A Parsing Expression Grammar-based Approach For Syntax Tree Pattern Matching

Guilherme Augusto Anício Drummond do Nascimento
Postgraduate Program in Computer Science, Federal University of Ouro Preto.
Ouro Preto, Brazil
guilherme.drummond@aluno.ufop.edu.br

Rodrigo Geraldo Ribeiro
Department of Computation, Federal University of Ouro Preto.
Ouro Preto, Brazil
rodrigo.ribeiro@ufop.edu.br

Abstract

We present a formal PEG-based approach to pattern matching on parse trees. Given an arbitrary Parsing Expression Grammar, we define a tree-producing semantics, a typing and subtyping relation for parse trees and patterns, and a pattern matching algorithm, all equipped with correctness proofs. The formalization is implemented as a Haskell library and we show some case studies like the design of simple static analysis tools.

Keywords

PEG, Pattern, Matching, Static analysis

1 Introduction

Pattern matching is the act of checking a given sequence of tokens for the presence of a given pattern. The match usually must be exact: "either it will or will not be a match". It is frequently used to output the locations (if any) of a pattern within a token sequence, output some component of the matched pattern, and to substitute the matching pattern with some other token sequence (i.e., search and replace). Patterns generally have the form of either sequences or tree structures. Often, patterns sequences are described using regular expressions.

In the source code analysis, pattern matching can be used to support program understanding, style enforcement, and refactoring [1]. Available tools use approaches ranging from text-based (grep, regular expressions) to tree-based (AST queries). Most of these tools either operate on plain text, losing structural information, or rely on language-specific abstract syntax trees, limiting reuse across languages. In this work, we address these concerns with a single formalism. Given a Parsing Expression Grammar (PEG) [2] that describes the language syntax, the tool parses a source file, produces a structured parse tree, and matches user-defined patterns against it. Because the grammar is supplied as input, the tool is inherently multi-language: the same engine handles Python, C, or any other language for which a PEG is provided.

More specifically, we contribute:

- A **tree-producing semantics** for arbitrary PEGs, together with a proof of equivalence to Ford's original semantics [2].
- A **typing and subtyping relation** for parse trees and patterns, with a coercion mechanism that simplifies pattern specification for choice and repetition expressions.
- A **pattern matching algorithm** over parse trees, which we argue that is correct. A machine-checked proof of its correctness is left for future work.
- A **tool implementation** and evaluation through case studies covering call graph extraction and detection of unauthorized constructs in tools for automated assessment of programming assignments.

The rest of this paper is structured as follows. Section 2 presents background on PEGs. Section 3 presents the formalization and implementation. Section 4 discusses case studies. Section 5 surveys related work, and Section 6 concludes.

2 Background

2.1 An Overview Of PEGs

Intuitively, PEGs are a formalism for describing top-down parsers. Formally, a PEG is a 4-tuple (V, Σ, R, e_S), where V is a finite set of variables, Σ is the alphabet, R is the finite set of rules, and e_S is the start expression. Each rule $r \in R$ is a pair (A, e), usually written $A \leftarrow e$, where $A \in V$ and e is a parsing expression. We let the meta-variable a denote an arbitrary alphabet symbol, A a variable and e a parsing expression. Following common practice, all meta-variables can appear primed or sub-scripted. The following context-free grammar defines the syntax of a parsing expression:

$$e \quad \rightarrow \quad \epsilon \mid a \mid A \mid e_1\, e_2 \mid e_1 \,/\, e_2 \mid e^\star \mid !\, e$$

The execution of parsing expressions is defined by an inductively defined judgment that relates pairs formed by a parsing expression and an input string to pairs formed by the consumed prefix and the remaining string. Notation $(e, s) \Rightarrow_G (s_p, s_r)$ denote that parsing expression e consumes the prefix s_p from the input string s leaving the suffix s_r. The notation $(e, s) \Rightarrow_G \perp$ denote the fact that s cannot be parsed by e. We let meta-variable r denote an arbitrary parsing result, i.e., either r is a pair (s_p, s_r) or $\perp$. We say that an expression e fails if its execution over an input produces $\perp$; otherwise, it succeeds. Figure 1 defines the PEG semantics.

Most rules are straightforward. Rule $_{Var}$ delegates parsing to the expression associated with the variable in the grammar. Rule $_{Alt_{S2}}$ enforces the ordered nature of choice: e_2 is only tried when e_1 fails. Rule $_{Star_{end}}$ ensures $e^\star$ always succeeds: when e fails, the star expression succeeds consuming nothing. The not-predicate $!\, e$ succeeds (consuming nothing) exactly when e fails, and fails when e succeeds.

Example 2.1. As an example, consider the PEG (Figure 2) which recognizes arithmetic expressions that apply the basic four operations to non-negative integers:

$$\frac{}{(\epsilon, s) \Rightarrow_G (\epsilon, s)}\ \{Eps\} \qquad \frac{}{(a, as_r) \Rightarrow_G (a, s_r)}\ \{ChrS\} \qquad \frac{a \neq b}{(a, bs_r) \Rightarrow_G \bot}\ \{ChrF\} \qquad \frac{}{(a, \epsilon) \Rightarrow_G \bot}\ \{ChrNil\}$$

$$\frac{(e_1, s_1s_2s_r) \Rightarrow_G (s_1, s_2s_r) \quad (e_2, s_2s_r) \Rightarrow_G (s_2, s_r)}{(e_1\ e_2, s_1s_2s_r) \Rightarrow_G (\langle s_1, s_2\rangle, s_r)}\ \{Cat_{S1}\} \qquad \frac{(e_1, s_ps_r) \Rightarrow_G (s_p, s_r) \quad (e_2, s_r) \Rightarrow_G \bot}{(e_1\ e_2, s_ps_r) \Rightarrow_G \bot}\ \{Cat_{F2}\}$$

$$\frac{(e_1, s) \Rightarrow_G \bot}{(e_1\ e_2, s) \Rightarrow_G \bot}\ \{Cat_{F1}\} \qquad \frac{(e_1, s_p\ s_r) \Rightarrow_G (s_p, s_r)}{(e_1\ /\ e_2, s_p\ s_r) \Rightarrow_G (s_p, s_r)}\ \{Alt_{S1}\} \qquad \frac{(e_1, s_p\ s_r) \Rightarrow\ \bot \quad (e_2, s_p\ s_r) \Rightarrow_G (s_p, s_r)}{(e_1\ /\ e_2, s_p\ s_r) \Rightarrow_G (s_p, s_r)}\ \{Alt_{S2}\}$$

$$\frac{(e, s_{p_1}s_{p_2}s_r) \Rightarrow_G (s_1, s_{p_2}s_r) \quad (e^\star, s_{p_2}s_r) \Rightarrow_G (s_2, s_r)}{(e^\star, s_1s_2s_r) \Rightarrow_G (s_1\ s_2, s_r)}\ \{Star_{rec}\} \qquad \frac{(e, s) \Rightarrow_G \bot}{(e^\star, s) \Rightarrow_G (\epsilon, s)}\ \{Star_{end}\} \qquad \frac{(e, s_p\ s_r) \Rightarrow_G (s_p, s_r)}{(!\ e, s_p\ s_r) \Rightarrow_G \bot}\ \{Not_F\}$$

$$\frac{(e, s) \Rightarrow_G \bot}{(!\ e, s) \Rightarrow_G (\epsilon, s)}\ \{Not_S\} \qquad \frac{A \leftarrow e \in R \quad (e, s) \Rightarrow_G (p, r) \quad s = pr}{(A, s) \Rightarrow_G (p, r)}\ \{Var\}$$

Figure 1: Parsing expressions operational semantics.

```
E  ←  T ('+' T)*
T  ←  F ('*' F)*
F  ←  n / '(' E ')'
```

Figure 2: PEG for arithmetic expressions.

Consider the string 1+2. The initial rule E first tries to parse a T that will, in turn, first try to consume a F, which recognizes the number '1'. It then consumes the + operator and tries to parse another T, which will consume 2 as a F and returns to E, which does not find another '+' operator and finalizes the parsing process.

3 Methodology

This section presents the pattern matching formalization and the current state of its implementation. Section 3.1 details the semantics for producing a parse tree from a parsing expression. Section 3.2 presents the syntax and semantics for parse tree patterns, along with a typing and subtyping relation. Section 3.3 discusses implementation details.

3.1 Parse Trees

In order to properly define pattern matching over trees, we need to define parse trees over an arbitrary PEG. We start by defining tree syntax and a PEG semantics which produces, as a result, such trees. Next, we present a typing relation which assigns a parsing expression for a given tree. Such step is necessary to formally state the equivalence between our proposed tree-producing semantics and PEG original semantics proposed by Ford [2].

Let $G = (V, \Sigma, R, e_s)$ be an arbitrary PEG, the meta-variable $a \in \Sigma$ an arbitrary alphabet symbol, $A \in V$ a variable and e a parsing expression. The following context-free grammar defines the syntax of a parse tree:

$$t \quad \rightarrow \quad \hat{\epsilon} \mid \hat{a} \mid A@t \mid \langle t_1, t_2\rangle \mid L\ t \mid R\ t \mid [\,] \mid t\hat{::}t \mid \eta$$

Where $\hat{\epsilon}$ represents that a parsing expression resulted in success without consuming any symbol of its input, $\hat{a}$ represents that the parsing expression consumed the symbol a from the input, $A@t$ represents that the parsing of the rule $A \leftarrow e \in R$ was successful and t is a tree for e. Notation $\langle t_1, t_2\rangle$ represents that a sequence of parsing expressions succeeded, $L\ t$ represents that the left branch in an ordered choice succeeded, while $R\ t$ represents that the right branch in an ordered choice succeeded. [] is an empty list of trees for e and $t_1\hat{::}t_2$ denotes a list of trees in which t_1 is a tree for e and t_2 is a tree for $e^\star$. Finally, η represents that a not predicate was successful.

Executing a parsing expressions produces a parsing tree and is defined by an inductively defined judgment that relates pairs formed by a parsing expression and an input string to pairs formed by the produced tree and the remaining string. Notation $(e, s_ps_r) \Downarrow_G (t, s_r)$ denote that parsing expression e consumes the prefix s_p and produces the parse tree t from the input string s_ps_r leaving the suffix s_r. The notation $(e, s) \Downarrow_G \bot$ denote the fact that s cannot be parsed by e. We let meta-variable r denote an arbitrary parsing result, i.e., either r is a pair (t, s_r) or $\bot$. We say that an expression e fails if its execution over an input produces $\bot$; otherwise, it succeeds. Figure 3 defines the PEG semantics for production of a tree.

A parse tree is directly related to its underlying parsing expression. We formalize this idea using a typing relation between grammars, trees and parsing expressions. Notation $G \vdash t : e$ means that tree t has type e using as assumption the variables and rules defined by grammar G. Figure 4 presents the rules of tree typing relation.

$$\frac{}{(\epsilon, s) \Downarrow_G (\hat{\epsilon}, s)}\ \{Eps\} \qquad \frac{}{(a, as_r) \Downarrow_G (\hat{a}, s_r)}\ \{ChrS\} \qquad \frac{a \neq b}{(a, bs_r) \Downarrow_G \bot}\ \{ChrF\} \qquad \frac{}{(a, \epsilon) \Downarrow_G \bot}\ \{ChrNil\}$$

$$\frac{(e_1, s_{p_1}s_{p_2}s_r) \Downarrow_G (t_1, s_{p_2}s_r) \quad (e_2, s_{p_2}s_r) \Downarrow_G (t_2, s_r)}{(e_1\ e_2, s_{p_1}s_{p_2}s_r) \Downarrow_G (\langle t_1, t_2\rangle, s_r)}\ \{Cat_{S1}\} \qquad \frac{(e_1, s_p s_r) \Downarrow_G (t_1, s_r) \quad (e_2, s_r) \Downarrow_G \bot}{(e_1\ e_2, s_p s_r) \Downarrow_G \bot}\ \{Cat_{F2}\}$$

$$\frac{(e_1, s) \Downarrow_G \bot}{(e_1\ e_2, s) \Downarrow_G \bot}\ \{Cat_{F1}\} \qquad \frac{(e_1, s_p\ s_r) \Downarrow_G (t, s_r)}{(e_1\ /\ e_2, s_p\ s_r) \Downarrow_G (L\ t, s_r)}\ \{Alt_{S1}\} \qquad \frac{(e_1, s_p\ s_r) \Downarrow_G \bot \quad (e_2, s_p\ s_r) \Downarrow_G t}{(e_1\ /\ e_2, s_p\ s_r) \Downarrow_G R\ t}\ \{Alt_{S2}\}$$

$$\frac{(e, s_{p_1}s_{p_2}s_r) \Downarrow_G (t_1, s_{p_2}s_r) \quad (e^\star, s_{p_2}s_r) \Downarrow_G (t_2, s_r)}{(e^\star, s_{p_1}s_{p_2}s_r) \Downarrow_G (t_1 \hat{::} t_2, s_r)}\ \{Star_{rec}\} \qquad \frac{(e, s) \Downarrow_G \bot}{(e^\star, s) \Downarrow_G ([\,], s)}\ \{Star_{end}\}$$

$$\frac{(e, s_p\ s_r) \Downarrow_G (t, s_r)}{(!\,e, s_p\ s_r) \Downarrow_G \bot}\ \{Not_F\} \qquad \frac{(e, s) \Downarrow_G \bot}{(!\,e, s) \Downarrow_G (\eta, s)}\ \{Not_S\} \qquad \frac{A \leftarrow e \in R \quad (e, s) \Downarrow_G (t, r)}{(A, s) \Downarrow_G (A@t, r)}\ \{Var\}$$

Figure 3: Parsing expressions operational semantics that produces a tree.

$$\frac{}{G \vdash \hat{\epsilon} : \epsilon}\ \{TEps\} \qquad \frac{}{G \vdash \hat{a} : a}\ \{TChr\}$$

$$\frac{A \leftarrow e \in R(G) \quad G \vdash t : e}{G \vdash A@t : A}\ \{TVar\} \qquad \frac{}{G \vdash \eta : !e}\ \{TNot\}$$

$$\frac{G \vdash t_1 : e_1 \quad G \vdash t_2 : e_2}{G \vdash \langle t_1, t_2\rangle : e_1\ e_2}\ \{TCat\} \qquad \frac{G \vdash t : e_1}{G \vdash L\ t : e_1/e_2}\ \{TLeft\}$$

$$\frac{G \vdash t : e_2}{G \vdash R\ t : e_1/e_2}\ \{TRight\} \qquad \frac{}{G \vdash [\,] : e^\star}\ \{TNil\}$$

$$\frac{G \vdash t_1 : e \quad G \vdash t_2 : e^\star}{G \vdash t_1 \hat{::} t_2 : e^\star}\ \{TCons\}$$

Figure 4: Typing relation for parse trees.

Each constructor of the parse tree syntax has a corresponding typing rule that mirrors the structure of its generating expression. The only non-obvious case is $_{TNot}$: tree η has type $!e$ for *any* e, since the not-predicate never consumes input and carries no sub-tree.

Next, we need to show that our tree producing semantics is equivalent to the original semantics for PEGs. Intuitively, our trees are a structured representation of the prefix parsed by a grammar. We name the textual representation of a parse tree as its flattening which is defined by recursion on the structure of the parse tree (Figure 5).

$$\begin{aligned}
|\hat{\epsilon}| &= \epsilon \\
|\hat{a}| &= a \\
|A@t| &= |t| \\
|\langle t_1, t_2\rangle| &= |t_1|\,|t_2| \\
|L\ t| &= |t| \\
|R\ t| &= |t| \\
|[\,]| &= \epsilon \\
|t_1 \hat{::} t_2| &= |t_1|\,|t_2|
\end{aligned}$$

Figure 5: Flattening of a parse tree.

Flattening collapses a parse tree into the string at its leaves, discarding structural annotations. Using this function we can state the equivalence between the tree-producing semantics and the original PEG semantics.

Theorem 3.1 (Semantics equivalence). *Let G be an arbitrary PEG. If $G \vdash t : e$ then $(e, |t|) \Downarrow_G (t, \epsilon)$ and $(e, |t|) \Rightarrow_G (|t|, \epsilon)$*

Proof. Induction over the derivation of $G \vdash t : e$ using the correspondent semantics rule in each case. □

In the next section, we use parse trees and its semantics to formally define a pattern matching algorithm.

3.2 Parse Tree Patterns And Its Semantics

We start this section presenting pattern syntax and its typing relation. Next, we present a subtyping relation between parsing expression which allow us to describe coercions between patterns to allow both an easy specification of tree patterns and immediate implementation of pattern matching for trees.

3.2.1 Pattern Syntax And Typing Relation. In simple terms, patterns describe the user intended structure for a given input parse tree. Pattern syntax shares the same structure of tree syntax with the addition of *pattern-variables*, which will match any tree of a given type. The syntax of pattern is presented next.

$$p \quad \rightarrow \quad \epsilon \mid a \mid A@p \mid p_1\, p_2 \mid p_1 \,/\, p_2 \mid p^\star \mid !\, p \mid M : e$$

Where ϵ is a pattern that matches with the tree of empty string ($\hat{\epsilon}$), a matches only with the tree of the symbol a ($\hat{a}$), $A@p$ matches with a subtree of type e and $(A, e) \in R$, $p_1\, p_2$ matches if both p_1 and p_2 matches sequentially. $p_1 \,/\, p_2$ matches if one of p_1 or p_2 matches, $p^\star$ will try to match p sequentially as many times as possible, $!p$ matches only if p does not matches. $M : e$ is a pattern variable that matches with any tree $t : e$.

Since patterns are similar to trees, we also define a typing relation for patterns.

$$\frac{}{G \vdash \epsilon : \epsilon}\ \{PEps\} \qquad \frac{}{G \vdash a : a}\ \{PChr\}$$

$$\frac{A \leftarrow e \in R(G) \quad G \vdash p : e}{G \vdash A@p : A}\ \{PVar\} \qquad \frac{G \vdash p : e}{G \vdash !p :!e}\ \{PNot\}$$

$$\frac{G \vdash p_1 : e_1 \quad G \vdash p_2 : e_2}{G \vdash p_1, p_2 : e_1\, e_2}\ \{PCat\} \qquad \frac{G \vdash p_1 : e_1}{G \vdash p_1 : e_1/e_2}\ \{PLeft\}$$

$$\frac{G \vdash p_2 : e_2}{G \vdash p_2 : e_1/e_2}\ \{PRight\} \qquad \frac{G \vdash p : e}{G \vdash p^\star : e^\star}\ \{PStar\}$$

Figure 6: Typing relation for patterns.

3.2.2 Subtyping For Parsing Expressions And Patterns. When specifying patterns over choice expressions, we may want to match against a specific branch of a choice. Instead of specifying a choice pattern that will fail for all other branches besides the intended one, we would like to specify one single pattern that would match only with the intended branch, decreasing the size of the written pattern. Consider the PEG shown in Figure 2. If we want to, e.g., match a multiplication between a parenthesized expressions and a number, we would have to write two choice patterns for rule F: one that accepts a number, but reject a parenthesized expression (e.g. $V_1 : n/\bot_E$) and one the rejects a number, but accepts a parenthesized expression (e.g. $\bot_n/'('V_2 : E')'$). Instead, it would be easier to just specify the patterns that matches against numbers ($V_1 : n$) and against parenthesized expressions ($V_2 : E$) and let the algorithm fill the missing part. To do this, we introduce a subtyping relation and subtype-based coercion both for parsing expressions and patterns.

We let notation $e <: e'$ denote that expression e is a subtype of expression e'. The idea of the subtyping relation is to express that is possible to "coerce" a tree t, such that $G \vdash t : e$, to a tree t', such that $G \vdash t' : e'$, in a way that the coercion preserves the parsed string, i.e., $|t| = |t'|$. Since patterns shares all the structure of parse trees, the coercion reasoning applies to patterns. The main objective of this section is to define, in a precise manner, the subtype-based coercion for patterns.

We start by defining the subtype relation between parsing expressions, presented in Figure 7.

$$\frac{}{e <: e}\ \{Refl\} \qquad \frac{e_1 <: e_2}{e_1 <: e_2/e_3}\ \{Alt_{Left}\}$$

$$\frac{e_1 <: e_3}{e_1 <: e_2/e_3}\ \{Alt_{Right}\} \qquad \frac{n \geq 0 \quad e_1 <: e_2}{e_1^n <: e_2^\star}\ \{Star\}$$

Figure 7: Subtype relation for parsing expressions

The relation is reflexive; either operand of a choice is a subtype of the whole choice; and a fixed-count repetition e_1^n is a subtype of $e_2^\star$ whenever $e_1 <: e_2$ (e^n denotes the n-fold concatenation of e).

Next, we define the syntax of terms to denote subtyping proofs, as follows:

$$d \quad \rightarrow \quad \textbf{refl} \mid \textbf{Inl}\, d \mid \textbf{Inr}\, d \mid \textbf{Rec}\, d$$

Term **refl** denotes a proof for $e <: e$ and **Inl** d denotes a subtyping derivation for $e_1 <: e_2/e_3$, where d represents the subderivation for $e_1 <: e_2$. Same reasoning applies to **Inr** and **Rec** for deductions of $e_1 <: e_2/e_3$ and $e_1^n <: e_2^\star$, respectively.

Finally, we show that the subtyping relation for parsing expressions is decidable.

Theorem 3.2 (Decidability of subtyping). *For any parsing expressions e_1 and e_2, we have that either $e_1 <: e_2$ or that $\neg(e_1 <: e_2)$*

Proof. Induction over the structure of e_1 and case analysis over the structure of e_2 using the definition of the subtyping relation. □

3.2.3 Subtype-based Coercion For Patterns. In this section we define an algorithm for coercing patterns using the subtyping relation between parsing expressions. The algorithm is defined by equations over pattern trees and subtyping terms.

$$\begin{array}{lcl} \textbf{coerce}(p, \textbf{refl}) & = & p \\ \textbf{coerce}(p, \textbf{Inl}\, d) & = & \textbf{coerce}(p, d) \,/\, \bot_p \\ \textbf{coerce}(p, \textbf{Inr}\, d) & = & \bot_p \,/\, \textbf{coerce}(p, d) \\ \textbf{coerce}(p^n, \textbf{Rec}\, d) & = & \textbf{coerce}(p, d)^\star \end{array}$$

Notation $\bot_p$ denotes the patterns which fails for any parse tree and it is equivalent to pattern $!M : e$, where $p : e$. Intuitively, the coercion allow us to match trees for the choice or star operator by just specifying the desired component, for ordered choice, or a specific number of matchings for the star operator. The coerce function satisfies the following property.

Theorem 3.3 (Correctness of coerce). *Let $G = (V, \Sigma, R, e_s)$ be an arbitrary PEG, p a pattern such that $G \vdash p : e$ and that d is a proof term for $e <: e'$, for some e'. Then, $G \vdash \textbf{coerce}(p, d) : e'$.*

Proof. Induction over the structure of $G \vdash p : e$ and case analysis over the derivation of $e <: e'$ using the definitions of **coerce** and the typing relation for patterns. □

Now with the definition of subtyping-based coercion for patterns we are ready to formally define an algorithm for pattern matching parse trees.

3.2.4 A Semantics For Parse Trees Pattern Matching. Before presenting the semantics for pattern matching we need some additional definitions. We name a *match* a sequence between pattern variables and parse trees. We let meta-variable S to represent an arbitrary match. In simple terms, the result of matching a pattern against a tree is a sequence which holds the trees associated with each pattern variable matched. Notation $[M \mapsto t]$ denotes the singleton match between a pattern-variable M with a tree t, $\bullet$ denotes an empty match and $\uplus$ denotes sequence concatenation. We let meta-variable r denote either a match S or a failure, $\bot$.

Matching a pattern p against a tree t is a process defined by an inductively defined judgment $G \vdash p : e; t \rightsquigarrow S$, where G denotes a PEG, p a pattern for e, t a parse tree and S is the result match. The algorithm is presented in Figure 8.

Rule $_{PPVar}$ binds a pattern variable M to any well-typed tree. Rules $_{PLeft}$ and $_{PRight}$ enforce that a choice pattern can only match a tree tagged with the corresponding branch constructor (L or R). Rule $_{PNot}$ always succeeds against η: since η is produced only when e fails during parsing, the parse tree itself witnesses that the not-predicate holds and there is no sub-tree to match further. Rule $_{PStarf}$ iterates p over each element of the list, concatenating all resulting binds.

Example 3.4. Consider the PEG for arithmetic expressions shown in Figure 2, the string $(1 + 2) * 3$ and the pattern $(\mathrm{X} : E) * \mathrm{Y} : F$, where X and Y are pattern variables of type E and F, respectively. Some things to note on the pattern are: 1) The pattern $(\mathrm{X} : E)$ has type F, $(\mathrm{X} : E) : F$, since it matches with the right branch of the choice expression of rule $F \longleftarrow$ n / '(' E ')', by rule $_{PRight}$; and 2) The pattern $*\mathrm{Y} : F$ has type $(*\mathrm{Y} : F)^\star$, by rule $_{PStar}$ where $n = 1$, and matches without problems. Since all components of the pattern are valid, then the whole pattern $(\mathrm{X} : E) * \mathrm{Y} : F$ is also valid and has type T, $\{(\mathrm{X} : E) * \mathrm{Y} : F\} : T$.

By the concatenation rule, the opening parenthesis ('(') matches, the pattern variable X binds to expression 1 + 2, the closing parenthesis (')') and multiplication operator ('*') also matches and finally pattern variable Y binds to 3. The matches found are $[X \mapsto 1 + 2, Y \mapsto 3]$.

Given a match S and a pattern p, we denote by $S[p]$ the parse tree obtained by replacing pattern variables in p by its corresponding tree in S. Now, we state the main property of the matching algorithm.

Theorem 3.5 (Matching correctness). *Let $G = (V, \Sigma, R, e_s)$ be an arbitrary PEG, p a pattern, e a parsing expression and t a parse tree. If $G \vdash p : e$, $G \vdash t : e$ and $G \vdash p : e; t \rightsquigarrow S$ then $S[p] = t$.*

Proof. By induction over the structure of derivation of $G \vdash p : e; t \rightsquigarrow S$. □

While the previous theorem result is important, it is limited: it only applies to trees and patterns that have the same parsing expression type e. Thanks to coercion, we can use the algorithm whenever the types of the pattern and the tree satisfy the subtyping relation.

Corollary 3.6. *Let $G = (V, \Sigma, R, e_s)$ be an arbitrary PEG, p a pattern, e a parsing expression and t a parse tree. If $G \vdash p : e$, $G \vdash t : e'$, $d :: e <: e'$ and $G \vdash \mathbf{coerce}(p, d) : e'; t \rightsquigarrow S$ then $S[p] = t$.*

Proof. Immediate from theorems 3.5 and 3.3. □

3.3 Implementation Details

This section presents implementation details and the tool architecture. Figure 9 presents a diagram of the tool structure.

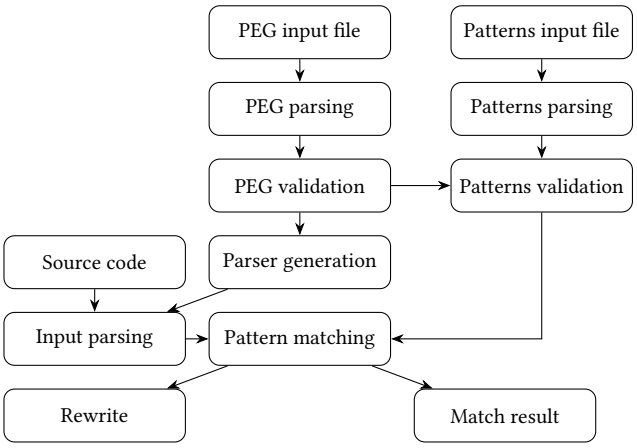

Figure 9: Tool architecture

We define two new operators for PEGs: flatten ($^\wedge e$) and indentation ($e_1 > e_2$). The former flattens the parsed tree in one single node that turns into a terminal (and can be matched as such via patterns) and the latter, acts as the sequence $e_1\ e_2^\star$ with the restriction that $e_2^\star$ must be indented with respect to e_1 and matches as if it was a normal sequence.

During PEG parsing, a sequence (e.g. $e_1\ e_2\ e_3$) is always nested to the right ($e_1\ (e_2\ e_3)$) and the operators &, ? and + are desugared: $\&e$ becomes $!!e$, $e?$ becomes e/ϵ and e^+ becomes $e\ e^\star$. PEG validation checks that the grammar has no left recursion and no parsing expression that accepts the empty string inside a star ($e^\star$), since either condition may cause the parser to loop indefinitely.

Pattern validation includes replacing references to other patterns with the pattern itself, coercing the patterns if necessary and checking if the pattern follows the input PEG's structure. If any pattern makes reference to an undefined pattern, the algorithm returns an error. To make the replacement between patterns, we create a dependency graph between the patterns, topologically sort and replace the references so that no resulting pattern contains references to other patterns and can be treated as a single pattern. This is done so that it is possible to write the patterns in any order, but no set of patterns may have circular dependency.

4 Experiments

This section presents the results obtained so far. To evaluate and demonstrate the capabilities of the tool, we present below some case studies: the generation of a call graph, a suggestion for rewriting the code, and a verification of the presence of specific constructions in the code. All case studies use a PEG that accepts a subset of Python.

$$\frac{G \vdash t_1 : e}{G \vdash M_1 : e; t_1 \rightsquigarrow [M_1 \mapsto t_1]}\ \{PPVar\} \qquad \frac{}{G \vdash \epsilon : \epsilon; \hat{\epsilon} \rightsquigarrow \bullet}\ \{PEps\} \qquad \frac{}{G \vdash a : a; \hat{a} \rightsquigarrow \bullet}\ \{PChr\}$$

$$\frac{a \neq b}{G \vdash a : a; \hat{b} \rightsquigarrow \bot}\ \{PChrf\} \qquad \frac{A \leftarrow e \in R(G) \quad G \vdash p : e; t \rightsquigarrow r}{G \vdash A@p : A; A@t \rightsquigarrow r}\ \{PVar\} \qquad \frac{G \vdash p_1 : e_1; t_1 \rightsquigarrow S_1 \quad G \vdash p_2 : e_2; t_2 \rightsquigarrow S_2}{G \vdash p_1\, p_2 : e_1\, e_2; \langle t_1, t_2 \rangle \rightsquigarrow S_1 \uplus S_2}\ \{PCat\}$$

$$\frac{G \vdash p_1 : e_1; t_1 \rightsquigarrow \bot}{G \vdash p_1\, p_2 : e_1\, e_2; \langle t_1, t_2 \rangle \rightsquigarrow \bot}\ \{PCat1\} \qquad \frac{G \vdash p_1 : e_1; t_1 \rightsquigarrow S_1 \quad G \vdash p_2 : e_2; t_2 \rightsquigarrow \bot}{G \vdash p_1\, p_2 : e_1\, e_2; \langle t_1, t_2 \rangle \rightsquigarrow \bot}\ \{PCat2\} \qquad \frac{G \vdash p_1 : e_1; t \rightsquigarrow r}{G \vdash p_1 \,/\, p_2 : e_1 \,/\, e_2 \,; L\, t \rightsquigarrow r}\ \{PLeft\}$$

$$\frac{G \vdash p_2 : e_2; t \rightsquigarrow r}{G \vdash p_1 \,/\, p_2 : e_1 \,/\, e_2; R\, t \rightsquigarrow r}\ \{PRight\} \qquad \frac{G \vdash p : e; t \rightsquigarrow \bot}{G \vdash !p : !e; \eta \rightsquigarrow \bullet}\ \{PNot\} \qquad \frac{G \vdash p : e; \eta \rightsquigarrow S}{G \vdash !p : !e; \eta \rightsquigarrow \bot}\ \{PNot1\}$$

$$\frac{}{G \vdash p^\star : e^\star; [] \rightsquigarrow \bullet}\ \{PStarn\} \qquad \frac{G \vdash p : e; t_1 \rightsquigarrow S_1 \quad G \vdash p^\star : e^\star; t_2 \rightsquigarrow S_2}{G \vdash p^\star : e^\star; t_1 \hat{:} t_2 \rightsquigarrow S_1 \uplus S_2}\ \{PStarf\}$$

Figure 8: Semantics for pattern matching parse trees.

4.1 Call Graph Generation

In this case study we try to extract a call graph from a given source code and to do so, we will need two different patterns: one that matches with all definitions of functions and one that matches with all functions calls.

```
pattern definition : function_def := ("def" @space
        #name:identifier "(" @space #p:id\list? ")"
        @space ":") #block:statement*
pattern call : function_call := #name:identifier
        @space
        "(" @space #v:expr\list? ")" \epsilon
pattern space : space := " "*
```

The syntax *pattern name* : *type* := *expression* represents a pattern identified by *name* that matches trees with type *type* and *expression* specifies how it will match. *@name* denotes a reference to another pattern and *#var* : *type* denotes a variable that matches with trees of type *type*. ϵ indicates that there must be nothing following the call.

The pattern *definition* matches with any function definition, storing the function identifier, parameters and function body, respectively, in variables *name*, *p* and *block*. The pattern *function_call* matches with any function call, storing the function name and arguments, respectively, in variables *name* and *v*. Then, by first matching the pattern *definition* in the source code and then matching *function_call* in each function's body using what was stored in each match of variable *block*, it is possible to make a list of pairs (*caller*, *calee*) and create a call graph. Consider the following Python code as an example:

```
import math

def delta(a, b, c):
    return b**2 - 4*a*c

def bhaskara(a, b, c):
    d = delta(a, b, c)
    x1 = (-b + math.sqrt(d)) / 2*a
    x2 = (-b - math.sqrt(d)) / 2*a
    return x1, x2

a = float(input("Digite a: "))
b = float(input("Digite b: "))
c = float(input("Digite c: "))

x1, x2 = bhaskara(a, b, c)

print(f"{a}x^2 + {b}x + {c}")
print(f"Raiz 1: {x1}")
print(f"Raiz 2: {x2}")
```

Pattern *definition* will match with functions *delta* and *bhaskara*. In *delta*'s body, pattern *function_call* won't match with any statement, since there are no calls in its body. In *bhaskara*'s body, *function_call* will match the call to *delta* and both calls to *math.sqrt*. This will produce the list [(*bhaskara*, *delta*), (*bhaskara*, *math.sqrt*), (*bhaskara*, *math.sqrt*)] and, after removing the duplicates, it is possible to make the call graph.

4.2 Source Code Validation

These examples were designed in the context of an introductory programming course. This course uses an automatic judge that

evaluates the results of programs submitted by the student, but does not perform a static analysis of the submitted source code. The following examples present two distinct cases that the judge fails to identify, but that could be identified and penalized (or even rejected) due to the use of unauthorized constructs.

4.2.1 Checking For Specific Constructs. Consider a question that asks the student to implement a program that calculates the factorial of an integer *n* provided by the user. The expected solution is for the student to use a loop, such as *while*, to implement the successive multiplication of the numbers.

However, within the Python `math` library, there is the `factorial` function, which, given an integer *n*, returns the result of *n*!. For this reason, some students end up importing the library and using the function, circumventing the objective of the exercise, which is to practice the repetition loop, as shown below.

```
import math
n = int(input("Digite um numero: "))
fatorial = math.factorial(n)
print(f"{n}! = {fatorial}")
```

The pattern presented below is capable of identifying a call to the *factorial* function.

```
pattern fact_call : function_call := (identifier :=
                    "math.factorial") @space "("
                   @space #v2:expr_list? ")" \epsilon
pattern space : space := " "*
```

Where *(identifier := "math.factorial")* means that the name of the function must be *math.factorial*, *#v2:expr_list?* means that the function's argument list will be stored in the variable *v2*. So, if the pattern matches, it means that the student is using the *factorial* function from the *math* library instead of writing the repetition loop, bypassing the original objective of the exercise.

4.2.2 Blocking Disallowed Constructs. Consider an exercise that asks students to read a target vector and a tree-planting matrix using the course-provided functions `inputVector` and `inputMatrix`, then print which states failed to meet their targets. The expected solution uses basic loops and indexing; no built-in higher-order functions or list comprehensions are permitted. One student submitted the following code:

```
def inputVetor():
    entrada = input("Informe as metas
        dos estados: ")
    return list(map(int, entrada.split('
        ,')))

def inputMatriz():
    entrada = input("Informe o plantio
        de arvores: ")
    linhas = entrada.split(';')
    matriz = [list(map(int, linha.split(
        ',')))  for linha in linhas]
    return matriz

def main():
    print("Ministerio do Meio Ambiente")
    metas = inputVetor()
    plantio = inputMatriz()

    num_estados = len(metas)
    totais_plantio = [sum(linha[i] for
        linha in plantio) for i in range
        (num_estados)]

    for i in range(num_estados):
        if totais_plantio[i] < metas[i]:
            print(f"Estado {i+1}, meta =
                {metas[i]}, plantio = {
                totais_plantio[i]}")

if __name__ == "__main__":
    main()
```

Although correct and producing the expected response, it uses Python language resources that ignore the intended learning objectives or were not presented in the course, such as the use of the `map` and `sum` functions, list comprehension and the use of the `__name__` attribute. The Python PEG used in the case studies does not accept constructions such as list comprehension, so this solution would be immediately rejected.

5 Related Work

Several tools address pattern matching and transformation over source code. TAWK [1] extends AWK with language-independent AST patterns and user-defined C actions, but requires a language-specific AST front-end. *Cubix* [4] supports multi-language source-to-source transformations via an incremental parametric syntax (IPS) that separates language-generic from language-specific structure. *Yogo* [6], built on Cubix, performs semantic code search over dataflow graphs using equational reasoning, but does not support rewriting. *RefDiff 2.0* [7] detects refactorings across multiple languages using a language-agnostic Code Structure Tree (CST), but does not expose a user-defined pattern language. *Comby* [8] lifts rewriting to a parsing problem via Parser Parser Combinators (PPC), operating on textual fragments rather than parse trees. *stsearch* [5] uses tree automata to support tree-aware wildcards in syntactic search, but does not provide a formal semantics. LPEG [3] uses PEGs as a pattern formalism, combining their expressive power with the ease of regular expressions, but targets *text* matching rather than parse trees.

Yamaguchi and Kuramitsu [9] propose CPEG, an extension of PEGs with regex-like capture annotations and a type inference algorithm that derives structural constraints on the generated syntax trees. CPEG proves soundness and uniqueness of type inference, offering formal guarantees on tree construction. This is the work most closely related to ours: both use PEGs as the core formalism and both operate on parse trees with a formal type system. The key differences are threefold. First, CPEG's type system characterises the *shape* of trees produced during parsing, whereas ours defines a *matching* relation between a pattern and an existing tree — the goals are complementary. Second, CPEG does not define or

prove a pattern matching algorithm; the capture mechanism extracts data during parsing, not by querying a tree after the fact. Third, our subtyping relation and coercion mechanism allow patterns to be written independently of the exact choice or repetition structure of the grammar, a flexibility that CPEG does not address. Neither CPEG nor any other surveyed tool combines tree-level matching, user-defined PEG-based patterns, a subtyping-driven coercion, rewriting, and formal correctness proofs — which together constitute the contribution of this work.

6 Conclusion

This work presented a formal PEG-based approach to pattern matching on parse trees. We defined a tree-producing semantics for arbitrary PEGs and proved its equivalence to the original PEG semantics. We then introduced a typing relation for parse trees and patterns, a subtyping relation that enables concise pattern specification via coercion, and a pattern matching algorithm whose correctness was formally established. The approach was implemented as a tool that takes a PEG, a set of patterns, and a source file as input, produces a parse tree, and reports the matches found. Case studies showed that the tool can extract call graphs, reject submissions that use unauthorized constructs, and suggest structural rewrites to students.

Future work includes extending the pattern language with binding reuse across patterns, improving error messages, formalizing the library main algorithms in a proof assistant and evaluating the tool in a real classroom setting to assess its pedagogical impact.

Artifact Availability

The Haskell source code is available at the following repository:

https://github.com/lives-group/peg-matching

Instructions for building and reproducing the experiments are available in the repository.

Use of Generative AI

No AI assistance was used in writing this work. All content was generated independently by the authors.

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. The authors also thank the Federal University of Ouro Preto (UFOP) for institutional support.

References

[1] Darren Atkinson and William Griswold. 2006. Effective pattern matching of source code using abstract syntax patterns. *Softw., Pract. Exper.* 36 (04 2006), 413–447. doi:10.1002/spe.704

[2] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.* 39, 1 (Jan. 2004), 111–122. doi:10.1145/982962.964011

[3] Roberto Ierusalimschy. 2009. A text pattern-matching tool based on Parsing Expression Grammars. *Softw. Pract. Exper.* 39, 3 (March 2009), 221–258.

[4] James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One tool, many languages: language-parametric transformation with incremental parametric syntax. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 122 (Oct. 2018), 28 pages. doi:10.1145/3276492

[5] Gabriel Matute, Wode Ni, Titus Barik, Alvin Cheung, and Sarah E. Chasins. 2024. Syntactic Code Search with Sequence-to-Tree Matching: Supporting Syntactic Search with Incomplete Code Fragments. *Proc. ACM Program. Lang.* 8, PLDI, Article 230 (June 2024), 22 pages. doi:10.1145/3656460

[6] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1066–1082. doi:10.1145/3385412.3386001

[7] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: A Multi-Language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 47, 12 (Dec 2021), 2786–2802. doi:10.1109/TSE.2020.2968072

[8] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight multi-language syntax transformation with parser parser combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 363–378. doi:10.1145/3314221.3314589

[9] Daisuke Yamaguchi and Kimio Kuramitsu. 2019. CPEG: A Typed Tree Construction from Parsing Expression Grammars with Regex-Like Captures. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3297280.3297433