An Attribute Language Definition for Adaptable Parsing Expression Grammars

ELTON M. CARDOSO
RODRIGO G. RIBEIRO
Universidade Federal de Ouro Preto

MARIZA A. S. BIGONHA
ROBERTO S. BIGONHA
Universidade Federal de Minas Gerais

LEONARDO V. S. REIS
Universidade Federal de Juiz de Fora
lvsreis@ice.ufjf.br

VLADIMIR O. DI IORIO
Universidade Federal de Viçosa

ABSTRACT

Adaptable Parsing Expression Grammars (APEG) are a formal model whose main purpose is to formally describe the syntax of extensible languages and their extension mechanisms. APEG extends Parsing Expression Grammar model with the notion of *syntactic attributes*, which are values passed through parse tree nodes and used during the parsing process. A grammar is a first-class value passed to every nonterminal, and the rules used during the parsing are fetched from this grammar. The ability to change and pass different grammars is the key to dynamically extend the original language grammar. The reported implementation of APEG attributes uses strings and ad hoc Java code to manipulate and build grammars during parsing time. This approach has at least three disadvantages: a grammar specification becomes dependent on the language in which the functions to manipulate and build new grammars were implemented; we may not assure that the grammars built are always syntactically correct; and it is virtually impossible to prove that the generated parser does not lead to a infinite loop. In this work, we formally define an attribute language for APEG containing operators to manipulate grammars. As a result, new rules and grammars built during parsing time are syntactically correct. In addition, we define a restriction on APEG rules that assures that any generated parser will terminate on all inputs.

CCS CONCEPTS

• **Theory of computation → Grammars and context-free languages**.

KEYWORDS

APEG, PEG, parsing, attributes, extensible languages

ACM Reference Format:
ELTON M. CARDOSO, RODRIGO G. RIBEIRO, LEONARDO V. S. REIS, MARIZA A. S. BIGONHA, ROBERTO S. BIGONHA, and VLADIMIR O. DI IORIO. 2019. An Attribute Language Definition for Adaptable Parsing Expression Grammars. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297280.3299738

1 INTRODUCTION

The term *extensible language* is used to refer to a language that has constructions to extend its own syntax, as well as to associate semantics to it. This idea of offering facilities to add syntactic constructions to a language is not new and it remotes to the Lisp language and its dialects, such as Scheme and Racket [21]. Recently, the interest in languages using such philosophy instead of S-expression format has increased. The main motivations are related to the use of embedded DSLs without syntax restrictions as libraries [6, 7] and to extend proof assistant notations [1].

As an example, Fig. 1 describes the syntax of the toy extensible language μSugar. A program in this language is a sequence of new syntax declarations followed by a list of, possibly extended, statements. A regular statement, represented by *stmt*, may be an assignment, an I/O command, a conditional statement, or a repeat statement. An extended statement block (*extBlock*) begins with a list of one or more syntax names, followed by a block.

Fig. 2 shows a program written in μSugar. In lines 1 to 4 two new rules are declared and named as *sfor*. The second rule, line 3, defines the syntax of *for* statements and the first one, line 2, defines that it may be used as μSugar statement. In that declaration, terminal symbols are denoted between primes. This part of code only declares a new set of rules and it does not effectively extend the language grammar. The code of lines 5 to 9 is an extended block statement. Its semantics is to activate the parsing process according to the specified syntactic extension, in this case *sfor*, so that it may be used in its block, therefore effectively extending the language. It also defines the scope of a new syntax. After line 9, the *for* statement is not available anymore. This scope mechanism allows to define several new syntaxes and use each one, combined or not, in different pieces of the program.

Fig. 1 Syntax of the extensible language μSugar.

⟨*Prog*⟩ ::= ⟨*newSyn*⟩* ⟨*extStmt*⟩+
⟨*newSyn*⟩ ::= `'define'` ⟨*sName*⟩ '{' ⟨*rule*⟩+ '}'
⟨*rule*⟩ ::= ⟨*ntName*⟩ `'->'` ⟨*pattern*⟩ ';'
⟨*pattern*⟩ ::= ⟨*pseq*⟩ ('/' ⟨*pseq*⟩)*
⟨*pseq*⟩ ::= ⟨*prefix*⟩ ⟨*prefix*⟩*
⟨*prefix*⟩ ::= `'!'` ⟨*pterm*⟩ | ⟨*pterm*⟩
⟨*pterm*⟩ ::= ⟨*pfactor*⟩ `'*'` | ⟨*pfactor*⟩
⟨*pfactor*⟩ ::= '(' ⟨*pattern*⟩ ')' | ⟨*ntName*⟩ | LITERAL
⟨*extStmt*⟩ ::= ⟨*extBlock*⟩ | ⟨*stmt*⟩
⟨*extBlock*⟩ ::= `'syntax'` ⟨*sName*⟩ (',' ⟨*sName*⟩)* ⟨*block*⟩
⟨*block*⟩ ::= '{' ⟨*stmt*⟩+ '}'
⟨*stmt*⟩ ::= ⟨*attr*⟩ ';' | `'print'` '(' ⟨*expr*⟩ ')' ';' | `'read'` '(' ⟨*var*⟩ ')' ';'
  | `'if'` '(' ⟨*expr*⟩ ')' ⟨*block*⟩ | `'loop'` '(' ⟨*expr*⟩ ')' ⟨*block*⟩
⟨*attr*⟩ ::= ⟨*var*⟩ `':='` ⟨*expr*⟩
⟨*expr*⟩ ::= ⟨*cexpr*⟩ (('+' | '-') ⟨*cexpr*⟩)*
⟨*cexpr*⟩ ::= ⟨*factor*⟩ (('<' | '=') ⟨*factor*⟩)*
⟨*factor*⟩ ::= '(' ⟨*expr*⟩ ')' | ⟨*var*⟩ | `'true'` | `'false'` | INT
⟨*var*⟩ ::= ID
⟨*sName*⟩ ::= ID
⟨*ntName*⟩ ::= ID

Fig. 2 An example of a μSugar program

```
define sfor {
  stmt -> nfor;
  nfor -> 'for' '('attr';' expr';' attr')' block;
}
syntax sfor {
  for (i := 1; i < 10; i := i + 1) {
     print i;
  }
}
i := 1;
loop (i < 10) {
  print(i);
  i := i + 1;
}
```

Although μSugar is a toy language, it mimics all extensible features provided by SugarJ language [7].

Considering the standard theory of context-free grammars (CFG), the grammar in Fig. 1 does not generate the program of Fig. 2, because the syntax of the new *for* command was not defined by the original production rules. In fact, the CFG model offers no support to handle dynamic changes in the grammar produced while an input is processed. Adaptable Parsing Expression Grammars (APEG) [15, 16] are an alternative to formally describe such on-the-fly grammar extensions. APEG extends Parsing Expression Grammar (PEG) [8] model with the notion of *syntactic attributes*, which are values passed through nonterminals and used during the parsing process. A grammar is a first-class value passed as attribute to every nonterminal, and the syntactic rules used during the parsing are fetched from this grammar. The ability to pass modified versions of the grammar as attributes is the key to dynamically extend the original grammar. Although it has been showed that syntactic attributes represent a possible solution for describing grammar extensibility, previous APEG works do not formally define the attribute language [15, 16]. Reported implementations [16, 17] use strings and ad hoc Java code to manipulate and build grammars during parsing time. This approach has at least three disadvantages. First, a grammar specification becomes dependent of the language in which the functions to manipulate and build new grammars were implemented. Second, it is not possible to assure that the grammars built are always syntactically correct. Third, it is virtually impossible to prove that the generated parser does not lead to an infinite loop.

In this work, after presenting a brief introduction to APEG in Section 2, we formally define an attribute language for APEG containing operators to manipulate grammars in Section 3, and a type system for it in Section 4. As a result, new rules and grammars built during parsing time are syntactically correct. It is known that some PEG grammars may lead to an infinite loop, and this problem affects APEG as well. To overcome that, in Section 5 we define a restriction on APEG rules which assures that any generated parser will terminate for all inputs. Section 6 discusses related works and Section 7 concludes this paper.

2 ADAPTABLE PARSING EXPRESSION GRAMMARS

The grammar presented in Fig. 1 is not able to generate the program of Fig. 2 because CFGs do not handle dynamic changes in their own set of production rules. Grammar extensions are achieved in APEG with a special syntactic attribute, called language attribute. It represents possibly modified versions of the grammar used during the parsing process. This mechanism may be used to formally define the syntax of extensible languages [17]. In the sequel, we use μSugar as an example to introduce the main ideas of APEG.

APEG draws a distinction between two kinds of attributes: *inherited* and *synthesized*. Inherited attributes are those whose values are defined outside of the nonterminal rule definition, acting such as function parameters. On the other hand, synthesized attribute values are defined by the evaluation of a nonterminal on the right-hand side of a rule definition. Therefore, synthesized attributes play a similar role of function return values. We use the notation $\langle A{\downarrow}\vartheta_1{\downarrow}\ldots{\downarrow}\vartheta_p{\uparrow}e_1{\uparrow}\ldots{\uparrow}e_q\rangle$ to denote a nonterminal A and its syntactic attributes. A downside arrow indicates an inherited attribute and an upside arrow, a synthesized attribute. For simplicity and without loss of generality, we assume that all inherited attributes are represented in a nonterminal before its synthesized attributes.

Fig. 3 presents an APEG specification for the μSugar syntax[1]. The first step to obtain an APEG grammar for the μSugar language is to include the inherited *language attribute* to each nonterminal. As mentioned before, this attribute represents the current grammar available during the parsing process. Exemplifying, the rule for the nonterminal *Prog* turns to $\langle Prog{\downarrow}g\rangle ::= \langle newSyn{\downarrow}g{\updownarrow}\sigma\rangle^{*}\ \langle extStmt{\downarrow}g{\downarrow}\sigma\rangle+$. Note that the only difference to the *Prog* rule of Fig. 1 are the attributes. In this rule definition, the same grammar value, named as

[1]For being consistent with EBNF style of Fig. 1, we use the symbol | as the prioritized choice instead of /.

Fig. 3 APEG specification of the μSugar syntax.

$\langle \mathit{Prog}{\downarrow}g \rangle$ ::= $\langle \mathit{newSyn}{\downarrow}g{\updownarrow}\sigma \rangle$* $\langle \mathit{extStmt}{\downarrow}g{\downarrow}\sigma \rangle$+
$\langle \mathit{newSyn}{\downarrow}g{\downarrow}\sigma{\uparrow}\sigma[n\ /\ rs] \rangle$ ::= '`define`' $\langle \mathit{sName}{\downarrow}g{\uparrow}n \rangle$ '`{`'
    ($\langle \mathit{rule}{\downarrow}g{\uparrow}r \rangle$ [rs$\leftarrow$rs $\lhd$ r;])+ '`}`'
$\langle \mathit{rule}{\downarrow}g{\uparrow}\mathit{def\ nt\ p} \rangle$ ::= $\langle \mathit{ntName}{\downarrow}g{\uparrow}nt \rangle$ '`->`'$\langle \mathit{pattern}{\downarrow}g{\uparrow}p \rangle$'`;`'
$\langle \mathit{pattern}{\downarrow}g{\uparrow}p \rangle$ ::= $\langle \mathit{pseq}{\uparrow}p \rangle$ ('`/`' $\langle \mathit{pseq}{\uparrow}p_1 \rangle$ [p$\leftarrow p \oslash p_1$;])*
$\langle \mathit{pseq}{\downarrow}g{\uparrow}p \rangle$ ::= $\langle \mathit{prefix}{\downarrow}g{\uparrow}p \rangle$ (($\langle \mathit{prefix}{\downarrow}g{\uparrow}p_1 \rangle$ [p$\leftarrow p \odot p_1$;])*
$\langle \mathit{prefix}{\downarrow}g{\uparrow}p \rangle$ ::= '`!`' $\langle \mathit{pterm}{\downarrow}g{\uparrow}p_1 \rangle$ [p$\leftarrow \overset{\circ}{!} p_1$;] | $\langle \mathit{pterm}{\downarrow}g{\uparrow}p \rangle$
$\langle \mathit{pterm}{\downarrow}g{\uparrow}p \rangle$ ::= $\langle \mathit{pfactor}{\downarrow}g{\uparrow}p_1 \rangle$ '`*`' [p$\leftarrow p_1 \circledast$;] | $\langle \mathit{pfactor}{\downarrow}g{\uparrow}p \rangle$
$\langle \mathit{pfactor}{\downarrow}g{\uparrow}p \rangle$ ::= '`(`' $\langle \mathit{pattern}{\downarrow}g{\uparrow}p \rangle$ '`)`' | $\langle \mathit{ntName}{\downarrow}g{\uparrow}p \rangle$
  | p=LITERAL
$\langle \mathit{extStmt}{\downarrow}g{\downarrow}\sigma \rangle$ ::= $\langle \mathit{extBlock}{\downarrow}g{\downarrow}\sigma \rangle$ | $\langle \mathit{stmt}{\downarrow}g \rangle$
$\langle \mathit{extBlock}{\downarrow}g{\downarrow}\sigma \rangle$ ::= '`syntax`' $\langle \mathit{sName}{\downarrow}g{\uparrow}n \rangle$ [$g_1\leftarrow$g $\lhd$ $\sigma[\![n]\!]$;]
    ('`,`'$\langle \mathit{sName}{\downarrow}g{\uparrow}n \rangle$[$g_1\leftarrow g_1$ $\lhd$ $\sigma[\![n]\!]$;])* $\langle \mathit{block}{\downarrow}g_1 \rangle$+
$\langle \mathit{ntName}{\downarrow}g{\uparrow}n \rangle$ ::= n=ID

g, inherited by nonterminal *Prog* is passed to nonterminals *newSyn* and *extStmt*, therefore there are not changes in the grammar. Changing happens only in the *extBlock* nonterminal using a map from name to syntax definition that is passed to it through its inherited attribute σ. The value of this map is calculated during the evaluation of the nonterminal *newSyn*. We use a syntactic sugar notation $\updownarrow$ to denote an inherited and a synthesized attribute with the same expression. Therefore, $\langle \mathit{newSyn}{\downarrow}g{\updownarrow}\sigma \rangle$ is equivalent to $\langle \mathit{newSyn}{\downarrow}g{\downarrow}\sigma{\uparrow}\sigma \rangle$ and means that the σ value is passed as the second inherited attribute to *newSyn* and the value of its synthesized attribute is set to σ after the nonterminal evaluation. Nonterminal *newSyn* has an inherited attribute named σ which is a map from name to rules and synthesizes a new map, using the expression $\sigma[n/rs]$. The evaluation of this expression returns a map that is equal to σ, except that the name n binds to the value of rs. The n comes from the synthesized value of nonterminal *sName*, which parses the name of the syntax being defined. The set of rules, rs, is built from each evaluation of the nonterminal *rule* by extending the grammar value of the previous grammar, using $\lhd$ operation, with the grammar value of its synthesized attribute r. Then, the name rs is bound to the new set of rules.

A grammar is a set of rules, therefore, a new grammar, with just one rule, is produced as the value of the synthesized attribute of the nonterminal *rule* using the expression *def nt p*. This expression creates a grammar where the right-hand side of the nonterminal, whose name is defined by *nt*, is the parsing expression that comes from p[2]. The *nt* value is obtained from the synthesized attribute of nonterminal *ntName*, which uses a *bind* APEG parsing expression (*n=ID*) to map n to the string parsed by *ID*. *ID* parses an identifier and its rule is omitted. The rules of nonterminals *var* and *sName* are similar to *ntName* rule and were omitted as well.

Nonterminal *pattern* produces on its synthesized attribute p a parsing expression representation, which is built using operators, which is discussed on Section 3, to construct the correct representation for each parsing expression type.

We highlight that, during the evaluation of nonterminal *newSyn*, no changes were made in the language attribute and it produces only a map with new syntaxes definitions. Therefore the initial grammar remains the same at this point. On-the-fly grammar modification happens only during the evaluation of nonterminal *extBlock* when parsing an extended statement. It creates a new grammar, g_1, that is an extension of the language attribute grammar containing the rules from the new syntax, which comes from the evaluation of the $\sigma[\![n]\!]$ map expression. Afterwards, the grammar g_1 is passed as the language attribute of nonterminal *block* making available the new syntax rules only in this parsing branch. Rules for the other nonterminals are the same except for the addition of the language attribute, so we omitted them in Fig. 3.

It is important to mention that, different from Attribute Grammars (AGs), APEG syntactic attributes are used and evaluated during parsing time, making it possible to change the set of rules used to parse the remained input, thus APEG semantics imposes a left to right evaluation order [16]. As a result, some issues common in AGs, such as circularities attributes and evaluation order, are not relevant in the APEG context.

3 APEG ATTRIBUTE LANGUAGE

The key of APEG on-the-fly grammar modification resides on the ability to build and use grammars during the parsing process and to treat them as first-class values, which are manipulated by *syntactic attributes*. Previous works [15, 16] did not approach an important aspect of APEG attributes, the formal definition of the attribute language.

Before presenting a formal definition of APEG attributes, we introduce some notations. We let $\overline{x}$ denote a finite sequence of elements and we allow ourselves a bit of informality by using set operations on sequences and their meaning as usual. Notation $\overline{x}^n$, $n \geq 0$, denotes a sequence with n elements. Finite mappings are represented as a sequence of key-value pairs denoted by $\overline{k\,/\,v}$. We use meta-variable σ to denote an arbitrary finite mapping. A map may be constructed based on another one σ, denoted by $\sigma[k/v]$. It means a map equals to σ, except on the key-entry k/v. Finally, notation $\sigma[\![k]\!]$ denote the value v associated with key k in map σ, i.e. $\sigma[\![k]\!] = v$, if $k/v \in \sigma$. When there is no entry for a key k in σ, $\sigma[\![k]\!] = \bot$, where $\bot$ denotes an undefined value.

Fig. 4 shows the abstract syntax of APEG language. The first two rules, r and p, denote APEG productions and parsing expressions, respectively. An APEG production is represented by the nonterminal name, denoted by the capital letter A, a list of inherited attributes and its respective type, $\overline{\vartheta :: \tau}^n$, a list of *attribute expressions* for its synthesized attributes annotated with their type and its right-hand parsing expression. For distinguishing between nonterminal name and attribute variable, we use the meta-variable ϑ to denote an attribute variable name and capital letters for nonterminal names. An

[2]For clarity, we simplify that definition. Sections 3 and 4 show that a nonterminal definition also includes information of its attributes.

Fig. 4 APEG abstract syntax.

$$\begin{aligned}
r &::= \langle A\ \overline{\vartheta :: \tau}^{n}\ \overline{e :: \tau}^{m} \rangle \rightarrow\ p \\
p &::= p\,.\,p \mid p\ /\ p \mid\ !p \mid p* \mid A\ \overline{e}\ \overline{\vartheta} \\
&\quad \mid \vartheta = p \mid \vartheta \leftarrow e \mid\ ?e \mid s \mid \lambda \\
e &::= l \mid \vartheta \mid \{\overline{e\ /\ e}\} \mid e\,[e\ /\ e] \mid e[\![e]\!] \\
&\quad \mid e \oplus e \mid e \lhd e \mid m \\
m &::= m_p \mid m_e \\
m_p &::= e \odot e \mid e \oslash e \mid \mathring{!}\ e \mid e\ \circledast \\
&\quad \mid \langle e\ \overline{e}\ \overline{e} \rangle \mid e \mathrel{\mathring{=}} m_p \mid e \mathrel{\mathring{\leftarrow}} m_e \mid \mathring{?}\ m_e \\
&\quad \mid s \mid \#e \mid \mathring{\lambda} \mid def\ \ e\ \overline{\vartheta :: \tau}^{n}\ \overline{e :: \tau}^{m}\ e \\
m_e &::= \#\{e\}
\end{aligned}$$

APEG parsing expression[3] may be a sequence, $p.p$; a prioritized choice, p/p; a not-predicate, $!p$; a repetition, $p*$; a nonterminal reference, $A\ \overline{e}\ \overline{\vartheta}$; a bind, $\vartheta = p$; an update, $\vartheta \leftarrow e$; a *constraint*, $?e$; a string literal, s; or an empty, λ, parsing expression [15].

The interesting rules are the ones for *attribute expressions*, denoted by e, which we defined as a simply typed language. The attribute language includes literals as primitive attribute expressions, variables and maps. We use the meta-variable l to denote an arbitrary literal.Following the common practice, all (meta-)variables may appear primed or subscripted. The syntax $e \oplus e$ represents all binary operator of the language, which includes usual operators for arithmetical, logical, relational expressions and concatenation of strings. We distinguish the operator $\lhd$ used to extend grammars, whose semantics is as defined in [16]. As grammars are first-class values, the attribute language must provide operators to build grammar values, which are meta-language constructors. All meta-language constructors are denoted by m. We separate these constructors in two categories: operators for building parsing expressions, m_p, and operators for building attribute expressions itself, m_e, except the meta-language expressions.

The operator symbols $\odot$, $\oslash$, $\mathring{!}$ and $\circledast$ are used to build sequence, prioritized choice, not-predicate and zero-or-more repetitions parsing expressions. Nonterminal parsing expressions are built using the operator pair $\langle$ and $\rangle$ which means a reference to nonterminal with its respectively attribute expressions. Note that a nonterminal name is also determined, on parsing time, by the evaluation of an expression. The $\mathring{\lambda}$ symbol builds a representation of an empty parsing expression and the operator # a literal parsing expression based on the value of an expression. Also, a string is used to build a literal parsing expression, representing a sequence of terminals. These operators build common parsing expressions defined by Ford [8], with the addition of attribute expressions on nonterminals. In addition, the unary operator $\mathring{?}$ builds a constraint parsing expression, which succeeds without consuming input symbols when the correspondent attribute expression is evaluated to true. The operators $\mathring{=}$ and $\mathring{\leftarrow}$ are used to build a bind or an update parsing expression, depending on type of the second operand. The meaning of both parsing expressions is to create a new environment by changing the bind of a variable. The main difference is that the former uses the text matched by a parsing expression and the latter the value evaluated from an attribute expression as the new bind value for the variable. A detailed formal description of APEG parsing expressions semantics is found in [16]. Finally, the enclosed operator symbols #{ and } build an abstract representation from an attribute expression.

[3]For now, we will use parsing expressions as a synonymous for APEG parsing expressions.

Fig. 5 Type Language for APEG Attributes

$$\begin{aligned}
\tau &::= \alpha \mid \mathring{\kappa} \mid \mathring{\gamma} \mid \phi \mid \psi \mid \overline{\tau}^{n} \rightarrow \overline{\tau}^{m} \\
\alpha &::= \rho \mid \gamma \mid \sigma[\![\tau]\!] \mid \rho \rightarrow \rho \rightarrow \rho
\end{aligned}$$

In order to allow the user to create rules, the *def* construction provides a way to define a new nonterminal in the meta-programming level. The new nonterminal is defined by an identifier, whose value comes from the evaluation of the expression e, followed by its inherited and synthesized attributes definition and by the abstract representation of the parsing expression that corresponds to its right-hand side.

4 APEG TYPE SYSTEM

Fig. 5 describes the type language for the APEG attributes. Meta-variable τ denotes an arbitrary type, and ρ, basic types. We assume that ρ contains, at least, type constructors for integers, strings, booleans and floating-point numbers. We also assume the existence of a function that returns the type of an input literal, $\varphi : l \rightarrow \rho$.

We let γ denote the type of parsing expressions, $\sigma[\![\tau]\!]$ the type for finite mappings with image formed by values of type τ, ϕ the type for the language attribute and ψ the type of set of rules. Types α do not represent the language attribute neither are meta-level expression types. Notation $\mathring{\kappa}$ denotes the type for meta expressions. Inhabitants of type $\mathring{\kappa}$ are ASTs whose structure is an expression of an arbitrary type. Notation $\mathring{\gamma}$ denotes the types of meta-level parsing expressions. Types of the form $\overline{\tau}^{n} \rightarrow \overline{\tau}^{m}$ $(n, m \geq 0)$ are assigned to non-terminals with n inherited attributes and m synthesized attributes. Finally, types for basic binary operators are denoted by $\rho \rightarrow \rho \rightarrow \rho$. We distinguish a type for the language attribute (ϕ), and a type of the rule set ψ, to ensure that grammar modifications are safe. Later, we present a motivation example that justified this decision.

A typing context Γ is a finite mapping between variable names and their types. Following common practice, we use notation $\Gamma(x)$ to denote the type associated with x in Γ, $\Gamma[\![x]\!]$. We let notation $\vec{\Gamma}$ denote the subset of key-value pairs of non-terminal types, i.e. $\vec{\Gamma} = \{x/\tau \mid \tau = \overline{\tau_1}^{n} \rightarrow \overline{\tau_2}^{m}\}$.

We split the type rules for attributes and meta-expressions in several figures for a better organization. For the sake of brevity, we omit the presentation and explanation of trivial rules. The type judgments have the form $\Gamma \vdash x :: \tau \rightsquigarrow \Gamma'$ where x is an attribute, e, or a parsing expression, p, τ is a type and Γ' is a context possibly produced as side-effect.

Fig. 6 presents typing rules for parsing expressions. Rule ***P-Alt*** states that a well typed choice parsing expressions must have its sub-expressions well-typed. It is our intention that the common definitions within both choices reach the resulting context, then existing after the choice of parsing the expression. Rule ***P-Call*** states that for a nonterminal call to be considered well-typed the attribute expressions must match the types with which the nonterminal was defined and the first attribute must have type ϕ. Furthermore, all synthesized attributes variables absent in Γ, referenced by S in rule ***P-Call***, must be included in Γ.

Rules ***P-Declare*** and ***P-Bind-Declare*** behave like ***P-Update*** and ***P-Bind***, except that the former requires the right hand side to have a type in the context while the latter act as a delcaration of a new identifier.

Fig. 6 Typing rules for parsing expressions.

$$\frac{\Gamma \vdash p_1 :: \gamma \rightsquigarrow \Gamma' \qquad \Gamma' \vdash p_2 :: \gamma \rightsquigarrow \Gamma''}{\Gamma \vdash p_1 . p_2 :: \gamma \rightsquigarrow \Gamma''}\ \textbf{\textit{P-Seq}}$$

$$\frac{\Gamma \vdash p_1 :: \gamma \rightsquigarrow \Gamma' \qquad \Gamma \vdash p_2 :: \gamma \rightsquigarrow \Gamma''}{\Gamma \vdash p_1 / p_2 :: \gamma \rightsquigarrow \Gamma' \cap \Gamma''}\ \textbf{\textit{P-Alt}}$$

$$\frac{\Gamma \vdash p :: \gamma \rightsquigarrow \Gamma'}{\Gamma \vdash p* :: \gamma \rightsquigarrow \Gamma'}\ \textbf{\textit{P-Star}} \qquad \frac{\Gamma \vdash e :: Bool \rightsquigarrow \Gamma}{\Gamma \vdash ?\,e :: \gamma \rightsquigarrow \Gamma}\ \textbf{\textit{P-Cond}}$$

$$\frac{\begin{array}{c}\Gamma(A) = \overline{\tau}^n \rightarrow \overline{\tau'}^m \\ \Gamma \vdash e_i :: \tau_i \rightsquigarrow \Gamma, 1 < i \leq n \\ S = \{\vartheta_j :: \tau'_j \mid \Gamma(\vartheta_j) = \bot, 1 \leq j \leq m\} \\ \forall_{\vartheta_k \notin S} \Gamma(\vartheta_k) = \tau'_k, 1 \leq k \leq m \\ \Gamma \vdash e_1 :: \phi \rightsquigarrow \Gamma\end{array}}{\Gamma \vdash A\ \overline{e}\ \overline{\vartheta} :: \gamma \rightsquigarrow \Gamma \cup S}\ \textbf{\textit{P-Call}}$$

$$\frac{\Gamma(\vartheta) = \tau \qquad \Gamma \vdash e :: \tau \rightsquigarrow \Gamma \qquad \tau \neq \gamma}{\Gamma \vdash \vartheta \leftarrow e :: \gamma \rightsquigarrow \Gamma}\ \textbf{\textit{P-Update}}$$

$$\frac{\Gamma(\vartheta) = \bot \qquad \Gamma \vdash e :: \tau \rightsquigarrow \Gamma \qquad \tau \neq \gamma}{\Gamma \vdash \vartheta \leftarrow e :: \gamma \rightsquigarrow \Gamma, \{\vartheta :: \tau\}}\ \textbf{\textit{P-Declare}}$$

$$\frac{\Gamma(\vartheta) = String \qquad \Gamma \vdash p :: \gamma \rightsquigarrow \Gamma}{\Gamma \vdash \vartheta = p :: \gamma \rightsquigarrow \Gamma}\ \textbf{\textit{P-Bind}}$$

$$\frac{\Gamma(\vartheta) = \bot \qquad \Gamma \vdash p :: \gamma \rightsquigarrow \Gamma}{\Gamma \vdash \vartheta = p :: \gamma \rightsquigarrow \Gamma, \{\vartheta :: String\}}\ \textbf{\textit{P-Bind-Declare}}$$

Fig. 7 presents the typing rules for map expressions and grammar extension operator. Rule ***T-MPA*** states that the lookup operation on a map is well-typed only when it is applied on an expression that has type of a map, $\sigma[\![\tau]\!]$, its argument expression must have type *String*, and the resulting type is the inner map type. Rule ***T-MPE*** determines that a map extension $e_1[e_2/e_3]$ operation is well-typed if e_1 has type $\sigma[\![\tau]\!]$, e_2 has type *String* and e_3 has a type τ, matching the inner type of the map. Rule ***T-Compose-Rules*** states that two rule sets may be concatenated, resulting in a new rule set. Finally, rule ***T-Compose*** ensures that a language value may only be constructed by composing the current language attribute with a rule set attribute. This rule is crucial to certify that the grammar is always being extended and non-terminal definitions are never removed from it. As an example of a bad behavior that might arise by the absence of this rule, consider Fig. 8. In this example, a new grammar, containing only one rule named C, is created and binded with name x. Next, that grammar is passed as the language attribute of rule B. As a result, the body of B must be retrieved from that grammar, however it does not have a definition for rule B thus, resulting in an error. Therefore, the ***T-Compose*** ensure that value passed as language attributes are safe.

Fig. 7 Typing rules for map and grammar extension.

$$\frac{\Gamma \vdash e_1 :: \sigma[\![\tau]\!] \rightsquigarrow \Gamma \qquad \Gamma \vdash e_2 :: String \rightsquigarrow \Gamma}{\Gamma \vdash e_1[\![e_2]\!] :: \tau \rightsquigarrow \Gamma}\ \textbf{\textit{T-MPA}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 :: \sigma[\![\tau]\!] \rightsquigarrow \Gamma \\ \Gamma \vdash e_2 :: String \rightsquigarrow \Gamma \\ \Gamma \vdash e_3 :: \tau \rightsquigarrow \Gamma\end{array}}{\Gamma \vdash e_1[e_2/e_3] :: \sigma[\![\tau]\!] \rightsquigarrow \Gamma}\ \textbf{\textit{T-MPE}}$$

$$\frac{\Gamma \vdash e_1 :: \phi \rightsquigarrow \Gamma \quad \Gamma \vdash e_2 :: \psi \rightsquigarrow \Gamma}{\Gamma \vdash e_1 \lhd e_2 :: \phi \rightsquigarrow \Gamma}\ \textbf{\textit{T-Compose}}$$

$$\frac{\Gamma \vdash e_1 :: \psi \rightsquigarrow \Gamma \quad \Gamma \vdash e_2 :: \psi \rightsquigarrow \Gamma}{\Gamma \vdash e_1 \lhd e_2 :: \psi \rightsquigarrow \Gamma}\ \textbf{\textit{T-Compose-Rules}}$$

All meta-expressions, presented in figures 9, 10 and 11, produce an abstract representation (AST) of APEG expressions rather than a value. Therefore, $\mathring{\gamma}$ is the type of such expressions whenever they produce an AST for a parsing expression or type $\mathring{\kappa}$ when they produce an AST for an attribute expression.

Rules in Fig. 9 are similar to the ones in Fig. 6, except that they all construct ASTs. Therefore, they conclude type $\mathring{\gamma}$, except the rule ***T-MEXP*** which builds an AST for an attribute expression and, thus, concludes type $\mathring{\kappa}$. None of these rules make changes in the context.

In Fig. 10, we present the rules for building *bind* and *update* parsing expressions. Rule ***T-PEG-Attr*** types an operation for building a *bind* parsing expressions. This construction results in the AST of an expression that will store the consumed input in a variable. Therefore the variable, which value comes from the evaluation of an expression, must have type *String*. The rule ***T-EXP-Attr*** expresses the case for building an *update* parsing expression, which is similar to the one for *bind* expression, except that the right-hand side of the rule is an AST for an attribute expression.

Fig. 11 shows the rule for nonterminals definition. The rule ***T-Nt-Decl*** defines a new nonterminal. Notice that this rule only check if the expression whose gives the nonterminal name has type *String* and if the correspondent right-hand definition has type $\mathring{\gamma}$. The reason for this is because some values are known only when parsing the input string. Therefore, a complete type verification is possible only after knowing these values.

The required dynamic type of meta-rules occurs only when the attributed language is extended. At this point, meta-rules,

Fig. 8 An example of an invalid grammar operation

```
⟨A g :: φ⟩ –> “00” (x ← def C g :: φ “10”) (B x)
⟨B g :: φ⟩ –> “11”
```

Fig. 9 Meta typing rules for basic parsing expressions.

$$\frac{\Gamma \vdash e_1 :: \mathring{\gamma} \leadsto \Gamma \qquad \Gamma \vdash e_2 :: \mathring{\gamma} \leadsto \Gamma}{\Gamma \vdash e_1 \odot e_2 :: \mathring{\gamma} \leadsto \Gamma}\ \textbf{\textit{T-MSeq}}$$

$$\frac{\Gamma \vdash e_1 :: \mathring{\gamma} \leadsto \Gamma \qquad \Gamma \vdash e_2 :: \mathring{\gamma} \leadsto \Gamma}{\Gamma \vdash e_1 \oslash e_2 :: \mathring{\gamma} \leadsto \Gamma}\ \textbf{\textit{T-MAlt}}$$

$$\frac{}{\Gamma \vdash \#\{e\} :: \mathring{\kappa} \leadsto \Gamma}\ \textbf{\textit{T-MExp}} \qquad \frac{\Gamma \vdash e :: \mathring{\kappa} \leadsto \Gamma}{\Gamma \vdash \mathring{?}e :: \mathring{\gamma} \leadsto \Gamma}\ \textbf{\textit{T-MCond}}$$

Fig. 10 Meta typing rules for update parsing expressions.

$$\frac{\Gamma(e_1) = String \qquad \Gamma \vdash e_2 :: \mathring{\gamma} \leadsto \Gamma}{\Gamma \vdash e_1 \overset{\circ}{=} e_2 :: \mathring{\gamma} \leadsto \Gamma}\ \textbf{\textit{T-PEG-Attr}}$$

$$\frac{\Gamma(e_1) = String \qquad \Gamma \vdash e_2 :: \mathring{\kappa} \leadsto \Gamma}{\Gamma \vdash e_1 \overset{\circ}{\leftarrow} e_2 :: \mathring{\gamma} \leadsto \Gamma}\ \textbf{\textit{T-EXP-Attr}}$$

Fig. 11 Typing rules for APEG: Nonterminals

$$\frac{\begin{array}{c}\Gamma \vdash p :: \mathring{\gamma} \leadsto \Gamma \\ \Gamma \vdash e_a :: String \leadsto \Gamma\end{array}}{\Gamma \vdash def\, e_a\, \overline{(\vartheta :: \tau)}^{n}\, \overline{(e :: \tau')}^{m} p :: \psi \leadsto \Gamma}\ \textbf{\textit{T-NT-Decl}}$$

whose type is $\mathring{\gamma}$, are evaluated resulting in the complete AST of a rule. The resulting AST is then submitted to the same typing rules presented in this text, in a typing context that contains the types for all the rules previously checked, plus the ones to be added.

5 WELL-FORMED APEG

A Parsing Expression Grammar could be viewed as a formalization of a top-down descendent recursive parsing. Therefore, left-recursive grammars could lead to an infinite loop, such as the grammar rule $A \rightarrow A$ 'a' / 'a'. Moreover, the greedy semantics of the repetition operator * implies that a parsing expression p, which the empty string is recognized by it, will turn the expression p^* on an infinite loop as well. Ford [8] uses a simple approach to assure termination by just requiring that at least one input symbol is consumed by the enclosed expression of the repetition operator and rules are not direct or mutual left-recursive.

We follow the same path of Ford [8] with two additional considerations: nonterminal-attribute evaluation must terminate and constraint and update parsing expressions are treated such as a lambda expression. The latter condition assures that new parsing expressions introduced in APEG, whose semantics does not consume any input symbol, also terminate when combined with repetition operator.

Before defining the concept of well-formed APEG, we first define a relation, $\rightharpoonup$, on APEG parsing expressions, which denotes an abstract simulation of it. The idea of this relation is to inform whether an interpretation of a parsing expression may succeed, consuming or not an input string, or fail. An element of this relation is a pair (p, o) where p

Fig. 12 Definition of APEG abstract relation.

$$\textbf{Update}\ \frac{}{\vartheta \leftarrow e \rightharpoonup 0} \qquad \textbf{Bind}\ \frac{p \rightharpoonup s}{\vartheta{=}p \rightharpoonup s}$$

$$\neg\textbf{Bind}\ \frac{p \rightharpoonup f}{\vartheta{=}p \rightharpoonup f} \qquad \textbf{True}\ \frac{}{?e \rightharpoonup 0} \qquad \textbf{False}\ \frac{}{?e \rightharpoonup f}$$

$$\textbf{Rule}\ \frac{\langle A\ \ \overline{\vartheta :: \tau}^{n}\ \ \overline{e' :: \tau}^{m}\rangle \rightarrow p \qquad p \rightharpoonup o}{\langle A\ \ \overline{e}^{n}\ \ \overline{\vartheta'}^{m}\rangle \rightharpoonup o}$$

is a parsing expression and $o \in \{0, 1, f\}$ represents if the parsing expression may succeed without consuming a symbol (0), consuming at least one symbol (1) or failing (f).

Fig. 12 shows the relation definition. This definition is straightforward to represent each possibility of a parsing expression to succeed or not on a given input. We omit the rules for standard parsing expressions and present just the rules for binds (rules **Bind** and **¬Bind**), constraints (rules **True** and **False**), updates (rule **Update**) and nonterminals (rule **Rule**) parsing expressions.

The semantics of a bind parsing expression $\vartheta{=}p$ is to bind the string consumed by the parsing expression p to variable ϑ. Then, it behaves exactly as parsing expression p, succeeding in consuming the same portion of the input string of p when it succeeds, and fails when p fails. Constraint expression defines a predicate on attributes, succeeding without consuming any input symbols if it evaluates to true and fails, otherwise. Therefore, there are two possibilities for an abstract evaluation of a constraint parsing expression: succeeding without consuming symbols (rule **True**) and fails (rule **False**). Note that this relation is defined independently of any input string and particular values of attributes. Assuming that the attribute type system assure all attribute expressions are correct and do not produce erroneous values, an update parsing expression just changes the bind of an attribute variable name and does not consume any input symbols. As a result, an update parsing expression always succeeds without consuming input symbols. This behavior is expressed by the rule **Update**. Finally, as the constraint rules, an abstract evaluation of a nonterminal does not consider expressions of inherited and synthesized attributes and its result is determined only by its right-hand side parsing expression abstract evaluation. In the following theorem, notation $E \vdash (p, x) \Rightarrow o \vdash E'$ comes from [15].

THEOREM 5.1. *Let E and E' be environments, p a parsing expression, x a string, o a string with at least one symbol and f the symbol representing fails. The relation $\rightharpoonup$ summarizes the semantics of APEG as follows:*

- *If $E \vdash (p, x) \Rightarrow o \vdash E'$, then $p \rightharpoonup 1$;*
- *If $E \vdash (p, x) \Rightarrow \lambda \vdash E'$, then $p \rightharpoonup 0$;*
- *If $E \vdash (p, x) \Rightarrow f \vdash E'$, then $p \rightharpoonup f$;*

PROOF. Straightforward induction over the input size. □

Based on the abstract relation $\rightharpoonup$, we define a set of well-formed parsing expressions, WF, inductive as presented in

Fig. 13 Definition of well-formed parsing expressions.

$$\textbf{Empty}\ \frac{}{\lambda \in WF} \qquad \textbf{Term}\ \frac{}{a \in WF} \qquad \textbf{True}\ \frac{}{?e \in WF}$$

$$\textbf{Seq}\ \frac{p_1 \in WF \qquad p_1 \rightharpoonup 0 \text{ implies } p_2 \in WF}{p_1 . p_2 \in WF} \qquad \textbf{Not}\ \frac{p \in WF}{!p \in WF}$$

$$\textbf{Alt}\ \frac{p_1 \in WF \qquad p_2 \in WF}{p_1 / p_2 \in WF} \qquad \textbf{Star}\ \frac{p \in WF \qquad p \not\rightharpoonup 0}{p^* \in WF}$$

$$\textbf{Bind}\ \frac{p \in WF}{\vartheta = p \in WF} \qquad \textbf{Update}\ \frac{}{\vartheta \leftarrow e \in WF}$$

$$\textbf{Rule}\ \frac{\langle A\ \ \overline{\vartheta :: \tau}^n\ \ \overline{e' :: \tau}^m \rangle \to p \qquad p \in WF}{\langle A\ \ \overline{e}^n\ \ \overline{\vartheta'}^m \rangle \in WF}$$

Fig. 13. A grammar is well-formed if all parsing expression and subexpression on it are well-formed.

THEOREM 5.2. *Let G be a well-formed adaptable parsing expression grammar. If every attribute expression in G is correct, then G handles all input string.*

PROOF. Straightforward induction over the input size. □

Because this definition does not depend on the input string and attribute expression, and there are a finite number of relevant expressions in a grammar, we may compute this set over any grammar by iteratively applying the rules on Fig. 13 until we reach a fixed point.

6 RELATED WORK

As extensible languages may change their own set of rules during parsing, the most appropriate formalisms to specify their syntaxes may be the ones which also allow modifying their own set of grammar rules. Several models have been proposed in this direction, for instance, [4, 5, 19, 20].

Christiansen [5] proposes *Adaptable Grammars*, which is essentially an Extended Attribute Grammar [22] where the first attribute of every nonterminal symbol is inherited and represents the *language attribute*. The language attribute contains the set of rules allowed in each derivation. The initial grammar works as the language attribute for the root node of the parse tree, and new language attributes may be built and used in different nodes. Each grammar adaptation is restricted to a specific branch of the parse tree. One advantage of this approach is that it is easy to define statically scoped dependent relations, such as block structure declarations of several programming languages. Although APEG was inspired in the Adaptable Grammars of Christiansen, the main difference between them are related to the models on which they are based [15]. The attribute expressions of Christiansen Adaptable Grammars may use any mathematical functions which are not clear defined. Moreover, Christiansen work does not treat termination.

Shutt [19] observes that Christiansen's *Adaptable Grammars* inherit the lack of orthogonality of attribute grammars. The CFG kernel is simple, generative, but computationally weak. The augmenting facility is obscure and computationally strong. He proposes *Recursive Adaptable Grammars (RAGs)*, where a single domain combines the syntactic elements, metasyntactic and semantic values. One problem of RAG and Christiansen's Adaptable Grammars is the difficulty to model forward references,which is important, for instance, to define the syntax of the Fortress language. Using APEG, forward references may be easily modeled with and-predicate and not-predicate operators [17]. There are no evidences that RAGs are suitable for automatic generated efficient parsers and termination guaranties. We argue that we solve Shutt's allegation about the lack of orthogonality of Adaptable Grammars giving a formal definition of a simple and restrict form of attribute language with limited computer power.

The interest on modular language extensions has been increasing in recent years. APEG is a formal model which aims to implement the method of defining modular extensions through sugar libraries and extensible languages [6, 7]. Silver [23] uses a similar approach. It is an extensible language of specification based on attribute grammars with addition of forward and high-order attributes. Silver defines a core syntax and languages are defined as extensions of it. Silver's attributes are used to define extension semantics. APEG attributes, instead, have a syntax purpose and are used during parsing time to guide it. It is an important difference on conventional attribute grammar system and APEG *syntactic* attributes. APEG does not define a way to give semantics to language constructions, although it is possible to use APEG attribute with this purpose. Another difference between APEG and Silver is the time when extensions occurs: Silver extensions are statically defined and APEG uses a dynamic on-the-fly grammar modification. To provide soundness composition of extension specifications, Schwerdfeger and Wyk [18] propose a test for checking whether grammar extensions do not produce LR conflicts when combined with others, under certain conditions. A proof that the composed compiler behavior is as expected when multiple extensions are combined is showed in Silver system [12].

Spoofax [13] relies on the idea of *language workbenches*, which provides resources to define syntax, semantics and IDE development. It underlies on SDF [9] formalism for syntax definition and uses rewrite rules on Stratego/XT [2] for defining semantic analyses and code desugaring. Given the semantics of extensions by desugaring extension nodes on base language nodes may produce code with type errors. Therefore, Lorenzen and Erdweg [14] propose a system for syntactic extensibility with guarantee that desugaring code is well-typed.

Metaprogramming is the act of writing programs that generate other programs. Our work is not exactly a metaprogramming system, but it has some similarities in the sense that our grammar language has constructors to build new grammars. Metaprogramming languages, such as MAJ [10] and SafeGen [11], use quote and unquote operator for building abstract syntax of the target language. Quote operator is used to construct the AST of the quoted string. When the target language is large, as the Java language, the quote operator simplify writing metaprograms. Our attribute language formalization does not use quote operator, instead we use explicit operators for build any kind of APEG parsing expressions. We prefer this approach because the set of APEG parsing expression is small and the explicit use of the operator clearly indicate what type of parsing expression are been built. We include unquote operator on APEG to build an abstract representation based on the value of a variable. This gives the flexibility to build programs based on dynamic value of expressions.

One advantage of metaprogramming techniques is to build syntactically correct programs, however, it may generate programs containing static semantics errors. SafeGen [11] tries to solve this problem using a theorem proof to assure that the generated code satisfies some proprieties on respect to the target type system. MetaOCaml [3] is a multistage language that keeps track of type contexts to guarantee that if the generator is type safe, the generated program is as well. Our approach keeps track the type context and uses it to assure the soundness of new rules. Note that our system does not have the problem of generated hygienic names. When a generated rule is built with the same name of an existed one, the APEG semantics combine these two definitions. And, if the definition of attributes are distinct, it is a type error.

7 CONCLUSIONS

Previous works on APEG do not formally define the attribute language, and reported implementation uses informal Java code to manipulate attributes and grammar modifications. This paper provides a formal definition of APEG type attribute language and operators for building and manipulating grammars on-the-fly. Our formalization assures that correct grammar rules are produced.

It is well known that some PEG rules may lead to an infinite loop, which is not a desired property for parser. Therefore, we presented a restricted form of grammar rules, called well-formed, and stated that well-formed APEG grammars may handle all inputs. For it, we assume that well-typed attributes terminate, however we do not provide a formal proof. Thus, the immediate future work is to prove it and, also, the soundness of the system and check it in proof assistance. Also, the well-formed relation is separated from the type system and it is performed after it. A future work is to integrate it on the type system in order to type only well-formed rules.

ACKNOWLEDGMENTS

This work is supported by the CNPq – Brazil under grant No.: 426232/2016.

REFERENCES

[1] William J. Bowman. 2016. Growing a Proof Assistant. https://www.williamjbowman.com/resources/cur.pdf

[2] Martin Bravenboer, Karl T. Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming* 72, 1-2 (2008), 52–70.

[3] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering*, Frank Pfenning and Yannis Smaragdakis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 57–76.

[4] Adam Carmi. 2010. *Adaptive Multi-pass Parsing.* Master's thesis. Israel Institute of Technology.

[5] Henning Christiansen. 1990. A survey of adaptable grammars. *SIGPLAN Not.* 25 (1990), 35–44. Issue 11.

[6] Sebastian Erdweg, Stefan Fehrenbach, and Klaus Ostermann. 2014. Evolution of Software Systems with Extensible Languages and DSLs. *IEEE Software* 31, 5 (2014), 68–75.

[7] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: library-based syntactic language extensibility. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA'11).* ACM, New York, NY, USA, 391–406.

[8] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proceedings of the 31st Symposium on Principles of Programming Languages (POPL'04).* ACM, New York, NY, USA, 111–122.

[9] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. 1989. The syntax definition formalism SDF reference manual. *SIGPLAN Not.* 24, 11 (1989), 43–75.

[10] S. Shan Huang, David Zook, and Yannis Smaragdakis. 2008. Domain-specific Languages and Program Generation with meta-AspectJ. *ACM Trans. Softw. Eng. Methodol.* 18, 2, Article 6 (2008), 32 pages.

[11] Shan S. Huang, David Zook, and Yannis Smaragdakis. 2011. Statically safe program generation with SafeGen. *Science of Computer Programming* 76, 5 (2011), 376 – 391. Special Issue on GPCE 2004/2005.

[12] Ted Kaminski and Eric Van Wyk. 2017. Ensuring Non-interference of Composable Language Extensions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017).* ACM, New York, NY, USA, 163–174.

[13] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10).* ACM, New York, NY, USA, 444–463.

[14] Florian Lorenzen and Sebastian Erdweg. 2016. Sound Type-dependent Syntactic Language Extension. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16).* ACM, New York, NY, USA, 204–216.

[15] Leonardo V.S. Reis, Roberto S. Bigonha, Vladimir O. Di Iorio, and Luis Eduardo S. Amorim. 2012. Adaptable Parsing Expression Grammars. In *Programming Languages*, Francisco Heron Carvalho Junior and Luis Soares Barbosa (Eds.). Lecture Notes in Computer Science, Vol. 7554. Springer Berlin Heidelberg, 72–86.

[16] Leonardo V.S. Reis, Roberto S. Bigonha, Vladimir O. Di Iorio, and Luis Eduardo S. Amorim. 2014. The formalization and implementation of Adaptable Parsing Expression Grammars. *Science of Computer Programming* 96, Part 2 (2014), 191 – 210. Selected and extended papers of the SBLP 2012.

[17] Leonardo V. S. Reis, Vladimir O. Di Iorio, and Roberto S. Bigonha. 2014. Defining the Syntax of Extensible Languages. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14).* ACM, New York, NY, USA, 1570–1576.

[18] August C. Schwerdfeger and Eric Van Wyk. 2009. Verifiable Composition of Deterministic Grammars. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09).* ACM, New York, NY, USA, 199–210.

[19] John N. Shutt. 1998. *Recursive Adaptable Grammars.* Master's thesis. Worchester Polytechnic Institute.

[20] Paul Stansifer and Mitchell Wand. 2011. Parsing Reflective Grammars. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications (LDTA '11).* ACM, New York, NY, USA, Article 10, 7 pages.

[21] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI'11).* ACM, New York, NY, USA, 132–141.

[22] David A. Watt and Ole Lehrmann Madsen. 1983. Extended Attribute Grammars. *Comput. J.* 26, 2 (1983), 142–153.

[23] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Science of Computer Programming* 75, 1 (2010), 39 – 54. Special Issue on LDTA 06/07.