A Semantic Framework for PEGs

Sérgio Queiroz de Medeiros
Carlos Olarte[*]
sergiomedeiros@ect.ufrn.br
carlos.olarte@gmail.com
ECT, Federal University of Rio Grande do Norte
Natal, Brazil

Abstract

Parsing Expression Grammars (PEGs) are a recognition-based formalism which allows to describe the syntactical and the lexical elements of a language. The main difference between Context-Free Grammars (CFGs) and PEGs relies on the interpretation of the choice operator: while the CFGs' unordered choice $e \mid e'$ is interpreted as the union of the languages recognized by e and e', the PEGs' prioritized choice e/e' discards e' if e succeeds. Such subtle, but important difference, changes the language recognized and yields more efficient parsing algorithms. This paper proposes a rewriting logic semantics for PEGs. We start with a rewrite theory giving meaning to the usual constructs in PEGs. Later, we show that cuts, a mechanism for controlling backtracks in PEGs, finds also a natural representation in our framework. We generalize such mechanism, allowing for both local and global cuts with a precise, unified and formal semantics. Hence, our work strives at better understanding and controlling backtracks in parsers for PEGs. The semantics we propose is executable and, besides being a parser with modest efficiency, it can be used as a playground to test different optimization ideas. More importantly, it is a mathematical tool that can be used for different analyses.

CCS Concepts: • **Theory of computation → Grammars and context-free languages**; **Rewrite systems**; • **Software and its engineering → Syntax**; **Parsers**.

Keywords: parsing expression grammars, rewriting logic

ACM Reference Format:
Sérgio Queiroz de Medeiros and Carlos Olarte. 2020. A Semantic Framework for PEGs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering (SLE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3426425.3426944

[*]Carlos Olarte was funded by CNPq.

1 Introduction

Parsing Expression Grammars (PEGs) [8] are the core of several widely used parsing tools. Visually, the description of a PEG is similar to the description of a Context-Free Grammar (CFG). Unlike CFGs, PEGs have a deterministic ordered choice operator, which allows a limited form of backtracking. This makes PEGs a suitable formalism for representing deterministic Context-Free Languages, i.e., the LR(k) languages. Another key difference between both formalisms is the presence of syntactic predicates, that allow PEGs to describe the lexical elements of a language. PEGs were conceived as a formalism to recognize strings, while CFGs are commonly used to generate strings. Hence, it may not be trivial to determine the language described by a PEG.

Although PEGs ordered choice operator avoids ambiguities when writing a grammar, it also poses some difficulties. To correctly recognize a language, the user of a PEG-based tool needs to be careful about the ordering of the alternatives in a choice $e_1 \ / \ e_2$, as e_2 will never match a string x when e_1 matches a prefix of x. Regarding performance, PEGs local backtracking may still impose a performance drawback. Thus, when possible, it is desirable to avoid the backtracking associated with a choice.

In this paper we study the problem of backtracking in PEGs through the lens of a formal approach based on the rewriting logic (RL) [17] framework. RL can be seen as a flexible and general model of computation where important properties of the modeled system can be specified and proved. We thus provide both: a formal foundation for better understanding and controlling backtracks in PEGs; and tools that can help the user to check whether her grammar complies with the intended meaning or not. As an interesting side effect, we show that our specification is not only a (correct by construction) recognition-based algorithm, but also a derivative parser, able to generate strings from a grammar.

Plan and contributions. After recalling PEGs in §2, we start in §3 with a rewrite theory modeling the natural semantics rules for PEGs originally proposed in [15]. Hence, we obtain a formal model of the derivability relation in PEGs.

Due to the ϵ representational distance typical of RL specifications, the model and the actual system are very close, thus making it easier to reason about grammars in our framework.

Inspired by a small-step semantics approach, §3.2 proposes an alternative (and equivalent) rewrite theory that eliminates the (unnecessary) non-deterministic steps of the first one. Both theories are executable in Maude [4], an efficient rewriting engine. We compare the two theories and show that the second one is more amenable for automatic verification techniques. Hence, besides being a formal tool for reasoning about PEGs, the proposed specification is actually a *correct by construction* parser with modest performance.

Cut operators [20] have been introduced in PEGs to reduce the number of backtracks and improve efficiency. However, the semantics of these operators has remained informal in the literature. Using RL, §4 gives a precise meaning to cuts. The proposed semantics makes evident why less memory is needed when cut annotations are added to a grammar. More importantly, such formal account allows us to generalize the concept of cuts from *local* to *global* cuts in §4.2. In some cases, global cuts may save more computations than the cuts proposed in [20]. In §4.3 and §4.4 we show that local and global cuts can coexist coherently with a clear and uniform semantics. §4.5 reports some benchmarks on grammars annotated with cuts.

The machinery developed here can be leveraged to perform other analyses in PEGs. We explore one of such analyses in §B, where we report on a preliminary attempt to use our specification to not only recognize a given string but, symbolically, produce all the possible strings (up to a bounded length) from a grammar. This kind of analyses can be useful to highlight unexpected behaviors when a PEG is designed, and also to explore optimization ideas.

§5 discusses related work and §6 concludes the paper. The companion appendix contains detailed proofs of the main results. All the rewrite theories proposed here and the benchmarks reported in §4.5 can be found (and reproduced) with the Maude and script files available at the public repository https://github.com/carlosolarte/RESPEG.

2 Parsing Expression Grammars

From now on, we shall write PEG to refer to the following definition of parsing expression grammars.

Definition 2.1 (Syntax)**.** A PEG G is a tuple (V, T, P, e_ι), where V is a finite set of non-terminals, T is a finite set of terminals, P is a total function from non-terminals to parsing expressions and e_ι is the initial parsing expression. We shall use A,B,C to range over elements in V and a, b, c to range over elements in T. Parsing expressions, ranged over by e, e_1, e_2, etc., are built from the syntax:

$$e := \epsilon \mid A \mid a \mid e_1\, e_2 \mid e_1/e_2 \mid e* \mid !e$$

We assume that V is partitioned into two (disjoint) sets V_{Lex} and V_{Syn}, where V_{Lex} is the set of non-terminals that match lexical elements, also known as tokens, and V_{Syn} represents the non-terminals that match syntactical elements.

The empty parsing expression is ϵ. The expression $e_1\ e_2$ stands for the (sequential) composition of e_1 and e_2. The expression e_1/e_2 denotes an ordered choice. The repetition of e is written as $e*$. The look-ahead or negative predicate is written as $!e$. A predicate $!e$ tests if the expression e matches the input, without consuming it. Predicates are handy to describe lexical elements and to act as guards in choice alternatives.

The function P maps non-terminals into parsing expressions. $P(A)$ denotes the expression associated to A. Alternatively, P can be seen as a set of rules of the form $A \leftarrow e$.

Strings are built from terminal symbols and ϵ denotes the empty string. We shall use x, y, w to range over (possibly empty) strings and xy denotes the concatenation of x and y.

Semantics. A parsing expression e, when applied to an input string x, either succeeds or fails. When the matching of e succeeds, it consumes a prefix of the input. Such prefix can be the empty string ϵ (and nothing is consumed).

We define the states of a PEG parser as follows.

Definition 2.2 (States)**.** Parsing states are built from

$$S ::= G[e]\ x \mid \texttt{fail} \mid x$$

In $G[e]\ x$, the expression e in the context of the PEG G is matched against the string x. The expression `fail` represents a failed attempt of matching[1]. The state x represents the successful matching of an expression returning the suffix x.

Definition 2.3 (Semantics)**.** The reduction relation $\overset{\text{PEG}}{\rightsquigarrow}$ is the least binary relation on parsing states satisfying the rules in Fig. 1. The language recognized by e in the context of G is the set $\mathcal{L}(e, G) = \{x \mid G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} y\}$. Two parsing expressions are equivalent (in the context of G), notation $e \equiv e'$, if $\mathcal{L}(e, G) = \mathcal{L}(e', G)$.

Let us dwell upon the rules in Fig. 1. If $G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} y$, the expression e consumes a (possibly empty) prefix of x and returns the remaining suffix y. For instance, rule **term.1** consumes the non-terminal a in ax and returns x. If $G[e]\ x \overset{\text{PEG}}{\rightsquigarrow}$ `fail`, e fails to match the input x (see **term.2** and **term.3**).

In $G[e_1\ e_2]\ x$, either e_1 fails to match x and the whole expression fails (**seq.2**) or e_1 succeeds on x and the final result depends on the outcome of matching e_2 against the remaining suffix y (**seq.1**).

When the ordered choice e_1/e_2 is applied to x, if e_1 matches x, then the alternative e_2 is discarded (**ord.1**). (Note the difference w.r.t. CFGs). On the other side, if e_1 fails, a backtrack

[1]Failing states can also take the form (`fail`, y) where y is the suffix of the input that could not be recognized. For the sake of presentation, we shall ignore here the suffix y.

$$\frac{}{G[\varepsilon]\ x \overset{\text{PEG}}{\rightsquigarrow} x}\ (\text{empty}) \qquad \frac{G[P(A)]\ x \overset{\text{PEG}}{\rightsquigarrow} S}{G[A]\ x \overset{\text{PEG}}{\rightsquigarrow} S}\ (\text{var})$$

$$\frac{}{G[a]\ ax \overset{\text{PEG}}{\rightsquigarrow} x}\ (\text{term.1}) \qquad \frac{b \neq a}{G[b]\ ax \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\text{term.2})$$

$$\frac{}{G[a]\ \varepsilon \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\text{term.3})$$

$$\frac{G[e_1]\ x \overset{\text{PEG}}{\rightsquigarrow} y \quad G[e_2]\ y \overset{\text{PEG}}{\rightsquigarrow} S}{G[e_1\ e_2]\ x \overset{\text{PEG}}{\rightsquigarrow} S}\ (\text{seq.1}) \qquad \frac{G[e_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[e_1\ e_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\text{seq.2})$$

$$\frac{G[e_1]\ x \overset{\text{PEG}}{\rightsquigarrow} y}{G[e_1\ /\ e_2]\ x \overset{\text{PEG}}{\rightsquigarrow} y}\ (\text{ord.1}) \qquad \frac{G[e_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail} \quad G[e_2]\ x \overset{\text{PEG}}{\rightsquigarrow} S}{G[e_1\ /\ e_2]\ x \overset{\text{PEG}}{\rightsquigarrow} S}\ (\text{ord.2})$$

$$\frac{G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[!e]\ x \overset{\text{PEG}}{\rightsquigarrow} x}\ (\text{not.1}) \qquad \frac{G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} y}{G[!e]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}\ (\text{not.2})$$

$$\frac{G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[e*]\ x \overset{\text{PEG}}{\rightsquigarrow} x}\ (\text{rep.1}) \qquad \frac{G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} y \quad G[e*]\ y \overset{\text{PEG}}{\rightsquigarrow} z}{G[e*]\ x \overset{\text{PEG}}{\rightsquigarrow} z}\ (\text{rep.2})$$

Figure 1. Semantics of PEGs. S is a parsing state denoting either `fail` or a string x (Definition 2.2).

is performed and e_2 is applied to the string x, regardless of whether e_1 consumed part of it or not (**ord.2**).

The look-ahead operator $!e$ (or negative predicate) fails when e succeeds (**not.2**) and succeeds (not consuming any input) when e fails (**not.1**). Note that the expression $!e$ does not consume any prefix of the input.

The repetition $e*$ greedily matches e against the input. If e fails on x, $e*$ does not consume any input (**rep.1**). Rule **rep.2** specifies the recursive case where $e*$ continues matching the suffix y. Note that this operator is different from the usual one in regular expressions: $G[a*]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ iff x is a (possibly empty) string containing only the terminal symbol a and y starts with $b \neq a$. This operator can be in fact derived from the others: let e be a parsing expression and A_e be a distinguished non-terminal symbol. Then, $e*$ is equivalent to A_e where $A_e \leftarrow e/\epsilon$ [8]. We shall keep this operator in the syntax since it greatly simplifies some examples.

Clearly, if $G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} y$ then y is a suffix of x. In order to guarantee termination on $\overset{\text{PEG}}{\rightsquigarrow}$, a well-formedness condition on grammars is assumed [8]: there are no left-recursive rules such as $A \leftarrow A\ e_1/e_2$. Moreover, there are no expressions of the form $e*$ where e succeeds on some input x not consuming any prefix of it ([25]).

Unlike CFGs, PEGs are *deterministic* [8]. This means that $\overset{\text{PEG}}{\rightsquigarrow}$ is a function.

Without loss of generality, our analyses consider only PEGs that satisfy the *unique token prefix* condition [6]. Roughly, tokens of the grammar are described by non-terminals $A \in V_{Lex}$. Hence, at most one non-terminal $A \in V_{Lex}$ matches a prefix of the current input. This is the typical behavior of parsing tools that have a separate lexer (e.g., yacc gets tokens from lex for a given input).

Definition 2.4 (Unique token prefix)**.** Let $G = (V, T, P, e_\iota)$, $V = V_{Lex} \cup V_{Syn}$, $A, B \in V_{Lex}$ s.t. $A \neq B$, $a \in T$ and xy be a string. G has the unique token prefix property iff $G[A]\ axy \overset{\text{PEG}}{\rightsquigarrow} y$ implies that $G[B]\ axy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$.

3 A Rewriting Logic Semantics for PEGs

This section proposes an executable rewriting logic semantics for PEGs that we later use to control backtracks (§4) and as a basis for a derivative parser (§B).

Rewriting Logic (RL) [17] is a general model of computation where proof systems, semantics of programming languages and, in general, transition systems can be specified and verified. RL can be seen as a logic of change that can naturally deal with states and concurrent computations. The reader can find a detailed survey of RL in [18] and [7].

In the following, we briefly introduce the main concepts of RL needed to understand this paper, and, at the same time, we introduce step by step the proposed semantics for PEGs. For the sake of readability, we shall adopt in most of the cases the notation of Maude [4], a high-level language that supports membership equational logic and rewriting logic specifications. Thanks to its efficient rewriting engine and its metalanguage capabilities, Maude turns out to be an excellent tool for creating executable environments of various logics and models of computation. This will make our specification *executable*.

A *rewrite theory* is a tuple $\mathcal{R} = (\Sigma, E \uplus B, R)$. The static behavior of the system is modeled by the order-sorted equational theory $(\Sigma, E \uplus B)$ and the dynamic behavior by the set of rewrite rules R. These components are explained below.

Equational theory. The signature Σ defines a set of typed operators used to build the terms of the language (i.e, the syntax of the modeled system). E is a set of equations over T_Σ (the set of terms built from Σ) of the form $t = t'$ if ϕ. The equations specify the algebraic identities that terms of the language must satisfy (e.g., $|\epsilon| = 0$ and $|ax| = 1 + |x|$ where $|x|$ denotes the length of x). B is a set of structural axioms over T_Σ for which there is a finitary matching algorithm. Such axioms include associativity, commutativity, and identity, or combinations of them. For instance, ϵ is the identity for concatenation and then, modulo this axiom, the term $x\epsilon$ is equivalent to x. The equational theory associated to $\mathcal{R}$ thus defines deterministic and finite computations as in a functional programming language.

Our semantics for PEGs starts by defining an appropriate equational theory or functional module –`fmod`– in Maude's terminology. Here we define the set of terminal and non-terminal symbols and strings:

```
fmod PEG-SYNTAX is
```

```
 sort NTSymbol . --- Non-terminal symbols
 sort TSymbol .  --- Terminal symbols
 sort Str .      --- Strings
 sorts TChar TExp . --- Chars and character classes
  ... --- to be completed/explained below
endfm
```

First, sorts (or types) are declared. As we shall see, `TChar` is the basic block for building terms of type `Str` (by juxtaposition/concatenation) and `TExp` will be populated with the usual patterns such as `[0-9]`, `[a-z]`, etc.

As we said before, the equational theory is ordered-sorted. This means that besides having many sorts, there is a partial order on sorts defining a sub-typing relation:

```
 subsorts String < TChar  < Str .   --- Sub-typing
 subsort TChar TExp < TSymbol .
 subsort Qid < NTSymbol .
```

The sort `String` is part of the standard library of Maude and it represents the usual strings in programming languages. Hence, "(" is a Maude's `String` and also a `TChar` in this specification (being `String` the *least sort*). Note that terms of type `TChar` and `TExp` are also terminal symbols due to the subsort relation. Hence, examples of valid terminal symbols are `";"`, `"c"`, `"3"`, `[0-9]`, etc. A Qid is a *qualified identifier* of the form `'X` (using an apostrophe at the beginning of the expression). Examples of non-terminal symbols are `'Begin`, `'Statement`, etc. In order to make the presentation cleaner, in the forthcoming examples, we shall omit the apostrophe and write simply `Begin` instead of `'Begin`.

Besides the sorts, the signature also specifies the functional symbols or operators that define the syntax of the model. Here some examples of constructors for the type `TExp`:

```
 op [.] : -> TExp . --- Any character
 ops [0-9] [a-z] [A-Z] ... : -> TExp .
```

All these operators are constants (functions without parameters) of type `TExp`. For the sort `Str`, we have:

```
 op eps : -> Str . --- empty string
  --- Strings (concatenated with whitespace)
 op __ : TChar Str -> Str [right id: eps ] .
```

`eps` is the defined constant to denote the empty string ϵ. In Maude, "_" denotes the position of an argument. The operator `op __` (usually called empty syntax in Maude) receives two parameters and returns a `Str`. Note that `eps` is the right identity (an *axiom* associated to `op __`) for this operator (i.e., $x\epsilon \equiv x$). As an example, the term `"(" "3" "+" "2" ")"` is an inhabitant of `Str`. For the sake of readability, when no confusion arises, we shall omit quotes on characters and simply write `( 3 + 2 )`.

Deterministic and terminating computations are specified via equations. In fact, an equational theory is executable only if it is terminating, confluent and sort-decreasing [4]. Under these conditions, the mathematical meaning of the equality $t \equiv t'$ coincides with the following strategy: reduce t and t' to their unique (due to termination and confluence) normal forms t_c and t'_c using the equations of the theory as *simplification rules* from left to right. Then, $t \equiv t'$ iff $t_c =_B t'_c$ (note that $=_B$, equality modulo B, is decidable since it is assumed a finitary matching algorithm for B).

As an example, the following specification checks whether a terminal symbol matches a `TChar`:

```
 op match : TSymbol TChar -> Bool .
 vars tc tc' : TChar .  --- logical variables
 eq match(tc, tc')     =   tc == tc' .
 eq match([.], tc)     =   true .
 eq match([0-9], tc)   =   tc >= "0" and-then tc <= "9" .
 ...
```

The `==` symbol is built-in Maude (in this case, checking whether two `Strings` are equal). Variables in equations are implicitly universally quantified and the second equation must be read as

$$(\forall \mathtt{tc} : \mathtt{TChar}).(\mathtt{match}([.], \mathtt{tc}) = \mathtt{true})$$

An equation then rewrites its left hand side into the right hand side *simplifying* terms. For instance,

the term `match([.], "a" )` reduces into `true`

The syntax of parsing expressions is defined as follows:

```
sort Exp . --- Parsing expressions
op emp : -> Exp . --- the empty expression
--- Terminal and non-terminal symbols are expressions
subsort TSymbol NTSymbol < Exp .
op _._ : Exp Exp -> Exp . --- Sequence
op _/_ : Exp Exp -> Exp . --- Ordered choice
op _* : Exp -> Exp .      --- Repetition
op !_ : Exp -> Exp .      --- Negative predicate
```

Note that sequential composition is represented as `e.e'`. For the sake of readability, we shall omit the "." and simply write `e e'` when no confusion arises.

It is also possible to specify derived constructors by defining the appropriate equations. For instance, the *and predicate* &e can be defined as !!e [8]:

```
op &_ : Exp -> Exp .      --- derived operator
eq & e = ! ! e .          --- equation given meaning to & e
```

Hence, &e attempts to match e and, if it succeeds, then it backtracks to the starting point (not consuming any part of the input). More examples of derived constructors are

```
op _? : Exp -> Exp .  --- zero or one
op _+ : Exp -> Exp .  --- one or more
eq e ? = e / emp .
eq e + = e . e * .
```

Production rules ($A \leftarrow p$), grammars (as sets of production rules) and parsing states are defined as follows:

```
sorts  Rule  Grammar State .
subsort Rule < Grammar . --- Every rule is a grammar
op _<-_ : NTSymbol Exp -> Rule .
op nil : -> Grammar . --- empty set of rules
--- Concatenating rules
op _,_ : Grammar Grammar->Grammar [comm assoc id:nil] .
--- Parsing states
op _[_]_ : Grammar Exp Str -> State . --- G[e]x
op fail : -> State .  --- fail
subsort Str < State . --- x is also a state
```

Note the axioms imposed on the operator for "concatenating" rules: the order of the rules (comm for commutativity) as well as parentheses (assoc for associativity) are irrelevant. Moreover, nil can be removed/added at will.

Thanks to the almost zero representational distance [18] between the above specification and the syntax of PEGs, we can see the system and its specification as isomorphic structures (with slightly different notations). From now on, it should be clear from the context that G in the expression "$G[e]\ x$" is a grammar while G in "G[e] x" is a term of sort Grammar. As noticed, we have consistently used the monospace font for objects in the specification.

This concludes the specification of the equational theory defining the syntax and basic operations on PEGs. The next step is to define rules encoding the semantics for PEG's constructors.

3.1 A First Rewrite Theory

Rewriting rules. The last component R in the rewrite theory $\mathcal{R} = (\Sigma, E \uplus B, R)$ is a finite set of rewriting rules of the form $t \longrightarrow t'$ ifϕ. A rule defines a state transformation and R models the dynamic behavior of the system (which is not necessarily deterministic, nor terminating). In Maude, rewrite theories are defined in modules:

```
mod PEG-SEMANTICS is
 pr PEG-SYNTAX . --- Importing PEG-SYNTAX
 var x : Str .
 var G : Grammar .
 rl [empty] : G[ emp ] x => x .
 ...
endm
```

"empty" is the name of the rule and variables are implicitly quantified. Hence, the above rule must be interpreted as:

$$(\forall \mathtt{G} : \mathtt{Grammar}, \mathtt{x} : \mathtt{Str}).(\mathtt{G}[\mathtt{emp}]\ \mathtt{x} \Rightarrow \mathtt{x})$$

This rule reflects the behavior of the rule **empty** in Fig. 1. The rules for terminal symbols are:

```
rl [Terminal12] :  G[t] tc x =>
   if match(t,tc)  then x else fail fi .
rl [Terminal3] :  G[t] eps  => fail .
```

The first rule encodes both **term.1** and **term.2** where the outcome depends on whether t matches tc. The second rule encodes the behavior of **term.3**.

The other rules are conditional rules (crl instead of rl) where the transition happens only if the condition after the symbol if holds:

```
crl [NTerminal] : ( G, N <- e ) [N] x => S
     if  (G, N <- e) [e] x => S .
crl [Seq1] :  G  [e . e'] x => S
     if  G[ e ] x  =>  y     /\
         G[ e' ] y  => S .
crl [Seq2] : G[e . e'] x => fail    if G[ e ] x  => fail .

crl [Choice1] :  G[ e / e'] x => y  if G[ e ] x  =>  y .
crl [Choice2] :  G[e / e'] x => S
     if G[ e ] x  => fail /\
        G[ e' ] x  =>  S .

crl [Star1] : G[ e * ] x => x if G[ e ] x  => fail .
crl [Star2] :  G[ e * ] x  => z
     if G[ e ] x  => y /\
        G[e *] y  => z .

crl [Neg1] : G  [ ! e ] x => x if G[ e ] x  => fail .
crl [Neg2] : G  [ ! e ] x => fail if G[ e ] x  => y .
```

In the case of [NTerminal], due to the axioms of the operator _,_ (i.e., [comm assoc id:nil]), the order of the rules is irrelevant. Hence, if the current expression is N (N is a variable of type NTerminal), this rule will unfold the production rule $N \longleftarrow e$ and try to match e with the current input.

In Seq1, it must be the case that G[e] x reduces to y and the configuration G[e'] y must reduce to the state S. This reflects exactly the behavior of the rule **seq.1**. The other rules can be explained similarly.

A rewrite theory $\mathcal{R}$ induces a rewrite relation $\overset{\mathcal{R}}{\longrightarrow}$ on $T_\Sigma(\mathcal{X})$ (the set of terms build from Σ and the countably set of variables $\mathcal{X}$) defined for every $t, u \in T_\Sigma(\mathcal{X})$ by $t \overset{\mathcal{R}}{\longrightarrow} u$ if and only if there is a rule $(l \longrightarrow r \textbf{ if } \phi) \in R$ and a substitution $\theta : \mathcal{X} \longrightarrow T_\Sigma(\mathcal{X})$ satisfying $t =_{E \uplus B} l\theta$, $u =_{E \uplus B} r\theta$, and $\phi\theta$ is (equationally) provable from $E \uplus B$ [2]. In words, t matches module $E \uplus B$ the left hand side of the rewrite rule under a suitable substitution and then evolves into the right hand side of the rule if the condition $\phi\theta$ holds. The relation $\overset{\mathcal{R}}{\longrightarrow}*$ is the reflexive and transitive closure of $\overset{\mathcal{R}}{\longrightarrow}$. Moreover, we shall use $t \overset{\mathcal{R}}{\longrightarrow}! t'$ to denote that $t \overset{\mathcal{R}}{\longrightarrow}* t'$ and $t' \overset{\mathcal{R}}{\not\longrightarrow}$, i.e., t reduces in zero or more steps to t' and it cannot be further reduced (t' is called a *normal form*).

We shall use $\overset{\text{PEG}}{\Longrightarrow}$ to denote the induced rewrite relation from the above described theory. As a simple example, note that G["a"."b"] "a" "b" $\overset{\text{PEG}}{\Longrightarrow}!$ eps. Note also that the states x (for any Str x) and fail are the only normal forms for $\overset{\text{PEG}}{\Longrightarrow}$.

Theorem 3.1 (Adequacy). *Let G be a grammar, e a parsing expression and x a string. Then the following holds:*

(1) $G[e]\ x \overset{PEG}{\rightsquigarrow} y$ *iff* G[e]x $\overset{\text{PEG}}{\Longrightarrow}!$ y.

(2) $G[e]\ x \overset{PEG}{\leadsto} \texttt{fail}$ *iff* $\texttt{G[e]x} \overset{PEG}{\Longrightarrow}! \texttt{fail}$.

Proof. (sketch). We can show that, for any state S, $G[e]\ x \overset{\text{PEG}}{\leadsto} S$ iff $\texttt{G[e]x} \overset{\text{PEG}}{\Longrightarrow}! \texttt{S}$. This discharges both (1) and (2). Note that such S must be a normal form and then, either $S = \texttt{fail}$ or $S = x$ for some string x. The implication ($\Rightarrow$) is proved by induction on the height of the derivation of $G[e]\ x \overset{\text{PEG}}{\leadsto} S$. For the ($\Leftarrow$) side, assume that $t = \texttt{G[e]x} \overset{\text{PEG}}{\Longrightarrow}! \texttt{S}$. This means that there exists $n > 0$ (since $G[e]\ x$ is not a normal form) and a derivation of the form $t = t_0 \overset{\text{PEG}}{\Rightarrow} t_1 \overset{\text{PEG}}{\Rightarrow} \cdots \overset{\text{PEG}}{\Rightarrow} t_n \overset{\text{PEG}}{\not\Rightarrow}$. Due to the side conditions in the rules, each step on this derivation may include the application of several rules. Hence, we proceed by induction on m where m is the total number of rules applied (including side conditions) in the above derivation. See §A for more details. □

We can use this theory as a (naive) PEG parser. It suffices to use the Maude's rewriting mechanism on a PEG state:

```
rewrite ('A <- "a" , 'B <- "b") ['A .'B] "a" "b" "c" .
result State: "c"
```

As noticed, our specification follows exactly the semantic rules in Fig. 1, which makes it easier to prove its correctness. However, the resulting rewrite theory is not completely satisfactory due to the conditional rules used in choices, sequences and negative predicates: a whole derivation must be built to check whether the associated conditions are met or not. As we know, PEGs are deterministic and it should be possible to make a more guided decision on, e.g., whether **ord.1** or **ord.2** should be used on a given string. For that, the strategy on e_1/e_2 could be: reduce e_1 and, in the end, decide whether or not e_2 is discarded. This reduction strategy resembles a small-step semantics and we shall explore it in the next section. We shall show that the new specification is more efficient for checking $x \in \mathcal{L}(G)$ and, more importantly, it is appropriate for other (symbolic) analyses.

3.2 A More Efficient Rewrite Theory

Semantic and logical frameworks are adequate tools for specifying and reasoning about different systems. By choosing the right abstractions in the framework, we can also have efficient procedures for such specifications. In the following, we introduce a rewrite theory that gives an alternative representation of choices and failures in PEGs.

We first introduce semantic-level constructs for negation, choices, sequential composition and replication:

```
op NEG :    State State -> State [ frozen (2)]  .
op CHOICE : State State -> State [ frozen (2)] .
op COMP :   State State -> State [ frozen (2)] .
op STAR :   State State -> State [ frozen (2)] .
```

Note that such operators are defined on states. The attribute `frozen` will be explained in brief.

The rules `empty`, `Terminal12` and `Terminal3` are the same as in $\overset{\text{PEG}}{\Longrightarrow}$. Note that those rules are not conditional.

Let us introduce the new rules:

```
rl [NTerm] : (G, N <- e) [N] x => (G, N <- e) [e] x .
rl [Sequence] :  G[e . e'] x  => COMP(G[e]x ,  G[e']x ) .
rl [Seq1] : COMP(y ,  G[e']x) =>  G[e'] y .
rl [Seq2] : COMP(fail , S) => fail .
```

In `NTerm`, `N` is "simplified" to the corresponding expression `e`. The rule `Sequence` replaces the PEG constructor `e.e'` with the corresponding semantic-level constructor `COMP`. The meaning of `COMP` is given by the two last rules: if the first component succeeds returning `y`, then the second expression `e'` must be evaluated against the input `y`. Moreover, if the first component fails, then the whole expression fails.

The rules for the other constructors follow similarly:

```
rl [Choice] : G[e / e']x => CHOICE(G[e] x , G[e'] x) .
rl [Choice1] : CHOICE(x , S) => x .
rl [Choice2] : CHOICE(fail , S) => S .
rl [Star] : G[e *]x =>  STAR(G[e] x , G[e *] x) .
rl [Star1] : STAR(fail,  G[e *] x ) => x .
rl [Star2] : STAR(y ,  G[e *]x) => STAR(G[e] y ,  G[e *]y) .
rl [Negative] : G[! e] x  => NEG(G[e]x ,  G[! e] x ) .
rl [Neg1] : NEG(fail,  G[! e] x) => x .
rl [Neg2] : NEG(y, S) => fail .
```

The first rule in each case reduces the current expression to the corresponding constructor; the second and third rules decide whether the expression fails or succeeds. For instance, `Choice1` discards the second alternative if the first one succeeds and `Choice2` selects the second alternative if the first one fails.

Rewriting logic is an inherent concurrent system where rules can be applied at any position/subterm of a bigger expression. Hence, the use of the attribute `frozen` is important to guarantee the adequacy of our specification. This attribute indicates that the second argument (of sort `State`) of the operators `COMP`, `CHOICE`, `STAR` and `NEG` cannot be subject of reduction. In words, the state `S'` in `CHOICE(S,S')` is not reduced until the first component is completely reduced.

We shall use $\overset{\text{PEG}}{\longrightarrow}$ to denote the induced rewrite relation of the above specification.

Theorem 3.2 (Adequacy). *Let G be a grammar, e a parsing expression and x a string. Then the following holds:*

(1) $\texttt{G[e]x} \overset{PEG}{\Longrightarrow}! \texttt{y}$ *iff* $\texttt{G[e]x} \overset{PEG}{\longrightarrow}! \texttt{y}$

(2) $\texttt{G[e]x} \overset{PEG}{\Longrightarrow}! \texttt{fail}$ *iff* $\texttt{G[e]x} \overset{PEG}{\longrightarrow}! \texttt{fail}$

Proof. We shall prove (1) and (2) simultaneously, i.e., we shall show that $G[e]\ x \overset{\text{PEG}}{\Longrightarrow}! S$ iff $G[e]\ x \overset{\text{PEG}}{\longrightarrow}! S$ for $S \in \{\texttt{fail}, x\}$. ($\Rightarrow$) We proceed by induction on the total number of rules (including side conditions) used in the derivation $G[e]\ x \overset{\text{PEG}}{\Longrightarrow}! S$. The result is not difficult by noticing that in $\overset{\text{PEG}}{\Longrightarrow}$, the condition of the rules are satisfied by shorter derivations and then, by induction, one can show that the second and third rules in the respective cases for $\overset{\text{PEG}}{\rightarrow}$ mimic the same behavior.

($\Leftarrow$) Assume that $G[e]\ x \overset{\text{PEG}}{\rightarrow!} S$. There are no side conditions but there are some extra intermediate steps due to the semantic-level constructors `CHOICE`, `COMP`, etc and the rules `Choice`, `Sequence`, etc. We note that `COMP(S, S')` is not a normal form: if S is a normal form, then `Seq1` or `Seq2` are used to continue reducing the term. Also, due to the `frozen` attribute, `S'` is never reduced in the scope of the `COMP` constructor. Similar observations apply for `CHOICE`, `NEG` and `STAR`. We then proceed by induction on the total number of steps needed to show $G[e]\ x \overset{\text{PEG}}{\rightarrow!} S$. The base cases are easy. Consider the inductive case when the derivation starts with the rule `Negative`. Due to the above observations, we are in the following situation:

$$G[!e]\ x \overset{\text{PEG}}{\rightarrow} \mathsf{NEG}(G[e]\ x, G[!e]\ x) \overset{\text{PEG}}{\rightarrow}* \mathsf{NEG}(S_c, G[!e]\ x) \overset{\text{PEG}}{\rightarrow} S'$$

where S_c is a normal form and the rule applied on that state can be either `Neg1` or `Neg2` depending on S_c. This means that there is a shorter derivation for $G[e]\ x \overset{\text{PEG}}{\rightarrow!} S_c$ and the result follows by induction and applying **not.1** or **not.2** accordingly. □

From the above result, determinism and Theorem 3.1, we know that $\overset{\text{PEG}}{\rightarrow}$ is a function. Also, a more direct proof of that is possible: By simply inspecting the rules and noticing that all the left-hand sides are pairwise distinct and only the left-most state can be reduced (due to the `frozen` attribute).

The analyses we are currently working on (see §6 and §B), use symbolic techniques in Maude, as *narrowing*, that do not support conditional rewrite rules. This is one of the reasons to prefer $\overset{\text{PEG}}{\rightarrow}$ over $\overset{\text{PEG}}{\Rightarrow}$. Moreover, avoiding conditional rules, $\overset{\text{PEG}}{\rightarrow}$ outperforms $\overset{\text{PEG}}{\Rightarrow}$ for membership checking. As a simple benchmark, consider the following grammar recognizing the language $a^n b^n c^n$:

```
S <- ( &(R1 "c") ) ("a" +) R2 (! [.]) ,
R1 <- "a" (R1 ?) "b",
R2 <- "b" (R2 ?) "c"
```

In $\overset{\text{PEG}}{\Rightarrow}$, the instance $n = 6$ is recognized in 30 seg, and the instance $n = 7$ in 4.1 min. Hence, this specification will be of little help for practical purposes. $\overset{\text{PEG}}{\rightarrow}$ recognizes the instance $n = 1000$ in less than one second. Although some improvements can be done such as using indices to avoid carrying in each state the fragment of the string being processed, state of the art parsers for PEGs (e.g., Mouse [26], Rats! [10] and Parboiled [22]) can certainly do much better than that. Our goal is not to build a parser but to provide a formal framework for PEGs and explore reasoning techniques for it. Hence, we prefer to keep the specification as simple and general as possible to widen the spectrum of analyses that can be performed on it. The reader may have probably recognized the simplicity of the formal specification, driven directly from the syntax and semantic rules presented in §2.

4 Backtracks and Cuts in PEGs

This section builds on the previous rewrite theory to study backtracks in ordered choices. For that, we first introduce semantic rules for the *cut* operator in [20]. By studying formally the meaning of cuts, we propose a deeper cut operator (§4.2) that is able to save more computations. In §4.3 we show how local and global/deeper cuts can be uniformly introduced in PEGs and some extra optimizations are presented in §4.4. Finally, §4.5 reports some experiments on grammars annotated with cuts.

4.1 Semantics for Cuts

The cut operator, inspired in Prolog's cut, was proposed in [20] with the aim of better controlling backtracks in PEGs. The idea is to annotate the grammar G with cuts leading to a modified grammar G' in such a way that the language recognized by G and G' is the same but G' may avoid some backtracks in choices during parsing. In fact, as shown in [20], this technique allows for dynamically reclaiming the unnecessary space for memoization in those branches.

Cuts, as proposed in [20], cannot be introduced on arbitrary positions of a parsing expression. They only make sense when the backtracking mechanism needs to be controlled. Hence, there is a restricted syntax for them:

$$e ::= \ldots \mid e_1 \uparrow e_2 / e_3 \ \mid \ (e_1 \uparrow e_2)*$$

The intended meaning for the $\uparrow$ operator is:
- In $e_1 \uparrow e_2/e_3$, if e_1 succeeds, e_2 is evaluated and e_3 is never considered even if e_2 fails.
- In $(e_1 \uparrow e_2)*$, when e_1 fails, the entire expression succeeds. If e_2 fails (after the successful matching of e_1) the whole expression fails.

Consider for instance the following grammar:

```
S <- TIF ↑ "(" ...  /  TWHILE ↑ "(" ... / TFOR ...
```

Clearly, if the token "`if`" was read from the input, a failure occurring right after that point does not need to backtrack to consider the other alternatives, that inevitable will also fail. The introduction of the cut guarantees that: (1) information about the other alternatives can be discarded once the rule `TIF` succeeds (thus saving space); and (2) there is no need to explore other alternatives after failure (thus failing faster).

Adding cuts to a grammar needs to be done carefully. For instance, the languages generated by the expressions

$$a\ e_1 / a\ e_2 \qquad \text{and} \qquad a \uparrow e_1 / a\ e_2$$

are not necessarily the same. Algorithms for adding cuts to grammars are proposed in [20]. Roughly, the `FIRST` set is used to check whether the alternatives are disjoint. If this is the case, it is safe to introduce a cut.

We can formally state the semantics of cuts in the scope of choices through the following rule:

```
rl [Choice^] CHOICE(G[↑  e] x , S) =>  G[e] x.
```

This rule reflects the fact that, once the symbol ↑ is found in the context of an ordered choice, the second alternative (the variable `S` of type `State`) can be discarded.

Let us analyze the case of repetition. Since this operator can be encoded using recursion, one may simply use the previous definition. However, in our framework, $*$ is not a derived constructor and we are forced to give meaning to $(e \uparrow e')*$. The first thing we note is that the expression $(ee')*$ never fails but $(e \uparrow e')*$ may fail. For instance, $(ab)*$ recognizes the string "abac" (returning "ac") while $(a \uparrow b)*$ fails to match the same string. This means that the use of cuts in the context of repetitions requires extra care. In fact, the grammar transformations proposed in [20] translates expressions of the form $e * \ e'$ into $(!e' \ \uparrow e) * e'$ whenever e, e' are both non-nullable expressions [25] (i.e., they do not recognize the empty string) and e, e' are disjoint. Hence, cuts in repetitions appear in very controlled ways.

We can give meaning to $(e \uparrow e')*$ by dividing its execution into two steps: when processing e, failures cause the success of the whole expression; when processing e', a failure must cause the failure of the whole expression. For that, besides the semantic-level operator `STAR` introduced in the previous section, we also consider the following one:

```
op STAR^ :  State State -> State [ctor frozen (2)] .
```

This operator will be used to keep track of the execution of the expression e' as follows:

```
rl [Star] :  STAR(G[↑ e'] x' , S ) => STAR^( G[e'] x', S) .
rl [Star^] : STAR^(y , G[ e *] x) => STAR(G[e] y, G[e *] y) .
rl [Star^] : STAR^(fail, S) => fail .
```

The first rule makes the transition from `STAR` to `STAR^`. This happens when the current expression is a cut. The second rule models the recursive behavior: if e' succeeds, then we are back into the `STAR` state. The third rule reduces to `fail` if e' fails.

These rules formalize the behavior of cuts in agreement with the intended meaning in [20]. However, the treatment of cuts does not look uniform: the meaning of ↑ is given in the context of other parsing expressions and not in a general way. The next section shows that it is possible to give a more precise meaning to failures, cuts and backtracks. On doing that, we propose a deeper cut operator with a more elegant semantics that, in some cases, may avoid more backtracks than ↑.

4.2 Global Errors and Cuts

The cut operator ↑ acts locally. Take for instance the PEG

```
  A <- T1 ↑ e1   /  T2 ↑ e2
  B <- T3 ↑ e3   /  T4 ↑ e4
```

$$\frac{}{G[\Uparrow]\ x \overset{\text{PEGe}}{\rightsquigarrow} \texttt{error}}\ (\text{throw})$$

$$\frac{G[e_1]\ x \overset{\text{PEGe}}{\rightsquigarrow} \texttt{error}}{G[e_1\ e_2]\ x \overset{\text{PEGe}}{\rightsquigarrow} \texttt{error}}\ (\text{seq.3}) \qquad \frac{G[e_1]\ x \overset{\text{PEGe}}{\rightsquigarrow} \texttt{error}}{G[e_1\ /\ e_2]\ x \overset{\text{PEGe}}{\rightsquigarrow} \texttt{error}}\ (\text{ord.3})$$

$$\frac{G[e]\ x \overset{\text{PEGe}}{\rightsquigarrow} \texttt{error}}{G[!e]\ x \overset{\text{PEGe}}{\rightsquigarrow} x}\ (\text{not.3}) \qquad \frac{G[e]\ x \overset{\text{PEGe}}{\rightsquigarrow} \texttt{error}}{G[e*]\ x \overset{\text{PEGe}}{\rightsquigarrow} \texttt{error}}\ (\text{rep.3})$$

Figure 2. Global errors. $\overset{\text{PEGe}}{\rightsquigarrow}$ extends $\overset{\text{PEG}}{\rightsquigarrow}$ in Fig 1.

```
  C <- A / B
```

where `T1`,`T2`,`T3`,`T4` are lexical non-terminal symbols and the grammar satisfies the unique token prefix condition (Def. 2.4). Assume also that: the input starts with the string recognized by `T1`; the initial expression is `C`; and the expression `e1` fails to match the input. Due to the semantics of ↑, the second alternative in `A` is discarded and `A` fails. However, such failure does not propagate to `C` and the alternative `B` is tried against the input. Since all the tokens are disjoint, after an attempt on `T3` and `T4`, the expression `C` finally fails. The key point is: failures in sub-expressions are *confined* and they do not *propagate* to outer levels. In the following, we propose a new operator that acts as a cut but *globally*.

We first extend the set of states (Def. 2.2) with the constant `error`, denoting the fact that an *unrecoverable* error has been *thrown* and therefore the global expression must fail:

```
op error : -> State .
```

Moreover, we add to the syntax of parsing expressions the operator ⇑ that, intuitively, *throws* an error:

```
op throw : -> Exp .     --- Representation of ⇑
op check : Exp -> Exp . --- Derived constructor
eq check(E) = E / throw .
```

The expression `check(e)` tries to match the expression `e` on the current input. If it fails, an error is thrown.

As we saw, the definition of the cut operator ↑ is problematic since its behavior depends on the context where it is evaluated. The operator ⇑ can appear in any context and its semantics will be given in a uniform way. In fact, the definition is quite simple: on any input, ⇑ raises an error:

```
rl [throw] : G[throw] x => error .
```

Since we have two failing states, namely `fail` and `error`, the set of rules in §3.2 must be extended accordingly:

```
rl [SeqE] :     COMP(error, S)    =>  error .
rl [ChoiceE] :  CHOICE(error, S)  =>  error .
rl [StarE] :    STAR(error, S)    =>  error .
rl [NegE] :     NEG(error, G[! e]x)  => x .
```

These definitions allow for the propagation of the error to the outermost context as the Example 4.1 below shows. From now on, we shall use $\overset{\text{PEGe}}{\longrightarrow}$ to denote the induced rewriting relation from the theory above. The new rules in natural-semantic style are in Figure 2.

Example 4.1. Consider the following grammar for labeled, goto and break statements:

```
St <- labeledSt / jumpSt
labeledSt <-  ID check(COLON statement) /
              CASE constantExp COLON statement
jumpSt <- GOTO check(ID SEMICOLON) /  BREAK SEMICOLON
```

Let G' be as G but removing the `check()` annotations. Since the token expressions `ID`, `CASE`, `GOTO` and `BREAK` are all disjoint, we can show that:

(1) `G[St]x` $\overset{\text{PEGe}}{\longrightarrow}*$ `error` implies `G'[St]x` $\overset{\text{PEG}}{\longrightarrow}*$ `fail`.

(2) `G'[St]x` $\overset{\text{PEG}}{\longrightarrow}*$ `fail` implies `G[St]x` $\overset{\text{PEGe}^*}{\longrightarrow}$ $l \in \{$ `fail`, `error`$\}$.

In (1), the $\overset{\text{PEGe}}{\longrightarrow}$-derivation does not need to match the (useless) alternatives, thus failing in fewer (or equal) steps when compared to the corresponding $\overset{\text{PEG}}{\leadsto}$-derivation

In the example above, the number of backtracks is reduced when processing syntactically invalid inputs. On valid inputs, the number of backtracks remains the same. When predicates are involved, it is also possible to save some few (unnecessary) backtracks.

Example 4.2. According to the ISO 7185 and ISO 10206 standards, Pascal allows comments opened with `(*` and closed with `}`. Consider the following grammar:

```
comment <- open  (!close [.]) * close
open <- "(" "*" / "{"
close <- "*" check(")") / "}"
```

On input "{ comment *here* *)", the first and second "*" fail immediately (without trying to match "}") and `!close` succeeds faster. Of course, this does not save much effort. As another example, a typical rule for identifiers looks like this:

```
ID <- ! KEYW [a-zA-Z] ([a-zA-Z0-9_] *)
```

Some cuts can be added to the definition of reserved words, thus making the above predicate to fail faster:

```
KEYW <- 'a' 'n' check('d') / 'a' check('s') /
        'b' 'o' check('o' 'l') / 'b' 'r' check('e' 'a' 'k')...
```

On input "bot", `ID` succeeds and only the first three choices of `KEYW` will be evaluated.

Similar to $\uparrow$, a careless use of $\Uparrow$ may change the language recognized. Indeed, the situation is aggravated by the fact that errors propagate to outer levels. For instance, if `St` in Example 4.1 is extended with a third choice recognizing `ID`, a failure in `labeledSt` should not be propagated to `St` since the new third alternative cannot be discarded. Hence, it is salutary to control the propagation of failures: a failure in `labeledSt` should avoid the (unnecessary) matching of `jumpSt` but such failure must remain *confined* to these two alternatives. This local behavior (akin to the one of $\uparrow$) in combination with global failures will be explored in the next section.

4.3 From Global to Local Errors

Now we introduce a *catch* mechanism to control the global behavior of $\Uparrow$ and confine errors when needed. The following definitions extend the syntax of expressions and states:

```
op catch : Exp -> Exp .      --- Catch operator (syntax)
op CATCH : State -> State .  --- Semantics (on states)
```

The intended meaning is the following. If the expression `e` succeeds, then `catch(e)` also succeeds. If `e` fails with final outcome $l \in \{$`error`, `fail`$\}$, the expression `catch(e)` fails with output `fail`. This is formalized with the following rules:

```
rl [Catch]   : G[catch(e)] x  => CATCH(G[e] x) .
rl [Catch1]  : CATCH(x) => x .
rl [Catch2]  : CATCH( fail )  => fail .
rl [Catch3]  : CATCH( error ) => fail .
```

As in the previous cases, the first rule moves from the syntactic level to the semantic operator at the level of `States`. The last three rules act on normal forms materializing the intuition given above: `errors` are transformed into `failures`.

Example 4.3. Consider the production rules for labeled and jump statements in Example 4.1 and the rules below:

```
S1 <- catch(labeledSt / jumpSt) / assignSt
S2 <- catch(labeledSt) / jumpSt / assignSt
assignSt <- ID EQ expr SEMICOLON
```

Since the token `ID` appears in both `assignSt` and `labeledSt`, `errors` must be confined to guarantee that the `check(.)` annotations do not modify the language recognized. `S1` and `S2` are two alternative solutions. Consider the input string "`x = 3 ;`". In `S1`, `labeledSt` produces `error` and `jumpSt` is not evaluated (global behavior). This error is confined in `S1` (local behavior) and `assignSt` successfully recognizes the input. A similar situation happens in `S2` but, when `labelSt` fails, `jumpSt` is also evaluated (and fails). Consider now the input "`goto l :`" (note the ":" instead of ";"). In `S1`, the failure of `jumpSt` is confined and `assignSt` is (unnecessarily) evaluated. In `S2`, such failure preempts the execution of `assignSt` and fails faster.

Some other transformations on this grammar are possible to control better the shared `ID` token between `labeledSt` and `assignSt`. For instance, it is possible to join together assignments and label statements, with the clear disadvantage of making the grammar more difficult to read and understand. The use of *catch* expressions is common in programming languages, thus making the grammar above more comprehensible.

We shall use $\overset{\text{PEGec}}{\longrightarrow}$ to denote the resulting relation extending $\overset{\text{PEGe}}{\longrightarrow}$ with the rules above. Figure 3 depicts the corresponding rules in natural-semantic style.

4.4 Simplifying States

Unlike $\uparrow$, the $\Uparrow$ operator has its own meaning independent of the context where it appears. In particular, it does not

$$\frac{G[e]\ x \overset{\text{PEGec}}{\leadsto} y}{G[\mathtt{catch}(e)]\ x \overset{\text{PEGec}}{\leadsto} y}\ (\text{catch.1}) \qquad \frac{G[e]\ x \overset{\text{PEGec}}{\leadsto} \mathtt{fail}}{G[\mathtt{catch}(e)]\ x \overset{\text{PEGec}}{\leadsto} \mathtt{fail}}\ (\text{catch.2})$$

$$\frac{G[e]\ x \overset{\text{PEGec}}{\leadsto} \mathtt{error}}{G[\mathtt{catch}(e)]\ x \overset{\text{PEGec}}{\leadsto} \mathtt{fail}}\ (\text{catch.3})$$

Figure 3. Semantic rules for the *catch* mechanism.

$$\frac{G[e]\ x \overset{\text{PEGtc}}{\leadsto} y}{G[\mathtt{try}(e)]\ x \overset{\text{PEGtc}}{\leadsto} y}\ (\text{try.1}) \qquad \frac{G[e]\ x \overset{\text{PEGtc}}{\leadsto} \mathtt{fail}}{G[\mathtt{try}(e)]\ x \overset{\text{PEGtc}}{\leadsto} \mathtt{error}}\ (\text{try.2})$$

$$\frac{G[e]\ x \overset{\text{PEGtc}}{\leadsto} \mathtt{error}}{G[\mathtt{try}(e)]\ x \overset{\text{PEGtc}}{\leadsto} \mathtt{error}}\ (\text{try.3})$$

Figure 4. Semantic rules for the *try* mechanism.

need to be used only inside a choice operator. For instance, the three expressions $\Uparrow$, $a \Uparrow$ and $a \Uparrow b$ are all valid expressions (whose language is empty). It is also worth noticing the difference on how the two cut operators deal with the remaining alternatives: when the expression `check(e) / e'` is evaluated, the second alternative `e'` is only discarded when `e` actually fails. Differently, the expression $\uparrow e/e'$ discards eagerly the second alternative, thus saving some space. One then may wonder whether it is possible to reconcile the idea of *saving memory* that motivated the introduction of cuts in [20] with the behavior proposed here for $\Uparrow$. It turns out that the rewriting logic framework can give us some ideas on how to do that as described below.

We first introduce an alternative version of `check`:

```
op try : Exp -> Exp .      --- Syntactic level
op TRY : State -> State . --- Semantic level
```

Unlike `check(.)`, `try(.)` is not a derived operator and hence, its meaning needs to be specified:

```
rl [Try] :  G[try(e)] x => TRY(G[e] x) .
rl [Try1] : TRY(x) => x .
rl [Try2] : TRY( fail )  => error .
rl [Try3] : TRY( error ) => error .
```

Note the duality between `TRY` and `CATCH`: the former converts `fail`ures into `error`s and the latter maps `error`s into `fail`ures. Intuitively, `try(e)` succeeds if `e` succeeds and fails with `error` if `e` fails. We shall use $\overset{\text{PEGtc}}{\longrightarrow}$ to denote the extension of $\overset{\text{PEGec}}{\longrightarrow}$ with the rules for `TRY`. The additional rules are also depicted in Fig. 4.

As a simple example, the languages recognized by the expressions `"a"` and `try("a")` are the same. However, the final failing states are different on input "b": `try("a")` ends with an `error` (that nobody caught).

The following equivalences are an easy consequence from the definition of $\overset{\text{PEGtc}}{\longrightarrow}$ (and determinism):

1. `catch(try(e))` $\equiv$ `e` (cancellation)
2. `catch(e) catch(e')` $\equiv$ `catch(e.e')`. (distributivity)
3. `catch(! e)` $\equiv$ `!catch(e)` (distributivity)

Note that, in general, `catch(.)` does not distribute on choices. As a simple counterexample, $\mathtt{catch}(\Uparrow /\epsilon)$ always reduces to `fail` while $\mathtt{catch}(\Uparrow)/\mathtt{catch}(\epsilon)$ succeeds on any string. Note also that, even if $\mathtt{catch}(\mathtt{try}(e))$ and e recognize the same language ($\mathtt{catch}(e)$ and $\mathtt{try}(e)$ accepts x iff e does), their failing states are different. Hence, e/e' is not necessarily equivalent to the expression $\mathtt{catch}(\mathtt{try}(e))/e'$ (i.e, $\equiv$ is not a congruence).

Now let us come back to the problem of saving the space needed to store backtracking information that will never be used. As explained in §3, equations in the equational theory can be used to "simplify" terms. The idea is to extend the equational theory, so that we can add some structural rules that govern the state of the parser. The simplification proposed is the following:

```
eq [simpl] : CHOICE(TRY(S), S') = TRY(S).
```

Here, the whole state `S'` can be ignored since it will never be evaluated. Note that this simplification is sound due to the semantics of `CHOICE`: if S fails then, `TRY(S)` reduces to `error` and `S'` will never be evaluated. Note that such simplification applies only when `TRY` is in the immediate scope of a `CHOICE`. Moreover, `[simpl]` may simplify several choices. For instance, in this theory, `CHOICE(CHOICE(TRY(S), S'), S'')` is equal to `TRY(S)` (discarding `S'` and `S''`).

Let $\overset{\text{PEGtc}'}{\longrightarrow}$ be the extension of $\overset{\text{PEGtc}}{\longrightarrow}$ with the equation above.

Theorem 4.4. $\mathtt{G[e]\ x} \overset{PEGtc}{\longrightarrow !}\ \mathtt{S}$ *iff* $\mathtt{G[e]\ x} \overset{PEGtc'}{\longrightarrow !}\ \mathtt{S}$.

Proof. Observe that the term `TRY(S)` either reduces to x or fail with label `error`. In the first case, `CHOICE(TRY(S), S')` reduces to x and, in the second case, reduces to `error`. Hence, the result depends only on the outcome of `TRY(S)`. □

4.5 Experimental Results

In this section we present some benchmarks performed on grammars manually annotated with the cut constructors proposed here. When annotating a grammar, given a concatenation $e_1\ e_2$, the general idea is to annotate the symbols in e_2 since those in e_1 match at least one input symbol (if e_1 is not nullable). When e_1 matches the input, we are in a (local or global) right path, so we can safely discard other alternative paths and avoid backtracking when e_2 fails.

The Maude's specification can be seen as an abstract machine whose derivation steps correspond, approximately, to the operations performed by a parser. We shall report the number of entry rules that need to be applied to reduce $G[e]x$ into a final state. Such number corresponds to the number of times a PEG constructor is evaluated. For instance, if $G = \{A \leftarrow a\}$, the expression $G[Ab]ab$ reduces to ϵ in 4 steps: $1_\times$ for sequential composition, $2_\times$ for the terminal rule and $1_\times$ for the non-terminal rule. With this methodology, it is not surprising that, on valid inputs, the annotated grammar performs more steps since entering on try(e) also counts as a step. We also report some time measures to show that it is feasible to use our formal framework for concrete experiments.

JSON. The main non-terminal of the JSON grammar tries all the possible shapes for a value:

```
value <- str / num / obj / arr / "true" / ...
```

Strings start with (simple) quotes, numbers with digits, objects with "`{`", arrays with "`[`", etc. Those tokens are unique and, once consumed, a failure in the remaining expression should make the whole expression, including `value`, to fail. This allows us to introduce some cuts, for instance,

```
obj <- '{' pair (',' pair)* try('}')  / '{' try('}')
```

Hence, an open curly bracket without the corresponding closing one causes a global failure. It is also possible to replace the second occurrence of `pair` with `try(pair)`: once `','` is consumed, the parser cannot fail in recognizing a pair.

We took some files from repositories benchmarking JSON parsers and produced random invalid files by word mutations [24]. More precisely, we randomly deleted some (maximum 10) symbols `']'`,`'}'`, `':'` and `','` in the corresponding file. For each of the 9 (correct) files, we generated 10 invalid files.

The results are in Table 1. The columns are: the size of the file; the sum of the steps in the 10 cases for the grammar and the annotated grammar; the percentage of reduced steps ($1.0 - Gcut/G$); the number of steps in the grammar and the annotated grammar when processing the original (valid) file; and the percentage of the steps increased ($1.0 - Gcut/G$).

Due to the format of JSON files, the same production rule is applied several times. Hence, if an error occurs towards the end of the file, the try($\cdot$) constructor will be invoked several times and, only in the end, it will save some backtracks. On a MacBook Pro (4 cores 2,3 GHz, 8GB of RAM) running Maude 3.0, processing in batch the 9 valid files and the 90 invalid files with the two grammars takes 4.76s (average of 10 runs).

Pallene. Consider now the grammar for Pallene [11], a statically-typed programming language derived from Lua. It is interesting to see the granularity we can achieve with the cut mechanism proposed here. For instance, in the rules

```
import <- 'local' try(name) try('=') 'import' ...
foreign <-'local' try(name) try('=') 'foreign' ...
```

`'import'` does not raise an error when failing (and other alternatives are still available) while a failure in `name` is global. Compare this behavior with the annotation $e \uparrow e'$: after $\uparrow$, it is not possible to control which sub-expressions of e' can be considered as normal failures (where alternatives cannot be discarded) or unrecoverable errors.

We took 74 invalid small programs (< 60 bytes per file) that were proposed by the designers of Pallene as benchmarks to test some specific features of the language. We also processed 87 correct files from the same repository.

The results are in Table 2. Maude took 2.67s to process the 322 tests (valid and invalid files with the two grammars).

C89. Finally, we considered the grammar of C89. Here some examples of annotations:

```
enumerator <- ID '=' try(const-exp)  /  ID
case <- "case" try(const-exp ':' stat)
```

When enumerating the elements of a `enum` expression, if after a name there is an an assignment operator ('='), then an expression must come after it. The second rule fails when the token `case` is consumed and any of the remaining expressions fail.

We took fragments of the files `trace.c`, `tree-diff.c`, `version.c`, `walker.c` in the implementation of Git (https://github.com/git/git). For each case, we generated 10 different files by randomly deleting curly brackets, parentheses and semicolons. The results are in Table 1. In this case, Maude took 91.3s (avg. of 10 runs) to process the whole collection of files.

5 Related Work

Hutton introduced the use of a new parser combinator to make a distinction between an error and a failure during the parsing process [12].

The idea of defining multiple kinds of failures for PEGs was proposed in [13]. In this work, the authors introduced labeled failures as a mechanism to improve syntactic error reporting in PEGs by associating a different error message for each label. This first formalization provided a kind of choice operator that could handle set of labeled failures. These labeled choices can also be used as prediction mechanism, where a label indicates which alternative of a choice should be tried [14], allowing PEGs to simulate the LL(*) algorithm used previously by ANTLR [23].

However, this first labeled failures formalization makes harder to recover from an error in its own context, since when a label is handled in a choice, the information about the error context is lost. Because of this, attempts to deal with error recovery in PEGs favored a semantics where the $\Uparrow$ operator itself handles the error [6, 16]. To recover from an error, the $\Uparrow$ operator tries to match a recovery expression (a regular parsing expression).

The present work uses the idea of labeled failures to make a distinction between local errors and global errors, where both kind of errors may avoid unnecessary backtracking when compared to a regular failure. The *catch* and *try* mechanisms can be simulated by using the error recovery semantics of the $\Uparrow$ operator. In the case of catch($\cdot$), the recovery expression of $\Uparrow$ should be an expression that always fail (e.g., $!aa$), thus resulting in a regular failure. In case of try($\cdot$),

Table 1. Experiments with JSON and C89.

JSON						
File	**Non-valid inputs**			**Valid inputs**		
Bytes	**Grammar**	**Grammar+cuts**	**%**	**Grammar**	**Grammar+cuts**	**%**
276	9,545	7,339	23.1%	2,096	2,103	-0.3%
872	12,682	10,141	20.0%	6,968	6,989	-0.3%
1k	34,149	29,092	14.8%	8,276	8,301	-0.3%
2k	27,075	23,180	14.4%	12,718	12,745	-0.2%
4k	46,611	42,614	8.6%	22,993	23,042	-0.2%
2k	112,805	108,687	3.7%	25,439	25,452	-0.1%
6k	61,323	57,369	6.4%	39,700	39,802	-0.3%
10k	72,036	68,265	5.2%	62,971	63,082	-0.2%
14k	117,316	113,892	2.9%	90,022	90,174	-0.2%
C89 (Git files)						
File	**Non-valid inputs**			**Valid inputs**		
lines	**Grammar**	**Grammar+cuts**	**%**	**Grammar**	**Grammar+cuts**	**%**
trace (329)	3,770,039	3,692,000	2.1%	1,222,637	1,224,472	-0.2%
tree (131)	1,481,679	1,317,197	11.1%	240,599	240,995	-0.2%
version (35)	826,840	798,336	3.4%	165,720	165,978	-0.2%
walker (144)	4,177,590	4,051,591	3.0%	787,573	789,148	-0.2%

Table 2. Experiments for Pallene

# Files	**Grammar**	**Gram+cuts**	**%**
74 non-valid files	124,130	102,056	17.8%
87 valid files	270,783	272,611	-0.7%

we simply do not provide a recovery expression for $\Uparrow$. Although $\Uparrow$ can simulate `catch(.)` and `try(.)`, the use of these specific constructs helped us to show how to control local and global backtracks in PEGs, and to properly formalize the cut mechanism proposed in [20].

Rewriting logic has been extensively used for specifying and verifying different systems [18]. In the context of programming languages, it is worth mentioning the K Framework [27]. Symbolic techniques in rewriting are currently the focus of intensive research (see a survey in [7]). In fact, one of the inspirations of this work came from an example of CFGs reported in [1] (and also used in [7]). Roughly, *narrowing* (rewriting with logical variables as in logic programming) was used to explore symbolically the state of programs and apply partial evaluation [5] (a program transformation) to improve the efficiency of Maude's specifications.

6 Concluding Remarks

We have proposed a framework based on rewriting logic to formally study the behavior of PEGs. Relying on this machinery, we proposed a general view of cuts where local and global failures can be treated uniformly and with a clear semantics. Such operators have a pleasant similarity to the usual try/catch expressions in programming languages, thus making it easier to understand and predict their behavior when designing a grammar. Our specification is not only a formal theory of the derivability relation in PEGs, but it is also an executable system. Based on it, we have tested some optimizations on grammars.

In §B, we report on a preliminary attempt of using the rewrite logic theory developed here, together with symbolic techniques, to propose a derivative parser that can be used as basis for other analyses. In particular, we show that it is possible to generate all the strings, up to a given length, from a grammar. This should provide more tools for PEG's user to verify whether a grammar correctly models a given language of interest. We still need to refine this work and to further investigate how it compares to other works that explored the use of derivatives in PEGs [9, 21] and in CFGs [19].

We also foresee to use the symbolic traces to generate positive and negative cases for a grammar [24]. It is also worth exploring if the symbolic semantics in §B can be useful for proving language equivalence on restricted fragments of PEGs (such problem is undecidable in general [8]).

The manual insertion of catch and try expressions in a grammar may be a tedious and error prone task. Moreover, it may hinder the grammar clarity. Fortunately, a good amount of theses annotations can be done automatically. We are currently exploring different automatic labeling algorithms inspired by those in [20] and [6]. Our framework can be handy in proving the correctness of the resulting algorithm (i.e., the language is preserved after the annotation). This approach based on the automatic annotations should preserve the grammar clarity while still avoiding some amount of unnecessary backtracking.

Finally, one of the main reasons for introducing cuts in PEGs is to save memory in packrat parsers [20]. The cuts proposed here generalize this idea and avoid further backtracks. Since our specification does not build a memoization table, measuring the usage memory when running Maude is pointless. Hence, it may be worth implementing global cuts in a packrat parser in order to evaluate the impact of the optimization in terms of memory consumption.

A Proofs of Adequacy Theorems

Theorem 3.1 ($\overset{\text{PEG}}{\Rightarrow}!$ and $\overset{\text{PEG}}{\rightsquigarrow}$ coincide)

Proof. Recall that strings (`x`) and `fail` are normal forms for $\overset{\text{PEG}}{\Rightarrow}$. Also, if $G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} S$, S is either `fail` or some string x. We shall show that $G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} S$ iff `G[e]x`$\overset{\text{PEG}}{\Rightarrow}!$ `S` for any S. This discharges both (1) and (2).
($\Rightarrow$). We proceed by induction on the height of the derivation of $G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} S$. Assume a derivation of height 1. Hence, either $S = x$ and the rules **empty** or **term.1** were used; or $S =$ `fail` and **term.2** or **term.3** were used. In the case **empty**, $x = y$, $e = \epsilon$ and clearly `G[emp]x` $\overset{\text{PEG}}{\Rightarrow}$ `x` by using the rule `empty`. In the case **term.1**, $x = ay$ and $e = a$. The corresponding term `G[a] ay` matches the left-hand side of `Terminal12`, `match(a,a)` reduces to `true` and the whole term reduces to `y` as expected. The cases when $S =$ `fail` are similar.

Assume now that $G[e]\ x \overset{\text{PEG}}{\rightsquigarrow} S$ is proved with a derivation of height > 1. He have 9 cases. Let us consider some of them since the others follow similarly. If **var** was used, $e = A$ and there is a (shorter) derivation of $G[P(A)]\ x \overset{\text{PEG}}{\rightsquigarrow} S$. By induction, we know that `G [P(A)]x` $\overset{\text{PEG}}{\Rightarrow}!$ `S`. Seen G as a set of rules, $G = G' \cup \{A \leftarrow P(A)\}$. If the current expression is `A` (a non-terminal), the only matching rule on the corresponding term `t = (G', A <- P(A)) [A] x` is `NTerminal` that unifies $G = G'$, $N = A$ and $e = P(A)$. Hence, $t \overset{\text{PEG}}{\Rightarrow}$`G[P(A)] x` which, in turns reduces to `S`. Assume now that the derivation ends with **ord.2**. By induction we know that `G[e1]x`$\overset{\text{PEG}}{\Rightarrow}!$`fail` and also, `G[e2]x`$\overset{\text{PEG}}{\Rightarrow}!$`S`. There are two rules that match `e1 / e2` (`Choice1` and `Choice2`). However, the side condition in `Choice1` does not hold. Using `Choice2`, we know that `G[e1 / e2]x`$\overset{\text{PEG}}{\Rightarrow}$`G[e2]x` that, in turns, reduces to `S`. The other cases follow similarly.
($\Leftarrow$). Assume that $t =$`G[e]x`$\overset{\text{PEG}}{\Rightarrow}!$ `S`. This means that there exists $n > 0$ (since $G[e]\ x$ is not a normal form) and a derivation of the form $t = t_0 \overset{\text{PEG}}{\Rightarrow} t_1 \overset{\text{PEG}}{\Rightarrow} \cdots \overset{\text{PEG}}{\Rightarrow} t_n \overset{\text{PEG}}{\not\Rightarrow}$. Due to the side conditions in the rules, each step on this derivation may include the application of several rules. Hence, we proceed by induction on m where m is the total number of rules applied (including side conditions) in the above derivation. If $m = 1$ then $n = 1$ and either `empty`, `Terminal12` or `Terminal3` were used. The needed derivation in $\overset{\text{PEG}}{\rightsquigarrow}$ results from applying the corresponding rule. If $m > 1$ we have several cases depending on the rule applied on t_0. Assume that the derivation starts with `Seq1`. This means that $e = e1.e2$. Moreover, $t_1 = G[e2]y$ and, by the side condition of the rule, `G[e1] x => y`. By induction, $G[e_2]\ y \overset{\text{PEG}}{\rightsquigarrow} S$. Since `y` is a normal form, `G[e1]x`$\overset{\text{PEG}}{\Rightarrow}!$ `y` with a smaller number of steps and, by induction, $G[e_1]\ x \overset{\text{PEG}}{\rightsquigarrow} y$. Now we use **seq.1** to conclude this case. The other cases follow similarly. □

B Symbolic Executions and Derivatives

One useful technique to predict the behavior of a grammar is to automatically generate strings from it and check whether the results match the intuitive behavior. There are recently works implementing derivative parsers [3] for PEGs to accomplish this task [9, 21]. One of the main difficulties is precisely the backtrack mechanism in PEGs. Hence, the algorithms need to keep track of the different branches [21] and compute possible over approximations [9] of different notions as testing whether an expression recognizes the empty string (which is undecidable in general [8]). This section shows that our formal specification can be adapted to implement a derivative tool. By using constraints, we give a symbolic and compact representation of the possible outputs (of a fixed length) derivable from a grammar.

Constrained Strings. The first step is to replace the input strings with sequences of *constraints* $c_1.c_2\ldots$. Intuitively, c_i represents the set of characters allowed in the i-th position of the string. To formalize this idea, we rely on the concept of constraint systems, commonly used in constraint logic programming [28]. A constraint system provides a signature from which constraints can be constructed as well as an entailment relation $\vdash$ specifying interdependencies between these constraints. A constraint represents a piece of *partial information* and $c \vdash d$ means that information d can be deduced from c. In the following definition, t is a terminal symbol (`TSymbol`), a is a character (`TChar`) and t_c represents a character class (`TExp`). Given a terminal symbol t, we shall use $dom(t)$ to denote the set of characters allowed by t (e.g., $dom(\text{“}x\text{”}) = \{x\}$, $dom([0\text{-}9]) = \{0, 1, \ldots, 9\}$, etc.).

Definition B.1 (Constraint System)**.** Constraints, usually ranged over $c_1, c_2, etc.$ are built from:

$$c ::= \mathtt{tt} \mid \mathtt{ff} \mid t \mid\sim t \mid c \wedge c$$

The entailment relation $\vdash$ is the least relation closed by the rules of intuitionistic logic and the following axioms:

$$\begin{array}{ll} t \wedge t' \vdash \mathtt{ff} & \text{whenever } dom(t) \cap dom(t') = \emptyset \\ t \wedge \sim t' \vdash \mathtt{ff} & \text{whenever } dom(t) \cap \overline{dom(t')} = \emptyset \\ t \vdash t' & \text{whenever } dom(t) \subseteq dom(t') \end{array}$$

Constrained string are sequences of constraints and c^n denotes the strings containing n copies of c.

Let us give some intuitions. The entailment $3 \vdash [0-9]$ holds since 3 is stronger (i.e., it constraints more) than [0-9]. When more constraints are added, the set of characters allowed decreases. Hence, conjunction corresponds to intersection of domains ($dom(c \wedge c') = dom(c) \cap dom(c')$). ff is the strongest constraint (since ff $\vdash c$ for any c) and it represents an inconsistent state (with empty domain). tt is the weakest (i.e., $c \vdash$ tt for any c) constraint and it is equivalent to [.]. $\sim t$ can be interpreted as the complement of the domain of t. Finally, note that the constraint "x" $\wedge [0-9]$ (resp. "3"$\wedge \sim$ [0-9]) is inconsistent in virtue of the first (resp. second) axiom of $\vdash$.

The above definition gives rise to the expected signature:

```
--- Atomic constraints and constraints
sort AConstraint Constraint .
subsort TSymbol < AConstraint < Constraint .
op ~_ : TSymbol -> AConstraint .
ops ff tt : -> Constraint . --- True and False
op _/\_ : Constraint Constraint  -> Constraint .
```

Our derivative parser will refine, monotonically, constrained strings. This means that, in each step, more information/constraints will be added. In order to keep the output as short/readable as possible, we define some simplifications on constraints. For example, if $d \vdash c$, then $c \wedge d$ can be simplified to d (since d contains more information than c):

```
eq c /\ ff = ff . --- ff entails any c
eq c /\ tt = c .  --- c entails tt
eq c /\ c = c .   --- idempotency
eq a /\ a' = if a == a' then a else ff fi .
eq a /\ ~ a' = if a == a' then ff else a fi .
```

For instance: "a" $\wedge [a-z]$ reduces to "a"; "a" $\wedge$ "b" reduces to ff; "a" $\wedge \sim$ "b" reduces to a; etc. More generally,

$$c \wedge c' = c' \quad \text{whenever } dom(c') \subseteq dom(c)$$
$$c \wedge c' = \texttt{ff} \quad \text{whenever } dom(c) \cap dom(c') = \emptyset$$

Now we define constrained strings:

```
sort CStr .  subsort  Constraint < CStr .
op _._ : Constraint CStr -> CStr [ctor right id: nil] .
eq ff . c . x = ff .  --- Simplifying inconsistencies
eq c . ff . x = ff .
```

Note that, e.g., the constrained string [0-9].ff."a"... collapses into ff.

Derivative Rules. The derivative parsing is obtained by adjusting the specification in §3. First, states take the form `G[e]  x :: y` meaning that the constrained string `y` was already processed and `x` is still being processed. Moreover, the final states take the form `ok(x,y)` (`y` was successfully consumed and the non-consumed input is `x`) and `fail(x,y)` (`x` could not be processed and failed). The main change occurs in the terminal rules:

```
rl [Terminal] : G[t] (c.x) :: y => ok(x , ins( c /\ t , y)) .
rl [Terminal] : G[t] (c.x) :: y => fail( (c /\ ~ t).x , y) .
```

Note that, unlike the theory in in §3, here we have some non-determinism. Under input $c.x$ (a constrained string) we have two possibilities: either the parser succeeds consuming c and adding to it the fact that the string must start with a character matching t; or, it fails and the string must start with a character in $\overline{dom(t)}$.

The other rules must be adjusted accordingly to carry the history of constraints accumulated so far. For instance, the rule for failures in choices becomes:

```
 rl [Choice] : CHOICE( fail(x, y) , G[e'] x':: y') =>
    G[e'] conj(x', concat(y, x)) :: y' .
```

This means that, if the choice failed, the second alternative e' must be evaluated on a constrained string with the additional information accumulated during the failure of the first alternative e. This is the purpose of the function *conj* that simply applies point-wise conjunction on the elements of the strings.

Example B.2 (Symbolic outputs). For each expression, we show the resulting final states of the form `ok(x,y)`. For readability, instead of `ok(x,y)` we write $y :: x$ (y was consumed and x was returned). Let's start with some simple cases:
- $[0-9]\ a$. Only one solution: `[0-9].a :: tt.`. This means that any valid string must start with $x \in dom([0-9])$, continue with a and then, any symbol is valid (for all x, $x \in dom(\texttt{tt})$).
- $!a\ b\ (c/d)$. Maude returns two solutions:

```
(~ a /\ b).c :: tt
(~ a /\ b). (~ c /\ d) ::tt
```

which further simplifies to `b.c::tt` and `b.d::tt`.
- $!a\ a$: `no solution`

Consider now the rule:

```
NUMBER <-  [0-9]+ . ("." . ( ! "." . [0-9])+)? .
```

For strings of at most 3 elements, we have 7 solutions:

```
[0-9]::(~ "." /\ ~ [0-9]).tt --- ex. "3ax"
[0-9].[0-9]::(~ "." /\ ~ [0-9]) --- ex. "23x"
[0-9].[0-9].[0-9] --- ex "123"
[0-9]::("." /\ ~ [0-9]) .".""--- ex "1.."
[0-9]::("." /\ ~ [0-9]).(~ "." /\ ~ [0-9])--- ex."1.a"
[0-9].[0-9]::("." /\ ~ [0-9]) --- ex. "12."
[0-9].("." /\ ~ [0-9]) .([0-9] /\ ~ ".") --- ex. "1.2"
```

After "::" we have the part of the string not consumed. The first output reads: the first character must be a number (and it is comsumed); then, the second character cannot be a digit, nor "."; and the last element can be any symbol. Then, e.g., the string "3ax" is accepted retuning the suffix "ax".

The PEG for $a^n b^n c^n$ in the end of §3 returns, as unique solution, the expected strings. As already noticed in [9], the grammar for the same language proposed in [8] is incorrect. For a length of 6, our tool finds the following solutions:

```
a a b b c c
a a a a a a
a a a a b c
```

The above symbolic strings are generated by using the search facilities in Maude:

```
search [n] G[e] tt^n =>* ok(x',y') such that c .
```

meaning "compute the first n states of the form `ok(x',y')` that satisfy the condition `c` and can be obtained by constraining the string $\texttt{tt}^n$". Note that rewriting is not enough since the theory is not longer deterministic and different paths must be considered in the `Terminal` rules. The `search` command implements a breadth-first search procedure and, therefore, no solution is lost. Needless to say that the search space may grow very quickly. Hence, either the condition c is used to filter some of the solutions (e.g., compute the symbolic strings that start with a digit) or specific/localized rules of the grammar are analyzed independently.

We shall use $\overset{\text{sPEG}}{\longrightarrow}$ to denote the induced rewrite relation using the rules above. For a constrained string $s = c_1.c_2.\cdots$ and a string $x = a_1.a_2\ldots$, we shall write $x \preceq s$ iff for each i, $a_i \in dom(c_i)$ (i.e., a_i is a legal character for c_i).

The next result shows that all the possible strings that can be generated from an expression e are covered by the symbolic output and, moreover, all instances of a symbolic output are indeed valid outputs for e.

Theorem B.3 (Adequacy). *For all G, e, x and y:*
Soundness: *If $G[e]\ xy \overset{PEG}{\rightsquigarrow} y$ then, there exists s_1 and s_2 s.t. $G[e]\ \texttt{tt}^{|xy|} \overset{sPEG}{\longrightarrow}!\ \texttt{ok}(s_2, s_1)$ and $xy \preceq s_1 :: s_2$.*
Completeness: *If $G[e]\ \texttt{tt}^n \overset{sPEG}{\longrightarrow}!\ \texttt{ok}(s_2, s_1)$ then, for all $xy \preceq s_1 :: s_2$, $G[e]\ xy \overset{PEG}{\rightsquigarrow} y$.*

Proof. We shall prove the correspondence between $\overset{\text{sPEG}}{\longrightarrow}$ and $\overset{\text{PEG}}{\longrightarrow}$. By Theorems 3.1 and 3.2 , the result extends to $\overset{\text{PEG}}{\rightsquigarrow}$.

We shall consider an alternative version of $\overset{\text{PEG}}{\longrightarrow}$ that, on failures, returns the non-consumed input. Hence, $G[e]\ xy \overset{\text{PEG}}{\longrightarrow} \texttt{fail}(y)$ means that x (a possible empty string) was consumed and y could not be recognized. This will simplify the arguments below. Our proof considers the failing cases, i.e., we shall prove the following:

- Soundness: If $G[e]\ xy \overset{\text{PEG}}{\longrightarrow}!\ y$ then, there exists s_1 and s_2 s.t. $G[e]\ \texttt{tt}^{|xy|} \overset{\text{sPEG}}{\longrightarrow} \texttt{ok}(s_2, s_1)$ and $xy \preceq s_1 :: s_2$. Moreover, if $G[e]\ xy \overset{\text{PEG}}{\longrightarrow}!\ \texttt{fail}(y)$ then $G[e]\ \texttt{tt}^{|xy|} \overset{\text{sPEG}}{\longrightarrow} \texttt{fail}(s_2, s_1)$ and $xy \preceq s_1 :: s_2$.
- Completeness: If $G[e]\ \texttt{tt}^n \overset{\text{sPEG}}{\longrightarrow} \texttt{ok}(s_2, s_1)$ then, for all $xy \preceq s_1 :: s_2$, $G[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$. Moreover, if $G[e]\ \texttt{tt}^n \overset{\text{sPEG}}{\longrightarrow} \texttt{fail}(s_2, s_1)$, for all $xy \preceq s_1 :: s_2$, $G[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}(y)$.

The main difference between $\overset{\text{sPEG}}{\longrightarrow}$ and $\overset{\text{PEG}}{\longrightarrow}$ is on the terminal rules. Namely, $\overset{\text{sPEG}}{\longrightarrow}$ considers two cases: either the string contains the needed terminal symbol t or it does not. In the second case, the derivation fails and adds the constraint $\sim t$.
Soundness. We proceed by induction on the length of the derivation of $G[e]\ x \overset{\text{PEG}}{\longrightarrow}!\ S$. In the base case, either $e = \epsilon$ or $e = a$. In the case of a terminal symbol, we have two possible outcomes: $G[a]\ x \overset{\text{PEG}}{\longrightarrow} x'$ (and $x = ax'$) or $G[a]\ x \overset{\text{PEG}}{\longrightarrow} \texttt{fail}(x)$ (and $x = bx'$ for $b \neq a$ or $x = \epsilon$). In the successful case, $n = |x| \geq 1$ and $G[a]\ \texttt{tt}^{|x|}$ has two possible outcomes: $\texttt{ok}(\texttt{tt}^{n-1}, a)$ and $\texttt{fail}((\sim\ a)\texttt{tt}^{n-1}, nil)$. Note that $ax' \preceq a\texttt{tt}^{|n-1|}$. The case when $x = bx'$ fails is considered in the second symbolic output: $bx' \preceq (\neg a)\texttt{tt}^{|n-1|}$ (note that, if $b \neq a$, then $b \in dom(\neg a)$). For the inductive case, we have several subcases. Consider the case when the derivation starts with `Choice`. Hence, $e = e_1/e_2$. There are two possible outcomes for e_1 and, by induction, both are instances of one of the symbolic outputs. The case when e_1 succeeds is immediate. Consider the case where $G[e_1]\ xyz \overset{\text{PEG}}{\longrightarrow}!\ \texttt{fail}(yz)$ and $G[e_2]\ xyz \overset{\text{PEG}}{\longrightarrow}!\ z$. By induction, $G[e_1]\ \texttt{tt}^n \overset{\text{sPEG}}{\longrightarrow} \texttt{fail}(s_y s_z, s_x)$ where s_y and s_z explain the failure of e_1 and s_x is in agreement with the part of the string consumed. We know that $x \preceq s_x$ and $yz \preceq s_y s_z$. By induction, we also know that $G[e_2]\ \texttt{tt}^n \overset{\text{sPEG}}{\longrightarrow} \texttt{ok}(\texttt{tt}^{|z|}, w_x w_y)$. However, the semantics executes e_2 on $\texttt{tt}^n \wedge s_x s_y s_z$ (and not on $\texttt{tt}^n$). Since the semantics only adds new constraints to the sequence of constraints, we can show that $G[e_2]\ \texttt{tt}^n \wedge s_x s_y s_z \overset{\text{sPEG}}{\longrightarrow} \texttt{ok}(s_z, (w_x \wedge s_x)(w_y \wedge s_y))$. Since $x \preceq s_x$ and $x \preceq w_x$, then $x \preceq s_x \wedge w_x$. Similarly for y and z and the result follows. The other cases are similar.
Completeness. We proceed by induction on the length of the derivation of $G[e]\ \texttt{tt}^n \overset{\text{sPEG}}{\longrightarrow}!\ S$. For the base case, consider a one-step derivation and assume that $S = \texttt{ok}(s_y, c)$. Hence, $e = t$, $c = t$ and $s_y = \texttt{tt}^{n-1}$. If $xy \preceq t :: s_y$ then $x \in dom(t)$ and clearly $G[t]\ xy \overset{\text{PEG}}{\longrightarrow} y$. If $S = \texttt{fail}((\sim\ t)s_y, nil)$ then $e = t$ and $s_y = \texttt{tt}^{n-1}$. If $xy \preceq (\sim\ t)s_y$, $x \in dom(\sim\ t)$ and then, $x \notin dom(t)$. This means that x does not match t and $G[t]\ xy \overset{\text{PEG}}{\longrightarrow} \texttt{fail}(xy)$.

For the inductive case, we have several subcases. Consider a derivation that starts with `Sequence`:

$G[e_1.e_2]\ \texttt{tt}^n \overset{\text{sPEG}}{\longrightarrow}!\ \texttt{ok}(s_z, s_x s_y)$. By the definition of $\overset{\text{sPEG}}{\longrightarrow}$, we know that $G[e_1]\ \texttt{tt}^n \overset{\text{sPEG}}{\longrightarrow}!\ \texttt{ok}(s_y s_z, s_x)$. Moreover,

$G[e_2]\ s_y s_z \overset{\text{sPEG}}{\longrightarrow}!\ \texttt{ok}(s_z, s_y)$. By induction, we deduce

$G[e_1]\ xyz \overset{\text{PEG}}{\longrightarrow}!\ yz$. From a similar observation about conjunction of sequences (as done in the proof of Soundness), we can also show that $G[e_2]\ yz \overset{\text{PEG}}{\longrightarrow}!\ z$. We then conclude

$G[e_1.e_2]\ xyz \overset{\text{PEG}}{\longrightarrow}!\ z$. The other cases are similar. □

References

[1] María Alpuente, Angel Cuenca-Ortega, Santiago Escobar, and José Meseguer. 2020. A partial evaluation framework for order-sorted equational programs modulo axioms. *J. Log. Algebraic Methods Program.* 110 (2020). https://doi.org/10.1016/j.jlamp.2019.100501

[2] Roberto Bruni and José Meseguer. 2006. Semantic Foundations for Generalized Rewrite Theories. *Theoretical Computer Science* 360, 1-3 (2006), 386–414. https://doi.org/10.1016/j.tcs.2006.04.012

[3] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott (Eds.). 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, Vol. 4350. Springer. https://doi.org/10.1007/978-3-540-71999-1

[5] Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.). 1996. *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*. Lecture Notes in Computer Science, Vol. 1110. Springer. https://doi.org/10.1007/3-540-61580-6

[6] Sérgio Queiroz de Medeiros, Gilney de Azevedo Alvez Junior, and Fabio Mascarenhas. 2020. Automatic syntax error reporting and recovery in parsing expression grammars. *Sci. Comput. Program.* 187 (2020), 102373. https://doi.org/10.1016/j.scico.2019.102373

[7] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn L. Talcott. 2020. Programming and symbolic computation in Maude. *J. Log. Algebraic Methods Program.* 110 (2020). https://doi.org/10.1016/j.jlamp.2019.100497

[8] 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, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 111–122. https://doi.org/10.1145/964001.964011

[9] Tony Garnock-Jones, Mahdi Eslamimehr, and Alessandro Warth. 2018. Recognising and Generating Terms using Derivatives of Parsing Expression Grammars. *CoRR* abs/1801.10490 (2018). arXiv:1801.10490 http://arxiv.org/abs/1801.10490

[10] Robert Grimm. 2006. Better extensibility through modular syntax. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 38–51. https://doi.org/10.1145/1133981.1133987

[11] Hugo Musso Gualandi and Roberto Ierusalimschy. 2018. Pallene: a statically typed companion language for lua. In *Proceedings of the XXII Brazilian Symposium on Programming Languages, SBLP 2018, Sao Carlos, Brazil, September 20-21, 2018*, Carlos Camarão and Martin Sulzmann (Eds.). ACM, 19–26. https://doi.org/10.1145/3264637.3264640

[12] Graham Hutton. 1992. Higher-order functions for parsing. *Journal of Functional Programming* 2, 3 (1992), 323–343. https://doi.org/10.1017/S0956796800000411

[13] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. 2013. Exception Handling for Error Reporting in Parsing Expression Grammars. In *Programming Languages - 17th Brazilian Symposium, SBLP 2013, Brasília, Brazil, October 3 - 4, 2013. Proceedings (Lecture Notes in Computer Science)*, André Rauber Du Bois and Phil Trinder (Eds.), Vol. 8129. Springer, 1–15. https://doi.org/10.1007/978-3-642-40922-6_1

[14] André Murbach Maidl, Fabio Mascarenhas, Sérgio Medeiros, and Roberto Ierusalimschy. 2016. Error reporting in Parsing Expression Grammars. *Sci. Comput. Program.* 132 (2016), 129–140. https://doi.org/10.1016/j.scico.2016.08.004

[15] Fabio Mascarenhas, Sérgio Medeiros, and Roberto Ierusalimschy. 2014. On the relation between context-free grammars and parsing expression grammars. *Sci. Comput. Program.* 89 (2014), 235–250. https://doi.org/10.1016/j.scico.2014.01.012

[16] Sérgio Medeiros and Fabio Mascarenhas. 2018. Syntax error recovery in parsing expression grammars. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir (Eds.). ACM, 1195–1202. https://doi.org/10.1145/3167132.3167261

[17] José Meseguer. 1992. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science* 96, 1 (1992), 73–155. https://doi.org/10.1016/0304-3975(92)90182-F

[18] José Meseguer. 2012. Twenty Years of Rewriting Logic. *The Journal of Logic and Algebraic Programming* 81, 7-8 (Oct. 2012), 721–781. https://doi.org/10.1016/j.jlap.2012.06.003

[19] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 189–195. https://doi.org/10.1145/2034773.2034801

[20] Kota Mizushima, Atusi Maeda, and Yoshinori Yamaguchi. 2010. Packrat parsers can handle practical grammars in mostly constant space. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010*, Sorin Lerner and Atanas Rountev (Eds.). ACM, 29–36. https://doi.org/10.1145/1806672.1806679

[21] Aaron Moss. 2020. Simplified Parsing Expression Derivatives. In *Language and Automata Theory and Applications - 14th International Conference, LATA 2020, Milan, Italy, March 4-6, 2020, Proceedings (Lecture Notes in Computer Science)*, Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron (Eds.), Vol. 12038. Springer, 425–436. https://doi.org/10.1007/978-3-030-40608-0_30

[22] Alexander A. Myltsev. 2019. parboiled2: a macro-based approach for effective generators of parsing expressions grammars in Scala. *CoRR* abs/1907.03436 (2019). arXiv:1907.03436 http://arxiv.org/abs/1907.03436

[23] Terence Parr and Kathleen Fisher. 2011. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 425–436. https://doi.org/10.1145/1993498.1993548

[24] Moeketsi Raselimo, Jan Taljaard, and Bernd Fischer. 2019. Breaking parsers: mutation-based generation of programs with guaranteed syntax errors. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira (Eds.). ACM, 83–87. https://doi.org/10.1145/3357766.3359542

[25] Roman R. Redziejowski. 2009. Applying Classical Concepts to Parsing Expression Grammar. *Fundam. Inform.* 93, 1-3 (2009), 325–336. https://doi.org/10.3233/FI-2009-0105

[26] Roman R. Redziejowski. 2015. Mouse: From Parsing Expressions to a Practical Parser. In *Concurrency Specification and Programming Workshop*.

[27] Grigore Rosu and Traian-Florin Serbanuta. 2014. K Overview and SIMPLE Case Study. *Electron. Notes Theor. Comput. Sci.* 304 (2014), 3–56. https://doi.org/10.1016/j.entcs.2014.05.002

[28] Vijay A. Saraswat and Martin C. Rinard. 1990. Concurrent Constraint Programming. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, Frances E. Allen (Ed.). ACM Press, 232–245. https://doi.org/10.1145/96709.96733