Regular Expressions & The Myhill-Nerode Theorem

Wu Chunhan

October 20, 2010

Outline

Outline

1. Regular Expression(brief)
2. Myhill-Nerode Theorem(Intro)
3. FA to Regular Expressions
4. Proving Myhill-Nerode Theorem
   - Well-Founded iterating principle
   - Invariant predicate
   - Generating initial ES
   - Iteration step of ES
   - Final Proof

Regular Expression

- What we may all know(in Compiling Principle)

Regular Expression

- What we may all know(in Compiling Principle)
  - An alphabet Σ where every language based

Regular Expression

- What we may all know(in Compiling Principle)
  - An alphabet Σ where every language based
  - $\emptyset \mid \lambda(\epsilon) \mid c \mid r_1 \cdot r_2 \mid r_1 | r_2 \mid r^*$

Regular Expression

- What we may all know(in Compiling Principle)
  - An alphabet Σ where every language based
  - $\emptyset \mid \lambda(\epsilon) \mid c \mid r_1 \cdot r_2 \mid r_1 | r_2 \mid r^*$
  - $\{\} \mid \{[]\} \mid \{[c]\} \mid L_1 ; L_2 \mid L_1 \cup L_2 \mid L*$

Regular Expression

- What we may all know(in Compiling Principle)
  - An alphabet Σ where every language based
  - $\emptyset \mid \lambda(\epsilon) \mid c \mid r_1 \cdot r_2 \mid r_1 | r_2 \mid r^*$
  - $\{\} \mid \{[]\} \mid \{[c]\} \mid L_1 ; L_2 \mid L_1 \cup L_2 \mid L*$

```
definition lang_seq :: "string set ⇒ string set ⇒ string set"
  ("_ ; _" [100,100] 100)
where
   "L1 ; L2 = {s1@s2 | s1 s2. s1 ∈ L1 ∧ s2 ∈ L2}"
```

Regular Expression

- What we may all know(in Compiling Principle)
  - An alphabet Σ where every language based
  - $\emptyset \mid \lambda(\epsilon) \mid c \mid r_1 \cdot r_2 \mid r_1 | r_2 \mid r^*$
  - $\{\} \mid \{[]\} \mid \{[c]\} \mid L_1 ; L_2 \mid L_1 \cup L_2 \mid L*$

```
definition lang_seq :: "string set ⇒ string set ⇒ string set"
  ("_ ; _" [100,100] 100)
where
  "L1 ; L2 = {s1@s2 | s1 s2. s1 ∈ L1 ∧ s2 ∈ L2}"


inductive_set Star :: "string set ⇒ string set" ("_⋆" [101] 102)
  for L :: "string set"
where
  start[intro]: "[] ∈ L⋆"
| step[intro]:  "⟦s1 ∈ L; s2 ∈ L⋆⟧ ⟹ s1@s2 ∈ L⋆"
```

Regular Expression(Formalization)

- In Isabelle/HOL

Regular Expression(Formalization)

- In Isabelle/HOL

```
datatype rexp =
   NULL
 | EMPTY
 | CHAR char
 | SEQ rexp rexp
 | ALT rexp rexp
 | STAR rexp
```

Regular Expression(Formalization)

- In Isabelle/HOL

```
datatype rexp =
   NULL
 | EMPTY
 | CHAR char
 | SEQ rexp rexp
 | ALT rexp rexp
 | STAR rexp


consts L:: "'a ⇒ string set"
overloading L_rexp == "L::  rexp ⇒ string set"
begin
fun L_rexp :: "rexp ⇒ string set"
where
    "L_rexp (NULL)    = {}"
  | "L_rexp (EMPTY)   = {[]}"
  | "L_rexp (CHAR c)  = {[c]}"
  | "L_rexp (SEQ r1 r2) = (L_rexp r1) ; (L_rexp r2)"
  | "L_rexp (ALT r1 r2) = (L_rexp r1) ∪ (L_rexp r2)"

  | "L_rexp (STAR r)    = (L_rexp r)⋆" end
```

Outline

Myhill-Nerode theorem

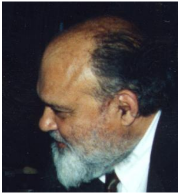
- In the theory of formal languages
- It provides a **necessary** & **sufficient** condition for a language to be regular
- Named after John Myhill and Anil Nerode
- Proved at University of Chicago, in 1958

Myhill-Nerode theorem

- In the theory of formal languages
- It provides a **necessary** & **sufficient** condition for a language to be regular
- Named after John Myhill and Anil Nerode
- Proved at University of Chicago, in 1958

Myhill-Nerode theorem

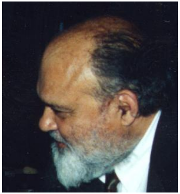
- In the theory of formal languages
- It provides a **necessary** & **sufficient** condition for a language to be regular
- Named after John Myhill and Anil Nerode

- Proved at University of Chicago, in 1958

Myhill-Nerode theorem

- In the theory of formal languages
- It provides a **necessary** & **sufficient** condition for a language to be regular
- Named after John Myhill and Anil Nerode



- Proved at University of Chicago, in 1958

Statement of the theorem

- A equivalence relation defined by *Lang*

$$x \equiv \text{Lang} \equiv y = (\forall z.\ (x\ @\ z \in \text{Lang}) = (y\ @\ z \in \text{Lang}))$$

Statement of the theorem

- A equivalence relation defined by *Lang*

  $$x \equiv\text{Lang}\equiv y = (\forall z.\ (x\ @\ z \in \text{Lang}) = (y\ @\ z \in \text{Lang}))$$

  - If $x \equiv\text{Lang}\equiv y$ and $x \in \text{Lang}$, then $y \in \text{Lang}$

Statement of the theorem

- A equivalence relation defined by *Lang*

$$x \equiv\text{Lang}\equiv y = (\forall z.\ (x \ @ \ z \in \text{Lang}) = (y \ @ \ z \in \text{Lang}))$$

  - If $x \equiv\text{Lang}\equiv y$ and $x \in \text{Lang}$, then $y \in \text{Lang}$
  - If $x \equiv\text{Lang}\equiv y$, then $(x@a) \equiv\text{Lang}\equiv (y@a)$

Statement of the theorem

- A equivalence relation defined by *Lang*

  $$x \equiv\mathrm{Lang}\equiv y = (\forall z.\ (x\ @\ z \in \mathrm{Lang}) = (y\ @\ z \in \mathrm{Lang}))$$

  - If $x \equiv\mathrm{Lang}\equiv y$ and $x \in \mathrm{Lang}$, then $y \in \mathrm{Lang}$
  - If $x \equiv\mathrm{Lang}\equiv y$, then $(x@a) \equiv\mathrm{Lang}\equiv (y@a)$

- A equivalence class defined by *Lang* & *x*

  $$[\![x]\!]\mathrm{Lang} \equiv \{y \mid x \equiv\mathrm{Lang}\equiv y\}$$

Statement of the theorem

- A equivalence relation defined by *Lang*

$$x \equiv\text{Lang}\equiv y = (\forall z.\ (x @ z \in \text{Lang}) = (y @ z \in \text{Lang}))$$

  - If x ≡Lang≡ y and x ∈ Lang, then y ∈ Lang
  - If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)

- A equivalence class defined by *Lang* & *x*

$$[\![x]\!]\text{Lang} \equiv \{y \mid x \equiv\text{Lang}\equiv y\}$$

- Partions of *Lang′* created by *Lang*

$$\text{Lang' Quo Lang} \equiv \{[\![x]\!]\text{Lang} \mid x \in \text{Lang'}\}$$

Statement of the theorem

- A equivalence relation defined by *Lang*

$$x \equiv\text{Lang}\equiv y = (\forall z.\ (x\ @\ z \in \text{Lang}) = (y\ @\ z \in \text{Lang}))$$

  - If x ≡Lang≡ y and x ∈ Lang, then y ∈ Lang
  - If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)

- A equivalence class defined by *Lang* & *x*

$$[\![x]\!]\text{Lang} \equiv \{y \mid x \equiv\text{Lang}\equiv y\}$$

- Partions of *Lang′* created by *Lang*

$$\text{Lang' Quo Lang} \equiv \{[\![x]\!]\text{Lang} \mid x \in \text{Lang'}\}$$

- Partions of Universal Language(*UNIV*)

Statement of the theorem

- A equivalence relation defined by *Lang*

  x ≡Lang≡ y = (∀z. (x @ z ∈ Lang) = (y @ z ∈ Lang))

  - If x ≡Lang≡ y and x ∈ Lang, then y ∈ Lang
  - If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)

- A equivalence class defined by *Lang* & *x*

  ⟦x⟧Lang ≡ {y | x ≡Lang≡ y}

- Partions of *Lang'* created by *Lang*

  Lang' Quo Lang ≡ {⟦x⟧Lang | x ∈ Lang'}

- Partions of Universal Language(*UNIV*)
  - Universal Language(*UNIV*) : $\Sigma*$

Statement of the theorem

- A equivalence relation defined by *Lang*

$$x \equiv\text{Lang}\equiv y = (\forall z.\ (x \mathbin{@} z \in \text{Lang}) = (y \mathbin{@} z \in \text{Lang}))$$

  - If x ≡Lang≡ y and x ∈ Lang, then y ∈ Lang
  - If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)

- A equivalence class defined by *Lang* & *x*

$$[\![x]\!]\text{Lang} \equiv \{y \mid x \equiv\text{Lang}\equiv y\}$$

- Partions of *Lang′* created by *Lang*

$$\text{Lang' Quo Lang} \equiv \{[\![x]\!]\text{Lang} \mid x \in \text{Lang'}\}$$

- Partions of Universal Language(*UNIV*)
  - Universal Language(*UNIV*) : $\Sigma*$

$$\text{Lang} = \bigcup \{X \mid \text{UNIV Quo Lang}\}\ (\forall x{\in}X.\ x \in \text{Lang})$$

Statement of the theorem (cont.)

Theorem

Lang is **regular** *iff it has finite partitions of UNIV*

(∃fa. lang_of_fa fa = Lang) = finite (UNIV Quo Lang)

Statement of the theorem (cont.)

Theorem

Lang is **regular** *iff it has finite partitions of UNIV*

$(\exists \text{fa. lang\_of\_fa fa} = \text{Lang}) = \text{finite (UNIV Quo Lang)}$

- *lang_of_fa* is for getting language from a FA

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]_{Lang}$ ($[|[]|]_{Lang}$)& Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0, 1\}$ & $Lang = L(0{\cdot}(0|1)^*)$

$[|\lambda|]Lang$ $\xrightarrow{\lambda \not\equiv Lang \not\equiv 0}$ $[|\lambda|]Lang$, $[|0|]Lang$ $\xrightarrow{0 \not\equiv Lang \not\equiv 1,\ \lambda \not\equiv Lang \not\equiv 1,\ \lambda \not\equiv Lang \not\equiv 0}$ $[|\lambda|]Lang$, $[|0|]Lang$, $[|1|]Lang$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]_{Lang}$ ($[|[]|]_{Lang}$)& Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0, 1\}$ & $Lang = L(0{\cdot}(0|1)^*)$

$$\boxed{[|\lambda|]_{Lang}} \xrightarrow{\lambda \not\equiv_{Lang} 0} \boxed{\begin{array}{l}[|\lambda|]_{Lang}\\ [|0|]_{Lang}\end{array}} \xrightarrow{\begin{array}{c}0 \not\equiv_{Lang} 1\\ \lambda \not\equiv_{Lang} 1\\ \lambda \not\equiv_{Lang} 0\end{array}} \boxed{\begin{array}{l}[|\lambda|]_{Lang}\\ [|0|]_{Lang}\\ [|1|]_{Lang}\end{array}}$$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]Lang$ ($[|[]|]Lang$)& Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0, 1\}$ & $Lang = L(0\cdot(0|1)^*)$

$$\boxed{[|\lambda|]Lang} \xrightarrow{\lambda \not\equiv Lang \not\equiv 0} \boxed{\begin{array}{l}[|\lambda|]Lang\\ [|0|]Lang\end{array}} \xrightarrow{\begin{array}{c}0 \not\equiv Lang \not\equiv 1\\ \lambda \not\equiv Lang \not\equiv 1\\ \lambda \not\equiv Lang \not\equiv 0\end{array}} \boxed{\begin{array}{l}[|\lambda|]Lang\\ [|0|]Lang\\ [|1|]Lang\end{array}}$$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]Lang$ $([|[]|]Lang)$ & Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0,1\}$ & $Lang = L(0\cdot(0|1)^*)$

$$\boxed{[|\lambda|]Lang} \xrightarrow{\lambda \not\equiv Lang \not\equiv 0} \boxed{\begin{array}{l}[|\lambda|]Lang\\ [|0|]Lang\end{array}} \xrightarrow{\begin{array}{c}0 \not\equiv Lang \not\equiv 1\\ \lambda \not\equiv Lang \not\equiv 1\\ \lambda \not\equiv Lang \not\equiv 0\end{array}} \boxed{\begin{array}{l}[|\lambda|]Lang\\ [|0|]Lang\\ [|1|]Lang\end{array}}$$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]Lang$ $([|[]|]Lang)$& Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0, 1\}$ & $Lang = L(0\cdot(0|1)^*)$

$\boxed{[|\lambda|]Lang}$ $\xrightarrow{\lambda \not\equiv_{Lang} 0}$ $\boxed{\begin{array}{l}[|\lambda|]Lang \\ [|0|]Lang\end{array}}$ $\xrightarrow{\begin{array}{l}0 \not\equiv_{Lang} 1 \\ \lambda \not\equiv_{Lang} 1 \\ \lambda \not\equiv_{Lang} 0\end{array}}$ $\boxed{\begin{array}{l}[|\lambda|]Lang \\ [|0|]Lang \\ [|1|]Lang\end{array}}$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]Lang$ ($[|[]|]Lang$)& Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0,1\}$ & $Lang = L(0\cdot(0|1)^*)$

$\boxed{[|\lambda|]Lang}$ $\quad \xrightarrow{\lambda \not\equiv_{Lang} 0} \quad$ $\boxed{\begin{matrix}[|\lambda|]Lang \\ [|0|]Lang\end{matrix}}$ $\quad \xrightarrow{\begin{matrix}0 \not\equiv_{Lang} 1 \\ \lambda \not\equiv_{Lang} 1 \\ \lambda \not\equiv_{Lang} 0\end{matrix}} \quad$ $\boxed{\begin{matrix}[|\lambda|]Lang \\ [|0|]Lang \\ [|1|]Lang\end{matrix}}$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]Lang$ ($[|[]|]Lang$)& Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0, 1\}$ & $Lang = L(0\cdot(0|1)^*)$

$[|\lambda|]Lang$ $\underset{\longrightarrow}{\lambda \not\equiv Lang \not\equiv 0}$ $[|\lambda|]Lang$ $[|0|]Lang$ $\underset{\longrightarrow}{0 \not\equiv Lang \not\equiv 1 \quad \lambda \not\equiv Lang \not\equiv 1 \quad \lambda \not\equiv Lang \not\equiv 0}$ $[|\lambda|]Lang$ $[|0|]Lang$ $[|1|]Lang$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]$*Lang* ($[|[]|]$*Lang*)& Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0,1\}$ & *Lang*$= L(0\cdot(0|1)^*)$

$\boxed{[|\lambda|]Lang} \xrightarrow{\lambda \not\equiv_{Lang} \not\equiv 0} \boxed{\begin{array}{l}[|\lambda|]Lang\\ [|0|]Lang\end{array}} \xrightarrow{\begin{array}{c}0 \not\equiv Lang \not\equiv 1\\ \lambda \not\equiv Lang \not\equiv 1\\ \lambda \not\equiv Lang \not\equiv 0\end{array}} \boxed{\begin{array}{l}[|\lambda|]Lang\\ [|0|]Lang\\ [|1|]Lang\end{array}}$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]Lang$ $([|[]|]Lang)$& Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0,1\}$ & $Lang = L(0{\cdot}(0|1)^*)$

$\boxed{[|\lambda|]Lang}$ $\xrightarrow{\lambda \not\equiv Lang \not\equiv 0}$ $\boxed{\begin{matrix}[|\lambda|]Lang\\ [|0|]Lang\end{matrix}}$ $\xrightarrow{\begin{matrix}0 \not\equiv Lang \not\equiv 1\\ \lambda \not\equiv Lang \not\equiv 1\\ \lambda \not\equiv Lang \not\equiv 0\end{matrix}}$ $\boxed{\begin{matrix}[|\lambda|]Lang\\ [|0|]Lang\\ [|1|]Lang\end{matrix}}$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]Lang$ ($[|[]|]Lang$)& Σ do a exhaustive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0, 1\}$ & $Lang = L(0{\cdot}(0|1)^*)$

$\boxed{[|\lambda|]Lang} \xrightarrow{\lambda \not\equiv Lang \not\equiv 0} \boxed{\begin{matrix}[|\lambda|]Lang \\ [|0|]Lang\end{matrix}} \xrightarrow{\begin{matrix}0 \not\equiv Lang \not\equiv 1 \\ \lambda \not\equiv Lang \not\equiv 1 \\ \lambda \not\equiv Lang \not\equiv 0\end{matrix}} \boxed{\begin{matrix}[|\lambda|]Lang \\ [|0|]Lang \\ [|1|]Lang\end{matrix}}$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]Lang$ ($[|[]|]Lang$)& Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0, 1\}$ & $Lang = L(0\cdot(0|1)^*)$

$\boxed{[|\lambda|]Lang}$ $\xrightarrow{\lambda \not\equiv Lang \not\equiv 0}$ $\boxed{\begin{array}{l}[|\lambda|]Lang\\ [|0|]Lang\end{array}}$ $\xrightarrow{\begin{array}{c}0 \not\equiv Lang \not\equiv 1\\ \lambda \not\equiv Lang \not\equiv 1\\ \lambda \not\equiv Lang \not\equiv 0\end{array}}$ $\boxed{\begin{array}{l}[|\lambda|]Lang\\ [|0|]Lang\\ [|1|]Lang\end{array}}$

Use and consequences

- To show a language is regular
  - prove the partition is finite
  - from $[|\lambda|]Lang$ ($[|[]|]Lang$)& Σ do a exhausitive search
- To show a language is not regular
  - prove the partition is infinite

$\Sigma = \{0, 1\}$ & $Lang = L(0 \cdot (0|1)^*)$

$\boxed{[|\lambda|]Lang} \xrightarrow{\lambda \not\equiv Lang \not\equiv 0} \boxed{\begin{matrix}[|\lambda|]Lang \\ [|0|]Lang\end{matrix}} \xrightarrow{\begin{matrix}0 \not\equiv Lang \not\equiv 1 \\ \lambda \not\equiv Lang \not\equiv 1 \\ \lambda \not\equiv Lang \not\equiv 0\end{matrix}} \boxed{\begin{matrix}[|\lambda|]Lang \\ [|0|]Lang \\ [|1|]Lang\end{matrix}}$

Proof(brief.)

Finite partitions $\longrightarrow$ Regular

- Exists a k, where k partitions (equiv-classes)
- We can get a DFA (Q,Σ,δ,q_0,F)
  - $Q = UNIV\ Quo\ Lang$
  - $\delta(p, a) = q$ iff exists a word $x \in p$ such that $x@a \in q$
  - $q_0 = [|[]|]Lang$
  - $q \in F$ iff exists a word $x \in q$ such that $x \in Lang$
  - δ is a function because:

    If $x \approx Lang \approx y$, then $(x@a) \approx Lang \approx (y@a)$
- For any string x, DFA ends in state $[|x|]Lang$
- $x \in Lang \longleftrightarrow DFA\ accepts$

Proof(brief.)

Finite partitions $\longrightarrow$ Regular

- Exists a k, where k partitions (equiv-classes)
- We can get a DFA (Q,Σ,δ,q_0,F)
  - $Q = UNIV\ \ Quo\ Lang$
  - $\delta(p, a) = q$ iff
    exists a word $x \in p$ such that $x@a \in q$
  - $q_0 = [|\lambda|]Lang$
  - $q \in F$ iff exists a word $x \in q$ such that $x \in Lang$
  - δ is a function because:
    If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)
- For any string x, DFA ends in state $[|x|]Lang$
- $x \in Lang \longleftrightarrow DFA\ accepts$

Proof(brief.)

Finite partitions $\longrightarrow$ Regular

- Exists a k, where k partitions (equiv-classes)
- We can get a DFA (Q,Σ,δ,q_0,F)
  - $Q = UNIV\ \ Quo\ Lang$
  - $\delta(p,a) = q$ iff
    exists a word $x \in p$ such that $x@a \in q$
  - $q_0 = [|\lambda|]Lang$
  - $q \in F$ iff exists a word $x \in q$ such that $x \in Lang$
  - δ is a function because:

    If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)
- For any string x, DFA ends in state $[|x|]Lang$
- $x \in Lang \longleftrightarrow DFA\ accepts$

Proof(brief.)

Finite partitions $\longrightarrow$ Regular

- Exists a k, where k partitions (equiv-classes)
- We can get a DFA (Q,Σ,δ,q_0,F)
  - $Q = UNIV\ Quo\ Lang$
  - $\delta(p, a) = q$ iff
    exists a word $x \in p$ such that $x@a \in q$
  - $q_0 = [|\lambda|]Lang$
  - $q \in F$ iff exists a word $x \in q$ such that $x \in Lang$
  - δ is a function because:
    If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)
- For any string x, DFA ends in state $[|x|]Lang$
- $x \in Lang \longleftrightarrow DFA\ accepts$

Proof(brief.)

Finite partitions $\longrightarrow$ Regular

- Exists a k, where k partitions (equiv-classes)
- We can get a DFA $(Q, \Sigma, \delta, q_0, F)$
  - $Q = UNIV\ Quo\ Lang$
  - $\delta(p, a) = q$ iff
    exists a word $x \in p$ such that $x@a \in q$
  - $q_0 = [|\lambda|]Lang$
  - $q \in F$ iff exists a word $x \in q$ such that $x \in Lang$
  - δ is a function because:
    If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)
- For any string x, DFA ends in state $[|x|]Lang$
- $x \in Lang \longleftrightarrow DFA\ accepts$

Proof(brief.)

Finite partitions $\longrightarrow$ Regular

- Exists a k, where k partitions (equiv-classes)
- We can get a DFA (Q,Σ,δ,q_0,F)
  - $Q = UNIV\ Quo\ Lang$
  - $\delta(p, a) = q$ iff
    exists a word $x \in p$ such that $x@a \in q$
  - $q_0 = [|\lambda|]Lang$
  - $q \in F$ iff exists a word $x \in q$ such that $x \in Lang$
  - δ is a function because:
    If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)
- For any string x, DFA ends in state $[|x|]Lang$
- $x \in Lang \longleftrightarrow DFA\ accepts$

Proof(brief.)

Finite partitions $\longrightarrow$ Regular

- Exists a k, where k partitions (equiv-classes)
- We can get a DFA $(Q, \Sigma, \delta, q_0, F)$
  - $Q = UNIV\ Quo\ Lang$
  - $\delta(p, a) = q$ iff
    exists a word $x \in p$ such that $x@a \in q$
  - $q_0 = [|\lambda|]Lang$
  - $q \in F$ iff exists a word $x \in q$ such that $x \in Lang$
  - δ is a function because:
    If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)
- For any string x, DFA ends in state $[|x|]Lang$
- $x \in Lang \longleftrightarrow DFA\ accepts$

Proof(brief.)

Finite partitions $\longrightarrow$ Regular

- Exists a k, where k partitions (equiv-classes)
- We can get a DFA (Q,Σ,δ,q_0,F)
  - $Q = UNIV\ Quo\ Lang$
  - $\delta(p, a) = q$ iff
    exists a word $x \in p$ such that $x@a \in q$
  - $q_0 = [|\lambda|]Lang$
  - $q \in F$ iff exists a word $x \in q$ such that $x \in Lang$
  - δ is a function because:
    If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)
- For any string x, DFA ends in state $[|x|]Lang$
- $x \in Lang \longleftrightarrow DFA\ accepts$

Proof(brief.)

Finite partitions $\longrightarrow$ Regular

- Exists a k, where k partitions (equiv-classes)
- We can get a DFA $(Q, \Sigma, \delta, q_0, F)$
    - $Q = UNIV\ Quo\ Lang$
    - $\delta(p, a) = q$ iff
      exists a word $x \in p$ such that $x@a \in q$
    - $q_0 = [|\lambda|]Lang$
    - $q \in F$ iff exists a word $x \in q$ such that $x \in Lang$
    - δ is a function because:
      If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)
- For any string x, DFA ends in state $[|x|]Lang$
- $x \in Lang \longleftrightarrow DFA\ accepts$

Proof(brief.)

Finite partitions $\longrightarrow$ Regular

- Exists a k, where k partitions (equiv-classes)
- We can get a DFA $(Q, \Sigma, \delta, q_0, F)$
  - $Q = UNIV\ Quo\ Lang$
  - $\delta(p, a) = q$ iff
    exists a word $x \in p$ such that $x@a \in q$
  - $q_0 = [|\lambda|]Lang$
  - $q \in F$ iff exists a word $x \in q$ such that $x \in Lang$
  - δ is a function because:

    If x ≡Lang≡ y, then (x@a) ≡Lang≡ (y@a)
- For any string x, DFA ends in state $[|x|]Lang$
- $x \in Lang \longleftrightarrow DFA\ accepts$

Proof (cont.)

Regular $\longrightarrow$ Finite partitions

- $x \equiv DFA \equiv y$ iff x & y end in the same state
- $\equiv DFA \equiv$ is an equivalence relation
- $x \equiv DFA \equiv y \longrightarrow x \equiv Lang \equiv y$
- Finite DFA

Regular $\longleftrightarrow$ **Finite partitions**

But if **Regular** is defined in **Reg Exps**, then **?**

Proof (cont.)

Regular $\longrightarrow$ Finite partitions

- $x \equiv DFA \equiv y$ iff x & y end in the same state
- $\equiv DFA \equiv$ is an equivalence relation
- $x \equiv DFA \equiv y \longrightarrow x \equiv Lang \equiv y$
- Finite DFA

Regular $\longleftrightarrow$ **Finite partitions**

But if **Regular** is defined in **Reg Exps**, then **?**

Proof (cont.)

Regular $\longrightarrow$ Finite partitions

- $x \equiv DFA \equiv y$ iff x & y end in the same state
- $\equiv DFA \equiv$ is an equivalence relation
- $x \equiv DFA \equiv y \longrightarrow x \equiv Lang \equiv y$
- Finite DFA

Regular $\longleftrightarrow$ **Finite partitions**

But if **Regular** is defined in **Reg Exps**, then **?**

Proof (cont.)

Regular $\longrightarrow$ Finite partitions

- $x \equiv DFA \equiv y$ iff x & y end in the same state
- $\equiv DFA \equiv$ is an equivalence relation
- $x \equiv DFA \equiv y \longrightarrow x \equiv Lang \equiv y$
- Finite DFA

Regular $\longleftrightarrow$ **Finite partitions**

But if **Regular** is defined in **Reg Exps**, then **?**

Proof (cont.)

Regular $\longrightarrow$ Finite partitions

- $x \equiv DFA \equiv y$ iff x & y end in the same state
- $\equiv DFA \equiv$ is an equivalence relation
- $x \equiv DFA \equiv y \longrightarrow x \equiv Lang \equiv y$
- Finite DFA

Regular $\longleftrightarrow$ **Finite partitions**

But if **Regular** is defined in **Reg Exps**, then **?**

Proof (cont.)

Regular $\longrightarrow$ Finite partitions

- $x \equiv DFA \equiv y$ iff x & y end in the same state
- $\equiv DFA \equiv$ is an equivalence relation
- $x \equiv DFA \equiv y \longrightarrow x \equiv Lang \equiv y$
- Finite DFA

Regular $\longleftrightarrow$ **Finite partitions**

But if **Regular** is defined in **Reg Exps**, then **?**

Proof (cont.)

Regular $\longrightarrow$ Finite partitions

- $x \equiv DFA \equiv y$ iff x & y end in the same state
- $\equiv DFA \equiv$ is an equivalence relation
- $x \equiv DFA \equiv y \longrightarrow x \equiv Lang \equiv y$
- Finite DFA

Regular $\longleftrightarrow$ Finite partitions

But if **Regular** is defined in **Reg Exps**, then **?**

Proof (cont.)

Regular $\longrightarrow$ Finite partitions

- $x \equiv DFA \equiv y$ iff x & y end in the same state
- $\equiv DFA \equiv$ is an equivalence relation
- $x \equiv DFA \equiv y \longrightarrow x \equiv Lang \equiv y$
- Finite DFA

Regular $\longleftrightarrow$ Finite partitions

But if **Regular** is defined in **Reg Exps**, then **?**

Outline

FA to Reg Exps

State Removal method

- How to do?
- Characters

FA to Reg Exps

State Removal method

- How to do?
  1. Identifies patterns within the graph
  2. Removes states
  3. Builds up bigger regular exps
- Characters
  1. Easy to visualize
  2. Hard to formalize
     - Simplified patterns in textbook
     - How to choose patterns ?

FA to Reg Exps

State Removal method

- How to do?
  1. Identifies patterns within the graph
  2. Removes states
  3. Builds up bigger regular exps
- Characters
  1. Easy to visualize
  2. Hard to formalize
     - How to choose patterns?

FA to Reg Exps

State Removal method

- How to do?
  1. Identifies patterns within the graph
  2. Removes states
  3. Builds up bigger regular exps
- Characters
  1. Easy to visualize
  2. Hard to formalize
     - Simplified patterns in textbook
     - How to choose patterns ?

FA to Reg Exps

State Removal method

- How to do?
  1. Identifies patterns within the graph
  2. Removes states
  3. Builds up bigger regular exps
- Characters
  1. Easy to visualize
  2. Hard to formalize
     - Simplified patterns in textbook
     - How to choose patterns ?

FA to Reg Exps

State Removal method

- How to do?
  1. Identifies patterns within the graph
  2. Removes states
  3. Builds up bigger regular exps
- Characters
  1. Easy to visualize
  2. Hard to formalize
     - Simplified patterns in textbook
     - How to choose patterns ?

FA to Reg Exps

State Removal method

- How to do?
  1. Identifies patterns within the graph
  2. Removes states
  3. Builds up bigger regular exps
- Characters
  1. Easy to visualize
  2. Hard to formalize
     - Simplified patterns in textbook
     - How to choose patterns ?

FA to Reg Exps

State Removal method

- How to do?
  1. Identifies patterns within the graph
  2. Removes states
  3. Builds up bigger regular exps
- Characters
  1. Easy to visualize
  2. Hard to formalize
     - Simplified patterns in textbook
     - How to choose patterns ?

FA to Reg Exps

State Removal method

- How to do?
  1. Identifies patterns within the graph
  2. Removes states
  3. Builds up bigger regular exps
- Characters
  1. Easy to visualize
  2. Hard to formalize
     - Simplified patterns in textbook
     - How to choose patterns ?

FA to Reg Exps

State Removal method

- How to do?
  1. Identifies patterns within the graph
  2. Removes states
  3. Builds up bigger regular exps
- Characters
  1. Easy to visualize
  2. Hard to formalize
     - Simplified patterns in textbook
     - How to choose patterns ?

FA to Reg Exps (cont.)

Transitive Clousre method

- Easy to formalize
- We have done it

Brozozowski Algebraic method

FA to Reg Exps (cont.)

Transitive Clousre method

- Easy to formalize
- We have done it

Brozozowski Algebraic method

FA to Reg Exps (cont.)

Transitive Clousre method

- Easy to formalize
- We have done it

Brozozowski Algebraic method

FA to Reg Exps (cont.)

Transitive Clousre method

- Easy to formalize
- We have done it

Brozozowski Algebraic method

FA to Reg Exps (cont.)

Transitive Clousre method

- Easy to formalize
- We have done it

Brozozowski Algebraic method

Brzozowski Algebraic method (revised.)

Example 1

Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

$+$ is actually $\cup$

ES Subst (1)

$R_2 = \lambda; 0 + R_2; (0|1)$

$R_3 = \lambda; 1 + R_3; (0|1)$

ES Arden (2)

$R_2 = (\lambda; 0)(0|1)^* = \lambda; (0 \cdot (0|1)^*)$

$R_3 = \lambda; 1 + R_3; (0|1)$

Result

$R_2 : (0 \cdot (0|1)^*)$

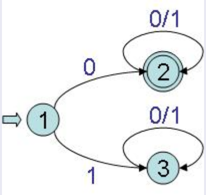
Arden's Lemma (revised.)

Given an equation of the form $X = X; A + B$ when $[] \notin A$ the equation has the solution $X = B; A*$

Brzozowski Algebraic method (revised.)

Example 1

DFA



Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

$+$ is actually $\cup$

ES Subst (1)

$R_2 = \lambda; 0 + R_2; (0|1)$

$R_3 = \lambda; 1 + R_3; (0|1)$

ES Arden (2)

$R_2 = (\lambda; 0)(0|1)^* = \lambda; (0 \cdot (0|1)^*)$

$R_3 = \lambda; 1 + R_3; (0|1)$

Result

$R_2 : (0 \cdot (0|1)^*)$

Arden's Lemma (revised.)

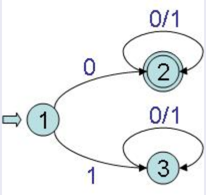
Given an equation of the form $X = X; A + B$ where $[] \notin A$,

the equation has the solution $X = B; A^*$

Brzozowski Algebraic method (revised.)

Example 1

DFA



Equation-System (0)
$R_1 = \lambda$
$R_2 = R_1; 0 + R_2; (0|1)$
$R_3 = R_1; 1 + R_3; (0|1)$
$+$ is actually $\cup$

ES Subst (1)
$R_2 = \lambda; 0 + R_2; (0|1)$
$R_3 = \lambda; 1 + R_3; (0|1)$

ES Arden (2)
$R_2 = (\lambda; 0)(0|1)^* = \lambda; (0 \cdot (0|1)^*)$
$R_3 = \lambda; 1 + R_3; (0|1)$

Result
$R_2 : (0 \cdot (0|1)^*)$

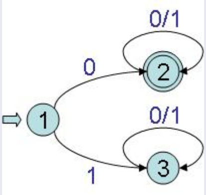
Arden's Lemma (revised.)

Given an equation of the form $X = X; A + B$ where $[] \notin A$, the equation has the solution $X = B; A*$

Brzozowski Algebraic method (revised.)

Example 1

DFA



Equation-System (0)
$R_1 = \lambda$
$R_2 = R_1; 0 + R_2; (0|1)$
$R_3 = R_1; 1 + R_3; (0|1)$
$+$ is actually $\cup$

ES Subst (1)
$R_2 = \lambda; 0 + R_2; (0|1)$
$R_3 = \lambda; 1 + R_3; (0|1)$

ES Arden (2)
$R_2 = (\lambda; 0)(0|1)^* = \lambda; (0 \cdot (0|1)^*)$
$R_3 = \lambda; 1 + R_3; (0|1)$

Result
$R_2 : (0 \cdot (0|1)^*)$

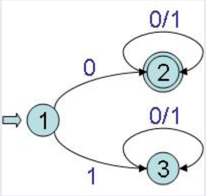
Arden's Lemma (revised.)

Given an equation of the form $X = X; A + B$ where $[] \notin A$, the equation has the solution $X = B; A*$

Brzozowski Algebraic method (revised.)

Example 1

DFA



Equation-System (0)
$R_1 = \lambda$
$R_2 = R_1; 0 + R_2; (0|1)$
$R_3 = R_1; 1 + R_3; (0|1)$
$+$ is actually $\cup$

ES Subst (1)
$R_2 = \lambda; 0 + R_2; (0|1)$
$R_3 = \lambda; 1 + R_3; (0|1)$

ES Arden (2)
$R_2 = (\lambda; 0)(0|1)^* = \lambda; (0 \cdot (0|1)^*)$
$R_3 = \lambda; 1 + R_3; (0|1)$

Result
$R_2 : (0 \cdot (0|1)^*)$

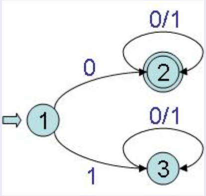
Arden's Lemma (revised.)

Given an equation of the form $X = X; A + B$ where $[] \notin A$, the equation has the solution $X = B; A*$

Brzozowski Algebraic method (revised.)

Example 1

DFA



Equation-System (0)
$R_1 = \lambda$
$R_2 = R_1; 0 + R_2; (0|1)$
$R_3 = R_1; 1 + R_3; (0|1)$
$+$ is actually $\cup$

ES Subst (1)
$R_2 = \lambda; 0 + R_2; (0|1)$
$R_3 = \lambda; 1 + R_3; (0|1)$

ES Arden (2)
$R_2 = (\lambda; 0)(0|1)^* = \lambda; (0 \cdot (0|1)^*)$
$R_3 = \lambda; 1 + R_3; (0|1)$

Result
$R_2 : (0 \cdot (0|1)^*)$

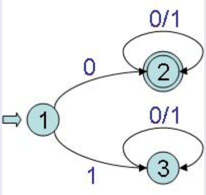
Arden's Lemma (revised.)

Given an equation of the form $X = X; A + B$ where $[] \notin A$, the equation has the solution $X = B; A*$

Brzozowski Algebraic method (revised.)

Example 1

DFA



Equation-System (0)
$R_1 = \lambda$
$R_2 = R_1; 0 + R_2; (0|1)$
$R_3 = R_1; 1 + R_3; (0|1)$
$+$ is actually $\cup$

ES Subst (1)
$R_2 = \lambda; 0 + R_2; (0|1)$
$R_3 = \lambda; 1 + R_3; (0|1)$

ES Arden (2)
$R_2 = (\lambda; 0)(0|1)^* = \lambda; (0 \cdot (0|1)^*)$
$R_3 = \lambda; 1 + R_3; (0|1)$

Result
$R_2 : (0 \cdot (0|1)^*)$

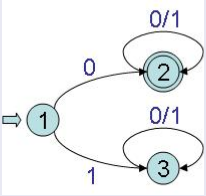
Arden's Lemma (revised.)

Given an equation of the form $X = X; A + B$ where $[] \notin A$, the equation has the solution $X = B; A*$

Brzozowski Algebraic method (revised.)

Example 1

DFA



Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

$+$ is actually $\cup$

ES Subst (1)

$R_2 = \lambda; 0 + R_2; (0|1)$

$R_3 = \lambda; 1 + R_3; (0|1)$

ES Arden (2)

$R_2 = (\lambda; 0)(0|1)^* = \lambda; (0 \cdot (0|1)^*)$

$R_3 = \lambda; 1 + R_3; (0|1)$

Result

$R_2 : (0 \cdot (0|1)^*)$

Arden's Lemma (revised.)

Given an equation of the form $X = X; A + B$ where $[] \notin A$, the equation has the solution $X = B; A*$

Brzozowski Algebraic method (revised.)

Example 2

Equation-System (0)

$R_1 = R_1; 1 + R_2; 1 + \lambda$

$R_2 = R_1; 0 + R_2; 0$

ES Arden (1)

$R_1 = R_2; (1 \cdot 1^*) + \lambda; 1^*$

$R_2 = R_1; 0 + R_2; 0$

ES Subst (2)

$R_2 = R_2; ((1 \cdot 1^* \cdot 0)|0) + \lambda; (1^* \cdot 0)$

ES Arden (3)

$R_2 = \lambda; (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

Result

$R_2 : (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

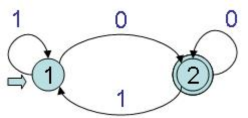
Arden's Lemma

$X = AX + B$ where $[] \notin A$, solution: $X = A * B$

Brzozowski Algebraic method (revised.)

Example 2

DFA



Equation-System (0)

$R_1 = R_1;1 + R_2;1 + \lambda$

$R_2 = R_1;0 + R_2;0$

ES Arden (1)

$R_1 = R_2;(1 \cdot 1^*) + \lambda;1^*$

$R_2 = R_1;0 + R_2;0$

ES Subst (2)

$R_2 = R_2;((1 \cdot 1^* \cdot 0)|0) + \lambda;(1^* \cdot 0)$

ES Arden (3)

$R_2 = \lambda;(1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

Result

$R_2 : (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

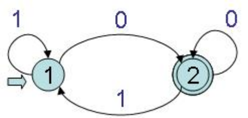
Arden's Lemma

$X = AX + B$ where $[] \notin A$, solution: $X = A * B$

Brzozowski Algebraic method (revised.)

Example 2

DFA



Equation-System (0)

$R_1 = R_1; 1 + R_2; 1 + \lambda$
$R_2 = R_1; 0 + R_2; 0$

ES Arden (1)

$R_1 = R_2; (1 \cdot 1^*) + \lambda; 1^*$
$R_2 = R_1; 0 + R_2; 0$

ES Subst (2)

$R_2 = R_2; ((1 \cdot 1^* \cdot 0)|0) + \lambda; (1^* \cdot 0)$

ES Arden (3)

$R_2 = \lambda; (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

Result

$R_2 : (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

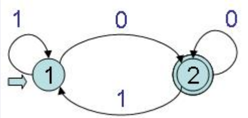
Arden's Lemma

$X = AX + B$ where $[] \notin A$, solution: $X = A * B$

Brzozowski Algebraic method (revised.)

Example 2

DFA



Equation-System (0)

$R_1 = R_1; 1 + R_2; 1 + \lambda$
$R_2 = R_1; 0 + R_2; 0$

ES Arden (1)

$R_1 = R_2; (1 \cdot 1^*) + \lambda; 1^*$
$R_2 = R_1; 0 + R_2; 0$

ES Subst (2)

$R_2 = R_2; ((1 \cdot 1^* \cdot 0)|0) + \lambda; (1^* \cdot 0)$

ES Arden (3)

$R_2 = \lambda; (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

Result

$R_2 : (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

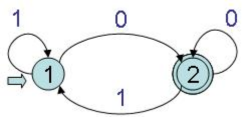
Arden's Lemma

$X = AX + B$ where $[] \notin A$, solution: $X = A * B$

Brzozowski Algebraic method (revised.)

Example 2

DFA



Equation-System (0)

$R_1 = R_1; 1 + R_2; 1 + \lambda$
$R_2 = R_1; 0 + R_2; 0$

ES Arden (1)

$R_1 = R_2; (1 \cdot 1^*) + \lambda; 1^*$
$R_2 = R_1; 0 + R_2; 0$

ES Subst (2)

$R_2 = R_2; ((1 \cdot 1^* \cdot 0)|0) + \lambda; (1^* \cdot 0)$

ES Arden (3)

$R_2 = \lambda; (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

Result

$R_2 : (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

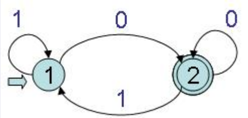
Arden's Lemma

$X = AX + B$ where $[] \notin A$, solution: $X = A * B$

Brzozowski Algebraic method (revised.)

Example 2

DFA



Equation-System (0)

$R_1 = R_1; 1 + R_2; 1 + \lambda$
$R_2 = R_1; 0 + R_2; 0$

ES Arden (1)

$R_1 = R_2; (1 \cdot 1^*) + \lambda; 1^*$
$R_2 = R_1; 0 + R_2; 0$

ES Subst (2)

$R_2 = R_2; ((1 \cdot 1^* \cdot 0)|0) + \lambda; (1^* \cdot 0)$

ES Arden (3)

$R_2 = \lambda; (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

Result

$R_2 : (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

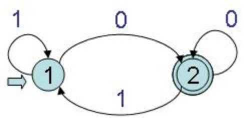
Arden's Lemma

$X = AX + B$ where $[] \notin A$, solution: $X = A * B$

Brzozowski Algebraic method (revised.)

Example 2

DFA



Equation-System (0)
$R_1 = R_1; 1 + R_2; 1 + \lambda$
$R_2 = R_1; 0 + R_2; 0$

ES Arden (1)
$R_1 = R_2; (1 \cdot 1^*) + \lambda; 1^*$
$R_2 = R_1; 0 + R_2; 0$

ES Subst (2)
$R_2 = R_2; ((1 \cdot 1^* \cdot 0)|0) + \lambda; (1^* \cdot 0)$

ES Arden (3)
$R_2 = \lambda; (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

Result
$R_2 : (1^* \cdot 0 \cdot ((1 \cdot 1^* \cdot 0)|0)^*)$

Arden's Lemma

$X = AX + B$ where $[] \notin A$, solution: $X = A * B$

Outline

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow$ $\exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
      1. Language of left(equiv-class) equal with right
      2. Right of last equation: λ; (*reg*)
      3. Language of λ is {[]}
      4. Language of right is *L reg*
      5. We find the Reg Exps for the equiv-class!
      6. *Lang* is a set of equiv-class
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
      1. Language of left(equiv-class) equal with right
      2. Right of last equation: λ; (reg)
      3. Language of λ is $\{[]\}$
      4. Language of right is *L reg*
      5. We find the Reg Exps for the equiv-class!
      6. *Lang* is a set of equiv-class
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow$ $\exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
      1. Language of left(equiv-class) equal with right
      2. Right of last equation: λ; (*reg*)
      3. Language of λ is $\{[]\}$
      4. Language of right is *L reg*
      5. We find the Reg Exps for the equiv-class!
      6. *Lang* is a set of equiv-class
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
      1. Language of left(equiv-class) equal with right
      2. Right of last equation: $\lambda; (reg)$
      3. Language of λ is $\{[]\}$
      4. Language of right is *L reg*
      5. We find the Reg Exps for the equiv-class!
      6. *Lang* is a set of equiv-class
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
      1. Language of left(equiv-class) equal with right
      2. Right of last equation: λ; (reg)
      3. Language of λ is $\{[]\}$
      4. Language of right is *L reg*
      5. We find the Reg Exps for the equiv-class!
      6. *Lang* is a set of equiv-class
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
      1. Language of left(equiv-class) equal with right
      2. Right of last equation: λ; (*reg*)
      3. Language of λ is {[]}
      4. Language of right is *L reg*
      5. We find the Reg Exps for the equiv-class!
      6. *Lang* is a set of equiv-class
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow$ $\exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
      1. Language of left(equiv-class) equal with right
      2. Right of last equation: λ; (*reg*)
      3. Language of λ is $\{[]\}$
      4. Language of right is *L reg*
      5. We find the Reg Exps for the equiv-class!
      6. *Lang* is a set of equiv-class
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow$ $\exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
      1. Language of left(equiv-class) equal with right
      2. Right of last equation: $\lambda; (reg)$
      3. Language of λ is $\{[]\}$
      4. Language of right is *L reg*
      5. We find the Reg Exps for the equiv-class!
      6. *Lang* is a set of equiv-class
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
      1. Language of left(equiv-class) equal with right
      2. Right of last equation: λ; *(reg)*
      3. Language of λ is {[]}
      4. Language of right is *L reg*
      5. We find the Reg Exps for the equiv-class!
      6. *Lang* is a set of equiv-class
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Proving thought Reg Exps

- Target: finite (UNIV Quo Lang) $\Longrightarrow \exists$ reg. Lang = L reg
- Main approach
  - Generate initial ES derived from *Lang*
  - Fetch the Reg Exp by Brozozowski method
  - How to prove?
    - If each equation in ES of every step has:
      1. Language of left(equiv-class) equal with right
      2. Right of last equation: $\lambda; (reg)$
      3. Language of λ is $\{[]\}$
      4. Language of right is *L reg*
      5. We find the Reg Exps for the equiv-class!
      6. *Lang* is a set of equiv-class
    - Based on an well-founded iterating principle
    - Invariant of each step of ES' invariation

Outline

WF-iter

- Elimination of ES can be abstracted as

WF-iter

- Elimination of ES can be abstracted as

$$\frac{\text{P e}}{\exists \text{e'}.\ \text{P e'} \wedge \text{Q e'}}$$

WF-iter

- Elimination of ES can be abstracted as

$$\frac{\text{P e}}{\exists \text{e'}.\ \text{P e'} \wedge \text{Q e'}}$$

- Like while in C

WF-iter

- Elimination of ES can be abstracted as

$$\frac{\mathrm{P\ e}}{\exists \mathrm{e'}.\ \mathrm{P\ e'} \wedge \mathrm{Q\ e'}}$$

- Like while in C
- Property Q: termination condition

WF-iter

- Elimination of ES can be abstracted as

$$\frac{\text{P e}}{\exists \text{e'}.\ \text{P e'} \wedge \text{Q e'}}$$

- Like while in C
- Property Q: termination condition

$$\text{TCon ES} \equiv \text{card ES} = 1$$

WF-iter

- Elimination of ES can be abstracted as

  $$\frac{\text{P e}}{\exists \text{e'}.\ \text{P e'} \wedge \text{Q e'}}$$

- Like while in C
- Property Q: termination condition

  $$\text{TCon ES} \equiv \text{card ES} = 1$$

- Property P is an invariant predicate

WF-iter

- Elimination of ES can be abstracted as

  $$\frac{\text{P e}}{\exists \text{e'}.\ \text{P e'} \wedge \text{Q e'}}$$

- Like while in C
- Property Q: termination condition

  $$\text{TCon ES} \equiv \text{card ES} = 1$$

- Property P is an invariant predicate
  - What is invariant?

WF-iter

- Elimination of ES can be abstracted as

$$\frac{\text{P e}}{\exists \text{e'}.\ \text{P e'} \wedge \text{Q e'}}$$

- Like while in C
- Property Q: termination condition

$$\text{TCon ES} \equiv \text{card ES} = 1$$

- Property P is an invariant predicate
  - What is invariant?
    1. Language of left equal with right

WF-iter

- Elimination of ES can be abstracted as

$$\frac{\text{P e}}{\exists \text{e'}.\ \text{P e'} \wedge \text{Q e'}}$$

- Like while in C
- Property Q: termination condition

$$\text{TCon ES} \equiv \text{card ES} = 1$$

- Property P is an invariant predicate
  - What is invariant?
    1. Language of left equal with right
    2. ES is finite

WF-iter

- Elimination of ES can be abstracted as

$$\frac{\text{P e}}{\exists \text{e'}.\ \text{P e'} \wedge \text{Q e'}}$$

- Like while in C
- Property Q: termination condition

$$\text{TCon ES} \equiv \text{card ES} = 1$$

- Property P is an invariant predicate
  - What is invariant?
    1. Language of left equal with right
    2. ES is finite
    3. Each equiv-class has only one equation

WF-iter

- Elimination of ES can be abstracted as

$$\frac{\text{P e}}{\exists \text{e'. P e'} \wedge \text{Q e'}}$$

- Like while in C
- Property Q: termination condition

$$\text{TCon ES} \equiv \text{card ES} = 1$$

- Property P is an invariant predicate
  - What is invariant?
    1. Language of left equal with right
    2. ES is finite
    3. Each equiv-class has only one equation
    4. Target equiv-class exists

WF-iter

- Elimination of ES can be abstracted as

$$\frac{\text{P e}}{\exists \text{e'. P e'} \wedge \text{Q e'}}$$

- Like while in C
- Property Q: termination condition

$$\text{TCon ES} \equiv \text{card ES} = 1$$

- Property P is an invariant predicate
  - What is invariant?
    1. Language of left equal with right
    2. ES is finite
    3. Each equiv-class has only one equation
    4. Target equiv-class exists
    5. ...

Outline

Formalization of Inv

```
definition
  distinct_rhs :: "t_equa_rhs ⇒ bool"
where "distinct_rhs rhs ≡ ∀ X reg₁ reg₂.
        (X, reg₁) ∈ rhs ∧ (X, reg₂) ∈ rhs ⟶ reg₁ = reg₂"
definition
  no_EMPTY_rhs :: "t_equa_rhs ⇒ bool"
where "no_EMPTY_rhs rhs ≡ ∀ X r.
        (X, r) ∈ rhs ∧ X ≠ {[]} ⟶ [] ∉ L r"

definition ardenable :: "t_equa ⇒ bool"
where "ardenable equa ≡ let (X, rhs) = equa in
        distinct_rhs rhs ∧ no_EMPTY_rhs rhs ∧ X = L rhs"
definition distinct_equas :: "t_equas ⇒ bool"
where
  "distinct_equas equas ≡ ∀ X rhs rhs'.
    (X, rhs) ∈ equas ∧ (X, rhs') ∈ equas ⟶ rhs = rhs'"
definition left_eq_cls :: "t_equas ⇒ (string set) set"
where "left_eq_cls ES ≡ {X. ∃ rhs. (X, rhs) ∈ ES} "
definition rhs_eq_cls :: "t_equa_rhs ⇒ (string set) set"
where "rhs_eq_cls rhs ≡ {Y. ∃ r. (Y, r) ∈ rhs}"
definition Inv :: "string set ⇒ t_equas ⇒ bool"
where
  "Inv X ES ≡ finite ES ∧ (∃ rhs. (X, rhs) ∈ ES) ∧ distinct_equas ES ∧
    (∀ X xrhs. (X, xrhs) ∈ ES ⟶ ardenable (X, xrhs) ∧ X ≠ {} ∧

    rhs_eq_cls xrhs ⊆ insert {[]} (left_eq_cls ES))"
```

Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

Formalization of Inv

```
definition
  distinct_rhs :: "t_equa_rhs ⇒ bool"
where "distinct_rhs rhs ≡ ∀ X reg₁ reg₂.
        (X, reg₁) ∈ rhs ∧ (X, reg₂) ∈ rhs ⟶ reg₁ = reg₂"
definition
  no_EMPTY_rhs :: "t_equa_rhs ⇒ bool"
where "no_EMPTY_rhs rhs ≡ ∀ X r.
        (X, r) ∈ rhs ∧ X ≠ {[]} ⟶ [] ∉ L r"

definition ardenable :: "t_equa ⇒ bool"
where "ardenable equa ≡ let (X, rhs) = equa in
        distinct_rhs rhs ∧ no_EMPTY_rhs rhs ∧ X = L rhs"
definition distinct_equas :: "t_equas ⇒ bool"
where
  "distinct_equas equas ≡ ∀ X rhs rhs'.
     (X, rhs) ∈ equas ∧ (X, rhs') ∈ equas ⟶ rhs = rhs'"
definition left_eq_cls :: "t_equas ⇒ (string set) set"
where "left_eq_cls ES ≡ {X. ∃ rhs. (X, rhs) ∈ ES} "
definition rhs_eq_cls :: "t_equa_rhs ⇒ (string set) set"
where "rhs_eq_cls rhs ≡ {Y. ∃ r. (Y, r) ∈ rhs}"
definition Inv :: "string set ⇒ t_equas ⇒ bool"
where
  "Inv X ES ≡ finite ES ∧ (∃ rhs. (X, rhs) ∈ ES) ∧ distinct_equas ES ∧
     (∀ X xrhs. (X, xrhs) ∈ ES ⟶ ardenable (X, xrhs) ∧ X ≠ {} ∧

     rhs_eq_cls xrhs ⊆ insert {[]} (left_eq_cls ES))"
```

Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

Formalization of Inv

```
definition
  distinct_rhs :: "t_equa_rhs ⇒ bool"
where "distinct_rhs rhs ≡ ∀ X reg₁ reg₂.
        (X, reg₁) ∈ rhs ∧ (X, reg₂) ∈ rhs ⟶ reg₁ = reg₂"
definition
  no_EMPTY_rhs :: "t_equa_rhs ⇒ bool"
where "no_EMPTY_rhs rhs ≡ ∀ X r.
        (X, r) ∈ rhs ∧ X ≠ {[]} ⟶ [] ∉ L r"

definition ardenable :: "t_equa ⇒ bool"
where "ardenable equa ≡ let (X, rhs) = equa in
        distinct_rhs rhs ∧ no_EMPTY_rhs rhs ∧ X = L rhs"
definition distinct_equas :: "t_equas ⇒ bool"
where
  "distinct_equas equas ≡ ∀ X rhs rhs'.
     (X, rhs) ∈ equas ∧ (X, rhs') ∈ equas ⟶ rhs = rhs'"
definition left_eq_cls :: "t_equas ⇒ (string set) set"
where "left_eq_cls ES ≡ {X. ∃ rhs. (X, rhs) ∈ ES} "
definition rhs_eq_cls :: "t_equa_rhs ⇒ (string set) set"
where "rhs_eq_cls rhs ≡ {Y. ∃ r. (Y, r) ∈ rhs}"
definition Inv :: "string set ⇒ t_equas ⇒ bool"
where
  "Inv X ES ≡ finite ES ∧ (∃ rhs. (X, rhs) ∈ ES) ∧ distinct_equas ES ∧
     (∀ X xrhs. (X, xrhs) ∈ ES ⟶ ardenable (X, xrhs) ∧ X ≠ {} ∧

     rhs_eq_cls xrhs ⊆ insert {[]} (left_eq_cls ES))"
```

Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

Formalization of Inv

```
definition
  distinct_rhs :: "t_equa_rhs ⇒ bool"
where "distinct_rhs rhs ≡ ∀ X reg₁ reg₂.
        (X, reg₁) ∈ rhs ∧ (X, reg₂) ∈ rhs ⟶ reg₁ = reg₂"
definition
  no_EMPTY_rhs :: "t_equa_rhs ⇒ bool"
where "no_EMPTY_rhs rhs ≡ ∀ X r.
        (X, r) ∈ rhs ∧ X ≠ {[]} ⟶ [] ∉ L r"

definition ardenable :: "t_equa ⇒ bool"
where "ardenable equa ≡ let (X, rhs) = equa in
        distinct_rhs rhs ∧ no_EMPTY_rhs rhs ∧ X = L rhs"
definition distinct_equas :: "t_equas ⇒ bool"
where
  "distinct_equas equas ≡ ∀ X rhs rhs'.
    (X, rhs) ∈ equas ∧ (X, rhs') ∈ equas ⟶ rhs = rhs'"
definition left_eq_cls :: "t_equas ⇒ (string set) set"
where "left_eq_cls ES ≡ {X. ∃ rhs. (X, rhs) ∈ ES} "
definition rhs_eq_cls :: "t_equa_rhs ⇒ (string set) set"
where "rhs_eq_cls rhs ≡ {Y. ∃ r. (Y, r) ∈ rhs}"
definition Inv :: "string set ⇒ t_equas ⇒ bool"
where
  "Inv X ES ≡ finite ES ∧ (∃ rhs. (X, rhs) ∈ ES) ∧ distinct_equas ES ∧
    (∀ X xrhs. (X, xrhs) ∈ ES ⟶ ardenable (X, xrhs) ∧ X ≠ {} ∧


    rhs_eq_cls xrhs ⊆ insert {[]} (left_eq_cls ES))"
```

Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

Formalization of Inv

```
definition
  distinct_rhs :: "t_equa_rhs ⇒ bool"
where "distinct_rhs rhs ≡ ∀ X reg₁ reg₂.
       (X, reg₁) ∈ rhs ∧ (X, reg₂) ∈ rhs ⟶ reg₁ = reg₂"
definition
  no_EMPTY_rhs :: "t_equa_rhs ⇒ bool"
where "no_EMPTY_rhs rhs ≡ ∀ X r.
       (X, r) ∈ rhs ∧ X ≠ {[]} ⟶ [] ∉ L r"

definition ardenable :: "t_equa ⇒ bool"
where "ardenable equa ≡ let (X, rhs) = equa in
       distinct_rhs rhs ∧ no_EMPTY_rhs rhs ∧ X = L rhs"
definition distinct_equas :: "t_equas ⇒ bool"
where
  "distinct_equas equas ≡ ∀ X rhs rhs'.
    (X, rhs) ∈ equas ∧ (X, rhs') ∈ equas ⟶ rhs = rhs'"
definition left_eq_cls :: "t_equas ⇒ (string set) set"
where "left_eq_cls ES ≡ {X. ∃ rhs. (X, rhs) ∈ ES} "
definition rhs_eq_cls :: "t_equa_rhs ⇒ (string set) set"
where "rhs_eq_cls rhs ≡ {Y. ∃ r. (Y, r) ∈ rhs}"
definition Inv :: "string set ⇒ t_equas ⇒ bool"
where
  "Inv X ES ≡ finite ES ∧ (∃ rhs. (X, rhs) ∈ ES) ∧ distinct_equas ES ∧
    (∀ X xrhs. (X, xrhs) ∈ ES ⟶ ardenable (X, xrhs) ∧ X ≠ {} ∧

    rhs_eq_cls xrhs ⊆ insert {[]} (left_eq_cls ES))"
```

Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

Outline

1. Regular Expression(brief)
2. Myhill-Nerode Theorem(Intro)
3. FA to Regular Expressions
4. Proving Myhill-Nerode Theorem
   - Well-Founded iterating principle
   - Invariant predicate
   - Generating initial ES
   - Iteration step of ES
   - Final Proof

Generating Initial Equation-System

```
definition
  CT :: "string set ⇒ char ⇒ string set ⇒ bool" ("_-_→_" [99,99]99)
where "X-c→Y ≡ ((X;{[c]}) ⊆ Y)"

 types t_equa_rhs = "(string set × rexp) set"
types t_equa = "(string set × t_equa_rhs)"
types t_equas = "t_equa set"

 definition
  empty_rhs :: "string set ⇒ t_equa_rhs"
where "empty_rhs X ≡ if ([] ∈ X) then {({[]}, EMPTY)} else {}"
 definition
  folds :: "('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b"
where "folds f z S ≡ SOME x. fold_graph f z S x"

 definition
  equation_rhs :: "(string set) set ⇒ string set ⇒ t_equa_rhs"
where "equation_rhs CS X ≡ if (X = {[]}) then {({[]}, EMPTY)}
                           else {(S, folds ALT NULL {CHAR c| c. S-c→X} )| S.
S ∈ CS} ∪
                              empty_rhs X"

 definition
  equations :: "(string set) set ⇒ t_equas"

where  "equations CS ≡ {(X, equation_rhs CS X) | X. X ∈ CS}"
```

Equation-System (0)

$R_1 = \lambda$
$R_2 = R_1; 0 + R_2; (0|1)$
$R_3 = R_1; 1 + R_3; (0|1)$

Generating Initial Equation-System

```
definition
  CT :: "string set ⇒ char ⇒ string set ⇒ bool" ("_-_→_" [99,99]99)
where "X-c→Y ≡ ((X;{[c]}) ⊆ Y)"

types t_equa_rhs = "(string set × rexp) set"
types t_equa = "(string set × t_equa_rhs)"
types t_equas = "t_equa set"

definition
  empty_rhs :: "string set ⇒ t_equa_rhs"
where "empty_rhs X ≡ if ([] ∈ X) then {({[]}, EMPTY)} else {}"
definition
  folds :: "('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b"
where "folds f z S ≡ SOME x. fold_graph f z S x"

definition
  equation_rhs :: "(string set) set ⇒ string set ⇒ t_equa_rhs"
where "equation_rhs CS X ≡ if (X = {[]}) then {({[]}, EMPTY)}
                      else {(S, folds ALT NULL {CHAR c| c. S-c→X} )| S.
S ∈ CS} ∪
                          empty_rhs X"

definition
  equations :: "(string set) set ⇒ t_equas"

where  "equations CS ≡ {(X, equation_rhs CS X) | X. X ∈ CS}"
```

Equation-System (0)

$$R_1 = \lambda$$
$$R_2 = R_1;0 + R_2;(0|1)$$
$$R_3 = R_1;1 + R_3;(0|1)$$

Generating Initial Equation-System

```
definition
  CT :: "string set ⇒ char ⇒ string set ⇒ bool" ("_-_→_" [99,99]99)
where "X-c→Y ≡ ((X;{[c]}) ⊆ Y)"

 types t_equa_rhs = "(string set × rexp) set"
types t_equa = "(string set × t_equa_rhs)"
types t_equas = "t_equa set"

 definition
  empty_rhs :: "string set ⇒ t_equa_rhs"
where "empty_rhs X ≡ if ([] ∈ X) then {({[]}, EMPTY)} else {}"
 definition
  folds :: "('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b"
where "folds f z S ≡ SOME x. fold_graph f z S x"

 definition
  equation_rhs :: "(string set) set ⇒ string set ⇒ t_equa_rhs"
where "equation_rhs CS X ≡ if (X = {[]}) then {({[]}, EMPTY)}
                          else {(S, folds ALT NULL {CHAR c| c. S-c→X} )| S.
S ∈ CS} ∪
                            empty_rhs X"

 definition
  equations :: "(string set) set ⇒ t_equas"

where  "equations CS ≡ {(X, equation_rhs CS X) | X. X ∈ CS}"
```

Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

Generating Initial Equation-System

```
definition
  CT :: "string set ⇒ char ⇒ string set ⇒ bool" ("_-_→_" [99,99]99)
where "X-c→Y ≡ ((X;{[c]}) ⊆ Y)"

 types t_equa_rhs = "(string set × rexp) set"
types t_equa = "(string set × t_equa_rhs)"
types t_equas = "t_equa set"

 definition
  empty_rhs :: "string set ⇒ t_equa_rhs"
where "empty_rhs X ≡ if ([] ∈ X) then {({[]}, EMPTY)} else {}"
 definition
  folds :: "('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b"
where "folds f z S ≡ SOME x. fold_graph f z S x"

 definition
  equation_rhs :: "(string set) set ⇒ string set ⇒ t_equa_rhs"
where "equation_rhs CS X ≡ if (X = {[]}) then {({[]}, EMPTY)}
                         else {(S, folds ALT NULL {CHAR c| c. S-c→X} )| S.
S ∈ CS} ∪
                              empty_rhs X"

 definition
  equations :: "(string set) set ⇒ t_equas"

where  "equations CS ≡ {(X, equation_rhs CS X) | X. X ∈ CS}"
```

Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

Generating Initial Equation-System

```
definition
  CT :: "string set ⇒ char ⇒ string set ⇒ bool" ("_-_→_" [99,99]99)
where "X-c→Y ≡ ((X;{[c]}) ⊆ Y)"

 types t_equa_rhs = "(string set × rexp) set"
types t_equa = "(string set × t_equa_rhs)"
types t_equas = "t_equa set"

 definition
  empty_rhs :: "string set ⇒ t_equa_rhs"
where "empty_rhs X ≡ if ([] ∈ X) then {({[]}, EMPTY)} else {}"
 definition
  folds :: "('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b"
where "folds f z S ≡ SOME x. fold_graph f z S x"

 definition
  equation_rhs :: "(string set) set ⇒ string set ⇒ t_equa_rhs"
where "equation_rhs CS X ≡ if (X = {[]}) then {({[]}, EMPTY)}
                         else {(S, folds ALT NULL {CHAR c| c. S-c→X} )| S.
S ∈ CS} ∪
                           empty_rhs X"

 definition
  equations :: "(string set) set ⇒ t_equas"

where  "equations CS ≡ {(X, equation_rhs CS X) | X. X ∈ CS}"
```

Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

Generating Initial Equation-System

```
definition
  CT :: "string set ⇒ char ⇒ string set ⇒ bool" ("_-_→_" [99,99]99)
where "X-c→Y ≡ ((X;{[c]}) ⊆ Y)"

 types t_equa_rhs = "(string set × rexp) set"
types t_equa = "(string set × t_equa_rhs)"
types t_equas = "t_equa set"

 definition
  empty_rhs :: "string set ⇒ t_equa_rhs"
where "empty_rhs X ≡ if ([] ∈ X) then {({[]}, EMPTY)} else {}"
 definition
  folds :: "('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b"
where "folds f z S ≡ SOME x. fold_graph f z S x"

 definition
  equation_rhs :: "(string set) set ⇒ string set ⇒ t_equa_rhs"
where "equation_rhs CS X ≡ if (X = {[]}) then {({[]}, EMPTY)}
                         else {(S, folds ALT NULL {CHAR c| c. S-c→X} )| S.
S ∈ CS} ∪
                            empty_rhs X"

 definition
  equations :: "(string set) set ⇒ t_equas"

where  "equations CS ≡ {(X, equation_rhs CS X) | X. X ∈ CS}"
```

Equation-System (0)

$R_1 = \lambda$
$R_2 = R_1; 0 + R_2; (0|1)$
$R_3 = R_1; 1 + R_3; (0|1)$

Generating Initial Equation-System

```
definition
  CT :: "string set ⇒ char ⇒ string set ⇒ bool" ("_-_→_" [99,99]99)
where "X-c→Y ≡ ((X;{[c]}) ⊆ Y)"

 types t_equa_rhs = "(string set × rexp) set"
types t_equa = "(string set × t_equa_rhs)"
types t_equas = "t_equa set"

 definition
  empty_rhs :: "string set ⇒ t_equa_rhs"
where "empty_rhs X ≡ if ([] ∈ X) then {({[]}, EMPTY)} else {}"
 definition
  folds :: "('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b"
where "folds f z S ≡ SOME x. fold_graph f z S x"

 definition
  equation_rhs :: "(string set) set ⇒ string set ⇒ t_equa_rhs"
where "equation_rhs CS X ≡ if (X = {[]}) then {({[]}, EMPTY)}
                            else {(S, folds ALT NULL {CHAR c| c. S-c→X} )| S.
S ∈ CS} ∪
                                 empty_rhs X"

 definition
  equations :: "(string set) set ⇒ t_equas"

where  "equations CS ≡ {(X, equation_rhs CS X) | X. X ∈ CS}"
```

Equation-System (0)

$R_1 = \lambda$

$R_2 = R_1; 0 + R_2; (0|1)$

$R_3 = R_1; 1 + R_3; (0|1)$

Outline

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

eliminating one equation

- Not the equation of target equiv-class
- Approach: substitution
  1. Well-formed substitutor equation
     - substitutor = an equiv-class
     - rhs should not contain itself
     - if not, use Arden's Lemma to reform itself
  2. Substituting
     - if substitutor is empty-string itself
     - then do nothing
     - else replace itself with rhs of the substitutor
     - merging
  3. Delete substitutor equation

formalization

```
definition
  seq_rhs_r :: "t_equa_rhs ⇒ rexp ⇒ t_equa_rhs"
where "seq_rhs_r rhs r ≡ (λ(X, reg). (X, SEQ reg r)) ‘ rhs"
definition
  del_x_paired :: "('a × 'b) set ⇒ 'a ⇒ ('a × 'b) set"
where  "del_x_paired S x ≡ S - {X. X ∈ S ∧ fst X = x}"
definition
  merge_rhs :: "t_equa_rhs ⇒ t_equa_rhs ⇒ t_equa_rhs"
where "merge_rhs rhs rhs' ≡ {(X, r). (∃ r1 r2. (X,r1) ∈ rhs ∧ (X, r2) ∈ rhs' ∧ r = ALT r1
r2) ∨
                    (∃ r1. (X, r1) ∈ rhs ∧ (¬ (∃ r2. (X, r2) ∈ rhs')) ∧ r = r1)     ∨
                    (∃ r2. (X, r2) ∈ rhs' ∧ (¬ (∃ r1. (X, r1) ∈ rhs)) ∧ r = r2)     }"

definition
  arden_variate :: "string set ⇒ rexp ⇒ t_equa_rhs ⇒ t_equa_rhs"
where "arden_variate X r rhs ≡ seq_rhs_r (del_x_paired rhs X) (STAR r)"

definition
  rhs_subst :: "t_equa_rhs ⇒ string set ⇒ t_equa_rhs ⇒ rexp ⇒ t_equa_rhs"
where "rhs_subst rhs X xrhs r ≡ merge_rhs (del_x_paired rhs X) (seq_rhs_r xrhs r)"

definition
  equas_subst_f :: "string set ⇒ t_equa_rhs ⇒ t_equa ⇒ t_equa"
where "equas_subst_f X xrhs equa ≡ let (Y, rhs) = equa in
   if (∃ r. (X, r) ∈ rhs) then (Y, rhs_subst rhs X xrhs (SOME r. (X, r) ∈ rhs)) else equa"
definition
  equas_subst :: "t_equas ⇒ string set ⇒ t_equa_rhs ⇒ t_equas"
where "equas_subst ES X xrhs ≡ del_x_paired (equas_subst_f X xrhs ‘ ES) X"
```

Outline

1. Regular Expression(brief)
2. Myhill-Nerode Theorem(Intro)
3. FA to Regular Expressions
4. Proving Myhill-Nerode Theorem
   - Well-Founded iterating principle
   - Invariant predicate
   - Generating initial ES
   - Iteration step of ES
   - Final Proof

WF-iter Usage

```
lemma iteration_step:
  assumes Inv_ES: "Inv X ES" and not_T: "¬ TCon ES"
  shows "(∃ ES'. Inv X ES' ∧ (card ES', card ES) ∈ less_than)"
proof -
  from Inv_ES not_T have another: "∃Y yrhs. (Y, yrhs) ∈ ES ∧ X ≠ Y" unfolding Inv_def
    by (clarify, rule_tac exist_another_equa[where X = X], auto)
  then obtain Y yrhs where subst: "(Y, yrhs) ∈ ES" and not_X: " X ≠ Y" by blast
  show ?thesis (is "∃ ES'. ?P ES'")
  proof (cases "Y = {[]}")
    case True — in this situation, we pick a λ equation, thus directly remove it
    have "?P (ES - {(Y, yrhs)})"    next
    case False — first use arden's lemma, then do the substitution
    hence "?P (equas_subst ES Y yrhs')"
  qed
qed

lemma iteration_conc:
  assumes history: "Inv X ES"
  shows "∃ ES'. Inv X ES' ∧ TCon ES'" (is "∃ ES'. ?P ES'")
proof (cases "TCon ES")
  case True    hence "?P ES" using history by simp
  thus ?thesis by blast
next
  case False
  thus ?thesis using history iteration_step
    by (rule_tac f = card in wf_iter, simp_all)
qed
```

proof: every equiv-class has a Reg Exp.

```
lemma every_eqcl_has_reg:
  assumes finite_CS: "finite (UNIV Quo Lang)" and X_in_CS: "X ∈ (UNIV Quo Lang)"
  shows "∃ (reg::rexp). L reg = X" (is "∃ r. ?E r")
proof-
  have "∃ES'. Inv X ES' ∧ TCon ES'" using finite_CS X_in_CS
    by (auto intro:init_ES_satisfy_Inv iteration_conc)   have "∃ rhs. ES' = {(X, rhs)}"
by (auto dest!:card_Suc_Diff1 simp:card_eq_0_iff)
  then obtain rhs where ES'_single_equa: "ES' = {(X, rhs)}" ..
  hence X_ardenable: "ardenable (X, rhs)" using Inv_ES'
    by (simp add:Inv_def)  show ?thesis
  proof (cases "X = {[]}")
    case True hence "?E EMPTY" by simp
    thus ?thesis by blast
  next
    case False with  X_ardenable
    have "∃ rhs'. X = L rhs' ∧ rhs_eq_cls rhs' = rhs_eq_cls rhs - {X} ∧ distinct_rhs rhs'"
      by (drule_tac ardenable_prop, auto)
    then obtain rhs' where X_eq_rhs': "X = L rhs'"
      and rhs'_eq_cls: "rhs_eq_cls rhs' = rhs_eq_cls rhs - {X}"
      and rhs'_dist : "distinct_rhs rhs'" by blast
    hence "rhs_eq_cls rhs' = {{[]}}" using X_not_empty X_eq_rhs'
      by (auto simp:rhs_eq_cls_def)
    hence "∃ r. rhs' = {({[]}, r)}"
    then obtain r where "rhs' = {({[]}, r)}" ..
    hence "?E r" using X_eq_rhs' by (auto simp add:lang_seq_def)
    thus ?thesis by blast

  qed qed
```

proof: Myhill-Nerode(one direction)

```
theorem myhill_nerode:
  assumes finite_CS: "finite (UNIV Quo Lang)"
  shows   "∃ (reg::rexp). Lang = L reg" (is "∃ r. ?P r")
proof -
  have has_r_each: "∀C∈{X ∈ UNIV Quo Lang. ∀x∈X. x ∈ Lang}. ∃(r::rexp). C = L r"
    using finite_CS
    by (auto dest:every_eqcl_has_reg)
  have "∃ (rS::rexp set). finite rS ∧
                         (∀ C ∈ {X ∈ UNIV Quo Lang. ∀x∈X. x ∈ Lang}. ∃ r ∈ rS. C = L r) ∧
                         (∀ r ∈ rS. ∃ C ∈ {X ∈ UNIV Quo Lang. ∀x∈X. x ∈ Lang}. C = L r)"
then obtain rS where finite_rS : "finite rS"
    and r_each': "∀ C ∈ {X ∈ UNIV Quo Lang. ∀x∈X. x ∈ Lang}. ∃ r ∈ (rS::rexp set). C = L
r"
    and cl_each: "∀ r ∈ (rS::rexp set). ∃ C ∈ {X ∈ UNIV Quo Lang. ∀x∈X. x ∈ Lang}. C = L
r"
    by blast
  have "?P (folds ALT NULL rS)"
  proof
    show "Lang ⊆ L (folds ALT NULL rS)"      apply (clarsimp simp:fold_alt_null_eqs) by
blast
  next
    show "L (folds ALT NULL rS) ⊆ Lang"      by (clarsimp simp:fold_alt_null_eqs)
  qed
  thus ?thesis by blast

qed
```