Pest control: A formal model of the Pest parser generator

Guilherme Daher
guilherme.daher@aluno.ufop.edu.br
Prog. de Pós-Graduação em Ciência da Computação
Universidade Federal de Ouro Preto
Ouro Preto, Minas Gerais, Brazil

Elton Cardoso
eltonmc@ufop.edu.br
Prog. de Pós-Graduação em Ciência da Computação
Universidade Federal de Ouro Preto
Ouro Preto, Minas Gerais, Brazil

Leonardo Reis
lvsreis@ice.ufjf.br
Departamento de Ciência da Computação
Universidade Federal de Juiz de Fora
Juiz de Fora, Minas Gerais, Brazil

Rodrigo Ribeiro
rodrigo.ribeiro@ufop.edu.br
Prog. de Pós-Graduação em Ciência da Computação
Universidade Federal de Ouro Preto
Ouro Preto, Minas Gerais, Brazil

ABSTRACT

Parsing expressions grammars (PEGs) are a recognition-based formalism for parsers, which has been gaining popularity because they avoid some problems present in context-free grammar based specifications. In the context of the Rust programming language, the most used PEG-based parser generator is Pest. An interesting feature of Pest grammars is that they support new operators for repetition and handling an implicit stack during parsing. In this work, we propose a formalization of these new constructs and extend a type inference algorithm which guarantee that all well-typed Pest-style grammars do not loop on inputs.

KEYWORDS

Parsing, Parsing Expression Grammars, Type systems

1 Introduction

Parsing is a core problem in computer science. In simple terms, a parsing algorithm, also named a parser, checks if a string of symbols conforms to some set of rules specified by formalisms like context-free grammars (CFGs) or regular expressions (REs). Usually, parsers construct data structures showing what rules were used to conclude that the input string was obtainable from the specification or an error message indicating that such string is unobtainable from the specification. Due to its importance, parsing is an extensively studied subject, as evidenced by Grune and Jacob's book "Parsing Techniques: A Practical Guide", which has more than 600 pages and covers only classic algorithms [10]. However, parsing is far from being a solved problem: in recent years, we have seen a growing interest in the development of new algorithms and formalisms [1, 9, 11, 16, 22] or the verification of well-known parsing techniques [4, 7, 8, 14].

In the early 2000's, Brian Ford proposed *Parsing Expression Grammars* (PEGs), a recognition-based formalism to define languages [9]. The novelty of PEGs is that they specify how to check whether a string is in the language being defined and not how it is generated by its specification. Formally a PEG is a 4-tuple (V_N, V_T, R, e_S) where V_N and V_T are as in CFGs, R is a function which maps a non terminal from V_N to a *parsing expression* and e_S is the initial parsing expression. A parsing expression is very similar to the right-hand side of a CFG in the extended Backus-Naur form with the addition of new operators such as the not-predicate (!) and the prioritized choice (/). A fundamental difference between PEG and CFGs is the choice operator, which impose priority between alternatives. It means that β in a prioritized choice expression α/β is tested only in case of failure on α. The not-predicate operator checks if the string matches some syntax without consuming the input.

In the context of the Rust programming language [12], the Pest library [21] is the most popular PEG-based parser generator for Rust, having more than 114.000 downloads registered at crates.io package registry. Besides supporting all traditional PEG operators, Pest support operators to manipulate an implicit stack during parser execution. The inclusion of a stack allows Pest parsers to elegantly express identation sensitive languages by controlling identation levels using the stack. While such new operators eases the task of writing grammars for complex formats, additional care must be taken in order to ensure PEG *well-formedness*. We say that a PEG is well-formed if it does not loop on any input strings [9]. Pest users have reported that the tool fails to detect grammars that loop on some inputs [20]. In this work, we propose a formalization of a generalized version of these Pest operators using operational semantics and extend a type system for PEGs [19] and its inference algorithm [5] to deal with these operators.

Specifically, we make the following contributions:

- We formalize both stack and the new repetitions Pest operators using operational semantics and extend a type system for PEGs [19] to deal with these new operations. The proposed semantics is realized by a intrisincally-typed definitional interpreter [3] using

the Agda programming language [17]. While the semantics is formalized in Agda, we present and describe it using LaTeX notation. We choose this presentation style for brevity and to not obsfucate the semantics core ideas with syntax details of the Agda language. The interested reader can check the formalization source code on-line [2].
- We include operations to query the top-most element of the stack which can be used to handle context-sensitive formats, like length indexed fields, common in binary data formats;
- We show how to extend a type inference algorithm [5] for PEGs to deal with Pest new operators. A prototype implementation of the inference algorithm was developed using the Racket programming language [6] and used to check some example grammars.

The rest of this paper is organized as follows. Section 2 presents an informal description of Pest operators. Section 3 reviews the syntax, semantics and a type system for PEGs. In Section 4 defines the syntax, semantics, the type system and its inference algorithm for the new PEG operators. Section 5 presents some use examples of grammars accepted by our tool. Related work is discussed on Section 6 and Section 7 concludes.

2 An overview of Pest new operators

Before starting the description of each new operator, we describe *repetition indexes*, which are values ranged over natural numbers and a special value, **infty**, which denotes an upper-bound over indexes[1]. Notation $\circ$ denotes either addition or multiplication operations over natural numbers. The constants **top.tonat** and **top.length** refer to the conversion of the stack top element to a numeric value and its size, respectively. More details on how these operations work will be described on the semantics.

The first operator we consider is the iterated repetition, denoted e^i in the abstract syntax and `e^i` in the concrete syntax. It describes that expression e should be repeated i times over the input, failing if any of these iterations results in a parser failure. As an example of the usefulness of this operator, consider the task of representing a PEG for a format formed by a two bytes values followed by a single bit terminator. We could express this simple format as the following grammar using the iterated repetition:

```
bit <-- '0' / '1'
byte <-- bit^8
start: byte^2 ~ bit
```

Notation `'0'` denotes an expression for a string, operator `~` denotes concatenation (following Pest concrete syntax), `A <-- e` denotes a grammar rule and `start:` specifies the starting expression for this grammar. The usage of the iterated repetition operator allows for concise description of pattern "2 bytes" as simply `byte^2`, which indicates that non terminal `byte` should be repeated two times over the input.

The Pest tool also introduces an interval repetition operator, $e[i, j]$, which executes the expression e a number of times between the indexes i and j, inclusive. As an example of this operator usage, consider the task of parsing either 2,3 or 4 bits sequences from input. This could be expressed elegantly by the following expression using the interval repetition: `('0' / '1')[2,4]`.

However, not all combinations of indexes are allowed in repetition operators. An example of an invalid repetition operator: `e[infty, 2]`, since a bounded repetition, `e[i,j]`, is valid if $i < j$ and i is not the constant `infty`. Regarding the exact repetition operator, e^i, we do not allow to use the index **infty**, since its semantics is to parse the input by repeating an expression e **exactly** i times, which makes the expression $e^{\textbf{infty}}$ to be repeat indefinitely over the input.

Now, we turn our attention to the stack manipulation operators. Expression `push(e)` parses the input using the expression `e` and pushes the consumed input on the parser stack. Operator `pop` removes the current stack top-most element and tries to match it on the current input, `peek` also matches the input without removing the element from the stack and `drop` removes the first element of current stack, without matching it. The operator `peekall` tries to match the input using the complete stack content, without empty the stack. Finally, `dropall` empties the stack.

One interesting example use of stack operators is to parse length-indexed fields present in several data formats. The next grammar shows how to express netstrings, which is a format method for strings which contains a field to denote the size of the represented data.

```
number <-- '0' / '1' / ... / '9'
letter <-- 'a' / ... / 'z'
char <-- number / letter

start: push(number+) ~ ':' ~ char^top.tonat ~ ','
```

The starting expression begins by pushing the input size value on the parser stack, and then access this size information using `top.tonat` to parse the exact quantity of input based on the size information that starts the netstring. Operators `top.tonat` and `top.length` are not present in the current definition of Pest parser generator. We include both since they have a direct formalization and increases the expressivity for Pest grammars which can be used to represent length-indexed fields, which are notoriously difficult to parse correctly [16, 22].

3 Background

An Overview of PEGs. Intuitively, PEGs are a formalism for describing top-down parsers. Formally, a PEG is a

[1] For any index $i \neq$ **infty**, we have that $i <$ **infty** and $i \circ$ **infty** $= \infty$

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. We comment on some rules of the semantics. Rule $_{Eps}$ specifies that expression ϵ will not fail on any input s by leaving it unchanged. Rule $_{ChrS}$ specifies that an expression a consumes the first character when the input string starts with an 'a' and rule $_{ChrF}$ shows that it fails when the input starts with a different symbol. Rule $_{Var}$ parses the input using the expression associated with the variable in the grammar G. When parsing a sequence expression, $e_1\, e_2$, the result is formed by e_1 and e_2 parsed prefixes and the remaining input is given by the result e_2. Rules $_{Cat_{F1}}$ and $_{Cat_{F2}}$ say that if e_1 or e_2 fail, then the whole expression fails. The rules for choice impose that we only try expression e_2 in e_1/e_2 when e_1 fails. Parsing a star expression $e^\star$ consists in repeatedly execute e on the input string. When e fails, $e^\star$ succeeds without consuming any symbol of the input string. Finally, the rules for the not-predicate expression, $!\, e$, specify that whenever the expression e succeeds on input s, $!\, e$ fails; and when e fails on s we have that $!\, e$ succeeds without consuming any input.

A type system for PEGs. A key issue in PEG formalism is to ensure the termination of the parsing algorithm for arbitrary inputs. Ford [9] proposed a well-formedness predicate which is computed as fixed-point over the set of grammar's sub-expressions. In simple terms, a PEG can loop indefinitely over an input string if: 1) it has direct or indirect left-recursive rules ; 2) expressions that succeed without consuming any symbol under the Kleene star operator. As pointed by [19], one inconvenience of Ford's well-formedness predicate is that it is not compositional. Based on this observation, Ribeiro et al. [19] propose a type system that guarantees termination by restricting valid expressions to those that do not obey conditions 1 and 2. In order to ensure these restrictions, types are interpreted as records containing two fields: a boolean value indicating that the current expression may succeed without consuming anything from the input string and a set of variables that can appear as the first symbol on the recognition branch from the current parsing expression. This set (named head-set) is defined by recursion on the parsing expression syntax as follows:

$$\begin{array}{lcll}
head(\epsilon) & = & \emptyset & \\
head(a) & = & \emptyset & \\
head(A) & = & \{A\} \cup head(R(A)) & \\
head(e_1\, e_2) & = & \begin{cases} head(e_1) \cup head(e_2) & \text{if } e_1 \rightharpoonup 0 \\ head(e_1) & \text{otherwise} \end{cases} & \\
head(e_1 / e_2) & = & head(e_1) \cup head(e_2) & \\
head(e^\star) & = & head(e) & \\
head(!\, e) & = & head(e) &
\end{array}$$

Basically, the head-set of ϵ and $a \in \Sigma$ is the empty set. For a non terminal, the head-set is formed by the non terminal itself plus the head-set of its associated parsing expression. The head-set of a sequence expression $e_1\, e_2$ need to include the head-set of e_2, whenever e_1 succeeds without consuming anything ($e_1 \rightharpoonup 0$). Otherwise, $head(e_1\, e_2)$ is just $head(e_1)$. The set of non terminals for the choice operator is the union of the head-set of its operands. Finally, the head-set for the Kleene star and not operator are just the set for its underlying expressions. Following Ribeiro et. al., τ denotes an arbitrary type, $\tau.null$ denotes the boolean field of τ, and $\tau.head$ the set of variables that can appear as the first symbol of τ's parsing expression. We let notation $\langle b, S\rangle$ denote a type τ formed by a boolean b and a set S. In the type system, Ribeiro et. al use the following operations:

$$\begin{array}{l}
b \Rightarrow S = \text{if } b \text{ then } S \text{ else } \emptyset \\
\tau_1 \otimes \tau_2 = \langle \tau_1.null \wedge \tau_2.null, \tau_1.head \cup (\tau_1.null \Rightarrow \tau_2.head)\rangle \\
\tau_1 \oplus \tau_2 = \langle \tau_1.null \vee \tau_2.null, \tau_1.head \cup \tau_2.head\rangle \\
!\, \tau = \langle true, \tau.head\rangle \\
\langle false, S\rangle^\star = \langle true, S\rangle
\end{array}$$

The first operation is a boolean test that returns the set S whenever the condition is true. The product operation $(\tau_1 \otimes \tau_2)$ combines two types by taking the conjunction of their boolean flags and the union of their respective head-sets in case $\tau_1.null$ is true. The co-product operation $(\tau_1 \oplus \tau_2)$ on types is similar to the product, but instead of using conjunction it takes the disjunction of their boolean fields. The not operation on types just change its boolean field to true and keep its head-set. The last operation is the Kleene star on types, which also sets its boolean field to true. However, the Kleene is only defined for types τ such that $\tau.null = false$, being undefined when $\tau.null = true$. Finally, following the common practice, the meta-variable Γ denotes a typing context and notation $\Gamma(A) = \tau$ holds whenever $(A, \tau) \in \Gamma$. The type system is defined as an inductive judgment $\Gamma \vdash e : \tau$, where Γ is a typing context, e is a parsing expression and

$$\frac{}{(\epsilon, s) \Rightarrow_G (\epsilon, s)}\ \{Eps\} \qquad \frac{}{(a, as_r) \Rightarrow_G (a, s_r)}\ \{ChrS\} \qquad \frac{a \neq b}{(a, bs_r) \Rightarrow_G \bot}\ \{ChrF\} \qquad \frac{A \leftarrow e \in R \quad (e, s) \Rightarrow_G r}{(A, s) \Rightarrow_G r}\ \{Var\}$$

$$\frac{(e_1, s_{p_1}s_{p_2}s_r) \Rightarrow_G (s_{p_1}, s_{p_2}s_r) \quad (e_2, s_{p_2}s_r) \Rightarrow_G (s_{p_2}, s_r)}{(e_1\, e_2, s_{p_1}s_{p_2}s_r) \Rightarrow_G (s_{p_1}s_{p_2}, s_r)}\ \{Cat_{S1}\} \qquad \frac{(e_1, s_p s_r) \Rightarrow_G (s_p, s_r) \quad (e_2, s_r) \Rightarrow_G \bot}{(e_1\, e_2, s_p s_r) \Rightarrow_G \bot}\ \{Cat_{F2}\}$$

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

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

$$\frac{(e, s_p\, s_r) \Rightarrow_G (s_p, s_r)}{(!\, e, s_p\, s_r) \Rightarrow_G \bot}\ \{Not_F\} \qquad \frac{(e, s) \Rightarrow_G \bot}{(!\, e, s) \Rightarrow_G (\epsilon, s)}\ \{Not_S\} \qquad \frac{}{(a, \epsilon) \Rightarrow_G \bot}\ \{ChrNil\}$$

Figure 1. Parsing expressions operational semantics.

τ is a type (Figure 2). A PEG $G = (V, \Sigma, R, e_S)$ is considered well-typed, written $\Gamma \vdash G$, if $\forall A \in V.R(A) : \Gamma(A)$.

$$\frac{}{\Gamma \vdash \epsilon : \langle true, \emptyset \rangle} \qquad \frac{}{\Gamma \vdash a : \langle false, \emptyset \rangle}$$

$$\frac{\Gamma(A) = \tau \quad A \notin \tau.head}{\Gamma \vdash A : \langle \tau.null, \tau.head \cup A \rangle}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \,/\, e_2 : \tau_1 \oplus \tau_2} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash\, ! e : !\, \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1 \otimes \tau_2} \qquad \frac{\Gamma \vdash e : \tau \quad \tau.null = false}{\Gamma \vdash e^\star : \tau^\star}$$

Figure 2. Type system definition.

The first rule of the type system specifies that a parsing expression ϵ has type $\langle true, \emptyset \rangle$. The rule of a single character expression says that its type is $\langle false, \emptyset \rangle$, since it does not succeed without consuming a symbol and no variable can appear as the head of its recognition branch, i.e., its head-set is $\emptyset$. The type of a prioritized choice expression (e_1/e_2) has type $\tau_1 \oplus \tau_2$, where $\Gamma \vdash e_1 : \tau_1$ and $\Gamma \vdash e_2 : \tau_2$. The type of a sequence expression is given by the product operation on their sub-expressions types, and the star operation is only well-typed if its underlying expression does not succeed without consuming a symbol. A not-predicate parsing expression always has the type $\langle true, \tau.head \rangle$ whenever its sub-expression is well-typed. Finally, a variable is well-typed only if its type is in the context and itself does not belong to its head-set. Ribeiro et al. [19] type system readily rejects invalid expressions like $a(!b/c)^\star$, since it only considers well-typed a Kleene parsing expressions that its underlying expression does not succeed without consuming the input. However, an inconvenience of it is the need to require type annotations on all grammar's non terminals. A later work [5] proposes a type inference algorithm, which guarantee termination of the parsing process.

4 Formalizing Pest operators

New operators syntax. The Pest tool includes two types of operators: stack and repetitions operators. The syntax of these operators are defined next:

$$\begin{array}{lcl} e & \rightarrow & \ldots \mid e^i \mid e[i,i] \mid \textbf{push} \mid \textbf{pop} \mid \textbf{peek} \mid \textbf{drop} \\ & \mid & \textbf{peekall} \mid \textbf{dropall} \\ i & \rightarrow & \textbf{top.tonat} \mid \textbf{top.length} \mid i \circ i \mid v \\ v & \rightarrow & \textbf{infty} \mid n, m \in \mathbb{N} \\ \sigma & \rightarrow & s : \sigma \mid [] \\ \circ & \rightarrow & + \mid \times \end{array}$$

Meta-variable s denotes an arbitrary string using grammar's alphabet. We let meta-variable σ denote the parser stack and we represent it using Haskell-style list notation.

Semantics of new operators. Now that we have introduced the syntax and the informal semantics of the new Pest operators, we turn our attention to its formal semantics. One important aspect of Pest repetition operators is the usage of indexes. Before evaluating any of these new operators we need to reduce indexes to a normal form. We consider that an index is in normal form if it is a number or the constant **infty**. We let meta-variable v denote an arbitrary index normal form. In simple terms, the index normalization algorithm performs the following steps:

1. Evaluate **top.length** and **top.tonat**, if they are present in the current index. The usage of these operators can result in a run-time error if the parser stack is empty

or the stack first element can not be converted to a number (for **top**.**tonat**).

2. Reduce all arithmetic operations using their usual meaning and the following reduction rules when the constant **infty** is present:

$$n \circ \textbf{infty} \rightarrow \textbf{infty}$$
$$\textbf{infty} \circ n \rightarrow \textbf{infty}$$
$$\textbf{infty} \circ \textbf{infty} \rightarrow \textbf{infty}$$

where $\circ$ denotes either addition or multiplication and n a numeric value.

The semantics is denoted by the inductive judgment (Figures 3, 4 and 5) $(e, w, \sigma) \Rightarrow_G r$, where e is the current expression being executed, w is the current input string, σ is the current input parser stack, G is the grammar and r is the output which can be: 1) success, denoted by triple (w_1, w_2, σ') where w_1 is the parsed prefix, w_2 is the remaining input and σ' is the resulting stack from executing the expression e, or; 2) failure, denoted by $\perp$. The inclusion a stack requires changing the semantics of all standard PEG operators to thread the current stack state through parser execution. We omit these modified rules, since they don't add any interesting insight.

Figure 3 presents the semantics for the stack manipulation operators. Rules $PUSH_S$ and $PUSH_F$ formalize the behaviour of the push operator, which pushes the prefix parsed by the push's expression argument into the parser stack or it fails when the expression fails. The semantics of the **pop** operator is defined by rules POP_S and POP_F, which removes the topmost element of the stack and matching it against the input string. When such a matching is not possible, we have a failure. The only difference between the semantics for the **peek** operation and **pop** is that the former keeps the top element in the result stack. The **drop** operator removes the top element of the stack without matching it against the current input or it fails when the stack is empty (check rules $DROP_S$ and $DROP_F$). Rules $PEEKALL_S$ and $PEEKALL_F$ uses the whole stack content to match the current input, failing if all the stack elements do not form a prefix of the current input and $DROPALL_S$ simply empties the parser stack.

The semantics for the exact repetition operator is presented in Figure 4. The first rule, EX_{S1}, shows that for any expression e, e^0 consumes the empty string from the input and does not change the parser stack. The second rule, EX_{S2}, shows that to evaluate e^i, we first normalize the index i to obtain a numeric value, n. Next, we check if $0 < n$ and parse the input using e producing a remaining suffix w_2, which is used as input for running the expression e^{n-1}. We say that the repetition operator e^i fails if any iteration of e results in a failure, as formalized by rules EX_{F1} and EX_{F2}.

The last set of semantics rules deals with interval repetition operator, $e[i, j]$. Rule $Inter_1$ parses the input using e^n, when both of indexes normalizes to the same numeric value n. The rule $Inter_2$ shows how to run an interval repetition when the normalized indexes are such that $n < m$, where $\textbf{norm}(i, \sigma) = n$ and $\textbf{norm}(j, \sigma) = m$: it tries to parse the input using e^m, if it succeeds then result is returned. In case of failure of e^m, rule $Inter_3$ applies we try to parse the input using the expression $e[n, m-1]$. Rule $Inter_{F1}$ specifies that an interval repetition fails when the formalized value of i is greater than normalized value of j. The rule deal with the situation of the upper limit bound normalizes to **infty**. In this situation, expression $e[i, j]$ is interpreted as $e^n e^*$, where n is the result of normalizing the index i.

4.1 Type system and its inference algorithm

Now, we turn our attention to extending a type system to avoid non termination problems during a grammar execution. We follow the approach of defining typing rules and its corresponding type inference using the strategy of constraint generation followed by a solving step [5, 18, 19].

Before starting discussing the typing rules for the new operators, we need to extend the *head* function. In Figure 6, we present the new equations which handle the both repetions and stack PEG construction.

$$\begin{array}{lcl} head(e^i) & = & head(e) \\ head(e[i, j]) & = & head(e) \\ head(\textbf{push}(e)) & = & head(e) \\ head(\textbf{pop}) & = & \emptyset \\ head(\textbf{peek}) & = & \emptyset \\ head(\textbf{drop}) & = & \emptyset \\ head(\textbf{dropall}) & = & \emptyset \\ head(\textbf{peekall}) & = & \emptyset \end{array}$$

Figure 6. Head-set equations for the new operators

The first three equations shows that the head-set of repetitions and the push operator are equal to their parameter expression set. All the remaining stack operators have their head-set equal to $\emptyset$, since they do not refer to non terminals.

The following subsections present the extensions needed to support typing the new operators.

4.1.1 Typing rules for the new operators.

We start by defining the typing rules for our new PEG repetition operators. Figure 7 presents the typing rules for the new repetition operators. Typing rules make use of the following notations: 1) $\textbf{norm}^*(i)$ denotes a static analysis version of the normalization algorithm in which operators **top.tonat** and **top.length** are replaced by 0[2]; 2) we let τ^v denote the type $\langle v = 0 \vee \tau.null, \tau.head \rangle$ and 3) notation $\tau[v_1, v_2]$ denote the type $\langle v_1 = 0 \vee \tau.null, \tau.head \rangle$.

The first type system rule shows that an exact repetition is considered nullable if either its underlying expression is

[2]This is necessary because these operators value depends on the current parser stack state, which is unknown during type inference.

$$\frac{(e, w, \sigma) \Rightarrow_G (w_1, w_2, \sigma')}{(\textbf{push}\ e, w, \sigma) \Rightarrow_G (w_1, w_2, (w_1 : \sigma'))}\ \{PUSH_S\} \qquad \frac{\exists w_2.w = w_1 w_2}{(\textbf{pop}, w, w_1 : \sigma) \Rightarrow_G (w_1, w_2, \sigma)}\ \{POP_S\} \qquad \frac{}{(\textbf{drop}, w, []) \Rightarrow_G \bot}\ \{DROP_F\}$$

$$\frac{\neg\exists w_1\, w_2\, \sigma'.w = w_1 w_2 \wedge \sigma = w_1 : \sigma'}{(\textbf{pop}, w, \sigma) \Rightarrow_G \bot}\ \{POP_F\} \qquad \frac{\neg\exists w_1\, w_2\, \sigma'.w = w_1 w_2 \wedge \sigma = w_1 : \sigma'}{(\textbf{peek}, w, \sigma) \Rightarrow_G \bot}\ \{PEEK_F\}$$

$$\frac{\exists w_2.w = w_1 w_2}{(\textbf{peek}, w, w_1 : \sigma) \Rightarrow_G (w_1, w_2, w_1 : \sigma)}\ \{PEEK_S\} \qquad \frac{\sigma = [w_0, ..., w_n] \quad w' = w_0 ... w_n \quad \exists w''.w = w' w''}{(\textbf{peekall}, w, \sigma) \Rightarrow_G (w', w'', \sigma)}\ \{PEEKALL_S\}$$

$$\frac{}{(\textbf{drop}, w, w_1 : \sigma) \Rightarrow_G (\epsilon, w, \sigma)}\ \{DROP_S\} \qquad \frac{\sigma = [w_1, ..., w_n] \quad w' = w_1 ... w_n \quad \neg\exists w''.w = w' w''}{(\textbf{peekall}, w, \sigma) \Rightarrow_G \bot}\ \{PEEKALL_F\}$$

$$\frac{(e, w, \sigma) \Rightarrow_G \bot}{(\textbf{push}(e), w, \sigma) \Rightarrow_G \bot}\ \{PUSH_F\} \qquad \frac{}{(\textbf{dropall}, w, []) \Rightarrow_G \bot}\ \{DROPALL_F\}$$

$$\frac{}{(\textbf{dropall}, w, w_1 : \sigma) \Rightarrow_G (\epsilon, w, [])}\ \{DROPALL_S\}$$

Figure 3. Semantics of the stack operators

$$\frac{\textbf{norm}(i, \sigma) = 0}{(e^i, w, \sigma) \Rightarrow_G (\epsilon, w, \sigma)}\ \{EX_{S1}\} \qquad \frac{\textbf{norm}(i, \sigma) = n \quad 0 < n \quad (e, w, \sigma) \Rightarrow_G (w_1, w_2, \sigma_1) \quad (e^{n-1}, w_2, \sigma_1) \Rightarrow_G (w_1', w_2', \sigma')}{(e^i, w, \sigma) \Rightarrow_G (w_1 w_1', w_2', \sigma')}\ \{EX_{S2}\}$$

$$\frac{\textbf{norm}(i, \sigma) = n \quad 0 < n \quad (e, w, \sigma) \Rightarrow_G \bot}{(e^i, w, \sigma) \Rightarrow_G \bot}\ \{EX_{F1}\} \qquad \frac{\textbf{norm}(i, \sigma) = n \quad 0 < n \quad (e, w, \sigma) \Rightarrow_G (w_1, w_2, \sigma_1) \quad (e^{n-1}, w_2, \sigma_1) \Rightarrow_G \bot}{(e^i, w, \sigma) \Rightarrow_G \bot}\ \{EX_{F2}\}$$

Figure 4. Semantics for the exact repetition operator

$$\frac{\textbf{norm}(i, \sigma) = n \quad \textbf{norm}(j, \sigma) = n \quad (e^n, w, \sigma) \Rightarrow r}{(e[i, j], w, \sigma) \Rightarrow_G r}\ \{Inter_1\} \qquad \frac{\textbf{norm}(i, \sigma) = n \quad \textbf{norm}(j, \sigma) = m \quad n < m \quad (e^m, w, \sigma) \Rightarrow_G (w_1, w_2, \sigma')}{(e[i, j], w, \sigma) \Rightarrow_G (w_1, w_2, \sigma')}\ \{Inter_2\} \qquad \frac{\textbf{norm}(i, \sigma) = n \quad \textbf{norm}(j, \sigma) = m \quad n > m}{(e[i, j], w, \sigma) \Rightarrow_G \bot}\ \{Inter_{F1}\}$$

$$\frac{\textbf{norm}(i, \sigma) = n \quad \textbf{norm}(j, \sigma) = m \quad n < m \quad (e^m, w, \sigma) \Rightarrow_G \bot \quad (e[n, m-1], w, \sigma) \Rightarrow_G r}{(e[i, j], w, \sigma) \Rightarrow_G r}\ \{Inter_3\}$$

$$\frac{\textbf{norm}(i, \sigma) = n \quad \textbf{norm}(j, \sigma) = \textbf{infty} \quad (e^n\, e^\star, w, \sigma) \Rightarrow_G r}{(e[i, j], w, \sigma) \Rightarrow_G r}\ \{Inter_I\}$$

Figure 5. Semantics for the interval repetition operator

nullable or its normalized index is equal to either 0. The head-set for an exact repetition is the same of its parameter expression. We also restrict that normalized indexes can only be numeric values, n. Typing of the interval repetition operator is similar to exact repetition, but we allow the upper interval bound to be **infty**. Intervals are considered nullable when its parameter expression is nullable or either the normalized left index, n, is equal to zero. We also do not allow intervals that have an **infty** upper bound to be nullable, since such parsing expression will diverge whenever they succeed without consuming any input.

$$\frac{\mathbf{norm}^*(i) = n \quad \Gamma \vdash e : \tau}{\Gamma \vdash e^i : \tau^n}$$

$$\frac{\Gamma \vdash e : \tau \quad \mathbf{norm}^*(i) = n \quad \mathbf{norm}^*(j) = v \quad \neg(v \equiv \mathbf{infty} \wedge \tau.null)}{\Gamma \vdash e[i,j] : \tau[n,v]}$$

Figure 7. Typing rules for new PEG repetition operators

Figure 8 presents the typing rules for the PEG stack operators. The first rule specifies that the type for **push**(e) is the same as e, since it simply pushes the e's consumed input and its head-set will be the same as $e's$ head-set. Since all other stack operators can succeed without consuming input, all of them are considered nullable. As presented in Figure 6, the head-set of **pop**, **drop**, **peek**, **peekall** and **dropall** are the empty set.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{push}(e) : \tau} \qquad \frac{}{\Gamma \vdash \mathbf{pop} : \langle true, \emptyset \rangle}$$

$$\frac{}{\Gamma \vdash \mathbf{peek} : \langle true, \emptyset \rangle} \qquad \frac{}{\Gamma \vdash \mathbf{peekall} : \langle true, \emptyset \rangle}$$

$$\frac{}{\Gamma \vdash \mathbf{drop} : \langle true, \emptyset \rangle} \qquad \frac{}{\Gamma \vdash \mathbf{dropall} : \langle true, \emptyset \rangle}$$

Figure 8. Typing rules for new PEG stack operators

4.2 Extending the type inference algorithm

In this section we describe the necessary extensions to the type inference algorithm created by Cardoso et. al. [5] to support the new repetitions and stack operators.

Extending the constraint syntax. We start by including index normal forms on the constraint grammar (taken from [5]). The non terminal v denotes a normal form of an index expression, which can be a numeric value or constant **infty**. The complete constraint grammar is as follows:

$$\begin{array}{lcl} \tau & \rightarrow & \alpha \mid \langle b, S \rangle \\ v & \rightarrow & n \mid \mathbf{infty} \\ \mathbf{t} & \rightarrow & A \mid b \mid S \mid \tau \mid v \\ C & \rightarrow & \mathbf{true} \mid \mathbf{false} \mid \mathbf{t} \equiv \mathbf{t} \mid C \wedge C \mid \exists \alpha.C \\ & \mid & \mathbf{def}\ A : \tau\ \mathbf{in}\ C \end{array}$$

Constraints are first-order logic formulas over types and its meaning can be defined inductively. We omit the constraint semantics, since its definition is the same presented in [5].

Extending constraint generation and solving. After including index expressions in the constraint syntax, we need to include equations for the new operations in the constraint generation algorithm. In order to generate constraints for the exact repetition operator, we use an existential quantifier to define a new variable, α, that is passed as argument to the generation of constraints for e. The algorithm creates an equality between the argument type, τ, and the repetition operator type over α. We also include an inequality to assert that the normalized index should not be **infty**.

$$\langle\langle e^i : \tau \rangle\rangle = \exists \alpha. \langle\langle e, \alpha \rangle\rangle \wedge \tau \equiv \alpha^{\mathbf{norm}^*(i)} \wedge \mathbf{norm}^*(i) \not\equiv \mathbf{infty}$$

Next, we consider the generation of constraints for the interval repetition operator, which follows the same pattern of the exact repetition: we generate the constraint for subexpression e, considering an existential quantified type variable and create an equality between the variable type and the input argument type, τ. We also impose that the interval lower-bound cannot be **infty** and that if the upper-bound is **infty**, the interval underlying expression cannot be nullable since it could make the parser loop indefinitely.

$$\begin{array}{rcl} \langle\langle e[i,j] : \tau \rangle\rangle = & \exists \alpha. \langle\langle e, \alpha \rangle\rangle \wedge \tau \equiv \alpha[\mathbf{norm}^*(i), \mathbf{norm}^*(j)] \\ \wedge & \mathbf{norm}^*(i) \not\equiv \mathbf{infty} \\ \wedge & \neg(\mathbf{norm}^*(j) \equiv \mathbf{infty} \wedge \alpha.null) \end{array}$$

The constraint for the **push** operator is just the constraint of its argument expression and all other stack expressions just generate an equality between its input argument type and $\langle true, \emptyset \rangle$.

$$\begin{array}{lcl} \langle\langle \mathbf{push}(e) : \tau \rangle\rangle & = & \langle\langle e : \tau \rangle\rangle \\ \langle\langle \mathbf{pop} : \tau \rangle\rangle & = & \tau \equiv \langle true, \emptyset \rangle \\ \langle\langle \mathbf{drop} : \tau \rangle\rangle & = & \tau \equiv \langle true, \emptyset \rangle \\ \langle\langle \mathbf{dropall} : \tau \rangle\rangle & = & \tau \equiv \langle true, \emptyset \rangle \\ \langle\langle \mathbf{peek} : \tau \rangle\rangle & = & \tau \equiv \langle true, \emptyset \rangle \\ \langle\langle \mathbf{peekall} : \tau \rangle\rangle & = & \tau \equiv \langle true, \emptyset \rangle \end{array}$$

Figure 9. Extending the constraint generation algorithm

The inclusion of index expressions do not change the constraint solver specification [5]. Normalized indexes are just integers or the **infty** and they only appear as part of the nullability field of a type, which has its dedicated equation in solver specification, or as part of an inequality, which can be easily solved by an SMT solver or a simple custom solving algorithm.

Since the proposed changes (the inclusion of indexes and new operators) do not change the constraint solver, just the constraint generation algorithm, we argue that Cardoso et al. [5] soundness and completeness argument for the type inference algorithm also holds for our Pest-style grammars. This lead to our main result:

Theorem 1 (Termination of typed grammars)**.** *Let $G = (V, \Sigma, R, e_s)$ be a well-typed grammar which may use any of Pest repetition or stack operators. Then, for all w, exists r such that $(e_s, w, []) \Rightarrow r$.*

The only way of a parsing expression to loop over an input is due to the presence of left-recusive rules or nullable expressions under a star operator in a grammar [9]. Since the type system (and its inference algorithm) reject any grammar that have these issues, any well-typed grammar is guaranteed to terminate.

5 Examples

We implemented a Racket library, called *typed-peg*, which implements the type inference and the semantics of grammars with the proposed new operators. In this section, we present a complete example grammar and discuss how our approach avoid the looping behavior reported by Pest users [20].

A grammar for the PNG format. A popular image file format is the Portable Network Graphics (PNG), which is represented by the following pieces of data: 1) A format signature (8 bytes), which always corresponds to the following integer values: 137, 80, 78, 71, 13, 10, 26, 10; 2) A 4 bytes natural number n, which represents the data length; 3) 4 bytes chunk type value; 4) The image data, a sequence of n bytes and 5) a 4 bytes CRC checksum. A possible grammar to specify the PNG format is:

```
#lang typed-peg
bit <-- '0' / '1'
byte <-- bit^8
fist-four <-- '137' ~ '80' ~ '78' ~ '71'
snd-four <-- '13' ~ '10' ~ '26' ~ '10'
signature <-- first-four ~ snd-four
length <-- push(byte^4)
type <-- byte^4
data <-- byte^top.tonat
crc <-- byte^4
start: signature ~ length ~ type ~ data ~ crc
```

Notice that the use of an exact repetition operator combined with index **top.tonat** allow us to guarantee that the data block has the correct size of the format length field.

Rejecting reported looping examples. Now, let's turn our attention to some expressions which Pest users reported as non terminating.

```
forever1 <-- (push('') ~ pop)*
forever2 <-- popall ~ (popall)*
forever3 <-- forever3
```

In the first two examples, we have nullable expressions under a star operator. In the first rule, we have a repetition of **push** operator, which has an empty string expression as its argument, followed by **pop**. A **push** is nullable only if its argument is nullable. Since it has an empty string as argument, the expression `push('')` accepts the empty string. Next, we have a pop operator which is nullable, by the definition. In this way, the type system rejects `forever1`, since it has a nullable expression under a star. The second example is also ill-typed since it has a **popall**, a nullable operator, as argument to a PEG repetition. The third example shows a simple left recursive rule which is also rejected by type inference algorithm.

6 Related work

The works of Ribeiro et al. [19] and Cardoso et. al. [5] proposed a type system and a type inference algorithm for PEGs that is sound with respect to Ford's well-formedness predicate. Authors argue that the use of a type system provide a more predictable and compositional behavior. Another work that uses types to ensure parsing termination was proposed by Krishnaswami et. al. [13]. The authors define a type system for μ-regular expressions, an algebraic presentation of context-free languages, and proved that their type system only accepts LL(1) grammars. Our work extend both the type system for PEGs and its inference algorithm to Pest operators while having the guarantee that no well-typed grammar will loop over inputs.

The development of formalisms and tools for correct parsing of complex data formats is subject of some recent works. Zhang et. al. [22] proposed interval parsing grammars (IPG) as formalism for express some context-sensitive formats present in binary data like video and image formats. An IPG attaches to every non terminal/terminal an interval, which specifies the range of input consumed. By connecting intervals and attributes, the context-sensitive patterns in file formats can be well handled. Unlike *typed-peg*, IPGs allow the use semantic actions to perform arbitrary computations over attributes. While such approach provide extra power, in our view, it clutter since the parser specification is not formed only by grammar rules but also program pieces to handle parsed data.

7 Conclusion

In this work we proposed a formalization of an extended version of Pest style PEG grammars. Besides a formal definition of the operator semantics, we extended a typing inference algorithm for PEGs [5] which guarantee termination of well-typed grammars to handle Pest operators. As future work, we intend to continue develop of the *typed-peg* Racket library and to integrate the ideas present in this work in future versions of the Pest tool.

ACKNOWLEDGMENTS

This work is supported by the Fundação de Amparo à Pesquisa de Minas Gerais under grant number APQ-01683-21.

REFERENCES

[1] Andrea Asperti, Claudio Sacerdoti Coen, and Enrico Tassi. 2010. Regular Expressions, au point. *CoRR* abs/1010.2604 (2010). arXiv:1010.2604 http://arxiv.org/abs/1010.2604

[2] Anonymous authors. [n. d.]. SECRET Library - online repository. https://anonymous.4open.science/r/pest-sblp2025.

[3] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2017. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158104

[4] Jean-Philippe Bernardy and Patrik Jansson. 2016. Certified Context-Free Parsing: A formalisation of Valiant's Algorithm in Agda. *Logical Methods in Computer Science* 12 (2016). Issue 2. https://doi.org/10.2168/LMCS-12(2:6)2016

[5] Elton M. Cardoso, Regina Sarah Monferrari Amorim De Paula, Daniel Freitas Pereira, Leonardo Vieira dos Santos Reis, and Rodrigo Geraldo Ribeiro. 2023. Type-based Termination Analysis for Parsing Expression Grammars. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, SAC 2023, Tallinn, Estonia, March 27-31, 2023*, Jiman Hong, Maart Lanperne, Juw Won Park, Tomás Cerný, and Hossain Shahriar (Eds.). ACM, 1372–1379. https://doi.org/10.1145/3555776.3577620

[6] Ryan Culpepper, Matthias Felleisen, Matthew Flatt, and Shriram Krishnamurthi. 2019. From Macros to DSLs: The Evolution of Racket. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs, Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:19. https://doi.org/10.4230/LIPICS.SNAPL.2019.5

[7] Denis Firsov and Tarmo Uustalu. 2013. Certified Parsing of Regular Languages. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings (Lecture Notes in Computer Science, Vol. 8307)*, Georges Gonthier and Michael Norrish (Eds.). Springer, 98–113. https://doi.org/10.1007/978-3-319-03545-1_7

[8] Denis Firsov and Tarmo Uustalu. 2014. Certified CYK parsing of context-free languages. *J. Log. Algebraic Methods Program.* 83, 5-6 (2014), 459–468. https://doi.org/10.1016/j.jlamp.2014.09.002

[9] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 111–122. https://doi.org/10.1145/964001.964011

[10] Dick Grune and Ceriel J.H. Jacobs. 1990. *Parsing Techniques - A Practical Guide*. Ellis Horwood, Chichester, England. 300+ pages. http://www.cs.vu.nl/~dick/PTAPG.html PDF online free.

[11] Trevor Jim, Yitzhak Mandelbaum, and David Walker. 2010. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 417–430. https://doi.org/10.1145/1706299.1706347

[12] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA.

[13] Neelakantan R. Krishnaswami and Jeremy Yallop. 2019. A Typed, Algebraic Approach to Parsing. 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, 379–393. https://doi.org/10.1145/3314221.3314625

[14] Raul Lopes, Rodrigo Geraldo Ribeiro, and Carlos Camarão. 2016. Certified Derivative-Based Parsing of Regular Expressions. In *Programming Languages - 20th Brazilian Symposium, SBLP 2016, Maringá, Brazil, September 22-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9889)*, Fernando Castor and Yu David Liu (Eds.). Springer, 95–109. https://doi.org/10.1007/978-3-319-45279-1_7

[16] Stefan Lucks, Norina Marie Grosch, and Joshua König. 2017. Taming the Length Field in Binary Data: Calc-Regular Languages. In *2017 IEEE Security and Privacy Workshops (SPW)*. 66–79. https://doi.org/10.1109/SPW.2017.33

[17] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming* (Heijen, The Netherlands) *(AFP'08)*. Springer-Verlag, Berlin, Heidelberg, 230–266.

[18] François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489.

[19] Rodrigo Ribeiro, Leonardo V. S. Reis, Samuel Feitosa, and Elton M. Cardoso. 2019. Towards Typed Semantics for Parsing Expression Grammars. In *Proceedings of the XXIII Brazilian Symposium on Programming Languages* (Salvador, Brazil) *(SBLP 2019)*. Association for Computing Machinery, New York, NY, USA, 70–77. https://doi.org/10.1145/3355378.3355388

[20] Tartasprint. [n. d.]. On the hunt of grammars making the parser run indefinitely.

[21] Pest Development Team. [n. d.]. Pest - online repository. https://github.com/pest-parser/pest.

[22] Jialun Zhang, Greg Morrisett, and Gang Tan. 2023. Interval Parsing Grammars for File Format Parsing. *Proc. ACM Program. Lang.* 7, PLDI, Article 150 (jun 2023), 23 pages. https://doi.org/10.1145/3591264