TRX: A Formally Verified Parser Interpreter

Adam Koprowski and Henri Binsztok

MLstate, Paris, France
{Adam.Koprowski,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, 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 (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 [20] 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 [17] about CompCert, 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 [17, Figure 1]. Can you trust your parser?

Parsing expression grammars (PEGs) [14] 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 [5]. 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 [2, 13, 12]. It is not easy to support (indirect) left-recursive rules in PEGs, as they lead to non-terminating parsers [29].

In this paper we present TRX: a PEG-based parser interpreter *formally developed* in the proof assistant Coq [28, 4]. As a result, expressing a grammar in Coq allows one, via its extraction capabilities [19], 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.

The contributions of this paper are:

- extension of PEGs with semantic actions,
- a Coq formalization of the theory of PEGs and
- a Coq development of TRX: a PEG interpreter allowing to obtain a parser with total correctness guarantees for an arbitrary PEG grammar.

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 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 a small case study of extracting an XML parser from it, presenting a benchmark of TRX against other parser generators and giving an account of our experience with extraction. We discuss related work in Section 7 and conclude in Section 8.

$$
\begin{array}{lllll}
\Delta ::= \epsilon & \text{empty expr.} & \mid e_1/e_2 & \text{a } \textit{prioritized} \text{ choice } (e_1, e_2 \in \Delta) \\
\quad\mid [\cdot] & \text{any character} & \mid e* & \text{a} \geq 0 \; \textit{greedy} \text{ repetition } (e \in \Delta) \\
\quad\mid [a] & \text{a terminal } (a \in \mathcal{V}_T) & \mid e+ & \text{a} \geq 1 \; \textit{greedy} \text{ repetition } (e \in \Delta) \\
\quad\mid [\text{``}s\text{''}] & \text{a literal } (s \in \mathcal{S}) & \mid e? & \text{an optional expression } (e \in \Delta) \\
\quad\mid [a\text{--}z] & \text{a range } (a, z \in \mathcal{V}_T) & \mid !e & \text{a not-predicate } (e \in \Delta) \\
\quad\mid A & \text{a non-terminal } (A \in \mathcal{V}_N) & \mid \&e & \text{an and-predicate } (e \in \Delta) \\
\quad\mid e_1; e_2 & \text{a sequence } (e_1, e_2 \in \Delta) & &
\end{array}
$$

Fig. 1. Parsing expressions

2 Parsing Expression Grammars (PEGs)

The content of this section is a different presentation of the ideas from [14]. For more details we refer to the original article. For a general overview of parsing we refer to, for instance, [1].

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 1 (Parsing expressions). *We introduce a set of* parsing expressions*, Δ, over a finite set of terminals $\mathcal{V}_T$ and a finite set of non-terminals $\mathcal{V}_N$. We denote the set of strings as $\mathcal{S}$ and a string $s \in \mathcal{S}$ is a list of terminals $\mathcal{V}_T$. The inductive definition of Δ is given in Figure 1.* ◇

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

- *Empty expression* ϵ 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* $[\text{``}s\text{''}]$ 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.

$$\frac{}{(\epsilon, s) \overset{1}{\rightsquigarrow} \surd_s} \qquad \frac{(\mathrm{P}_{\mathrm{exp}}(A), s) \overset{n}{\rightsquigarrow} r}{(A, s) \overset{n+1}{\rightsquigarrow} r} \qquad \frac{}{([\cdot], x :: xs) \overset{1}{\rightsquigarrow} \surd_{xs}}$$

$$\frac{}{([\cdot], []) \overset{1}{\rightsquigarrow} \bot} \qquad \frac{}{([x], x :: xs) \overset{1}{\rightsquigarrow} \surd_{xs}} \qquad \frac{}{([x], []) \overset{1}{\rightsquigarrow} \bot}$$

$$\frac{x \neq y}{([y], x :: xs) \overset{1}{\rightsquigarrow} \bot} \qquad \frac{(e, s) \overset{m}{\rightsquigarrow} \bot}{(!e, s) \overset{m+1}{\rightsquigarrow} \surd_s} \qquad \frac{(e, s) \overset{m}{\rightsquigarrow} \surd_{s'}}{(!e, s) \overset{m+1}{\rightsquigarrow} \bot}$$

$$\frac{(e_1, s) \overset{m}{\rightsquigarrow} \bot}{(e_1; e_2, s) \overset{m+1}{\rightsquigarrow} \bot} \qquad \frac{(e_1, s) \overset{m}{\rightsquigarrow} \surd_{s'} \qquad (e_2, s') \overset{n}{\rightsquigarrow} r}{(e_1; e_2, s) \overset{m+n+1}{\rightsquigarrow} r} \qquad \frac{(e_1, s) \overset{m}{\rightsquigarrow} \bot \qquad (e_2, s) \overset{n}{\rightsquigarrow} r}{(e_1/e_2, s) \overset{m+n+1}{\rightsquigarrow} r}$$

$$\frac{(e_1, s) \overset{m}{\rightsquigarrow} \surd_{s'}}{(e_1/e_2, s) \overset{m+1}{\rightsquigarrow} \surd_{s'}} \qquad \frac{(e, s) \overset{m}{\rightsquigarrow} \surd_{s'} \qquad (e*, s') \overset{n}{\rightsquigarrow} \surd_{s''}}{(e*, s) \overset{m+n+1}{\rightsquigarrow} \surd_{s''}} \qquad \frac{(e, s) \overset{m}{\rightsquigarrow} \bot}{(e*, s) \overset{m+1}{\rightsquigarrow} \surd_s}$$

Fig. 2. Formal semantics of PEGs

- An *and-predicate (resp. not-predicate)* $\&e$ (*resp.* $!e$) succeeds only if the input conforms to e (*resp.* does not conform to e) but does not consume any input.

We now define PEGs, which are essentially a finite set of non-terminals, also referred to as *productions*, with their corresponding parsing expressions.

Definition 2 (Parsing Expressions Grammar (PEG)). *A parsing expressions grammar (PEG), $\mathcal{G}$, is a tuple $(\mathcal{V}_T, \mathcal{V}_N, \mathrm{P}_{\mathrm{exp}}, v_{\mathrm{start}})$, where:*

- *$\mathcal{V}_T$ is a finite set of terminals,*
- *$\mathcal{V}_N$ is a finite set of non-terminals,*
- *$\mathrm{P}_{\mathrm{exp}}$ is the interpretation of the productions, i.e., $\mathrm{P}_{\mathrm{exp}} : \mathcal{V}_N \to \Delta$ and*
- *v_{start} is the start production, $v_{\mathrm{start}} \in \mathcal{V}_N$.* ◇

We will now present the formal semantics of PEGs. The semantics is given by means of tuples $(e, s) \overset{m}{\rightsquigarrow} r$, which indicate that parsing expression $e \in \Delta$ applied on a string $s \in \mathcal{S}$ gives, in m steps, the result r, where r is either $\bot$, denoting that parsing failed, or $\surd_{s'}$, indicating that parsing succeeded and s' is what remains to be parsed. We will drop the m annotation whenever irrelevant.

The complete semantics is presented in Figure 2. Please note that the following operators from Definition 1 can be derived and therefore are not included in the semantics:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
[a\text{–}z] & ::= & [a] \ / \ \ldots \ / \ [z] & e+ & ::= & e; e* & \&e & ::= & !!e \\
[\text{“}s\text{”}] & ::= & [s_0] \,; \ldots ; [s_n] & e? & ::= & e/\epsilon & & &
\end{array}$$

2.2 CFGs vs PEGs

The main differences between PEGs and CFGs are the following:

- the choice operator, e_1/e_2, is *prioritized*, *i.e.*, e_2 is tried only if e_1 fails;

- the repetition operators, $e*$ and $e+$, are *greedy*, which allows to easily express "longest-match" parsing, which is almost always desired;
- *syntactic predicates* [22], $\&e$ 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 32. 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* [5] and performing this type of parsing for PEGs is straightforward (more on that in Section 5). By using the technique of *packrat parsing* [2, 13], *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 [13].

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 3 (PEG for simple mathematical expressions). Consider a PEG for simple mathematical expressions over 5 non-terminals: $\mathcal{V}_N ::= \{\texttt{ws}, \texttt{number}, \texttt{term}, \texttt{factor}, \texttt{expr}\}$ with the following productions ($\mathrm{P}_{\mathrm{exp}}$ function from Definition 2):

$$\begin{array}{rcl}
\texttt{ws} & ::= & ([\text{␣}] \;/\; [\backslash t])* \\
\texttt{number} & ::= & [0\text{–}9]+ \\
\texttt{term} & ::= & \texttt{ws}\ \texttt{number}\ \texttt{ws} \;/\; \texttt{ws}\ [(]\ \texttt{expr}\ [)]\ \texttt{ws} \\
\texttt{factor} & ::= & \texttt{term}\ [*]\ \texttt{factor} \;/\; \texttt{term} \\
\texttt{expr} & ::= & \texttt{factor}\ [+]\ \texttt{expr} \;/\; \texttt{factor}
\end{array}$$

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 $v_{\mathrm{start}} ::= \texttt{expr}$.

First, let us note that lexical analysis is incorporated into this grammar by means of the 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 `expr` with:

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

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. ◁

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 — for more details we refer to [12, Chapter 2.4].

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 Δ with a family of types Δ_α, where the index α 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 α is the type of lists of elements of α for any type α. Also string ::= list char.
- $\alpha_1 * \ldots * \alpha_n$ 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.

$$\frac{}{\epsilon \,:\, \Delta_{\mathrm{True}}} \qquad \frac{}{[\cdot] \,:\, \Delta_{\mathrm{char}}} \qquad \frac{a \in \mathcal{V}_T}{[a] \,:\, \Delta_{\mathrm{char}}}$$

$$\frac{A \in \mathcal{V}_N}{A \,:\, \Delta_{\mathrm{P_{type}}(A)}} \qquad \frac{e_1 \,:\, \Delta_\alpha \qquad e_2 \,:\, \Delta_\beta}{e_1; e_2 \,:\, \Delta_{\alpha * \beta}} \qquad \frac{e_1 \,:\, \Delta_\alpha \qquad e_2 \,:\, \Delta_\alpha}{e_1/e_2 \,:\, \Delta_\alpha}$$

$$\frac{e \,:\, \Delta_\alpha}{e* \,:\, \Delta_{\mathrm{list}\,\alpha}} \qquad \frac{e \,:\, \Delta_\alpha}{!e \,:\, \Delta_{\mathrm{True}}} \qquad \frac{e \,:\, \Delta_\alpha \qquad f \,:\, \alpha \to \beta}{e[\mapsto]f \,:\, \Delta_\beta}$$

Fig. 3. Typing rules for parsing expressions with semantic actions

- option α is the type optionally holding a value of type α, with two constructors None and Some v with $v \,:\, \alpha$.

Definition 4 (Parsing expressions with semantic values). *We introduce a set of* parsing expressions with semantic values, Δ_α, *as an inductive family indexed by the type α of semantic values of an expression. The typing rules for Δ_α 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 few coercions:

$$\begin{array}{rcl}
[a\text{–}z] : \Delta_{\mathrm{char}} & ::= & [a] \;\; / \;\; \ldots \;\; / \;\; [z] \\
[\text{“}s\text{”}] : \Delta_{\mathrm{string}} & ::= & [s_0] \,;\, \ldots \,;\, [s_n] \;\; [\mapsto] \;\; \mathrm{tuple2str} \\
e+ : \Delta_{\mathrm{list}\,\alpha} & ::= & e; e* \;\; [\mapsto] \;\; \lambda x \,.\, x_1 :: x_2 \\
e? : \Delta_{\mathrm{option}\,\alpha} & ::= & e \;\; [\mapsto] \;\; \lambda x \,.\, \mathrm{Some}\; x \\
 & & /\, \epsilon \;\; [\mapsto] \;\; \lambda x \,.\, \mathrm{None} \\
\&e : \Delta_{\mathrm{True}} & ::= & !!e
\end{array}$$

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

The definition of an extended parsing expression grammar (XPEG) is as expected (compare with Definition 1).

Definition 5 (Extended Parsing Expressions Grammar (XPEG)). *An extended parsing expressions grammar (XPEG), $\mathcal{G}$, is a tuple* $(\mathcal{V}_T, \mathcal{V}_N, \mathrm{P_{type}}, \mathrm{P_{exp}}, v_{\mathrm{start}})$*, where:*

- $\mathcal{V}_T$ *is a finite set of terminals,*
- $\mathcal{V}_N$ *is a finite set of non-terminals,*
- $\mathrm{P_{type}} : \mathcal{V}_N \to \mathrm{Type}$ *is a function that gives types of semantic values of all productions.*
- $\mathrm{P_{exp}}$ *is the interpretation of the productions of the grammar, i.e.,* $\mathrm{P_{exp}} : \forall_{A:\mathcal{V}_N} \Delta_{\mathrm{P_{type}}(A)}$ *and*
- v_{start} *is the start production,* $v_{\mathrm{start}} \in \mathcal{V}_N$*.* ◇

$$\frac{}{(\epsilon, s) \overset{1}{\rightsquigarrow} \surd^{I}_{s}} \qquad \frac{(\mathrm{P}_{\exp}(A), s) \overset{m}{\rightsquigarrow} r}{(A, s) \overset{m+1}{\rightsquigarrow} r} \qquad \frac{}{([\cdot], x :: xs) \overset{1}{\rightsquigarrow} \surd^{x}_{xs}}$$

$$\frac{}{([\cdot], []) \overset{1}{\rightsquigarrow} \bot} \qquad \frac{(e_1, s) \overset{m}{\rightsquigarrow} \bot \qquad (e_2, s) \overset{n}{\rightsquigarrow} r}{(e_1/e_2, s) \overset{m+n+1}{\rightsquigarrow} r} \qquad \frac{(e_1, s) \overset{m}{\rightsquigarrow} \surd^{v}_{s'}}{(e_1/e_2, s) \overset{m+1}{\rightsquigarrow} \surd^{v}_{s'}}$$

$$\frac{}{([x], x :: xs) \overset{1}{\rightsquigarrow} \surd^{x}_{xs}} \qquad \frac{}{([x], []) \overset{1}{\rightsquigarrow} \bot} \qquad \frac{x \neq y}{([y], x :: xs) \overset{1}{\rightsquigarrow} \bot}$$

$$\frac{(e_1, s) \overset{m}{\rightsquigarrow} \surd^{v_1}_{s'} \qquad (e_2, s') \overset{n}{\rightsquigarrow} \bot}{(e_1; e_2, s) \overset{m+n+1}{\rightsquigarrow} \bot} \qquad \frac{(e_1, s) \overset{m}{\rightsquigarrow} \surd^{v_1}_{s'} \qquad (e_2, s') \overset{n}{\rightsquigarrow} \surd^{v_2}_{s''}}{(e_1; e_2, s) \overset{m+n+1}{\rightsquigarrow} \surd^{(v_1, v_2)}_{s''}} \qquad \frac{(e_1, s) \overset{m}{\rightsquigarrow} \bot}{(e_1; e_2, s) \overset{m+1}{\rightsquigarrow} \bot}$$

$$\frac{(e, s) \overset{m}{\rightsquigarrow} \bot}{(e*, s) \overset{m+1}{\rightsquigarrow} \surd^{[]}_{s}} \qquad \frac{(e, s) \overset{m}{\rightsquigarrow} \surd^{v}_{s'} \qquad (e*, s') \overset{n}{\rightsquigarrow} \surd^{vs}_{s''}}{(e*, s) \overset{m+n+1}{\rightsquigarrow} \surd^{v::vs}_{s''}} \qquad \frac{(e, s) \overset{m}{\rightsquigarrow} \bot}{(!e, s) \overset{m+1}{\rightsquigarrow} \surd^{I}_{s}}$$

$$\frac{(e, s) \overset{m}{\rightsquigarrow} \surd^{v}_{s'}}{(!e, s) \overset{m+1}{\rightsquigarrow} \bot} \qquad \frac{(e, s) \overset{m}{\rightsquigarrow} \surd^{v}_{s'}}{(e[\mapsto]f, s) \overset{m+1}{\rightsquigarrow} \surd^{f(v)}_{s'}} \qquad \frac{(e, s) \overset{m}{\rightsquigarrow} \bot}{(e[\mapsto]f, s) \overset{m+1}{\rightsquigarrow} \bot}$$

Fig. 4. Formal semantics of XPEGs with semantic actions.

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

Example 6 (Simple mathematical expressions ctd.). Let us extend the grammar from Example 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 $e[\sharp] ::= e \quad [\mapsto] \quad \lambda x \,.\, I$.

$$\begin{array}{rcll}
\texttt{ws} & ::= & ([\text{␣}] \ / \ [\backslash t])* & [\sharp] \\
\texttt{number} & ::= & [0\text{–}9]+ & [\mapsto] \ \ \mathrm{digListToNat} \\
\texttt{term} & ::= & \texttt{ws number ws} & [\mapsto] \ \ \lambda x \,.\, x_2 \\
 & / & \texttt{ws} \ [(] \ \texttt{expr} \ [)] \ \texttt{ws} & [\mapsto] \ \ \lambda x \,.\, x_3 \\
\texttt{factor} & ::= & \texttt{term} \ [*] \ \texttt{factor} & [\mapsto] \ \ \lambda x \,.\, x_1 * x_3 \\
 & / & \texttt{term} & \\
\texttt{expr} & ::= & \texttt{factor} \ [+] \ \texttt{expr} & [\mapsto] \ \ \lambda x \,.\, x_1 + x_3 \\
 & / & \texttt{factor} &
\end{array}$$

where digListToNat converts a list of digits to their decimal representation.

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 semantics of (X)PEGs. They are all variants of results obtained by Ford [14], only now we extend them to XPEGs. First we prove that, as expected, the parsing only consumes a prefix of a string.

Theorem 31 *If* $(e, s) \overset{m}{\rightsquigarrow} \surd^{v}_{s'}$ *then* s' *is a suffix of* s*.*

Proof. Induction on the derivation of $(e, s) \overset{m}{\rightsquigarrow} \surd^{v}_{s'}$ using transitivity of the prefix property for sequence and repetition cases. □

As mentioned earlier, (X)PEGs are unambiguous:

Theorem 32 *If* $(e, s) \overset{m_1}{\rightsquigarrow} r_1$ *and* $(e, s) \overset{m_2}{\rightsquigarrow} r_2$ *then* $m_1 = m_2$ *and* $r_1 = r_2$*.*

Proof. By complete induction on m_1. All cases immediate from the semantics of XPEGs. □

We wrap up this section with a simple property about the repetition 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 33 *If* $(e, s) \overset{m}{\rightsquigarrow} \surd^{v}_{s}$ *then* $(e*, s) \not\rightsquigarrow r$ *for all* r*.*

Proof. Assume $(e, s) \overset{m}{\rightsquigarrow} \surd^{v}_{s}$ and $(e*, s) \overset{n}{\rightsquigarrow} \surd^{vs}_{s'}$ for some n, vs and s' (we cannot have $(e*, s) \overset{n}{\rightsquigarrow} \bot$ as $e*$ never fails). By the first rule for repetition $(e*, s) \overset{m+n+1}{\rightsquigarrow} \surd^{v::vs}_{s'}$, which contradicts the second assumption by Theorem 32. □

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 $\mathcal{G}$.

4.1 Termination problem for XPEGs

Ensuring termination of a PEG parser essentially comes down to two problems:

- termination of all semantic actions in $\mathcal{G}$ and
- completeness of $\mathcal{G}$ with respect to PEGs semantics.

As for the first problem it means that all f functions used in coercion operators $e[\mapsto]f$ in $\mathcal{G}$, 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 $\mathcal{G}$ under consideration is *complete*, *i.e.*, whether it either succeeds or fails on all input strings. The only potential source of incompleteness of $\mathcal{G}$ is (mutual) *left-recursion* in the grammar.

We already hinted at this problem in Example 3 with the rule:

$$\texttt{expr} ::= \texttt{expr}\ [+]\ \texttt{factor}\ /\ \texttt{factor}.$$

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:

$$\texttt{A} ::= \texttt{B}\ /\ \texttt{C}\ \texttt{!D}\ \texttt{A}$$

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 [29], 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 $\mathcal{G}$ is empty [14].

While in general checking grammar completeness is undecidable, we follow [14] 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 $\mathcal{G}$ as:

$$\mathrm{E}(\mathcal{G}) = \{e' \mid e' \sqsubseteq e, e \in \mathrm{P}_{\mathrm{exp}}(A), A \in \mathcal{V}_N\}$$

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
- "$\perp$": 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 $\mathbb{P}_\perp$). We will also write $e \in \mathbb{P}_{\geq 0}$ to denote $e \in \mathbb{P}_0 \vee e \in \mathbb{P}_{>0}$. We define inference rules for deriving those properties in Figure 5.

We start with empty sets of properties and apply those inference rules over $\mathrm{E}(\mathcal{G})$ until reaching a fix-point. The existence of the fix-point is ensured by the fact that we extend those property sets monotonically and they are bounded by the finite set $\mathrm{E}(\mathcal{G})$. We summarize the semantics of those properties in the following lemma:

Lemma 41 ([14]) *For arbitrary $e \in \Delta$ and $s \in \mathcal{S}$:*

- *if $(e, s) \overset{n}{\rightsquigarrow} \surd_s$ then $e \in \mathbb{P}_0$,*
- *if $(e, s) \overset{n}{\rightsquigarrow} \surd_{s'}$ and $|s'| < |s|$ then $e \in \mathbb{P}_{>0}$ and*
- *if $(e, s) \overset{n}{\rightsquigarrow} \perp$ then $e \in \mathbb{P}_\perp$.*

Proof. Induction over n. All cases easy by the induction hypothesis and semantical rules of XPEGs, except for $e*$ which requires use of Lemma 33. □

$$\frac{}{\epsilon \in \mathbb{P}_0} \qquad \frac{}{[\cdot] \in \mathbb{P}_{>0}} \qquad \frac{}{[\cdot] \in \mathbb{P}_{\bot}} \qquad \frac{a \in \mathcal{V}_T}{[a] \in \mathbb{P}_{>0}} \qquad \frac{a \in \mathcal{V}_T}{[a] \in \mathbb{P}_{\bot}} \qquad \frac{e \in \mathbb{P}_{\bot}}{e* \in \mathbb{P}_0} \qquad \frac{e \in \mathbb{P}_{>0}}{e* \in \mathbb{P}_{>0}}$$

$$\frac{\star \in \{0, >0, \bot\} \qquad A \in \mathcal{V}_N \qquad \mathrm{P}_{\mathrm{exp}}(A) \in \mathbb{P}_\star}{A \in \mathbb{P}_\star} \qquad \frac{e_1 \in \mathbb{P}_{\bot} \vee (e_1 \in \mathbb{P}_{\geq 0} \wedge e_2 \in \mathbb{P}_{\bot})}{e_1; e_2 \in \mathbb{P}_{\bot}}$$

$$\frac{(e_1 \in \mathbb{P}_{>0} \wedge e_2 \in \mathbb{P}_{\geq 0}) \vee (e_1 \in \mathbb{P}_{\geq 0} \wedge e_2 \in \mathbb{P}_{>0})}{e_1; e_2 \in \mathbb{P}_{>0}} \qquad \frac{e_1 \in \mathbb{P}_0 \qquad e_2 \in \mathbb{P}_0}{e_1; e_2 \in \mathbb{P}_0}$$

$$\frac{e_1 \in \mathbb{P}_0 \vee (e_1 \in \mathbb{P}_{\bot} \wedge e_2 \in \mathbb{P}_0)}{e_1/e_2 \in \mathbb{P}_0} \qquad \frac{e_1 \in \mathbb{P}_{\bot} \qquad e_2 \in \mathbb{P}_{\bot}}{e_1/e_2 \in \mathbb{P}_{\bot}}$$

$$\frac{e_1 \in \mathbb{P}_{>0} \vee (e_1 \in \mathbb{P}_{\bot} \wedge e_2 \in \mathbb{P}_{>0})}{e_1/e_2 \in \mathbb{P}_{>0}} \qquad \frac{e \in \mathbb{P}_{\bot}}{!e \in \mathbb{P}_0} \qquad \frac{e \in \mathbb{P}_{\geq 0}}{!e \in \mathbb{P}_{\bot}}$$

Fig. 5. Deriving grammar properties.

4.3 PEG well-formedness

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

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

Theorem 42 ([14]) *If $\mathcal{G}$ is well-formed then it is complete.*

Proof. We will say that (e, s) is complete iff $\exists_{n,r}\ (e, s) \overset{n}{\rightsquigarrow} r$. So we have to prove that (e, s) is complete for all $e \in \mathrm{E}(\mathcal{G})$ and all strings s. We proceed by induction over the length of the string s ($\mathrm{IH}_{\mathrm{out}}$), followed by induction on the depth of the derivation tree of $e \in \mathrm{WF}$ ($\mathrm{IH}_{\mathrm{in}}$). 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 $e_1; e_2 \in \mathrm{WF}$ then $e_1 \in \mathrm{WF}$, so (e_1, s) is complete by $\mathrm{IH}_{\mathrm{in}}$. If e_1 fails then $e_1; e_2$ fails. Otherwise $(e_1, s) \overset{n}{\rightsquigarrow} \sqrt{}^{v}_{s'}$. If $s = s'$ then $e_1 \in \mathbb{P}_0$ (Lemma 41) and hence $e_2 \in \mathrm{WF}$ and (e_2, s') is complete by $\mathrm{IH}_{\mathrm{in}}$. If $s \neq s'$ then $|s'| < |s|$ (Theorem 31) and (e_2, s') is complete by $\mathrm{IH}_{\mathrm{out}}$. Either way (e_2, s') is complete and we conclude by semantical rules for sequence.
- For a repetition $e*$, $e \in \mathrm{WF}$ gives us completeness of (e, s) by $\mathrm{IH}_{\mathrm{in}}$. If e fails then we conclude by the base rule for repetition. Otherwise $(e*, s) \overset{n}{\rightsquigarrow} s'$ with $|s'| < |s|$ as $e \notin \mathbb{P}_0$. Hence we get completeness of $(e*, s')$ by $\mathrm{IH}_{\mathrm{out}}$ 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

$$\frac{A \in \mathcal{V}_N \qquad \mathrm{P}_{\mathrm{exp}}(A) \in \mathrm{WF}}{A \in \mathrm{WF}} \qquad \frac{}{\epsilon \in \mathrm{WF}} \qquad \frac{}{[\cdot] \in \mathrm{WF}} \qquad \frac{a \in \mathcal{V}_T}{[a] \in \mathrm{WF}} \qquad \frac{e \in \mathrm{WF}}{!e \in \mathrm{WF}}$$

$$\frac{e_1 \in \mathrm{WF} \qquad e_1 \in \mathbb{P}_0 \Rightarrow e_2 \in \mathrm{WF}}{e_1; e_2 \in \mathrm{WF}} \qquad \frac{e_1 \in \mathrm{WF} \qquad e_2 \in \mathrm{WF}}{e_1/e_2 \in \mathrm{WF}} \qquad \frac{e \in \mathrm{WF}, \quad e \notin \mathbb{P}_0}{e* \in \mathrm{WF}}$$

Fig. 6. Deriving well-formedness property for a PEG.

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 5. They are specified over a finite enumeration of non-terminals (corresponding to $\mathcal{V}_N$) with their types ($\mathrm{P}_{\mathrm{type}}$):

```
Parameter prods : Enumeration.
Parameter prods_type : prods → Type.
```

We do not parameterize XPEGs by the set of terminals, as for that we simply use the existing *ascii* type of Coq, encoding standard ASCII characters. Building on that we define parsing expressions Δ_α, with the typing discipline from Figure 3 in an expected way. Finally the definitions of non-terminals ($\mathrm{P}_{\mathrm{exp}}$) and the starting production (v_{start}) become:

```
Parameter production : ∀ p : prods, PExp (prods_type p).
Parameter start : prods.
```

There are two observations that we would like to make at this point. First, by means of the above embedding of XPEGs in the logic of 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 7. Figure 7 presents a precise Coq rendering of the productions of the XPEG grammar from Example 6. It is not much more verbose than the original example. The most awkward part are the projections for tuples for which we use a family of functions $Pi\_n(v_1, \ldots, v_i, \ldots, v_n) ::= v_i$ ◁

```
Program Definition production p :=
  match p return PExp (prod_type p) with
   | ws     ⇒ (" " / "\t") [*]          [#]
   | number ⇒ ["0" −− "9"] [+]      [→] digListToRat
   | term ⇒ ws; number; ws            [→] (λv ⇒ P2_3 v)
            / ws; "("; expr; ")"; ws  [→] (λv ⇒ P3_5 v)
   | factor ⇒ term; "*"; factor       [→] (λv ⇒ P1_3 v ∗ P3_3 v)
            / term
   | expr  ⇒ factor; "+"; expr        [→] (λv ⇒ P1_3 v + P3_3 v)
            / factor
  end.
```

Fig. 7. A Coq version of the XPEG for mathematical expressions from Example 6

5.2 Checking well-formedness of an XPEG

To check well-formedness of XPEGs we implement the procedure from Section 4. The main difficulty is 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 new Program feature of Coq makes specifying such functions much easier. We illustrate it on the well-formedness check (computing properties is analogous), which is realized with the following procedure:

```
Program Fixpoint wf_compute (wf : WFset)
  {measure (wf_measure wf)} : WFset :=
  let wf′ := wf_derive wf in
  if PES.equal wf wf′ then wf else wf_compute wf′.
```

where *WFset* is a set of well-formed expressions and *wf_derive* performs one-step derivation with the rules of Figure 6 over $\mathrm{E}(\mathcal{G})$. The measure (into $\mathbb{N}$) is defined as:

$$\mathit{wf\_measure} ::= |\,\mathrm{E}(\mathcal{G})| - |\mathit{wf}\,|$$

We can prove this procedure terminating, as the set of well-formed expressions is growing monotononically and is limited by $\mathrm{E}(\mathcal{G})$:

$$\mathit{wf} \subseteq \mathit{wf\_derive}\ \mathit{wf}$$
$$\mathit{wf} \subseteq \mathrm{E}(\mathcal{G}) \implies \mathit{wf\_derive}\ \mathit{wf} \subseteq \mathrm{E}(\mathcal{G})$$

Please note that our formalized interpreter (more about it in the following section), and hence the analysis sketched above, is based on XPEGs, not on PEGs. However, we still formalized simple parsing expressions, Definition 1 (though not their semantics, Figure 2), and the projection, defined as expected, from Δ_α to Δ.

This is because the well-formedness procedure needs to maintain a set of parsing expressions (*WFset* above) and for that we need a decidable equality over parsing expressions. Equality over Δ_α is not decidable, as, within coercion operator $e[\mapsto]f$ they contain arbitrary functions f, for which we cannot decide equality.

An alternative approach would be to consider *WFset* modulo an equivalence relation on parsing expressions coarser than the syntactic equality, which would ignore f components in $e[\mapsto]f$ coercions. We chose the former approach as developing the PEG analysis and well-formedness check over a non-dependently typed expressions Δ seemed to be easier than over Δ_α and the results carry over to this richer structure immediately.

5.3 A formal interpreter for XPEGs

For the development of a formal interpreter for XPEGs we used the *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.

The only difference in comparison with the theory presented in the preceding sections is that we implemented the range operator $[a\text{-}z]$ as a primitive (so we had to extend the semantics of Figure 4 with this operator), as in practice it occurs frequently in parsers and implementing it with a choice over all the characters in the range is inefficient.

The interpreter is defined as a function with the following header:

$$\begin{array}{l}\textbf{Program Fixpoint}\ \mathit{parse}\ (T : \mathit{Type})\ (e : \mathit{PExp}\ T \mid \mathit{is\_gr\_exp}\ e)\ (s : \mathit{string})\\ \quad \{\textbf{measure}\ (e, s)(\succ)\} :\ \{r : \mathit{ParsingResult}\ T \mid \exists\ n, [e, s] \Rightarrow [n, r]\}\end{array}$$

So this function takes three arguments (the first one implicit):

- T: a type of the result of parsing (α),
- e: a parsing expression of type T (Δ_α), which 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 $(e, s) \overset{n}{\leadsto} r$. So the *parse* function produces the parsing result r (either $\bot$ or $\surd^{v}_{s}$, with $v : T$), such that $(e, s) \overset{n}{\leadsto} r$ for some n, *i.e.*, it is correct with respect to the semantic of XPEGs.

The body of the *parse* function performs pattern matching on expression e and interprets it according to the semantics from Figure 2.

This function is again not structurally recursive, but the recursive calls are decreasing with respect to the following $\succ$ relation on pairs of parsing expressions and strings:

$$(e_1, s_1) \succ (e_2, s_2) \iff \exists_{n_1, r_1, n_2, r_2} (e_1, s_1) \overset{n_1}{\leadsto} r_1 \wedge (e_2, s_2) \overset{n_2}{\leadsto} r_2 \wedge n_1 > n_2$$

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 $>$, the well-founded order on $\mathbb{N}$. Since the semantics of $\mathcal{G}$ is complete (due to Theorem 42 and the check for well-formedness of $\mathcal{G}$ as described in Section 5.2) we can prove that all recursive calls are indeed decreasing with respect to $\succ$.

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 $\mathcal{G}$, to specify it in Coq and, using Coq's extraction capabilities [19], to obtain a certified parser for $\mathcal{G}$. 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 [18], Haskell [23] and Scheme [26]. We use the FSets [11] library, developed using Coq's modules and functors [7], which are not yet supported by extraction to Haskell or Scheme. However, there is an ongoing work on porting FSets to type classes [25], which are supported by extraction. In this section we will describe our experience with OCaml extraction on the example of an XML parser.

A well-known issue with extraction is the performance of obtained programs [8, 19]. 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 [8]. On the other hand the CompCert project [17] 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. We found however that the first version's performance was unacceptable and required several improvements. In the remainder of this section, we will describe those improvements and compare certified TRX's performance with a few other tools.

For our benchmarking experiment, see Figure 8 on the following page, we used the following parsers, all of them OCaml-based to avoid differences coming from the use of different programming languages:

- TRX-cert: the certified TRX interpreter, which is the subject of this paper and is described in more detail in Section 5.
- TRX-int: a simple prototype with comparable functionality to TRX-cert, though developed manually. It does *not* produce a parse tree (just checks whether the input conforms to the grammar).
- TRX-gen: MLstate's own production-used PEG-based parser generator (for experiments we used its simple version without memoization).
- Aurochs [10]: the only PEG-based parser generator (apart from TRX) we are aware of that supports OCaml as the target language. It uses packrat parsing.
- xml-light [6]: a popular XML parser and printer for OCaml, internally using ocamllex for lexical analysis and ocamlyacc for syntactical analysis (based on *LALR(1)* parsing).

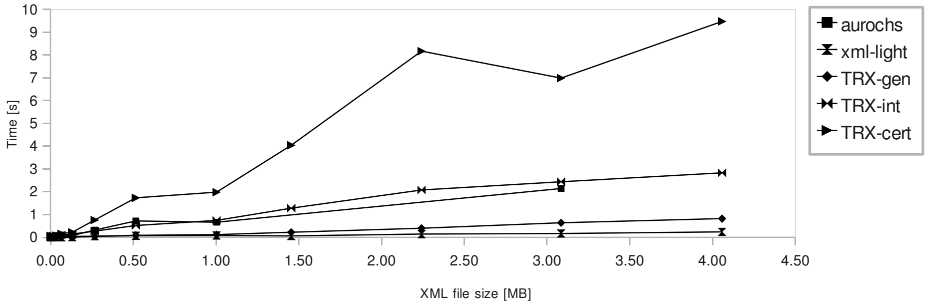

Fig. 8. Performance of certified TRX compared to a number of other tools

6.1 Improving performance of certified TRX

The first extracted version of TRX-cert (not shown on Figure 8) 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 heart 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 *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 shown in Figure 8. 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.

6.2 Performance comparison

Figure 8 plots performance of the 5 aforementioned tools on a number of XML files (the biggest one of more than 4MB). For all PEG-based parsers, that is all tools except xml-light, we used the same PEG grammar (with minor tweaks due to differences in the tools). Few missing values for Aurochs are due to stack overflow errors.

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 2.5x slower, mso certainly there is room for improvement, especially given the fact that for the development of TRX-int we put emphasis on its simplicity (the actual interpreter is around 100 lines long) and not on efficiency.

The two main directions for improving performance seem to be:

- *Memoization* (packrat parsing): it does not help for simple grammars, as that of XML (TRX-gen with memoization is actually slower than without, due to the overhead of keeping the memoization table), but it does pay off for more complex grammars.
- *Code generation*: as witnessed by the difference between TRX-int and TRX-gen turning from interpretation to code generation can have a substantial impact on performance.

Admittedly XML is not the best test-case for TRX, due to its simple format, for which the expressive power offered by PEGs is an overkill. Parsing Java seems to be an established benchmark for PEGs [24, 13, 12, 29]. One difficulty with the grammar of Java [15] is that it naturally contains left-recursive rules, most of which can be easily replaced with iteration, with the exception of a single definition [24], and for the moment TRX lacks the ability to handle left-recursive rules. Also obtaining reasonable (linear) performance for such a complicated grammar would require either packrat parsing or very careful crafting of the grammar. It is reported by Redziejowski [24] that “the resulting primitive parser shows an acceptable behavior, indicating that packrat parsing might be an overkill for practical languages”, but is very sparse on details of what a reasonable performance is.

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. And at the moment TRX is the only tool that comes with such guarantees.

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.

In Danielsson and Norell [9] a library of parser combinators (see Hutton [16]) with termination guarantees has been developed in the dependently typed functional programming language Agda [27]. The main difference in comparison with our work is that they provide a library of combinators, whereas we aim at parser generator for PEG grammars (though at the moment we only have an interpreter). The problem of termination is also handled differently: "[we] use dependent types to add information in the form of type indices to the parser type, and use these indices to ensure that left recursion is not possible" [9]. In many cases those type indices can be automatically inferred by Agda, however, if this is not possible they have to be provided by the user of the library, which requires some expertise and understanding of the underlying formal model. In our approach we proved correct a well-formedness checker for PEG grammars, making the termination analysis completely transparent to the user of TRX.

Ideas similar to Danielsson and Norell [9] were previously put forward, though just as a proof of concept, by McBride and McKinna [21].

Probably the closest work to ours is that of Barthwal and Norrish [3], where the authors developed an SLR parser in HOL. The main differences with our work are:

- PEGs are more expressive that 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" [3] 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. [30], where the given PEG grammar is compiled into an imperative computation within the Ynot framework, that 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 Conclusions and Future Work

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. Although TRX can still be improved (see future work discussion below), it is the first tool capable of generating provably correct parsers. 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.

To extend our research, we identify the following subjects for future work:

1. A realistic *case study* of a practical language, such as Java, should be conducted to ensure scalability of this methodology and acceptable performance. This would also allow us to compare directly with other experiments of parsing Java with PEGs (see for instance Redziejowski [24] or Ford [12]). This would undoubtedly lead to some improvements to TRX making it easier to use.
2. In connection with the aforementioned case study the *performance* of our parser interpreter should be better understood and improved upon. One possibility here is implementation of packrat parsing, by means of implementing memoization in our interpreter [13].
3. Support for *error messages*, for instance following that of the PEG-based parser generator Puppy [12], should be added.
4. 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 [12], *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 [29].

Acknowledgements We would like to thank Matthieu Sozeau for his help with the new Program feature of Coq and the ESOP reviewers for valuable feedback.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling - Vol. I: Parsing*. Prentice Hall, 1972.
3. A. Barthwal and M. Norrish. Verified, executable parsing. In *ESOP '09*, volume 5502 of *Lecture Notes in Computer Science*, pages 160–174, 2009.

4. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.
5. W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley, 1975.
6. N. Cannasse. Xml-light. `http://tech.motion-twin.com/xmllight.html`, 2003.
7. J. Chrzaszcz. Implementing modules in the Coq system. In *TPHOL '03*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286, 2003.
8. L. Cruz-Filipe and P. Letouzey. A large-scale experiment in executing extracted programs. *Electronic Notes in Theoretical Computer Science*, 151(1):75–91, 2006.
9. N. A. Danielsson and U. Norell. Structurally recursive descent parsing, 2008. Draft. `http://www.cs.nott.ac.uk/~nad/publications`.
10. B. Durak. Aurochs. `http://aurochs.fr/`, 2009.
11. J.-C. Filliâtre and P. Letouzey. Functors for proofs and programs. In *ESOP '04*, volume 2986 of *Lecture Notes in Computer Science*, pages 370–384, 2004.
12. B. Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, 2002.
13. B. Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP '02*, pages 36–47, 2002.
14. B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04*, pages 111–122, 2004.
15. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification, 3rd Edition.* Addison-Wesley, 2005.
16. G. Hutton. Higher-order functions for parsing. *The Journal of Functional Programming*, 2(3):323–343, 1992.
17. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
18. X. Leroy et al. Objective caml. `http://caml.inria.fr`, 1996.
19. P. Letouzey. Extraction in Coq: An overview. In *CiE '08*, volume 5028 of *Lecture Notes in Computer Science*, 2008.
20. J. R. Levine, T. Mason, and D. Brown. *Lex & yacc.* O'Reilly, 1992.
21. C. McBride and J. McKinna. Seeing and doing, 2002. Presentation at the Workshop on Termination and Type Theory.
22. T. J. Parr and R. W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *CC '94*, volume 786 of *Lecture Notes in Computer Science*, pages 263–277, 1994.
23. S. Peyton-Jones et al. Haskell 98 language and libraries: The revised report, 2002. `http://haskell.org/`.
24. R. R. Redziejowski. Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Fundamenta Informaticae*, 79(3-4):513–524, 2007.
25. M. Sozeau and N. Oury. First-class type classes. In *TPHOL '08*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293, 2008.
26. G. J. Sussman and G. L. Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.
27. The Agda team. The Agda wiki. `http://wiki.portal.chalmers.se/agda/`, 2008.
28. The Coq Development Team. The Coq proof assistant: Reference manual, version 8.2. `http://coq.inria.fr`, 1989–2009.
29. A. Warth, J. R. Douglass, and T. D. Millstein. Packrat parsers can support left recursion. In *PEPM '08*, pages 103–110, 2008.
30. R. Wisnesky, G. Malecha, and G. Morrisett. Certified web services in Ynot. In *Proceedings of WWV'09*, pages 5–19, 2009.