions defining X;
  - attributes in A_2 must be in the *defined set* of the actions associated with productions defining X.

  - attributes in $A_\perp$ are the attributes that are defined, in actions of $\mathcal{S}_P$, before evaluating an e*val* of an instance of the nonterminal X. If there is no e*val* of an instance of the nonterminal X then $A_\perp = \perp$
- Γ has the meaning of Γ_D in the type-system, i.e., associates the nonterminal $X \in Def(P)$ with their required and provided attributes derived from the analysis of the actions of $\mathcal{S}_P$.

T_S is defined in terms of the *type inference function for actions* T_a, which takes as input a statement s and the associated production $p : X \to X_1 \cdots X_q$ and, if defined, is such that $T_a(s, p) = \gamma, (R, D)$. The set γ has the same meaning as for T_S, i.e., the constraints on $NT(p) - \{X\}$. The sets R and D are the required and defined attributes for X derived from the action s.

We now show, through a simple example, how *type inference of slice composition* is performed. Let P contain the single production $p_X : X \to XY$ and let P' contain the single production $p_Y : Y \to XY$. Therefore $NT(P) = NT(P') = \{X, Y\}$, $Def(P) = \{X\}$ and $Def(P) = \{Y\}$. Consider the slices $\mathcal{S}_P$ and $\mathcal{S}_{P'}$ containing semantic actions for the corresponding productions. Assume that

- $T_S(\mathcal{S}_P) = \{Y{:}(A_1^Y, A_2^Y, A_\perp^Y)\}, \{X{:}(R^X, D^X)\}$ and
- $T_S(\mathcal{S}_{P'}) = \{X{:}(A_1^X, A_2^X, A_\perp^X)\}, \{Y{:}(R^Y, D^Y)\}$.

The constraints generated by the type inference about the two given slices must be *consistent* so that the two slices can be composable. That is, the requirements on the nonterminal Y made by its use in $\mathcal{S}_P$, $Y{:}(A_1^Y, A_2^Y, A_\perp^Y)$ and those provided by the the semantic action associated with the production p_Y in $\mathcal{S}_{P'}$, $Y{:}(R^Y, D^Y)$. These constraints are consistent if

- all the attributes required by an instance of the nonterminal Y in the action associated to X are defined by the action associated Y, i.e., $A_2^Y \subseteq D^Y$, and
- if there are `eval` of an instance of the nonterminal Y in the action associated to X, i.e. $A_\perp^Y \neq \perp$, then all the attributes required by Y by the action associated Y are defined before the eval in the action associated to X, i.e. $R^Y \subseteq A_\perp^Y$.

(This should hold also for X, i.e., the requirement made for X in $\mathcal{S}_{P'}$ must be satisfied by the semantic action associated with the production p_X in $\mathcal{S}_P$.)

If the constraints returned by $T_S(\mathcal{S}_P)$ and $T_S(\mathcal{S}_{P'})$ are consistent, then $T_S(\mathcal{S}_P \circ \mathcal{S}_{P'}) = \varnothing, \{X{:}(R^X \cup A_1^X, D^X), Y{:}(R^Y \cup A_1^Y, D^Y)\}$.

The type inference function for slices, in addition to returning the constraints on nonterminals produces also a typed version of the input slice, by attaching to the actions s of the slice the pairs (R, D) such that $T_a(s, p) = \gamma, (R, D)$.

Correctness of the inference is stated by the following theorem.

Theorem 2 (Correctness). *Let $T_S(\mathcal{S}_P) = \gamma, \Gamma$ and let $\mathcal{TS}_P$ be the typed version of the slice. Then $\{X{:}(A_1, A_2) \mid \exists A_\perp\ X{:}(A_1, A_2, A_\perp) \in \gamma\} \vdash \mathcal{TS}_P : \Gamma$*

To state completeness we have to relate typed slices with their underlying untyped version. Therefore, we introduce the *erasure* of a typed slice, $erase(\mathcal{TS}_P)$, which is the

slice obtained by erasing the decoration of actions in $\mathcal{TS}_P$. Note that, the typed slice $\mathcal{TS}_P$ returned by $T_S(\mathcal{S}_P)$ is such that $erase(\mathcal{TS}_P) = \mathcal{S}_P$.

Theorem 3 (Completeness). *Let $\mathcal{S}_P$ be a slice. If $T_S(\mathcal{S}_P)$ is not defined, then for no typed slice $\mathcal{TS}_P$ such that $erase(\mathcal{TS}_P) = \mathcal{S}_P$ there are Γ and Γ' such that $\Gamma' \vdash \mathcal{TS}_P : \Gamma$.*

Conclusions. The type and inference system described in this Appendix can be used to prevent erroneous modifications of the interpreter by enforcing constraints when interpreter elements, as per Definition 2, are altered. This would guarantee that removing, adding or replacing semantic actions will always produce an interpreter that provides all the necessary attributes that are needed by other components. Although it can guarantee that the execution will not get stuck, the type system, however, is unable to guarantee that the final behavior will correspond to the expected behavior which is the responsibility of the language developer

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.

[2] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. *Transaction on Aspect-Oriented Software Development*, 1(1):135–173, March 2006.

[3] Kenichi Asai. Compiling a Reflective Language Using MetaOCaML. In Matthew Flatt, editor, *Proceedings of the 13th International Conference on Generative Programming: Concepts and Experiences (GPCE'14)*, pages 113–122, Västerås, Sweden, September 2014. ACM.

[4] Kevin Backhouse. A Functional Semantics of Attribute Grammars. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, pages 142–157, Grenoble, France, April 2002.

[5] Mehdi Bagherzadeh, Hridesh Rajan, Gary T Leavens, and Sean Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 141–152. ACM, 2011.

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[7] Pablo Barceló. Querying Graph Patterns. In Wenfei Fan, editor, *Proceedings of the ACM Symposium on Principles of database Systems (PODS'13)*, pages 175–188, New York, NY, USA, June 2013. ACM.

[8] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. Approaches to Interpreter Composition. *Computer Languages, Systems & Structures*, 44(Part C):199–217, December 2015.

[9] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. PACKT Publishing Ltd, August 2013.

[10] Gavin Bierman, Matthew Parkinson, and James Noble. UpgradeJ: Incremental Typechecking for Class Upgrades. In Jan Vitek, editor, *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, Lecture Notes in Computer Science 5142, pages 235–259, Paphos, Cyprus, July 2008. Springer.

[11] Carl Friedrich Bolz. *Meta-Tracing Just-in-Time Compilation for RPython*. Phd thesis, Heinrich-Heine-Universität Düsseldorf, Düsseldorf, Germany, September 2013.

[12] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[13] Carl Friedrich Bolz and Laurence Tratt. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming*, 98(3):408–421, February 2015.

[14] Gilad Bracha and William Cook. Mixin-Based Inheritance. In Akinori Yonezawa, editor, *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/E-COOP'90)*, pages 303–311, Ottawa, Canada, October 1990. ACM.

[15] Martin Büchi and Wolfgang Weck. The greybox approach: When blackbox specifications hide too much. Technical report, Citeseer, 1999.

[16] Howard I. Cannon. Flavors: A non-hierarchical approach to object-oriented programming. In *Technical Report*. MIT Artificial Intelligence Laboratory Cambridge, Mass, 1980.

[17] Walter Cazzola. Remote Method Invocation as a First-Class Citizen. *Distributed Computing*, 16(4):287–306, December 2003.

[18] Walter Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Proceedings of the 11th International Conference on Software Composition (SC'12)*, Lecture Notes in Computer Science 7306, pages 162–177, Prague, Czech Republic, 31st of May-1st of June 2012. Springer.

[19] Walter Cazzola, Ruzanna Chitchyan, Awais Rashid, and Albert Shaqiri. μ-DSU: A Micro-Language Based Approach to Dynamic Software Updating. *Computer Languages, Systems & Structures*, 2017.

[20] Walter Cazzola, Antonio Cisternino, and Diego Colombo. Freely Annotating C#. *Journal of Object Technology*, 4(10):31–48, December 2005.

[21] Walter Cazzola, Paola Giannini, and Albert Shaqiri. Formal Attributes Traceability in Modular Language Development Frameworks. *Electronic Notes In Theoretical Computer Science*, 322:119–134, April 2016.

[22] Walter Cazzola and Diego Mathias Olivares. Gradually Learning Programming Supported by a Growable Programming Language. *IEEE Transactions on Emerging Topics in Computing*, 4(3):404–415, September 2016. Special Issue on Emerging Trends in Education.

[23] Walter Cazzola and Sonia Pini. On the Footprints of Join Points: The Blueprint Approach. *Journal of Object Technology*, 6(7):167–192, August 2007.

[24] Walter Cazzola, Sonia Pini, Ahmed Ghoneim, and Gunter Saake. Co-Evolving Application Code and Design Models by Exploiting Meta-Data. In *Proceedings of the 22nd Annual ACM Symposium on Applied Computing (SAC'07)*, pages 1275–1279, Seoul, South Korea, on 11th-15th of March 2007. ACM Press.

[25] Walter Cazzola and Stefano Salvotelli. Recognizing Join Points from their Context through Graph Grammars. In *Proceedings of the 13th Aspect-Oriented Modeling Workshop (AOM'09)*, pages 37–42, Charlottesville, Virginia, USA, on 2nd of March 2009. ACM.

[26] Walter Cazzola and Albert Shaqiri. Dynamic Software Evolution through Interpreter Adaptation. In *Proceedings of the 15th International Conference on Modularity (Modularity'16)*, pages 16–19, Málaga, Spain, 14th-17th of March 2016. ACM.

[27] Walter Cazzola and Albert Shaqiri. Modularity and Optimization in Synergy. In Don Batory, editor, *Proceedings of the 15th International Conference on Modularity (Modularity'16)*, pages 70–81, Málaga, Spain, 14th-17th of March 2016. ACM.

[28] Walter Cazzola and Albert Shaqiri. Context-Aware Software Variability through Adaptable Interpreters. *IEEE Software*, 2017. Special Issue on Context Variability Modeling.

[29] Walter Cazzola and Albert Shaqiri. Open Programming Language Interpreters. *The Art, Science, and Engineering of Programming Journal*, 1(2):5–1–5–34, April 2017.

[30] Walter Cazzola and Edoardo Vacchi. Neverlang 2: Componentised Language Development for the JVM. In Walter Binder, Eric Bodden, and Welf Löwe, editors, *Proceedings of the 12th International Conference on Software Composition (SC'13)*, Lecture Notes in Computer Science 8088, pages 17–32, Budapest, Hungary, 19th of June 2013. Springer.

[31] Walter Cazzola and Edoardo Vacchi. @Java: Bringing a Richer Annotation Model to Java. *Computer Languages, Systems & Structures*, 40(1):2–18, April 2014.

[32] Guido Chari, Diego Garbervetsky, and Stefan Marr. Building Efficient and Highly Run-Time Adaptable Virtual Machines. In Roberto Ierusalimschy, editor, *Proceedings of the 12th Symposium on Dynamic Languages (DLS'16)*, pages 60–71, Amsterdam, Netherlands, November 2016.

[33] Thomas E Cheatham Jr. Motivation for extensible languages. *ACM SIGPLAN Notices*, 4(8):45–49, 1969.

[34] Shigeru Chiba. A Meta-Object Protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, volume 30 of *Sigplan Notices*, pages 285–299, Austin, Texas, USA, October 1995. ACM.

[35] Shigeru Chiba. Load-Time Structural Reflection in Java. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, LNCS 1850, pages 313–336, Cannes, France, June 2000. Springer-Verlag.

[36] Ruzanna Chitchyan, Walter Cazzola, and Awais Rashid. Engineering Sustainability through Language. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, pages 501–504, Firenze, Italy, 16th-24th of May 2015. IEEE. Track on Software Engineering in Society.

[37] Thomas Cleenewerck. *Modularizing Language Constructs: A Reflective Approach*. PhD thesis, Vrije Universiteit Brussel, Brussel, Belgium, July 2007.

[38] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A Graphical Query Language Supporting Recursion. In Umeshwar Dayal, editor, *Proceedings of the 13th International Conference on Management of Data (SIGMOD'87)*, pages 323–330, San Francisco, CA, USA, May 1987. ACM.

[39] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, June 2000.

[40] Linda De Michiel and Richard P. Gabriel. The Common Listp Object System: An Overview. In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *Proceedings of the 1st European Conference on Object-Oriented Programming (ECOOP'87)*, Lecture Notes in Computer Science 276, pages 151–170, Paris, France, June 1987. Springer.

[41] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: a Meta-Language for Modular and Reusable Development of DSLs. In Davide Di Ruscio and Markus Völter, editors, *Proceedings of the 8th International Conference on Software Language Engineering (SLE'15)*, pages 25–36, Pittsburgh, PA, USA, October 2015. ACM.

[42] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuprolog: A light-weight prolog for internet applications and infrastructures. In *International Symposium on Practical Aspects of Declarative Languages*, pages 184–198. Springer, 2001.

[43] Theo D'Hondt, Kris De Volder, Kim Mens, and Roel Wuyts. Co-Evolution of Object-Oriented Software Design and Implementation. In Mehmet Akşit, editor, *Proceedings of the International Symposium on Software Architectures and Component Technology*, pages 207–224, Twente, The Netherlands, January 2000. Kluwer.

[44] Mikhail Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In Vinny Cahill, Siobhán Clarke, Simon Dobson, and Robert Filman, editors, *Proceedings of the 1st Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE'01)*, pages 14–18, Tampa Bay, FL, USA, October 2001.

[45] Peter Ebraert, Yves Vandewoude, Theo D'Hont, and Yolande Berbers. Pitfalls in Unanticipated Dynamic Software Evolution. In Walter Cazzola, Shigeru Chiba, Gunter Saake, and Tom Tourwé, editors, *Proceedings of ECOOP'2005 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05)*, pages 3–8, Glasgow, Scotland, July 2005.

[46] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, USA, October 2000.

[47] Matthew Flatt. Creating Languages in Racket. *ACM Queue*, 9(11):1–15, November 2011.

[48] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In David B. MacQueen and Luca Cardelli, editors, *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL'98)*, pages 171–183. ACM, January 1998.

[49] Martin Fowler and Rebecca Parsons. *Domain Specific Languages*. Addison Wesley, September 2010.

[50] Richard P. Gabriel. The Structure of a Programming Language Revolution. In Jonathan Edwards, editor, *Proceedings of the ACM international Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)*, pages 195–214, Tucson, AZ, USA, October 2012.

[51] Andreas Gal, Christian W Probst, and Michael Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153. ACM, 2006.

[52] Letterio Galletta. *Adaptivity: Linguistic Mechanisms and Static Analysis Techniques*. PhD Thesis, Università degli Studi di Pisa, Pisa, Italy, May 2014.

[53] Benoit Gennart and Roger D. Hersch. Computer-Aided Synthesis of Parallel Image Processing Applications. In *Proceedings of the Conference on Parallel and Distributed Methods for Image Processing*, pages 48–61, Denver, USA, 1999.

[54] Michael Golm. *Design and implementation of a meta architecture for Java*. PhD thesis, Master's thesis, Friedrich-Alexander-University, Erlangen-Nurenburg, 1997.

[55] Sebastián González, Kim Mens, Marius Colăcioiu, and Walter Cazzola. Context Traits: Dynamic Behaviour Adaptation through Run-Time Trait Recomposition. In Jörg Kienzle, editor, *Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD'13)*, pages 209–220, Fukuoka, Japan, 24th-29th of March 2013. ACM.

[56] Phil Greenwood and Lynne Blair. A Framework for Policy Driven Auto-adaptive Systems Using Dynamic Framed Aspects. *Transactions on Aspect-Oriented Software Development*, 2:30–65, 2006.

[57] Kris Gybels. Soul and smalltalk-just married. In *Draft Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, page 1. Citeseer, 2003.

[58] Görel Hedin. An Introductory Tutorial on JastAdd Attribute Grammars. In João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, LNCS 6491, pages 166–200. Springer, 2011.

[59] Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt. Reflection for the Masses. In Robert Hirshfeld and Kim Rose, editors, *Self-Sustaining Systems*, LNCS 5146, chapter 6, pages 87–122. Springer, 2008.

[60] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-Oriented Programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[61] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern Matching in Trees. *Journal of ACM*, 29(1):68–95, 1982.

[62] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In Pierre America, editor, *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP'91)*, LNCS 512, pages 21–38, Geneve, Switzerland, July 1991. Springer.

[63] Gang Huang, Hong Mei, and Fu-Qing Yang. Runtime Software Architecture Based on Reflective Middleware. *Journal of Information Science*, 47(5):555–576, October 2004.

[64] Stephen C. Johnson. YACC: Yet Another Compiler-Compiler. Technical Report CS-TR-32, Bell Laboratories, Hill, NJ, USA, July 1975.

[65] Jevgeni Kabanov and Varmo Vene. A Thousand Years of Productivity: The JRebel Story. *Software: Practice and Experience*, 44(1):105–127, January 2014.

[66] Ted Kaminski and Eric Van Wyk. Modular Well-Definedness Analysis for Attribute Grammars. In Krzysztof Czarnecki and Görel Hedin, editors, *Proceedings of the 5th International Conference on Software Language Engineering (SLE'13)*, Lecture Notes in Computer Science 7745, pages 352–371, Dresden, Germany, September 2013. Springer.

[67] Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In Martin Rinard, Kevin J. Sullivan, and Daniel H. Steinberg, editors, *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 444–463, Reno, Nevada, USA, October 2010. ACM.

[68] Roger Keays and Andry Rakotonirainy. Context-Oriented Programming. In Sujata Banerjee and Mitch Cherniack, editors, *Proceedings of the 3rd ACM International*

Workshop on Data Engineering for Wireless and Mobile Access (MobiDe'03), pages 9–16, San Diego, CA, USA, September 2003. ACM.

[69] John Keeney and Vinny Cahill. Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework. In *Proceedings of the 4th International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 3–14, Como, Italy, June 2003. IEEE.

[70] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*, Nantes, France, July 2006. Springer.

[71] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[72] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeff Palm, and Bill Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, LNCS 2072, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[73] Gregor Kiczales, John Lamping, Christina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th international conference on Software engineering*, pages 481–490. ACM, 1997.

[74] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science 1241, pages 220–242, Helsinki, Finland, June 1997. Springer-Verlag.

[75] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[76] Ján Kollár and Michal Forgáč. Combined Approach to Program and Language Evolution. *Computing and Informatics*, 29(6):1103–1116, 2010.

[77] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In Lionel C. Briand and Alexander L. Wolf, editors, *Proceedings of 29th International Conference on Software Engineering (ICSE'07): Future of Software Engineering (FoSE'07)*, pages 259–268, Minneapolis, MN, USA, May 2007. IEEE Computer Society.

[78] Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares. Choosy and Picky: Configuration of Language Product Lines. In Goetz Botterweck and Jules White, editors, *Proceedings of the 19th International Software Product Line*

Conference (SPLC'15), pages 71–80, Nashville, TN, USA, 20th-24th of July 2015. ACM.

[79] Guy Lapalme and Suzanne Chapleau. Logicon: an integration of prolog into icon. *Software: Practice and Experience*, 16(10):925–944, 1986.

[80] John R Levine, Tony Mason, and Doug Brown. *Lex & yacc*. " O'Reilly Media, Inc.", 1992.

[81] Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.

[82] Stefan Marr, Chris Seaton, and Stéphane Ducasse. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises. In Steve Blackburn, editor, *Proceedings of the 36th Conference on Programming Language Design and Implementation (PLDI'15)*, Portland, OR, USA, June 2015.

[83] Hidehiko Masuhara, Satoshi Matsuoka, Kenichi Asai, and Akinori Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *ACM Sigplan Notices*, volume 30, pages 300–315. ACM, 1995.

[84] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving Code and Design Using Intensional Views - A Case Study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, July/October 2006.

[85] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in Software Evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 13–22, Lisbon, Portugal, September 2005. IEEE Press.

[86] Marjan Mernik. An Object-Oriented Approach to Language Compositions for Software Language Engineering. *Journal of Systems and Software*, 86(9):2451–2464, September 2013.

[87] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[88] David A. Moon. Object-Oriented Programming with Flavors. In Daniel Ingalls, editor, *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, pages 1–8, Portland, OR, USA, September/October 1986. ACM.

[89] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, April 2001.

[90] Oscar Nierstrasz and Tudor Gîrba. Lessons in Software Evolution Learned by Listening to Smalltalk. In Jan Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'10)*, LNCS 5901, pages 77–95, Špindlerův Mlýn, Czech Republic, January 2010. Springer.

[91] Alexandre Oliva, Islene Calciolari Garcia, and Luiz Eduardo Buzato. The reflexive architecture of guaran a. Technical report, Citeseer, 1998.

[92] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 177–186, Kyoto, Japan, April 1998. IEEE Computer Society.

[93] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A Technique for Dynamic Updating of Java Software. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 649–658, Montréal, Canada, October 2002. IEEE Press.

[94] David L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[95] David Lorge Parnas. Information distribution aspects of design methodology. 1971.

[96] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, May 2007.

[97] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.

[98] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for Java on a Stock JVM. In Todd Millstein, editor, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'14)*, pages 103–119, Portland, OR, USA, October 2014. ACM.

[99] Mario Pukall, Alexander Grebhahn, Reimar Schröter, Christian Kästner, Walter Cazzola, and Sebastian Götz. JavAdaptor: Unrestricted Dynamic Software Updates for Java. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 989–991, Waikiki, Honolulu, Hawaii, on 21st-28th of May 2011. IEEE.

[100] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schöter, and Gunter Saake. JavAdaptor — Flexible Runtime Updates of Java Applications. *Software—Practice and Experience*, 43(2):153–185, February 2013.

[101] Hridesh Rajan and Leavens Gary T. Ptolemy: A Language with Quantified, Typed Events. In Jan Vitek, editor, *Proceedings of the Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, LNCS 5142, pages 155–179, Paphos, Cyprus, July 2008. Springer.

[102] Ramana Rao. Implementational Reflection in Silica. In Pierre America, editor, *Proceedings of ECOOP'91*, pages 251–266, Geneva, Switzerland, July 1991. Springer-Verlag.

[103] Barry Redmond and Vinny Cahill. Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java. In Walter Cazzola, Shigeru Chiba, and Thomas Ledoux, editors, *Proceedings of ECOOP Workshop on Reflection and Metalevel Architectures (RMA'oo)*, June 2000.

[104] John Alan Robinson and EE Silbert. *LOGLISP: an alternative to PROLOG*. School of Computer and Information Science, Syracuse University, 1980.

[105] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated Partial Behavioral Reflection: Adapting Applications at Run-Time. *Computer Languages, Systems & Structures*, 34(2/3):46–65, 2008.

[106] Erik Ruf. Partial evaluation in reflective system implementations. In *Workshop on Reflection and Metalevel Architecture*, 1993.

[107] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, May 2009.

[108] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-Oriented Programming: A Software Engineering Perspective. *Journal of Systems and Software*, 85(8):1801–1817, August 2012.

[109] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. An Analysis of Language-Level Support for Self-Adaptive Software. *ACM Transactions on Autonomous and Adaptive Systems*, 8(2):7:1–7:29, July 2013.

[110] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behaviour. In Luca Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, Lecture Notes in Computer Science 2743, pages 248–274, Darmstadt, Germany, July 2003. Springer.

[111] Matthias Springer, Fabio Niephaus, Robert Hirschfeld, and Hidehiko Masuhara. Matriona: Class Nesting with Parametrization in Squeak/Smalltalk. In Don Batory, editor, *Proceedings of the 15th International Conference on Modularity (Modularity'16)*, pages 118–129, Málaga, Spain, March 2016. ACM.

[112] Gregory T Sullivan. Dynamic partial evaluation. In *Programs as Data Objects*, pages 238–256. Springer, 2001.

[113] Kevin Sullivan, William G Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. Modular aspect-oriented design with xpis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(2):5, 2010.

[114] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information Hiding Interfaces for Aspect-Oriented Design. In *Proceedings of the 10th European Software Engineering Conference (ESEC'13)*, pages 166–175, Lisbon, Portugal, September 2005. ACM.

[115] Éric Tanter. Reflection and Open Implementation. Technical Report TR-DCC-20091123-013, DCC, University of Chile, November 2009.

[116] Éric Tanter, Noury MN Bouraqadi-Saâdani, and Jacques Noyé. Reflex—towards an open reflective extension of java. In *International Conference on Metalevel Architectures and Reflection*, pages 25–43. Springer, 2001.

[117] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In Guy L Steele, Jr, editor, *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'03)*, pages 27–46, Anaheim, CA, USA, October 2003. ACM.

[118] Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, and José Piquer. Altering Java Semantics via Bytecode Manipulation. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Proceedings of Generative Programming and Component Engineering (GPCE'02)*, LNCS 2487, pages 283–298, Pittsburgh, PA, USA, October 2002. Springer.

[119] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. OpenJava: A Class-based Macro System for Java. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 119–135. Springer-Verlag, Heidelberg, Germany, June 2000.

[120] Warren Teitelman. *PILOT: A Step Toward Man-Computer Symbiosis*. Phd thesis, Massachusetts Institute of Technology, September 1966.

[121] Edoardo Vacchi and Walter Cazzola. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures*, 43(3):1–40, October 2015.

[122] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[123] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming*, 75(1-2):39–54, January 2010.

[124] Harald H. Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. Higher Order Attribute Grammars. In Richard L. Wexelblat, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'89)*, pages 131–145, Portland, OR, USA, June 1989.

[125] Markus Völter and Vaclav Pech. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 1449–1450, Zürich, Switzerland, June 2012. IEEE.

[126] Daniel Weinreb and Moon David A. Flavors: Message Passing in the Lisp Machine. Technical Report 602, Massachusetts Institute of Technology, November 1980.

[127] Ian Welch and Robert J Stroud. Kava-a reflective java based on bytecode rewriting. In *Workshop on Reflection and Software Engineering*, pages 155–167. Springer, 1999.

[128] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In Alessandro Warth, editor, *Proceedings of the 8th Symposium on Dynamic languages (DSL'12)*, pages 73–82, Tucson, AZ, USA, October 2012. ACM.

[129] Zhenxiao Yang, Betty H. C. Cheng, R. E. Kurt Stirewalt, J Sowell, Seyed Masoud Sadjadi, and Philip K. McKinley. An Aspect-Oriented Approach to Dynamic Adaptation. In David Garlan, Jeff Kramer, and Alexander L. Wolf, editors, *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02)*, pages 85–92, Charleston, SC, USA, November 2002. ACM.

[130] Guangquan Zhang and Mei Rong. A Framework for Dynamic Evolution Based on Reflective Aspect-Oriented Software Architecture. In *Proceedings of the 4th International Conference on Computer Sciences and Convergence Information Technology (ICCIT'09)*, pages 7–10, Seoul, South Korea, November 2009.