KLEENE ALGEBRA WITH TESTS

by Alexandra Silva

= applications to program equivalence and verification =

\[
\begin{align*}
\text{while } a \& b \text{ do} & \quad \text{while } a \text{ do} \\
\text{while } a \text{ do} & \quad \text{if } b \text{ then} \\
\text{?} & \quad \text{p else} \\
\text{while } a \& b \text{ do} & \quad \text{q}
\end{align*}
\]

two programs that both have tests in red and programs expressions like assignment in blue

are they doing the same thing? are they equal?

we will cover a way to decide this for uninterpreted simple imperative programs

regular expressions were introduced to reason about control flow

we will take an algebraic approach with regular expressions

Regular expressions (Syntax)

for talking about Regular languages (Semantics)

denotational correspondence by Kleene

Deterministic Finite Automata (Semantics, Operational)

\[ RE \in \{ 0, 1 \} \cup \{ a \in A \} \cup \{ e \} \cup \{ e \}^* \]

basis elements/finite composition

constants
\[ [e] : 2^{A^*} = \mathcal{P}(A^*) \]

\[ [0] = \emptyset \quad \text{no behavior} \]
\[ [1] = \{e\} \quad \text{empty word} \]
\[ [a] = \{a\} \quad \text{syntax} \quad \text{word in } A^* \quad \text{(semantics)} \]

\[ [e + f] = [e] \cup [f] \]

\[ [e; f] = [e] \cdot [f] \quad \text{elements of } e \text{ is followed by } f \]

\[ [e^*] = \bigcup_{n \in \mathbb{N}} [e]^n \quad \text{iterate } e \text{ as many times as one can} \]

\[ U, V : 2^{A^*} \quad \text{(sets of)} \]

\[ U \cdot V = \{uv \mid u \in U, v \in V\} \quad \text{concatenating words} \]

\[ V^0 = \{\varepsilon\} \quad \text{Semantics of star} \]

\[ V^{n+1} = V^n \cdot V \]
Examples of Regular Expressions

\[(1 + a) ; (1 + b) \rightarrow \{\varepsilon, a, b, a b\}\]

\[(a + b)^* \rightarrow \{a, b\}^* \]

\[(a^* b)^* \rightarrow \{a, b\}^* \cdot \cdot \cdot \]

\[(a + b)^* \]

\[\text{equal in the sense that their denotations are the same} \]

\[\text{called denesting! related to compiler optimizations} \]

\[\text{can apply this to go from one program to another} \]

\[0 + e \Xi e \]

\[\text{would like to reason at the program level, rather} \]

\[\text{than denotational level} \]

\[\text{is there a finite number of equations that allow program} \]

\[\text{refinement (i.e. reason about syntax level)?} \]
Kleene's Theorem

Let \( L \) be a regular language, then the following are equivalent:

1. \( L = \{ e \} \) for some regular exp. \( e \)
2. \( L \) is accepted by a DFA

Chomsky Hierarchy

Reg. Langs can be defined w/o respect to reg. exps.
Can be defined using regular sets
Syntax was set up to emulate regular sets
Exists research between the image of the syntax + regular set \( L \) (rational)
Syntactic way of building the automaton from expression via derivatives (small step semantics)

DFA:

\[ F : S \rightarrow 2 \]

- \( S \) - finite set of states
- Is a state final? \( 2 = \text{"true"} \)
- \( t : S \rightarrow S^A \)
  - Transitions deterministic, one state

Need corresponding functions for reg. exps!

\[ E : RE \rightarrow 2 \]
- Is this a final state, i.e., does it accept empty word!

\[ D : RE \rightarrow RE^A \]
- Next state, i.e., takes one letter off!

\[
\begin{align*}
E(0) &= 0 \quad \text{(false)} = E(a) = 0 \\
E(1) &= 1 \quad \text{(true)} = E(e^*) = 1 \\
E(e + f) &= E(e) \lor E(f) \\
E(e ; f) &= E(e) \land E(f) \\
E(e^*) &= 1
\end{align*}
\]

Given language \( L : 2^A^+ \)

Derivative \( L_a = \{ au | au \in L \} \)
Da : RE → RE
Da (0) = 0
Da (1) = 0 if \( a = b \) then 1
Da (b) = \{ \begin{align*}
& a \neq b \text{ then } 0 \\
& \text{ (removed } a \text{ from } e) \\
& \text{ key idea!}
\end{align*} \}

\[ E(e) \cdot D_e(f) \]

\[ D_{e+f} = D_e + D_f \]
\[ D_{e*} = D_e \cdot e^* \]

not quite a DFA though! need start states! and finiteness!

Can pick e as our start state, then need to argue that the set we reach from e is finite

is this true? almost! let's see →
\[ c = (a^*)^* \quad \text{Da}(e) = (1; a^*); (a^*)^* \]

\[ \text{Da} \left( \text{Da}(e) \right) = (0a^*+1a^*)a^*+\ (1a^*a^*)^* \]

This expression keeps showing up for each derivative!

\[ \text{Da} \left( \text{Da} \left( \text{Da}(e) \right) \right) = \ldots + (1a^*a^*)^* + 1(a^*a^*)^* \]

It's but this is equivalent to the previous derivative!

to get the finiteness, need equivalence classes of derivatives need to delete repeated expressions in a +

modulo ACI - associativity, commutativity, idempotence

\[ \text{Da} : \text{RE} \rightarrow \text{P(RE)} \]

Can write \[ \text{Da}(e+f) = \{ \text{Da}(e), \text{Da}(f) \} \]

get ACI for free! Shown by Antimorov

no need to reason syntactically i.e. modulo ACI

\[ \text{Da}(e+f) \]

\[ a \]

\[ \text{Da}(e) \]

\[ \text{Da}(f) \]

\[ \text{Da}(g) \]

Can combine into one transition

but still NFA! But can combine into one transition
Thompson construction

e; f

concatenate two automaton!

0 \rightarrow 0 \cdot a, b

1 \rightarrow 1 \cdot a, b

a \rightarrow a, b \cdot a, b

e + f

can use determinization/empty transition elimination to turn NFA to DFA