supported by: THE ROYAL SOCIETY

A dependently-typed regex parser.

Ohad Kammar and Katarzyna Marek

University of Edinburgh

Regular expressions

```
r "(([01][0-9])|([2][0-4])):([0-5][0-9])"
```

	Validation : Bool	Parsing : Maybe (Int, Int)
”12:03”	True	Just (12, 3)
”12:63”	False	Nothing

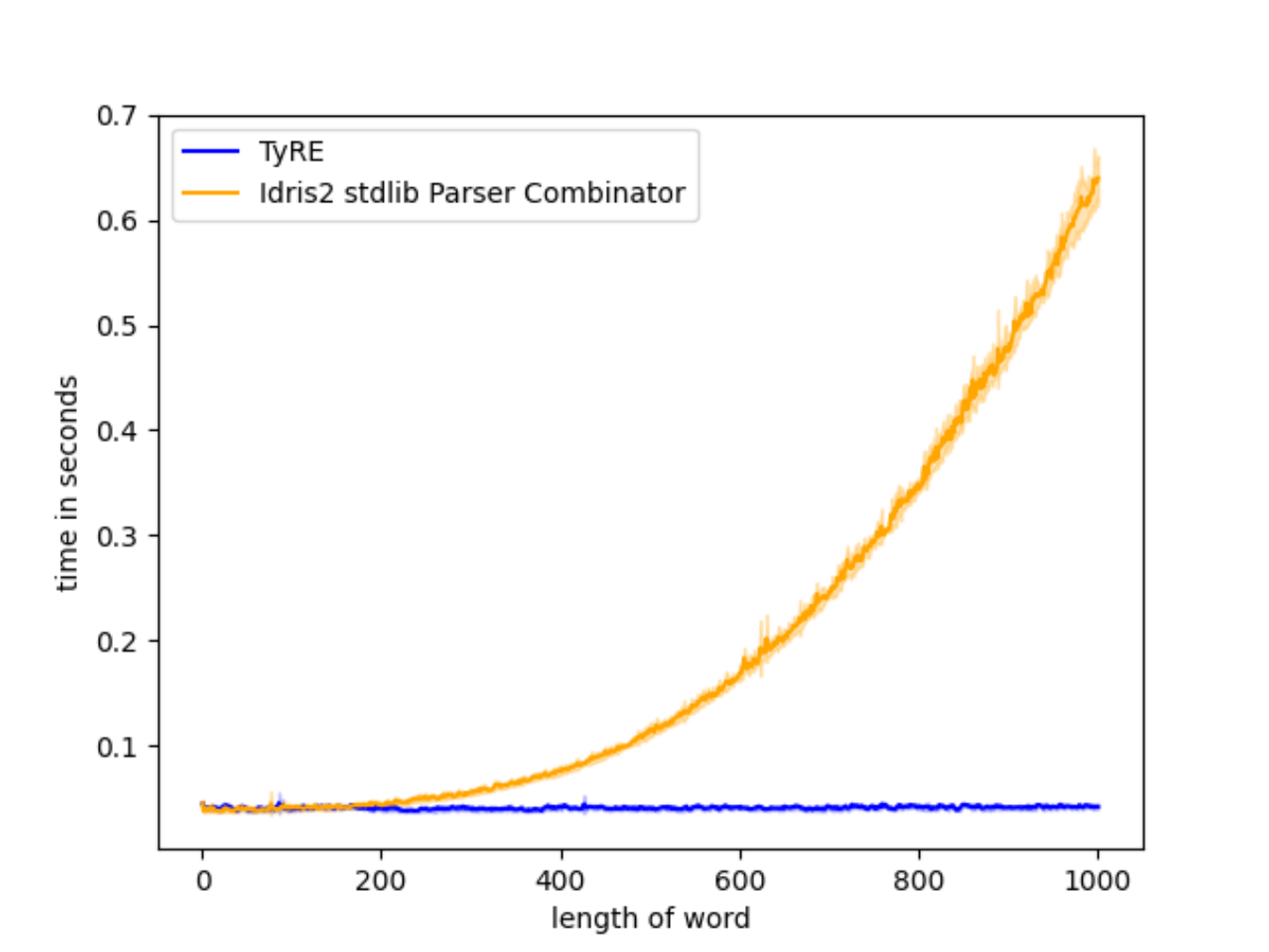
TyRE : Typed RegExes

G. Radanne. [PEPM'19]
Typed parsing and unparsing for untyped regular expression engines.

Idris-TyRE vs. Radanne's tyre:

- tyre layer - `digit <*> any : TyRE (Int, Char)`
- unparsing - `Conv snd (r "[AB][0-9]")`
- custom parser (there is no regex parser in Idris)
- type safety throughout all layers

Why Regexes?



regex: $((a*c)|a)*b$; string: $a^{n-1}b$

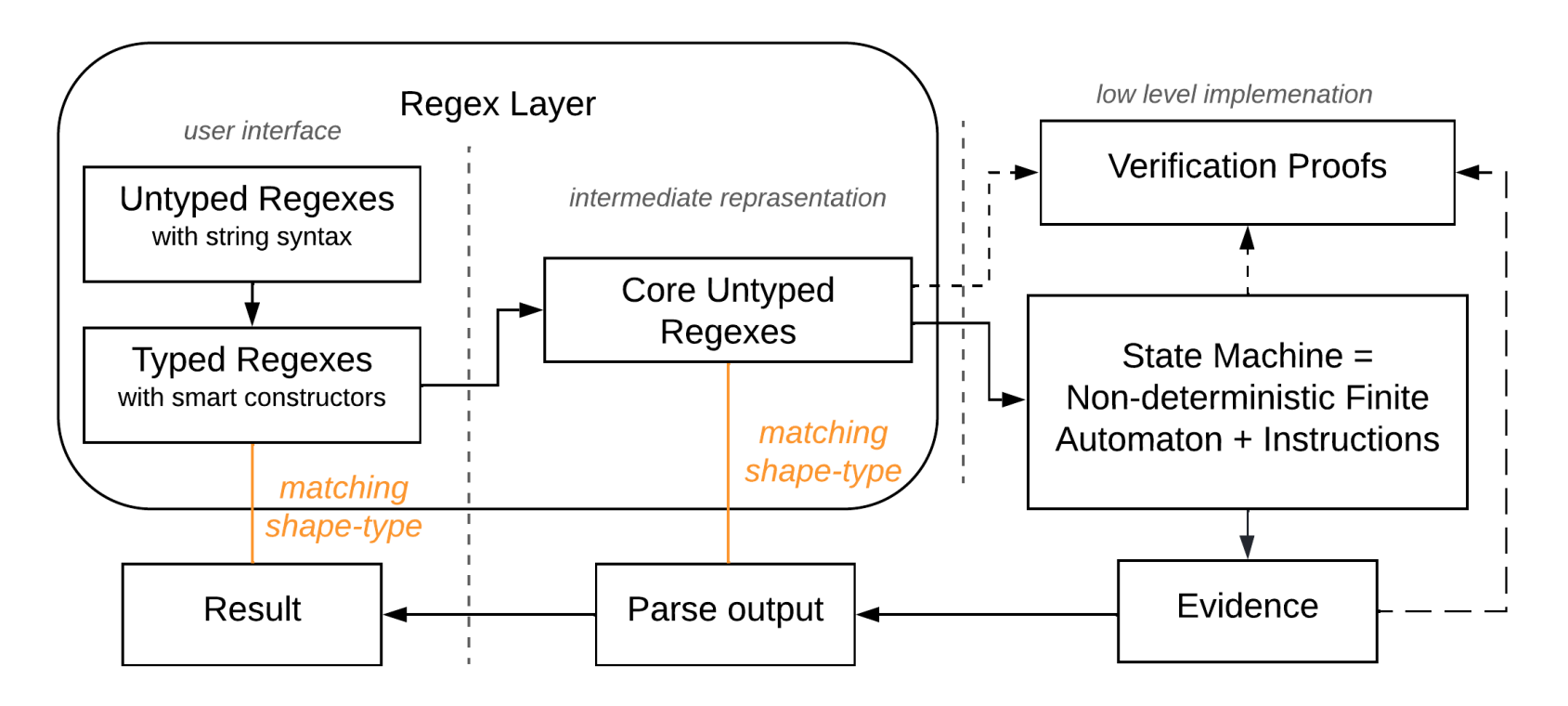
Shape of a regex

```
r "[AB][0-9]" : TyRE (Char, Char)
                     ^^^^^^^^^^^
                       shape
```

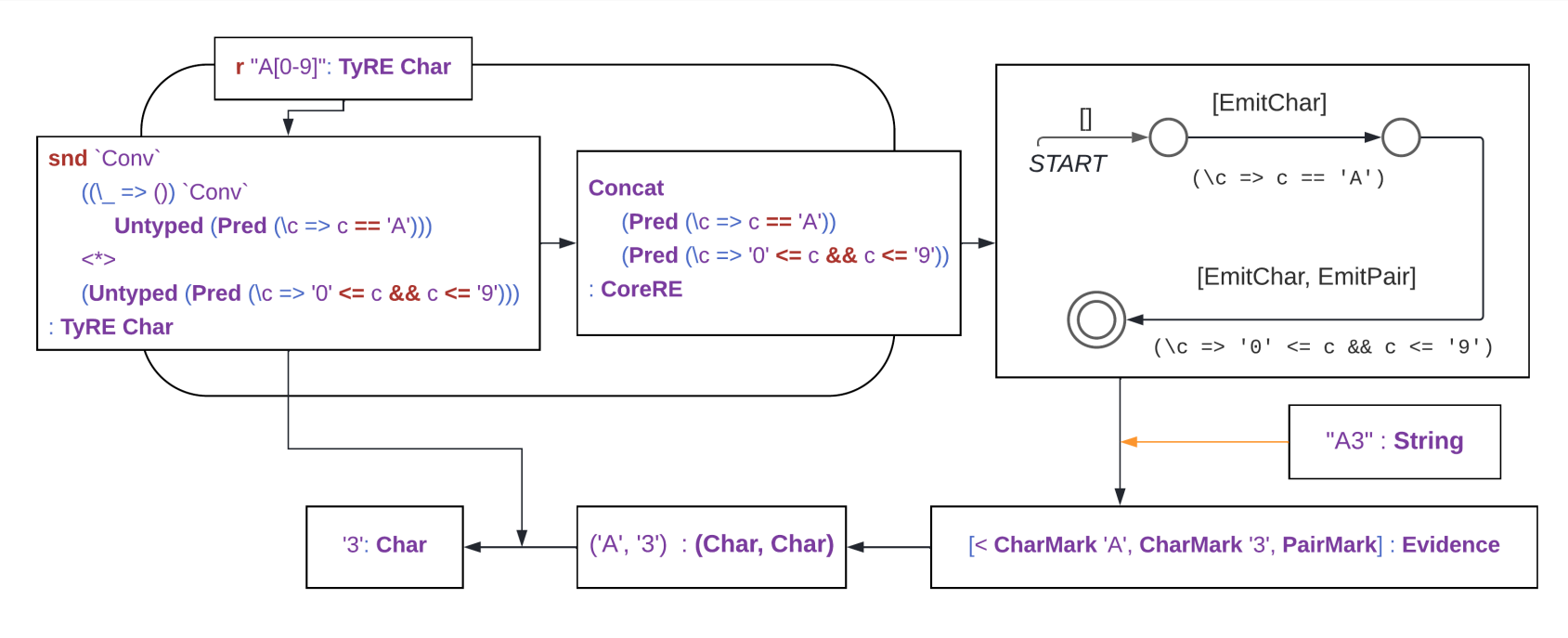
Literal	RE	Shape
`a`	`Exactly a`	`Unit`
`[abc]`	`OneOf ['a','b','c']`	`Char`
`[a-c]`	`'a' `To` 'c'`	`Char`
`.`	`Any`	`Char`
`AB`	`A `Concat` B`	`(Shape A, Shape B)`
`A|B`	`A `Alt` B`	`Either (Shape A) (Shape B)`
`A?`	`Maybe A`	`Maybe (Shape A)`
`A*`	`Rep0 A`	`List (Shape A)`
`A+`	`Rep1 A`	`List (Shape A)`
`` `A` ``	`Group A`	`String`

Demo

Architecture

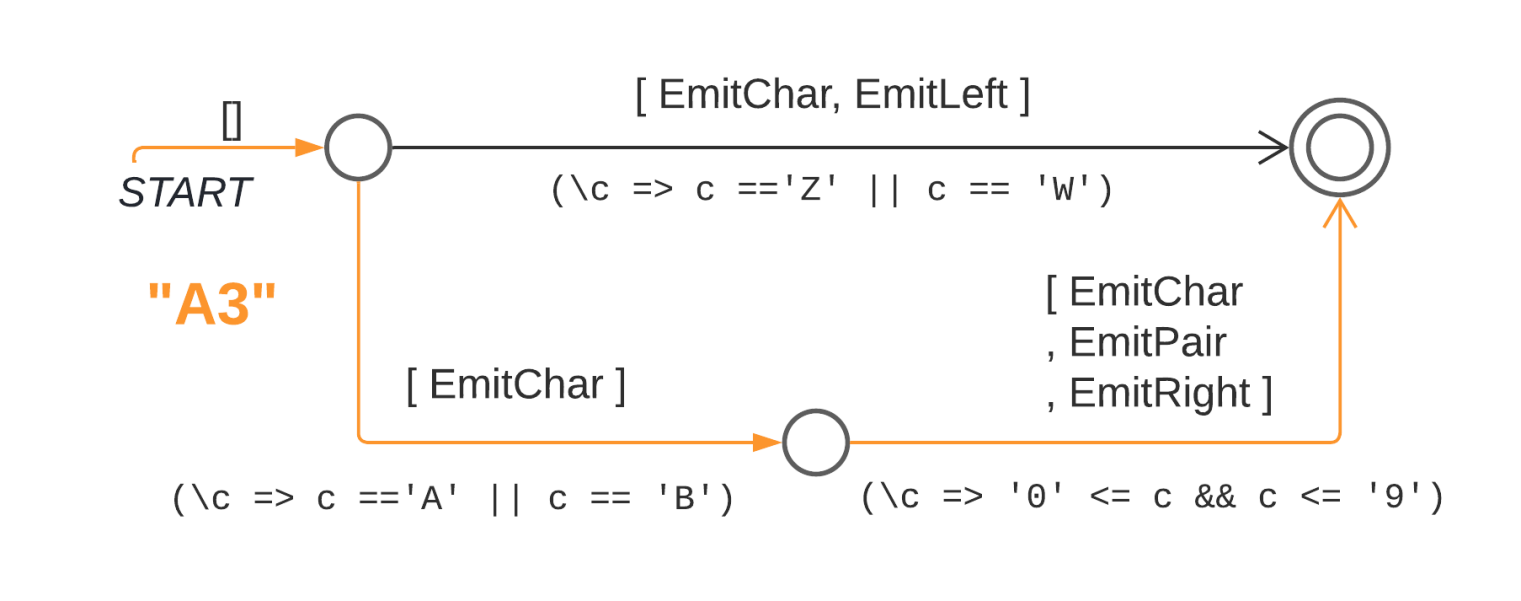


Example


r "A[0-9]": TyRE Char
snd `Conv`
((_ => ()) `Conv`
Untyped (Pred (\c => c == 'A')))
<*>
(Untyped (Pred (\c => '0' <= c && c <= '9')))
: TyRE Char
Concat
(Pred (\c => c == 'A'))
(Pred (\c => '0' <= c && c <= '9'))
: CoreRE
[]
START
[EmitChar]
(\c => c == 'A')
[EmitChar, EmitPair]
(\c => '0' <= c && c <= '9')
"A3" : String
'3': Char
('A', '3') : (Char, Char)
[< CharMark 'A', CharMark '3', PairMark] : Evidence

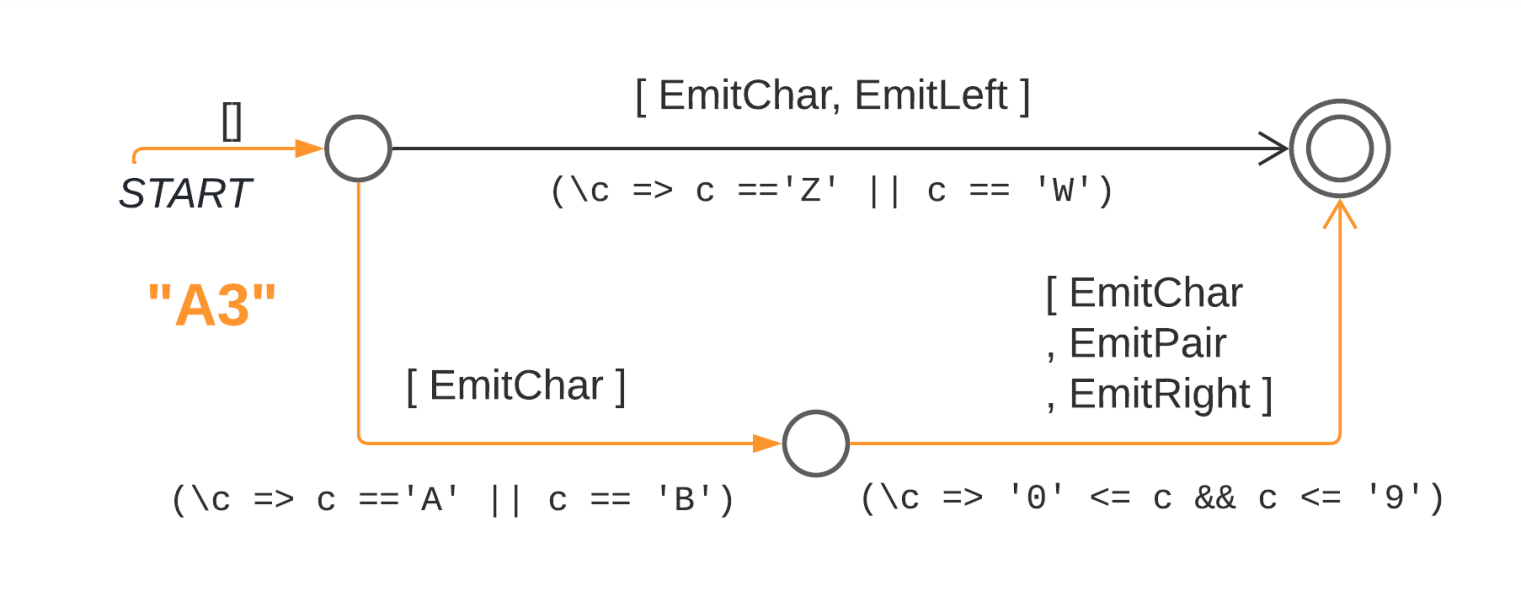
Verification proofs I - Creating an accepting path

Given a state machine SM, that accepts a string S,
then there exists an accepting path through SM,
s.t. the evidence collected by following this path is equal to evidence collected when running the SM on S.



Verification proofs II - Evidence shape

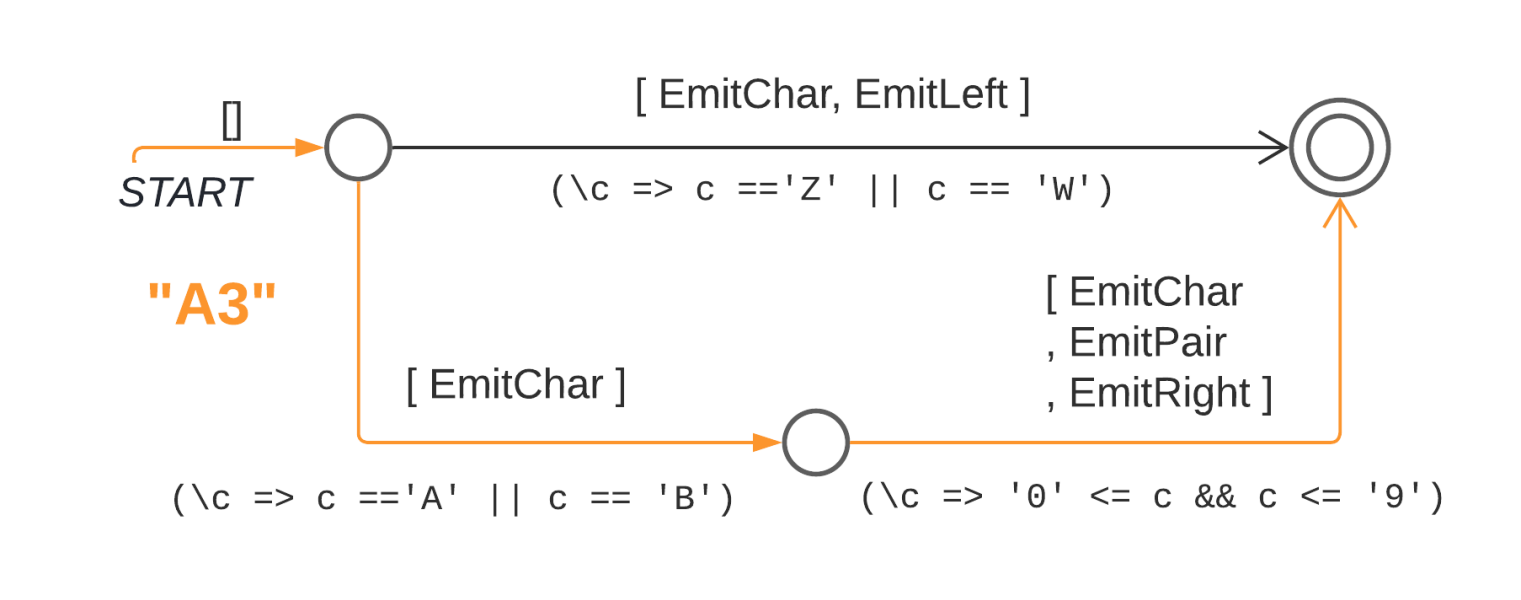
Given a state machine SM, that was constructed for a regex RE
and an accepting path A through SM,
then the evidence collected by following A through SM encodes the shape of RE.




```
r "[ZW]|([AB][0-9])"
  : TyRE (Either Char (Char, Char))
```

Verification proofs III

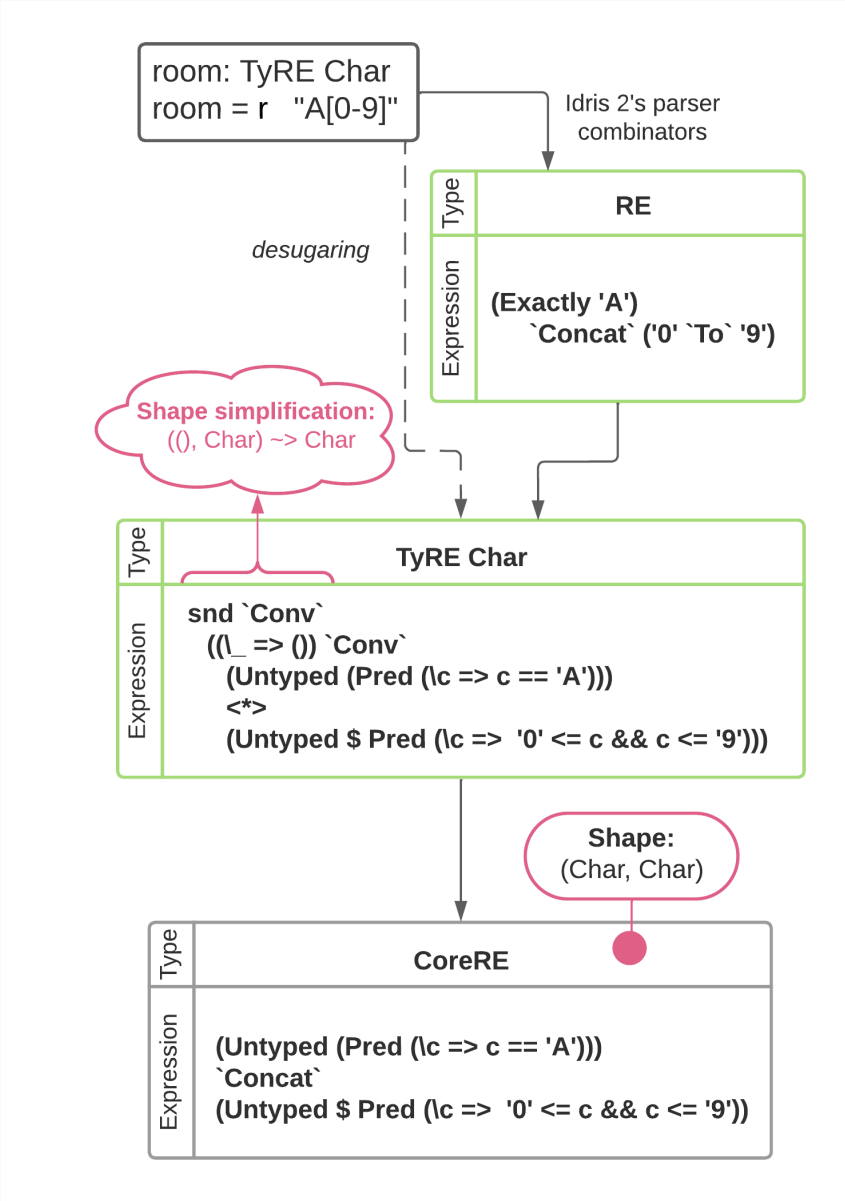
```
r "[ZW]|([AB][0-9])"
: TyRE (Either Char (Char, Char))
```



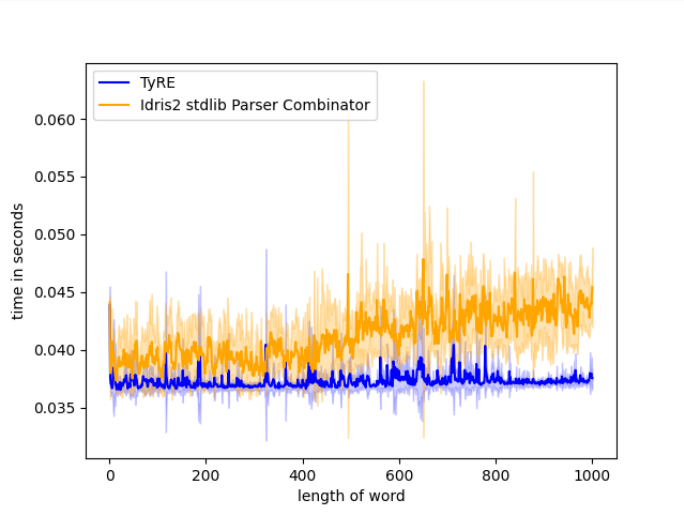
The proofs are erased at runtime.

String syntax

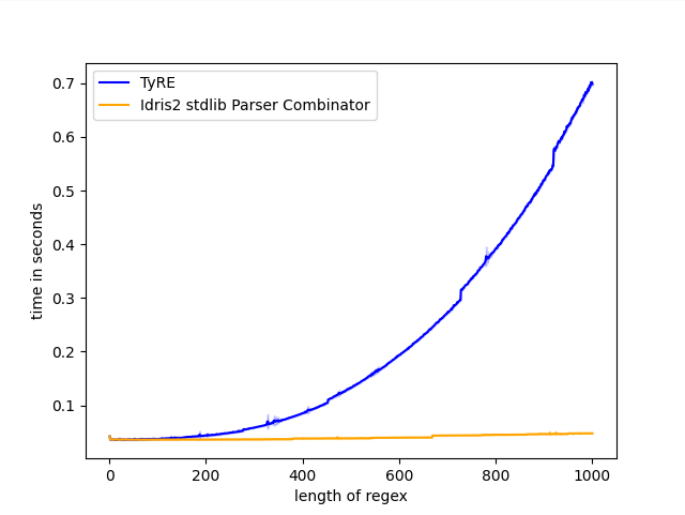
1. We parse strings into *RE* during compile-time.
   - We use a copy of Idris 2's parser combinators library.
2. More type-level calculation to get the *TyRE*.
   - One to one translation.
   - Shape simplification.



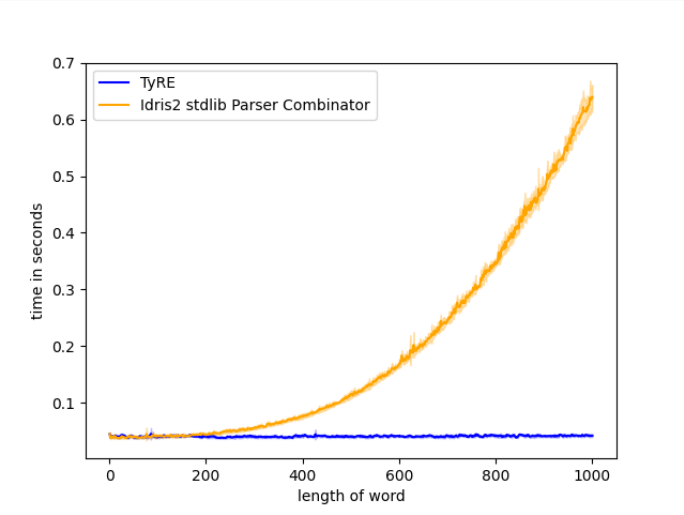
Performance



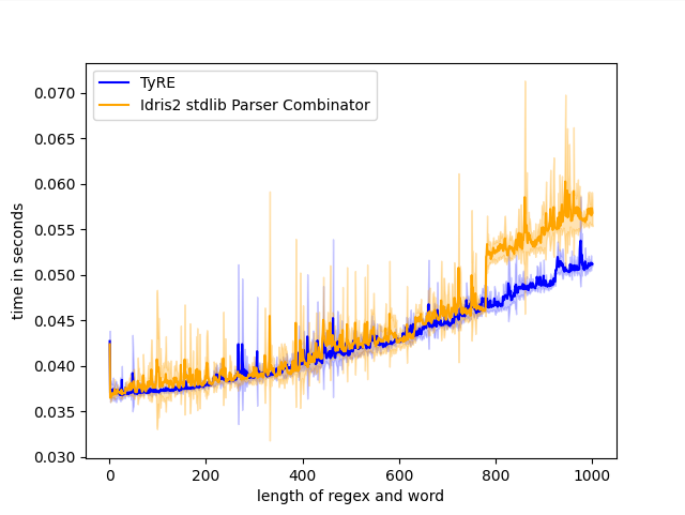
regex: $a*$; string: a^n



regex: $a(\,|\,a\,)^{n-1}$; string: a



regex: $((a*c)|a)*b$; string: $a^{n-1}b$



regex: a^n; string: a^n

Limitations

- Library supports only regular expressions (in formal language theory meaning).
- Minimal performance engineering
  - no support for SM pre-compilation,
  - no automata minimisation.
- We prove safety, not functional correctness.
- Missing features:
  - regex string literals syntax, e.g. ”a{0, 3}” is not supported,
  - common regex usages, e.g. find all and replace.
- Compile time performance:
  - parsing string literals,
  - running TyRE at compile-time.

Possible future work

- Integrate TyRE with Idris 2's stdlib parser combinators library.
  - This would allow bootstrapping and using TyRE to parse regex string literals.
  - Challenge : compile-time parsing.
- Adding unparsing.
  - Possible use case:
    Collie: command-line interface generator [Allais and Kammar]
- Extending to μ-regular expressions (context-free expressions) [Krishnaswami and Yallop, 2019].

Questions?

References

Krishnaswami, N. R. and Yallop, J. (2019).
A typed, algebraic approach to parsing.
In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 379–393, New York, NY, USA. Association for Computing Machinery.

Radanne, G. (2019).
Typed parsing and unparsing for untyped regular expression engines.
In *PEPM 2019 - Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Co-located with POPL 2019*.