Typed Parsing and Unparsing for Untyped Regular Expression Engines

Gabriel Radanne
University of Freiburg
Germany
radanne@informatik.uni-freiburg.de

Abstract

Regular expressions are used for a wide variety of purposes from web-page input validation to log file crawling. Very often, they are used not only to match strings, but also to extract data from them. Unfortunately, most regular expression engines only return a list of the substrings captured by the regular expression. The data has to be extracted from the matched substrings to be validated and transformed manually into a more structured format.

For richer classes of grammars like CFGs, such issues can be solved using type-indexed combinators. Most combinator libraries provide a monadic API to track the type returned by the parser through easy-to-use combinators. This allows users to transform the input into a custom data-structure and go through complex validations as they describe their grammar.

In this paper, we present the Tyre library which provides type-indexed combinators for regular languages. Our combinators provide type-safe extraction while delegating the task of substring matching to a preexisting regular expression engine. To do this, we use a two layer approach where the typed layer sits on top of an untyped layer. This technique is also amenable to several extensions, such as routing, unparsing and static generation of the extraction code. We also provide a syntax extension, which recovers the familiar and compact syntax of regular expressions. We implemented this technique in a very concise manner and evaluated its usefulness on two practical examples.

CCS Concepts • **Theory of computation → Regular languages**; *Parsing*; • **Software and its engineering → Functional languages**;

Keywords Functional programming, Static typing, Regular expressions, unparsing, OCaml

PEPM '19, January 14–15, 2019, Cascais, Portugal

ACM Reference Format:
Gabriel Radanne. 2019. Typed Parsing and Unparsing for Untyped Regular Expression Engines. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3294032.3294082

1 Introduction

Regular expressions are used in a wide variety of contexts, from checking inputs of web forms to extracting data contained in text files. In many cases, the goal of the regular expression is not only to match a given text, but also to extract information from it through capture groups. Capture groups are annotations on parts of the regular expression that indicates that the substring matched by these subexpressions should be returned by the matching engine. For instance, `"([a-zA-Z]*):([0-9]*)"` matches strings of the form `"myid:42"` and captures the part before and after the colon.

Extraction using capture groups, however, only returns strings. In the regex above, the semantics we assign to the second group is clearly the one of a number, but this information is not provided to the regular expression engine, and it is up to the programmer to parse the stream of digits into a proper integer. This problem only grows more complex with the complexity of the data to extract. For instance, if we were to consider parsing URIs with regular expressions, we would need to extract a complex structure with multiple fields, lists and alternatives. Even static typing does not help here, since all the captured fields are strings! This problem, often described as input validation, can not only be the source of bugs but also serious security vulnerabilities.

One approach that might help is to rely on "full regular expression parsing" which consists in returning the parsetree of a string matched by a given regular expression. This is generally coupled with an interpretation of regular expressions as types to determine the shape of the resulting parsetree: a concatenation of regular expression is interpreted as a product type, an alternative as a sum type and the Kleene star as a list, thus forming a small algebra of types that reflects the compositional properties of regular expressions.

Full regular expression parsing is not completely sufficient: indeed we want both the ability to use structured types, but also to transform and verify the input. We also want to ignore the semantically-irrelevant parts of the string. A final problem is that full regular expression parsing is a very rare feature among regex engines. Mature implementations such as Re2 [Cox 2010], PCRE [Hazel 2015], Javascript regular expressions [ECMAScript 2018] or Hyperscan [Intel 2018] do not provide it. While writing a new regular expression engine might seem tempting, writing a featureful, efficient and portable engine that supports features such as online determinization, lookarounds operators and both POSIX and greedy semantics is a significant undertaking, as demonstrated by the projects listed above.

In this paper, we present a technique to provide type-safe extraction based on the typed interpretation of regular expressions while delegating the actual matching to an external regular expression engine that only supports substring extraction. This provides users with the convenience and type-safety of parsing combinators, while using a mature regular expression engine. Our technique relies on two-layer regular expressions where the upper layer allows to composes and transforms data in a well-typed way, while the lower layer is simply composed of untyped regular expressions that can leverage features from the underlying engine. This two-layer approach is also compatible with routing, unparsing, and either online of offline compilation of regular expressions. We implemented our technique in the OCaml Tyre package, along with a syntax extension that allows the use of a syntax similar to PCREs. While our main implementation relies on `ocaml-re`, an optimized and mature regular expression library for OCaml, we also implemented it as an OCaml functor (i.e., a function over modules) that is independent of the engine.

Our main contribution is the design and the implementation of Tyre, including a typed transformation technique which can be used to decompose such input validation problems into an untyped parsing step which can be delegated to preexisting engines, and a validation step. The rest of the paper is structured as followed. We first introduce typed regular expressions through real world code examples that uses Tyre in Section 2. We then describe our technique in Sections 3 and 4. We evaluate the expressivity and the performance claims in Section 5. Finally, we compare with existing approaches in Section 6 and conclude.

2 The Tyre library

Tyre has been used in real-world applications for parsing, printing and routing complex regular-expression-based formats such as logs and URLs. To introduce Tyre, we use the example of a simple website that classifies species of the Camelidae family. Regular expressions match URLs and define a REST API for our Camelidae classifiers. For instance, we could obtain information on a specific species with the URL camelidae.ml/name/Dromedary/, or we could list Camelids

```
1  open Tyre
2
3  let name : string Tyre.t = [%tyre "[[:alnum:]␣]+"]
4
5  let host : < domain:string ; port:int option > Tyre.t =
6    [%tyre "(?<domain>[^/:?#]+)(:(?<port>(?&int)))?"]
7
8  let api = function%tyre
9    |"(?&h:host)/name/(?&species:name)"
10     -> h, ‘Host species
11   |"(?&h:host)/hump/(?&hump:int)(\?extinct=(?&b:bool))?"
12     -> h, ‘Hump (hump, b)
```

Figure 1. HTTP endpoints for a website to classify camlids.

that have two humps through the URL camelidae.ml/humps/2. Finally, we want the ability to filter extinct species by adding ?extinct=true to the URL. Writing such API with normal regular expressions is often delicate since it not only needs to match the URL, but also to extract and convert parts of it. In the rest of this section, we assume basic familiarity with OCaml.

Figure 1 implements the routing for this REST API with typed regular expressions with automatic extraction. It uses an extended syntax similar to Perl Compatible Regular Expressions. We first define two independent typed regular expressions, `name` (Line 3) and `host` (Line 5), we then define our two URL endpoints and name the resulting route in `api` (Line 8). For ease of understanding, we added some type annotations, but they are not necessary.

The syntax `[%tyre "..."]` specifies a typed regular expression using the PCRE syntax. The `name` typed regular expression defined Line 3 is in fact a normal regular expression which recognizes species names made of a non-empty succession of alphanumerical characters and spaces. The resulting value is of type `string Tyre.t` which represents a typed regular expression `Tyre.t` which captures data of type `string`.

The `host` regular expression defined Line 5 matches (simplified) host names composed of a domain and an optional port using the syntax `"foo.com:123"`. To capture these two parts, it uses two *named* capture groups for the domain and the port, using the syntax `"(?<name>re)"`. Since we have two named capture groups, the regex `host` captures a domain and a port. The type indeed shows we capture an object with two fields. The domain is represented by the regular expression `[^/:?#]+` and thus captured as a string. The port is represented by `(?&int)` which uses another regular expression previous declared, in this case `Tyre.int` of type `int Tyre.t` capturing an integer. Since the port is optional, the name capture is wrapped by an option denoted with the syntax `(...)?`. This also changes the type of the capture to be an `int option`. The type of the capture for `host` is thus

`<domain:string; port:int option>`, which is an object with two fields, `domain` of type `string`, and `port` of type `int option`. Unlike usual PCREs, normal parentheses do not capture their content and are only used for priorities, like in the option in the case above.

Given these two regular expressions `name` and `host`, we can finally define the routing for our REST API. The routing regular expression is named `api` on Line 8 and defines two routes for search by names and by number of humps. It uses a syntax similar to pattern matching, but takes (typed) regular expressions. The syntax `(?&h:host)` is a shortcut for `(?<h>(?&host))` and means that it uses the regex `host` and binds the capture to `h`. The constant part of the path, `/name/`, is not captured and does not influence the type of the capture. In the `/hump/` path, we also capture a boolean that indicates if we must consider extincts Camelids. As said before, normal parentheses are non-capturing in Tyre, but since the query argument is optional, we use `?`. The type of `b` is thus `bool option`.

A note on performance While well known in theory, the practical performance of regular expressions is a delicate topic. In particular, the various Perl additions go far beyond the definition of regular languages. In Tyre, the typed layer is only composed of *regular* operators. The untyped layer however can use many features from the underlying engine and thus depends on its complexity. Regardless of this, composition of typed regular expressions, including through the syntax `"(?&re)"`, is free. The main implementation of Tyre relies on `ocaml-re`, an efficient OCaml library for regular expression matching using a lazy-DFA approach which provides matching in linear time.

2.1 Matching, routing and unparsing

In the previous example, we defined two regular expressions and a router. Tyre also provides an API to use typed regular expressions, presented in Figure 2. While the type of `name` and `host` is `Tyre.t`, the router `api` is in fact of type `Tyre.re` which represents the type of *compiled* regular expressions. The `Tyre.compile` function compiles arbitrary typed regular expressions while `Tyre.route` allows routing, as presented previously. Both return compiled regular expressions of type `Tyre.re`. Compiled typed regular expressions can be used with `Tyre.exec`, as shown below. We use the standard `result` type to account for errors. Here, the matching is a success, as shown by the `Result.Ok` constructor.

```
# Tyre.exec api "camlidae.ml/name/Bactrian␣camel"
- : (domain * [ ‘Host of string | ‘Hump of int * bool
    option ], error) result
= Result.Ok (camlidae.ml:-, ‘Host "Bactrian␣camel")
```

Regular expressions can also be *unparsed* with `Tyre.eval`. Unparsing consists of taking some fully formed data-structure of type `'a` and a typed regular expression of type `'a Tyre.t`, and output a string such that matching would provide the same data. We use it below to provide easy and automatic construction of links that will be automatically compatible with our REST API.

```
type 'a t
(* A typed regular expression capturing
   data of type 'a *)

type 'a re
(** A compiled typed regular expression of type 'a *)

val compile : 'a t -> 'a re

val route : 'a route list -> 'a re
(** [route [ tyre1 --> f1 ; tyre2 --> f2 ]] produces
    a compiled tyregex such that, if [tyre1] matches,
    [f1] is called, and so on. *)

val exec : 'a re -> string -> ('a, error) result
(** [exec cre s] matches the string [s] with [cre] and
    returns the extracted value. *)

val eval : 'a t -> 'a -> string
(** [eval re v] returns a string [s] such
    as [exec (compile re) s = v] *)
```

Figure 2. API to use Typed Regular Expressions

```
# let link_name h s =
    Tyre.eval [%tyre "(?&host)/name/(?&name)"] (h, s)
val link_name : domain -> string -> string
```

Unparsing can't fail but the result is not unique. Indeed, in the regular expression `"[a-z]*(?<number>[0-9]+)"`, only the number part is captured and the rest needs to be synthesized. This is where the properties of regular languages come into play, as we will see in Section 4.4.

2.2 Regular expressions without the sugar

As any tea or coffee aficionado will tell you, the flavor is best enjoyed without sugar. In the previous section, we used syntactic sugar to provide a familiar PCRE-like regular expression syntax. This syntax is implemented by simple decomposition into a set of combinators that are presented in Figure 3.

2.2.1 Simple combinators and composition

Just like before, the host is directly recognized by a regular expression as a string. The `Tyre.pcre` combinator allows to provide an untyped regular expression in PCRE syntax and will capture a string. This regular expression is completely untyped, and can thus use most features from the underlying engine.

```
let host : string Tyre.t = Tyre.pcre "[^/:?#]+"
```

To construct regular expressions with more complex extractions, we use operators such as <&> and *>. <&>, also called "`seq`", allows to compose regular expression sequentially. The data captured by `re1 <&> re2` is the pair of the data captured by `re1` and `re2`. Similarly, *>, also called "prefix", composes regular expression sequentially but ignores the data captured by the left hand side and only returns the data capture by the right part. By combining these operators, we can decide exactly which parts of the data we want to capture, in this case the host name and the species name. We obtain a regular expression which captures a pair of strings. `Tyre.str` allows us to define a *constant* regular expression that will always parse the given regular string `"/name/"` and return a unit.

```
let name : string Tyre.t = Tyre.pcre "[[:alpha:]␣]"
let by_name
  : (string * string) Tyre.t
  = host <&> (Tyre.str "/name/" *> name)
```

2.2.2 Introducing new data types

Previously, we used the combinator `Tyre.int: int Tyre.t` which matches the regular expression `[0-9]*` and returns an integer. The definition is shown below. It uses the constructor `Tyre.conv` to transform the data captured by a regular expression using two functions that turn `'a` into `'b` back and forth. It then returns a regular expression matching the same content, but capturing data of type `'b`. Here, we use it with the two functions `string_of_int` and `int_of_string` to convert back and forth between integers and strings.

```
let int : int Tyre.t =
  Tyre.conv string_of_int int_of_string
    (Tyre.pcre "[0-9]+")
```

More complex and structured types can be introduced. For instance one might note that our syntax extension returns object with fields corresponding to named capture groups even though our <&> combinator returns tuples. To do so, we write a converter that converts back and forth between these representations, as below. Of course, we are not limited to objects: records or smart constructors can also be used for instance.

```
let host =
  let to_ (domain, port) = object
    method domain = domain
    method port = port
  end
  in
  let of_ o = (o#domain, o#port) in
  conv to_ of_
    (pcre "[^/:?#]+") <&> opt (str":" *> int))
```

```

val pcre : string -> string t
(* [pcre "re"] parses the given PCRE and captures the
     matched string. *)

val str : string -> unit t
(* [str "..."] parses the given constant string and
     captures no data. *)

val (<&>) : 'a t -> 'b t -> ('a * 'b) t
(* [re1 <&> re2] matches [re1] then [re2] and captures
     the pairs. *)

val ( *>) : _ t -> 'a t -> 'a t
(* [re1 *> re2] is [re1 <&> re2] but ignores the capture
     by [re1]. *)

val (<|>) : 'a t -> 'b t -> [`Left of 'a|`Right of 'b] t
(* [re1 <|> re2] matches [re1] or [re2] and captures
     their data. *)

val opt : 'a t -> 'a option t
val list : 'a t -> 'a list t

val conv :  ('a -> 'b) -> ('b -> 'a) -> 'a t -> 'b t
(** [conv to_ from_ re] matches the same text as [re],
    but transforms the captured data. *)
```

Figure 3. Combinator API to create Typed Regular Expressions

One strength of this approach is that it puts the conversion functions together with the definition of the regular expressions. The conversion and validation functions are thus not only easier to verify, since the definitions are closer, but also to compose since simply composing the typed regular expressions is sufficient. This ease of composition allow programmers to scale to more complex grammars, as we show in Section 5.

3 Untyped regular expressions

Before presenting compilation, matching and unparsing for typed regular expressions, let us detail what we expect from the underlying untyped regular expression engine. We present this expected API as a module type `RE` shown in Figure 4.

Untyped regular expressions, represented by the type `re`, feature the usual regular operators `alt`, `concat` and `star` along with the base operator `char`. For simplicity we don't parameterize over the type of symbols, although that would be equally easy. We also assume that grouping is explicit, through the `group` and `rm_group` functions which respectively adds a group over a given regular expression and removes all underlying groups. The regular expression type can also contains other arbitrary operators (character sets, lookaround operators, bounded repetitions, ...). Finally, we assume the existence of an explicit `compile` function which takes a regular expression and turns it into an automaton. This API is not particularly unusual (although it is rarely expressed in terms of combinators) and can be directly mapped to the syntax of most regular expression engines.

Matching The literature on regular expression often distinguishes three levels of matching for regular expressions:

- Acceptance test, which returns a boolean.
- Substring matching, which returns an array where each index corresponds to a capture group. If the capture group matches, the substring is placed at the index. For capture groups under repetitions, the usual choice is to only returns the *last* matched substring.
- Full RE parsing, which returns the complete parsetree captured by the regular expression.

Here, we assume that the underlying engine only supports substring matching. The `exec` function takes a compiled regular expression, a string, and returns a list of groups which can be queried by indices. We also take the start and end position of the substring on which the matching should be done. The `all` function which matches repeatedly the given regular expression behaves similarly.

In addition, we suppose the existence of a *marking* API. The `mark` function marks a given regular expression with a `markid`. After matching, users can test if this particular regular expression was used during the matching. While this might seem like an unusual feature, it is in fact very similar to what is already implemented in engines supporting routing such as Ragel [Adrian Thurston 2017] or most regex-based lexers. Marks are also compatible with both the eager and the POSIX semantics. In Section 4.5, we will present some leads on how to remove the need for this API.

Inhabitants Finally, we assume that it is possible to obtain an arbitrary inhabitant of a given regular expression. This operation is straightforward for most regular expression languages, and can fail only if operations such as lookaround, intersection or negations are provided.

4 Typed regular expressions

We can now express our typed regular expression matcher as a module which takes an untyped regular expression matcher as an argument. We consider the rest of the code presented in this section as parameterized over a module `Re` which answers the signature `RE` presented in Figure 4.

Typed regular expressions, of type `'a tre` are either an alternative (capturing a sum type), concatenation (a tuple) or repetition (a list). They can also be an untyped regular expression, which will capture a string. Finally, a typed regular expression can be combined with a converter f to change the type of its capture. A converter, of type `('a, 'b) conv` is simply represented by a pair of functions in each direction. For convenience, we note `'a + 'b` for ``[`Left of 'a|`Right of 'b]``. Finally, we can ignore the result of a regular expression. The capture is then of type `unit`.

```
module type RE : sig
  type re

  val char : char -> re
  val alt : re list -> re
  val concat : re list -> re
  val star : re -> re
  val group : re -> re
  val rm_group : re -> re
  ...

  (* Compilation *)
  type automaton
  val compile : re -> automaton

  (* Matching *)
  type groups
  val exec :
    int -> int -> automaton -> string -> groups
  val all :
    int -> int -> automaton -> string -> groups list
  val get : groups -> int -> string

  (* Marking *)
  type markid
  val mark : re -> markid * re
  val test : markid -> groups -> bool

  (* Inhabitant *)
  val inhabitant : re -> string
end
```

Figure 4. API for untyped regular expressions

```
type _ tre =
  | Alt : 'a tre * 'b tre -> ( 'a + 'b ) tre
  | Concat : 'a tre * 'b tre -> ('a * 'b) tre
  | Star : 'a tre -> ('a list) tre
  | Untyped : Re.re -> string tre
  | Conv : ('a, 'b) conv * 'a tre -> 'b tre
  | Ignore : _ tre -> unit tre

type ('a, 'b) conv = {
  a : 'a -> 'b ;
  b : 'b -> 'a ;
}
```

Figure 5. Definition of typed regular expressions

```
type _ witness =
  | WGroup : int -> string witness
  | WConv : ('a, 'b) conv * 'a witness -> 'b witness
  | WAlt : markid * 'a witness * 'b witness -> ('a + 'b)
      witness
  | WConcat : 'a witness * 'b witness -> ('a * 'b)
     witness
  | WRep : int * 'a witness * Re.automaton -> 'a list
     witness
  | WIgnore : unit witness
```

Figure 6. Definition of the witness type

Given these two types, compiling and matching typed regular expressions can be achieved through the following steps:

1. Derive an untyped regular expression from the typed one and compile it to an automaton.
2. Compute a witness which reifies the type of the output and its relation with the capture groups.
3. Match the strings using the automaton and obtain the capture groups.
4. Reconstruct the output using the groups and the witness.

To achieve these steps, two functions are sufficient:

- `build` of type `'a tre -> Re.re * 'a witness` which builds the untyped regular expression and the witness.
- `extract` of type `'a witness -> Re.groups -> 'a` which uses a witness to extract the data from the capture.

We can then provide the typed compilation and matching functions `Tyre.compile` and `Tyre.exec` by using the untyped API:

```
type 'a cre = automaton * 'a witness
let compile : 'a tre -> 'a cre = fun tre ->
  let re, wit = build tre in
  (rcompile re, wit)

let exec ((automaton, witness) : 'a cre) s : 'a =
  extract witness (rexec automaton s)
```

4.1 Capturing witnesses

Before presenting an implementation of the functions `build` and `extract`, let us detail which pieces of information the witness will have to contain. The definition of the witness type is given in Figure 6. First off, the witness type is indexed by the type of the capture. In particular this means that it will track alternatives, concatenations and repetitions at the type level similarly to typed regular expressions. It must also contain the converter functions in the `WConv` case.

Capture groups and repetitions The witness must contain the index and the nature of each capture group. Capturing groups mostly correspond to the leaves of the typed regular expressions using the constructor `Untyped`. However, we must consider what happens for `Untyped` constructors under repetitions. Indeed, most regular expression engines only return strings for each capture group, following the API in Figure 4. If a capture group is under a repetition and matches multiple times, only the last capture is returned. To resolve this issue, we record in the witness the regular expression under the repetition and use a multi-step approach to re-match the repeated segment.

More concretely, if we want to match the regular expression `"a(b(?<x>[0-9]))*"`, we start by matching the simplified regular expression `"a(?<rep>(b[0-9])*)"`. We identify the range of the repeated segment, we then match `"b(?<x>[0-9])"` repeatedly on that range. Naturally, this incurs a cost proportional to the star height of the regular expression. Note that this does *not* imply an exponential complexity of matching in terms of length of the input, unlike PCREs. For a fixed regular expression, the matching is still linear in terms of the length of the string if the underlying untyped matching is linear.

Consequently, the `WRep` constructor contains the index of the group which will match the whole repeated substring, a witness that can extract the captured data and the corresponding compiled regular expression.

Alternatives The witness must also inform us which branch of an alternative was taken during matching. This is where the marking API presented in Section 3 come into play. For each alternative, we record a mark that indicates which branch of the alternative was used. This can also be used to encode all the other construction that present a choice such as options, routing, ... The constructor `WAlt` thus contains a `markid` and two witnesses for each branch of the alternative.

4.2 Building the witness

The `build` function, presented in Figure 7, derives an untyped regular expression and a witness from a typed regular expression. It relies on an internal function, `build_at` which takes the current capturing group index, the typed regular expression, and returns a triplet composed of the next capturing group index, the witness and the untyped regular expression. Unusually for OCaml, the type annotation on Line 2 is mandatory due to the use of GADTs. The universal quantification is made explicit through the use of the syntax "`type a.`" This also guarantees that the type of the regular expression and of the witness correspond. The `Untyped` case, on Line 4, proceeds by simply removing all the internal groups present inside the regular expression, wrapping the whole thing in a group, to ensure that only one group is present, and appropriately incrementing the

```
1 let rec build_at
2   : type a. int -> a t -> int * a witness * re
3   = fun i tre -> match tre with
4     | Untyped re ->
5       (i+1), WGroup i, Re.group (Re.rm_group re)
6     | Conv (conv, e) ->
7       let i', w, re = build_at i e in
8       i', WConv (conv, w), re
9     | Alt (e1,e2) ->
10      let i', w1, re1 = build_at i e1 in
11      let id1, re1 = Re.mark re1 in
12      let i'', w2, re2 = build_at i' e2 in
13      i'', WAlt (id1, w1, w2), Re.alt [re1; re2]
14    | Concat (e1,e2) ->
15      let i', w1, re1 = build_at i e1 in
16      let i'', w2, re2 = build_at i' e2 in
17      i'', WConcat (w1, w2), Re.concat [re1; re2]
18    | Rep e ->
19      let _, w, re = build_at 1 e in
20      let re_star = Re.group (Re.rep (Re.rm_group re)) in
21      (i+1), WRep (i,w,Re.compile re), re_star
22    | Ignore e ->
23      let _, _, re = build_at 1 e in
24      i, WIgnore, Re.rm_group re
25
26 let build tre = let _, w, re = build_at 1 tre in (w, re)
```

Figure 7. Implementation of the `build` function

number of groups. Most of the other cases proceed in similarly straightforward ways. Of particular note is the `Alt` case, where the regular expression in the left branch is marked using `Re.mark` on Line 11. For the `Rep` case, note that the recursive case, on Line 19, starts again at group index 1, and the resulting group number is ignored. Indeed, since we will use this regular expression independently, the other groups do not matter. Finally, for the main `build` function, we start with index `1`, `0` being usually reserved for the complete matched string.

4.3 Extracting using the witness

After matching, we use the witness to extract information from the capture groups. Extraction is implemented by a pair of mutually recursive functions, `extract` and `extract_list` which are shown in Figure 8. Most of the cases follow directly from the structure of the witness. The typing ensures that the type returned by the extraction is correct, including in the converter case on Line 6. The main case of interest is the `WRep` case, which calls the `extract_list` function. This function applies the regular expression under the list repeatedly using `Re.all`, as described previously.

```
1 let rec extract
2   : type a. string -> a witness -> groups -> a
3   = fun str w g -> match w with
4     | WGroup i -> Re.get g i
5     | WIgnore -> ()
6     | Conv (w, conv) -> conv.a (extract str w g)
7     | WAlt (i1,w1,w2) ->
8       if Re.marked g i1 then
9         ‘Left (extract str w1 g)
10      else
11        ‘Right (extract str w2 g)
12    | WConcat (e1,e2) ->
13      let v1 = extract str e1 g in
14      let v2 = extract str e2 g in
15      (v1, v2)
16    | WRep (i,we,re) -> extract_list str we re i g
17
18 and extract_list
19   : type a.  string -> a witness -> Re.re -> int ->
20     groups -> a list
21   = fun str w re i g ->
22     let f = extract str w in
23     let pos, pos' = Re.offset g i in
24     let len = pos' - pos in
25     List.map f @@ Re.all pos len re original
```

Figure 8. Implementation of `extract`

4.4 Unparsing

In addition to traditional parsing, our technique allows us to easily support unparsing. Unparsing takes a typed regular expression, a value, and returns a string. Note that this is slightly different than "flattening" used in the full RE parsing literature. Indeed, here the value (and the typed regular expression) does not cover the complete parsetree. Some bits of the input can be ignored using the `*>` and `<*` combinators (or the `Ignore` constructor, in our simplified version). However, thanks to the fact that the language is regular, we can invent new inhabitants for the missing parts. The `unparse` function, shown in Figure 9, can thus be implemented by a recursive walk over the structure of the typed regular expression. The function `inhabitant` used in Line 14 is an extension of the function `Re.inhabitant` to typed regular expressions, with a type `_ tre -> string`.

4.5 Extensions

4.5.1 Routing

Let us consider n routes each composed of a typed regular expression `'a`$_k$ `tre` and a function of type `'a`$_k$ `-> 'output` for k from 0 to $n-1$. To compile this set of routes, we first build the alternative of each of the associated untyped regular expressions. We also collect all the witnesses, along with the associated callbacks. To know which route was matched, we

```
let rec unparseb
  : type a . a tre -> Buffer.t -> a -> unit
  = fun tre b v -> match tre with
    | Regexp (_, lazy cre) -> begin
        if Re.test cre v then Buffer.add_string b v
        else  raise Wrong_string
       end
    | Conv (tre, conv) -> unparseb tre b (conv.from_ v)
    | Opt p -> Option.iter (unparseb p b) v
    | Seq (tre1,tre2) ->
      let (x1, x2) = v in
      unparseb tre1 b x1 ; unparseb tre2 b x2
    | Ignore tre ->
      Buffer.add_string b (inhabitant tre)
    | Alt (treL, treR) -> begin match v with
        | `Left x -> unparseb treL b x
        | `Right x -> unparseb treR b x
      end
    | Rep tre -> List.iter (unparseb tre b) v

let unparse : 'a t -> 'a -> string = fun tre v ->
  let b = Buffer.create 17 in
  try unparseb tre b; Some (Buffer.contents b)
  with Wrong_string -> None
```

Figure 9. Implementation of unparsing

simply mark each of the routes using the marking API. Such a compilation process is fairly direct and allows the regular expression engine to efficiently match the total set of routes, without having to match each route one by one.

4.5.2 Extraction without marks

In Section 4.1 we used marks to track which branches were taken during matching. Unfortunately, marks are not available in many regular expression engines, most notably the native Javascript one. Additionally, emulating marks using groups is not as easy as it might seem: Let us consider the regular expression `line(?<len>[0-9]*)|(?<empty>)` that matches strings of the format `line12`, `line` without any number, or the empty string. Under the POSIX semantics, while matching the string `line`, neither of the capture groups will capture, thus preventing us from detecting which branch was taken. We need to add yet another extra group around the complete left part of the alternative. Even then, if both sides of the alternative are nullable, it is possible that none of the group matches and we must favor one of the branches.

While this allows us to implement our technique on engines that do not support marks, as is the case in Javascript, this method forces us to add many additional groups throughout the regular expression, thus degrading performance.

4.5.3 Staged extraction

We previously built a capturing witness during compilation and used this witness to reconstruct the output datatype during extraction. Using staged meta-programming, we could build the extraction code directly and either evaluate it to extract the captured data or output it and compile it, for offline usage.

We implemented a prototype following this idea using MetaOCaml [Kiselyov 2014], an extension of OCaml for staged meta-programming. MetaOCaml provides a new type `'a code` which represents quoted code that evaluates to values of type `'a`, and new quotation syntax for staged code: `.< get .~s i >.` represents a piece of staged code while `.~s` represents an antiquotation for staged code.

The only required changes in the API is to replace converters by staged functions. Internally, `extract` is replaced by `codegen : 'a witness -> groups code -> 'a code` which takes a staged identifier referring to the capture groups and generates the code doing the extraction.

For instance, the code handling concatenation is shown below. The final code emitted by `extract` is very close to the optimal hand-written extraction code.

```
  | Concat (re1,re2) ->
    let code1 = codegen re1 groups in
    let code2 = codegen re2 groups in
    .< (.~code1, .~code2) >.
```

5 Evaluation

We aim to evaluate our approach in two aspects: an informal look at Tyre's ease of use compared to similar libraries and various benchmarks that measure both the comparative performances and the overhead cost. We considered two scenarios: an HTTP parser and an URI parser. Unfortunately, due to various limitations regarding MetaOCaml and native code, we were not able to compare our staged version with the normal Tyre extraction.

5.1 HTTP parsing with parser combinators

The Angstrom library is a high-performance parsing combinator library for OCaml inspired by Haskell's attoparsec. It claims very high efficiency and was explicitly built for dealing with network protocols. One of the basic examples provided by the library is a simplified HTTP parser. Since such simplified HTTP requests and responses form a regular language, we converted this parser to Tyre. Angstrom's API relies heavily on an applicative API composed of operators such as `*>`, `<*` and `lift`, making the translation to Tyre straightforward. The simplified parser for HTTP requests is shown in Figure 10 and the complete parser is roughly the same size as the Angstrom one. One major difference between combinator libraries such as Angstrom and Tyre is the bi-directional aspect of the latter. In the case of this HTTP

```
type request = {
  meth : string ;
  uri : string ;
  version : int * int ;
  headers : (string * string) list ;
}

let lex p = p <* blanks
let version =
  str"HTTP/" *>
    ((int <* char '.') <&> int)

let request_first_line =
  lex meth <&> lex uri <&> version

let header =
  token <&> (char ':' *> blanks *> take_till P.eol)

let request =
  let to_ (((meth, uri), version), headers) =
    {meth; uri; version; headers}
  and of_ {meth; uri; version; headers} =
    (((meth, uri), version), headers)
  in
  conv to_ of_ @@ seq
    (request_first_line   <* eol)
    (rep (header <* eol) <* eol)
```

Figure 10. Simplified HTTP request parser using Tyre

Angstrom	Tyre	Tyre, test-only
$28.3 \pm 1.8ms$	$11.6 \pm 0.13ms$	$7.6 \pm 0.013ms$

Figure 11. Parsing 100 HTTP requests with various parsers

parser, the cost incurred for unparsing is a single additional function in the final `conv` call.

To evaluate performances, we measure the time to parse a list of 100 HTTP requests. We measure Angstrom and Tyre's extraction 200 times and show the results in Figure 11. In addition, we measured the matching part of the Tyre-based parser, without extraction. This allows to measure the overhead of Tyre's extraction compared to directly using `ocaml-re`. Tyre is around 2.4 times faster than Angstrom. Testing using the underlying regular expression engine takes around 60% of Tyre's parsing time. This result confirms our expectations: parsing with regular expressions is much faster than using a general purpose parsing combinator library like Angstrom and Tyre's automated extraction doesn't introduce unexpected costs that would compromise its usability for parsing. We also note that for such a simple extraction (only simple combination of tuples), Tyre parsing is mostly spent in the underlying regex engine itself.

The main take-away here is not that Tyre is faster (which was expected), but that Tyre's API, which mimics the applicative functor fragment of common parser combinator library, allows to easily convert parsing code. By doing this conversion, we gain both performance benefits and additional features such as unparsing.

5.2 URI parsing

The `ocaml-uri` library is an OCaml library to parse, print and manipulate Uniform Resource Identifiers, or URIs, including URLs. `ocaml-uri` uses a regular-expression based parser using `ocaml-re`. While the regular expression is complex, it is completely specified by the RFC 3986 [Berners-Lee et al. 2005] and has numerous test cases. The extraction code, however, is less specified and is implemented by a fairly delicate piece of code with numerous branches that must also decode the encoded parts of the URI. To simplify this process, the original parser was in fact broken into two pieces: the authority (often of the form `"user@domain.com:8080"`) was re-parsed separately. Furthermore, the definition of the regular expression and the extraction code doing the decoding were completely disconnected.

We reimplemented the parsing code of `ocaml-uri` with Tyre. Our version is feature-par with the original and passes all the tests. This new implementation brings the decoding logic and the parsing logic closer, making the code cleaner. We show the resulting code to decode the authority section of the URI in Figure 12. Tyre allows us to define utility combinators such as `encoded` and `decode` which respectively cast a string into an abstract `encoded` type and decode said encoded strings. We can then compose the various pieces of the parser by following the specification. The `(|>)` operator is the reverse function application. Note that the detailed content of the base regular expressions for user-info and host are not shown here, as the precise definitions are rather complex. In the Tyre version of the library, the authority section is not parsed separately anymore. Since tyre allows both syntax description and extraction to compose freely, we can simply inline the authority parser in the larger regular expression.

We compare the performances of our modified implementation of `ocaml-uri` with the original version in Figure 13. We measured the time taken to parse various URIs with the original library, our modified version, and finally the time taken by matching without extraction. This last time serves as a baseline, as it is common to both versions. Our benchmark set is composed of 6 URIs taken from `ocaml-uri`'s own testsuite that exercises various parts of the grammar and are shown in Figure 14. We repeated the measurements during one minute, and shows the 95% confidence interval.

For this more complex example, we can see that regular expression matching only occupies 10% to 20% of the parsing time. Most of the time is taken by the extraction since it involves some decoding. The Tyre version is always faster

```
let encoded = conv Pct.cast_encoded Pct.uncast_encoded
let decode = conv Pct.decode Pct.encode

let scheme =
  pcre "[^:/?#]+" <* char ':'
  |> encoded |> decode

let userinfo : Userinfo.t t =
  pcre Uri_re.Raw.userinfo <* char '@'
  |> encoded
  |> conv userinfo_of_encoded encoded_of_userinfo

let host =
  pcre Uri_re.Raw.host
  |> encoded |> decode

let port =
  let flatten = conv Option.flatten (fun x -> Some x) in
  char ':' *> opt pos_int
  |> opt |> flatten

let authority =
  str "//" *> opt userinfo <&> host <&> port
```

Figure 12. Parsing of the authority section of an URL

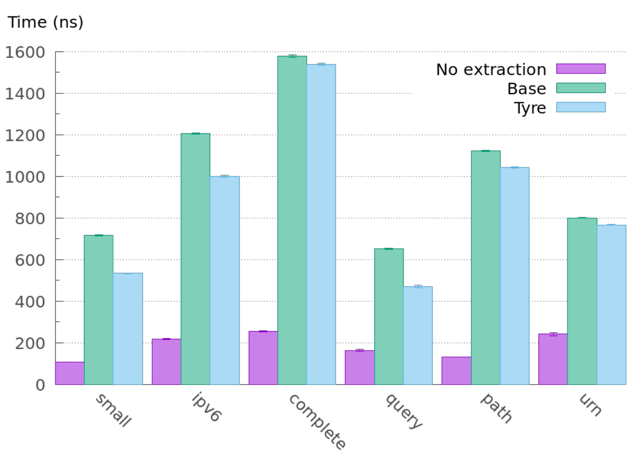

Figure 13. Performances of URI parsing

small: `http://foo.com`
ipv6: `http://%5Bdead%3Abeef%3A%3Adead%3A0%3Abeaf%5D`
complete: `https://user:pass@foo.com:123/a/b/c?foo=1&bar=5#5`
query: `//domain?f+1=bar&+f2=bar%212`
path: `http://a/b/c/g;x?y#s`
urn: `urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6`

Figure 14. Definition of the URIs

than the original version, sometimes marginally so. Part of this speedup is due to the absence of separate parsing for the authority section. Indeed, "small" and "ipv6" URIs, which showcase large speedups, almost only contain the authority field. On the other hand "path" and "urn", which showcase very small speedups, do not contain a significant authority section. Nevertheless, even in these cases, the Tyre version still showcase a small speedup. We believe this is due to the fact that Tyre will extract only the necessary part by using the branching directly deduced from the regular expressions. Given that the branching for URIs is quite complex, a manual version is more likely to extract parts of the string even if not strictly necessary.

6 Comparison

Regular expression matching Substring matching for regular expressions is a well explored field. Even though the classics are well known [Brzozowski 1964; Thompson 1968], new techniques are still being discovered [Fischer et al. 2010; Groz and Maneth 2017; Vasiliadis et al. 2011]. Cox [2010] presents a survey of various efficient approaches. In particular, significant efforts have been dedicated to improve both the theoretical and practical performances of substring matching which resulted in high quality implementation such as Re2 [Cox 2007], PCRE and Hyperscan [Intel 2018]. One objective of Tyre is precisely to reuse these high quality implementations and be able to choose between their various trade-offs, while still enjoying type-safe extraction.

Full regular expression parsing Full regular expression parsing consists of obtaining the complete parsetree for a string according to a given regular expressions. Frisch and Cardelli [2004]; Grathwohl et al. [2014]; Kearns [1991] present efficient algorithms for the greedy semantics while Sulzmann and Lu [2014] present an algorithm for POSIX parsing. All these algorithms are more efficient than the technique presented here. They however only account for parsing in the presence of a fairly restricted set of features and have rarely led to a reusable, optimized and featureful regular expression library.

In particular, none of these approaches account for lazy-DFA, regular expression containing both greedy and POSIX semantics, or the numerous extensions to regular expressions such as lookaround operators. They also aren't necessarily portable to environments where the engine is already provided, such as Javascript.

Parser combinators Parser combinators are well known for providing a convenient API to build parsers. Most parser combinator libraries allow to express classes of grammar bigger than regular languages. The parsing technique are quite varied, from recursive-decent-like techniques which can express context-sensitive grammars with arbitrary look-ahead to PEGs [Ford 2004], thus offering far less efficient parsing than regular languages.

Many parser combinator libraries offer a monadic API, extended with applicative operators (`<*>`, `*>`, `<*`, ...) and alternatives (`<|>`). A recurring topic for monadic APIs is to leverage their applicative subset, which corresponds to the "static" portion of the language, for optimisations. In this context, `regex-applicative` [Cheplyaka [n. d.]] provides the applicative subset of the common parser combinator API and corresponds exactly to regular expression parsing. We are not aware of any parser combinator libraries for general grammars that uses this characteristic as an optimisation technique.

Unlike `regex-applicative`, Tyre's typed regular expressions do not form an applicative functor. Indeed, a functor would give `<|>` the type `'a tre -> 'a tre -> 'a tre` and `fmap` the type `('a -> 'b) -> 'a tre -> 'b tre`, which would render unparsing impossible. We were however able to provide most of the common applicative operators with the expected type, as shown in the HTTP parsing example. Another difference between `regex-applicative` and Tyre is that the former uses a custom regex parser while Tyre can delegate matching to a pre-existing one.

Lexer and Parser generators Generators for parser generators are very commonly used to define lexers, such as `lex`. Others, such as Ragel [Adrian Thurston 2017] or Kleenex [Grathwohl et al. 2016], are intended for more general purposes. Most of those, Kleenex excepted, rely on either regular expression substring matching, or even just matching. Kleenex, on the other hand, provides efficient streaming parsing for finite state transducers which allows to write complex string transformations easily. The main characteristic of generators is of course their static nature: the regular expression is available statically and turned into code at compile time. On the other hand, regular expression engines such as Re2 can compile a regular expression dynamically. This enable many uses cases such as reading a file, composing a regular expression, and applying it. Techniques such as lazy-DFA or JITs that minimize the compilation time are extremely valuable in these dynamic use-cases. By being parametric in the underlying engine, Tyre aims to be usable in both static and dynamic contexts. In particular, our staged version presented in Section 4.5.3 can both execute the extraction code, but also emit the generated code in a file and use it later, potentially enabling further optimizations.

7 Conclusion

Writing a complete, efficient regular expression engine that supports a rich feature set is a complex task. Furthermore, combining such rich feature set with full regular expression parsing is still mostly an open problem. In this paper, we presented a technique which provides automatic and type-safe parsing and unparsing for regular expression engines as long as they provide substring matching. While not as efficient as full regular-expression parsing, this technique allows to easily leverage the rich feature set of these engines, while still enjoying type-safe extraction. In particular, we showed that our technique can be extended with a staged-metaprogramming approach compatible with both online and offline compilation of regular expressions.

We implemented our technique as a convenient and efficient OCaml combinator library, Tyre, along with a syntax extension that provides a familiar PCRE-like syntax. We evaluated the practicality of our library through real use cases. The Tyre library has been used by external users on various contexts, ranging from web-programming to log search.

In the future, we aim to diversify the underlying engines available out of the box. In particular, we would like to be able to write typed regular expressions that can run either with online compilation engines like `ocaml-re`, offline compilation for lexers, or Javascript regular expressions. We also aim to improve the general performances of Tyre, notably for repetitions.

References

Adrian Thurston. 2017. *Ragel*. https://www.colm.net/open-source/ragel/.

Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. 2005. *Uniform Resource Identifier (URI): Generic Syntax*. STD 66. RFC Editor. http://www.rfc-editor.org/rfc/rfc3986.txt http://www.rfc-editor.org/rfc/rfc3986.txt.

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494.

Roman Cheplyaka. [n. d.]. *regex-applicative*. https://github.com/feuerbach/regex-applicative.

Russ Cox. 2007. Implementing Regular Expressions. (2007). https://swtch.com/~rsc/regexp/.

Russ Cox. 2010. Regular Expression Matching in the Wild. (March 2010). https://swtch.com/~rsc/regexp/regexp3.html.

ECMAScript 2018. *ECMAScript Specification Suite – RegExp Objects*. Standard ISO/IEC 22275:2018. https://www.ecma-international.org/ecma-262/9.0/index.html#sec-regexp-regular-expression-objects

Sebastian Fischer, Frank Huch, and Thomas Wilke. 2010. A play on regular expressions: functional pearl. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 357–368. https://doi.org/10.1145/1863543.1863594

Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 111–122. https://doi.org/10.1145/964001.964011

Alain Frisch and Luca Cardelli. 2004. Greedy Regular Expression Matching. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings (Lecture Notes in Computer Science)*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.), Vol. 3142. Springer, 618–629. https://doi.org/10.1007/978-3-540-27836-8_53

Niels Bjørn Bugge Grathwohl, Fritz Henglein, and Ulrik Terp Rasmussen. 2014. Optimally Streaming Greedy Regular Expression Parsing. In *Theoretical Aspects of Computing - ICTAC 2014 - 11th International Colloquium, Bucharest, Romania, September 17-19, 2014. Proceedings (Lecture*

Notes in Computer Science), Gabriel Ciobanu and Dominique Méry (Eds.), Vol. 8687. Springer, 224–240. https://doi.org/10.1007/978-3-319-10882-7_14

Niels Bjørn Bugge Grathwohl, Fritz Henglein, Ulrik Terp Rasmussen, Kristoffer Aalund Søholm, and Sebastian Paaske Tørholm. 2016. Kleenex: compiling nondeterministic transducers to deterministic streaming transducers. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 284–297. https://doi.org/10.1145/2837614.2837647

Benoît Groz and Sebastian Maneth. 2017. Efficient testing and matching of deterministic regular expressions. *J. Comput. Syst. Sci.* 89 (2017), 372–399. https://doi.org/10.1016/j.jcss.2017.05.013

Philip Hazel. 2015. *PCRE.* http://pcre.org/.

Intel. 2018. *Hyperscan.* https://github.com/intel/hyperscan.

Steven M. Kearns. 1991. Extending Regular Expressions with Context Operators and Parse Extraction. *Softw., Pract. Exper.* 21, 8 (1991), 787–804. https://doi.org/10.1002/spe.4380210803

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

Martin Sulzmann and Kenny Zhuo Ming Lu. 2014. POSIX Regular Expression Parsing with Derivatives. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer, 203–220. https://doi.org/10.1007/978-3-319-07151-0_13

Ken Thompson. 1968. Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422. https://doi.org/10.1145/363347.363387

Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2011. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization, IISWC 2011, Austin, TX, USA, November 6-8, 2011*. IEEE Computer Society, 216–225. https://doi.org/10.1109/IISWC.2011.6114181