

TRX: A Formally Verified Parser Interpreter

Adam Koprowski Henri Binsztok

MLstate, Paris, France
<http://mlstate.com>

European Symposium on Programming (ESOP '10)
22 March

MLstate

Outline

- 1 Context
- 2 Objective
- 3 PEGs: Parsing Expression Grammars
- 4 XPEGs: PEGs eXtended with with Semantic Actions
- 5 Termination Analysis for (X)PEGs
- 6 PEG interpreter in Coq
- 7 Performance evaluation
- 8 Related work
- 9 Conclusions

- **SaaS** (Software as a service) is a new model of software deployment that is gaining popularity.
- **OPA** (One-Pot Application) is a new unified technology for developing rich web applications.
The language has built-in: persistency, parsing capabilities, client-code generation capabilities, concurrency, ...
- **Safety** is a distinguished feature of OPA:
 - OPA features a number of static checks for its applications,
 - OPA has a verification platform build on top of it [WIP],
 - OPA components will be subjected to formal verification [WIP].

Context

- **SaaS** (Software as a service) is a new model of software deployment that is gaining popularity.



- **OPA** (One-Pot Application) is a new unified technology for developing rich web applications. The language has built-in: persistency, parsing capabilities, client-code generation capabilities, concurrency, ...
- **Safety** is a distinguished feature of OPA:
 - OPA features a number of static checks for its applica...

- **SaaS** (Software as a service) is a new model of software deployment that is gaining popularity.
- **OPA** (One-Pot Application) is a new unified technology for developing rich web applications.
The language has built-in: persistency, parsing capabilities, client-code generation capabilities, concurrency, ...
- **Safety** is a distinguished feature of OPA:
 - OPA features a number of static checks for its applications,
 - OPA has a verification platform build on top of it [WIP],
 - OPA components will be subjected to formal verification [WIP].

Context

- **SaaS** (Software as a service) is a new model of software deployment that is gaining popularity.
- **OPA** (One-Pot Application) is a new unified technology for developing rich web applications.
The language has built-in: persistency, parsing capabilities, client-code generation capabilities, concurrency, ...

```
// Database
db /wiki: stringmap(string)
db /wiki = "this is a new page"
save(name,content) = /wiki[name] = jQuery.getVal(#{content}); {}

// User interface
text(t) =
  <textarea rows={5} cols={80} id="text">{t}</textarea>

page(name) =
  <h1>{ name:string }</h1>
  <p>{ text /wiki[name] }</p>
  <a onclick={save(name, "text")}>save</a>

main(name) = html("MLstate wiki :: {name}", page(name))

/* Dispatching urls: http://www.myserver.com/some_note should
   point to note_some_note */
urls = parser | "/" ([A-Za-z]+) -> main(Text.to_string( __2 ))
      | "/" -> main("Home")

// Starting the application and setting a default port
server = simple_server(urls, 2009)
```

- **Safety** is a distinguished feature of OPA:
 - OPA features a number of static checks for its applications,
 - OPA has a verification platform build on top of it [M...]
 - OPA components will be subjected to formal verification [M...]

- **SaaS** (Software as a service) is a new model of software deployment that is gaining popularity.
- **OPA** (One-Pot Application) is a new unified technology for developing rich web applications.
The language has built-in: persistency, parsing capabilities, client-code generation capabilities, concurrency, ...
- **Safety is a distinguished feature of OPA:**
 - OPA features a number of static checks for its applications,
 - OPA has a verification platform build on top of it [WIP],
 - OPA components will be subjected to formal verification [WIP].

- **SaaS** (Software as a service) is a new model of software deployment that is gaining popularity.
- **OPA** (One-Pot Application) is a new unified technology for developing rich web applications.
The language has built-in: persistency, parsing capabilities, client-code generation capabilities, concurrency, ...
- **Safety** is a distinguished feature of OPA:
 - OPA features a number of static checks for its applications,
 - OPA has a verification platform build on top of it [WIP],
 - OPA components will be subjected to formal verification [WIP].

- **SaaS** (Software as a service) is a new model of software deployment that is gaining popularity.
- **OPA** (One-Pot Application) is a new unified technology for developing rich web applications.
The language has built-in: persistency, parsing capabilities, client-code generation capabilities, concurrency, ...
- **Safety** is a distinguished feature of OPA:
 - OPA features a number of static checks for its applications,
 - **OPA has a verification platform build on top of it [WIP],**
 - OPA components will be subjected to formal verification [WIP].

- **SaaS** (Software as a service) is a new model of software deployment that is gaining popularity.
- **OPA** (One-Pot Application) is a new unified technology for developing rich web applications.
The language has built-in: persistency, parsing capabilities, client-code generation capabilities, concurrency, ...
- **Safety** is a distinguished feature of OPA:
 - OPA features a number of static checks for its applications,
 - OPA has a verification platform build on top of it [WIP],
 - **OPA components will be subjected to formal verification [WIP].**

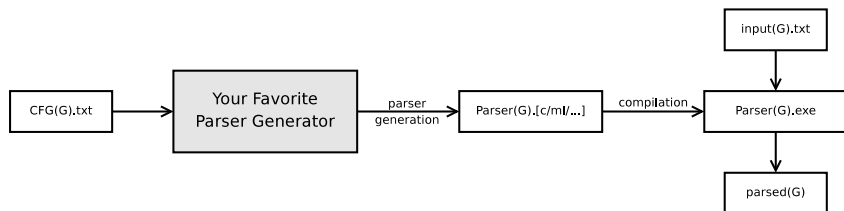
- **SaaS** (Software as a service) is a new model of software deployment that is gaining popularity.
- **OPA** (One-Pot Application) is a new unified technology for developing rich web applications.
The language has built-in: persistency, **parsing capabilities**, client-code generation capabilities, concurrency, ...
- **Safety** is a distinguished feature of OPA:
 - OPA features a number of static checks for its applications,
 - OPA has a verification platform build on top of it [WIP],
 - OPA components will be subjected to **formal verification** [WIP].

Objective

- Parsing is a well-known and well-studied topic.
- Typical approach to parsing is by means of parser generators:
- ... can we use formal methods to verify some properties of generated parsers?

Objective

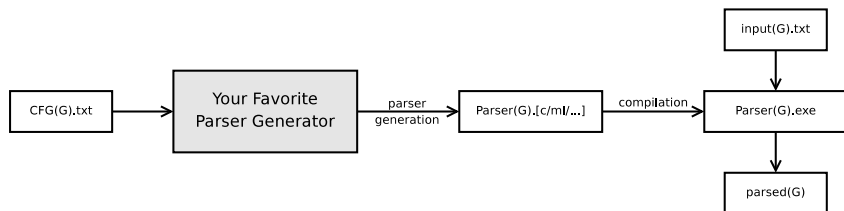
- Parsing is a well-known and well-studied topic.
- Typical approach to parsing is by means of parser generators:



- ... can we use formal methods to verify some properties of generated parsers?

Objective

- Parsing is a well-known and well-studied topic.
- Typical approach to parsing is by means of parser generators:



- ... can we use formal methods to verify some properties of generated parsers?

Main reference



Bryan Ford

Parsing expression grammars: a recognition-based syntactic foundation

POPL '04

Parsing expressions

Definition

Parsing expressions over non-terminals \mathcal{V}_N and terminals \mathcal{V}_T :

$\Delta :=$	ϵ	empty expression	
	$[\cdot]$	any character	
	$[a]$	a terminal symbol	$(a \in \mathcal{V}_T)$
	$[“s”]$	a literal	$(s \in \mathcal{S})$
	$[a-z]$	a range	$(a, z \in \mathcal{V}_T)$
	A	a non-terminal	$(A \in \mathcal{V}_N)$
	$e_1; e_2$	a sequence	$(e_1, e_2 \in \Delta)$
	e_1/e_2	a <i>prioritized</i> choice	$(e_1, e_2 \in \Delta)$
	e^*	a zero-or-more <i>greedy</i> repetition	$(e \in \Delta)$
	e^+	a one-or-more <i>greedy</i> repetition	$(e \in \Delta)$
	$e?$	an optional expression	$(e \in \Delta)$
	$!e$	a not-predicate	$(e \in \Delta)$
	$\&e$	an and-predicate	$(e \in \Delta)$

Parsing expressions

Definition

Parsing expressions over non-terminals \mathcal{V}_N and terminals \mathcal{V}_T :

$\Delta :=$	ϵ	empty expression	
	$[\cdot]$	any character	
	$[a]$	a terminal symbol	$(a \in \mathcal{V}_T)$
	$["s"] := [s_1]; \dots; [s_n]$	a literal	$(s \in \mathcal{S})$
	$[a-z] := [a] / \dots / [z]$	a range	$(a, z \in \mathcal{V}_T)$
	A	a non-terminal	$(A \in \mathcal{V}_N)$
	$e_1; e_2$	a sequence	$(e_1, e_2 \in \Delta)$
	e_1/e_2	a <i>prioritized</i> choice	$(e_1, e_2 \in \Delta)$
	e^*	a zero-or-more <i>greedy</i> repetition	$(e \in \Delta)$
	$e+ := e; e^*$	a one-or-more <i>greedy</i> repetition	$(e \in \Delta)$
	$e? := e/\epsilon$	an optional expression	$(e \in \Delta)$
	$!e$	a not-predicate	$(e \in \Delta)$
	$\&e := !!e$	an and-predicate	$(e \in \Delta)$

Parsing expressions

Definition

Parsing expressions over non-terminals \mathcal{V}_N and terminals \mathcal{V}_T :

$\Delta :=$	ϵ	empty expression	
	$[\cdot]$	any character	
	$[a]$	a terminal symbol	$(a \in \mathcal{V}_T)$
	$[“s”] := [s_1]; \dots; [s_n]$	a literal	$(s \in \mathcal{S})$
	$[a-z] := [a] / \dots / [z]$	a range	$(a, z \in \mathcal{V}_T)$
	A	a non-terminal	$(A \in \mathcal{V}_N)$
	$e_1; e_2$	a sequence	$(e_1, e_2 \in \Delta)$
	e_1/e_2	a prioritized choice	$(e_1, e_2 \in \Delta)$
	e^*	a zero-or-more greedy repetition	$(e \in \Delta)$
	$e+ := e; e^*$	a one-or-more greedy repetition	$(e \in \Delta)$
	$e? := e/\epsilon$	an optional expression	$(e \in \Delta)$
	$!e$	a not-predicate	$(e \in \Delta)$
	$\&e := !!e$	an and-predicate	$(e \in \Delta)$

PEG example

Example

A very simple PEG for mathematical expressions:

```
ws := ([\s] / [\\t])*  
number := [0-9]+  
term := ws number ws / ws ([ expr ]) ws  
factor := term [*] factor / term  
expr := factor [+] expr / factor
```

Example

How about making addition left-associative?

```
expr := expr [+] factor / factor
```

... but that makes the grammar *left-recursive*, leading to a *non-terminating* parser.

PEG example

Example

A very simple PEG for mathematical expressions:

```
ws := ([\ ] / [\t])*  
number := [0-9]+  
term := ws number ws / ws [(] expr [)] ws  
factor := term [*] factor / term  
expr := factor [+] expr / factor
```

Example

How about making addition left-associative?

```
expr := expr [+] factor / factor
```

... but that makes the grammar *left-recursive*, leading to a *non-terminating* parser.

PEG example

Example

A very simple PEG for mathematical expressions:

```
ws := ([\ ] / [\t])*  
number := [0-9]+  
term := ws number ws / ws ([ (] expr [ ) ]) ws  
factor := term [*] factor / term  
expr := factor [+] expr / factor
```

Example

How about making addition left-associative?

```
expr := expr [+] factor / factor
```

... but that makes the grammar *left-recursive*, leading to a *non-terminating* parser.

Examples ctd.

Example (Reserved words)

A PEG grammar for recognizing identifiers that are not reserved words:

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

Example ("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 (e_1) if (e_2) s_1 else s_2

This ambiguity is easy to resolve using PEGs.

Examples ctd.

Example (Reserved words)

A PEG grammar for recognizing identifiers that are not reserved words:

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

Example ("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 (e_1) if (e_2) s_1 else s_2

This ambiguity is easy to resolve using PEGs.

Formal definition of PEGs

Definition (PEG)

A parsing expressions grammar (PEG), \mathcal{G} , is a quadruple $(\mathcal{V}_N, \mathcal{V}_T, P_{\text{exp}}, e_s)$, where:

- \mathcal{V}_T is a finite set of terminals,
- \mathcal{V}_N is a finite set of non-terminals of the grammar,
- P_{exp} is the interpretation of the productions of the grammar, *i.e.*,
 $P_{\text{exp}} : \mathcal{V}_N \rightarrow \Delta$ and
- e_s is the start production of the grammar, $e_s \in \mathcal{V}_N$.

Definition

Semantics of PEGs:

- $(e, s) \xrightarrow{m} \perp$: expression e fails on input string s .
- $(e, s) \xrightarrow{m} \sqrt{s'}$: expression e succeeds on s , leaving s' to be parsed.

Formal definition of PEGs

Definition (PEG)

A parsing expressions grammar (PEG), \mathcal{G} , is a quadruple $(\mathcal{V}_N, \mathcal{V}_T, P_{\text{exp}}, e_s)$, where:

- \mathcal{V}_T is a finite set of terminals,
- \mathcal{V}_N is a finite set of non-terminals of the grammar,
- P_{exp} is the interpretation of the productions of the grammar, *i.e.*, $P_{\text{exp}} : \mathcal{V}_N \rightarrow \Delta$ and
- e_s is the start production of the grammar, $e_s \in \mathcal{V}_N$.

Definition

Semantics of PEGs:

- $(e, s) \xrightarrow{m} \perp$: expression e fails on input string s .
- $(e, s) \xrightarrow{m} \sqrt{s'}$: expression e succeeds on s , leaving s' to be parsed.

Definition (PEGs semantics, (1/2))

$$\frac{}{(\epsilon, s) \overset{1}{\rightsquigarrow} \sqrt{s}} \qquad \frac{(\text{P}_{\text{exp}}(p), s) \overset{m}{\rightsquigarrow} r}{(p, s) \overset{m+1}{\rightsquigarrow} r}$$

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

$$\frac{}{(x, x :: xs) \overset{1}{\rightsquigarrow} \sqrt{xs}} \qquad \frac{}{(x, []) \overset{1}{\rightsquigarrow} \perp}$$

$$\frac{x \neq y}{(y, x :: xs) \overset{1}{\rightsquigarrow} \perp}$$

Definition (PEGs semantics, (1/2))

$$\frac{}{(\epsilon, s) \overset{1}{\rightsquigarrow} \sqrt{s}}$$

$$\frac{(P_{\text{exp}}(p), s) \overset{m}{\rightsquigarrow} r}{(p, s) \overset{m+1}{\rightsquigarrow} r}$$

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

$$\frac{}{([\cdot], []) \overset{1}{\rightsquigarrow} \perp}$$

$$\frac{}{(x, x :: xs) \overset{1}{\rightsquigarrow} \sqrt{xs}}$$

$$\frac{}{(x, []) \overset{1}{\rightsquigarrow} \perp}$$

$$\frac{x \neq y}{(y, x :: xs) \overset{1}{\rightsquigarrow} \perp}$$

Definition (PEGs semantics, (1/2))

$$\frac{}{(\epsilon, s) \overset{1}{\rightsquigarrow} \sqrt{s}} \qquad \frac{(\text{P}_{\text{exp}}(p), s) \overset{m}{\rightsquigarrow} r}{(p, s) \overset{m+1}{\rightsquigarrow} r}$$

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

$$\frac{}{(x, x :: xs) \overset{1}{\rightsquigarrow} \sqrt{xs}} \qquad \frac{}{(x, []) \overset{1}{\rightsquigarrow} \perp}$$

$$\frac{x \neq y}{(y, x :: xs) \overset{1}{\rightsquigarrow} \perp}$$

Definition (PEGs semantics (2/2))

$$\frac{(e_1, s) \xrightarrow{m} \perp}{(e_1; e_2, s) \xrightarrow{m+1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \sqrt{s'} \quad (e_2, s') \xrightarrow{n} r}{(e_1; e_2, s) \xrightarrow{m+n+1} r}$$

$$\frac{(e_1, s) \xrightarrow{m} \sqrt{s'}}{(e_1/e_2, s) \xrightarrow{m+1} \sqrt{s'}}$$

$$\frac{(e_1, s) \xrightarrow{m} \perp \quad (e_2, s) \xrightarrow{n} r}{(e_1/e_2, s) \xrightarrow{m+n+1} r}$$

$$\frac{(e, s) \xrightarrow{m} \sqrt{s'} \quad (e^*, s') \xrightarrow{n} \sqrt{s''}}{(e^*, s) \xrightarrow{m+n+1} \sqrt{s''}}$$

$$\frac{(e, s) \xrightarrow{m} \perp}{(e^*, s) \xrightarrow{m+1} \sqrt{s}}$$

$$\frac{(e, s) \xrightarrow{m} \perp}{(!e, s) \xrightarrow{m+1} \sqrt{s}}$$

$$\frac{(e, s) \xrightarrow{m} \sqrt{s'}}{(!e, s) \xrightarrow{m+1} \perp}$$

Definition (PEGs semantics (2/2))

$$\frac{(e_1, s) \xrightarrow{m} \perp}{(e_1; e_2, s) \xrightarrow{m+1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \sqrt{s'} \quad (e_2, s') \xrightarrow{n} r}{(e_1; e_2, s) \xrightarrow{m+n+1} r}$$

$$\frac{(e_1, s) \xrightarrow{m} \sqrt{s'}}{(e_1/e_2, s) \xrightarrow{m+1} \sqrt{s'}}$$

$$\frac{(e_1, s) \xrightarrow{m} \perp \quad (e_2, s) \xrightarrow{n} r}{(e_1/e_2, s) \xrightarrow{m+n+1} r}$$

$$\frac{(e, s) \xrightarrow{m} \sqrt{s'} \quad (e^*, s') \xrightarrow{n} \sqrt{s''}}{(e^*, s) \xrightarrow{m+n+1} \sqrt{s''}}$$

$$\frac{(e, s) \xrightarrow{m} \perp}{(e^*, s) \xrightarrow{m+1} \sqrt{s}}$$

$$\frac{(e, s) \xrightarrow{m} \perp}{(!e, s) \xrightarrow{m+1} \sqrt{s}}$$

$$\frac{(e, s) \xrightarrow{m} \sqrt{s'}}{(!e, s) \xrightarrow{m+1} \perp}$$

Definition (PEGs semantics (2/2))

$$\frac{(e_1, s) \overset{m}{\rightsquigarrow} \perp}{(e_1; e_2, s) \overset{m+1}{\rightsquigarrow} \perp}$$

$$\frac{(e_1, s) \overset{m}{\rightsquigarrow} \sqrt{s'} \quad (e_2, s') \overset{n}{\rightsquigarrow} r}{(e_1; e_2, s) \overset{m+n+1}{\rightsquigarrow} r}$$

$$\frac{(e_1, s) \overset{m}{\rightsquigarrow} \sqrt{s'}}{(e_1/e_2, s) \overset{m+1}{\rightsquigarrow} \sqrt{s'}}$$

$$\frac{(e_1, s) \overset{m}{\rightsquigarrow} \perp \quad (e_2, s) \overset{n}{\rightsquigarrow} r}{(e_1/e_2, s) \overset{m+n+1}{\rightsquigarrow} r}$$

$$\frac{(e, s) \overset{m}{\rightsquigarrow} \sqrt{s'} \quad (e^*, s') \overset{n}{\rightsquigarrow} \sqrt{s''}}{(e^*, s) \overset{m+n+1}{\rightsquigarrow} \sqrt{s''}}$$

$$\frac{(e, s) \overset{m}{\rightsquigarrow} \perp}{(e^*, s) \overset{m+1}{\rightsquigarrow} \sqrt{s}}$$

$$\frac{(e, s) \overset{m}{\rightsquigarrow} \perp}{(!e, s) \overset{m+1}{\rightsquigarrow} \sqrt{s}}$$

$$\frac{(e, s) \overset{m}{\rightsquigarrow} \sqrt{s'}}{(!e, s) \overset{m+1}{\rightsquigarrow} \perp}$$

PEGs in a nutshell

- Expressiveness:

- ✓ PEGs are *unambiguous*,
- ✓ PEGs can handle *lexical analysis*.
- ✓ PEGs support *backtracking* and *unlimited lookahead*
- ✗ ... but cannot easily handle *left-recursion*.
- ✓ PEGs can parse all *LL(k)* and *LR(k)* languages... and more (including non-context-free grammars).

- Implementation:

- ✓ Allow very easy implementation by means of a recursive descent parser.
- ✗ Simple implementation can result in *exponential time* for parsing.
- ✓ Using memoization ensures *linear time* complexity (packrat parsing),
- ✗ ... but is expensive in terms of *memory consumption*.

PEGs in a nutshell

- Expressiveness:
 - ✓ PEGs are *unambiguous*,
 - ✓ PEGs can handle *lexical analysis*.
 - ✓ PEGs support *backtracking* and *unlimited lookahead*
 - ✗ ... but cannot easily handle *left-recursion*.
 - ✓ PEGs can parse all *LL(k)* and *LR(k)* languages... and more (including non-context-free grammars).
- Implementation:
 - ✓ Allow very easy implementation by means of a recursive descent parser.
 - ✗ Simple implementation can result in *exponential time* for parsing.
 - ✓ Using memoization ensures *linear time* complexity (packrat parsing),
 - ✗ ... but is expensive in terms of *memory consumption*.

So far we can recognize whether a string belongs to the language defined by \mathcal{G} .

We need to be able to get a parse trace (AST).

For that we extend the type of parsing expressions Δ to 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 we introduce a new construct: coercion, $e[\mapsto]f$, which converts a semantic value v associated with e to $f(v)$.

So far we can recognize whether a string belongs to the language defined by \mathcal{G} .

We need to be able to get a parse trace (AST).

For that we extend the type of parsing expressions Δ to 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 we introduce a new construct: coercion, $e[\mapsto]f$, which converts a semantic value v associated with e to $f(v)$.

So far we can recognize whether a string belongs to the language defined by \mathcal{G} .

We need to be able to get a parse trace (AST).

For that we extend the type of parsing expressions Δ to 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 we introduce a new construct: coercion, $e[\mapsto]f$, which converts a semantic value v associated with e to $f(v)$.

Semantic Actions

So far we can recognize whether a string belongs to the language defined by \mathcal{G} .

We need to be able to get a parse trace (AST).

For that we extend the type of parsing expressions Δ to 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 we introduce a new construct: coercion, $e[\mapsto]f$, which converts a semantic value v associated with e to $f(v)$.

So far we can recognize whether a string belongs to the language defined by \mathcal{G} .

We need to be able to get a parse trace (AST).

For that we extend the type of parsing expressions Δ to 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 we introduce a new construct: coercion, $e[\mapsto]f$, which converts a semantic value v associated with e to $f(v)$.

Typing discipline

Definition (XPEGs typing)

$$\frac{}{\epsilon \in \Delta_{\text{True}}}$$

$$\frac{}{[\cdot] \in \Delta_{\text{char}}}$$

$$\frac{a \in \mathcal{V}_T}{a \in \Delta_{\text{char}}}$$

$$\frac{A \in \mathcal{V}_N}{A \in \Delta_{\text{P}_{\text{type}}(A)}}$$

$$\frac{e_1 \in \Delta_{\alpha} \quad e_2 \in \Delta_{\beta}}{e_1; e_2 \in \Delta_{\alpha * \beta}}$$

$$\frac{e_1 \in \Delta_{\alpha} \quad e_2 \in \Delta_{\alpha}}{e_1 / e_2 \in \Delta_{\alpha}}$$

$$\frac{e \in \Delta_{\alpha}}{e^* \in \Delta_{\text{list } \alpha}}$$

$$\frac{e \in \Delta_{\alpha}}{!e \in \Delta_{\text{True}}}$$

$$\frac{e \in \Delta_{\alpha} \quad f : \alpha \rightarrow \beta}{e[\mapsto]f \in \Delta_{\beta}}$$

Typing discipline

Definition (XPEGs typing)

$$\frac{}{\epsilon \in \Delta_{\text{True}}}$$

$$\frac{}{[\cdot] \in \Delta_{\text{char}}}$$

$$\frac{a \in \mathcal{V}_T}{a \in \Delta_{\text{char}}}$$

$$\frac{A \in \mathcal{V}_N}{A \in \Delta_{\text{P}_{\text{type}}(A)}}$$

$$\frac{e_1 \in \Delta_{\alpha} \quad e_2 \in \Delta_{\beta}}{e_1; e_2 \in \Delta_{\alpha * \beta}}$$

$$\frac{e_1 \in \Delta_{\alpha} \quad e_2 \in \Delta_{\alpha}}{e_1 / e_2 \in \Delta_{\alpha}}$$

$$\frac{e \in \Delta_{\alpha}}{e^* \in \Delta_{\text{list } \alpha}}$$

$$\frac{e \in \Delta_{\alpha}}{!e \in \Delta_{\text{True}}}$$

$$\frac{e \in \Delta_{\alpha} \quad f : \alpha \rightarrow \beta}{e[\mapsto]f \in \Delta_{\beta}}$$

Typing discipline

Definition (XPEGs typing)

$$\frac{}{\epsilon \in \Delta_{\text{True}}}$$

$$\frac{}{[\cdot] \in \Delta_{\text{char}}}$$

$$\frac{a \in \mathcal{V}_T}{a \in \Delta_{\text{char}}}$$

$$\frac{A \in \mathcal{V}_N}{A \in \Delta_{\text{P}_{\text{type}}(A)}}$$

$$\frac{e_1 \in \Delta_{\alpha} \quad e_2 \in \Delta_{\beta}}{e_1; e_2 \in \Delta_{\alpha * \beta}}$$

$$\frac{e_1 \in \Delta_{\alpha} \quad e_2 \in \Delta_{\alpha}}{e_1 / e_2 \in \Delta_{\alpha}}$$

$$\frac{e \in \Delta_{\alpha}}{e^* \in \Delta_{\text{list } \alpha}}$$

$$\frac{e \in \Delta_{\alpha}}{!e \in \Delta_{\text{True}}}$$

$$\frac{e \in \Delta_{\alpha} \quad f : \alpha \rightarrow \beta}{e[\mapsto]f \in \Delta_{\beta}}$$

Typing discipline

Definition (XPEGs typing)

$$\frac{}{\epsilon \in \Delta_{\text{True}}}$$

$$\frac{}{[\cdot] \in \Delta_{\text{char}}}$$

$$\frac{a \in \mathcal{V}_T}{a \in \Delta_{\text{char}}}$$

$$\frac{A \in \mathcal{V}_N}{A \in \Delta_{\text{P}_{\text{type}}(A)}}$$

$$\frac{e_1 \in \Delta_{\alpha} \quad e_2 \in \Delta_{\beta}}{e_1; e_2 \in \Delta_{\alpha * \beta}}$$

$$\frac{e_1 \in \Delta_{\alpha} \quad e_2 \in \Delta_{\alpha}}{e_1 / e_2 \in \Delta_{\alpha}}$$

$$\frac{e \in \Delta_{\alpha}}{e^* \in \Delta_{\text{list } \alpha}}$$

$$\frac{e \in \Delta_{\alpha}}{!e \in \Delta_{\text{True}}}$$

$$\frac{e \in \Delta_{\alpha} \quad f : \alpha \rightarrow \beta}{e[\mapsto]f \in \Delta_{\beta}}$$

Typing discipline

Definition (XPEGs typing)

$$\frac{}{\epsilon \in \Delta_{\text{True}}}$$

$$\frac{}{[\cdot] \in \Delta_{\text{char}}}$$

$$\frac{a \in \mathcal{V}_T}{a \in \Delta_{\text{char}}}$$

$$\frac{A \in \mathcal{V}_N}{A \in \Delta_{\text{P}_{\text{type}}(A)}}$$

$$\frac{e_1 \in \Delta_{\alpha} \quad e_2 \in \Delta_{\beta}}{e_1; e_2 \in \Delta_{\alpha * \beta}}$$

$$\frac{e_1 \in \Delta_{\alpha} \quad e_2 \in \Delta_{\alpha}}{e_1 / e_2 \in \Delta_{\alpha}}$$

$$\frac{e \in \Delta_{\alpha}}{e * \in \Delta_{\text{list } \alpha}}$$

$$\frac{e \in \Delta_{\alpha}}{!e \in \Delta_{\text{True}}}$$

$$\frac{e \in \Delta_{\alpha} \quad f : \alpha \rightarrow \beta}{e[\mapsto]f \in \Delta_{\beta}}$$

Example

The typed versions of our derived operators become:

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

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

Definition (Extended Parsing Expressions Grammar (XPEG))

An extended parsing expressions grammar (XPEG), \mathcal{G} , is a tuple $(\mathcal{V}_N, \mathcal{V}_T, P_{\text{type}}, P_{\text{exp}}, v_{\text{start}})$, where:

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

Definition (Semantics of XPEGs (1/2))

$$\frac{}{(\epsilon, s) \xrightarrow{1} \sqrt{s}}$$

$$\frac{(P_{\text{exp}}(p), s) \xrightarrow{m} r}{(p, s) \xrightarrow{m+1} r}$$

$$\frac{}{([\cdot], x :: xs) \xrightarrow{1} \sqrt{x}_{xs}}$$

$$\frac{}{([\cdot], []) \xrightarrow{1} \perp}$$

$$\frac{}{(x, x :: xs) \xrightarrow{1} \sqrt{x}_{xs}}$$

$$\frac{}{(x, []) \xrightarrow{1} \perp}$$

$$\frac{x \neq y}{(y, x :: xs) \xrightarrow{1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \perp}{(e_1; e_2, s) \xrightarrow{m+1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \sqrt{v_1}_{s'} \quad (e_2, s') \xrightarrow{n} \perp}{(e_1; e_2, s) \xrightarrow{m+n+1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \sqrt{v_1}_{s'} \quad (e_2, s') \xrightarrow{n} \sqrt{v_2}_{s''}}{(e_1; e_2, s) \xrightarrow{m+n+1} \sqrt{(v_1, v_2)}_{s''}}$$

Definition (Semantics of XPEGs (1/2))

$$\frac{}{(\epsilon, s) \xrightarrow{1} \sqrt{s}^I}$$

$$\frac{(P_{\text{exp}}(p), s) \xrightarrow{m} r}{(p, s) \xrightarrow{m+1} r}$$

$$\frac{}{([\cdot], x :: xs) \xrightarrow{1} \sqrt{xs}^X}$$

$$\frac{}{([\cdot], []) \xrightarrow{1} \perp}$$

$$\frac{}{(x, x :: xs) \xrightarrow{1} \sqrt{xs}^X}$$

$$\frac{}{(x, []) \xrightarrow{1} \perp}$$

$$\frac{x \neq y}{(y, x :: xs) \xrightarrow{1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \perp}{(e_1; e_2, s) \xrightarrow{m+1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \sqrt{s'}^{v_1} \quad (e_2, s') \xrightarrow{n} \perp}{(e_1; e_2, s) \xrightarrow{m+n+1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \sqrt{s'}^{v_1} \quad (e_2, s') \xrightarrow{n} \sqrt{s''}^{v_2}}{(e_1; e_2, s) \xrightarrow{m+n+1} \sqrt{s''}^{(v_1, v_2)}}$$

Definition (Semantics of XPEGs (1/2))

$$\frac{}{(\epsilon, s) \xrightarrow{1} \sqrt{s}}$$

$$\frac{(P_{\text{exp}}(p), s) \xrightarrow{m} r}{(p, s) \xrightarrow{m+1} r}$$

$$\frac{}{([\cdot], x :: xs) \xrightarrow{1} \sqrt{x}_{xs}}$$

$$\frac{}{([\cdot], []) \xrightarrow{1} \perp}$$

$$\frac{}{(x, x :: xs) \xrightarrow{1} \sqrt{x}_{xs}}$$

$$\frac{}{(x, []) \xrightarrow{1} \perp}$$

$$\frac{x \neq y}{(y, x :: xs) \xrightarrow{1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \perp}{(e_1; e_2, s) \xrightarrow{m+1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \sqrt{v_1}_{s'} \quad (e_2, s') \xrightarrow{n} \perp}{(e_1; e_2, s) \xrightarrow{m+n+1} \perp}$$

$$\frac{(e_1, s) \xrightarrow{m} \sqrt{v_1}_{s'} \quad (e_2, s') \xrightarrow{n} \sqrt{v_2}_{s''}}{(e_1; e_2, s) \xrightarrow{m+n+1} \sqrt{(v_1, v_2)}_{s''}}$$

Some meta-properties of XPEGs

Lemma

If $(e, s) \rightsquigarrow \sqrt{s}^v$ then $(e, s) \not\rightsquigarrow r$ for all r .*

Theorem

If $(e, s) \xrightarrow{m} \sqrt{s'}^v$ then s' is a suffix of s .

Theorem (unambiguity)

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

Some meta-properties of XPEGs

Lemma

If $(e, s) \rightsquigarrow \sqrt{s}^v$ then $(e*, s) \not\rightsquigarrow r$ for all r .

Theorem

If $(e, s) \overset{m}{\rightsquigarrow} \sqrt{s'}^v$ then s' is a suffix of s .

Theorem (unambiguity)

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

Some meta-properties of XPEGs

Lemma

If $(e, s) \rightsquigarrow \sqrt{s}^v$ then $(e*, s) \not\rightsquigarrow r$ for all r .

Theorem

If $(e, s) \overset{m}{\rightsquigarrow} \sqrt{s'}^v$ then s' is a suffix of s .

Theorem (unambiguity)

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

Termination problem for PEGs

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

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

1 with Coq we get that for free.

2 in general: undecidable

but conservative analysis is sufficient in practice (essentially complete)

Termination problem for PEGs

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

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

1 with Coq we get that for free.

2 in general: undecidable

but conservative analysis is sufficient in practice (essentially complete)

Termination problem for PEGs

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

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

Example

Remember:

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

But also:

$$A := B / C ! D A$$

if B may fail and C and D may succeed, the former without consuming any input ... and it can get even worse with mutual recursion.

- 1 with Coq we get that for free.
- 2 in general: undecidable

but conservative analysis is sufficient in practice (essentially)

Termination problem for PEGs

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

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

Example

Remember:

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

But also:

$$A := B / C !D A$$

if B may fail and C and D may succeed, the former without consuming any input ... and it can get even worse with mutual recursion.

- 1 with Coq we get that for free.
- 2 in general: undecidable

but conservative analysis is sufficient in practice (essentially)

MLstate

Termination problem for PEGs

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

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

Example

Remember:

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

But also:

$$A := B / C !D A$$

if B may fail and C and D may succeed, the former without consuming any input ... and it can get even worse with mutual recursion.

- 1 with Coq we get that for free.
- 2 in general: undecidable

but conservative analysis is sufficient in practice (essentially)

MLstate

Termination problem for PEGs

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

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

1 with Coq we get that for free.

2 in general: undecidable

but conservative analysis is sufficient in practice (essentially complete)

Termination problem for PEGs

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

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

1 with Coq we get that for free.

2 **in general: undecidable**

but conservative analysis is sufficient in practice (essentially: complete)

Termination problem for PEGs

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

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

1 with Coq we get that for free.

2 in general: undecidable

but conservative analysis is sufficient in practice (essentially: complete)

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,
- “ \perp ”: parsing expression can fail.

$e \in \mathbb{P}_0$ means that expression e has property “0”.

$e \in \mathbb{P}_{\geq 0} := e \in \mathbb{P}_0 \vee e \in \mathbb{P}_{>0}$.

Theorem

For arbitrary $e \in \Delta$ and $s \in \mathcal{S}$:

$e \in \mathbb{P}_0 \iff \exists s \in \mathcal{S}. e \rightarrow s$

$e \in \mathbb{P}_{>0} \iff \exists s \in \mathcal{S}. e \rightarrow s$

$e \in \mathbb{P}_{\perp} \iff \exists s \in \mathcal{S}. e \rightarrow s$

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,
- “ \perp ”: parsing expression can fail.

$e \in \mathbb{P}_0$ means that expression e has property “0”.

$e \in \mathbb{P}_{\geq 0} := e \in \mathbb{P}_0 \vee e \in \mathbb{P}_{> 0}$.

Theorem

For arbitrary $e \in \Delta$ and $s \in \mathcal{S}$:

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

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,
- “ \perp ”: parsing expression can fail.

$e \in \mathbb{P}_0$ means that expression e has property “0”.

$e \in \mathbb{P}_{\geq 0} := e \in \mathbb{P}_0 \vee e \in \mathbb{P}_{> 0}$.

Theorem

For arbitrary $e \in \Delta$ and $s \in \mathcal{S}$:

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

Definition (PEG analysis (1/2))

$$\begin{array}{ccccc}
 \frac{}{\epsilon \in \mathbb{P}_0} & \frac{}{[\cdot] \in \mathbb{P}_{>0}} & \frac{}{[\cdot] \in \mathbb{P}_\perp} & \frac{a \in \mathcal{V}_T}{[a] \in \mathbb{P}_{>0}} & \frac{a \in \mathcal{V}_T}{[a] \in \mathbb{P}_\perp} \\
 \frac{e \in \mathbb{P}_\perp}{e^* \in \mathbb{P}_0} & \frac{e \in \mathbb{P}_{>0}}{e^* \in \mathbb{P}_{>0}} & \frac{e \in \mathbb{P}_\perp}{!e \in \mathbb{P}_0} & \frac{e \in \mathbb{P}_{\geq 0}}{!e \in \mathbb{P}_\perp} &
 \end{array}$$

$$\frac{\star \in \{0, > 0, \perp\} \quad A \in \mathcal{V}_N \quad P_{\text{exp}}(A) \in \mathbb{P}_\star}{A \in \mathbb{P}_\star}$$

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

$$\frac{e_1 \in \mathbb{P}_0 \quad e_2 \in \mathbb{P}_0}{e_1; e_2 \in \mathbb{P}_0}$$

Definition (PEG analysis (1/2))

$$\begin{array}{ccccc}
 \overline{\epsilon \in \mathbb{P}_0} & \overline{[\cdot] \in \mathbb{P}_{>0}} & \overline{[\cdot] \in \mathbb{P}_\perp} & \frac{a \in \mathcal{V}_T}{[a] \in \mathbb{P}_{>0}} & \frac{a \in \mathcal{V}_T}{[a] \in \mathbb{P}_\perp} \\
 \overline{e \in \mathbb{P}_\perp} & \overline{e \in \mathbb{P}_{>0}} & \overline{e \in \mathbb{P}_\perp} & \overline{e \in \mathbb{P}_{\geq 0}} & \\
 \overline{e^* \in \mathbb{P}_0} & \overline{e^* \in \mathbb{P}_{>0}} & \overline{!e \in \mathbb{P}_0} & \overline{!e \in \mathbb{P}_\perp} &
 \end{array}$$

$$\frac{\star \in \{0, > 0, \perp\} \quad A \in \mathcal{V}_N \quad P_{\text{exp}}(A) \in \mathbb{P}_\star}{A \in \mathbb{P}_\star}$$

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

$$\frac{e_1 \in \mathbb{P}_0 \quad e_2 \in \mathbb{P}_0}{e_1; e_2 \in \mathbb{P}_0}$$

Definition (PEG analysis (1/2))

$$\begin{array}{ccccc}
 \overline{\epsilon \in \mathbb{P}_0} & \overline{[\cdot] \in \mathbb{P}_{>0}} & \overline{[\cdot] \in \mathbb{P}_\perp} & \frac{a \in \mathcal{V}_T}{[a] \in \mathbb{P}_{>0}} & \frac{a \in \mathcal{V}_T}{[a] \in \mathbb{P}_\perp} \\
 \overline{e \in \mathbb{P}_\perp} & \overline{e \in \mathbb{P}_{>0}} & \overline{e \in \mathbb{P}_\perp} & \overline{e \in \mathbb{P}_{\geq 0}} & \\
 \overline{e^* \in \mathbb{P}_0} & \overline{e^* \in \mathbb{P}_{>0}} & \overline{!e \in \mathbb{P}_0} & \overline{!e \in \mathbb{P}_\perp} &
 \end{array}$$

$$\frac{\star \in \{0, > 0, \perp\} \quad A \in \mathcal{V}_N \quad P_{\text{exp}}(A) \in \mathbb{P}_\star}{A \in \mathbb{P}_\star}$$

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

$$\frac{e_1 \in \mathbb{P}_0 \quad e_2 \in \mathbb{P}_0}{e_1; e_2 \in \mathbb{P}_0}$$

Definition (PEG analysis (1/2))

$$\begin{array}{ccccc}
 \overline{\epsilon \in \mathbb{P}_0} & \overline{[\cdot] \in \mathbb{P}_{>0}} & \overline{[\cdot] \in \mathbb{P}_\perp} & \frac{a \in \mathcal{V}_T}{[a] \in \mathbb{P}_{>0}} & \frac{a \in \mathcal{V}_T}{[a] \in \mathbb{P}_\perp} \\
 \frac{e \in \mathbb{P}_\perp}{e^* \in \mathbb{P}_0} & \frac{e \in \mathbb{P}_{>0}}{e^* \in \mathbb{P}_{>0}} & \frac{e \in \mathbb{P}_\perp}{!e \in \mathbb{P}_0} & \frac{e \in \mathbb{P}_{\geq 0}}{!e \in \mathbb{P}_\perp} &
 \end{array}$$

$$\frac{\star \in \{0, > 0, \perp\} \quad A \in \mathcal{V}_N \quad P_{\text{exp}}(A) \in \mathbb{P}_\star}{A \in \mathbb{P}_\star}$$

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

$$\frac{e_1 \in \mathbb{P}_0 \quad e_2 \in \mathbb{P}_0}{e_1; e_2 \in \mathbb{P}_0}$$

Definition (PEG analysis (2/2))

$$\frac{(e_1 \in \mathbb{P}_{>0} \wedge e_2 \in \mathbb{P}_{\geq 0}) \vee (e_1 \in \mathbb{P}_{\geq 0} \wedge e_2 \in \mathbb{P}_{>0})}{e_1; e_2 \in \mathbb{P}_{>0}}$$
$$\frac{e_1 \in \mathbb{P}_{\perp} \quad e_2 \in \mathbb{P}_{\perp}}{e_1/e_2 \in \mathbb{P}_{\perp}}$$
$$\frac{\star \in \{0, > 0\} \quad e_1 \in \mathbb{P}_{\star} \vee (e_1 \in \mathbb{P}_{\perp} \wedge e_2 \in \mathbb{P}_{\star})}{e_1/e_2 \in \mathbb{P}_{\star}}$$

Definition (Expression set of \mathcal{G})

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

\Rightarrow We compute those properties by iterating over $E(\mathcal{G})$ from \emptyset until reaching a fixpoint.

Definition (PEG analysis (2/2))

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

$$\frac{e_1 \in \mathbb{P}_{\perp} \quad e_2 \in \mathbb{P}_{\perp}}{e_1/e_2 \in \mathbb{P}_{\perp}}$$

$$\frac{\star \in \{0, > 0\} \quad e_1 \in \mathbb{P}_{\star} \vee (e_1 \in \mathbb{P}_{\perp} \wedge e_2 \in \mathbb{P}_{\star})}{e_1/e_2 \in \mathbb{P}_{\star}}$$

Definition (Expression set of \mathcal{G})

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

\Rightarrow We compute those properties by iterating over $E(\mathcal{G})$ from \emptyset until reaching a fixpoint.

Definition (PEG analysis (2/2))

$$\frac{(e_1 \in \mathbb{P}_{>0} \wedge e_2 \in \mathbb{P}_{\geq 0}) \vee (e_1 \in \mathbb{P}_{\geq 0} \wedge e_2 \in \mathbb{P}_{>0})}{e_1; e_2 \in \mathbb{P}_{>0}}$$
$$\frac{e_1 \in \mathbb{P}_{\perp} \quad e_2 \in \mathbb{P}_{\perp}}{e_1/e_2 \in \mathbb{P}_{\perp}}$$
$$\frac{\star \in \{0, > 0\} \quad e_1 \in \mathbb{P}_{\star} \vee (e_1 \in \mathbb{P}_{\perp} \wedge e_2 \in \mathbb{P}_{\star})}{e_1/e_2 \in \mathbb{P}_{\star}}$$

Definition (Expression set of \mathcal{G})

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

\Rightarrow We compute those properties by iterating over $E(\mathcal{G})$ from \emptyset until reaching a fixpoint.

Definition (PEG analysis (2/2))

$$\frac{(e_1 \in \mathbb{P}_{>0} \wedge e_2 \in \mathbb{P}_{\geq 0}) \vee (e_1 \in \mathbb{P}_{\geq 0} \wedge e_2 \in \mathbb{P}_{>0})}{e_1; e_2 \in \mathbb{P}_{>0}}$$
$$\frac{e_1 \in \mathbb{P}_{\perp} \quad e_2 \in \mathbb{P}_{\perp}}{e_1/e_2 \in \mathbb{P}_{\perp}}$$
$$\frac{\star \in \{0, > 0\} \quad e_1 \in \mathbb{P}_{\star} \vee (e_1 \in \mathbb{P}_{\perp} \wedge e_2 \in \mathbb{P}_{\star})}{e_1/e_2 \in \mathbb{P}_{\star}}$$

Definition (Expression set of \mathcal{G})

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

⇒ We compute those properties by iterating over $E(\mathcal{G})$ from \emptyset until reaching a fixpoint.

Definition (Checking WF)

$$\begin{array}{c} \frac{e_1 \in \text{WF} \quad e_2 \in \text{WF}}{e_1/e_2 \in \text{WF}} \qquad \frac{}{\epsilon \in \text{WF}} \\ \\ \frac{A \in \mathcal{V}_N \quad P_{\text{exp}}(A) \in \text{WF}}{A \in \text{WF}} \qquad \frac{}{[\cdot] \in \text{WF}} \\ \\ \frac{e_1 \in \text{WF} \quad e_1 \in \mathbb{P}_0 \Rightarrow e_2 \in \text{WF}}{e_1; e_2 \in \text{WF}} \qquad \frac{a \in \mathcal{V}_T}{[a] \in \text{WF}} \\ \\ \frac{e \in \text{WF}, \quad e \notin \mathbb{P}_0}{e^* \in \text{WF}} \qquad \frac{e \in \text{WF}}{!e \in \text{WF}} \end{array}$$

⇒ Again iteration from \emptyset until reaching a fixpoint

Well-formedness of (X)PEGs

Definition (WF)

We say that \mathcal{G} is well-formed if $E(\mathcal{G}) = \text{WF}$.

Theorem

If \mathcal{G} is well-formed then it is complete.

Well-formedness of (X)PEGs

Definition (WF)

We say that \mathcal{G} is well-formed if $E(\mathcal{G}) = \text{WF}$.

Theorem

If \mathcal{G} is well-formed then it is complete.

Example

Parameter $prods : Enumeration.$

Parameter $prods_type : prods \rightarrow Type.$

Inductive $PExp : Type \rightarrow Type :=$

| *Terminal* : $char \rightarrow PExp\ char$

| *NonTerminal* : $\forall p, PExp\ (prods_type\ p)$

| *Star* : $\forall A, PExp\ A \rightarrow PExp\ (list\ A)$

| *Choice* : $\forall A, PExp\ A \rightarrow PExp\ A \rightarrow PExp\ A$

| ...

\Rightarrow Every such PEG is well-defined and well-typed.

Example

Parameter $prods : Enumeration.$

Parameter $prods_type : prods \rightarrow Type.$

Inductive $PExp : Type \rightarrow Type :=$

| *Terminal* : $char \rightarrow PExp\ char$

| *NonTerminal* : $\forall p, PExp\ (prods_type\ p)$

| *Star* : $\forall A, PExp\ A \rightarrow PExp\ (list\ A)$

| *Choice* : $\forall A, PExp\ A \rightarrow PExp\ A \rightarrow PExp\ A$

| ...

\Rightarrow Every such PEG is well-defined and well-typed.

Example (Coq's grammar of our leading example)

Program Definition *production p* :=

match *p* **return** *PExp (prod_type p)* **with**

<i>ws</i> ⇒ (" " / "\\t") [*]	[#]	
<i>number</i> ⇒ ["0" -- "9"] [+]	[→]	<i>digListToRat</i>
<i>term</i> ⇒ <i>ws</i> ; <i>number</i> ; <i>ws</i>	[→]	(λ <i>v</i> ⇒ <i>A2_3 v</i>)
/ <i>ws</i> ; "("; <i>expr</i> ; ")"	[→]	(λ <i>v</i> ⇒ <i>A3_5 v</i>)
<i>factor</i> ⇒ <i>term</i> ; "*" ; <i>factor</i>	[→]	(λ <i>v</i> ⇒ <i>A1_3 v</i> * <i>A3_3 v</i>)
/ <i>term</i>		
<i>expr</i> ⇒ <i>factor</i> ; "+" ; <i>expr</i>	[→]	(λ <i>v</i> ⇒ <i>A1_3 v</i> + <i>A3_3 v</i>)
/ <i>factor</i>		

end.

⇒ Thanks to notations and coercions this is not too different from what we had before...

Example (Coq's grammar of our leading example)

Program Definition *production p* :=

match *p* **return** *PExp* (*prod_type p*) **with**

<i>ws</i> ⇒ (" " / "\\t") [*]	[#]	
<i>number</i> ⇒ ["0" -- "9"] [+]	[→]	<i>digListToRat</i>
<i>term</i> ⇒ <i>ws</i> ; <i>number</i> ; <i>ws</i>	[→]	($\lambda v \Rightarrow A2_3 v$)
/ <i>ws</i> ; "("; <i>expr</i> ; ")"	[→]	($\lambda v \Rightarrow A3_5 v$)
<i>factor</i> ⇒ <i>term</i> ; "*" ; <i>factor</i>	[→]	($\lambda v \Rightarrow A1_3 v * A3_3 v$)
/ <i>term</i>		
<i>expr</i> ⇒ <i>factor</i> ; "+" ; <i>expr</i>	[→]	($\lambda v \Rightarrow A1_3 v + A3_3 v$)
/ <i>factor</i>		

end.

⇒ Thanks to notations and coercions this is not too different from what we had before...

Example

```
Program Fixpoint wf_compute (wf : PES.t | wf_prop wf)  
  {measure (wf_measure wf) } :  
  {wf : PES.t | wf_prop wf } :=  
  let wf' := wf_derive wf in  
  match PES.equal wf wf' with  
  | true  $\Rightarrow$  wf  
  | false  $\Rightarrow$  wf_compute wf'  
  end.
```

Theorem

- $wf \subseteq E(G) \implies wf_derive\ wf \subseteq E(G)$
- $wf \subseteq wf_derive\ wf$

Example

```
Program Fixpoint wf_compute (wf : PES.t | wf_prop wf)  
  {measure (wf_measure wf) } :  
  {wf : PES.t | wf_prop wf } :=  
  let wf' := wf_derive wf in  
  match PES.equal wf wf' with  
  | true  $\Rightarrow$  wf  
  | false  $\Rightarrow$  wf_compute wf'  
  end.
```

Theorem

- $wf \subseteq E(\mathcal{G}) \implies wf_derive\ wf \subseteq E(\mathcal{G})$
- $wf \subseteq wf_derive\ wf$.

Example

Program Fixpoint *parse*

$(T : \text{Type}) (e : \text{PExp } T \mid \text{is_grammar_exp } e) (s : \text{string})$
 $: \{r : \text{ParsingResult } T \mid \exists n, [e, s] \Rightarrow [n, r]\} :=$

match *e* **with**

| ...
end.

Example

Program Fixpoint *parse*

```
(T : Type) (e : PExp T | is_grammar_exp e) (s : string)  
: {r : ParsingResult T | ∃ n, [e, s] ⇒ [n, r]} :=
```

match e with

```
| Terminal c ⇒
```

match s with

```
| nil ⇒ Fail
```

```
| x :: xs ⇒
```

match CharAscii.eq_dec c x with

```
| left _ ⇒ Ok xs c
```

```
| right _ ⇒ Fail
```

```
end
```

```
end
```

```
| ...
```

```
end.
```

Example

Program Fixpoint *parse*

$(T : Type) (e : PExp\ T \mid is_grammar_exp\ e) (s : string)$
 $: \{r : ParsingResult\ T \mid \exists n, [e, s] \Rightarrow [n, r]\} :=$

match *e* **with**

| *NonTerminal* *p* \Rightarrow
 parse (*production* *p*) *s*

| ...

end.

Example

Program Fixpoint *parse*

$(T : \text{Type}) (e : \text{PExp } T \mid \text{is_grammar_exp } e) (s : \text{string})$
 $: \{r : \text{ParsingResult } T \mid \exists n, [e, s] \Rightarrow [n, r]\} :=$

match *e* **with**

| *Choice* _ *e1* *e2* \Rightarrow

match *parse* *e1* *s* **with**

 | *PR_ok* *s'* *v* \Rightarrow *Ok* *s'* *v*

 | *PR_fail* \Rightarrow *parse* *e2* *s*

end

| ...

end.

Example

Program Fixpoint *parse*

(*T* : Type) (*e* : PExp *T* | *is_grammar_exp* *e*) (*s* : string)
: {*r* : ParsingResult *T* | $\exists n, [e, s] \Rightarrow [n, r]$ } :=

match *e* **with**

| *Star* _ *e* \Rightarrow

match *parse* *e* *s* **with**

| *PR_fail* \Rightarrow *Ok* *s* []

| *PR_ok* *s'* *v* \Rightarrow

match *parse* (*e* [*]) *s'* **with**

| *PR_fail* \Rightarrow *Fail*

| *PR_ok* *s''* *v'* \Rightarrow *Ok* *s''* (*v* :: *v'*)

end

end

| ...

end.

Example

Program Fixpoint *parse*

$(T : \text{Type}) (e : \text{PExp } T \mid \text{is_grammar_exp } e) (s : \text{string})$
 $: \{r : \text{ParsingResult } T \mid \exists n, [e, s] \Rightarrow [n, r]\} :=$

match *e* **with**

| ...

end.

How about termination?

Example

$$(e_1, s_1) \succ (e_2, s_2) \iff \exists n_1, r_1, n_2, r_2 (e_1, s_1) \rightsquigarrow^{n_1} r_1 \wedge (e_2, s_2) \rightsquigarrow^{n_2} r_2 \wedge n_1 > n_2$$

Program Fixpoint *parse*

$(T : Type) (e : PExp T \mid is_grammar_exp e) (s : string)$

$\{measure (e, s) (\succ)\}$

$: \{r : ParsingResult T \mid \exists n, [e, s] \Rightarrow [n, r]\} :=$

match *e* **with**

| ...

end.

How about termination?

Example

Program Fixpoint *parse*

$(T : \text{Type}) (e : \text{PExp } T \mid \text{is_grammar_exp } e) (s : \text{string})$

$\{\text{measure } (e, s) (\succ)\}$

$: \{r : \text{ParsingResult } T \mid \exists n, [e, s] \Rightarrow [n, r]\} :=$

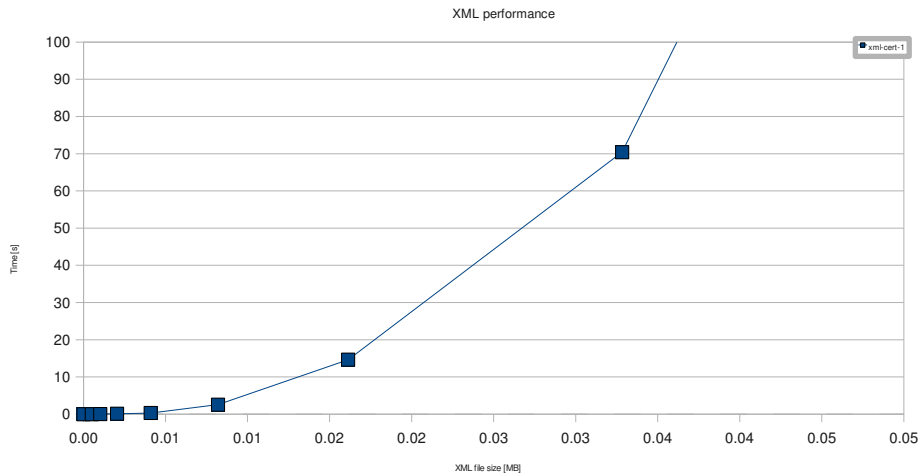
match *e* **with**

| ...

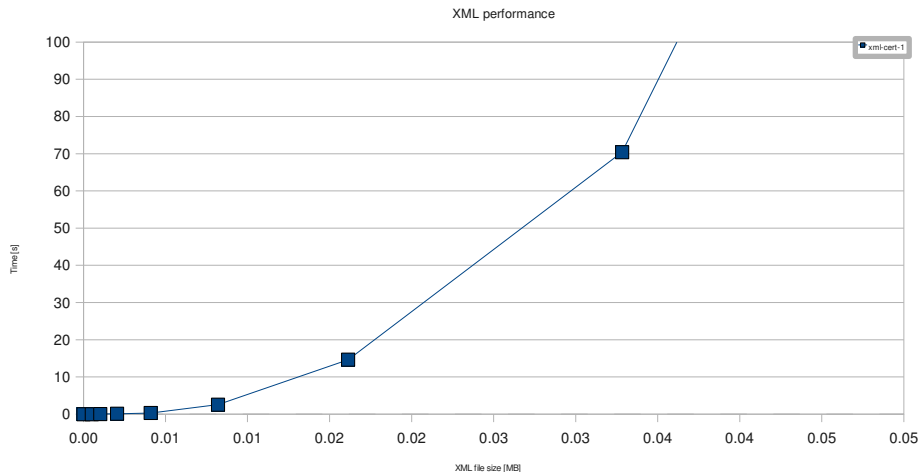
end.

+ 43 TCCs.

Evolution of TRX's performance

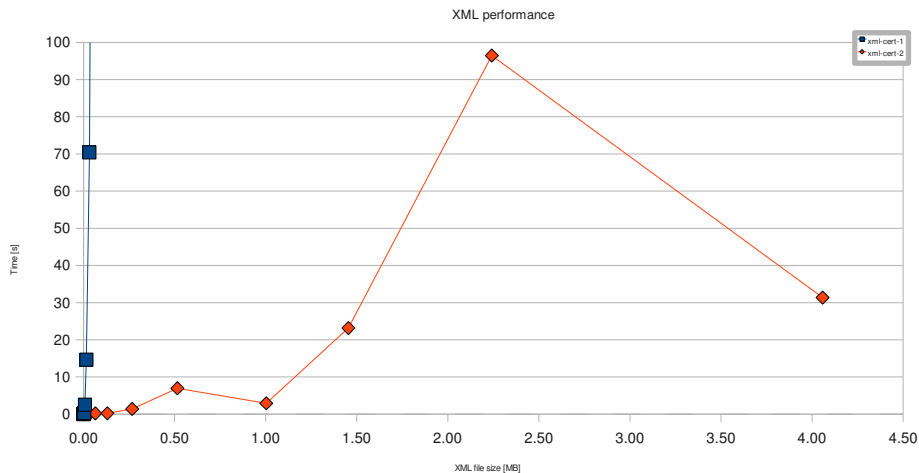


Evolution of TRX's performance

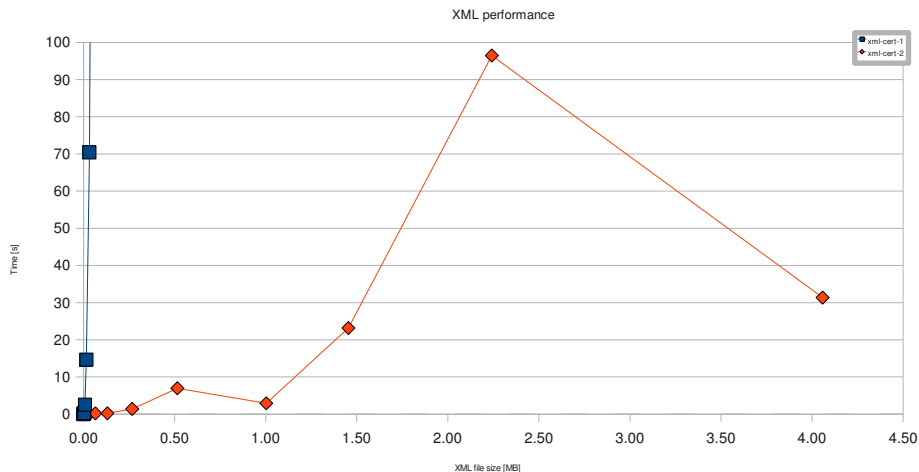


⇒ rev uses append at every step and hence is quadratic

Evolution of TRX's performance

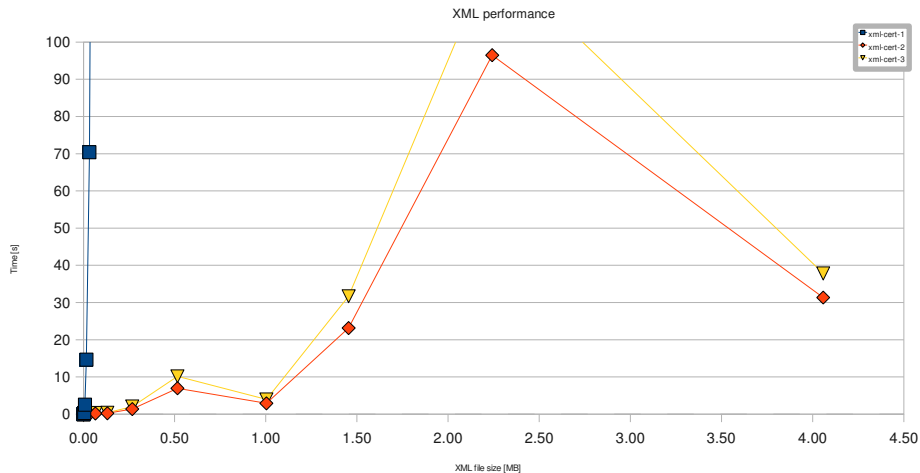


Evolution of TRX's performance

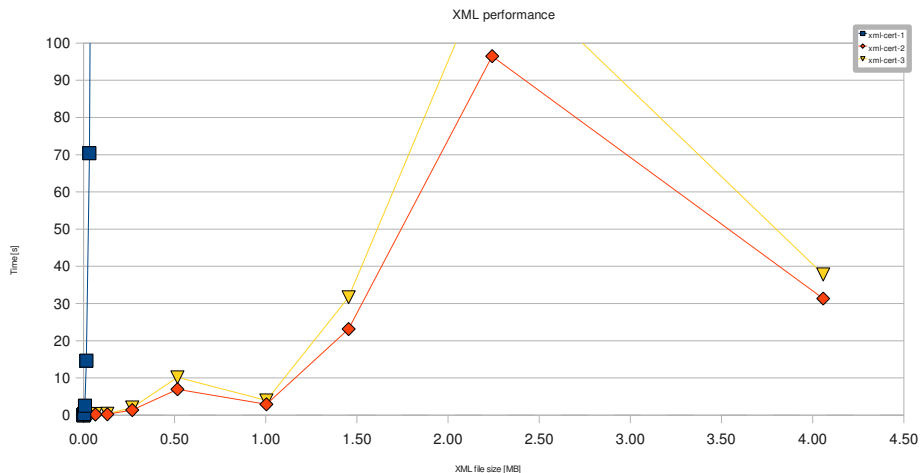


⇒ We better implement $[a-z]$ as a primitive

Evolution of TRX's performance

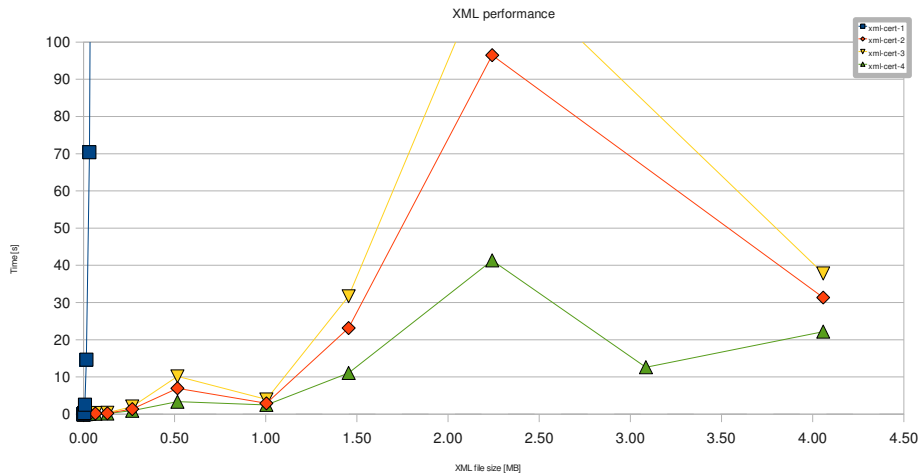


Evolution of TRX's performance

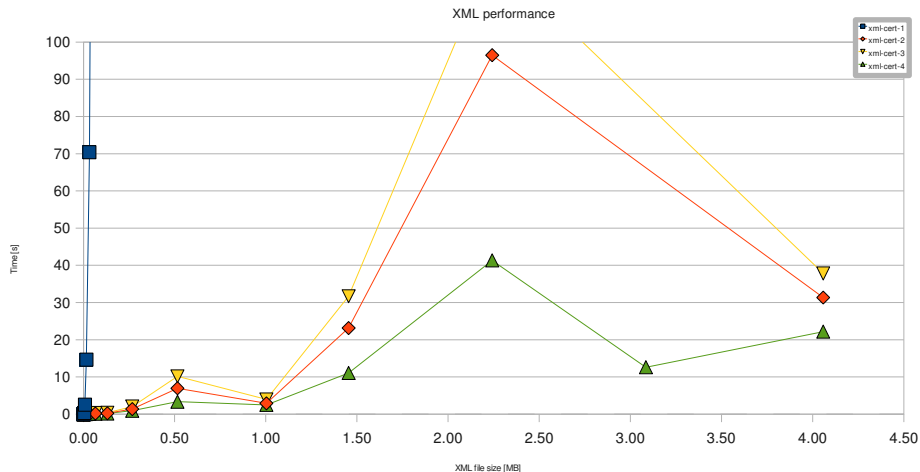


⇒ ... and we better use binary \mathbb{N} to compare characters

Evolution of TRX's performance



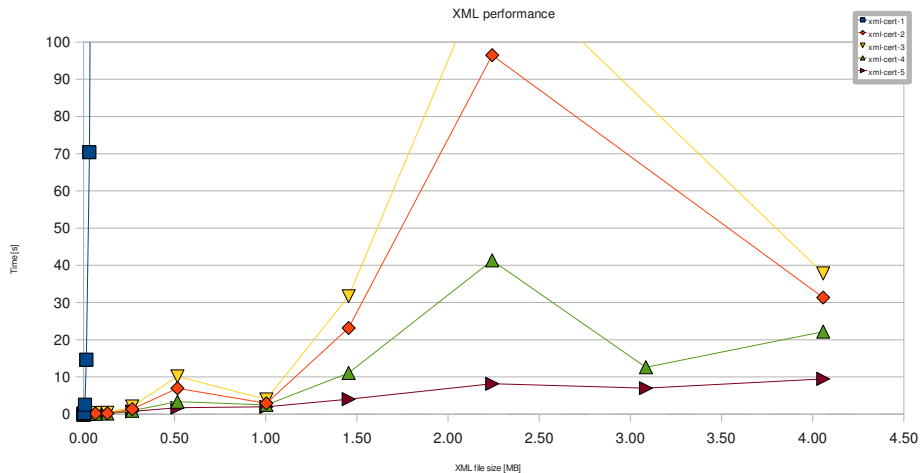
Evolution of TRX's performance



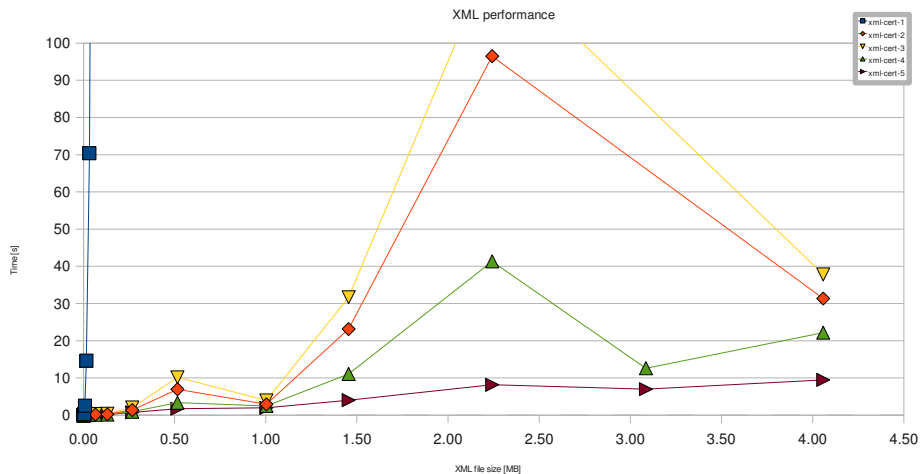
⇒ 85% of the time the parser is busy with GC! Let us tweak it

MLstate

Evolution of TRX's performance

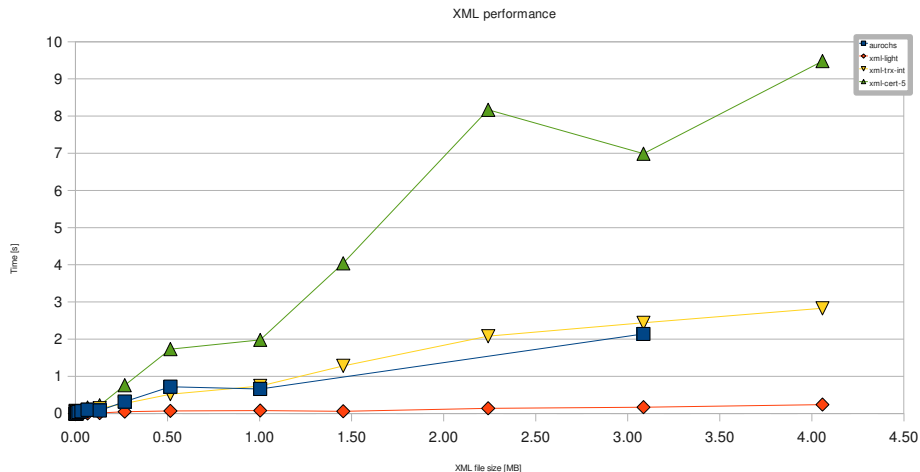


Evolution of TRX's performance



⇒ Can we do better than that?

TRX: performance comparison with other tools



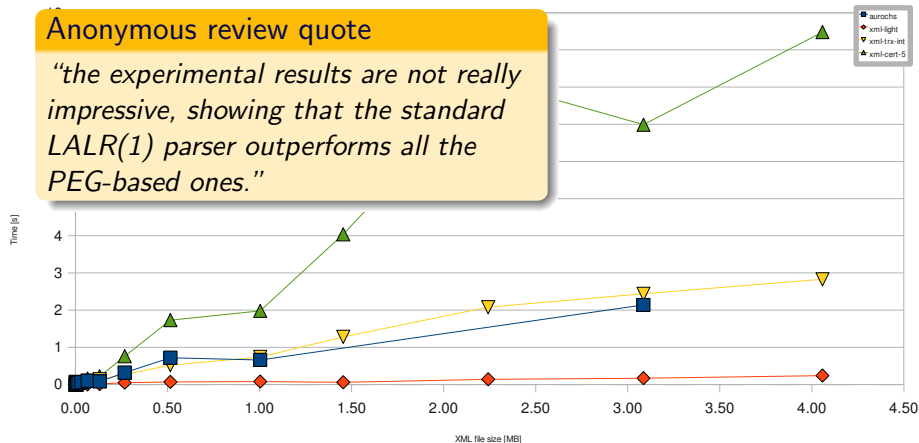
- 7x slower than Aurochs (but more robust?).
- 32x slower than xml-light.

TRX: performance comparison with other tools

XML performance

Anonymous review quote

"the experimental results are not really impressive, showing that the standard LALR(1) parser outperforms all the PEG-based ones."



- 7x slower than Aurochs (but more robust?).
- 32x slower than xml-light.

TRX: performance comparison with other tools

Anonymous review quote

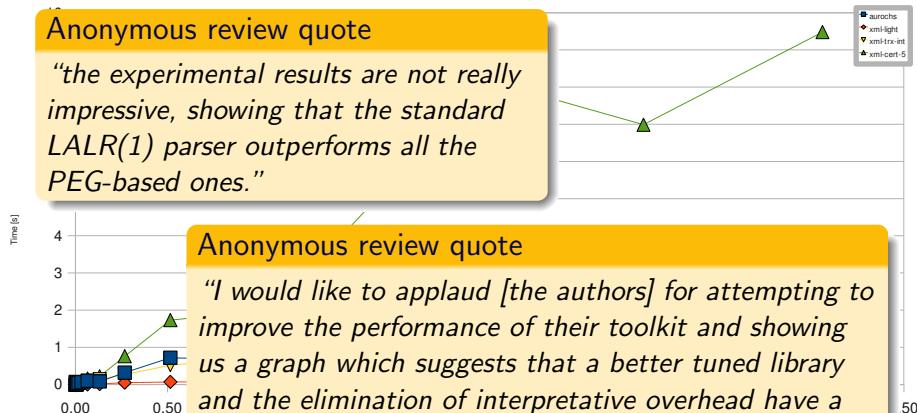
"the experimental results are not really impressive, showing that the standard LALR(1) parser outperforms all the PEG-based ones."

Anonymous review quote

"I would like to applaud [the authors] for attempting to improve the performance of their toolkit and showing us a graph which suggests that a better tuned library and the elimination of interpretative overhead have a good chance of delivering competitive results. The mere fact that it is even feasible to consider 'Performance comparison' is progress indeed."

- 7x slower
- 32x slower

XML performance



TRX: performance comparison with other tools

Anonymous review quote

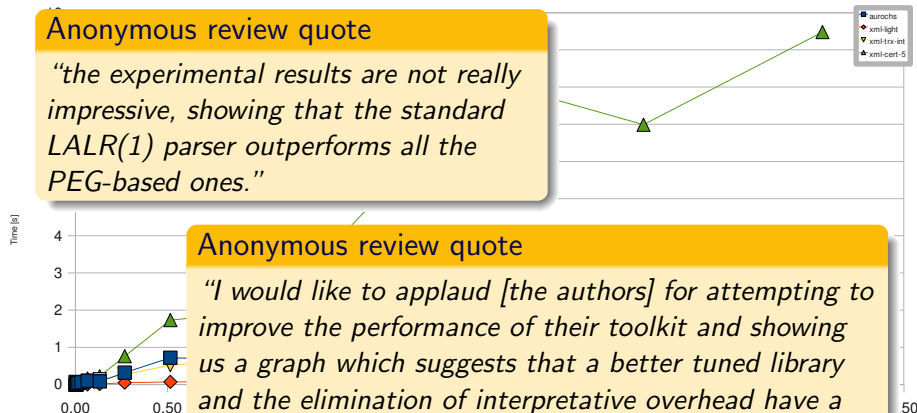
"the experimental results are not really impressive, showing that the standard LALR(1) parser outperforms all the PEG-based ones."

Anonymous review quote

*"I would like to applaud [the authors] for attempting to improve the performance of their toolkit and showing us a graph which suggests that a better tuned library and the elimination of interpretative overhead have a good chance of delivering competitive results. **The mere fact that it is even feasible to consider 'Performance comparison' is progress indeed.**"*

- 7x slower
- 32x slower

XML performance



Work on **formally verified** parsing:

- **SLR parser in HOL**



Aditi Barthwal and Michael Norrish:
Verified, Executable Parsing.
ESOP '09

Related work

Work on **formally verified** parsing:

- SLR parser in HOL



Aditi Barthwal and Michael Norrish:
Verified, Executable Parsing.
ESOP '09

- **library of parser combinators in Agda**



Nils Anders Danielsson and Ulf Norell
Structurally Recursive Descent Parsing.
Draft, '08

Work on **formally verified** parsing:

- SLR parser in HOL (**termination not addressed**)



Aditi Barthwal and Michael Norrish:
Verified, Executable Parsing.
ESOP '09

- library of parser combinators in Agda (**type-indices ruling out left-recursion**)



Nils Anders Danielsson and Ulf Norell
Structurally Recursive Descent Parsing.
Draft, '08

- PEG parser in Ynot/Coq (**termination not addressed**)



Ryan Wisnesky and Gregory Malecha and Greg Morrisett
Certified Web Services in Ynot.
WWW '09

Conclusions

- We presented an interpreter for PEGs developed in Coq,
- along with the proofs of semantic preservation and termination ensuring total correctness of parsing.
- Using Coq's extraction capabilities this enables us to obtain formally correct parsers.

Conclusions

- We presented an interpreter for PEGs developed in Coq,
- along with the proofs of semantic preservation and termination ensuring total correctness of parsing.
- Using Coq's extraction capabilities this enables us to obtain formally correct parsers.

Conclusions

- We presented an interpreter for PEGs developed in Coq,
- along with the proofs of semantic preservation and termination ensuring total correctness of parsing.
- Using Coq's extraction capabilities this enables us to obtain formally correct parsers.

Conclusions

- We presented an interpreter for PEGs developed in Coq,
- along with the proofs of semantic preservation and termination ensuring total correctness of parsing.
- Using Coq's extraction capabilities this enables us to obtain formally correct parsers.

If you want to learn more about OPA I will be happy to discuss it over lunch!



<http://mlstate.com>

Thank you for your attention.

MLstate