Ragax: Ragalur Expressions

Using derivatives to validate Indian Classical Music

Walter Schulze

2 May 2018

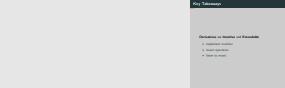Amsterdam Functional Programming Meetup

Hello

Welcome to my talk on Ragax or Ragalur Expressions ... pun intended.

Key Takeaways

Derivatives are **Intuitive** and **Extendable**

- implement matcher
- invent operators
- listen to music

Ragax: Ragalur Expressions

└─Key Takeaways

Key Takeaways

Derivatives are Intuitive and Extendable
- implement matcher
- invent operators
- listen to music

I am going to use derivatives to show you how easy it is to implement your own regular expression matcher function. Derivatives are so trivial and extendable, that you should even be able to invent your own regular expression operators. This also means that we can extend derivatives to even validate Context Free Grammars, which we can use for Ragas, the whole point of this talk.

$$a(a|b)*$$

ab	✓
aabbba	✓
ac	✗
ba	✗

$$a \cdot (a \mid b)^*$$

Regular Expressions

Let's do a quick review of regular expressions, just to make sure that we are on the same page. Here we see an expression that matches a string that starts with an 'a', which is followed by any number of 'a's and 'b's
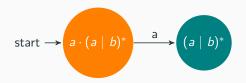
We can also write it like this (point to bottom). Which is the syntax we'll use in the rest of this talk. So the concat operator is represented with a dot.

Also notice that we are not doing substring matching and that we only match the whole string. See for example "ba" that is not matched.

Any questions?

What is a Brzozowski Derivative
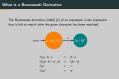
The Brzozowski derivative (1964) [1] of an expression is the expression that is left to match after the given character has been matched.

$$\partial_a a \cdot b \cdot c \quad = \quad b \cdot c$$
$$\partial_a (a \cdot b \mid a \cdot c) \quad = \quad (b \mid c)$$
$$\partial_c c \quad = \quad \epsilon$$
$$\partial_a a^* \quad = \quad a^*$$

What is a derivative? ... The derivative of an expression is the expression that
is left to match after the given character has been matched. So the derivative
of the expression 'a' concatenated with 'a' "or" 'b' star with respect to 'a' is 'a'
"or" 'b' star.

Here are some more examples:

- The derivative of "abc" with respect to 'a' is "bc".

- With an "or" we have to try both alternatives so we take the derivative of
  both.

- The derivative of 'c' with respect to 'c' is the empty string

- The derivative of 'a' star with respect to 'a' stays 'a' star.

I will now explain the formal rules, but I hope that these examples give you an
intuition.

Basic Operators

empty set	$\emptyset$
empty string	ε
character	a
concatenation	$r \cdot s$
zero or more	r^*
logical or	$r \mid s$

Ragax: Ragalur Expressions

2018-04-20

└─Basic Operators

Basic Operators

empty set	∅
empty string	ε
character	a
concatenation	r · s
zero or more	r*
logical or	r | s

First we have a few basic operators.

If we think of the set of strings that matches a regular expression, then the empty set does not match any strings.
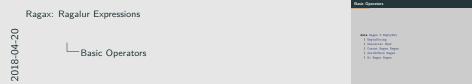
The empty string matches only the empty string.

The character a matches only the character a, not the substring a, remember we are only dealing with whole strings.

And then we have the rest: concat, zero or more and or.

Basic Operators

```
data Regex = EmptySet
  | EmptyString
  | Character Char
  | Concat Regex Regex
  | ZeroOrMore Regex
  | Or Regex Regex
```

Basic Operators

```
data Regex = EmptySet
   | EmptyString
   | Character Char
   | Concat Regex Regex
   | ZeroOrMore Regex
   | Or Regex Regex
```

We can represent this in haskell using an algebriac data type.

But you don't need to understand the haskell to understand most of this talk.

I just thought that for those who know it, it might make the math easier.

Nullable

Does the expression match the empty string.

$$
\begin{aligned}
\nu(\emptyset) &= \text{false} \\
\nu(\varepsilon) &= \text{true} \\
\nu(a) &= \text{false} \\
\nu(r \cdot s) &= \nu(r) \text{ and } \nu(s) \\
\nu(r^*) &= \text{true} \\
\nu(r \mid s) &= \nu(r) \text{ or } \nu(s)
\end{aligned}
$$

Nullable

Nullable

Does the expression match the empty string.

$$\nu(\emptyset) = false$$
$$\nu(\varepsilon) = true$$
$$\nu(a) = false$$
$$\nu(r \cdot s) = \nu(r) \ and \ \nu(s)$$
$$\nu(r^*) = true$$
$$\nu(r \mid s) = \nu(r) \ or \ \nu(s)$$

Before we can explain the derivative algorithm we first need to understand the nullable function.

The nullable function returns true if the set of strings that the regular expression matches includes the empty string.

The emptyset matches no strings, so it also does not match the empty string.

the empty string, well uhm yes

the character a, no, because it only matches the character a

The concatenation of r and s, well only if r and s contain the empty string.
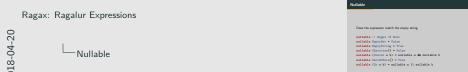
r star, zero or more, includes zero, which is the empty string

r or s, well r or s needs to include the empty string.

Does the expression match the empty string.

```
nullable :: Regex -> Bool
nullable EmptySet = False
nullable EmptyString = True
nullable Character{} = False
nullable (Concat a b) = nullable a && nullable b
nullable ZeroOrMore{} = True
nullable (Or a b) = nullable a || nullable b
```

Nullable

Does the expression match the empty string.

```haskell
nullable :: Regex -> Bool
nullable EmptySet = False
nullable EmptyString = True
nullable Character{} = False
nullable (Concat a b) = nullable a && nullable b
nullable ZeroOrMore{} = True
nullable (Or a b) = nullable a || nullable b
```

Or if you prefer the haskell

Nullable Examples

$$\nu(a \cdot b \cdot c) \quad = \quad \textcolor{orange}{\times}$$
$$\nu(\varepsilon) \quad = \quad \textcolor{green}{\checkmark}$$
$$\nu(a \mid b) \quad = \quad \textcolor{orange}{\times}$$
$$\nu(\varepsilon \mid a) \quad = \quad \textcolor{green}{\checkmark}$$
$$\nu(a \cdot \varepsilon) \quad = \quad \textcolor{orange}{\times}$$
$$\nu((a \cdot b)^*) \quad = \quad \textcolor{green}{\checkmark}$$
$$\nu(c \cdot (a \cdot b)^*) \quad = \quad \textcolor{orange}{\times}$$

2018-04-20

Nullable Examples

$$\nu(a \cdot b \cdot c) \quad = \quad \times$$
$$\nu(\varepsilon) \quad = \quad \checkmark$$
$$\nu(\varepsilon \mid b) \quad = \quad \checkmark$$
$$\nu(\varepsilon \mid a) \quad = \quad \checkmark$$
$$\nu(a \cdot \varepsilon) \quad = \quad \times$$
$$\nu((a \cdot b)^*) \quad = \quad \checkmark$$
$$\nu(c \cdot (a \cdot b)^*) \quad = \quad \times$$

Let's look at some examples or maybe you can tell me if there is an example here that you disagree with or don't understand … everyone happy?

Derivative Rules

$$
\begin{aligned}
\partial_a \emptyset &= \emptyset \\
\partial_a \epsilon &= \emptyset \\
\partial_a a &= \epsilon \\
\partial_a b &= \emptyset && \text{for } b \neq a \\
\partial_a (r \cdot s) &= \partial_a r \cdot s && not(\nu(r)) \\
\partial_a (r \cdot s) &= \partial_a r \cdot s \mid \partial_a s && \nu(r) \\
\partial_a (r^*) &= \partial_a r \cdot r^* \\
\partial_a (r \mid s) &= \partial_a r \mid \partial_a s
\end{aligned}
$$

Derivative Rules

$$\partial_a \emptyset = \emptyset$$
$$\partial_a \epsilon = \emptyset$$
$$\partial_a a = \epsilon$$
$$\partial_a b = \emptyset \qquad \text{for } b \neq a$$
$$\partial_a(r \cdot s) = \partial_a r \cdot s \qquad not(v(r))$$
$$\partial_a(r \cdot s) = \partial_a r \cdot s \mid \partial_a s \qquad v(r)$$
$$\partial_a(r^*) = \partial_a r \cdot r^*$$
$$\partial_a(r \mid s) = \partial_a r \mid \partial_a s$$

Now lets do the formal derivative rules. The derivative of the emptyset is always going to be the emptyset. The derivative of the empty string is also always the emptyset. The empty string does not expect any more characters. It is done.

The derivative of a single character given the same character is the empty string, but otherwise it won't match any string, so it is the emptyset.

The derivative of a concatenation has two cases. If the left expression is not nullable then we simply take the derivative of the left expression concatenated to the right. But if the left expression is nullable, then we have to consider the case where we skip over it and take the derivative of the right expression.

The derivative of the zero or more expression is the concatenation of the derivative of the contained expression and the original zero or more expression. The or expression simply pushes its problems down to its children.

Derivative Rules

```
deriv :: Regex -> Char -> Regex
deriv EmptyString _ = EmptySet
deriv EmptySet _ = EmptySet
deriv (Character a) c = if a == c
  then EmptyString else EmptySet
deriv (Concat r s) c =
  let left = deriv r c
      right = deriv s c
  in if nullable r
     then Or (Concat left s) right
     else Concat left s
deriv (ZeroOrMore r) c =
  Concat (deriv r c) (ZeroOrMore r)
deriv (Or r s) c =
  Or (deriv r c) (deriv s c)
```

Derivative Rules

```
deriv :: Regex -> Char -> Regex
deriv EmptyString _ = EmptySet
deriv EmptySet _ = EmptySet
deriv (Character a) c = if a == c
  then EmptyString else EmptySet
deriv (Concat r s) c =
  let left = deriv r c
      right = deriv s c
  in if nullable r
     then Or (Concat left s) right
     else Concat left s
deriv (ZeroOrMore r) c =
  Concat (deriv r c) (ZeroOrMore r)
deriv (Or r s) c =
  Or (deriv r c) (deriv s c)
```

Maybe the haskell is easier to understand.

Our regular expression matcher

$$\nu(foldl(\partial, r, str)) \qquad\qquad\qquad \text{where}$$
$$foldl(\partial, r, str) \quad = \quad r \qquad\qquad \text{if } str == ""$$
$$\qquad\qquad\qquad = \quad foldl(\partial, \partial_{s[0]}(r), s[1:]) \qquad \text{otherwise}$$

```
match :: Regex -> String -> Bool
match r str = nullable (foldl deriv r str)

func matches(r *expr, str string) bool {
    for _, c := range str {
        r = deriv(r, c)
    }
    return nullable(r)
}
```

Our regular expression matcher

$$\kappa(foldl(\partial, r, str))$$
$$foldl(\partial, r, str) \quad = \quad r \qquad\qquad\qquad \text{if } str == ""$$
$$\qquad\qquad\qquad = \quad foldl(\partial, \partial_{s[0]}(r), s[1:]) \quad \text{otherwise}$$

```
match :: Regex -> String -> Bool
match r str = nullable (foldl deriv r str)

func matches(r *expr, str string) bool {
    for _, c := range str {
        r = deriv(r, c)
    }
    return nullable(r)
}
```
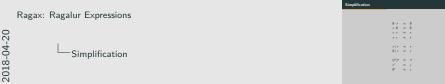
Now lets put our two functions together to create a regular expression matcher.

Here we have the math, haskell and Go implementations of the match function. You only need to understand one.

We simply apply the derivative function to each character in the string starting with our regular expression.

If the regular expression that we end up with matches the empty string, then the regular expression matches the input string.

Simplification

$$
\begin{aligned}
\emptyset \cdot r &\approx \emptyset \\
r \cdot \emptyset &\approx \emptyset \\
\varepsilon \cdot r &\approx r \\
r \cdot \varepsilon &\approx r \\
\\
r \mid r &\approx r \\
\emptyset \mid r &\approx r \\
\\
(r^*)^* &\approx r^* \\
\varepsilon^* &\approx \varepsilon \\
\emptyset^* &\approx \varepsilon
\end{aligned}
$$

Simplification

$\emptyset \cdot r \;\approx\; \emptyset$
$r \cdot \emptyset \;\approx\; \emptyset$
$r \cdot \varepsilon \;\approx\; r$
$\varepsilon \cdot r \;\approx\; r$

$r \mid r \;\approx\; r$
$\emptyset \mid r \;\approx\; r$

$(r^*)^* \;\approx\; r^*$
$\varepsilon^* \;\approx\; \varepsilon$
$\emptyset^* \;\approx\; \varepsilon$

Finally we have simplification which is optional, but is great for optimization and will make our examples much more readable. Here are some of the rules, which should be intuitive:

- Any expression concatenated with the emptyset is equivalent to the emptyset.

- Any expression concatenated with the empty string is equivalent to the expression.

- Any expression ored with itself is equivalent to that expression.

- Any expression ored with the emptyset is equivalent to that expression.

- Zero or more, zero or more times, is still just zero or more.

Simplification can be part of the derivative function or happen in the constructors.

Example: Matching a sequence of notes

Using a regex we can validate the C Major Pentatonic Scale.

$$c \cdot (c|d|e|g|a)^*$$

$$\text{ceg} \quad \checkmark$$

$$
\begin{aligned}
\partial_c c \cdot (c|d|e|g|a)^* &= & \varepsilon \cdot (c|d|e|g|a)^* \\
\partial_e \varepsilon \cdot (c|d|e|g|a)^* &= & (\emptyset \cdot (c|d|e|g|a)^*) \mid (\emptyset|\emptyset|\varepsilon|\emptyset|\emptyset) \cdot (c|d|e|g|a)^* \\
&= & (c|d|e|g|a)^* \\
\partial_g (c|d|e|g|a)^* &= & (\emptyset|\emptyset|\emptyset|\varepsilon|\emptyset) \cdot (c|d|e|g|a)^* \\
&= & (c|d|e|g|a)^*
\end{aligned}
$$

$$\nu((c|d|e|g|a)^*) = \checkmark$$

Ragax: Ragalur Expressions

Example: Matching a sequence of notes

Ok finally we have another example. Let's say our characters are musical notes. And we have an expression for a c major pentatonic scale.

We can take the derivative of the regular expression with respective to each input note to get a resulting regular expression. It matches if the resulting regular expression is nullable.

Let's walk through it. The derivative of the initial expression with respect to c is the empty string concatenated with the zero or more expression. We could simplify that, but lets first try taking the next derivative, for the sake of example.

So the derivative of the empty string concatenated with the zero or more expression is the emptyset concatenated with the zero or more expression, but since the empty string is nullabe we also have to take the derivative of the right expression as an alternative.
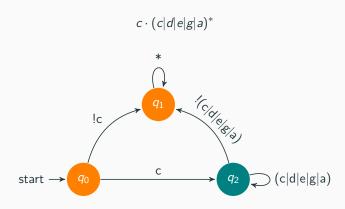
Questions?

Here we can see that every musical note has become the emptyset, except the matching character which has become the empty string. This big Or is then concatenated with the original zero or more expression. So here we had to apply the concat rule, the empty string rule, zero or more rule and the or rule. This easily simplifies to the zero or more expression.

Finally we take the derivative with respect to g which is almost a repeat of the previous equation.
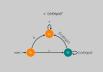
Then we are left with the zero or more expression, which is nullable and means that our string matches the expression and thats it.

This is the part of the talk that everything builds on. If you don't understand something lets quickly take some time to try and get it right. At this point you should be able to write your own regular expression matcher function.

$$c \cdot (c|d|e|g|a)^*$$

Let's take a quick tangent.

Do you remember deterministic finite automata. If you don't, don't worry about it, you won't need this for the next part. I just think it is really cool.

Here is one for the regular expression we just evaluated.

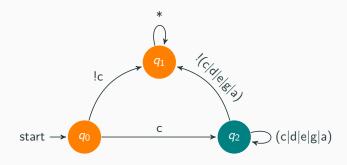Given a c we go the accepting state. Then we accept any note in the pentatonic scale zero or more times. Otherwise the song is rejected.

Memoization and Simplification

$$
\begin{aligned}
q_0 &= c \cdot (c|d|e|g|a)^* \\
q_1 &= \emptyset \\
q_2 &= (c|d|e|g|a)^*
\end{aligned}
$$

- Memoizing deriv = transition function
- Memoizing nullable = accept function
- Simplification = minimization [4]

We can create the same DFA with derivatives by simply memoizing the derivative function for all possible inputs.

The memoized derivative function is the transition function where the regular expressions themselves are states. The memoized nullable function is the accept function. Our simplification rules can be used for minimization.

I just think this is so cool that we get this for free.

Recursive Regular Expressions

$$((a \cdot b)^* \mid c)$$

Two new operations:

- Define a reference: $\#myref = (a \cdot b)^*$
- Use a reference: $(@myref \mid c)$

$$
\begin{array}{rcl}
\partial_a @q & = & \partial_a \#q \\
\nu(@q) & = & \nu(\#q)
\end{array}
$$

└─Recursive Regular Expressions

Ok and back from the tangent.

We now need to add references for recursion, to define our Ragas with.

For this we need two operators: the definition of a reference and the use of the reference.

The derivative of @q with respect to a is the derivative of whatever q's definition was with respect to a and the same with nullable.

Ragas - Indian Classical Music

Now that we have all the tools that we need, we can move on to Ragas. First I have to thank Ryan Lemmer and Antoine Van Gelder for introducing me to Ragas and for coming up with the idea to combine Ragas with derivatives.

https://youtu.be/iElMWziZ62A?t=136

2018-04-20

Here is an example of someone playing a Raga. Notice how the player slides his hand up and down the strings. There are very strict rules that decide what notes he is allowed to play next.

Ragas

Ragas are indian version of western scales [5]:

- Stricter
- Next note depends on current note.
- Notes named differently and relative to root note.

Raga	S	r	R	g	G	m	M	P	d	D	n	N
Western	c	c♯	d	d♯	e	f	f♯	g	g♯	a	a♯	b

Ragas are basically the indian version of western scales.

They are stricter than western scales.
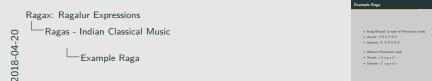
The possible next note is not simply chosen from a set, but rather depends on the current note.

Also notes are labeled a bit differently. They are labeled relatively to the start note. So this is simply how they are labeled if you start at c.

I will skip over microtones and all the other theory, just because it is not relevant to this talk.

Example Raga

- Raag Bhupali (a type of Pentatonic scale)
- Ascent: S R G P D S'
- Descent: S' D P G R S

- Western Pentatonic scale
- Ascent: c d e g a c^1
- Descent: c^1 a g e d c

Example Raga

- Raag Bhupali (a type of Pentatonic scale)
- Ascent: S R G P D S'
- Descent: S' D P G R S

- Western Pentatonic scale
- Ascent: c d e g a c'
- Descent: c' a g e d c

Here is Raag Bhupali a type of Pentatonic scale

Given the current note you must choose the next ascending or descending note.

For example given G (e) you can choose P (g) if you want to ascend or R (d) if you want to descend.

http://raag-hindustani.com/22_files/
ArohBhupali.mp3

2018-04-20

Let's listen to this scale

Questions?

Questions

Is everyone still with us. After this things start to speed up a little. It starts to become less of a lesson and more a presentation. So its good to have this part solid.

A Grammar for a Raga

- Raag Bhupali (a type of Pentatonic scale)
- Ascent: S R G P D S'
- Descent: S' D P G R S

$$
\begin{aligned}
\#S &= (S \cdot (@R \mid @D))^* \\
\#R &= R \cdot (@G \mid \varepsilon) \\
\#G &= G \cdot (@P \mid @R) \\
\#P &= P \cdot (@D \mid @G) \\
\#D &= D \cdot (\varepsilon \mid @P)
\end{aligned}
$$

A Grammar for a Raga

- Raag Bhupali (a type of Pentatonic scale)
- Ascent: S R G P D S'
- Descent: S' D P G R S

$$\#S = (S \cdot (@R \mid @D))^*$$
$$\#R = R \cdot (@G \mid \varepsilon)$$
$$\#G = G \cdot (@P \mid @R)$$
$$\#P = P \cdot (@D \mid @G)$$
$$\#D = D \cdot (\varepsilon \mid @P)$$

Ok so here finally we have a raga expressed as a recursive regular expression.

Our first note S is followed by its ascent or descent note and the whole expression is repeated zero or more times.

The ascending note R is ascended by G or descended back to S which is just the empty string, since the termination of the expression results in the end or the repetition of the whole expression which starts with S.

Same goes for the rest.

G is followed by P or R

P is followed by D or G

and D is followed by P or a termination, since it can be followed by S.

Demo

Let's see it in action

This program basically does not allow me to play a note that will make the expression go into an emptyset state.

We can even add random input to create a generated piece that satisfies the raga rules.

Context Free Grammars

I hope you enjoyed that, because now for pudding I have some vegetables.

Don't worry if you don't understand the following part, because it took me quite a while. I think it is valuable to see some functional concepts applied in a creative way.

Oh and by the way if you are not going to be pendantic then Context free grammars are just another name for recursive regular expressions. Our example from before was already a context free grammar.

Left Recursive Raga

$$\begin{aligned}
\#S &= (S \cdot (@R \mid @D))^* &=& \; @S \cdot (S \cdot (@R \mid @D)) \mid \varepsilon \\
\#R &= R \cdot (@G \mid \varepsilon) \\
\#G &= G \cdot (@P \mid @R) \\
\#P &= P \cdot (@D \mid @G) \\
\#D &= D \cdot (\varepsilon \mid @P)
\end{aligned}$$

nullable and derivative each have infinite recursion.

$$\nu(\#S) = (\nu(@S) \text{ and } \nu(S \cdot (@R \mid @D))) \text{ or } \nu(\varepsilon)$$

$$\#S = [S \cdot (@R \mid @D)]^* = @S \cdot (S \cdot (@R \mid @D)) \mid \varepsilon$$
$$\#R = R \cdot (@G \mid \varepsilon)$$
$$\#G = G \cdot (@P \mid @R)$$
$$\#P = P \cdot (@D \mid @G)$$
$$\#D = D \cdot (\varepsilon \mid @P)$$

nullable and derivative each have infinite recursion.

$$\nu(\#S) = (\nu(@S) \text{ and } \nu(S \cdot (@R \mid @D))) \text{ or } \nu(\varepsilon)$$

Here is our expression from before, but just written a bit differently. We cannot really control how a user uses our expression language, so we have to think of all cases.

The definition of S is either the empty string or it is the S again followed by the expression contained in the original zero or more expression.

You can see how these are equivalent.

Unfortunately this causes infinite recursion for the nullable and derivative function.

We can see that calculating nullable for S requires the calculation of the nullability of S.

Parsing with Derivatives

This has been solved using [3] functional concepts:

- laziness - The Brake
- memoization - The Handbrake
- least fixed point - The Gas

We are going to solve this problem with 3 functional concepts.

- laziness

- memoization

- least fixed point

I like to call this these the: brake, handbrake and gas respectively.

Laziness

Strict:

```
func eval(params ...) value {
    ...
    return value(...)
}
```

Lazy:

```
func lazyEval(params ...) func() value {
    return func() value {
        return eval(params)
    }
}
```

```
Strict:

func eval(params ...) value {
    ...
    return value(...)
}

Lazy:

func lazyEval(params ...) func() value {
    return func() value {
        return eval(params)
    }
}
```

Laziness is when instead of evaluating a function and returning a value, we defer the evaluation and return a function that will return the value. Sorry it is kind of hard to demonstrate laziness in haskell, since it is lazy by default.

$$\lambda \implies \text{laziness}$$

$$
\begin{aligned}
\partial_a(r|s) &= \partial_a r \mid \partial_a s &&= \lambda(\partial_a r) \mid \lambda(\partial_a s) \\
\partial_a(r^*) &= \partial_a r \cdot r^* &&= \lambda(\partial_a r) \cdot r^* \\
\partial_a(r \cdot s) &= \partial_a r \cdot s \mid \jmath(r) \cdot \partial_a s &&= \lambda(\lambda(\partial_a r) \cdot s) \mid \lambda(\lambda(\jmath(r)) \cdot \lambda(\partial_a s))
\end{aligned}
$$

$$
\begin{aligned}
\jmath(r) &= \epsilon && \text{if } \nu(r) \\
&= \emptyset && \text{otherwise}
\end{aligned}
$$

$$\partial_n \#S = \lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D)))) \mid \lambda(\partial_n \varepsilon)$$

23

Laziness - The Brake

$$\lambda \implies \text{laziness}$$

$$\partial_a(r_1 j r_2) = \partial_a r_1 \mid \partial_a r_2 = \lambda(\partial_a r_1) \mid \lambda(\partial_a r_2)$$
$$\partial_a(r^*) = \partial_a r \cdot r^* = \lambda(\partial_a r) \cdot r^*$$
$$\partial_a(r \cdot s) = \partial_a r \cdot s \mid \delta(r) \cdot \partial_a s = \lambda(\lambda(\partial_a r) \cdot s) \mid \lambda(\lambda(\delta(r)) \cdot \lambda(\partial_a s))$$

$$\delta(r) = \epsilon \quad \text{if } \nu(r)$$
$$= \emptyset \quad \text{otherwise}$$

$$\partial_a \#S = \lambda(\partial_a(\texttt{@S} \cdot (S \cdot (\texttt{@R} \mid \texttt{@D})))) \mid \lambda(\partial_a r)$$

I used the lambda symbol here to represent a thunk or function that will return a value when called with no parameters.

So instead of storing the field values of concat, or and zero or more we rather store functions that when called will return the value of the field.

This allows us to avoid any recursion until the value is needed.

The j is used as shorthand to write concat in a single line.

Below we can see our expression rewritten with laziness.

Nullable is called:

$$
\begin{aligned}
\nu(\partial_n \# S) &= \nu(\lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D)))) \mid \lambda(\partial_n \varepsilon)) \\
&= \nu(\lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D))))) \mid \nu(\lambda(\partial_n \varepsilon))
\end{aligned}
$$

Lazy function is executed:

$$
\begin{aligned}
\lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D)))) &= \partial_n(@S \cdot (S \cdot (@R \mid @D))) \\
&= \lambda(\lambda(\partial_n @S) \cdot \lambda((S \cdot (@R \mid @D)))) \mid \\
&\quad \lambda(\lambda(\jmath(@S)) \cdot \lambda(\partial_n(S \cdot (@R \mid @D)))) \\
\lambda(\partial_n @S) &= \partial_n @S
\end{aligned}
$$

Infinite recursion:

$$
\partial_n @S = \partial_n \# S
$$

Memoizing closes the loop:

$$
\partial_n @S = \lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D)))) \mid \lambda(\partial_n \varepsilon)
$$

Eventually nullability is going to be called and this won't be stopped by laziness alone. It will result in the execution of a lazy derivative function.

The point of these equations are that we still need the nullability of the derivative of S to calculate the nullability of the derivative of S.

Memoizing helps to close the loop.

It stops the recursive execution of the derivative by only calculating the derivative once and returning the same lazy function for following executions.

Memoization

```go
func memoize(eval func(a) b) func(a) b {
    mem := make(map[a]b)
    return func(input a) b {
        if output, ok := mem[input]; ok {
            return output
        }
        output := eval(a)
        mem[input] = output
        return output
    }
}
```

```go
func memoize(eval func(a) b) func(a) b {
    mem := make(map[a]b)
    return func(input a) b {
        if output, ok := mem[input]; ok {
            return output
        }
        output := eval(a)
        mem[input] = output
        return output
    }
}
```

Memoizing a function is when you cache the results for the previous inputs and so only calculate the value for the same inputs once. Just like caching it is typically a great way to optimize your code. But in this case, we are using it to close a loop. Unfortunately it is not enough. We need another tool.

$$f(x) = x^2$$

$$f(0) = 0^2$$

$$f(1) = 1^2$$

fixed points $= \{0, 1\}$

least fixed point $= 0$

Least Fixed Point

$f(x) = x^2$

$f(0) = 0^2$

$f(1) = 1^2$

fixed points = {0, 1}

least fixed point = 0

A fixed point of a function is where the function output equals the input. For example: x squared has two fix points, one and zero. The least fixed point would be the zero, since it is smaller than one.

Least Fixed Point of Derivative

$$\partial_a r = r$$

$$\partial_a \emptyset = \emptyset$$

$$\partial_a a^* = a^*$$

fixed points $= \{\emptyset, a^*\}$

least fixed point $= \emptyset$

Least Fixed Point of Derivative

$\partial_a r = r$
$\partial_a \emptyset = \emptyset$
$\partial_a a^* = a^*$
fixed points $= \{\emptyset, a^*\}$
least fixed point $= \emptyset$

What fixed points do we have for the derivative function? The derivative of the empty set is always the empty set and the derivative of 'a' star given an 'a' is also 'a' star. The empty set is the least fixed point, given it is the simplest expression.

Least Fixed Point - The Gas

Nullable is relentless:

$$\nu(\lambda(\partial_n \# S)) = \dots$$
$$\nu(\lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D))))) \mid \nu(\lambda(\partial_n \varepsilon)) = \dots$$
$$\nu(\lambda(\lambda(\partial_n @S) \cdot \lambda(\dots))) = \dots$$
$$\nu(\lambda(\partial_n @S)) = \nu(\text{fix})$$
$$= \nu(\emptyset)$$
$$= \text{false}$$
$$\nu(\lambda(\lambda(\partial_n @S) \cdot \lambda(\dots))) = \text{false \& false}$$
$$\nu(\lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D))))) \mid \nu(\lambda(\partial_n \varepsilon)) = \text{false} \mid \text{false}$$
$$\nu(\lambda(\partial_n \# S)) = \text{false}$$

Least Fixed Point - The Gas

Nullable is relentless:

$$\nu(\lambda(A_\epsilon\#S))$$ = ...
$$\nu(\lambda(A_\epsilon\#S \cdot (S \cdot (@R \mid @D))))) \mid \nu(\lambda(@A_\epsilon))$$ = ...
$$\nu(\lambda(A_\epsilon\#S) \cdot \lambda(\ldots)))$$ = ...
$$\nu(\lambda(A_\epsilon\#S))$$ = $$\nu(\tau_\epsilon)$$
= $$\nu(l)$$
= false
$$\nu(\lambda(\lambda(A_\epsilon\#S) \cdot \lambda(\ldots)))$$ = false & false
$$\nu(\lambda(A_\epsilon\#S \cdot (S \cdot (@R \mid @D))))) \mid \nu(\lambda(@A_\epsilon))$$ = false | false
$$\nu(\lambda(A_\epsilon\#S))$$ = false

Eventually nullable still needs to return a true or a false.

We can use our least fixed point to return a derivative of empty set when
nullable revisits the same lazy expression.

http://awalterschulze.github.io/ragax/

And thats it we have solved the infinite recursion. Let's see it in action.

Here we see both versions of the grammars and how they both validate the same strings.

Yacc is Dead

Yacc, Antlr, Flex, Bison, etc. perform better.

But derivatives:

- more intuitive than LR and LALR parsers;
- only use functional techniques;
- recognize generalized Context Free Grammars, not just a subset.

Now we have a fully generalized Context Free Grammar validator.

Yacc, Antlr, Flex, Bison, etc. definitely still performs better, especially in worst case.

But derivatives:

- are a lot easier to implement and understand than LR and LALR parsers
- are implemented using only functional techniques
- can validate the full set of Context Free Grammars, not just a subset.

Trees

2018-04-20

Finally lets quickly brush over trees

Relaxing

http://relaxng.org/ [2] - RELAX NG is a schema language for XML, like XSchema and DTD.

Derivatives used for Implementation and Specification.

Polymorphic Regular Expressions: Characters $=>$ XMLNodes.

New Operators:

$$
\begin{array}{rcl}
\partial_a(r \,\&\&\, s) & = & (\partial_a r \,\&\&\, s) \mid (\partial_a s \,\&\&\, r) \\
\nu(r \,\&\&\, s) & = & \nu(r) \text{ and } \nu(s) \\
\emptyset \,\&\&\, r & \approx & \emptyset \\
\varepsilon \,\&\&\, r & \approx & r \\[1em]
\partial_a!(r) & = & !(\partial_a r) \\
\nu(!(r)) & = & \text{not}(\nu(r)) \\[1em]
(r)? & \approx & r \mid \varepsilon
\end{array}
$$

Relaxing

http://relaxng.org/ [2] - RELAX NG is a schema language for XML, like XSchema and DTD.

Derivatives used for implementation and Specification.

Polymorphic Regular Expressions: Characters => XMLNodes.

New Operators:

$$\partial_a(r \;\&\&\; s) = (\partial_a r \;\&\&\; s) \mid (\partial_a s \;\&\&\; r)$$
$$\nu(r \;\&\&\; s) = \nu(r) \text{ and } \nu(s)$$
$$\emptyset \;\&\&\; r \approx \emptyset$$
$$\varepsilon \;\&\&\; r \approx r$$

$$\partial_a(!(r)) = !(\partial_a r)$$
$$\nu(!(r)) = not(\nu(r))$$

$$(r)? \approx r \mid \varepsilon$$

The expressions recognized by derivatives can be polymorphic. They don't need to only apply to characters. They can rather refer to XML Nodes and this is how the RelaxNGs implementation and specification is done. RELAX NG is a schema language for XML, like XSchema and DTD. Relaxing also introduces some extra operators: Compliment, Interleave and Optional.

With interleave the derivative can take the derivative of any one of the interleaving patterns. The other pattern needs to keep its original form. Nullability is easy.

Not, like all the logical operators just passes its problem down.

And optional is just syntactic sugar.

TreeNode

```haskell
data Expr = ...
    TreeNode String Expr
    ...

deriv :: Expr -> Tree -> Expr
deriv (TreeNode nameExpr childExpr)
      (Node name children) =
  if nameExpr == name then
     let childDeriv = foldl deriv childExpr children
     in if nullable childDeriv
        then Empty
        else EmptySet
  else EmptySet

nullable (TreeNode _ _) = False
```

2018-04-20

Ragax: Ragalur Expressions

└─ Trees

        └─ TreeNode

```
data Expr = ...
     TreeNode String Expr
     ...

deriv :: Expr -> Tree -> Expr
deriv (TreeNode nameExpr childExpr)
        (Node name children) =
    if nameExpr == name then
      let childDeriv = foldl deriv childExpr children
      in if nullable childDeriv
          then Empty
          else EmptySet
    else EmptySet

nullable (TreeNode _ _) = False
```

The "hard" part is defining deriative function for the XML or Tree node operator. Here is an implementation in Haskell.

The TreeNode pattern has a name expression and a child expression. To keep it basic I have made the name expression just a plain string.

The Node has a label string and a list of child nodes. If the name expression equals the label string we have to take the derivative of the children. Otherwise we return the empty set just like with a character.

If the result of taking the derivative of the children is nullable we return the empty pattern else we return the empty set. Nullability of a treenode is always false, just like a character.

https://youtu.be/SvjSP2xYZm8
katydid.github.io

2018-04-20

I think its time for another video by Ze Frank.

And here is my project page.

Katydid: Relapse

Relapse: Tree Validation Language.

JSON, Protobufs, Reflected Go Structures and XML

Go, Haskell + Cross language testsuite

New Operators:

$$\begin{aligned}
\partial_a(r \,\&\, s) &= (\partial_a r \,\&\, \partial_a s) \\
\nu(r \,\&\, s) &= \nu(r) \text{ and } \nu(s) \\
\emptyset \,\&\, r &\approx \emptyset \\
r \,\&\, r &\approx r \\
* &\approx \,!(\emptyset) \\
.r &\approx * \cdot r \cdot *
\end{aligned}$$

https://github.com/katydid/katydid-haskell

Ragax: Ragalur Expressions
└─ Trees

└─ Katydid: Relapse

Finally the thing I am actually working on. Katydid is a tree toolkit, which includes Relapse: a validation language based on relaxing.

I use derivatives and memoization to build a visual pushdown automata. This allows me to do matching with zero memory allocations once the automata has been compiled.

I support: Json, Protobuf, XML and reflected structures, but its easy to add your own parser. I currently have implementations in Go and Haskell.

I also added some new operators: And, Contains and ZAny. Zany is just zero or more of anything, like .* in a regular expression. Which is just syntatic sugar for the compliment of the emptyset. And, again like all logical operators, just passes its problem down. Contains is just syntatic sugar for an expression surrounded by concatenated zanys.

```
http://katydid.github.io/play/
http://katydid.github.io/tour/
```

http://katydid.github.io/play/
http://katydid.github.io/tour/

I can now show you the playground.

And I also have a tour.

References

J. A. Brzozowski.
Derivatives of regular expressions.
Journal of the ACM (JACM), 11(4):481–494, 1964.

M. Makoto and J. Clark.
RELAX NG home page.
http://relaxng.org.

M. Might, D. Darais, and D. Spiewak.
Parsing with derivatives: a functional pearl.
In *Acm sigplan notices*. ACM, 2011.

S. Owens, J. Reppy, and A. Turon.
Regular-expression derivatives re-examined.
Journal of Functional Programming, 19(02):173–190, 2009.

Sādhana.
Hindustani Classical Music.
http://raag-hindustani.com/.

2018-04-20

References

📄 J. A. Brzozowski.
   Derivatives of regular expressions.
   Journal of the ACM (JACM), 11(4):481–494, 1964.

📄 M. Makoto and J. Clark.
   RELAX NG home page.
   http://relaxng.org.

📄 M. Might, D. Darais, and D. Spiewak.
   Parsing with derivatives: a functional pearl.
   In *Acm sigplan notices*. ACM, 2011.

📄 S. Owens, J. Reppy, and A. Turon.
   Regular-expression derivatives re-examined.
   Journal of Functional Programming, 19(02):173–190, 2009.

📄 Sādhana.
   Hindustani Classical Music.
   http://raag-hindustani.com/.

Thank you for listening.