Formal Attributes Traceability in Modular Language Development Frameworks

Walter Cazzola[a,1] Paola Giannini[b,2] Albert Shaqiri[a,1]

[a] *Università degli Studi di Milano, Dipartimento di Informatica*

[b] *Università del Piemonte Orientale, Computer Science Institute, Disit*

Abstract

Modularization and component reuse are concepts that can speed up the design and implementation of domain specific languages. Several modular development frameworks have been developed that rely on attributes to share information among components. Unfortunately, modularization also fosters development in isolation and attributes could be undefined or used inconsistently due to a lack of coordination. This work presents 1) a type system that permits to trace attributes and statically validate the composition against attributes lack or misuse and 2) a correct and complete type inference algorithm for this type system. The type system and inference are based on the Neverlang development framework but it is also discussed how it can be used with different frameworks.

Keywords: modularity and composition, modular language implementation, formal validation of the composition, type inference

1 Introduction

Domain specific languages (DSLs) are getting more and more relevant nowadays but their development is still difficult and this contains their proper spread. One way to ease DSL development, consists in maximizing reuse by modularizing the language and its implementation in the composition of loosely coupled *language component*s where a language component is any language-oriented concept that should be part of the language shipped together with its implementation. According to this trend several modular development frameworks have been developed such as Lisa [9], JastAdd [5], Silver [11], Spoofax [7] and Neverlang [10].

The implementation of each language component provides a syntactic description of the language concept itself and the code necessary to support the expected semantics for such a concept. Composition is typically driven by the syntactic description of the language component that provides an interface to the other language components. Even if loosely coupled the code realizing the semantic of the

⋆ Partly funded by "Progetto MIUR PRIN CINA Prot. 2010LHT4KM".

[1] Email: {cazzola,shaqiri}@di.unimi.it

[2] Email: giannini@di.unipmn.it

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

different language components relies on data computed by the other components and the sharing of these data is typically delegated to and relies on the presence of attributes [8]. Being the composition syntax-driven, semantic constraints as attributes presence are rarely considered at development/composition time.

Modular development eases the reuse of language components fostering their separate development. Language components are developed in isolation and reused as black boxes relying on naming conventions or similar expedients. This may become unreliable when the language components can be separately compiled and dynamically composed, as in Neverlang, since they rely on information that is not part of component's interface. Without some static check, the composition may turn out in a real mess.

Apart from Spoofax [7] that provides a language transformation engine all the other approaches exploit variants of the attribute grammars [8] and syntax direct translation [1]. A typical (dangerous) situation consists of a language component's implementation that relies on an attribute that should be provided by the implementation of another component and the attribute is not defined, is defined with a different name or is inconsistently used. In such a situation, even if the composition could be done and the compiler generated, it will fail when it is used. Also when the attributes are declared as in Lisa [9] or precomputed as in Silver [11] there is still no assurance that they are consistently used.

In this work, we formalize the problem of attributes traceability over separate language component implementations by providing a type system that statically validates the composition with respect to the attributes. The validation is tailored on the Neverlang framework but it can be easily adapted to other modular language development frameworks as Lisa and Silver.

The paper is organized as follows. Section 2 introduces the problem of well-definedness in attribute grammars, how this is contextualized to the framework Neverlang and an overview of the proposed solution. Section 3 introduces the formalization of Neverlang slice that is used in Section 4 to define a small-step operational semantics that specifies how the semantic actions define, access and modify the attributes of syntax-trees. Section 5 introduces a type system for a type decorated version of slices which prevent runtime errors and states the soundness result. Section 6 outlines the type inference algorithm and states that it is correct and complete for the type system. In Sect. 7 some related work and the application of the presented system to them is discussed. In Sect. 8 we draw some conclusions.

2 Overview

Well-definedness in attribute grammars. The problem of ensuring that a grammar is well-defined has been addressed since the genesis of attribute grammars. In the context of pure attribute grammars, the *well-definedness* is traditionally expressed in terms of *closure* and *non-circularity* properties. The closure property states that for each attribute there is a semantic rule calculating its value. Pure attribute grammars always associates an attribute value with a function without side-effects; by definition of function this always grant the attribute definition. In this context the check for the closure property only consists of verifying the definition of the functions. Unfortunately, many tools do not adopt full-fledged attribute

grammars and although a rule exists it might not set the attribute. Among such tools we find Yacc, ANTLR, Silver [11], Neverlang [10] and Lisa [9]. The non-circularity property states that an attribute value must not depend on the attribute itself. In a monolithic setting, Knuth [8] presented an algorithm for testing a grammar for both closure and non-circularity. Vogt *et al.* [12] extended Knuth's algorithm in the context of higher-order attribute grammars. Backhouse [2] presented a definedness test that embodies both closure and circularity checks.

Although not trivial, the well-definedness analysis in a monolithic approach is simple when compared to the same analysis carried on a modular model, like ours, where information hiding introduces interesting challenges. In such a setting, Kaminski *et al.* [6] 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.

Neverlang. Neverlang is a framework for compositional language development. A complete exposition of Neverlang's features can be found in [3,4,10]. In the following we only describe those features necessary in this work.

The basic unit in Neverlang is the *slice* that implements a *language feature* [10]. A slice defines its own syntax through some grammar rules. It also defines semantic roles, i.e., a set of semantic actions associated with the grammar rules. Roles represent single tree traversals and the semantic actions are executed during the traversal. Semantic actions are written in a Java-like DSL that permits to define and work with attributes. Roles can be defined in modules collected and/or reused by the slices.

```
module IfThenElse {
  reference syntax {
    IF: Exp <-- "if" Exp "then" Exp "else" Exp;
  }
  role ( evaluation ) {
    IF: .{
      eval $IF[1];
      if (toBool($IF[1].val)) {
        eval $IF[2]; $IF.val = $IF[2].val;
      } else {
        eval $IF[3]; $IF.val = $IF[3].val; }
    }.
  }
}
```

Listing 1: if-then-else implementation.

```
module Numbers {
  reference syntax {
    INT: Exp <-- /\d+/
    DBL: Exp <-- /\d+\.\d+/
  }
  role ( evaluation ) {
    INT: .{
      $INT.val = new Integer(#0.text);
    }.
    DBL: .{
      $DBL.value = new Double(#0.text);
    }.
  }
}
```

Listing 2: Numbers implementation.

Listing 1 shows a module implementing a functional version of the if-then-else construct. The **`reference syntax`** defines the if expression syntax through a single production. Nonterminals are capitalized by convention, while terminals are enclosed in quotes. Each production may be preceded by a label (e.g., `IF`) that can be used in semantic actions to refer to the production nonterminals. Each role is associated with a name, in our example `evaluation`. Semantic actions are written enclosed between `.{` and `}.` symbols. An action is associated with its production by a label that precedes its definition. The example in Listing 1 has a single semantic action associated with the only production in the grammar.

Given a set of modules, Neverlang builds a tree-based interpreter/compiler for the implemented language. For a given input program the interpreter will construct its syntax tree and will traverse it for each role in the order specified in a separate

configuration file. Whenever a node n is visited, the run-time system will execute the semantic action associated with the production used to build a subtree rooted at n. In our example, whenever a node representing the head nonterminal `Exp` is visited the action labeled `IF` will be executed. During traversals a tree can be decorated with arbitrary attributes dynamically attached to tree nodes.

The semantic action code can refer to all nonterminals of the associated production by using the following syntax `$label[offset]`, where `label` is the label identifying the associated production. The head nonterminal has offset 0. So in the above example, the head nonterminal can be referred by `$IF[0]` or simply `$IF`. The second nonterminal representing the condition part of the if-then-else expression can be referred to by `$IF[1]` and so on.

The semantic action, first, traverses the subtree representing the condition part of the if-then-else construct (**`eval`** `$IF[1]`). This traversal may define new attributes in the node represented by `$IF[1]`. These attributes may come from other modules implementing the `Exp` nonterminal. Depending on the value of the `$IF[1].val` attribute, we continue by traversing one of the branches of the if-then-else construct (**`eval`** `$IF[2]` or **`eval`** `$IF[3]`). In both branches, we copy the `val` attribute of the branch to the current node.

Problem Statement. Since the nature of attributes in Neverlang is dynamic, an attribute might not be defined when needed. There are basically two reasons for this. First, an attribute might get defined only when a specific computational path is followed. For example, let us suppose that the else branch in Listing 1 misses the `$IF.val = $IF[3].val;` statement. In that case, at the end of the semantic action execution the attribute `$IF.val` might not be defined. On the other hand, attributes with the same goal could be labeled with different name in different modules/semantic actions due to different naming conventions or simply by distraction. Let us consider Listing 2 where both integer and floating point numbers are defined. The two rules define regular expressions to match integers and doubles respectively. 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.

Solution Overview. To identify erroneous situations like these a formalization of Neverlang has been provided that describes all the relevant entities involved in the framework and their formal semantics. To prevent errors, we introduce a version of Neverlang, in which semantic actions and nonterminals are decorated 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 computations 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 1 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 if 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 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 does not defines 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 Listing 2 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 1.

The type decoration needed for type-checking can be inferred. In Section 6, 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.

3 Formalizing the Syntax of Neverlang

In this section we introduce the formalization of the Neverlang syntax. Some restrictions over the Neverlang framework have been considered. *Slice*s are the considered modularity unit. Slices provide only one role. These simplifications do not represent a real limitation. In Neverlang semantic actions are specified by adding to full Java the "domain specific" constructs for accessing, defining and updating attributes and for forcing the visit of subtrees of nodes of the syntax-tree. In order to focus on attribute manipulation, we choose to limit the non "domain specific" constructs to a simple expression language.

A slice is composed of a portion of a grammar, i.e., a set of productions, and a role definition. Now we formally describe the slice syntax. We refer to the language we are defining as *target language*.

Productions and grammars. Typically, 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 our purposes, terminal symbols are useless, so they will not be included in the slice 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 3.1 Let p be the label for $X_0 \to X_1 \cdots X_q$,

(i) $p[i]$ 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
(ii) $|p| = q$
(iii) $NT(p, i) = X_i$, for $i = 0, \ldots, q$ and
(iv) $NT(p) = \cup_{0 \le i \le q}\{NT(p, i)\}$.

Definition 3.2 Given a sequence of productions $P = p_1 \ldots p_m$,

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

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

Definition 3.3 [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 $\mathcal{G}$.

Slices and language for semantic actions. As mentioned at the end of Sect. 2 slices are formalized with a single role and an action per production. Actions are statements of the language defined by the following grammar.

$$s ::= \texttt{unit} \mid \texttt{if}\ e\ \texttt{then}\ s\ \texttt{else}\ s \mid s; s \mid p[i].a = e \mid \texttt{eval}\ p[i]$$

$$e ::= v \mid p[i].a \mid op(e_1, \ldots, e_n)$$

$$v ::= \texttt{tr} \mid \texttt{fls} \mid n \qquad\qquad v_e ::= \texttt{unit} \mid v$$

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, `eval` $p[i]$ 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 3.4 [Slice and Slice Compositon]

- Given a sequence of productions P, a *slice* $\mathcal{S}_P$ on P is a set of *actions* labelled 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 slices* $\mathcal{S}_P$ *and* $\mathcal{S}_{P'}$, $\mathcal{S}_P \circ \mathcal{S}_{P'}$, denotes the slice, $\mathcal{S}_P \cup \mathcal{S}_{P'}$ on $P\,P'$.

In the definition of slice, the assumption that all productions in P are associated with an action is not a limitation, as we can always associate to a production the

null statement `unit`. Composition of slices is associative and commutative and, since in a slice productions and corresponding semantic actions could be relabeled, we can always define the composition of two slices.

4 Operational Semantics

The semantics for the semantic actions is specified in a "small step" style, by describing how the execution of its statements defines/modifies the attributes associated with a syntax-tree for a string in the target language. In the formalization for the operational semantics, we refer to a single slice on a sequence of productions specifying a full grammar, $\mathcal{G}$, as defined in Def. 3.3 (it could be a sublanguage of the target language). Even though a language implementation may be defined by the composition of some slices, by Def. 3.4, we know that this is equivalent to a single slice containing the union of the semantic actions of the composing slices. During the evaluation of semantic actions, the structure of the syntax-tree is fixed, whereas the attributes associated with its nodes vary. Attributes are separated from the syntax-tree.

Syntax-tree and Attributes. The definition of *syntax-tree* formalizes the data structure resulting from the parsing of a string in the target language. 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. Unique identifiers are associated with subtrees of syntax-trees.

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

Definition 4.1 [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 $\forall i\ 1 \leq i \leq q\ \exists p' : X_i \to \cdots \in \mathcal{G}$ such that η_i is a syntax-tree for p'.
- η *is a syntax-tree for a string of* $\mathcal{G}$, written $\mathcal{G} \models \eta$, 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, $dom(\eta) = \{id \mid \eta(id) \text{ is defined}\}$,

Example 4.2 Let us consider the productions `IF`, `INT` and `DBL` in Listings 1, 2 plus the start production `S: S <- Exp;` (`S` is the start symbol) which is added to transform the productions into a grammar according to Def. 3.3. Define the slice containing the actions specified in Listings 1, 2 plus the action:

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

The syntax-tree η for the input string `"if 1 then 2 else 3"` will be:

$$id_1 : (\texttt{S}, id_2{:}(\texttt{IF}, id_3{:}(\texttt{INT}, \epsilon)\ id_4{:}(\texttt{INT}, \epsilon)\ id_5{:}(\texttt{INT}, \epsilon)))$$

So $dom(\eta) = \{id_i \mid 1 \leq i \leq 5\}$, $\eta(id_1) = \eta$ and $\eta(id_3) = id_3{:}(\texttt{INT}, \epsilon)$.

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 4.3 [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 4.2. The attribute store $[id_1 \mapsto [\texttt{val} \mapsto 2], id_2 \mapsto [\texttt{val} \mapsto 2], id_3 \mapsto [\texttt{val} \mapsto 1], id_4 \mapsto [\texttt{val} \mapsto 2], id_5 \mapsto [\,]]$ says that for the node associated with the condition of the construct (id_3) the attribute `val` is defined and has 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 `val` is defined and has value 2. For the node id_5, associated with the else condition, no attribute is defined. 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 not defined since `toBool(1)` is true and so the node id_5 is not 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 giving the attributes currently defined for η (and their value) and the term t (either 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:

$$\eta \models t \,|\, \mu \rightarrow t' \,|\, \mu'$$

We put the syntax-tree η on the left of $\models$ since it does not change 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 by using 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 rewriting the language for semantic actions by substituting: $p[i].a$ with $id.a$, $p[i].a = e$ with $id.a = e$, and `eval` $p[i]$ with `eval` id.

Rules of the Operational Semantics. The rules of the operational semantics, given in Fig. 1, specify how the execution of a construct of the language 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 `eval` id with the action associated with the production, p, generating the id node. In the action, instances of nonterminals $p[i]$ are substituted

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

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

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

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

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

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

Fig. 1: Rules of operational semantics.

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 $s_{\texttt{in}} \,|\, \mu_{\texttt{in}}$ where:

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

For instance, for the Ex. 4.2, let $\mu_{\texttt{in}} = [id_j \mapsto [\,]]_{1\le j\le 5}$ the initial configuration is:

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

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

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

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

5 Type system

This section introduces a type system that traces the attributes definition and prevents the access to undefined attributes of the syntax tree nodes. In order to do this, we assume that the actions are decorated with both the set of attributes required by the nonterminal on the left-side of the production for its correct execution and the set of attributes that will be defined for the same nonterminal after the execution of the action.

Let $\mathcal{A} = \{\overline{a}\}$ be a set of attributes with a fixed type. This restriction permits to focus on the "definedness" of attributes rather than on their effective type, which is an orthogonal problem with a wide number of solutions. T_a denotes the type of the attribute a.

Definition 5.1 A *typed slice*, $\mathcal{TS}_P$, 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.

For the type checking of semantic actions, the definedness of attributes is traced through attribute contexts.

Definition 5.2 [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, 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.

Information about attributes required/defined by the execution of semantic actions associated to production for the nonterminals in P are collected in 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 the nonterminals X_i are distinct. If $X{:}(R, D) \in \Gamma$, then the execution of any semantic action associated with a production defining the nonterminal X relies on the definedness for the node associated with X of some of the attributes in R. On the other side, the execution of any of these semantic actions will assure that at the end at least all 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 $\Gamma \restriction M = \{X{:}(R, D) \mid X{:}(R, D) \in \Gamma \wedge X \in M\}$.

The considered primitive types are:

$$T = \texttt{Unit} \mid \texttt{Int} \mid \texttt{Bool}$$

`Unit` is the type of statements whereas `Int` and `Bool` are the types for expressions.

The type judgment for terms t, representing 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 according with 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 $\Psi' = \emptyset$, since their evaluation may not define attributes.

The type rules for the judgment $\Gamma; \Psi \vdash_p t : T; \Psi'$ are given in Fig. 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 side, 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

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

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

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

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

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

Fig. 2: Rules of the type system for terms.

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

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

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

Fig. 3: Well typed semantic actions, slices and slice coposition.

and s_2 must be statements their type must be `Unit`. For a conditional statement, rule (T-If), the condition is a boolean expression, both branches are statements, so they must have type `Unit` 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 `Unit` and they define the attribute a of $p[i]$. Finally, to check `eval` $p[i]$ 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 `eval` $p[i]$ 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 `eval` $p[i]$ the attributes in D will be defined for $p[i]$.

Figure 3 shows the typing for semantic actions and typed slices. 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 slice, rule (T-Slice) 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. 3.4, 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 $\Gamma(NT(p_r[0]))=(\emptyset, D)$.

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., $\mathcal{G} \models \eta$. Soundness is stated by Theo. 5.3 where $s_{\texttt{in}} \mid \mu_{\texttt{in}}$ is the initial configuration as defined in Sect. 4 for η in $\mathcal{TS}_\mathcal{G}$.

Theorem 5.3 (Soundness) *If* $\eta \models s_{\texttt{in}} \,|\, \mu_{\texttt{in}} \rightarrow^* s \,|\, \mu$, *then either* $s = \texttt{unit}$ *or* $\eta \models s \,|\, \mu \rightarrow s' \,|\, \mu'$ *for some* s' *and* μ'.

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

6 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 5.

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 `eval` of an instance of the nonterminal X. If there is no `eval` 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 $p_X : X \to X\,Y$ and let P' contain the single production $p_Y : Y \to X\,Y$. 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 $T_S(\mathcal{S}_P \circ$

$\mathcal{S}_{P'}) = \emptyset, \{X{:}(R^X \cup A_1^X, D^X), Y{:}(R^Y \cup A_1^Y, D^Y)\}$.

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 6.1 (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 6.2 (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 $\Gamma' \vdash \mathcal{TS}_P : \Gamma$.*

7 Related Work and Discussion

As discussed in Section 2 the tools which is more closely related to Nerverlang are Lisa [9] and Silver [11]. In Lisa an attribute must be declared and thus it is impossible to use an undefined attribute by mistake. Lisa also checks that the value of a declared attribute is set at least at some point in the semantic action. However, it does not perform any control-flow analysis to check that the attribute value is *always* set. For example, if we have a production `START` $\rightarrow$ `E` and the action code is `if(false) START.val = E.val;`, Lisa will accept it as a valid grammar definition, although `START.val` will never be defined. Our type system can capture this kind of errors and detect that the grammar definition is incomplete. In Silver the well-definedness analysis proposed by Kaminski *et al.* [6] successfully addresses the problems of closure and non-circularity. However, similarly to Lisa, Silver does not guarantee that equations actually set the values of attributes. Such errors are captured only at run-time.

We could apply our system to the previous frameworks. For Lisa, among others we would have to map Lisa's DSL constructs to the language defined in Sect. 3. Particular attention would be needed when mapping the construct for reading an attribute a at node N as in Lisa this involves also visiting the subtree rooted at N. So the attribute access construct would have to be mapped into the sequence `eval` $p[i]$; $p[i].a$ where $p[i] = N$. Silver would benefit from the proposed type system to assure that attributes always have the desired values. Moreover, the proposed type system can be used in alternative to Kaminski *et al.*'s [6] work. Applying our system to Silver would require appropriate mapping of the source DSL to the constructs of our DSL and, as with Lisa, when accessing an attribute we would have to perform an explicit visit of the node to which the attribute is attached.

The proposed type system is also applicable to some non-modular development frameworks, although the lack of modularity makes the well-definedness analysis less challenging. Even though ANTLR forces to declare attributes the compiler generator does not check that attribute values are actually set. When an attribute is defined

but not set the compiler generator assigns a default value to it and this might drive to some unexpected behavior. With our approach such errors wold be detected. Similar considerations can be done for Yacc. On the other hand, Spoofax [7] does not rely on attributes and JastAdd [5] that uses full-fledged attribute grammars cannot benefit from the proposed type system.

8 Conclusions

This paper presents a type system equipped with a type inference algorithm, for the Neverlang platform. The Neverlang platform is a modular development frameworks for language implementation, in which, programming components rely on attributes to share semantic information. The proposed type system traces attribute definition and use. Well typed components cannot cause runtime errors due to attribute misuse. The type decorations needed for the type system can be inferred from the definition/use of the attributes in the semantic actions associated with the components without knowledge of the whole language implementation.

The formalization considers a restriction of the Neverlang platform. In future work we would like to extend it to full-fledged Neverlang. In particular, have the possibility of specifying more than one role in a single slice and extend the expression language to full Java. Regarding this last point, however, we would like to separate "attribute specific" constructs, from the "host" language. This would make our approach usable also in other frameworks.

Acknowledgments. We are grateful to the anonymous reviewers for their useful suggestions and remarks.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, Massachusetts, 1986.

[2] K. Backhouse. A Functional Semantics of Attribute Grammars. In *Proc. of TACAS'02*, pp. 142–157, Grenoble, France, Apr. 2002.

[3] W. Cazzola. Domain-Specific Languages in Few Steps: The Neverlang Approach. In *Proc. of SC'12*, LNCS 7306, pp. 162–177, Prague, Czech Republic, June 2012. Springer.

[4] W. Cazzola and E. Vacchi. Neverlang 2: Componentised Language Development for the JVM. In *Proc. of SC'13*, LNCS 8088, pp. 17–32, Budapest, Hungary, June 2013. Springer.

[5] G. Hedin. An Introductory Tutorial on JastAdd Attribute Grammars. In *Proc. of GTTSE III*, LNCS 6491, pp. 166–200. Springer, 2011.

[6] T. Kaminski and E. Van Wyk. Modular Well-Definedness Analysis for Attribute Grammars. In *Proc. of SLE'13*, LNCS 7745, pp. 352–371, Dresden, Germany, Sept. 2013. Springer.

[7] L. C. L. Kats and E. Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proc. of OOPSLA'10*, pp. 444–463, Reno, Nevada, USA, Oct. 2010. ACM.

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

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

[10] E. Vacchi and W. Cazzola. Neverlang: A Framework for Feature-Oriented Language Development. *Computer Languages, Systems & Structures*, 43(3):1–40, Oct. 2015.

[11] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an Extensible Attribute Grammar System. *Science of Computer Programming*, 75(1-2):39–54, Jan. 2010.

[12] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher Order Attribute Grammars. In *Proc. of PLDI'89*, pp. 131–145, Portland, OR, USA, June 1989.