Regular Expression Derivatives in Python

Michael Paddon
mwp@google.com

These slides are licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
Motivation

• I want to generate scanners that have guaranteed linear performance and understand Unicode.
• Owens, Reppy and Turon[1] describe how regular expression derivatives may be used to easily convert a regular expression into a deterministic finite automaton. They observe that "RE derivatives have been lost in the sands of time, and few computer scientists are aware of them".

Refresher: Regular Expressions

∅         null string
ε         empty string
a         symbol in alphabet Σ
r · s     concatenation
r*        Kleene closure
r + s     logical or (alternation)
r & s     logical and
¬r        complement

Examples:
    h · e · l · l · o
    (a · b · c) + (1 · 2 · 3)
    a · b* · c
Refresher: Deterministic Finite Automata (DFAs)

- Defined as:
  - <states, start, transitions, accepting, error>
Did you know you can take the derivative of a regular expression?

• It’s simply what’s left after feeding a symbol to an expression…

\[ \partial_a a = \varepsilon \]
\[ \partial_a b = \emptyset \]
\[ \partial_a (a \cdot b) = b \]
\[ \partial_a (a^*) = a^* \]
\[ \partial_a (a + b) = \partial_a a + \partial_a b = \varepsilon + \emptyset = \varepsilon \]

• This is called a Brzozowski derivative.
  – Invented by Janusz Brzozowski in 1964
More generally...

\[ \partial_a \emptyset = \emptyset \]
\[ \partial_a \varepsilon = \emptyset \]
\[ \partial_a a = \varepsilon \]
\[ \partial_a b = \emptyset \]
\[ \partial_a (\mathit{r} \cdot \mathit{s}) = \partial_a \mathit{r} \cdot \mathit{s} + \nu(\mathit{r}) \cdot \partial_a \mathit{s} \]
\[ \partial_a (\mathit{r}^*) = \partial_a \mathit{r} \cdot \mathit{r}^* \]
\[ \partial_a (\mathit{r} + \mathit{s}) = \partial_a \mathit{r} + \partial_a \mathit{s} \]
\[ \partial_a (\mathit{r} \& \mathit{s}) = \partial_a \mathit{r} \& \partial_a \mathit{s} \]
\[ \partial_a (\neg \mathit{r}) = \neg (\partial_a \mathit{r}) \]

Helper function \( \nu \):

\[ \nu(\varepsilon) = \varepsilon \]
\[ \nu(a) = \emptyset \]
\[ \nu(\emptyset) = \emptyset \]
\[ \nu(\mathit{r} \cdot \mathit{s}) = \nu(\mathit{r}) \& \nu(\mathit{s}) \]
\[ \nu(\mathit{r} + \mathit{s}) = \nu(\mathit{r}) + \nu(\mathit{s}) \]
\[ \nu(\mathit{r}^*) = \varepsilon \]
\[ \nu(\mathit{r} \& \mathit{s}) = \nu(\mathit{r}) \& \nu(\mathit{s}) \]
\[ \nu(\neg \mathit{r}) = \varepsilon, \text{ if } \nu(\mathit{r}) = \emptyset \]
\[ \nu(\neg \mathit{r}) = \emptyset, \text{ if } \nu(\mathit{r}) = \varepsilon \]

If \( \nu(\mathit{r}) = \varepsilon \), \( \mathit{r} \) is **nullable**

These rules taken from Owens, S., Reppy, J. and Turon, A.,
*Regular-expression derivatives re-examined*
How is this useful?

```
start = expr
states = {start}
transitions = {start: {}}

stack = [expr]
while stack:
    state = stack.pop()
    for symbol in alphabet:
        next_state = state.derivative(symbol)

        if next_state not in states:
            states.add(state)
            transitions[state] = []
            stack.append(next_state)

        transitions[state].add((symbol, next_state))

accepts = [state for state in states if state.nullable()]
error = states[∅]
```
What about large alphabets?

We can calculate *derivative classes*: 

\[
\begin{align*}
C() &= \{\Sigma\} \\
C(S) &= \{S, \Sigma \setminus S\}, \ S \subseteq \Sigma \\
C(r \cdot s) &= C(r), \text{ if } r \text{ is not nullable} \\
C(r \cdot s) &= C(r) \land C(s), \text{ if } r \text{ is nullable} \\
C(r + s) &= C(r) \land C(s) \\
C(r \& s) &= C(r) \land C(s) \\
C(r^*) &= C(r) \\
C(\neg r) &= C(r)
\end{align*}
\]

For example:

\[
\begin{align*}
C(a) &= \{a, \Sigma \setminus a\} \\
C(a \cdot b^*) &= C(a) = \{a, \Sigma \setminus a\} \\
C(a + b) &= C(a) \land C(b) \\
&= \{a, \Sigma \setminus a\} \land \{b, \Sigma \setminus b\} \\
&= \{\emptyset, a, b, \Sigma \setminus \{a, b\}\}
\end{align*}
\]

We only need to take a partial derivative for each class instead of each symbol.

These rules taken from Owens, S., Reppy, J. and Turon, A., *Regular-expression derivatives re-examined*
Now we can handle Unicode

```python
start = expr
states = {start}
transitions = {start: {}}

stack = [expr]
while stack:
    state = stack.pop()
    for dclass in state.derivative_classes():
        symbol = dclass.any_member_symbol()
        next_state = state.derivative(symbol)

        if next_state not in states:
            states.add(state)
            transitions[state] = []
            stack.append(next_state)

        transitions[state].add((symbol, next_state))

accepts = [state for state in states if state.nullable()]
error = states[φ]
```
Regular Vectors

- We can easily construct a single DFA from a vector of regular expressions!
  - $\partial_a <r_1, ..., r_n> = <\partial_a r_1, ..., \partial_a r_n>$
  - $C(r_1, ..., r_n) = \land C(r_i)$

- A sequence of regular expressions, each representing a token, can be reduced to a single DFA.

Vector rules taken from Owens, S., Reppy, J. and Turon, A., *Regular-expression derivatives re-examined*
Implementing in Python

- How do we represent large sets of symbols?
- How do we represent expressions?
- How do we compare expressions for equality?
- How do we build a scanner from a DFA?
Large Sets of Symbols

- Represent as ordered disjoint intervals of codepoints
  - e.g. [A-Za-z0-9] → ((48, 57), (65, 90), (97, 122))
  - Testing membership using `bisect()` is $O(\log N)$.
  - Union, intersection, difference is $O(N)$

- Tempting to subclass `collections.abc.Set()` to present a set of integers.
  - But want to support sets of symbol sets → need `hash()`
  - All sets with the same members should hash to the same value
  - The standard hash requires iterating over each member
  - Subclass `tuple` instead with set-like methods.
Expression Class Hierarchy

Expression
  \textit{derivative(symbol)}
  \textit{derivative\_class()}
  \textit{nullable()}

SymbolSet

KleeneClosure

Complement

LogicalAnd

Epsilon

LogicalOr

Concatenate
Expression Trees

\[ [A-Za-z] \cdot [A-Za-z]^* = \]

- **Concatenation**
  - **SymbolSet**
    - \(((65, 90), (97, 122))\)
  - **KleeneClosure**
    - **SymbolSet**
      - \(((65, 90), (97, 122))\)
Expression Equality

- Use `__new__() as a smart constructor for weak equivalence form which has a total ordering:

  \[
  \begin{align*}
  r \ &\& r &\approx r \\
  r \ &\& s &\approx s \ &\& r \\
  (r \ &\& s) \ &\& t &\approx r \ &\& (s \ &\& t) \\
  \emptyset \ &\& r &\approx \emptyset \\
  \neg \emptyset \ &\& r &\approx r \\
  (r \cdot s) \cdot t &\approx r \cdot (s \cdot t) \\
  \emptyset \cdot r &\approx \emptyset \\
  r \cdot \emptyset &\approx \emptyset \\
  \varepsilon \cdot r &\approx r \\
  r \cdot \varepsilon &\approx r \\
  r + r &\approx r \\
  r + s &\approx s + r \\
  (r + s) + t &\approx r + (s + t) \\
  \neg \emptyset + r &\approx \neg \emptyset \\
  \emptyset + r &\approx r \\
  (r\star)\star &\approx r\star \\
  \varepsilon\star &\approx \varepsilon \\
  \emptyset\star &\approx \varepsilon \\
  \neg(\neg r) &\approx r
  \end{align*}
  \]

These rules taken from Owens, S., Reppy, J. and Turon, A., *Regular-expression derivatives re-examined*
class Concatenation(Expression):
    def __new__(cls, left, right):
        if isinstance(left, Concatenation):
            left, right = left._left,
            Concatenation(left._right, right)
        if left == cls.NULL:
            return left
        elif right == cls.NULL:
            return right
        elif left == cls.EPSILON:
            return right
        elif right == cls.EPSILON:
            return left
        self = super().__new__(cls)
        self._left = left
        self._right = right
        return self
Building a Scanner

```python
state = start
match = None
for symbol in text:
    if state in accepts:
        match = state
        position = current_position()

    state = transition[state][symbol]
    if state == error:
        if match:
            yield match
            rewind_to(position)
            state = start

if match:
    yield match
```
Simple Example

Input looks like a configparser file:

```
[example]
_letter = [_A-Za-z]
_digit = [0-9]
identifier = <_letter>
               (<_letter>|<_digit>)*
number = <_digit>+
operator = [\-+\*/=]
onother = .
```
Resulting DFA
Pascal Lexer

• A larger example:
  - 51 expressions/tokens
  - `flex` → 174 states
  - Implemented in `epsilon` → 169 states
εpsilon

- Supports rich expression syntax:
  - Operators: (), [], !, &, |, ?, *, +, {count}, {min, max}
  - Escapes: mostly perlre compatible, including Unicode classes
- Designed to generate code for multiple targets
  - Currently Python and Dot
- Not done yet:
  - Start conditions, more targets including C
- Code at https://github.com/MichaelPaddon/epsilon
  - Beta testers and contributors welcome!
Acknowledgements

• epsilon was inspired by and directly based on the work of Owens, Reppy, and Turon

• Without the groundbreaking work of Janusz Brzozowski, none of this would be possible.
Thanks!