TRX: A FORMALLY VERIFIED PARSER INTERPRETER

ADAM KOPROWSKI AND HENRI BINSZTOK

MLstate, Paris, France
e-mail address: Adam.Koprowski@mlstate.com, Henri.Binsztok@mlstate.com

ABSTRACT. Parsing is an important problem in computer science and yet surprisingly little attention has been devoted to its formal verification. In this paper, we present TRX: a parser interpreter formally developed in the proof assistant Coq, capable of producing formally correct parsers. We are using parsing expression grammars (PEGs), a formalism essentially representing recursive descent parsing, which we consider an attractive alternative to context-free grammars (CFGs). From this formalization we can extract a parser for an arbitrary PEG grammar with the warranty of total correctness, i.e., the resulting parser is terminating and correct with respect to its grammar and the semantics of PEGs; both properties formally proven in Coq.

1. Introduction

Parsing is of major interest in computer science. Classically discovered by students as the first step in compilation, parsing is present in almost every program which performs data-manipulation.

For instance, the Web is built on parsers. The HyperText Transfer Protocol (HTTP) is a parsed dialog between the client, or browser, and the server. This protocol transfers pages in HyperText Markup Language (HTML), which is also parsed by the browser. When running web-applications, browsers interpret JavaScript programs which, again, begins with parsing. Data exchange between browser(s) and server(s) uses languages or formats like XML and JSON. Even inside the server, several components (for instance the trio made of the HTTP server Apache, the PHP interpreter and the MySQL database) often manipulate programs and data dynamically; all require parsers.

Parsing is not limited to compilation or the Web: securing data flow entering a network, signaling mobile communications, and manipulating domain specific languages (DSL) all require a variety of parsers.

The most common approach to parsing is by means of parser generators, which take as input a grammar of some language and generate the source code of a parser for that language. They are usually based on regular expressions (REs) and context-free grammars.

1998 ACM Subject Classification: D.3.4, D.2.4, F.3.1, F.4.2.

Key words and phrases: parser generation, formal verification, coq proof assistant, parsing expression grammars, recursive descent parsing.

* An extended abstract of this paper appeared in the Proceedings of the 19th European Symposium on Programming [KB10].
(CFGs), the latter expressed in Backus-Naur Form (BNF) syntax. They typically are able to
deal with some subclass of context-free languages, the popular subclasses including $LL(k)$,
$LR(k)$ and $LALR(k)$ grammars. Such grammars are usually augmented with semantic
actions that are used to produce a parse tree or an abstract syntax tree (AST) of the input.

What about correctness of such parsers? Yacc is the most widely used parser generator
and a mature program and yet the reference book about this tool [LMB92] devotes a whole
section (“Bugs in Yacc”) to discuss common bugs in its distributions. Furthermore, the
code generated by such tools often contains huge parsing tables making it near impossible
for manual inspection and/or verification. In the recent article about CompCert [Ler09],
an impressive project formally verifying a compiler for a large subset of C, the introduction
starts with a question “Can you trust your compiler?”. Nevertheless, the formal verification
starts on the level of the AST and does not concern the parser [Ler09, Figure 1]. Can you
trust your parser?

Parsing expression grammars (PEGs) [For04] are an alternative to CFGs, that have
recently been gaining popularity. In contrast to CFGs they are unambiguous and allow
easy integration of lexical analysis into the parsing phase. Their implementation is easy, as
PEGs are essentially a declarative way of specifying recursive descent parsers [Bur75]. With
their backtracking and unlimited look-ahead capabilities they are expressive enough to cover
all $LL(k)$ and $LR(k)$ languages as well as some non-context-free ones. However, recursive
descent parsing of grammars that are not $LL(k)$ may require exponential time. A solution
to that problem is to use memoization giving rise to packrat parsing and ensuring linear
time complexity at the price of higher memory consumption [AU72, For02b, For02a]. It is
not easy to support (indirect) left-recursive rules in PEGs, as they lead to non-terminating
parsers [WDM08].

In this paper we present TRX: a PEG-based parser interpreter formally developed
in the proof assistant Coq [Coq, BC04]. As a result, expressing a grammar in Coq allows
one, via its extraction capabilities [Let08], to obtain a parser for this grammar with total
correctness guarantees. That means that the resulting parser is terminating and correct
with respect to its grammar and the semantics of PEGs; both of those properties formally
proved in Coq. Moreover every definition and theorem presented in this paper has been
expressed and verified in Coq.

Our emphasis is on the practicality of such a tool. We perform two case studies: on
a simple XML format but also on the full grammar of the Java language. We present
benchmarks indicating that the performance of obtained parsers is reasonable. We also
sketch ideas on how it can be improved further, as well as how TRX could be extended into
a tool of its own, freeing its users from any kind of interaction with Coq and broadening its
applicability.

This work was carried out in the context of improving safety and security of OPA (One
Pot Application): an integrated platform for web development [RTS]. As mentioned above
parsing is of uttermost importance for web-applications and TRX is one of the components
in the OPA platform.

The remainder of this paper is organized as follows. We introduce PEGs in Section 2
and in Section 3 we extend them with semantic actions. Section 4 describes a method
for checking that there is no (indirect) left recursion in a grammar, a result ensuring that
parsing will terminate. Section 5 reports on our experience with putting the ideas of the
preceding sections into practice and implementing a formally correct parser interpreter
in Coq. Section 6 is devoted to a practical evaluation of this interpreter and contains case
TRX: A FORMALLY VERIFIED PARSER INTERPRETER

2. Parsing Expression Grammars (PEGs)

The content of this section is a different presentation of the results by Ford [For04]. For more details we refer to the original article. For a general overview of parsing we refer to, for instance, Aho, Seth & Ullman [ASU86].

PEGs are a formalism for parsing that is an interesting alternative to CFGs. We will formally introduce them along with their semantics in Section 2.1. PEGs are gaining popularity recently due to their ease of implementation and some general desirable properties that we will sketch in Section 2.2 while comparing them to CFGs.

2.1. Definition of PEGs.

Definition 2.1 (Parsing expressions). We introduce a set of parsing expressions, \( \Delta \), over a finite set of terminals \( V_T \) and a finite set of non-terminals \( V_N \). We denote the set of strings as \( S \) and a string \( s \in S \) is a list of terminals \( V_T \). The inductive definition of \( \Delta \) is given in Figure 1.

Later on we will present the formal semantics but for now we informally describe the language expressed by such parsing expressions.

- **Empty expression** \( \epsilon \) always succeeds without consuming any input.
- **Any-character** \([\cdot]\), a **terminal** \([a]\) and a **range** \([a−z]\) all consume a single terminal from the input but they expect it to be, respectively: an arbitrary terminal, precisely \( a \) and in the range between \( a \) and \( z \).
- **Literal** \( ["s"] \) reads a string (i.e., a sequence of terminals) \( s \) from the input.
- Parsing a **non-terminal** \( A \) amounts to parsing the expression defining \( A \).
- A **sequence** \( e_1; e_2 \) expects an input conforming to \( e_1 \) followed by an input conforming to \( e_2 \).
- A **choice** \( e_1/e_2 \) expresses a prioritized choice between \( e_1 \) and \( e_2 \). This means that \( e_2 \) will be tried only if \( e_1 \) fails.
- A zero-or-more (resp. one-or-more) repetition \( e* \) (resp. \( e+ \)) consumes zero-or-more (resp. one-or-more) repetitions of \( e \) from the input. Those operators are greedy, i.e., the longest match in the input, conforming to \( e \), will be consumed.
2.2. CFGs vs PEGs. The main differences between PEGs and CFGs are the following:

- the choice operator, e₁/e₂, is prioritized, i.e., e₂ is tried only if e₁ fails;
- the repetition operators, e⁺ and e⁺, are greedy, which allows to easily express “longest-match” parsing, which is almost always desired;
- syntactic predicates [PQ94], &e and !e, both of which consume no input and succeed if e, respectively, succeeds or fails. This effectively provides an unlimited look-ahead and, in combination with choice, limited backtracking capabilities.
An important consequence of the choice and repetition operators being deterministic (choice being prioritized and repetition greedy) is the fact that PEGs are unambiguous. We will see a formal proof of that in Theorem 3.5. This makes them unfit for processing natural languages, but is a much desired property when it comes to grammars for programming languages.

Another important consequence is ease of implementation. Efficient algorithms are known only for certain subclasses of CFGs and they tend to be rather complicated. PEGs are essentially a declarative way of specifying recursive descent parsers [Bur75] and performing this type of parsing for PEGs is straightforward (more on that in Section 5). By using the technique of packrat parsing [AU72, For02b], i.e., essentially adding memoization to the recursive descent parser, one obtains parsers with linear time complexity guarantees. The downside of this approach is high memory requirements: the worst-time space complexity of PEG parsing is linear in the size of the input, but with packrat parsing the constant of this correlation can be very high. For instance Ford reports on a factor of around 700 for a parser of Java [For02b].

CFGs work hand-in-hand with REs. The lexical analysis, breaking up the input into tokens, is performed with REs. Such tokens are subject to syntactical analysis, which is executed with CFGs. This split into two phases is not necessary with PEGs, as they make it possible to easily express both lexical and syntactical rules with a single formalism. We will see that in the following example.

Example 2.3 (PEG for simple mathematical expressions). Consider a PEG for simple mathematical expressions over 5 non-terminals: $V_N := \{\text{ws, number, term, factor, expr}\}$ with the following productions ($P_{\text{expr}}$ function from Definition 2.2):

- $\text{ws} ::= ([\w] / [\t])^*$
- $\text{number} ::= [0-9]+$
- $\text{term} ::= \text{ws number ws} / \text{ws [(} expr [)] ws []} expr [{] ws$
- $\text{factor} ::= \text{term [}* factor / term}$
- $\text{expr} ::= \text{factor [+] expr / factor}$

Please note that in this and all the following examples we write the sequence operator $e_1; e_2$ implicitly as $e_1 e_2$. The starting production is $v_{\text{start}} ::= \text{expr}$.

First, let us note that lexical analysis is incorporated into this grammar by means of the $\text{ws}$ production which consumes all white-space from the beginning of the input. Allowing white-space between “tokens” of the grammar comes down to placing the call to this production around the terminals of the grammar. If one does not like to clutter the grammar with those additional calls then a simple solution is to re-factor all terminals into separate productions, which consume not only the terminal itself but also all white-space around it.

Another important observation is that we made addition (and also multiplication) right-associative. If we were to make it, as usual, left-associative, by replacing the rule for $\text{expr}$ with:

- $\text{expr} ::= \text{expr [+] factor / factor}$

then we get a grammar that is left-recursive. Left-recursion (also indirect or mutual) is problematic as it leads to non-terminating parsers. We will come back to this issue in Section 4.

$\triangleright$
PEGs can also easily deal with some common idioms often encountered in practical grammars of programming languages, which pose a lot of difficulty for CFGs, such as modular way of handling reserved words of a language and a “dangling” else problem — we present them on two examples and refer for more details to Ford [For02a, Chapter 2.4].

**Example 2.4** (Reserved words). One of the difficulties in tokenization is that virtually every programming language has a list of *reserved words*, which should not be accepted as identifiers. PEGs allow an elegant pattern to deal with this problem:

```
identifier ::= !reserved letter+ ws
reserved ::= IF / ... IF ::= ["if"] !letter ws
```

The rule `identifier` for identifiers reads a non-empty list of letters but only after checking, with the not-predicate, that there is no reserved word at this position. The rules for the reserved words ensure that it is not followed by a letter (“ifs” is a valid identifier) and consume all the following white space. In this example we only presented a single reserved word “if” but adding a new word requires only adding a rule similar to `IF` and extending the choice in `reserved`.

**Example 2.5** (“Dangling” else). Consider the following part of a CFG for the C language:

```
stmt ::= IF ( expr ) stmt |
       IF ( expr ) stmt ELSE stmt |
     ...
```

According to this grammar there are two possible readings of a statement

```
if (e1) if (e2) s₁ else s₂
```

as the “else s₂” branch can be associated either with the outer or the inner if. The desired way to resolve this ambiguity is usually to bind this else to the innermost construct. This is exactly the behavior that we get by converting this CFG to a PEG by replacing the symmetrical choice operator “|” of CFGs with the prioritized choice of PEGs “/”.<

### 3. Extending PEGs with Semantic Actions

#### 3.1. XPEGs: Extended PEGs

In the previous section we introduced parsing expressions, which can be used to specify which strings belong to the grammar under consideration. However the role of a parser is not merely to recognize whether an input is correct or not but also, given a correct input, to compute its representation in some structured form. This is typically done by extending grammar expressions with *semantic values*, which are a representation of the result of parsing this expression on (some) input and by extending a grammar with *semantic actions*, which are functions used to produce and manipulate the semantic values. Typically a semantic value associated with an expression will be its parse tree so that parsing a correct input will give a *parse tree* of this input. For programming languages such parse tree would represent the AST of the language.

In order to deal with this extension we will replace the simple type of parsing expressions $\Delta$ with a family of types $\Delta_\alpha$, where the index $\alpha$ is a type of the semantic value associated with the expression. We also compositionally define default semantic values for all types...
of expressions and introduce a new construct: coercion, $e\mapsto f$, which converts a semantic value $v$ associated with $e$ to $f(v)$.

Borrowing notations from Coq we will use the following types:

- **Type** is the universe of types.
- **True** is the singleton type with a single value $I$.
- **char** is the type of machine characters. It corresponds to the type of terminals $\mathcal{V}_T$, which in concrete parsers will always be instantiated to char.
- **list $\alpha$** is the type of lists of elements of $\alpha$ for any type $\alpha$. Also string ::= list char.
- **$\alpha_1 \ast \ldots \ast \alpha_n$** is the type of $n$-tuples of elements $(a_1, \ldots, a_n)$ with $a_1 \in \alpha_1, \ldots, a_n \in \alpha_n$ for any types $\alpha_1, \ldots, \alpha_n$. If $v$ is an $n$-tuple then $v_i$ is its $i$'th projection.
- **option $\alpha$** is the type optionally holding a value of type $\alpha$, with two constructors None and Some $v$ with $v : \alpha$.

**Definition 3.1** (Parsing expressions with semantic values). We introduce a set of **parsing expressions with semantic values**, $\Delta_\alpha$, as an inductive family indexed by the type $\alpha$ of semantic values of an expression. The typing rules for $\Delta_\alpha$ are given in Figure 3.

Note that for the choice operator $e_1/e_2$ the types of semantic values of $e_1$ and $e_2$ must match, which will sometimes require use of the coercion operator $e\mapsto f$.

Let us again see the derived operators and their types, as we need to insert a few coercions:

$$
\begin{align*}
[a\neg z] : \Delta_{\text{char}} &::= [a] / \ldots / [z] \\
["s"] : \Delta_{\text{string}} &::= [s_0]; \ldots; [s_n] \mapsto \text{tuple2str} \\
e+ : \Delta_{\text{list } \alpha} &::= e; e* \mapsto \lambda x. x_1 :: x_2 \\
e? : \Delta_{\text{option } \alpha} &::= e \mapsto \lambda x. \text{Some } x \\
/ e &::= \lambda x. \text{None} \\
&\& e : \Delta_{\text{True}} &::= !! e
\end{align*}
$$

where tuple2str$(c_1, \ldots, c_n) = [c_1; \ldots; c_n]$.

The definition of an extended parsing expression grammar (XPEG) is as expected (compare with Definition 2.1).
• $P_{\text{exp}}$ is the interpretation of the productions of the grammar, i.e., $P_{\text{exp}} : \forall A : \forall v : \nu_{\mathcal{N}} s \cdot P_{\text{type}}(A)$.
• $v_{\text{start}}$ is the start production, $v_{\text{start}} \in \nu_{\mathcal{N}}$.

We extended the semantics of PEGs from Figure 2 to semantics of XPEGs in Figure 4.

Example 3.3 (Simple mathematical expressions). Let us extend the grammar from Example 2.3 with semantic actions. The grammar expressed mathematical expressions and we attach semantic actions evaluating those expressions, hence obtaining a very simple calculator.

It often happens that we want to ignore the semantic value attached to an expression. This can be accomplished by coercing this value to $I$, which we will abbreviate by $e [\mapsto] \lambda x . I$.

\[
\begin{align*}
\text{ws} & ::= \left[ \mathbf{t} \right] / \left[ \mathbf{t} \right]^* \quad [\mathbf{t}] \quad \text{digListToNat} \\
\text{number} & ::= \left[ 0-9 \right] + \quad [\mapsto] \lambda x . x_2 \\
\text{term} & ::= \text{ws} \text{ number} \text{ ws} \quad [\mapsto] \lambda x . x_3 \\
& /\text{ ws} \left[ \mathbf{t} \right] \text{ expr} \left[ \mathbf{t} \right] \text{ ws} \quad [\mapsto] \lambda x . x_3 \\
\text{factor} & ::= \text{term} \left[ \mathbf{t} \right] \text{ factor} \quad [\mapsto] \lambda x . x_1 \times x_3 \\
& /\text{ term} \\
\text{expr} & ::= \text{factor} \left[ + \right] \text{ expr} \quad [\mapsto] \lambda x . x_1 + x_3 \\
& /\text{ factor}
\end{align*}
\]

where digListToNat converts a list of digits to their decimal representation and $x_i$ in the productions is the $i$-th projection of the vector of values $x$, resulting from parsing a sequence.

This grammar will associate, as expected, the semantical value 36 with the string "(1+2) * (3 * 4)". Of course in practice instead of evaluating the expression we would usually write semantic actions to build a parse tree of the expression for later processing.
3.2. Meta-properties of (X)PEGs. Now we will present some results concerning seman-
tics of (X)PEGs. They are all variants of results obtained by Ford [For04], only now we
extend them to XPEGs. First we prove that, as expected, the parsing only consumes a
prefix of a string.

**Theorem 3.4.** If \((e, s) \xrightarrow{m} \sqrt{v}_s\) then \(s'\) is a suffix of \(s\).

**Proof.** Induction on the derivation of \((e, s) \xrightarrow{m} \sqrt{v}_s\) using transitivity of the prefix property
for sequence and repetition cases. □

As mentioned earlier, (X)PEGs are unambiguous:

**Theorem 3.5.** If \((e, s) \xrightarrow{m_1} r_1\) and \((e, s) \xrightarrow{m_2} r_2\) then \(m_1 = m_2\) and \(r_1 = r_2\).

**Proof.** Induction on the derivation \((e, s) \xrightarrow{m_1} r_1\) followed by inversion of \((e, s) \xrightarrow{m_2} r_2\). All cases
immediate from the semantics of XPEGs. □

We wrap up this section with a simple property about the repet ition operator, that we
will need later on. It states that the semantics of a repetition expression \(e^*\) is not defined
if \(e\) succeeds without consuming any input.

**Lemma 3.6.** If \((e, s) \xrightarrow{m} \sqrt{v}_s\) then \((e^*, s) \not\xrightarrow{r}\) for all \(r\).

**Proof.** Assume \((e, s) \xrightarrow{m} \sqrt{v}_s\) and \((e^*, s) \xrightarrow{n} \sqrt{v}_s'\) for some \(n, vs\) and \(s'\) (we cannot have
\((e^*, s) \xrightarrow{n} \perp\) as \(e^*\) never fails). By the first rule for repetition \((e^*, s) \xrightarrow{m+n+1} \sqrt{v}_s'\) which
contradicts the second assumption by Theorem 3.5. □

4. Well-formedness of PEGs

We want to guarantee total correctness for generated parsers, meaning they must be correct
(with respect to PEGs semantics) and terminating. In this section we focus on the latter
problem. Throughout this section we assume a fixed PEG \(G\).

4.1. Termination problem for XPEGs. Ensuring termination of a PEG parser essen-
tially comes down to two problems:
• termination of all semantic actions in \(G\) and
• completeness of \(G\) with respect to PEGs semantics.

As for the first problem it means that all \(f\) functions used in coercion operators \(e[\rightarrow]f\)
in \(G\), must be terminating. We are going to express PEGs completely in Coq (more on that
in Section 5) so for our application we get this property for free, as all Coq functions are
total (hence terminating).

Concerning the latter problem, we must ensure that the grammar \(G\) under consideration
is complete, i.e., it either succeeds or fails on all input strings. The only potential source of
incompleteness of \(G\) is (mutual) left-recursion in the grammar.

We already hinted at this problem in Example 2.3 with the rule:

\[
\text{expr ::= expr [+] factor / factor}
\]
Recursive descent parsing of expressions with this rule would start with recursively calling a function to parse expression on the same input, obviously leading to an infinite loop. But not only direct left recursion must be avoided. In the following rule:

\[
A ::= B / C !D A
\]

a similar problem occurs provided that \(B\) may fail and \(C\) and \(D\) may succeed, the former without consuming any input.

While some techniques to deal with left-recursive PEGs have been developed recently [WDM08], we choose to simply reject such grammars. In general it is undecidable whether a PEG grammar is complete, as it is undecidable whether the language generated by \(G\) is empty [For04].

While in general checking grammar completeness is undecidable, we follow Ford [For04] to develop a simple syntactical check for well-formedness of a grammar, which implies its completeness. This check will reject left-recursive grammars even if the part with left-recursion is unreachable in the grammar, but from a practical point of view this is hardly a limitation.

### 4.2. PEG Analysis

We define the expression set of \(G\) as:

\[
E(G) = \{ e' \mid e' \sqsubseteq e, e \in P_{\exp}(A), A \in \mathcal{V}_N \}
\]

where \(\sqsubseteq\) is a (non-strict) sub-expression relation on parsing expressions.

We define three groups of properties over parsing expressions:

- “0”: parsing expression can succeed without consuming any input,
- “> 0”: parsing expression can succeed after consuming some input and
- “⊥”: parsing expression can fail.

We will write \(e \in \mathbb{P}_0\) to indicate that the expression \(e\) has property “0” (similarly for \(\mathbb{P}_{>0}\) and \(\mathbb{P}_\bot\)). We will also write \(e \in \mathbb{P}_{\geq 0}\) to denote \(e \in \mathbb{P}_0 \lor e \in \mathbb{P}_{>0}\). We define inference rules for deriving those properties in Figure 5.

Figure 5: Deriving grammar properties.

| \(e \in \mathbb{P}_0\) | \(\epsilon \in \mathbb{P}_0\) | \(A \in \mathcal{V}_N\) | \(P_{\exp}(A) \in \mathbb{P}_*\) | \(a \in \mathcal{V}_T\) | \(a \in \mathbb{P}_*\) | \(e* \in \mathbb{P}_{>0}\) |
| \(\star \in \{0, >, 0, \bot\}\) | \(\epsilon \in \mathbb{P}_0\) | \(P_{\exp}(A) \in \mathbb{P}_*\) | \(e* \in \mathbb{P}_{>0}\) | \(\epsilon \in \mathbb{P}_0\) | \(e \in \mathbb{P}_\bot\) | \(e* \in \mathbb{P}_{>0}\) |
| \((e_1 \in \mathbb{P}_{>0} \land e_2 \in \mathbb{P}_{\geq 0}) \lor (e_1 \in \mathbb{P}_{>0} \land e_2 \in \mathbb{P}_{>0})\) | \(e_1; e_2 \in \mathbb{P}_{>0}\) | \(e_1/e_2 \in \mathbb{P}_0\) | \((e_1 \in \mathbb{P}_{>0} \land e_2 \in \mathbb{P}_{>0})\) | \(e_1/e_2 \in \mathbb{P}_{>0}\) | \(e \in \mathbb{P}_{>0}\) | \(e \in \mathbb{P}_\bot\) |
| \(e_1 \in \mathbb{P}_0 \lor (e_1 \in \mathbb{P}_\bot \land e_2 \in \mathbb{P}_0)\) | \(e_1/e_2 \in \mathbb{P}_{>0}\) | \(e_1/e_2 \in \mathbb{P}_0\) | \(e \in \mathbb{P}_0 \lor (e_1 \in \mathbb{P}_{>0} \land e_2 \in \mathbb{P}_{>0})\) | \(e_1/e_2 \in \mathbb{P}_0\) | \(e \in \mathbb{P}_0 \lor (e_1 \in \mathbb{P}_{>0} \land e_2 \in \mathbb{P}_{>0})\) | \(e \in \mathbb{P}_{>0}\) | \(e \in \mathbb{P}_\bot\) |
those property sets monotonically and they are bounded by the finite set $E(G)$. We summarize the semantics of those properties in the following lemma:

**Lemma 4.1 ([For04]).** For arbitrary $e \in \Delta$ and $s \in S$:

- if $(e, s) \xrightarrow{n} \top_s$ then $e \in P_0$,
- if $(e, s) \xrightarrow{n} \top_s$, and $|s'| < |s|$ then $e \in P_{>0}$ and
- if $(e, s) \xrightarrow{n} \bot$ then $e \in P_\bot$.

**Proof.** Induction over $n$. All cases easy by the induction hypothesis and semantical rules of XPEGs, except for $e*$ which requires use of Lemma 3.6.

Those properties will be used for establishing well-formedness of a PEG, as we will see in the following section. It is worth noting here that checking whether $e \in P_0$ also plays a crucial role in the formal approach to parsing developed by Danielsson [Dan10] (we will say more about his work in Section 7).

It is also interesting to consider such a simplified analysis in our setting, i.e., only considering $e \in P_0$ and collapsing derivations of Figure 5 by assuming $e \in P_{>0}$ and $e \in P_\bot$ hold for every expression $e$. At first it seems we would lose some precision by such an over-approximation as for instance that would lead us to conclude $!\epsilon \in P_0$, whereas in fact this expression can never succeed without consuming any input (as, quite simply, it can never succeed). As we will see soon this would lead us to reject a valid definition:

$$A ::= !\epsilon \; ; \; A$$

However, this definition of $A$ is not very interesting as it always fails. In fact, we conjecture that the differences occur only in such degenerated cases and that in practice such a simplified analysis would be as efficient as that of [For04].

### 4.3. PEG well-formedness

Using the semantics of those properties of parsing expression we can perform the completeness analysis of $G$. We introduce a set of well-formed expressions $WF$ and again iterate from an empty set by using derivation rules from Figure 6 over $E(G)$ until reaching a fix-point.

We say that $G$ is well-formed if $E(G) = WF$. We have the following result:

**Theorem 4.2 ([For04]).** If $G$ is well-formed then it is complete.

**Proof.** We will say that $(e, s)$ is complete iff $\exists_{n,r} (e, s) \xrightarrow{\omega} r$. So we have to prove that $(e, s)$ is complete for all $e \in E(G)$ and all strings $s$. We proceed by induction over the length of the string $s$ ($IH_{out}$), followed by induction on the depth of the derivation tree of $e \in WF$ ($IH_{in}$). So we have to prove correctness of a one step derivation of the well-formedness property (Figure 6) assuming that all expressions are total on shorter strings. The interesting cases are:
• For a sequence \( e_1; e_2 \) if \( e_1; e_2 \in \text{WF} \) then \( e_1 \in \text{WF} \), so \( (e_1, s) \) is complete by \( \text{IH}_\text{in} \). If \( e_1 \) fails then \( e_1; e_2 \) fails. Otherwise \( (e_1, s) \overset{s'}{\rightarrow} \). If \( s = s' \) then \( e_1 \in \mathbb{P}_0 \) (Lemma 4.1) and hence \( e_2 \in \text{WF} \) and \( (e_2, s') \) is complete by \( \text{IH}_\text{in} \). If \( s \neq s' \) then \( |s'| < |s| \) (Theorem 3.4) and \( (e_2, s') \) is complete by \( \text{IH}_\text{out} \). Either way \( (e_2, s') \) is complete and we conclude by semantical rules for sequence.

• For a repetition \( e^* \), \( e \in \text{WF} \) gives us completeness of \( (e, s) \) by \( \text{IH}_\text{in} \). If \( e \) fails then we conclude by the base rule for repetition. Otherwise \( (e^*, s) \overset{s'}{\rightarrow} \) with \( |s'| < |s| \) as \( e \notin \mathbb{P}_0 \). Hence we get completeness of \( (e^*, s') \) by \( \text{IH}_\text{out} \) and we conclude with the inductive rule for repetition.

5. Formally Verified XPEG interpreter

In this Section we will present a Coq implementation of a parser interpreter. This task consists of formalizing the theory of the preceding sections and, based on this, writing an interpreter for well-formed XPEGs along with its correctness proofs. The development is too big to present it in detail here, but we will try to comment on its most interesting aspects.

We will describe how PEGs are expressed in Coq in Section 5.1, comment on the procedure for checking their well-formedness in Section 5.2 and describe the formal development of an XPEG interpreter in Section 5.3.

5.1. Specifying XPEGs in Coq. XPEGs in Coq are a simple reflection of Definition 3.2. They are specified over a finite enumeration of non-terminals (corresponding to \( \mathcal{V}_N \)) with their types (\( P_{\text{type}} \)):

\[
\text{Parameter} \quad \text{prod} : \text{Enumeration}.
\]

\[
\text{Parameter} \quad \text{prod}_\text{type} : \text{prod} \rightarrow \text{Type}.
\]

Building on that we define:

• \( p\text{exp} \): un-typed parsing expressions, \( \Delta \), and
• \( P\text{Exp} \): their typed variant, \( \Delta^\alpha \), which follows the typing discipline from Figure 3.

We present both definitions side by side:

\[
\text{Inductive} \quad p\text{exp} : \text{Type} :=
\]

\[
| \text{empty} \]
\[
| \text{anyChar} \]
\[
| \text{terminal} \ (a : \text{char}) \]
\[
| \text{range} \ (a z : \text{char}) \]
\[
| \text{nonTerminal} \ (p : \text{prod}) \]
\[
| \text{seq} \ (e_1 e_2 : p\text{exp}) \]
\[
| \text{choice} \ (e_1 e_2 : p\text{exp}) \]
\[
| \text{star} \ (e : p\text{exp}) \]
\[
| \text{not} \ (e : p\text{exp}) \]
\[
| \text{id} \ (e : p\text{exp}).
\]

\[
\text{Inductive} \quad P\text{Exp} : \text{Type} \rightarrow \text{Type} :=
\]

\[
| \text{Empty} : P\text{Exp True} \]
\[
| \text{AnyChar} : P\text{Exp char} \]
\[
| \text{Terminal} : \text{char} \rightarrow P\text{Exp char} \]
\[
| \text{Range} : \text{char} \ast \text{char} \rightarrow P\text{Exp char} \]
\[
| \text{NonTerminal} : \forall p, P\text{Exp (prod}_\text{type} \ p) \]
\[
| \text{Seq} : \forall A B, P\text{Exp A} \rightarrow P\text{Exp B} \rightarrow P\text{Exp (A \ast B)} \]
\[
| \text{Choice} : \forall A, P\text{Exp A} \rightarrow P\text{Exp A} \rightarrow P\text{Exp A} \]
\[
| \text{Star} : \forall A, P\text{Exp A} \rightarrow P\text{Exp (list A)} \]
\[
| \text{Not} : \forall A, P\text{Exp A} \rightarrow P\text{Exp True} \]
\[
| \text{Action} : \forall A B, P\text{Exp A} \rightarrow (A \rightarrow B) \rightarrow P\text{Exp B}.
\]

Those definitions are straight-forward encodings of Definitions 2.1 and 3.1. We implemented the range operator \( [a\cdots z] \) as a primitive, as in practice it occurs frequently in parsers and
implementing it as a derived operation by a choice over all the characters in the range is inefficient. That means that in the formalization we had to extend the semantics of Figure 4 with this operator, in a straightforward way.

It is worth noting here that \( PExp \) is large, in terms of Coq universe levels, as its index lives in \( \text{Type} \). We never work with propositional equality of types, so the constraints on types used in constructors of \( PExp \), come only from the inductive definition itself. In particular, \( PExp \) must live at a higher universe level than any type used in its constructors.

For “regular” use of our parsing machinery this should pose no problems. However, should we want to develop some higher-order grammars (grammars that upon parsing return another grammar) we would very soon run into Coq’s Universe Inconsistency problems. In fact higher-order grammars are not expressible in our framework anyway, due to the use of Coq’s module system. We will return to this issue in Section 8.

With \( pexp \) and \( PExp \) in place we continue by defining, in an obvious way, conversion functions from one structure to the another.

\[
\text{Fixpoint } pexp\_project \ T \ (e : PExp \ T) : pexp \ := \ \{ \ldots \} \\
\text{Fixpoint } pexp\_promote \ (e : pexp) : PExp \ True \ := \ \{ \ldots \}
\]

Conversion from \( PExp \) to \( pexp \) simply erases types and maps Actions to dummy constructor \( id \). Conversion in the other direction maps to expressions of a singleton type \( True \), inserting, where needed, type coercions using Action operator.

To complete the definition of XPEG grammar, Definition 3.2, we declare definitions of non-terminals (\( P\exp \)) and the starting production (\( \nu_{\text{start}} \)) as:

Parameter production : \( \forall \ p : \text{prod}, PExp (\text{prod}\_\text{type} \ p) \).

Parameter start : prod.

There are two observations that we would like to make at this point. First, by means of the above embedding of XPEGs in Coq, every such XPEG is well-defined (though not necessarily well-formed). In particular there can be no calls to undefined non-terminals and the conformance with the typing discipline from Figure 3 is taken care of by the type-checker of Coq.

Secondly, thanks to the use of Coq’s mechanisms, such as notations and coercions, expressing an XPEG in Coq is still relatively easy as we will see in the following example.

**Example 5.1.** Figure 7 presents a precise Coq rendering of the productions of the XPEG grammar from Example 3.3. It is not much more verbose than the original example. Each \( Pi \) function corresponds to \( i \)’th projection and they work with arbitrary \( n \)-tuples thanks to the type-class mechanism.

5.2. **Checking well-formedness of an XPEG.** To check well-formedness of XPEGs we implement the procedure from Section 4. It is worth noting that the function to compute XPEG properties, by iterating the derivation rules of Figure 5 until reaching a fix-point, is not structurally recursive. Similarly for the well-formedness check with rules from Figure 6. Fortunately the Program feature [Soz07] of Coq makes specifying such functions much easier. We illustrate it on the well-formedness check (computing properties is analogous).

We begin by one-step well-formedness derivation corresponding to Figure 6.

**Definition** \( \text{wf} \_\text{analyse} \ (exp : pexp) (wf : PES.\ t) : \text{bool} := \)

match exp with
Program Definition production p :=
match p return PExp (prod_type p) with
| ws ⇒ (" "/"\t") [∗ ] [ ] [# ]
| number ⇒ ["0"−−"9"] [+] [ → ] digListToRat
| term ⇒ ws; number; ws [ → ] (λ v ⇒ P2 v)
| / ws; "("; expr; ")"; ws [ → ] (λ v ⇒ P3 v)
| factor ⇒ term; "*"; factor [ → ] (λ v ⇒ P1 v * P3 v)
| expr ⇒ factor; "+"; expr [ → ] (λ v ⇒ P1 v + P3 v)
| / factor
end.

This function take a set of well-formed expressions computed so far (PES standing for “parsing expression set”) and an expression exp and returns true iff exp should also be consider well-formed, according to the derivation system of Figure 6. Here gp is the set of global properties computed following the procedure of Section 4.2 (again, we do not show the code here, as that procedure is very analogous to the inference of well-formedness, that we describe). Hence e − [gp] → 0 should be read as e ∈ P0 and is_wf is an abbreviation for set membership, i.e.:

Definition is_wf : pexp → PES.t → bool := PES.mem.

With that in place we continue with a simple function that extends the set of well-formed expressions with the one being considered now, in case it was established to be well-formed by invocation of wf_analyse and otherwise leaves this set unchanged.

Definition wf_analyse_exp (exp : pexp) (wf : PES.t) : PES.t :=
if wf_analyse exp wf then PES.add exp wf else wf.

Now the one step derivation over all expressions E(ː), represented by the constant grammarExpSet below, can be realized as a simple fold operation using the above function:

Definition wf_derive (wf : PES.t) : PES.t :=
PES.fold wf_analyse_exp grammarExpSet wf.
Now, the complete analysis is a fixpoint of applying one-step derivation $\text{wf}_\text{derive}$.

**Program Fixpoint** $\text{wf}_\text{compute}$ ($\text{wf} : \text{WFset}$) \{\text{measure} $\text{wf}_\text{measure} \text{wf}$ \} : $\text{WFset} ::= \\
\text{let} \text{wf}' := \text{wf}_\text{derive} \text{wf} \text{in} \\
\text{if} \ PES.\text{equal} \text{wf} \text{wf}' \\text{then} \text{wf} \text{else} \text{wf}_\text{compute} \text{wf}'$.

Here $\text{WFset}$ is a set of well-formed expressions:

**Definition** $\text{WFset} := \{ e : \text{PES} \cdot t \mid \text{wf}_\text{prop} e \}$

where $\text{wf}_\text{prop}$ is a predicate capturing well-formedness of an expression.

The main difficulty here is that $\text{wf}_\text{compute}$ is not structurally recursive. However, we can construct a measure (into $\mathbb{N}$) that will decrease along recursive calls as:

$$\text{wf}_\text{measure} ::= |E(G)| - |\text{wf}|$$

Now we can prove this procedure terminating, as the set of well-formed expressions is growing monotonically and is contained in $E(G)$:

$$\text{wf} \subseteq \text{wf}_\text{derive} \text{wf}$$

$$\text{wf} \subseteq E(G) \implies \text{wf}_\text{derive} \text{wf} \subseteq E(G)$$

The Program feature \[Soz07\] of Coq, is very helpful in expressing such non structurally recursive functions, as well as in general programming with dependent types. The downside of Program is that it inserts type casts, making reasoning about such functions more difficult. This can be usually overcome with the use of sigma-types capturing the function specification ($\text{wf}_\text{prop}$ in our example) together with its return value. This style of programming seems to be particularly well suited when working with Program.

Finally we obtain the set of well-formed expressions of a grammar by iterating to a fix-point, starting with an empty set:

**Program Definition** $\text{WFexps} : \text{PES} \cdot t := \text{wf}_\text{compute} \text{PES} \cdot \text{empty}$.

A grammar expression $\text{exp}$ is well-formed if it belongs to this set

**Definition** $\text{WF} (\text{exp} : \text{pexp}) : \text{Prop} := \text{PES} \cdot \text{In} \text{exp} \text{ WFexps}$.

And a grammar is well-formed if all its expressions are well-formed:

**Definition** $\text{grammar} \cdot \text{WF} : \text{Prop} := \text{grammarExpSet} [=] \text{WFexps}$.

Above we presented a complete code of the well-formedness analysis (Section 4.3), excluding the inference of properties (Section 4.2). Naturally, every of those functions is accompanied with some lemmas stating its correctness and their proofs. Those proofs, with Ltac definitions used to discard them, constitute roughly 4-5x the size of the definitions. This factor is so low thanks to heavy use of Ltac automation in the proofs; the proof style advocated by Chlipala \[Chl09\], which we, eventually, learned to embrace fully.

Our interpreter (more on it in the following section) will work on XPEGs, not on PEGs. However, the termination analysis sketched above considers un-typed parsing expressions $\text{pexp}$, obtained by projecting XPEGs expressions (with $\text{pexp}_\text{project}$). The reason is two-fold.

Firstly, semantic actions are embedded in Coq’s programming language and hence are terminating and have no influence on the termination analysis of the grammar. Hence
a termination of the parser on expression \( e : PExp \ T \) is immediate from termination of \( \text{pexp\_project} \ e : \text{pexp} \).

Secondly, the well-formedness procedure presented above needs to maintain a set of parsing expressions (\( \text{WFset} \)) and for that we need a decidable equality over parsing expressions. Equality over \( \Delta_\alpha \) is not decidable, as, within coercion operator \( e[\rightarrow]f \) they contain arbitrary functions \( f \).

An alternative approach would be to consider \( \text{WFset} \) modulo an equivalence relation on parsing expressions coarser than the syntactic equality, which would ignore \( f \) components in \( e[\rightarrow]f \) coercions. That would avoid formalization of the un-typed structure \( \text{pexp} \) altogether for the price of reasoning with dependently typed \( PExp \)'s in the well-formedness analysis.

5.3. A formal interpreter for XPEGs. For the development of a formal interpreter for XPEGs we used the \texttt{ascii} type of Coq for the set of terminals \( \mathcal{V}_T \). The string type from the standard library of Coq is isomorphic to lists of characters. In its place we just used a list of characters, in order to be able to re-use a rich set of available functions over lists.

First let us define the result of parsing an expression \( PExp \ T \) on some string:

\[
\text{Inductive ParsingResult (} T : \text{Type}) : \text{Type} := \\
| \text{PR\_fail} \\
| \text{PR\_ok} \ s \ v
\]

i.e., a parsing can either fail (\( \text{PR\_fail} \)) or succeed (\( \text{PR\_ok} \ s \ v \)), in which case we obtain a suffix \( s \) that remains to be parsed and an associated semantic value \( v \).

Now after requiring a well-formed grammar, interpreter can be defined as a function with the following header:

\[
\text{Variable GWF : grammar_\text{WF}.} \\
\text{Program Fixpoint parse (} T : \text{Type}) (e : PExp \ T \ | is\_grammar\_exp e) \ (s : \text{string}) \\
\{ \text{measure} \ \ (e,s) \succ \} : \{ r : \text{ParsingResult} \ T \ \exists \ n, \ [e,s] \Rightarrow [n,r] \}
\]

So this function takes three arguments (the first one implicit):
- \( T \): a type of the result of parsing (\( \alpha \)),
- \( e \): a parsing expression of type \( T (\Delta_\alpha) \), with a proof (\( is\_grammar\_exp e \)) that it belongs to the grammar \( \mathcal{G} \) (which in turn is checked beforehand to be well-formed) and
- \( s \): a string to be parsed.

The last line in the above header describes the type of the result of this function, where \( [e,s] \Rightarrow [n,r] \) is the expected encoding of the semantics from Figure 4 and corresponds to \( (e,s) \leadsto r \). So the \texttt{parse} function produces the parsing result \( r \) (either \( \bot \) or \( \sqrt{v} \), with \( v : T \)), such that \( (e,s) \leadsto r \) for some \( n \), \( i.e., \) it is correct with respect to the semantic of XPEGs.

The body of the \texttt{parse} function performs pattern matching on expression \( e \) and interprets it according to the semantics from Figure 2. We show a simplified (the actual pattern matching is slightly more involved due to dealing with dependent types) excerpt of this function for a few types of expressions:

\[
\text{match } e \ \text{with} \\
| \text{Empty} \Rightarrow \text{Ok} \ s \ i \\
| \text{Terminal } c \Rightarrow \\
\text{match } s \ \text{with}
\]
The termination argument for this function is based on the decrease of the pair of arguments 
\((e, s)\) in recursive calls with respect to the following relation \(\succ\): 
\[
(e_1, s_1) \succ (e_2, s_2) \iff \exists n_1, r_1, n_2, r_2 (e_1, s_1) \xrightarrow{n_1} r_1 \land (e_2, s_2) \xrightarrow{n_2} r_2 \land n_1 > n_2
\]
So \((e_1, s_1)\) is bigger than \((e_2, s_2)\) in the order if its step-count in the semantics is bigger. 
The relation \(\succ\) is clearly well-founded, due to the last conjunct with \(\succ\), the well-founded 
order on \(\mathbb{N}\). Since the semantics of \(G\) is complete (due to Theorem 4.2 and the check for 
well-formedness of \(G\) as described in Section 5.2) we can prove that all recursive calls are 
indeed decreasing with respect to \(\succ\).
Clearly this function also generates a number of proof obligations for expressing correctness of the returned result with respect to the semantics of PEGs. Dismissing them is actually rather straightforward, due to the fact that the implementation of the interpreter and the operation semantics of PEGs are very close to each other. That means that by far the majority of our work was in establishing termination, not correctness.

6. Extracting a Parser: Practical Evaluation

In the previous section we described a formal development of an XPEG interpreter in the proof assistant Coq. This should allow us for an arbitrary, well-formed XPEG $G$, to specify it in Coq and, using Coq’s extraction capabilities [Let08], to obtain a certified parser for $G$. We are interested in code extraction from Coq, to ease practical use of TRX and to improve its performance. At the moment target languages for extraction from Coq are OCaml [L+96], Haskell [PJ+02] and Scheme [SJ98]. We use the FSets [FL04] library (part of the Coq standard library for manipulation of the set data-type) developed using Coq’s modules and functors [Chr03], which are not yet supported by extraction to Haskell or Scheme. However, there is an ongoing work on porting FSets to type classes [SO08], which are supported by extraction.

First, in Section 6.1, we will sketch the various performance-related improvements that we made along our development and present case studies on two examples: XML and Java. Then in Section 6.2 we will present a benchmark of certified TRX again a number of other tools on those two examples.

6.1. Case study of TRX on XML and Java. A well-known issue with extraction is the performance of obtained programs [CFL06, Let08]. Often the root of this problem is the fact that many formalizations are not developed with extraction in mind and trying to extract a computational part of the proof can easily lead to disastrous performance [CFL06]. On the other hand the CompCert project [Ler09] is a well-known example of extracting a certified compiler with satisfactory performance from a Coq formalization.

As most of TRX’s formalization deals with grammar well-formedness, which should be discarded in the extracted code, we aimed at comparable performance for certified TRX and its non-certified counterpart that we prototyped manually. We found however that the first version’s performance was unacceptable and required several improvements, which we will discuss in the remainder of this section.

We started with a case study of XML using an XML PEG developed internally at MLstate. The first extracted version of TRX-cert parsed 32kB of XML in more than one minute. To our big surprise, performance was somewhere between quadratic and cubic with rather large constants. To our even bigger surprise, inspection of the code revealed that the rev function from Coq’s standard library (from the module Coq.Lists.List) that reverses a list was the source of the problem. The rev function is implemented using append to concatenate lists at every step, hence yielding quadratic time complexity.

We used this function to convert the input from OCaml strings to the extracted type of Coq strings. This is another difficulty of working with extracted programs: all the data-types in the extracted program are defined from scratch and combining such programs with un-certified code, even just to add a minimal front-end, as in our case, sometimes requires translating back and forth between OCaml’s primitive types and the extracted types of Coq.
Fixing the problem with \textit{rev} resulted in a linear complexity but the constant was still unsatisfactory. We quickly realized that implementing the range operator by means of repeated choice is suboptimal as a common class of letters \([a-z]\) would lead to a composition of 26 choices. Hence we extended the semantics of XPEGs with semantics of the range operator and instead of deriving it implemented it “natively”.

Yet another surprise was in store for us as the performance instead of improving got worse by approximately 30\%. This time the problem was the fact that in Coq there is no predefined polymorphic comparison operator (as in OCaml) so for the range operation we had to implement comparison on characters. We did that by using the predefined function from the standard library converting a character to its ASCII code. And yet again we encountered a problem that the standard library is much better suited for reasoning than computing: this conversion function uses natural numbers in Peano representation. By re-implementing this function using natural numbers in binary notation (available in the standard library) we decreased the running time by a factor of 2.

Further profiling the OCaml program revealed that it spends 85\% of its time performing garbage collection (GC). By tweaking the parameters of OCaml’s GC, we obtained an important 3x gain, leading to TRX-cert’s current performance as presented in the following section. We believe a more careful inspection will reveal more potential sources of improvements, as there is still a gap between the performance that we reached now and the one of our prototype written by hand.

We continued with a more realistic case study based on parsing the Java language, using the PEG for Java developed by Redziejowski [Red07]. The grammar, consisting of 216 rules, was automatically translated to TRX format. We immediately hit performance problems as our encoding contains a type enumerating all the rules \((\text{prod})\) and proving that equality is decidable on this type, using Coq’s \textit{decide equality} tactic, took initially 927 sec. \((\approx 15\text{ minutes})\). We were able to improve it by writing a tactic dedicated to such simple enumeration types (using Coq’s Ltac language) and decrease this time to 104 sec.

We did not meet any more scaling difficulties. Testing XML and Java grammars for well-formedness, with the extracted Ocaml code, took, respectively, 0.1 and 0.7 sec. (this test needs to be performed only once). We will discuss the performance of the parsing itself, and compare it with other tools, in the following section.

6.2. \textbf{Performance comparison.} For our benchmarking experiment, see Figure 8 on the following page, we used the following tools:

- \textbf{JAXP:} a reference implementation for the XML parser, using a DOM parser of the “Java API for XML processing”, JAXP [JAX].
- \textbf{TRX-cert:} the certified TRX interpreter, which is the subject of this paper and is described in more detail in Section 5.
- \textbf{TRX-gen:} MLstate’s own production-used PEG-based parser generator (for experiments we used its simple version without memoization).
- \textbf{TRX-int:} a simple prototype with comparable functionality to TRX-cert, though developed manually.
- \textbf{Mouse:} a PEG-based parser generator, with no memoization, implemented in Java by Redziejowski [Red09].

Figure 8 plots performance of the aforementioned tools on two benchmarks:
tool | XML parser | Java parser
--- | --- | ---
JAXP | 2.3s. | 23.0s. |
JavaCC | 5.1s. | 25.5s. |
TRX-gen | 40.0s. | 289.3s. |
TRX-int | 128.9s. | 662.4s. |
TRX-cert | 206.4s. | 269.6s. |
Mouse | 206.4s. | 269.6s. |

Figure 8: Performance of certified TRX (TRX-cert) compared to a number of other tools on the examples of parsing Java and XML.

**XML:** 10 XML files with a total size of 40MB generated using the XML benchmarks generator XMark [SWK+02].

**Java:** a complete source code of the J2SE JDK 5.0 consisting of nearly 11,000 files with a total size of 117MB.

The most interesting comparison is between TRX-cert and TRX-int. The latter was essentially a prototype of the former but developed manually, whereas TRX-cert is extracted from a formal Coq development. At the moment the certified version is approximately 2–3x slower. In principle this difference can be attributed either to the verification overhead (computations that are but should not be performed, as they are part of the logical reasoning to prove correctness and not of the actual algorithm), extraction overhead (sub-optimal code generated by the extraction process) or algorithmic overhead (the algorithm that we coded in Coq is sub-optimal in itself).

We believe there is no verification overhead in TRX-cert, as all the correctness proofs are discarded by the process of extraction and we never used the proof mode of Coq to define objects with computational content (which are extracted).

The extraction overhead in our case mainly manifests itself in many dispensable conversions. For instance the second component of the sigma type \( \{ x : T \mid P(x) \} \) is discarded during the extraction, so such a type is extracted simply as \( T \) and the first projection function \( \text{proj1}_\text{sig} \) as identity. Since sigma types are used extensively in our verification, the extracted code is full of such vacuous conversions. However, our experiments seem to indicate that Ocaml’s compiler is capable of optimizing such code, so that this should have no noticeable impact on performance.

Apart from those two types of overheads associated with extraction, often the sub-optimal extracted code can be traced back to sub-optimal code in the development itself or in Coq libraries. We already mentioned few of such problems in Section 6.1. We believe another one is the model of characters from the standard library of Coq, `Coq.Strings.Ascii`, which we used in this work. The characters are modeled by 8 booleans, *i.e.*, 8 bits of the character:

**Inductive ascii : Set := Ascii (---: bool).**

Not surprisingly such characters induce larger memory footprint and also comparison between such structures is much less efficient than between native (1-byte) characters of Ocaml. There is an on-going work on improving interplay between Ocaml’s native types and their Coq counter-parts, which should hopefully address this problem.
However, the main opportunity for improving performance seems to be in switching from interpretation to code generation. As witnessed by the difference between TRX-int and TRX-gen this can have a very substantial impact on performance. We will say some more about that in discussion in Section 8.

It is worth noting that the performance of TRX-cert is quite competitive when compared with Java code generated by Mouse.

We would like to conclude this section with the observation that even though making such benchmarks is important it is often just one of many factors for choosing a proper tool for a given task. There are many applications which will never parse files exceeding 100kB and it is often irrelevant whether that will take 0.1s. or 0.01s. For some of those applications it may be much more relevant that the parsing is formally guaranteed to be correct.

7. Related Work

Parsing is a well-studied and well-understood topic and the software for parsing, parser generators or libraries of parser combinators, is abundant. And yet there does seem to be hardly any work on formally verified parsing.

Danielsson [Dan10] develops a library of parser combinators (see Hutton [Hut92]) with termination guarantees in the dependently typed functional programming language Agda [Agd] (see also joined work with Norell [DN08]). The main difference in comparison with our work is that Danielsson provides a library of combinators, whereas we aim at a parser generator for PEG grammars (though at the moment we only have an interpreter). Perhaps more importantly, the approach of Danielsson allows many forms of left recursion, which we cannot handle at present. Another difference is in the way termination is ensured: Danielsson uses dependent types to extend type of parser combinators with the information about whether or not they accept the empty string; which is subsequently used to guarantee termination. In contrast we use deep embedding of the grammar and a reflective procedure to check whether a given grammar is terminating. Some consequences of those choices will be explored in more depth in the following section.

Ideas similar to Danielsson and Norell [DN08] were previously put forward, though just as a proof of concept, by McBride and McKinna [MM02].

Probably the closest work to ours is that of Barthwal and Norrish [BN09], where the authors developed an SLR parser in HOL. The main differences with our work are:

- PEGs are more expressive than SLR grammars, which are usually not adequate for real-world computer languages,
- as a consequence of using PEGs we can deal with lexical analysis, while it would have to be formalized and verified in a separate stage for the SLR approach,
- our parser is proven to be totally correct, i.e., correct with respect to its specification and terminating on all possible inputs (which was actually far more difficult to establish than correctness), while the latter property does not hold for the work of Barthwal and Norrish.
- performance comparison with this work is not possible as the paper does not present any case-studies, benchmarks or examples, but the fact that “the DFA states are computed on the fly” [BN09] suggests that the performance was not the utmost goal of that work.
Finally there is the recent development of a packrat PEG parser in Coq by Wisnesky et al. [WMM09], where the given PEG grammar is compiled into an imperative computation within the Ynot framework, that when run over an arbitrary imperative character stream, returns a parsing result conforming with the specification of PEGs. Termination of such generated parsers is not guaranteed.

8. Discussion and Future Work

One of the main challenges in developing a certified parser is ensuring its termination. In this paper we presented an extrinsic approach to this problem: we use a deep embedding to represent parsing expressions in Coq and then develop a certified algorithm to verify that a given PEG is well-formed. We then express the parser (interpreter) with non-structural recursion and the well-formedness of the grammar allows us to justify that the recursion is well-founded.

There is an alternative, intrinsic approach to the problem of termination, which is, for instance, used by Danielsson [DN08, Dan10], as mentioned in the previous section. They develop a library of parser combinators and use the type system of the host language – in this case, Agda – to restrict the parser combinators to well-formed ones.

This is a very attractive approach, as by cleverly using the type system of the host language we obtain certain verified properties for free, hence decreasing the formalization overhead. However, it has the usual drawback of a shallow embedding approach: it is tied to the host language, i.e. Danielsson’s parsers must unavoidably be written in Agda.

At the moment the same is true about our work: to use certified TRX, as presented in this paper, the grammar must be expressed in Coq. However, this is not a necessity with our approach, as we will sketch in a moment. The motivation for avoiding the need to use Coq is clear: this could make our certified parser technology usable for people outside of the small community of theorem provers (Coq, in particular) experts.

As our work uses deep embedding of parsing expressions, it should be possible to turn it into a generic parser generator. Doing so could be accomplished by bootstrapping TRX: it should be possible to write a grammar in it that would synthesize a PEG in Coq (in our format; Section 5.1) from its textual description. After this transformation the grammar could be checked for well-formedness (with our generic procedure for checking well-formedness of PEGs; Section 5.2) finally allowing parsing with this grammar (with our interpreter; Section 5.3). This would result (via extraction) in a tool that would be capable of parsing grammars expressed in a simple textual markup, hence surpassing any need to use/know Coq for the users of such a tool.

The main difficulty with obtaining such a tool lies in the bootstrapping process. To do so we would need a kind of a higher-order grammar: a PEG formally describing its own syntax, that would take a textual description of a grammar and turn it into a PEG in our format. Such a grammar would need to have the type $PExp (PExp (\_))$ and, as already hinted in Section 5.1 with our present encoding, that would lead to universe inconsistency problems. Also, our current use of module system precludes such use-case as modules are not first-class citizens in Coq and one cannot construct higher-order functors.

But there is a more fundamental problem here: how do we synthesize semantic actions from their textual description? If the semantics actions were to be expressed in the calculus of constructions of Coq, the way they are now, this seems to be futile.
Let us step back a bit for a moment and consider a simpler problem: what if we only wanted a recognizer, i.e., a parser that does not return any result, but only indicates whether a given string is in the language described by the grammar or not. To address the aforementioned problem with modules ([Chr03]) we could switch to type classes ([SO08]) instead. Then we could build a generic recognizer as follows (pseudo-code):

**Definition** \(\text{PEG}_{\text{grammar}} : \text{PExp} \ x := \ldots\)

**Program Definition** \(\text{do-parser} (\text{grammar} : \text{string}) (\text{input} : \text{string}) := \)

\[
\begin{align*}
\text{match } \text{parse} & \text{PEG}_{\text{grammar}} \text{ grammar with} \\
| \text{PR}_{\text{ok}} \_\text{peg} & \Rightarrow \text{parse} (\text{promote peg}) \text{ input} \\
| \text{PR}_{\text{fail}} & \Rightarrow \text{PR}_{\text{fail}}
\end{align*}
\]

Here \(\text{PEG}_{\text{grammar}}\) is the grammar for PEGs. The main \text{do-parser} function takes two arguments: \text{grammar} with the textual description of the grammar to use and \text{input} being the \text{input} which we want to parse using the given \text{grammar}. We use \(\text{PEG}_{\text{grammar}}\) to parse \text{grammar} and, hopefully, obtain its internal representation \(\text{peg} : \text{pexp}\), in which case we again invoke \text{parse} with \text{promote peg} \text{grammar} and \text{input} as the input string. Extracting \text{do-parser} would give us a generic recognizer, that could be used without Coq (or any knowledge thereof).

Admittedly, in practice we are rarely interested in merely validating the input; usually we really want to \text{parse} it, obtaining its structural representation. How can the above approach be extended to accommodate that and still result in a stand-alone tool, not requiring interaction with Coq?

One option would be to move from interpretation to code generation and then using the target language to express semantic actions. An additional advantage is that this should result in a big performance gain (compare the performance of TRX and TRX-int in Figure 8). But that would be a major undertaking requiring reasoning with respect to the target language’s semantics for the correctness proofs and some sort of (formally verified) termination analysis for that language, to ensure termination of the code of semantic actions (and hence the generated parser).

The aforementioned termination problem for a parser generator could be simplified by restricting the code allowed in semantic actions to some subset of the target language, which is still expressive enough for this purpose but for which the termination analysis is simpler. For instance for a purely functional target language one could disallow recursion altogether in productions (making termination evident), only allowing use of some predefined set of combinators (to improve expressivity of semantic actions), which could be proven terminating manually.

Another solution would be not to use semantic actions altogether, but construct a parse tree, the shape of which could be influenced by annotations in the grammar. This is the approach used, for instance, in the Ocaml PEG-based parser generator Aurochs [Dur09]. We believe this is a promising approach that we hope to explore in the future work.

A complete different approach to developing a practical, certified parser generator would be the standard technique of verification \textit{a posteriori}: use an untrusted parser that, apart from its result, generates some sort of a certificate (parse tree) and develop a (formally correct) tool to verify, using the certificate, that the output of the tool (for a given input and given grammar) is correct. The attractiveness of this approach lies in the fact that such
a verifier would typically be much simpler than the parser itself. There are two problems with this approach though:

- this approach could at best give us partial correctness guarantees, as we would not be able to ensure termination of the un-trusted parser (unless we also prove it in some way);
- if the parsing is successful it is relatively clear what a certificate should be (parse tree), but what if it is not? How can we certify incorrectness of input with respect to the grammar?

Apart from making the certified TRX a Coq independent, standalone tool and moving from interpretation to code generation we also identify a number of other possible improvements to TRX as future work:

1. Linear parsing time with PEGs can be ensured by using packrat parsing [For02b], i.e., enhancing the parser with memoization. This should be relatively easy to implement (it has, respectively, no and little impact on the termination and correctness arguments for certified TRX), but induces high memory costs (and some performance overhead), so it is not clear whether this would be beneficial. An alternative would be to develop (formally verified?) tools to perform grammar analysis and warn the user in case the grammar can lead to exponential parsing times.

2. Another important aspect is that of left-recursive grammars, which occur naturally in practice. At the moment it is the responsibility of the user to eliminate left-recursion from a grammar. In the future, we plan to address this problem either by means of left-recursion elimination [For02a], i.e., transforming a left-recursive grammar to an equivalent one where left-recursion does not occur (this is not an easy problem in presence of semantic actions, especially if one also wants to allow mutually left-recursive rules). Another possible approach is an extension to the memoization technique that allows dealing with left-recursive rules [WDM08].

3. Finally support for error messages, for instance following that of the PEG-based parser generator Puppy [For02a], would greatly improve usability of TRX.

9. Conclusions

In this paper we described a Coq formalization of the theory of PEGs and, based on it, a formal development of TRX: a formally verified parser interpreter for PEGs. This allows us to write a PEG, together with its semantic actions, in Coq and then to extract from it a parser with total correctness guarantees. That means that the parser will terminate on all inputs and produce parsing results correct with respect to the semantics of PEGs. Considering the importance of parsing, this result appears as a first step towards a general way to bring added quality and security to all kinds of software.

The emphasis of our work was on practicality, so apart from treating this as an interesting academic exercise, we were aiming at obtaining a tool that scales and can be applied to real-life problems. We performed a case study with a (complete) Java grammar and demonstrated that the resulting parser exhibits a reasonable performance. We also stressed the importance of making those results available to people outside of the small circle of theorem-proving experts and presented a plan of doing so as future work.
Acknowledgments. We would like to thank Matthieu Sozeau for his invaluable help with the Program feature [Soz07] of Coq and the anonymous referees for their helpful comments, which greatly improved presentation of this paper. Also the very pragmatic (and immensely helpful) book of Chipala [Chl09], as well as friendly advice from people on Coq’s mailing list turned out to be invaluable in the course of this work.

References


