Oregano: Staging Regular Expressions with Moore Cayley Fusion

Jamie Willis
Imperial College London
United Kingdom

Nicolas Wu
Imperial College London
United Kingdom

Tom Schrijvers
KU Leuven
Belgium

Abstract

Regular expressions are a tool for recognising regular languages, historically implemented using derivatives or non-deterministic finite automata. They are convenient for many light-weight parsing workloads, but their traditional formulation only lends them to matching text, not returning fully-structured results. This contrasts with other forms of parsing, where the aim is to extract meaningful data, for example abstract syntax trees. Yet, most regular expression libraries do not support this useful output, and those that do are often slower, and backed by parser combinator libraries.

This paper presents Oregano, a redesign of regular expressions to make use of Moore machines as the underlying machinery; this way the regular expression matcher can produce results. We further show how to produce heterogeneous results, providing a classic applicative interface. To make this representation performant, we leverage the relationship between Cayley representations, continuation-passing style, and staged meta-programming to generate performant code for regular expression matching with fully-structured results.

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

Keywords: Regular Expressions, Parsing, Staging

ACM Reference Format:
Jamie Willis, Nicolas Wu, and Tom Schrijvers. 2022. Oregano: Staging Regular Expressions with Moore Cayley Fusion. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium (Haskell '22), September 15–16, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3546189.3549916

Haskell '22, September 15–16, 2022, Ljubljana, Slovenia

ACM ISBN 978-1-4503-9438-3/22/09…$15.00
https://doi.org/10.1145/3546189.3549916

1 Introduction

Regular expressions [13] are perhaps the most commonly used form of parser. Not limited to the realms of compiler engineering, or domain specific data languages, regular expressions are an incredibly useful tool applicable to a wide variety of different problems. They are theoretically grounded in finite automata: either deterministic (DFA), or non-deterministic (NFA) [2, 23, 29]. This gives a way of describing regular expressions and provides a model for their execution. Typical regex engines do not produce results, as such, but matches; this means that given a regex and some input, it is possible to query if (or where) matches occur within the input. However, this does not produce meaningful parse trees, or other data, other than extracting results within *match groups*.

In contrast, more general tools for parsing larger classes of languages have more focus on producing these results; in particular, parser combinator libraries tie the production of results intrinsically into the description of the parser for a grammar. To their credit, regex engines are more specialised, and, as such, can achieve a higher performance.

As an example of a useful result, here is the (regular—so recursion is not allowed) BNF grammar for emails[1]:

⟨*word*⟩ ::= '0' .. '9' | 'a' .. 'z' | 'A' .. 'Z' | '_' | '-'
⟨*extword*⟩ ::= '.' | '+' | ⟨*word*⟩
⟨*email*⟩ ::= ⟨*user*⟩ '@' ⟨*domain*⟩
⟨*user*⟩ ::= ⟨*extword*⟩+
⟨*domain*⟩ ::= ⟨*word*⟩ ('.' ⟨*word*⟩)*

The result of matching this regular expression is a representation of the email using the Email datatype.

```
data Email = Email {user :: String, domain :: [String]}
```

This datatype encodes the username and the domain of the email address. The superfluous `"@"` in the email does not form any part of this type. Each `"."` in the email delimits part of the domain, so this represented by a list. Using vanilla regex libraries, these results could be extracted using capture groups, but these would be provided back as a list of strings (along with the match positions) and not as the more immediately useful Email type. In constast, parser combinator libraries [9–11, 17, 27, 28] allow the user to specify how results should be generated by the parser: this is usually achieved in a monadic way, though applicative or

[1] BNF is chosen instead of regex because it allows the explicit naming of sub-expressions that otherwise obscure the expression. For completeness, here is the regex: `[a-zA-Z0-9_\-.+]+@[a-zA-Z0-9_\-](\.[a-zA-Z0-9_\-])*`

selective [18, 32] styles are applicable to a wide variety of practical parsing tasks, including regular expressions.

We present Oregano, a library where interesting results can be combined arbitrarily and superfluous ones can be forgotten. Its API is similar to a traditional applicative parser combinator library—in fact, anything more powerful, like selective functors or monads, would cease to be regular. In addition, any recursion in the description of these regular expressions results in non-termination: introducing recursion into an applicative-style interface yields context-free power, which is undesirable here. Here is an implementation of the email parser in this style:

```
sepBy1 :: Reg a → Reg b → Reg [a]
sepBy1 r sep = (:) ‹$› r ‹*› some (sep *› r)

word = concat [['a'..'z'], ['A'..'Z'], ['0'..'9']
              , ['_'], ['-']]
extword = '.' : '+' : word

email :: Reg Email
email =
  Email ‹$› (some (charClass extword) ‹* lit '@')
        ‹*› sepBy1 (some (charClass word)) (lit '.')
```

Here the Applicative combinators (‹$›), (‹*›), (*›), and (‹*) are used as simple regular expression concatenation, but where the results are processed or combined in some way: (‹$›) alters the captured result of a parser using a function, (‹*›) combines two captured results by applying the first (a function) to the second (a value), and (*›) / (‹*) forgets the result of the regular expression it is not "pointing" at. The some combinator is acting as the regex + operation: it reads a sub-expression one or more times; similarly, many corresponds to *. The lit combinator matches a specific character in the input, and charClass matches any one of a set of characters, returning the matched character, if any.

Assuming match collects all parses in a list, we have:

```
match email "someone@example.com" ≡
[Email "someone" ["example", "com"]]
```

The problem is that NFAs do not naturally lend themselves to producing results of richer types like Email. Instead, our paper presents a library that uses Non-Deterministic Moore Machines as its theoretic basis, allowing for the outputting of meaningful, user-directed, results during the execution of the automaton on the given input. This provides a library with a natural API using non-recursive applicative functors.

To remain efficient, constructing and interpreting the resulting automaton at runtime must be avoided. Instead the idea of staging [26] is leveraged, which allows for statically determinable information to be hoisted out from runtime and into compile time. Using this technique, the overhead of constructing and evaluating the automaton can be removed, leaving a performant hand-written parser in its place. To facilitate staging, static information needs to be exposed in a way that avoids tight coupling with dynamic information: this can be done with a continuation-passing style approach, which naturally falls out when making use of the Cayley representation. This paper shows how by applying successive Cayley transformations, the structure of the regex can be exposed and easily staged.

Our experiments show that our approach out-performs commonly used regular expression libraries in Haskell at the matching task, as well as the `parsley` parser combinator library, which is built on similar principles [32].

Our paper is structured as follows: in Section 2 the relevant background on Cayley representations, NFAs, the translation from regular expression to NFA, and Moore machines are described. Then, our contributions are to:

- Demonstrate that the translation between a regular expression and its underlying automaton is an instance of a Cayley representation with a continuation-passing style compiler falling out of the structure. (Section 2.3)
- Show how, by shifting to a Non-deterministic Moore Machine (NMA) for an underyling abstraction, regular expressions can support user-directed results. (Section 3)
- Introduce furthermore heterogeneity into the NMA to allow for fully-structured results, that allow Oregano to adhere to classic applicative parser combinator APIs. (Section 3.2)
- Show how staging can be employed to improve the performance of this machinery, eliminating the overheads of translation and interpretation. (Section 4)
- Show how fold-build fusion can be staged to provide *staged fusion*, where not computing a value results in its code not being generated: this allows for zero-cost multiple-interpretations of regular expressions. (Section 4.4)
- Compare the performance of our approach against popular regular expression libraries in Haskell, as well as a C library. (Section 5)
- Provide a complete implementation of Oregano in the supplementary material.

The paper conludes by discussing related work (Section 6) and summarising (Section 7).

2 Background

This section describes Cayley representations, the translation from regular expressions to non-deterministic finite automata, as well as Moore machines.

2.1 Cayley Representations for Monoids

Monoids. Monoids are a set (or type) M associated with both an identity element e and a binary operation $\bullet$ subject to the following laws:

$$x \bullet (y \bullet z) \equiv (x \bullet y) \bullet z \tag{1}$$

$$x \bullet e \equiv x \equiv e \bullet x \tag{2}$$

A monoid $(M, e, \bullet)$ is usually represented in Haskell by the following typeclass:

```
class Monoid m where          -- M
  mempty :: m                 -- e
  (•)    :: m → m → m         -- •
```

Two useful monoids for this paper are lists ($[\mathsf{a}], [\,], (+)$) and endomorphisms ($\mathsf{a} \to \mathsf{a}, \mathsf{id}, (\cdot)$).

Cayley Representation. Cayley's original theorem [4] states that there is an isomorphism between groups and their permutations. However, this is more obviously applicable to function programming when applied to monoids, yielding an alternative formulation [24]: every monoid $(M, e, \bullet)$ is isomorphic to a submonoid of the monoid of endomorphisms on M. In Haskell:

```
newtype Cayley m = Cayley {unCayley :: m → m}
  deriving Monoid

toCayley :: Monoid m ⇒ m → Cayley m
toCayley x = Cayley (x•)

fromCayley :: Monoid m ⇒ Cayley m → m
fromCayley (Cayley cx) = cx mempty
```

The idea is that the Cayley transformation is a monoid homomorphism: instead of using • for the original monoid, regular function composition is used on the partial application of •; instead of mempty, the identity function is used. To convert back to the underlying monoid, the endomorphism is provided with the underlying mempty. Cayley representations are commonly used to avoid potentially expensive left-associated operations on tree-like structures.

Difference Lists. As a concrete, and well known example, difference lists [8] are an instance of the Cayley transformation on lists [24]:

```
type Diff a = Cayley [a]


diffCons :: a → Diff a → Diff a
diffCons x dxs = Cayley (x:) • dxs
```

A common use of the difference list is to improve the pathological performance of left-associated (+) operations, which might arise as part of tree-like recursion. The complexity of (+) is $O(n)$ in the length of the first argument but does not depend on the second. This means that left-associated (+) results in an $O(n^2)$ traversal time (as lists on the left are repeatedly retraversed) as opposed to $O(n)$ traversal time for right-associated (+) (where no list is retraversed). By using difference lists instead, the (+) is automatically associated to the right by function composition (which always replaces the monoidal operator), and an otherwise expensive traversal becomes far cheaper.

In contrast, this paper uses Cayley representations to improve the static knowledge of regular expressions.

2.2 Non-deterministic Finite Automata

The basis for most regular expression engines is the non-deterministic finite automaton (NFA). This is defined as a set of states paired with a set of transitions between states and a specific character to be matched (or indeed the ϵ transition). This can be represented by the following Haskell type:

```
data State = Split State State | Item Char State
           | Accept | Fail
```

Here Split represents the unconditional and non-deterministic ϵ transitions to two following states. The Item constructor represents a state which can transition to another state the given character appears at the head of the input. If the end of the input is reached when the machine is in the Accept state, this should result in a success. If the Fail state is reached, then this branch of the parse fails. As an example, the regular expression $a\,(b \mid c)\,d$ is represented as a value by:

```
ex1 = let join = Item 'd' Accept
      in Item 'a' (Split (Item 'b' join) (Item 'c' join))
```

Here, Haskell's let-bindings are leveraged to rejoin split states back into a common suffix (in this case join). The recursive nature of Haskell's let bindings can also be used to cleanly create self-looping (but still finite) automata too. As an example, here is the representation of the regex $(a|b)*$:

```
ex2 = let self = Split (Split (Item 'a' self) (Item 'b' self))
                       Accept
      in self
```

While Haskell's let suffices for now, when staging is incorporated to process these automata at compile-time, the let-bindings will need to be more explicitly inspectable. For that, an explicit JoinPoint node will later be added.

2.3 Translating Regular Expressions into NFAs

To express regular expressions, the following type is more convenient than automata:

```
data Reg = Lit Char | Cat Reg Reg | Alt Reg Reg
         | Eps | Rep Reg | Empt
```

Here, Lit matches a specific character, Cat combines two regexes in sequence, Alt tries one regex or the other, Eps matches the empty string, Empt just fails, and Rep is the Kleene-$\star$ operation. The earlier examples can be written as:

```
ex1 = Lit 'a' `Cat` (Alt (Lit 'b') (Lit 'c')) `Cat` Lit 'd'
ex2 = Rep (Alt (Lit 'a') (Lit 'b'))
```

Compared with their automata equivalents, these definitions do not use the same implicit stitching together of the states; instead Cat is used to connect two expressions, and Rep is used to provide the looping behaviour required.

The translation from this representation into the NFA description means translating the explicit concatenation found in Reg into the implied concatenation in State. In fact, the

relationship between Reg and State is that State is the defunctionalisation of the evaluation of Reg's Cayley representation: notice that Reg forms a monoid (quotiented by normalisation) with the unit being Eps and the append being Cat, now apply the Cayley transformation on Reg. This means that Eps should be mempty and Cat should be the Cayley-(•). Then the constructors are shaped as follows:

```
eps :: Cayley Reg
eps = mempty

cat :: Cayley Reg → Cayley Reg → Cayley Reg
cat = (•)

empt :: Cayley Reg
empt = Cayley (const Empt)

lit :: Char → Cayley Reg
lit c = Cayley (λk → Cat (Lit c) k)

alt :: Cayley Reg → Cayley Reg → Cayley Reg
alt (Cayley r1) (Cayley r2) =
  Cayley (λk → Alt (r1 k) (r2 k))
```

Post-transformation, Empt becomes a constant function that forgets the remaining structure (since failing ensures it cannot be reached); Lit becomes the partial concatenation of a character read with the remaining structure left to come; and Alt pushes the remaining structure into both branches.

The tricky case, Rep r, can be converted in several ways: most directly, it can be translated by exiting the Cayley representation and then re-injecting into it after applying Rep.

```
rep :: Cayley Reg → Cayley Reg
rep = toCayley · Rep · fromCayley
```

However, a different approach is to create a composite implementation of Rep using Cat, Alt, and Eps:

```
rep r = let self = Cat r self `Alt` Eps in self
```

This needs to be translated into the Cayley representation by simple subsitution of the old constructors for the new ones:

```
rep r = Cayley (λk → let self = cat r self `alt` eps
                     in unCayley self k)
```

Then we can simplify this term using the definitions above and equational reasoning to arrive at a new implementation for the Cayley representation:

```
rep :: Cayley Reg → Cayley Reg
rep (Cayley r) =
  Cayley (λk → let self = Alt (r self) k in self)
```

The fact that rep is implemented using explicit sharing of states is important, as this creates a finite automaton as opposed to an infinitely sized one; in Haskell, truly infinite structures are usually fine, since lazy evaluation means that the structure is only unwound as much as is required. However, for our purposes this will not suffice, as an aim of this paper will be to stage the implementation (Section 4): to statically compile the representation of one of these machines to a finite amount of code, it must necessarily be finite itself.

The idea now is to eliminate Cat and Eps entirely, resulting in a true continuation-passing style. There is one use of Cat left in the above definitions for the operators, and that is in lit. Notice that this is also the only case where the types do not completely line up with the constructors of State. Instead, State has an explicit continuation state for the Item case, which does remove the need for an explicit Cat. In addition, the Eps node is now only required for the fromCayley :: Cayley Reg → Reg, which provides the epsilon expression to complete the end of the expression, which is then optimised away by normalising with respect to the monoid-laws. Instead, the corresponding idea of the State is the Accept state, which, like Eps in the Cayley-representation, only appears at the very terminus of an automaton. That makes it the natural replacement for Eps.

The conversion from Reg to State now follows the definitions of the above Cayley representation operations, with the constructors substituted and Cat and Eps eliminated:

```
compile :: Reg → State
compile reg = cps reg Accept where
  cps :: Reg → State → State
  cps Eps next = next
  cps Empt _ = Fail
  cps (Lit c) next = Item c next
  cps (Cat r1 r2) next = cps r1 (cps r2 next)
  cps (Alt r1 r2) next = Split (cps r1 next) (cps r2 next)
  cps (Rep r) next = let self = Split (cps r self) next in self
```

The translation is, for the most part, direct. In the NFA formulation, Cat just corresponds to the composition of two states, and has no special representation. By passing the next parameter directly into the continuations of r1 and r2 in the Alt case, the states unify in the same way that they did when using a let-binding explicitly in the original example. The Rep case is handled by feeding the resulting state back into itself when the argument r succeeds.

2.4 Executing NFAs

To evaluate an NFA, two approaches can be taken: either a breadth-first traversal, with multiple "cursors" that point at the current states, or a depth-first traversal which requires backtracking. In either case, the parser consumes the whole string and returns the number of matches.

```
count :: (a → Bool) → [a] → Int

runBFS :: State → String → Int
runBFS = runCursors · stepEps where
  stepEps :: State → [State]
  stepEps (Split s1 s2) = stepEps s1 ++ stepEps s2
  stepEps s = [s]
```

```
isAccept :: State → Bool
isAccept Accept = True
isAccept _ = False

runCursors :: [State] → String → Int
runCursors sts [ ] = count isAccept sts
runCursors sts (c : cs) =
  let derive (Item c′ st′) | c ≡ c′ = Just st′
      derive _ = Nothing
  in runCursors (mapMaybe derive sts »= stepEps) cs

runDFS :: State → String → Int
runDFS Accept        [ ]      = 1
runDFS (Item c′ st′) (c : cs) | c ≡ c′ = runDFS st′ cs
runDFS (Split s1 s2) cs       = runDFS s1 cs + runDFS s2 cs
runDFS _             _        = 0
```

Both approaches share the idea of parsing with Brzozowski derivatives [3]: this is easiest to see in the derive helper function in runBFS. The idea is that the differentiation of a grammar yields its parser: making progress in a parser must discriminate on the current input head and then run the remainder on the rest.

A key difference between these two approaches is how often they inspect the input string: runBFS reads each character only once, and runDFS looks at each character multiple times as backtracks occur. Yet, due to the space overhead of this particular implementation of runBFS, along with its additional complexity from traversing the state list at each character, it is often not cheaper than the repeated backtracking of the depth-first version. That said, this is not a fundamental limitation of a BFS approach: improvements to such parsers have been explored by Claessen [5]. However, the problem is that transformations like this do not eliminate the intertwining of static and dynamic components of a parser, which prevent staging from being performed: this is discussed in Section 4.2 in more detail. As such, in the remainder of the paper we focus only on the depth-first strategy, as staging is a crucial component in improving the performance of Oregano.

2.5 Moore Machines

As opposed to DFAs and NFAs, Moore machines additionally incorporate the idea that the machine, on entry to a state, writes an item onto a tape. We model Moore machines by adding an Output constructor to the State datatype.

The capability to output can be introduced by adding a new type Output state as well as a new type parameter:

```
data State a = Split (State a) (State a) | Accept | Fail
             | Output a (State a) | Item Char (State a)
```

This definition deviates in three ways from the formal definition of Moore machines. Firstly, traditionally they are untyped abstractions, but in keeping with Haskell's static typing, we lock the emitted values to some type a.

Secondly, traditionally every state outputs a value. That is equivalent to our representation if other states are assumed to output a meaningless value that is filtered out of the final result. Instead, every existing type state can output a meaningful value on entry *when required* by pre-composing it with an Output state.

Thirdly, traditional Moore machine are deterministic. We extend this to Non-deterministic Moore Machines (NMA), where execution can split, fail and make ϵ-transitions.

Summary. The Cayley representation has already been employed to connect the datatype for regular expressions to the datatype of automata. It will be applied many more times throughout this paper, to improve the time-complexity of the result collection—via difference lists—as well as for facilitating tail-recursion and continuation-passing style evaluation of the automata.

The traditional implementation for regular expressions is the NFA. The compilation function demonstrated in this section is the base on which the subsequent iterations of the code is based upon. The continuation-passing style shape that fell out through the natural Cayley transformation of regular expression to state machine is important as it helps expose the control-flow of the automaton, which in turn helps incorporate a parametricity-preserving heterogeneous tape later.

The extension of NFAs to emit results is known as a Moore machine. Moving forward, the Moore machine will form the basis for the final implementation of regular expressions in the paper. The description outlined here will be expanded to support non-deterministic execution, and then made heterogeneous to facilitate the creation of results.

3 Non-Deterministic Moore Machines

This section adapts the existing Reg type to the Moore machine of Section 2.5 and discusses how the latter are executed.

Adapting expressions. To adapt regular expressions as described in Section 2.2 to work with NMAs, the Reg type gets a parameter a for the result type and a new Out constructor:

```
data Reg a = Cat (Reg a) (Reg a) | Alt (Reg a) (Reg a)
           | Eps | Rep (Reg a) | Empt | Out a | Lit Char
```

Compilation now requires an additional case for output:

```
cps (Out x) next = Output x next
```

The interpretation now returns the list of possible output tapes (each represented by a list).

```
runNMA :: State a → String → [[a]]
runNMA Accept        [ ] = [[ ]]
runNMA (Item c′ st′) (c : cs)
  | c ≡ c′                 = runNMA st′ cs
runNMA (Split s1 s2) cs   = runNMA s1 cs
                          + runNMA s2 cs
```

```
runNMA (Output x st') cs = map (x:) (runNMA st' cs)
runNMA _              _  = [ ]
```

For the Output case, the emitted value x is added to the front of all resulting tapes for the continuation st′.

This implementation introduces an inefficiency in the Split case: the (**+**) may have to re-traverse results multiple times. This can be dealt with by making use of the Diff a type defined in Section 2.1.

```
runNMA :: State a → String → [[a]]
runNMA st = fromCayley · go st where
  go :: State a → String → Diff [a]
  go Accept         [ ]      = toCayley [[ ]]
  go (Item c' st')  (c : cs) | c ≡ c' = go st' cs
  go (Split s1 s2)  cs       = go s1 cs • go s2 cs
  go (Output x st') cs       = Cayley (map (x:)) • go st' cs
  go _              _        = mempty
```

By formulating the evaluation in this way, the two calls to go in the Split case are in continuation-passing style (once the Cayley monoid is inlined): this helps by introducing tail-recursion, which will improve performance further.

The Output case does prevent another instance of tail-recursion from falling out of the evaluator here: by building the tape back-to-front, the work to add the new output value must be performed after having finished the rest of the parse. This situation can be ameliorated by carrying a front-to-back construction of the tape, threaded through as an accumulating parameter: essentially a stack to push intermediate results onto. The stack-like behaviour of the threading would require reversing at the end to obtain the queue-like semantics instead. However, by using another difference list, tapes can be constructed in the correct ordering using (•).

```
runNMA :: State a → String → [[a]]
runNMA st cs = fromCayley (go st cs mempty) where
  go :: State a → String → Diff a → Diff [a]
  go Accept         [ ]      t = toCayley [fromCayley t]
  go (Item c' st')  (c : cs) t
    | c ≡ c'                   = go st' cs t
  go (Split s1 s2)  cs       t = go s1 cs t • go s2 cs t
  go (Output x st') cs       t = go st' cs (diffCons x t)
  go _              _        _ = mempty
```

The go function now takes the partial tape (of type Diff a) built up to this point. This way the Output case can add to the accumulator. The difference list structure is collapsed into a list in the Accept case. All the calls to go are now in tail-call positions (after inlining of the Cayley monoid), which makes this evaluator equivalent to a loop: this is to be expected for regular expressions, which are an inherently iterative (and not recursive) abstraction.

3.1 Eliminating Lists

The [a] resulting tapes we have introduced will often be consumed to produce a single value of some other type b. In these instances, generating the intermediate list is inefficient. By Church-encoding the tapes instead, the lists can be deforested, producing an intermediate that is folded into the final result "in-place": this is known as *fold-build* fusion. To Church-encode a structure f, each constructor must be represented by function where every reference to the type f is replaced by the result type b. Instead of a difference list for the tape, the monoid on endomorphisms is used (not the Cayley-variant, though, since b is not necessarily a monoid): this preserves the same behaviour as the difference list.

```
type Endo a = a → a
runNMA :: (a → b → b) → b → State a → String → [b]
runNMA cons nil st cs = fromCayley (go st cs mempty)
  where
    go :: State a → String → Endo b → Diff b
    go Accept         [ ]      t = toCayley [t nil]
    go (Item c' st')  (c : cs) t | c ≡ c' = go st' cs t
    go (Split s1 s2)  cs       t = go s1 cs t • go s2 cs t
    go (Output x st') cs       t = go st' cs (cons x • t)
    go _              _        _ = mempty
```

The transformation is simple: introduce the two "constructors" nil and cons, and then replace the concrete references to [ ] and (:) with each, respectively. By doing this, the results of the parser are more similar to what we might expect out of an applicative regular expression library.

Yet, there is still a disconnect between the advertised result type a of State and the final result type b. This comes with inflexiblity: different parts of the parser cannot return different results, they must be combined using the Church-encoded cons function. In the original email example, the type does vary, since the final result should be an Email, with intermediate types of String and [String]. These are combined in a tree-like way, as opposed to sequenced. So, despite the advantage that the encoding brings in terms of efficiency, it is necessary to take a slightly different approach. The deforestation of the tape can still be done using staging, however, and this is explored in Section 4.4.

3.2 Heterogeneous Moore Machines

To allow the types to vary at different levels of the regex, the result tape needs to be heterogeneous. As before, the tape threaded through the evaluation can take a stack-like semantics: this time, however, the stack structure is preserved explicitly, to allow elements on the stack to be reduced in a principled, and type-preserving way. For this to work, the tape must be more structured than before, encoding not only the types of the values placed onto it, but the operations that should be performed on them to reduce to a final value.

```
data Op (ins :: [∗]) (outs :: [∗]) where
  Push :: x → Op ins (x : ins)
  Red  :: (x → y → z) → Op (y : x : ins) (z : ins)
```

The Op type captures this idea. Assuming that the remaining tape models a stack with elements of type ins (which is a list of types), Push adapts this stack so that it has an extra x at the front, and Red combines the top two elements. The Push and Red operations will be interleaved in a way that respects the original reduction order for the regex.

Now, the old [a] type of the tape (without the difference list) needs to be one that can safely relate Ops to each other:

```
data Tape (xs :: [∗]) where
  Empty :: Tape [ ]
  Cell  :: Op ins outs → Tape ins → Tape outs
```

Here, each tape should start with Empty, which has no elements in it. Then, each Cell on the tape will have an Op which relates the old tape of ins to a new one of outs. This way of preserving the type-safety of in-flight partial results as been previously explored, by either encoding as a stack [32] or a continuation driven system Hughes and Swierstra [9]. Here, by encoding these results in the slightly more heavy-weight Tape datatype, a closeness to the underlying abstraction of the Moore machine is maintained.

A cell with a Red operation contains a function to combine elements on the stack with. This function needs to be provided as part of the concatenation of regular expressions: sequencing a Reg a with a Reg b will produce a Reg c, so long as an a → b → c function is provided. To illustrate this more concretely, here is the new type of Reg, which now allows the types of the sub-expressions to vary:

```
data Reg a where
  Lit  :: Char → Reg Char
  Cat  :: (a → b → c) → Reg a → Reg b → Reg c
  Alt  :: Reg a → Reg a → Reg a
  Eps  :: a → Reg a
  Rep  :: Reg a → Reg [a]
  Empt :: Reg a
```

The effect of pushing the combination of results into the regular expression language is evident here: not only has Cat received a function to combine sub-expressions with, Lit now returns the character it matches, and Eps replaces the Out node from before (Eps :: Reg () is possible, but this a then just a special case of Out (), which also consumes no input and just succeeds: it is sensible to merge them together). Repetition of a regex, via Rep, now returns multiple results as a list too. This is now much more similar to the design of a traditional applicative-style library, where Cat corresponds to liftA2, Eps to pure, and Rep to many. Indeed, this is the interface chosen by many classic implementations of parser combinators [10, 17, 27]; importantly though this deep-embedded representation allows for the inspection of the parser, which has been leveraged before in the literature [5, 31, 32].

The type of the State changes too. Its type has to reflect the abstract stack being expressed by the tape of Ops:

```
data State (xs :: [∗]) res where
  Accept :: State (res : [ ]) res
  Fail   :: State xs res
  Item   :: Char → State xs res → State xs res
  Output :: Op ins outs → State outs res → State ins res
  Split  :: State xs res → State xs res → State xs res
```

Here State xs res describes the state of an automaton that produces a tape with xs on it, which can eventually be collapsed into the type res. The Accept node enforces this, by demanding that, on entry, the tape represents a stack with exactly one element: the result of the parse. The other interesting case is Output, which now takes an Op that adapts the input tape into an output tape.

The conversion between the new Reg a and State [ ] a (all tapes must start empty), is slightly more involved than it was previously, owing to the need to carefully output the right ops for each of the regular expression primitives:

```
compile :: Reg a → State [ ] a
compile reg = cps reg Accept where
  cps :: Reg a → State (a : xs) r → State xs r
  cps (Eps x)       k = Output (Push x) k
  cps (Lit c)       k = Item c (Output (Push c) k)
  cps (Cat f r1 r2) k = cps r1 (cps r2 (Output (Red f) k))
  cps (Alt r1 r2)   k = Split (cps r1 k) (cps r2 k)
  cps (Rep r)       k = Output (Push mempty) self where
    self = Split (cps r (Output (Red (flip diffCons)) self))
                 (Output (Push fromCayley)
                         (Output (Red (flip ($))) k))
```

The type of cps is more interesting in this version: it demonstrates that each complete regular sub-expression will end up placing its results onto the front of the stack, with the stack serving as a place to store all the previous results, before they are reduced down on completion of a sub-expression. The Eps case now produces the Push operation to place the correct result onto the stack (this is similar for Lit); the Cat case, which used to just stitch two states together in sequence, now has to ensure that a Red operation is emitted to the tape after both sub-expressions have pushed their results onto the stack; the Rep rule has also changed here, this time constructing a difference-list on the stack, since the results are found on the stack in the reverse order that they should be produced in the results – it is cheaper to use a difference list here than neededlessly reverse the results after. When the Rep node is ready to exit, it pushes and applies the fromCayley function to obtain a regular list.

The parametricity of the cps transform here provides a light-weight guarantee that the conversion was faithful to

the original representation [19, 32] – if it were not, the states of the automaton would not slot together properly!

The execution of this new State structure is not too different from how it was before, however, the difference-list structure for the tape is no longer required: the stack will be collapsed in a way that is more consistent with the order in which things were added.

```
runNMA :: State [ ] a → String → [ a ]
runNMA st cs = fromCayley (go st cs Empty) where
  go :: State xs a → String → Tape xs → Diff a
  go Accept          [ ]      t  = toCayley [ top t ]
  go (Item c′ st′)   (c : cs) t  | c ≡ c′ = go st′ cs t
  go (Split s1 s2)   cs       t  = go s1 cs t • go s2 cs t
  go (Output op st′) cs       t  = go st′ cs (Cell op t)
  go _               _        _  = mempty
```

This is very similar to the definition from before, except a Tape with the same type as *each* intermediate state is provided at each step in the execution. When an Accept state is reached, this entire tape structure can be collapsed into a stack and the top value returned: this will be the result type a. The collapsing is done by executing the stack operations on a real, heterogeneous, stack:

```
data Stack (xs :: [∗]) where
  Nil  :: Stack [ ]
  Cons :: x → Stack xs → Stack (x : xs)
top :: Tape (x : xs) → x
top tape = let Cons x _ = toStack tape in x where
  toStack :: Tape xs → Stack xs
  toStack Empty = Nil
  toStack (Cell (Push x) tape) = Cons x (toStack tape)
  toStack (Cell (Red f) tape) =
    let Cons y (Cons x stack) = toStack tape
    in  Cons (f x y) stack
```

The flat tape structure is first converted into a stack of the right shape: this is done by interpreting the Push operation as a Cons on a stack, and Red f by de-constructing two Cons nodes, applying their contents with f, and then generating a new Cons node with the result. Since top is called within go Accept, we know that the given tape is Tape (a : [ ]) where a matches the result type of the regex: the final stack will have exactly one value (confirmed by the types themselves), and this can be matched against at the end of top, though it works more generally on stacks of larger size.

The conversion between Tape and Stack might seem redundant: why not just feed the stack through the machine directly and avoid the conversion at the end? As hinted earlier, by using Tape, the representation of the Moore machine remains faithful to the original description.

Summary. Non-deterministic Moore Machines are a generalisation of the Moore state machine that allows for multiple tapes to be produced during the execution of the machine. The values stores on this tape are traditionally untyped, but by providing stronger types to them, they can be used to encode a stack of values produced by an applicative regular expression parser, allowing for the fully-structured results expected by parser combinator libraries.

While the abstractions introduced so incur some overhead, this is of little concern, as the next section demonstrates how staged meta-programming can be used to eliminate all of the structure and computation that is known, or can be executed, at compile-time. It is for this reason that the heavy abstraction of the Tape datatype does not pose a problem for performance moving forward: it, in particular, can be entirely eliminated at compile-time.

4 Staging to remove Overhead

The approach so far introduces a lot of interpretive overhead: the regular expression must be converted to the automaton, and then this has to be traversed and executed, including building and tearing down a stack.

Fortunately, this can be resolved with *Staged Metaprogramming* [25, 26] (or *staging*, for short). This technique promotes code fragments to be first-class values that can be manipulated by programs. It allows the programmer to encode domain-specific knowledge and optimisations into a form that the compiler can actually use during program compilation. This can produce results that are otherwise unobtainable relying on domain-inspecific optimisations that general purpose language compilers rely on.

In particular, we use staging not only to translate from regular expression to automaton at compile time, but also to eliminate the interpretive overhead of executing the automaton, generating efficient code in its place. This is possible, since the representation of the automaton is finite and thus can be generated into finite amount of code.

In fact, we take this technique one step further, and even use the static structures of the lambda calculus and character classes to improve the generated code. Additionally, we exploit the static structure of church-encoded constructors to generalise to multiple different strategies for producing results with zero-cost abstraction.

4.1 Setting the Stage

Staging introduces a new type Code a for abstract syntax trees of expressions of type a. Two operators are provided: the quote of e :: a, denoted ⟦e⟧ :: Code a is the abstract syntax tree representing the expression e; the splice of qe :: Code a, denoted $qe :: a incorporates a syntax tree within qe either into another quote, or evaluates the code and inserts it into the top-level program. Consider the exactly combinator:

```
(<:>) :: Reg a → Reg [a] → Reg [a]
rx <:> rxs = (:) <$> r <*> rxs

exactly :: Int → Reg a → Reg [a]
exactly 0 _ = pure [ ]
exactly n r = r <:> exactly (n − 1) r
```

The behaviour of exactly n r is to match the regex r exactly n times, much like the RegEx: $r\{n\}$. This combinator can be specialised to provide a specific combinator for a set n:

```
thrice :: Reg a → Reg [a]
thrice = exactly 3
```

For the sake of example, assume that Reg is executed directly, without compiling it to an automaton. At runtime, the recursion will have to be unwound, which is unnecessary overhead. GHC will not help, as is unwilling to inline recursive functions. Instead, the programmer may manually specialise the thrice combinator:

```
thriceFast :: Reg a → Reg [a]
thriceFast r = r <:> (r <:> (r <:> pure [ ]))
```

Staging avoids this manual labour, leaving the statically known integer as is, but turning the rest into Code:

```
exactly′ :: Int → Code (Reg a → Reg [a])
exactly′ 0 = ⟦λ_ → pure [ ]⟧
exactly′ n = ⟦λr → r <:> $(exactly (n − 1)) r⟧
```

To demonstrate the effect, here is what thrice would look like in the generated code:

```
thrice′ :: Reg a → Reg [a]
thrice′ = $(exactly′ 3)
        = λr1 → r1 <:> (λr2 → r2 <:> (λr3 → r3 <:>
            (λ_ → pure [ ]) r3)r2) r1
```

This expression, while free of recursion, still has a lot of redundant, easily reducible functions. GHC might optimise these, but static knowledge about the code structure can be leveraged to eliminate this without GHC's intervention. The structure in question is the fact that when a function and its argument are known, we know where that argument would be substituted into the function body. As a result, Code (a → b) can be transformed into Code a → Code b, which shifts the structure of the function to compile-time:

```
exactly′′ :: Int → Code (Reg a) → Code (Reg [a])
exactly′′ 0 qr = ⟦pure [ ]⟧
exactly′′ n qr = ⟦$qr <:> $(exactly′′ n qr)⟧
```

Here, the application of the value qr to the recursive call is performed inside the splice and not immediately in the quote. This has the following effect on the definition of thrice:

```
thrice′′ :: Reg a → Reg [a]
thrice′′ r = $(exactly′′ 3 ⟦r⟧) = r <:> (r <:> (r <:> pure [ ]))
```

This is exactly the same as thriceFast! The benefit is that this generates specialised exactly combinators for any size n. For this example, this is about as far as the specialisation can go.

A note about CPS. Any static information used to inform the code generation must be threaded through the generator in a top-down fashion. The information cannot be passed from deeper in the code generation upwards, because of the order in which the code must be generated. Consel and Danvy [6] show that continuation-passing style brings static values closer together so that they can be optimized statically. It is for this reason, that we have taken so much care to create a representation of automata that is a continuation-passing style datatype.

4.2 Identifying Static Information

The first step of staging is to determine what the static information is that will be exploited: the regular expression.

There are two pieces of dynamic information: the first is the string that should be applied to the regular expression matcher; the second is the means by which the user wishes to combine and produce results. The latter seem like they should be static information (as they form part of the static expression), but since their use as static information is limited for our approach, they will be left as fully dynamic values.

In Section 2.4, it was mentioned that the breadth-first approach to evaluating the automata is not suitable: this is because a breadth-first traversal must keep a list of in-flight states, but this nests statically known states inside a dynamic structure – the numbers of in-flight states, and specifically which, are dependent on the dynamic input. When static data is held inside dynamic data, it is no longer accessible, and as such the staging cannot progress in the desired way. With the depth-first approach, however, only the future paths to be taken are dynamic, and this is separated from the states that generate them, bringing static information close to the surface, and pushing dynamic information into the branches.

Section 3 leveraged Haskell's recursive let bindings to represent self-edges in finite automata. This works fine for interpretation, however for staging the let-bindings exist only at compile-time, and do not automatically mirror themselves in the generated code. Instead, we need an explicit representation of named nodes to operate on and to map to Haskell's recursive let structures in the generated code. To support that, we extend the the State type with the MkJoin and Join constructors. They relate in the following way:

```
old = let      x = (. . (      x) . .) in (. . (      x) . .)
ϕ   = Φ 1
new = MkJoin ϕ   (. . (Join ϕ) . .)     (. . (Join ϕ) . .)
```

A MkJoin node sets up a binding to a given name, with a given body, and an expression for which this binding is in scope; a Join node, on the other hand, references this binding.

```
data Reg a where
  Lit   :: Char → Reg Char
  Cat   :: Code (a → b → c) → Reg a → Reg b → Reg c
  Alt   :: Reg a → Reg a → Reg a
  Eps   :: Code a → Reg a
  Rep   :: Reg a → Reg [a]
  Empt :: Reg a
data Op (ins :: [∗]) (outs :: [∗]) where
  Push    :: Code x → Op ins (x : ins)
  Red     :: Code (x → y → z) → Op (y : x : ins) (z : ins)
data Tape (xs :: [∗]) where
  Empty :: Tape [ ]
  Cell  :: Op ins outs → Tape ins → Tape outs
newtype Φ x = Φ Word
data State (xs :: [∗]) res where
  Accept  :: State (res : [ ]) res
  Fail    :: State xs res
  Item    :: Char → State xs res → State xs res
  Output  :: Op ins outs → State outs res → State ins res
  Split   :: State xs res → State xs res → State xs res
  MkJoin  :: Φ x → State (x : xs) res
          → State xs res → State xs rs
  Join    :: Φ x → State (x : xs) res
```

Figure 1. Final definitions of Reg, State, Tape, and Op

Each binding has a unique Φ node associated with it: Φ 1, Φ 2, and so on.

With this all in mind, Figure 1 shows the final definitions for Reg, State, Tape, and Op. The adjustments to the compile function are minimal, except that implicit Haskell bindings are replaced by explicit MkJoin.

4.3 Performing Staging on the Automata

The compilation from Reg to State is transforming static data into static data; staging is not required. Dynamic information and staging first appear during the evaluation of an automaton. This will be the focus of the rest of this section.

Like for exactly, the type of the runNMA function changes to reflect the dynamic nature of the input and the results:

```
runNMA :: State [ ] a → Code String → Code [a]
```

The evaluation of MkJoin and Join introduce a new problem, however. At compile-time, a map from the bindings to their code (i.e., the names of Haskell bindings in the generated code) is required. This can be accomplished using a dependent map (DMap), whose key and value types can vary between different key-value pairs[2]:

```
(!) :: GOrd k ⇒ DMap k v → k a → v a
insert :: GOrd k ⇒ k a → v a → DMap k v → DMap k v
empty :: DMap k v
```

Here, the GOrd typeclass generalises orderings to work on values where the type indexes of each value may vary. In our case, the concrete map required has the following type:

```
newtype Binding a x =
  Binding (Code (String → x → Diff a))
type Ctx a = DMap Φ (Binding a)
```

This maps a join-point to the code that represents that let-binding, which will be a function that takes the input string, the value fed into the join-point, and then returns the difference list of results that this join point will contribute.

```
runNMA :: State [ ] a → Code String → Code [a]
runNMA st = fromCayley · go st empty Empty where
  go :: State xs a → Ctx a → Tape xs
     → Code String → Code (Diff a)
  go Accept _ t qcs =
    ⟦ case $qcs of [ ] → toCayley [$(top t)]
                   _   → mempty⟧
  go (Item c′ st′) ctx t qcs =
    ⟦ case $qcs of c : cs | c ≡ c′ → $(go st′ ctx t ⟦cs⟧)
                   _                → mempty⟧
  go (Split s1 s2) ctx t qcs =
    ⟦$(go s1 ctx t qcs) • $(go s2 ctx t qcs)⟧
  go (Output x st′) ctx t qcs = go st′ ctx (Cell op t) qcs
  go (Join ϕ) ctx t qcs =
    let Binding qjoin = ctx ! ϕ in ⟦$qjoin $qcs $(top t)⟧
  go (MkJoin ϕ bndr recv) ctx t qcs =
    ⟦ let join cs x =
            $(go bndr (ctx′ ⟦join⟧) (Cell (Push ⟦x⟧) t) ⟦cs⟧)
      in $(go recv (ctx′ ⟦join⟧) t qcs)⟧
    where ctx′ qjoin = insert ϕ (Binding qjoin) ctx
  go Fail _ _ _ = ⟦mempty⟧
```

The staging of runNMA has altered its structure somewhat: the fall-through cases for Item and Accept cannot be used, as the check on the input takes place dynamically, whereas the pattern match on the State happens statically. Otherwise, the code is similar, but with quotes and splices added into the right places. The introduction of join points, however means that the map of bindings ctx must be threaded through the go function, and bindings are added to both recursive calls in the MkJoin case, and bindings are summoned from the map in the Join case.

Staging Cayley. As in the "exactly" example, the inherently static structure of functions was exploited to ensure that the dynamic input is threaded straight to the use-sites in the generated code without requiring quoted function. The

[2] https://hackage.haskell.org/package/dependent-map-0.4.0.0

difference list that collects results shares the same static structure. It can thus be split to avoid constructing unnecessary lambdas at runtime. More concretely, any Code (Cayley m) can be tranformed statically into Code m → Code m. The effect of the transformation is to introduce an extra argument to both go and Binding, which represents the results collected during the parse. By applying this transformation, the code for runNMA changes:

```
newtype Binding a x =
  Binding (Code (String → x → [a] → [a]))
runNMA :: State [] a → Code String → Code [a]
runNMA st cs = go st empty Empty cs [|[]|] where
  go :: State xs a → Ctx a → Tape xs → Code String
     → Code [a] → Code [a]
  go Accept _ t qcs res =
    [| case $qcs of [] → $(top t) : $res
                    _  → $res |]
  go (Item c' st') ctx t qcs res =
    [| case $qcs of
         c : cs | c ≡ c' → $(go st' ctx t [|cs|] res)
         _               → $res |]
  go (Split s1 s2) ctx t qcs res =
    go s1 ctx t qcs (go s2 ctx t qcs res)
  go (Output x st') ctx t qcs res =
    go st' ctx (Cell op t) qcs res
  go (Join φ) ctx t qcs res =
    let Binding qjoin = ctx ! φ
    in [|$qjoin $qcs $(top t) $res |]
  go (MkJoin φ bndr recv) ctx t qcs res =
    [| let join cs x res = $(go bndr (ctx' [|join|])
                                     (Cell (Push [|x|]) t)
                                     [|cs|] [|res|])
       in $(go recv (ctx' [|join|]) t qcs res) |]
    where ctx' qjoin = insert φ (Binding qjoin) ctx
  go Fail _ _ _ res = res
```

By breaking up the structure statically, the Split operation no longer has any quotes in its interpretation: it is purely static, with no runtime overhead. This does not compromise the performance benefits of the difference list: its structure is still there, but the static component has been extracted. This improvement to the Split case could not have been achieved without the continuation-passing style of the automaton: the CPS shape of the go function ensures that the code for the continuation of the automaton is generated directly into the position where it is required, eliminating any left-association in the result collection.

4.4 Generalising the Results

In Section 3.1, fold-build fusion was performed on the result tapes to process them into some other value without constructing a list. A similar principle can be performed on the current program, where the final result of type Code [a] can be deforested into a value of type Code b as long as the church-encoded cons and nil functions are provided. In fact, staging and fusion (in this case fold fusion) can be combined as *staged fusion*: by exploiting the static structure of church-encoded constructors, values otherwise discarded by laziness in a dynamic computation are discarded at compile-time, and their code is never generated.

In this case, there are several useful alternative interpretations that can be used as substitutes for lists. In particular, one might wish to return only the first result using Maybe a, whether any match can occur using Bool, or the number of full matches using Int. To achieve this fold-build fusion, it suffices to pass in Code (a → b → b) and Code b into the runNMA function, to replace (:) and [] respectively. However, once again the static structure of functions can be leveraged and instead Code a → Code b → Code b can be used. The code is modified as follows:

```
runNMA :: (Code a → Code b → Code b) → Code b
       → State [] a → Code String → Code b
runNMA qcons qnil st cs = go st empty Empty cs qnil
  where
    go :: State xs a → Ctx a → Tape xs → Code String
       → Code b → Code b
    go Accept _ t qcs res =
      [| case $qcs of [] → $(qcons (top t) res)
                      _  → $res|]
    go (Item c' st') ctx t qcs res = [| case $qcs of
      c : cs | c ≡ c' → $(go st' ctx t [|cs|] res)
      _               → $res|]
    ...
```

Now the alternative interpretations can be given:

```
firstMatch :: State [] a → Code String
           → Code (Maybe a)
firstMatch = runNMA (λx _ → [|Just $x|]) [|Nothing|]

anyMatch :: State [] a → Code String → Code Bool
anyMatch = runNMA (λ_ _ → [|True|]) [|False|]

numMatches :: State [] a → Code String → Code Int
numMatches = runNMA (λ_ n → [|$n + 1|]) [|0|]
```

In each of these cases, one or more of the arguments to the church-encoded cons function is ignored. In a regular fold, this would result in the lazy value not being computed and thrown away. When staging, however, this actually results in the code not even being generated to begin with: in other words, when a match succeeds, the first two implementations

will not even generate the alternative parses from that point, and the latter two will not generate any results using the parse. This improves the generated code, and therefore the efficiency of the parsers when dealing with these use-cases.

4.5 Character Classes

The use of individual characters for matching results in an unnecessary amount of generated code, even if join-points are used to share common code. This encourages GHC to stop optimising the code, and harms cache coherency.

A better approach is to use character classes, where a set of characters to be matched is provided [21]. The Class state unlike Item, which it replaces, must output the matched character immediately onto the tape, as the specific character will be statically unknown:

```
Item  :: Char      → State xs res          → State xs res
Class :: Set Char → State (Char : xs) res → State xs res
```

It is possible to generate an efficient function for matching any character within the class: given a statically known Set Char, it is possible to stage the membership of that set for some dynamic character:

```
member :: Set Char → Code Char → Code Bool
```

The structure of a binary search tree offers a $O(\log n)$ function for testing membership in that tree, but staging removes much of the overhead this entails, including pattern matching and recursion. We omit the details of this staged membership function for brevity. However, the go function within runNMA changes to accomodate Class instead of Item:

```
go (Class set st') ctx tape qcs res = ⟦ case $qcs of
  c : cs | $(member set ⟦c⟧) →
    $(go st' ctx (Cell (Push ⟦c⟧) tape) ⟦cs⟧ res)
  _ → $res⟧
```

This simple change does take the abstraction slightly further from a traditional Moore machine, but it is easy to show that the two representations are equivalent. This does have a significant impact on the generated code, however: in the email example (Section 1), the wordOrMinusOrPlusOrDot rule requires at worst 5 comparisons with an optimised binary-search for Class, or 66 case and let expressions with Item!

This is the final change that is made to the runNMA function, which now generates high-quality, fast, code. GHC will likely[3] optimise away any remaining minor imperfections like performing β-reduction of terms. Yet, instead we accomplish this during code generation by modelling expression reduction as a Higher-Order Abstract Syntax (HOAS) [22] representation of the Lambda Calculus. This, partially-static, datastructure's evaluation can be staged to easily optimise and normalise any expressions within the generated code: whilst omitted from the paper, it is employed in the implementation. This encourages GHC to focus on optimising parts of the generated code that cannot be optimised easily using staging.

Summary. By leveraging the static nature of a regular expressions description, it is relatively straightforward to eliminate the associated overheads of translating our regular expression representation to automata as well as those of executing the automata itself.

The complex representation of the stack as a tape of stack operations has no effect on the generated code, since the shape of the tapes at each point in the execution is fully statically known. Instead, values are reduced at compile-time and inserted directly into the the generated code.

A lot of emphasis is put on the static structure of functions themselves: this enabled numerous optimisations, such as lifting the Cayley representations to compile-time, and eliminating entire branches of the code when switching to alternative evaluation strategies like anyMatch, which just returns a Bool: this technique is called *staged fusion*.

By leveraging character classes, and representing terms in the generated code as a partially-static lambda calculus, further improvements can be achieved, that both reduces terms to a normal form, and simplifies the number of branches required to match on specific characters: this exploits the structure of the membership test of a statically known set.

5 Benchmarks

This section presents a few benchmarks to compare the performance of Oregano against that of common Haskell alternatives [15, 16, 20]. In addition, we have tested Oregano against the `pcre` C library. For these tests, the anyMatch semantics is used, returning a Bool.

We have also tested against `parsley` [30, 32], as it uses a similar staging approach but as a full parser combinator library capable of recursive-descent as well as context-sensitive parsing. Unlike the alternative regex libraries, it is able to build structured results. For these tests, the firstMatch semantics is used, returning Just a result, or Nothing. We have omitted the regex – applicative library because it is many times slower than `parsley`.

All libraries are tested against the email example from Section 1. Then `parsley` and Oregano are also tested on a regex for matching IPV4 addresses. This is a backtracking heavy parser, so it allows for a comparison of backtracking overheads in both libraries. All benchmarking code is included in th e supplementary material.

Figure 2 shows the relative performance of other Haskell and C libraries against Oregano. The test checks whether a given email address is valid. Oregano is very competitive, with a solid 5× lead on the next fastest solution. This is likely because Oregano uses staging whereas the others interpret the automaton at runtime. In Figure 3, where we check how

[3] As code size increases, GHC can become less willing to continue optimising.

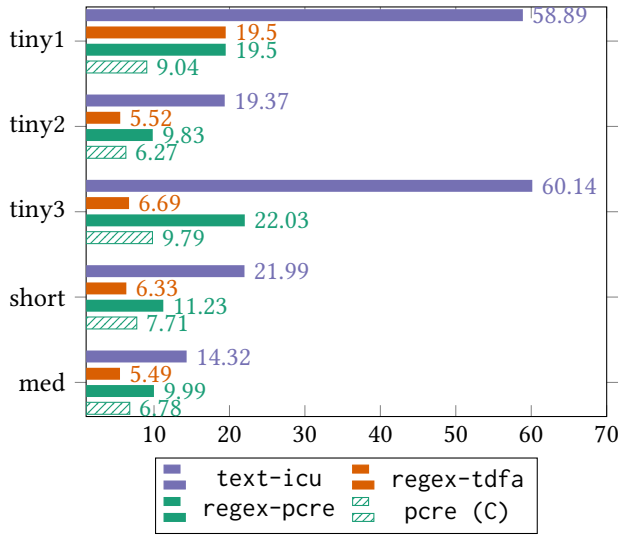

Figure 2. Performance of libraries matching emails, time relative to Oregano

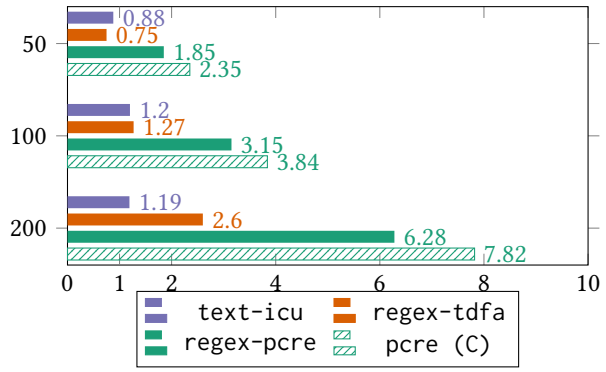

Figure 3. Performance of Oregano counting emails anywhere in a larger text, time relative to other libraries

many addresses are contained in a larger text, Oregano incurs a slow down that is linear in the size of the document. The reason is the heavy backtracking of the depth-first search for this highly nondeterministic regex.[4]

Figure 4 shows the same workload as Figure 2, but returning an Email datatype. Here, both Oregano and `parsley` are tested. As a staged selective parser combinator library, `parsley` will generate code of a similar quality to our own. This is evident here, with not much difference between the two at the maximum optimisation level. It is interesting to note that `parsley` seems to optimise better, with more improvement at higher optimisation levels; it is possible Oregano generates more concise code to begin with (owing to its simpler model), but the underlying machinery of `parsley` seems to be favoured more by GHC's optimiser.

[4]Though not relevant for this benchmark, Oregano counts the number of ambiguous matches whereas the others cannot.

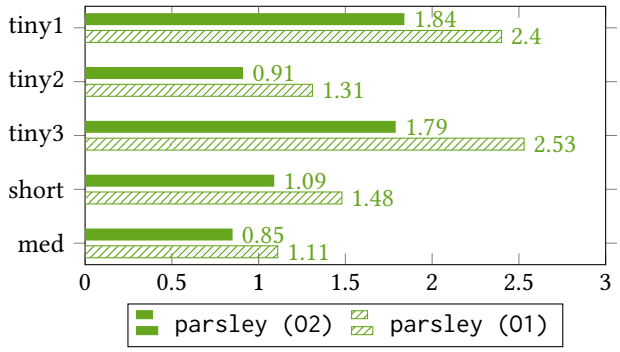

Figure 4. Performance of libraries matching emails into datatype, time relative to Oregano

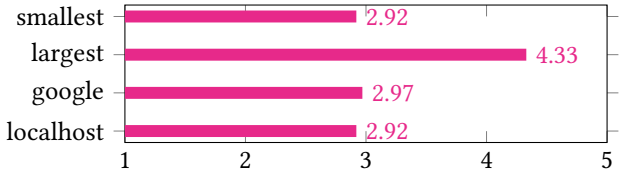

Figure 5. Performance of `parsley` matching IPV4 addresses into datatype, time relative to Oregano

As a final example, Figure 5 shows the results of comparing on a parser for IPV4. Whilst these inputs are naturally small, the parser was written in a backtrack heavy way, and this highlights that the non-deterministic backtracking semantics of Oregano seem to be more effective than the mechanisms employed by `parsley` with around a 3× improvement.

6 Related Work

Monadic and selective parser combinator libraries like `parsec` [17] and `parsley` [32] offer a similar interface to Oregano. By supporting selective and monadic combinators, as well as general recursion, they can parse both context-free and context-sensitive grammars. By restricting Oregano to applicatives, and only supporting iteration, we have only enough power to parse regular expressions, but the structure is fully static and easier to inspect. Additional optimisations could be applied on the high-level regular expression datatype exploiting this.

CPS improvements for staged parsers have been explored in the context of creating a well-typed staged library for parsing μ-regular expressions [14]. They support recursion in their parsers, which enables more power than regular expressions alone. The typing they employ helps to ensure that their parsers are productive when recursing. This is less of a problem for regular expressions, however their techniques would prevent infinite iteration in the Kleene-$\star$ operation. They use a direct-style interpretation instead of translation through an automaton.

Staging parser combinators has also been explored by both Willis et al. [32] and Jonnalagedda et al. [12]. Both achieve competitive performance in their respective languages, and eliminate the overheads of building intermediate structures, like Oregano does. In particular Jonnalagedda et al. [12] performs runtime code generation, which – unlike our purely static approach – is closer to the way in which regular expressions are usually compiled on-the-fly before they are used in other libraries. The optimisations performed by these libraries are more domain-specific than ours, which really focuses on the natural exploitation of static structure, like Krishnaswami and Yallop [14], and less on optimising the high-level representations. By leveraging these techniques as well, our work could be improved further.

Our work has focused on the implementation of regular languages via automata. Another approach is to derive the grammar [1, 3, 7]. This also produces a continuation-passing style evaluation, which should be amenable to staging, however it does introduce an interleaving of dynamic and static information, which makes static exploitation more difficult.

7 Conclusion

We presented a simple encoding of regular expressions and its translation to non-deterministic finite automata. By adding outputting nodes that write to a tape, these automata become non-deterministic Moore machines that produce results of some given type. As homogeneous types are not general enough for a fully applicative library, we added a stack that is reduced after the parse to distil the results within. As the naïve implementation of the heterogeneous NMA incurs heavy interpretive overheads, we turned to staging.

The Cayley representation was instrumental throughout. It not only improved list concatenation, but also enabled the continuation-passing style that facilitates the threading of static data during staging, maximising the potential gains.

The benchmarks show that Oregano competes with other regex libraries, and with the `parsley` staged parser library, but with a simpler underlying implementation.

Acknowledgements

We would like to thank the reviewers for their constructive remarks. This work was supported by EPSRC grant number EP/S028129/1 on 'Scoped Contextual Operations and Effects', by FWO project G095917N, and KU Leuven project C14/20/079.

References

[1] Michael D. Adams, Celeste Hollenbeck, and Matthew Might. 2016. On the Complexity and Performance of Parsing with Derivatives. *SIGPLAN Not.* 51, 6 (2016), 224–236. https://doi.org/10.1145/2908080.2908128

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249

[4] A. Cayley. 1854. VII. On the theory of groups, as depending on the symbolic equation θn =1. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 7, 42 (1854), 40–47. https://doi.org/10.1080/14786445408647421

[5] Koen Claessen. 2004. Parallel Parsing Processes. *J. Funct. Program.* 14, 6 (nov 2004), 741–757. https://doi.org/10.1017/S0956796804005192

[6] Charles Consel and Olivier Danvy. 1991. For a Better Support of Static Data Flow. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, 496–519. https://doi.org/10.5555/127960.128060

[7] Ian Henriksen, Gianfranco Bilardi, and Keshav Pingali. 2019. Derivative Grammars: A Symbolic Approach to Parsing with Derivatives. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 127 (2019), 28 pages. https://doi.org/10.1145/3360553

[8] J. Hughes. 1986. A novel representation of lists and its application to the function "reverse". *Inform. Process. Lett.* 22, 3 (1986), 141–144. https://doi.org/10.1016/0020-0190(86)90059-1

[9] J. Hughes and S. Doaitse Swierstra. 2003. Polish Parsers, Step by Step. *SIGPLAN Not.* 38, 9 (aug 2003), 239–248. https://doi.org/10.1145/944746.944727

[10] Graham Hutton. 1992. Higher-order functions for parsing. *J. Funct. Program.* 2, 3 (1992), 323–343. https://doi.org/10.1017/S0956796800000411

[11] Graham Hutton and Erik Meijer. 1996. Monadic Parser Combinators.

[12] Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. *SIGPLAN Not.* 49, 10 (2014), 637–653. https://doi.org/10.1145/2714064.2660241

[13] Stephen C. Kleene et al. 1956. Representation of events in nerve nets and finite automata. *Automata studies* 34 (1956), 3–41.

[14] Neelakantan R. Krishnaswami and Jeremy Yallop. 2019. A Typed, Algebraic Approach to Parsing. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 379–393. https://doi.org/10.1145/3314221.3314625

[15] Christopher Kuklewicz and Andreas Abel. 2008. `regex-tdfa`: Pure Haskell Tagged DFA Backend for "Text.Regex" (`regex-base`). https://hackage.haskell.org/package/regex-tdfa.

[16] Christopher Kuklewicz, Herbert Valerio Riedel, and Andreas Abel. 2007. `regex-pcre`: PCRE Backend for "Text.Regex" (`regex-base`). https://hackage.haskell.org/package/regex-pcre.

[17] Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators For The Real World.* Technical Report. Microsoft.

[18] Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jeremie Dimino. 2019. Selective Applicative Functors. *Proc. ACM Program. Lang.* 3, ICFP, Article 90 (2019), 29 pages. https://doi.org/10.1145/3341694

[19] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. 2002. Stack-Based Typed Assembly Language. *J. Funct. Program.* 12, 1 (2002), 43–88. https://doi.org/10.1017/S0956796801004178

[20] Bryan O'Sullivan and Vladimir Shabanov. 2009. `text-icu`: Bindings to the ICU library. https://hackage.haskell.org/package/text-icu.

[21] Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-Expression Derivatives Re-Examined. *J. Funct. Program.* 19, 2 (2009), 173–190. https://doi.org/10.1017/S0956796808007090

[22] F. Pfenning and C. Elliott. 1988. Higher-Order Abstract Syntax. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 199–208. https://doi.org/10.1145/53990.54010

[23] M. O. Rabin and D. Scott. 1959. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development* 3, 2 (1959), 114–125. https://doi.org/10.1147/rd.32.0114

[24] Exequiel Rivas and Mauro Jaskelioff. 2017. Notions of computation as monoids. *J. Funct. Program.* 27 (2017), e21. https://doi.org/10.1017/S0956796817000132

[25] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *SIGPLAN Not.* 46, 2 (2010), 127–136. https://doi.org/10.1145/1942788.1868314

[26] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. *SIGPLAN Not.* 37, 12 (2002), 60–75. https://doi.org/10.1145/636517.636528

[27] Doaitse Swierstra. 2008. *Combinator Parsing: A Short Tutorial.* Technical Report UU-CS-2008-044. Department of Information and Computing Sciences, Utrecht University.

[28] S. Doaitse Swierstra and Luc Duponcheel. 1996. Deterministic, Error-Correcting Combinator Parsers. In *Advanced Functional Programming, Second International School-Tutorial Text*. Springer-Verlag, London, UK, 184–207. http://dl.acm.org/citation.cfm?id=647699.734159

[29] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[30] Jamie Willis. 2021. `parsley`: A fast parser combinator library backed by Typed Template Haskell. https://hackage.haskell.org/package/parsley.

[31] Jamie Willis and Nicolas Wu. 2018. Garnishing Parsec with Parsley. In *9th ACM SIGPLAN International Symposium on Scala* (St. Louis, MO, USA) *(Scala '18)*. ACM, New York, NY, USA, 24–34. https://doi.org/10.1145/3241653.3241656

[32] Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged Selective Parser Combinators. *Proc. ACM Program. Lang.* 4, ICFP (2020). https://doi.org/10.1145/3409002