Automata and Formal Languages II

Tree Automata

Peter Lammich

SS 2015

Overview by Lecture

- Apr 14: Slide 3
- Apr 21: Slide 2
- Apr 28: Slide 4
- May 5: Slide 50
- May 12: Slide 56
- May 19: Slide 64
- May 26: Holiday
- Jun 02: Slide 79
- Jun 09: Slide 90
- Jun 16: Slide 106
- Jun 23: Slide 108
- Jun 30: Slide 116
- Jul 7: Slide 137
- Jul 14: Slide 148

Organizational Issues

Lecture Tue 10:15 – 11:45, in MI 00.09.38 (Turing)

Organizational Issues

Lecture Tue 10:15 – 11:45, in MI 00.09.38 (Turing)

Tutorial ? Wed 10:15 – 11:45, in MI 00.09.38 (Turing)

- Weekly homework, will be corrected. Hand in before tutorial. Discussion during tutorial.

Organizational Issues

Lecture Tue 10:15 – 11:45, in MI 00.09.38 (Turing)

Tutorial ? Wed 10:15 – 11:45, in MI 00.09.38 (Turing)

- Weekly homework, will be corrected. Hand in before tutorial. Discussion during tutorial.

Exam Oral, Bonus for Homework!

- $\geq$ 50% of homework $\implies$ 0.3/0.4 better grade
  On first exam attempt. Only if passed w/o bonus!

Organizational Issues

Lecture Tue 10:15 – 11:45, in MI 00.09.38 (Turing)

Tutorial ? Wed 10:15 – 11:45, in MI 00.09.38 (Turing)

- Weekly homework, will be corrected. Hand in before tutorial. Discussion during tutorial.

Exam Oral, Bonus for Homework!

- $\geq$ 50% of homework $\implies$ 0.3/0.4 better grade
  On first exam attempt. Only if passed w/o bonus!

Material Tree Automata: Techniques and Applications (TATA)

- Free download at `http://tata.gforge.inria.fr/`

Organizational Issues

Lecture Tue 10:15 – 11:45, in MI 00.09.38 (Turing)

Tutorial ? Wed 10:15 – 11:45, in MI 00.09.38 (Turing)

- Weekly homework, will be corrected. Hand in before tutorial. Discussion during tutorial.

Exam Oral, Bonus for Homework!

- $\geq$ 50% of homework $\implies$ 0.3/0.4 better grade
  On first exam attempt. Only if passed w/o bonus!

Material Tree Automata: Techniques and Applications (TATA)

- Free download at `http://tata.gforge.inria.fr/`

Conflict with Equational Logic.

Proposed Content

- Finite tree automata: Basic theory (TATA Ch. 1)
  - Pumping Lemma, Closure Properties, Homomorphisms, Minimization, ...

Proposed Content

- Finite tree automata: Basic theory (TATA Ch. 1)
  - Pumping Lemma, Closure Properties, Homomorphisms, Minimization, ...
- Regular tree grammars and regular expressions (TATA Ch. 2)

Proposed Content

- Finite tree automata: Basic theory (TATA Ch. 1)
    - Pumping Lemma, Closure Properties, Homomorphisms, Minimization, ...
- Regular tree grammars and regular expressions (TATA Ch. 2)
- Hedge Automata (TATA Ch. 8)
    - Application: XML-Schema languages

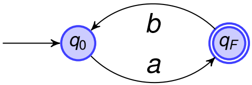
Proposed Content

- Finite tree automata: Basic theory (TATA Ch. 1)
  - Pumping Lemma, Closure Properties, Homomorphisms, Minimization, ...
- Regular tree grammars and regular expressions (TATA Ch. 2)
- Hedge Automata (TATA Ch. 8)
  - Application: XML-Schema languages
- Application: Analysis of Concurrent Programs
  - Dynamic Pushdown Networks (DPN)

Table of Contents

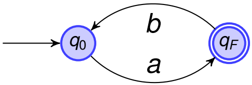
Tree Automata

- Finite automata recognize words, e.g.:



Tree Automata

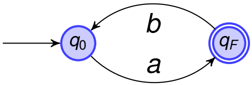
- Finite automata recognize words, e.g.:



$$q_0 \rightarrow a(q_F) \qquad\qquad q_F \rightarrow b(q_0)$$

Tree Automata

- Finite automata recognize words, e.g.:

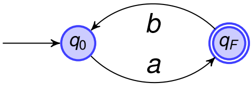


$q_0 \to a(q_F)$ $\qquad$ $q_F \to b(q_0)$

  - Words of alternating *a*s and *b*s, ending with *a*, e.g., *aba* or *abababa*

Tree Automata

- Finite automata recognize words, e.g.:



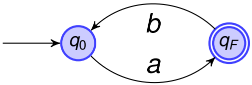
$$q_0 \rightarrow a(q_F) \qquad\qquad q_F \rightarrow b(q_0)$$

  - Words of alternating *a*s and *b*s, ending with *a*, e.g., *aba* or *abababa*
- Generalize to trees

$$q_0 \rightarrow a(q_1, q_1) \qquad q_1 \rightarrow b(q_0, q_0) \qquad q_1 \rightarrow L()$$

Tree Automata

- Finite automata recognize words, e.g.:



$$q_0 \to a(q_F) \qquad\qquad q_F \to b(q_0)$$

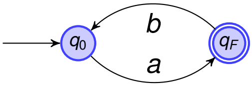
  - Words of alternating *a*s and *b*s, ending with *a*, e.g., *aba* or *abababa*
- Generalize to trees

$$q_0 \to a(q_1, q_1) \qquad\qquad q_1 \to b(q_0, q_0) \qquad\qquad q_1 \to L()$$

  - Trees with alternating „layers" of *a* nodes and *b* nodes.

Tree Automata

- Finite automata recognize words, e.g.:



$$q_0 \to a(q_F) \qquad\qquad q_F \to b(q_0)$$

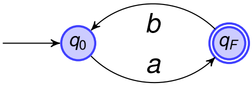
  - Words of alternating *a*s and *b*s, ending with *a*, e.g., *aba* or *abababa*
- Generalize to trees

$$q_0 \to a(q_1, q_1) \qquad q_1 \to b(q_0, q_0) \qquad q_1 \to L()$$

  - Trees with alternating „layers” of *a* nodes and *b* nodes.
    - Leafs are *L*-nodes, as node labels will have fixed arity.

Tree Automata

- Finite automata recognize words, e.g.:



$$q_0 \rightarrow a(q_F) \qquad\qquad q_F \rightarrow b(q_0)$$

  - Words of alternating *a*s and *b*s, ending with *a*, e.g., *aba* or *abababa*
- Generalize to trees

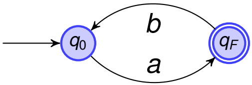
$$q_0 \rightarrow a(q_1, q_1) \qquad\qquad q_1 \rightarrow b(q_0, q_0) \qquad\qquad q_1 \rightarrow L()$$

  - Trees with alternating „layers” of *a* nodes and *b* nodes.
    - Leafs are *L*-nodes, as node labels will have fixed arity.

```
    a
   / \
  b   b
 / \ / \
a  a a  a
/\ /\ /\ /\
LLLLLLLL
```

Tree Automata

- Finite automata recognize words, e.g.:



$$q_0 \to a(q_F) \qquad\qquad q_F \to b(q_0)$$

  - Words of alternating *a*s and *b*s, ending with *a*, e.g., *aba* or *abababa*
- Generalize to trees

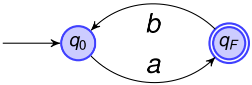
$$q_0 \to a(q_1, q_1) \qquad q_1 \to b(q_0, q_0) \qquad q_1 \to L()$$

  - Trees with alternating „layers" of *a* nodes and *b* nodes.
    - Leafs are *L*-nodes, as node labels will have fixed arity.

```
    a           a
   / \         / \
  b   b       b   L
 / \ / \     / \
a a a a     a a
/\ /\ /\ /\ /\ /\
LLLLLLLL    LLLL
```

Tree Automata

- Finite automata recognize words, e.g.:



$$q_0 \to a(q_F) \qquad\qquad q_F \to b(q_0)$$

  - Words of alternating *a*s and *b*s, ending with *a*, e.g., *aba* or *abababa*
- Generalize to trees

$$q_0 \to a(q_1, q_1) \qquad q_1 \to b(q_0, q_0) \qquad q_1 \to L()$$

  - Trees with alternating „layers" of *a* nodes and *b* nodes.
    - Leafs are *L*-nodes, as node labels will have fixed arity.

```
     a              a
    / \            / \
  b     b        b     L
 / \   / \      / \
a  a  a  a     a  a
/\ /\ /\ /\    /\ /\
LLLLLLLL       LLLL
```

- We also write trees as terms
  - $a(b(a(L, L), a(L, L)), b(a(L, L), a(L, L)))$
  - $a(b(a(L, L), a(L, L)), L)$

Properties

- Tree automata share many properties with word automata
  - Efficient membership query, union, intersection, emptiness check, ...

Properties

- Tree automata share many properties with word automata
  - Efficient membership query, union, intersection, emptiness check, ...
  - Deterministic and non-deterministic versions equally expressive

Properties

- Tree automata share many properties with word automata
  - Efficient membership query, union, intersection, emptiness check, ...
  - Deterministic and non-deterministic versions equally expressive
    - Only for deterministic bottom-up tree automata

Properties

- Tree automata share many properties with word automata
  - Efficient membership query, union, intersection, emptiness check, ...
  - Deterministic and non-deterministic versions equally expressive
    - Only for deterministic bottom-up tree automata
  - Minimization

Properties

- Tree automata share many properties with word automata
  - Efficient membership query, union, intersection, emptiness check, ...
  - Deterministic and non-deterministic versions equally expressive
    - Only for deterministic bottom-up tree automata
  - Minimization
  - ...

Applications

- Tree automata recognize sets of trees

Applications

- Tree automata recognize sets of trees
- Many structures in computer science are trees

Applications

- Tree automata recognize sets of trees
- Many structures in computer science are trees
    - XML documents

Applications

- Tree automata recognize sets of trees
- Many structures in computer science are trees
  - XML documents
  - Computations of parallel programs with fork/join

Applications

- Tree automata recognize sets of trees
- Many structures in computer science are trees
  - XML documents
  - Computations of parallel programs with fork/join
  - Values of algebraic datatypes in functional languages

Applications

- Tree automata recognize sets of trees
- Many structures in computer science are trees
  - XML documents
  - Computations of parallel programs with fork/join
  - Values of algebraic datatypes in functional languages
  - ...

Applications

- Tree automata recognize sets of trees
- Many structures in computer science are trees
  - XML documents
  - Computations of parallel programs with fork/join
  - Values of algebraic datatypes in functional languages
  - ...
- Tree automata can be used to

Applications

- Tree automata recognize sets of trees
- Many structures in computer science are trees
  - XML documents
  - Computations of parallel programs with fork/join
  - Values of algebraic datatypes in functional languages
  - ...
- Tree automata can be used to
  - Define XML schema languages

Applications

- Tree automata recognize sets of trees
- Many structures in computer science are trees
  - XML documents
  - Computations of parallel programs with fork/join
  - Values of algebraic datatypes in functional languages
  - ...
- Tree automata can be used to
  - Define XML schema languages
  - Model-check parallel programs

Applications

- Tree automata recognize sets of trees
- Many structures in computer science are trees
  - XML documents
  - Computations of parallel programs with fork/join
  - Values of algebraic datatypes in functional languages
  - ...
- Tree automata can be used to
  - Define XML schema languages
  - Model-check parallel programs
  - Analyze functional programs

Applications

- Tree automata recognize sets of trees
- Many structures in computer science are trees
  - XML documents
  - Computations of parallel programs with fork/join
  - Values of algebraic datatypes in functional languages
  - ...
- Tree automata can be used to
  - Define XML schema languages
  - Model-check parallel programs
  - Analyze functional programs
  - ...

Table of Contents

Table of Contents

Terms and Trees

- Let $\mathcal{F}$ be a finite set of symbols, and arity $: \mathcal{F} \to \mathbb{N}$ a function.

Terms and Trees

- Let $\mathcal{F}$ be a finite set of symbols, and arity $: \mathcal{F} \to \mathbb{N}$ a function.
  - $(\mathcal{F}, \text{arity})$ is a *ranked alphabet*. We also identify $\mathcal{F}$ with $(\mathcal{F}, \text{arity})$.

Terms and Trees

- Let $\mathcal{F}$ be a finite set of symbols, and arity : $\mathcal{F} \to \mathbb{N}$ a function.
  - $(\mathcal{F}, \text{arity})$ is a *ranked alphabet*. We also identify $\mathcal{F}$ with $(\mathcal{F}, \text{arity})$.
  - $\mathcal{F}_n := \{f \in \mathcal{F} \mid \text{arity}(f) = n\}$ is the set of symbols with arity n

Terms and Trees

- Let $\mathcal{F}$ be a finite set of symbols, and arity $: \mathcal{F} \to \mathbb{N}$ a function.
  - $(\mathcal{F}, \text{arity})$ is a *ranked alphabet*. We also identify $\mathcal{F}$ with $(\mathcal{F}, \text{arity})$.
  - $\mathcal{F}_n := \{f \in \mathcal{F} \mid \text{arity}(f) = n\}$ is the set of symbols with arity n
- Let $\mathcal{X}$ be a set of *variables*. We assume $\mathcal{X} \cap \mathcal{F}_0 = \emptyset$.

Terms and Trees

- Let $\mathcal{F}$ be a finite set of symbols, and arity : $\mathcal{F} \to \mathbb{N}$ a function.
  - $(\mathcal{F}, \text{arity})$ is a *ranked alphabet*. We also identify $\mathcal{F}$ with $(\mathcal{F}, \text{arity})$.
  - $\mathcal{F}_n := \{f \in \mathcal{F} \mid \text{arity}(f) = n\}$ is the set of symbols with arity n
- Let $\mathcal{X}$ be a set of *variables*. We assume $\mathcal{X} \cap \mathcal{F}_0 = \emptyset$.
- Then the set $T(\mathcal{F}, \mathcal{X})$ of terms over alphabet $\mathcal{F}$ and variables $\mathcal{X}$ is defined as the least solution of

$$
\begin{aligned}
T(\mathcal{F}, \mathcal{X}) &\supseteq \mathcal{F}_0 \\
T(\mathcal{F}, \mathcal{X}) &\supseteq \mathcal{X} \\
p \geq 1, f \in F_p, \text{ and } t_1, \ldots, t_p \in T(\mathcal{F}, \mathcal{X}) &\implies f(t_1, \ldots, t_n) \in T(\mathcal{F}, \mathcal{X})
\end{aligned}
$$

Terms and Trees

- Let $\mathcal{F}$ be a finite set of symbols, and arity : $\mathcal{F} \to \mathbb{N}$ a function.
  - $(\mathcal{F}, \text{arity})$ is a *ranked alphabet*. We also identify $\mathcal{F}$ with $(\mathcal{F}, \text{arity})$.
  - $\mathcal{F}_n := \{f \in \mathcal{F} \mid \text{arity}(f) = n\}$ is the set of symbols with arity n
- Let $\mathcal{X}$ be a set of *variables*. We assume $\mathcal{X} \cap \mathcal{F}_0 = \emptyset$.
- Then the set $T(\mathcal{F}, \mathcal{X})$ of terms over alphabet $\mathcal{F}$ and variables $\mathcal{X}$ is defined as the least solution of

$$\begin{aligned} T(\mathcal{F}, \mathcal{X}) &\supseteq \mathcal{F}_0 \\ T(\mathcal{F}, \mathcal{X}) &\supseteq \mathcal{X} \\ p \geq 1, f \in F_p, \text{ and } t_1, \dots, t_p \in T(\mathcal{F}, \mathcal{X}) &\implies f(t_1, \dots, t_n) \in T(\mathcal{F}, \mathcal{X}) \end{aligned}$$

  - Intuitively: Terms over functions from $\mathcal{F}$ and variables from $\mathcal{X}$.

Terms and Trees

- Let $\mathcal{F}$ be a finite set of symbols, and arity $: \mathcal{F} \to \mathbb{N}$ a function.
  - $(\mathcal{F}, \text{arity})$ is a *ranked alphabet*. We also identify $\mathcal{F}$ with $(\mathcal{F}, \text{arity})$.
  - $\mathcal{F}_n := \{f \in \mathcal{F} \mid \text{arity}(f) = n\}$ is the set of symbols with arity n
- Let $\mathcal{X}$ be a set of *variables*. We assume $\mathcal{X} \cap \mathcal{F}_0 = \emptyset$.
- Then the set $T(\mathcal{F}, \mathcal{X})$ of terms over alphabet $\mathcal{F}$ and variables $\mathcal{X}$ is defined as the least solution of

$$\begin{aligned} T(\mathcal{F}, \mathcal{X}) &\supseteq \mathcal{F}_0 \\ T(\mathcal{F}, \mathcal{X}) &\supseteq \mathcal{X} \\ p \geq 1, f \in F_p, \text{ and } t_1, \dots, t_p \in T(\mathcal{F}, \mathcal{X}) &\implies f(t_1, \dots, t_n) \in T(\mathcal{F}, \mathcal{X}) \end{aligned}$$

  - Intuitively: Terms over functions from $\mathcal{F}$ and variables from $\mathcal{X}$.
- Ground terms: $T(\mathcal{F}) := T(\mathcal{F}, \emptyset)$. Terms without variables.

Examples

- We also write a ranked alphabet as $\mathcal{F} = f_1/a_1, f_2/a_2, \ldots, f_n/a_n$, meaning $\mathcal{F} = (\{f_1, \ldots, f_n\}, (f_1 \mapsto a_1, \ldots, f_n \mapsto a_n))$

Examples

- We also write a ranked alphabet as $\mathcal{F} = f_1/a_1, f_2/a_2, \ldots, f_n/a_n$, meaning $\mathcal{F} = (\{f_1, \ldots, f_n\}, (f_1 \mapsto a_1, \ldots, f_n \mapsto a_n))$
- $\mathcal{F} = \mathit{true}/0, \mathit{false}/0, \mathit{and}/2, \mathit{not}/1$ - Syntax trees of boolean expressions

Examples

- We also write a ranked alphabet as $\mathcal{F} = f_1/a_1, f_2/a_2, \ldots, f_n/a_n$, meaning $\mathcal{F} = (\{f_1, \ldots, f_n\}, (f_1 \mapsto a_1, \ldots, f_n \mapsto a_n))$
- $\mathcal{F} = \mathit{true}/0, \mathit{false}/0, \mathit{and}/2, \mathit{not}/1$ - Syntax trees of boolean expressions
  - $\mathit{and}(\mathit{true}, \mathit{not}(x)) \in T(\mathcal{F}, \{x\})$

Examples

- We also write a ranked alphabet as $\mathcal{F} = f_1/a_1, f_2/a_2, \ldots, f_n/a_n$, meaning $\mathcal{F} = (\{f_1, \ldots, f_n\}, (f_1 \mapsto a_1, \ldots, f_n \mapsto a_n))$
- $\mathcal{F} = \mathit{true}/0, \mathit{false}/0, \mathit{and}/2, \mathit{not}/1$ - Syntax trees of boolean expressions
  - $\mathit{and}(\mathit{true}, \mathit{not}(x)) \in T(\mathcal{F}, \{x\})$
- $\mathcal{F} = 0/0, \mathit{Suc}/1, +/2, */2$ - Arithmetic expressions over naturals (using unary representation)

Examples

- We also write a ranked alphabet as $\mathcal{F} = f_1/a_1, f_2/a_2, \ldots, f_n/a_n$, meaning $\mathcal{F} = (\{f_1, \ldots, f_n\}, (f_1 \mapsto a_1, \ldots, f_n \mapsto a_n))$
- $\mathcal{F} = \mathit{true}/0, \mathit{false}/0, \mathit{and}/2, \mathit{not}/1$ - Syntax trees of boolean expressions
  - $\mathit{and}(\mathit{true}, \mathit{not}(x)) \in T(\mathcal{F}, \{x\})$
- $\mathcal{F} = 0/0, \mathit{Suc}/1, +/2, */2$ - Arithmetic expressions over naturals (using unary representation)
  - $\mathit{Suc}(0) + (\mathit{Suc}(\mathit{Suc}(0)) * x) \in T(\mathcal{F}, \{x\})$

Examples

- We also write a ranked alphabet as $\mathcal{F} = f_1/a_1, f_2/a_2, \ldots, f_n/a_n$, meaning $\mathcal{F} = (\{f_1, \ldots, f_n\}, (f_1 \mapsto a_1, \ldots, f_n \mapsto a_n))$
- $\mathcal{F} = \mathit{true}/0, \mathit{false}/0, \mathit{and}/2, \mathit{not}/1$ - Syntax trees of boolean expressions
  - $\mathit{and}(\mathit{true}, \mathit{not}(x)) \in T(\mathcal{F}, \{x\})$
- $\mathcal{F} = 0/0, \mathit{Suc}/1, +/2, */2$ - Arithmetic expressions over naturals (using unary representation)
  - $\mathit{Suc}(0) + (\mathit{Suc}(\mathit{Suc}(0)) * x) \in T(\mathcal{F}, \{x\})$
    - We will use infix-notation for terms when appropriate

Trees

- Terms can be identified by trees: Nodes with p successors labeled with symbol from $\mathcal{F}_p$.

Trees

- Terms can be identified by trees: Nodes with p successors labeled with symbol from $\mathcal{F}_p$.
- $\mathit{and}(\mathit{true}, \mathit{not}(x)) \in T(\mathcal{F}, \{x\})$

```
    and
   /   \
true   not
        |
        x
```

Trees

- Terms can be identified by trees: Nodes with p successors labeled with symbol from $\mathcal{F}_p$.
- $and(true, not(x)) \in T(\mathcal{F}, \{x\})$

```
     and
    /   \
 true   not
         |
         x
```

- $Suc(0) + (Suc(Suc(0)) * x)$

```
      +
     / \
  Suc   *
   |   / \
   0 Suc  x
      |
     Suc
      |
      0
```

Tree Automata

- A (nondeterministic) finite tree automaton (NFTA) over alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where

Tree Automata

- A (nondeterministic) finite tree automaton (NFTA) over alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - Q is a finite set of *states*. $Q \cap F_0 = \emptyset$

Tree Automata

- A (nondeterministic) finite tree automaton (NFTA) over alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - Q is a finite set of *states*. $Q \cap F_0 = \emptyset$
  - $Q_f \subseteq Q$ is a set of *final states*

Tree Automata

- A (nondeterministic) finite tree automaton (NFTA) over alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - Q is a finite set of *states*. $Q \cap F_0 = \emptyset$
  - $Q_f \subseteq Q$ is a set of *final states*
  - Δ is a set of rules of the form

$$f(q_1, \ldots, q_n) \to q$$

  where $f \in \mathcal{F}_n$ and $q, q_1, \ldots, q_n \in Q$

Tree Automata

- A (nondeterministic) finite tree automaton (NFTA) over alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - Q is a finite set of *states*. $Q \cap F_0 = \emptyset$
  - $Q_f \subseteq Q$ is a set of *final states*
  - Δ is a set of rules of the form

    $$f(q_1, \ldots, q_n) \to q$$

    where $f \in \mathcal{F}_n$ and $q, q_1, \ldots, q_n \in Q$
- Intuition: Use the rules from Δ to re-write a given tree to a final state

Tree Automata

- A (nondeterministic) finite tree automaton (NFTA) over alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - Q is a finite set of *states*. $Q \cap F_0 = \emptyset$
  - $Q_f \subseteq Q$ is a set of *final states*
  - Δ is a set of rules of the form

$$f(q_1, \ldots, q_n) \to q$$

    where $f \in \mathcal{F}_n$ and $q, q_1, \ldots, q_n \in Q$
- Intuition: Use the rules from Δ to re-write a given tree to a final state
- For a tree $t \in T(\mathcal{F})$ and a state q, we define $t \to_{\mathcal{A}} q$ as the least relation that satisfies

$$f(q_1, \ldots, q_n) \to q \in \Delta, \forall 1 \leq i \leq n.\ t_i \to_{\mathcal{A}} q_i \implies f(t_1, \ldots, t_n) \to_{\mathcal{A}} q$$

Tree Automata

- A (nondeterministic) finite tree automaton (NFTA) over alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - Q is a finite set of *states*. $Q \cap F_0 = \emptyset$
  - $Q_f \subseteq Q$ is a set of *final states*
  - Δ is a set of rules of the form

    $$f(q_1, \dots, q_n) \to q$$

    where $f \in \mathcal{F}_n$ and $q, q_1, \dots, q_n \in Q$
- Intuition: Use the rules from Δ to re-write a given tree to a final state
- For a tree $t \in T(\mathcal{F})$ and a state q, we define $t \to_{\mathcal{A}} q$ as the least relation that satisfies

  $$f(q_1, \dots, q_n) \to q \in \Delta, \forall 1 \leq i \leq n.\ t_i \to_{\mathcal{A}} q_i \implies f(t_1, \dots, t_n) \to_{\mathcal{A}} q$$

  - $t \to_{\mathcal{A}} q$: Tree t is *accepted* in state q

Tree Automata

- A (nondeterministic) finite tree automaton (NFTA) over alphabet $\mathcal{F}$ is a tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ where
  - Q is a finite set of *states*. $Q \cap F_0 = \emptyset$
  - $Q_f \subseteq Q$ is a set of *final states*
  - Δ is a set of rules of the form

    $$f(q_1, \dots, q_n) \to q$$

    where $f \in \mathcal{F}_n$ and $q, q_1, \dots, q_n \in Q$
- Intuition: Use the rules from Δ to re-write a given tree to a final state
- For a tree $t \in T(\mathcal{F})$ and a state q, we define $t \to_{\mathcal{A}} q$ as the least relation that satisfies

  $$f(q_1, \dots, q_n) \to q \in \Delta, \forall 1 \le i \le n.\ t_i \to_{\mathcal{A}} q_i \implies f(t_1, \dots, t_n) \to_{\mathcal{A}} q$$

  - $t \to_{\mathcal{A}} q$: Tree t is *accepted* in state q
- The language $L(\mathcal{A})$ of $\mathcal{A}$ are all trees accepted in final states

  $$L(\mathcal{A}) := \{t \mid \exists q \in Q_f.\ t \to_{\mathcal{A}} q\}$$

Example

- Tree automaton accepting arithmetic expressions that evaluate to even numbers

$$
\begin{array}{llll}
\mathcal{F} = 0/0, Suc/1, +/2 & & & \\
Q := \{e, o\} & Q_f = \{e\} & & \\
0 \to e & Suc(e) \to o & Suc(o) \to e & \\
e + e \to e & e + o \to o & o + e \to o & o + o \to e
\end{array}
$$

Example

- Tree automaton accepting arithmetic expressions that evaluate to even numbers

$$
\begin{array}{llll}
\mathcal{F} = 0/0, Suc/1, +/2 & & & \\
Q := \{e, o\} & Q_f = \{e\} & & \\
0 \to e & Suc(e) \to o & Suc(o) \to e & \\
e + e \to e & e + o \to o & o + e \to o & o + o \to e
\end{array}
$$

- Examples for runs on board
  - $Suc(Suc(0)) + Suc(0) + Suc(0)$
  - $0 + Suc(0)$

Remark

- In TATA, a move-relation is defined. $t \xrightarrow[\mathcal{A}]{} t'$ rewrites a node in the tree according to a rule.

Remark

- In TATA, a move-relation is defined. $t \underset{\mathcal{A}}{\longrightarrow} t'$ rewrites a node in the tree according to a rule.
- Another version even keeps track of the tree nodes, and just adds the states as additional nodes of arity 1.

Remark

- In TATA, a move-relation is defined. $t \underset{\mathcal{A}}{\longrightarrow} t'$ rewrites a node in the tree according to a rule.
- Another version even keeps track of the tree nodes, and just adds the states as additional nodes of arity 1.
- Examples on board

Table of Contents

Epsilon rules

- As for word automata, we may add ϵ-rules of the form

$$q \to q' \text{ for } q, q' \in Q$$

Epsilon rules

- As for word automata, we may add ϵ-rules of the form

$$q \to q' \text{ for } q, q' \in Q$$

- The acceptance relation is extended accordingly

$$\begin{aligned} f(q_1, \ldots, q_n) \to q \in \Delta, \forall 1 \leq i \leq n.\ t_i \to_{\mathcal{A}} q_i &\implies f(t_1, \ldots, t_n) \to_{\mathcal{A}} q \\ q \to q' \in \Delta, t \to_{\mathcal{A}} q &\implies t \to_{\mathcal{A}} q' \end{aligned}$$

Epsilon rules

- As for word automata, we may add ϵ-rules of the form

$$q \to q' \text{ for } q, q' \in Q$$

- The acceptance relation is extended accordingly

$$f(q_1, \ldots, q_n) \to q \in \Delta, \forall 1 \leq i \leq n.\ t_i \to_{\mathcal{A}} q_i \implies f(t_1, \ldots, t_n) \to_{\mathcal{A}} q$$
$$q \to q' \in \Delta, t \to_{\mathcal{A}} q \implies t \to_{\mathcal{A}} q'$$

- Example: (Non-empty) lists of natural numbers

$$\begin{aligned} 0 &\to q_n & Suc(q_n) &\to q_n \\ nil &\to q_l & cons(q_n, q_l) &\to q_l' \\ q_l' &\to q_l \end{aligned}$$

Epsilon rules

- As for word automata, we may add ϵ-rules of the form

$$q \to q' \text{ for } q, q' \in Q$$

- The acceptance relation is extended accordingly

$$\begin{array}{c} f(q_1, \ldots, q_n) \to q \in \Delta, \forall 1 \leq i \leq n.\ t_i \to_{\mathcal{A}} q_i \implies f(t_1, \ldots, t_n) \to_{\mathcal{A}} q \\ q \to q' \in \Delta, t \to_{\mathcal{A}} q \implies t \to_{\mathcal{A}} q' \end{array}$$

- Example: (Non-empty) lists of natural numbers

$$\begin{array}{rlrl} 0 & \to q_n & Suc(q_n) & \to q_n \\ nil & \to q_l & cons(q_n, q_l) & \to q_l' \\ q_l' & \to q_l & & \end{array}$$

  - Last rule converts non-empty list (q_l') to list (q_l)

Epsilon rules

- As for word automata, we may add ϵ-rules of the form

  $$q \to q' \text{ for } q, q' \in Q$$

- The acceptance relation is extended accordingly

  $$f(q_1, \ldots, q_n) \to q \in \Delta, \forall 1 \leq i \leq n.\ t_i \to_{\mathcal{A}} q_i \implies f(t_1, \ldots, t_n) \to_{\mathcal{A}} q$$
  $$q \to q' \in \Delta, t \to_{\mathcal{A}} q \implies t \to_{\mathcal{A}} q'$$

- Example: (Non-empty) lists of natural numbers

  $$\begin{array}{rlrl} 0 & \to q_n & Suc(q_n) & \to q_n \\ nil & \to q_l & cons(q_n, q_l) & \to q_l' \\ q_l' & \to q_l & & \end{array}$$

  - Last rule converts non-empty list (q_l') to list (q_l)
- On board: Accepting [], and $[0, Suc(0)]$

Equivalence of NFTAs with and without ϵ - rules

Theorem

For a NFTA $\mathcal{A}$ with ϵ-rules, there is a NFTA without ϵ-rules that recognizes the same language

- Proof sketch:

Equivalence of NFTAs with and without ϵ - rules

Theorem

For a NFTA $\mathcal{A}$ with ϵ-rules, there is a NFTA without ϵ-rules that recognizes the same language

- Proof sketch:
  - Let $cl(q)$ denote the ϵ-closure of q

$$q \in cl(q) \qquad\qquad q' \in cl(q), q' \to q'' \implies q'' \in cl(q)$$

Equivalence of NFTAs with and without ϵ - rules

Theorem

For a NFTA $\mathcal{A}$ with ϵ-rules, there is a NFTA without ϵ-rules that recognizes the same language

- Proof sketch:
  - Let $cl(q)$ denote the ϵ-closure of q

$$q \in cl(q) \qquad\qquad q' \in cl(q), q' \to q'' \implies q'' \in cl(q)$$

  - Define $\Delta' := \{f(q_1, \ldots, q_n) \to q' \mid f(q_1, \ldots, q_n) \to q \in \Delta \wedge q' \in cl(q)\}$

Equivalence of NFTAs with and without ϵ - rules

Theorem

For a NFTA $\mathcal{A}$ with ϵ-rules, there is a NFTA without ϵ-rules that recognizes the same language

- Proof sketch:
  - Let $cl(q)$ denote the ϵ-closure of q

$$q \in cl(q) \qquad\qquad q' \in cl(q), q' \to q'' \implies q'' \in cl(q)$$

  - Define $\Delta' := \{f(q_1, \ldots, q_n) \to q' \mid f(q_1, \ldots, q_n) \to q \in \Delta \wedge q' \in cl(q)\}$
  - Define $A' := (Q, \mathcal{F}, Q_f, \Delta')$

Equivalence of NFTAs with and without ϵ - rules

Theorem

For a NFTA $\mathcal{A}$ with ϵ-rules, there is a NFTA without ϵ-rules that recognizes the same language

- Proof sketch:
  - Let $cl(q)$ denote the ϵ-closure of q

$$q \in cl(q) \qquad\qquad q' \in cl(q), q' \to q'' \implies q'' \in cl(q)$$

  - Define $\Delta' := \{f(q_1, \ldots, q_n) \to q' \mid f(q_1, \ldots, q_n) \to q \in \Delta \wedge q' \in cl(q)\}$
  - Define $A' := (Q, \mathcal{F}, Q_f, \Delta')$
  - Show: $t \to_{\mathcal{A}} q$ iff $t \to_{\mathcal{A}'} q$

Equivalence of NFTAs with and without ϵ - rules

Theorem

For a NFTA $\mathcal{A}$ with ϵ-rules, there is a NFTA without ϵ-rules that recognizes the same language

- Proof sketch:
  - Let $cl(q)$ denote the ϵ-closure of q

$$q \in cl(q) \qquad\qquad q' \in cl(q), q' \to q'' \implies q'' \in cl(q)$$

  - Define $\Delta' := \{f(q_1, \ldots, q_n) \to q' \mid f(q_1, \ldots, q_n) \to q \in \Delta \wedge q' \in cl(q)\}$
  - Define $A' := (Q, \mathcal{F}, Q_f, \Delta')$
  - Show: $t \to_{\mathcal{A}} q$ iff $t \to_{\mathcal{A}'} q$
    - on board

Equivalence of NFTAs with and without ϵ - rules

Theorem

For a NFTA $\mathcal{A}$ with ϵ-rules, there is a NFTA without ϵ-rules that recognizes the same language

- Proof sketch:
  - Let $cl(q)$ denote the ϵ-closure of q

$$q \in cl(q) \qquad\qquad q' \in cl(q), q' \to q'' \implies q'' \in cl(q)$$

  - Define $\Delta' := \{f(q_1, \ldots, q_n) \to q' \mid f(q_1, \ldots, q_n) \to q \in \Delta \wedge q' \in cl(q)\}$
  - Define $A' := (Q, \mathcal{F}, Q_f, \Delta')$
  - Show: $t \to_{\mathcal{A}} q$ iff $t \to_{\mathcal{A}'} q$
    - on board
- From now on, we assume tree automata without ϵ-rules, unless noted otherwise.

Last Lecture

- Nondeterministic Finite Tree Automata (NFTA)
  - Ranked alphabet, Terms/Trees
  - Rules: $f(q_1, \ldots, q_n) \rightarrow q$
  - Intuition: Rewrite tree to single state
- Epsilon rules
  - $q \rightarrow q'$
  - Do not increase expressiveness (recognizable languages)

Table of Contents

Deterministic Finite Tree Automata

Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a finite tree automaton.

- $\mathcal{A}$ is *deterministic* (DFTA), if there are no two rules with the same LHS (and no ϵ-rules), i.e.

$$l \to q_1 \in \Delta \wedge l \to q_2 \in \Delta \implies q_1 = q_2$$

Deterministic Finite Tree Automata

Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a finite tree automaton.

- $\mathcal{A}$ is *deterministic* (DFTA), if there are no two rules with the same LHS (and no ϵ-rules), i.e.

$$l \to q_1 \in \Delta \wedge l \to q_2 \in \Delta \implies q_1 = q_2$$

  - For a DFTA, every tree is accepted in at most one state

Deterministic Finite Tree Automata

Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a finite tree automaton.

- $\mathcal{A}$ is *deterministic* (DFTA), if there are no two rules with the same LHS (and no ϵ-rules), i.e.

$$l \to q_1 \in \Delta \land l \to q_2 \in \Delta \implies q_1 = q_2$$

  - For a DFTA, every tree is accepted in at most one state
- $\mathcal{A}$ is *complete*, if for every $f \in F_n, q_1, \ldots, q_n \in Q$, there is a rule $f(q_1, \ldots, q_n) \to q$

Deterministic Finite Tree Automata

Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a finite tree automaton.

- $\mathcal{A}$ is *deterministic* (DFTA), if there are no two rules with the same LHS (and no ϵ-rules), i.e.

$$l \to q_1 \in \Delta \land l \to q_2 \in \Delta \implies q_1 = q_2$$

  - For a DFTA, every tree is accepted in at most one state
- $\mathcal{A}$ is *complete*, if for every $f \in F_n, q_1, \ldots, q_n \in Q$, there is a rule $f(q_1, \ldots, q_n) \to q$
  - For a complete tree automata, every tree is accepted in at least one state

Deterministic Finite Tree Automata

Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a finite tree automaton.

- $\mathcal{A}$ is *deterministic* (DFTA), if there are no two rules with the same LHS (and no ϵ-rules), i.e.

$$l \to q_1 \in \Delta \wedge l \to q_2 \in \Delta \implies q_1 = q_2$$

  - For a DFTA, every tree is accepted in at most one state
- $\mathcal{A}$ is *complete*, if for every $f \in F_n, q_1, \ldots, q_n \in Q$, there is a rule $f(q_1, \ldots, q_n) \to q$
  - For a complete tree automata, every tree is accepted in at least one state
  - For a complete DFTA, every tree is accepted in exactly one state

Deterministic Finite Tree Automata

Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a finite tree automaton.

- $\mathcal{A}$ is *deterministic* (DFTA), if there are no two rules with the same LHS (and no ϵ-rules), i.e.

$$l \rightarrow q_1 \in \Delta \wedge l \rightarrow q_2 \in \Delta \implies q_1 = q_2$$

  - For a DFTA, every tree is accepted in at most one state
- $\mathcal{A}$ is *complete*, if for every $f \in F_n, q_1, \ldots, q_n \in Q$, there is a rule $f(q_1, \ldots, q_n) \rightarrow q$
  - For a complete tree automata, every tree is accepted in at least one state
  - For a complete DFTA, every tree is accepted in exactly one state
- A state $q \in Q$ is *accessible*, if there is a t with $t \rightarrow_{\mathcal{A}} q$.

Deterministic Finite Tree Automata

Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a finite tree automaton.

- $\mathcal{A}$ is *deterministic* (DFTA), if there are no two rules with the same LHS (and no ϵ-rules), i.e.

$$l \to q_1 \in \Delta \wedge l \to q_2 \in \Delta \implies q_1 = q_2$$

  - For a DFTA, every tree is accepted in at most one state
- $\mathcal{A}$ is *complete*, if for every $f \in F_n, q_1, \ldots, q_n \in Q$, there is a rule $f(q_1, \ldots, q_n) \to q$
  - For a complete tree automata, every tree is accepted in at least one state
  - For a complete DFTA, every tree is accepted in exactly one state
- A state $q \in Q$ is *accessible*, if there is a t with $t \to_{\mathcal{A}} q$.
- $\mathcal{A}$ is *reduced*, if all states in Q are accessible.

Membership Test for DFTA

- Complete DFTAs have a simple (and efficient) membership test

```
acc (f (t1, ..., tn)) =
  let
    q1 = acc t1; ...; qn = acc tn
  in
    the q with f(q1,...,qn) ∈ Δ
```

Membership Test for DFTA

- Complete DFTAs have a simple (and efficient) membership test

```
acc (f (t1, ..., tn)) =
  let
    q1 = acc t1; ...; qn = acc tn
  in
    the q with f(q1,...,qn) ∈ Δ
```

- Note: For NFTAs, we need to backtrack, or use on-the-fly determinization

Reduction Algorithm

- Obviously, removing inaccessible states does not change the language of an NFTA.

Reduction Algorithm

- Obviously, removing inaccessible states does not change the language of an NFTA.
- The following algorithm computes the set of accessible states in polynomial time

```
A := ∅
repeat
   A := a ∪ {q} for q with
        f(q_1, ..., q_n) → q ∈ Δ, q_1, ..., q_n ∈ A
until no more states can be added to A
```

Reduction Algorithm

- Obviously, removing inaccessible states does not change the language of an NFTA.
- The following algorithm computes the set of accessible states in polynomial time

$$\begin{array}{l} \mathrm{A} \ := \ \emptyset \\ \textbf{repeat} \\ \quad \mathrm{A} \ := \ a \cup \{q\} \ \textbf{ for } \ q \ \textbf{ with} \\ \qquad f(q_1, \ldots, q_n) \to q \in \Delta, q_1, \ldots, q_n \in \mathrm{A} \\ \textbf{until} \ \text{ no more states can be added to A} \end{array}$$

- Proof sketch

Reduction Algorithm

- Obviously, removing inaccessible states does not change the language of an NFTA.
- The following algorithm computes the set of accessible states in polynomial time

$$
\begin{array}{l}
\mathrm{A} \;:=\; \emptyset \\
\textbf{repeat} \\
\quad \mathrm{A} \;:=\; a \cup \{q\} \;\; \textbf{for} \;\; q \;\; \textbf{with} \\
\qquad f(q_1, \ldots, q_n) \to q \in \Delta, q_1, \ldots, q_n \in \mathrm{A} \\
\textbf{until} \;\; \text{no more states can be added to A}
\end{array}
$$

- Proof sketch
  - Invariant: All states in A are accessible.

Reduction Algorithm

- Obviously, removing inaccessible states does not change the language of an NFTA.
- The following algorithm computes the set of accessible states in polynomial time

$$
\begin{array}{l}
\mathrm{A} \ := \ \emptyset \\
\textbf{repeat} \\
\quad \mathrm{A} \ := \ a \cup \{q\} \ \ \textbf{for} \ \ q \ \ \textbf{with} \\
\qquad f(q_1, \ldots, q_n) \rightarrow q \in \Delta, q_1, \ldots, q_n \in \mathrm{A} \\
\textbf{until} \ \ \text{no more states can be added to A}
\end{array}
$$

- Proof sketch
  - Invariant: All states in A are accessible.
  - If there is an accessible state not in A, saturation is not complete

Reduction Algorithm

- Obviously, removing inaccessible states does not change the language of an NFTA.
- The following algorithm computes the set of accessible states in polynomial time

```
A := ∅
repeat
   A := a ∪ {q}  for q with
      f(q_1, ..., q_n) → q ∈ Δ, q_1, ..., q_n ∈ A
until no more states can be added to A
```

- Proof sketch
  - Invariant: All states in A are accessible.
  - If there is an accessible state not in A, saturation is not complete
    - Induction on $t \rightarrow_{\mathcal{A}} q$

Determinization (Powerset construction)

- Theorem: For every NFTA, there exists a complete DFTA with the same language

Determinization (Powerset construction)

- Theorem: For every NFTA, there exists a complete DFTA with the same language
- Let $Q_d := 2^Q$ and $Q_{df} := \{s \in Q_d \mid s \cap Q_f \neq \emptyset\}$

Determinization (Powerset construction)

- Theorem: For every NFTA, there exists a complete DFTA with the same language
- Let $Q_d := 2^Q$ and $Q_{df} := \{s \in Q_d \mid s \cap Q_f \neq \emptyset\}$
- Let $f(s_1, \ldots, s_n) \rightarrow s \in \Delta_d$ iff
  $s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n \mid f(q_1, \ldots, q_n) \rightarrow q \in \Delta\}$

Determinization (Powerset construction)

- Theorem: For every NFTA, there exists a complete DFTA with the same language
- Let $Q_d := 2^Q$ and $Q_{df} := \{s \in Q_d \mid s \cap Q_f \neq \emptyset\}$
- Let $f(s_1, \ldots, s_n) \to s \in \Delta_d$ iff
  $s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n \mid f(q_1, \ldots, q_n) \to q \in \Delta\}$
- Define $\mathcal{A}_d := (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$

Determinization (Powerset construction)

- Theorem: For every NFTA, there exists a complete DFTA with the same language
- Let $Q_d := 2^Q$ and $Q_{df} := \{s \in Q_d \mid s \cap Q_f \neq \emptyset\}$
- Let $f(s_1, \ldots, s_n) \to s \in \Delta_d$ iff $s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n \mid f(q_1, \ldots, q_n) \to q \in \Delta\}$
- Define $\mathcal{A}_d := (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$
- Idea: $\mathcal{A}_d$ accepts tree t in the set of all states in that $\mathcal{A}$ accepts t (maybe the empty set)

Determinization (Powerset construction)

- Theorem: For every NFTA, there exists a complete DFTA with the same language
- Let $Q_d := 2^Q$ and $Q_{df} := \{s \in Q_d \mid s \cap Q_f \neq \emptyset\}$
- Let $f(s_1, \ldots, s_n) \to s \in \Delta_d$ iff
  $s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n \mid f(q_1, \ldots, q_n) \to q \in \Delta\}$
- Define $\mathcal{A}_d := (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$
- Idea: $\mathcal{A}_d$ accepts tree t in the set of all states in that $\mathcal{A}$ accepts t (maybe the empty set)
  - Formally: $t \to_{\mathcal{A}_d} s$ iff $s = \{q \in Q \mid t \to_{\mathcal{A}} q\}$

Determinization (Powerset construction)

- Theorem: For every NFTA, there exists a complete DFTA with the same language
- Let $Q_d := 2^Q$ and $Q_{df} := \{s \in Q_d \mid s \cap Q_f \neq \emptyset\}$
- Let $f(s_1, \ldots, s_n) \to s \in \Delta_d$ iff
  $s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n \mid f(q_1, \ldots, q_n) \to q \in \Delta\}$
- Define $\mathcal{A}_d := (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$
- Idea: $\mathcal{A}_d$ accepts tree t in the set of all states in that $\mathcal{A}$ accepts t (maybe the empty set)
  - Formally: $t \to_{\mathcal{A}_d} s$ iff $s = \{q \in Q \mid t \to_{\mathcal{A}} q\}$
- Lemma: The automaton $\mathcal{A}_d$ is a complete DFTA, and we have $L(\mathcal{A}) = L(\mathcal{A}_d)$. (On board)

Determinization (Powerset construction)

- Theorem: For every NFTA, there exists a complete DFTA with the same language
- Let $Q_d := 2^Q$ and $Q_{df} := \{s \in Q_d \mid s \cap Q_f \neq \emptyset\}$
- Let $f(s_1, \ldots, s_n) \to s \in \Delta_d$ iff
  $s = \{q \in Q \mid \exists q_1 \in s_1, \ldots, q_n \in s_n \mid f(q_1, \ldots, q_n) \to q \in \Delta\}$
- Define $\mathcal{A}_d := (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$
- Idea: $\mathcal{A}_d$ accepts tree t in the set of all states in that $\mathcal{A}$ accepts t (maybe the empty set)
  - Formally: $t \to_{\mathcal{A}_d} s$ iff $s = \{q \in Q \mid t \to_{\mathcal{A}} q\}$
- Lemma: The automaton $\mathcal{A}_d$ is a complete DFTA, and we have $L(\mathcal{A}) = L(\mathcal{A}_d)$. (On board)
- Theorem follows from this.

Determinization with reduction

- Above method always construct exponentially many states

Determinization with reduction

- Above method always construct exponentially many states
  - Typically, many of the inaccessible

Determinization with reduction

- Above method always construct exponentially many states
  - Typically, many of the inaccessible
- Idea: Combine determinization and reduction

Determinization with reduction

- Above method always construct exponentially many states
  - Typically, many of the inaccessible
- Idea: Combine determinization and reduction
  - Only construct accessible states of $\mathcal{A}_d$

Determinization with reduction

- Above method always construct exponentially many states
  - Typically, many of the inaccessible
- Idea: Combine determinization and reduction
  - Only construct accessible states of $\mathcal{A}_d$

$Q_d \; := \; \emptyset$
$\Delta_d \; := \; \emptyset$
repeat
$\quad Q_d \; := \; Q_d \cup \{s\}$
$\quad \Delta_d \; := \; \Delta_d \cup \{f(s_1, \dots, s_n) \to s\}$
$\quad$ where
$\qquad f \in \mathcal{F}_n, s_1 \dots, s_n \in Q_d$
$\qquad s = \{q \in Q \mid \exists q_1 \in s_1, \dots, q_n \in s_n.\ f(q_1, \dots, q_n) \to q \in \Delta\}$
until No more rules can be added to Δ_d
$Q_{df} \; := \; \{s \in Q_d \mid s \cap Q_f \neq \emptyset\}$
$\mathcal{A}_d := (Q_d, \mathcal{F}, Q_{df}, \Delta_d)$

Examples

- Automaton is already deterministic

Examples

- Automaton is already deterministic
  - Naive method generates exponentially many rules

Examples

- Automaton is already deterministic
  - Naive method generates exponentially many rules
  - Reduction method does not increase size of automaton

Examples

- Automaton is already deterministic
  - Naive method generates exponentially many rules
  - Reduction method does not increase size of automaton
- Also advantageous if automaton is „almost” deterministic

Examples

- Automaton is already deterministic
  - Naive method generates exponentially many rules
  - Reduction method does not increase size of automaton
- Also advantageous if automaton is „almost” deterministic
- But, exponential blowup not avoidable in general

Examples

- Let $\mathcal{F} = f/1, g/1, a/0$

Examples

- Let $\mathcal{F} = f/1, g/1, a/0$
- Consider the language $L_n := \{t \in T(\mathcal{F}) \mid \text{The } n\text{th symbol of } t \text{ is } f\}$

Examples

- Let $\mathcal{F} = f/1, g/1, a/0$
- Consider the language $L_n := \{t \in T(\mathcal{F}) \mid \text{The } n\text{th symbol of } t \text{ is } f\}$
  - Automaton $Q = \{q, q_1, \ldots, q_n\}$, $Q_f = \{q_n\}$ and Δ

$$
\begin{array}{ccc}
a \to q & f(q) \to q & g(q) \to q \\
f(q) \to q_1 & & \\
f(q_i) \to q_{i+1} & g(q_i) \to q_{i+1} & \text{for } i < n
\end{array}
$$

Examples

- Let $\mathcal{F} = f/1, g/1, a/0$
- Consider the language $L_n := \{t \in T(\mathcal{F}) \mid \text{The } n\text{th symbol of } t \text{ is } f\ \}$
  - Automaton $Q = \{q, q_1, \ldots, q_n\}$, $Q_f = \{q_n\}$ and Δ

$$\begin{array}{ccc} a \to q & f(q) \to q & g(q) \to q \\ f(q) \to q_1 & & \\ f(q_i) \to q_{i+1} & g(q_i) \to q_{i+1} & \text{for } i < n \end{array}$$

  - Nondeterministically decides which symbol to count

Examples

- Let $\mathcal{F} = f/1, g/1, a/0$
- Consider the language $L_n := \{t \in T(\mathcal{F}) \mid \text{The } n\text{th symbol of } t \text{ is } f\ \}$
  - Automaton $Q = \{q, q_1, \ldots, q_n\}$, $Q_f = \{q_n\}$ and Δ

$$
\begin{array}{ccc}
a \to q & f(q) \to q & g(q) \to q \\
f(q) \to q_1 & & \\
f(q_i) \to q_{i+1} & g(q_i) \to q_{i+1} & \text{for } i < n
\end{array}
$$

  - Nondeterministically decides which symbol to count
- However, any DFTA has to memorize the last n symbols

Examples

- Let $\mathcal{F} = f/1, g/1, a/0$
- Consider the language $L_n := \{t \in T(\mathcal{F}) \mid \text{The } n\text{th symbol of } t \text{ is } f\ \}$
  - Automaton $Q = \{q, q_1, \ldots, q_n\}$, $Q_f = \{q_n\}$ and Δ

$$
\begin{array}{ccc}
a \rightarrow q & f(q) \rightarrow q & g(q) \rightarrow q \\
f(q) \rightarrow q_1 & & \\
f(q_i) \rightarrow q_{i+1} & g(q_i) \rightarrow q_{i+1} & \text{for } i < n
\end{array}
$$

  - Nondeterministically decides which symbol to count
- However, any DFTA has to memorize the last n symbols
  - Thus, it has at least 2^n states

Examples

- Let $\mathcal{F} = f/1, g/1, a/0$
- Consider the language $L_n := \{t \in T(\mathcal{F}) \mid \text{The } n\text{th symbol of } t \text{ is } f\}$
  - Automaton $Q = \{q, q_1, \ldots, q_n\}$, $Q_f = \{q_n\}$ and Δ

$$\begin{array}{ccc} a \to q & f(q) \to q & g(q) \to q \\ f(q) \to q_1 & & \\ f(q_i) \to q_{i+1} & g(q_i) \to q_{i+1} & \text{for } i < n \end{array}$$

  - Nondeterministically decides which symbol to count
- However, any DFTA has to memorize the last n symbols
  - Thus, it has at least 2^n states
- Note: The same example is usually given for word automata

Examples

- Let $\mathcal{F} = f/1, g/1, a/0$
- Consider the language $L_n := \{t \in T(\mathcal{F}) \mid \text{The } n\text{th symbol of } t \text{ is } f\ \}$
  - Automaton $Q = \{q, q_1, \ldots, q_n\}$, $Q_f = \{q_n\}$ and Δ

$$\begin{array}{rcc} a \to q & f(q) \to q & g(q) \to q \\ f(q) \to q_1 & & \\ f(q_i) \to q_{i+1} & g(q_i) \to q_{i+1} & \text{for } i < n \end{array}$$

  - Nondeterministically decides which symbol to count
- However, any DFTA has to memorize the last n symbols
  - Thus, it has at least 2^n states
- Note: The same example is usually given for word automata
  - $L = (a+b)^* a(a+b)^n$

Table of Contents

Example

- Consider the language $L := \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$

Example

- Consider the language $L := \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$
- Not recognizable by an FTA.

Example

- Consider the language $L := \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$
- Not recognizable by an FTA.
- Assume we have $\mathcal{A}$ with $L(\mathcal{A}) = L$ and $|Q| = n$

Example

- Consider the language $L := \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$
- Not recognizable by an FTA.
- Assume we have $\mathcal{A}$ with $L(\mathcal{A}) = L$ and $|Q| = n$
- During recognizing $g^{n+1}(a)$, the same state must occur twice, say
  - $g^i(a) \to_{\mathcal{A}} q$ and $g^j(a) \to_{\mathcal{A}} q$ for $i \neq j$

Example

- Consider the language $L := \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$
- Not recognizable by an FTA.
- Assume we have $\mathcal{A}$ with $L(\mathcal{A}) = L$ and $|Q| = n$
- During recognizing $g^{n+1}(a)$, the same state must occur twice, say
    - $g^i(a) \rightarrow_{\mathcal{A}} q$ and $g^j(a) \rightarrow_{\mathcal{A}} q$ for $i \neq j$
- As $f(g^i(a), g^i(a)) \in L(\mathcal{A})$, we also have $f(g^i(a), g^j(a)) \in L(\mathcal{A})$

Example

- Consider the language $L := \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$
- Not recognizable by an FTA.
- Assume we have $\mathcal{A}$ with $L(\mathcal{A}) = L$ and $|Q| = n$
- During recognizing $g^{n+1}(a)$, the same state must occur twice, say
  - $g^i(a) \rightarrow_{\mathcal{A}} q$ and $g^j(a) \rightarrow_{\mathcal{A}} q$ for $i \neq j$
- As $f(g^i(a), g^i(a)) \in L(\mathcal{A})$, we also have $f(g^i(a), g^j(a)) \in L(\mathcal{A})$
- Contradiction! L not tree-regular

Towards a Pumping Lemma

- A term $t \in T(\mathcal{F}, \mathcal{X})$ is called linear, if no variable occurs more than once

Towards a Pumping Lemma

- A term $t \in T(\mathcal{F}, \mathcal{X})$ is called linear, if no variable occurs more than once
- A context with n holes is a linear term over variables $x_1, \ldots, x_n$

Towards a Pumping Lemma

- A term $t \in T(\mathcal{F}, \mathcal{X})$ is called linear, if no variable occurs more than once
- A context with n holes is a linear term over variables $x_1, \ldots, x_n$
  - For a context C with n holes, we define

$$C[t_1, \ldots, t_n] := C(x_1 \mapsto t_1, \ldots, x_n \mapsto t_n)$$

Towards a Pumping Lemma

- A term $t \in T(\mathcal{F}, \mathcal{X})$ is called linear, if no variable occurs more than once
- A context with n holes is a linear term over variables $x_1, \ldots, x_n$
  - For a context C with n holes, we define

$$C[t_1, \ldots, t_n] := C(x_1 \mapsto t_1, \ldots, x_n \mapsto t_n)$$

- A context that consists of a single variable is called trivial.

Pumping Lemma

Theorem

Let L be a regular language. Then, there is a constant $k > 0$ such that for every $t \in L$ with $Height(t) > k$, there is a context C, a non-trivial context C', and a term u such that

$$t = C[C'[u]] \qquad \forall n \geq 0.\ C[C'^n[u]] \in L$$

- Proof sketch:

Pumping Lemma

Theorem

Let L be a regular language. Then, there is a constant $k > 0$ such that for every $t \in L$ with $Height(t) > k$, there is a context C, a non-trivial context C', and a term u such that

$$t = C[C'[u]] \qquad\qquad \forall n \geq 0.\ C[C'^n[u]] \in L$$

- Proof sketch:
  - Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ with $L = L(\mathcal{A})$, and $t \to_{\mathcal{A}} q, q \in Q_f$

Pumping Lemma

Theorem

Let L be a regular language. Then, there is a constant $k > 0$ such that for every $t \in L$ with $Height(t) > k$, there is a context C, a non-trivial context C', and a term u such that

$$t = C[C'[u]] \qquad \forall n \geq 0.\ C[C'^n[u]] \in L$$

- Proof sketch:
  - Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ with $L = L(\mathcal{A})$, and $t \to_{\mathcal{A}} q, q \in Q_f$
  - Choose path through t with length $> k$

Pumping Lemma

Theorem

Let L be a regular language. Then, there is a constant $k > 0$ such that for every $t \in L$ with $Height(t) > k$, there is a context C, a non-trivial context C', and a term u such that

$$t = C[C'[u]] \qquad \forall n \geq 0.\ C[C'^n[u]] \in L$$

- Proof sketch:
  - Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ with $L = L(\mathcal{A})$, and $t \rightarrow_{\mathcal{A}} q, q \in Q_f$
  - Choose path through t with length $> k$
  - Two subtrees on this path accepted in same state.

Pumping Lemma

Theorem

Let L be a regular language. Then, there is a constant $k > 0$ such that for every $t \in L$ with $Height(t) > k$, there is a context C, a non-trivial context C', and a term u such that

$$t = C[C'[u]] \qquad\qquad \forall n \geq 0.\ C[C'^n[u]] \in L$$

- Proof sketch:
  - Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ with $L = L(\mathcal{A})$, and $t \rightarrow_{\mathcal{A}} q, q \in Q_f$
  - Choose path through t with length $> k$
  - Two subtrees on this path accepted in same state.
  - Identify them by C and C'

Example

- Consider $\mathcal{F} = f/2, a/0$, and $L := \{t \in T(\mathcal{F}) \mid |t| \text{ is prime}\}$

Example

- Consider $\mathcal{F} = f/2, a/0$, and $L := \{t \in T(\mathcal{F}) \mid |t| \text{ is prime}\}$
  - $|t|$ is number of nodes in t

Example

- Consider $\mathcal{F} = f/2, a/0$, and $L := \{t \in T(\mathcal{F}) \mid |t| \text{ is prime}\}$
  - $|t|$ is number of nodes in t
- L is not regular.

Example

- Consider $\mathcal{F} = f/2, a/0$, and $L := \{t \in T(\mathcal{F}) \mid |t| \text{ is prime}\}$
  - $|t|$ is number of nodes in t
- L is not regular.
  - Proof by contradiction. Assume L is regular, and k is pumping constant

Example

- Consider $\mathcal{F} = f/2, a/0$, and $L := \{t \in T(\mathcal{F}) \mid |t| \text{ is prime}\}$
  - $|t|$ is number of nodes in t
- L is not regular.
  - Proof by contradiction. Assume L is regular, and k is pumping constant
  - Choose $t \in L$ with $\mathit{height}(t) > k$

Example

- Consider $\mathcal{F} = f/2, a/0$, and $L := \{t \in T(\mathcal{F}) \mid |t| \text{ is prime}\}$
  - $|t|$ is number of nodes in t
- L is not regular.
  - Proof by contradiction. Assume L is regular, and k is pumping constant
  - Choose $t \in L$ with $\mathit{height}(t) > k$
  - We obtain C, C', u such that $t = C[C'[u]]$ and $\forall n.\ C[C'^n[u]] \in L$

Example

- Consider $\mathcal{F} = f/2, a/0$, and $L := \{t \in T(\mathcal{F}) \mid |t| \text{ is prime}\}$
  - $|t|$ is number of nodes in t
- L is not regular.
  - Proof by contradiction. Assume L is regular, and k is pumping constant
  - Choose $t \in L$ with $\mathit{height}(t) > k$
  - We obtain C, C', u such that $t = C[C'[u]]$ and $\forall n.\ C[C'^n[u]] \in L$
  - We have $|C[C'^n[u]]| = |C| - 1 + n(|C'| - 1) + |u|$

Example

- Consider $\mathcal{F} = f/2, a/0$, and $L := \{t \in T(\mathcal{F}) \mid |t| \text{ is prime}\}$
  - $|t|$ is number of nodes in t
- L is not regular.
  - Proof by contradiction. Assume L is regular, and k is pumping constant
  - Choose $t \in L$ with $height(t) > k$
  - We obtain C, C', u such that $t = C[C'[u]]$ and $\forall n.\ C[C'^n[u]] \in L$
  - We have $|C[C'^n[u]]| = |C| - 1 + n(|C'| - 1) + |u|$
    - Choose $n = |C| + |u| - 1$ to show that this is not prime for all n

Corollaries

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be an FTA.

Corollaries

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be an FTA.
  1. $L(\mathcal{A})$ is non-empty, iff $\exists t \in L(\mathcal{A}).height(t) \leq |Q|$

Corollaries

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be an FTA.
  1. $L(\mathcal{A})$ is non-empty, iff $\exists t \in L(\mathcal{A}).\mathit{height}(t) \leq |Q|$
  2. $L(\mathcal{A})$ is infinite, iff $\exists t \in L(\mathcal{A}).|Q| < \mathit{height}(t) \leq 2|Q|$

Corollaries

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be an FTA.
  1. $L(\mathcal{A})$ is non-empty, iff $\exists t \in L(\mathcal{A}).\mathit{height}(t) \leq |Q|$
  2. $L(\mathcal{A})$ is infinite, iff $\exists t \in L(\mathcal{A}).|Q| < \mathit{height}(t) \leq 2|Q|$
- Proof ideas:

Corollaries

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be an FTA.
  1. $L(\mathcal{A})$ is non-empty, iff $\exists t \in L(\mathcal{A}).height(t) \leq |Q|$
  2. $L(\mathcal{A})$ is infinite, iff $\exists t \in L(\mathcal{A}).|Q| < height(t) \leq 2|Q|$
- Proof ideas:
  1. Remove duplicate states of accepting run repeatedly

Corollaries

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be an FTA.
  1. $L(\mathcal{A})$ is non-empty, iff $\exists t \in L(\mathcal{A}).height(t) \leq |Q|$
  2. $L(\mathcal{A})$ is infinite, iff $\exists t \in L(\mathcal{A}).|Q| < height(t) \leq 2|Q|$
- Proof ideas:
  1. Remove duplicate states of accepting run repeatedly
  2. $\Longrightarrow$: Take $t \in L(\mathcal{A})$ high enough. Remove duplicate states repeatedly, until longest path has exactly one duplication.

Corollaries

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be an FTA.
  1. $L(\mathcal{A})$ is non-empty, iff $\exists t \in L(\mathcal{A}).\mathit{height}(t) \leq |Q|$
  2. $L(\mathcal{A})$ is infinite, iff $\exists t \in L(\mathcal{A}).|Q| < \mathit{height}(t) \leq 2|Q|$
- Proof ideas:
  1. Remove duplicate states of accepting run repeatedly
  2. $\Longrightarrow$: Take $t \in L(\mathcal{A})$ high enough. Remove duplicate states repeatedly, until longest path has exactly one duplication.
     - $\Longleftarrow$: Pump with infinitely many n

Last Lecture

- Deterministic Automata
  - Powerset construction
- Pumping Lemma

Table of Contents

Closure Properties

Theorem

- *The class of regular languages is closed under union, intersection, and complement.*
- *Automata for union, intersection, and complement can be computed.*

Union

- Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.

Union

- Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$

Union

- Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $\mathcal{A} = (Q_1 \cup Q_2, \mathcal{F}, Q_{f1} \cup Q_{f2}, \Delta_1 \cup \Delta_2)$

Union

- Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $\mathcal{A} = (Q_1 \cup Q_2, \mathcal{F}, Q_{f1} \cup Q_{f2}, \Delta_1 \cup \Delta_2)$
  - Straightforward: $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$

Union

- Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $\mathcal{A} = (Q_1 \cup Q_2, \mathcal{F}, Q_{f1} \cup Q_{f2}, \Delta_1 \cup \Delta_2)$
  - Straightforward: $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$
- However: $\mathcal{A}$ may be nondeterministic and not complete, even if $\mathcal{A}_1$ and $\mathcal{A}_2$ were.

Union

- Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $\mathcal{A} = (Q_1 \cup Q_2, \mathcal{F}, Q_{f1} \cup Q_{f2}, \Delta_1 \cup \Delta_2)$
  - Straightforward: $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$
- However: $\mathcal{A}$ may be nondeterministic and not complete, even if $\mathcal{A}_1$ and $\mathcal{A}_2$ were.
- Let $\mathcal{A}_1, \mathcal{A}_2$ be deterministic and complete. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ with

Union

- Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $\mathcal{A} = (Q_1 \cup Q_2, \mathcal{F}, Q_{f1} \cup Q_{f2}, \Delta_1 \cup \Delta_2)$
  - Straightforward: $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$
- However: $\mathcal{A}$ may be nondeterministic and not complete, even if $\mathcal{A}_1$ and $\mathcal{A}_2$ were.
- Let $\mathcal{A}_1, \mathcal{A}_2$ be deterministic and complete. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ with
  - $Q = Q_1 \times Q_2$, $Q_f = Q_{f1} \times Q_2 \cup Q_1 \times Q_{f2}$, and $\Delta = \Delta_1 \times \Delta_2$ where

$$\Delta_1 \times \Delta_2 := \{f((q_1, q_1'), \ldots, (q_n, q_n')) \to (q, q') \mid \\ f(q_1, \ldots, q_n) \to q \in \Delta_1 \land f(q_1', \ldots, q_n') \to q' \in \Delta_2\}$$

Union

- Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $\mathcal{A} = (Q_1 \cup Q_2, \mathcal{F}, Q_{f1} \cup Q_{f2}, \Delta_1 \cup \Delta_2)$
  - Straightforward: $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$
- However: $\mathcal{A}$ may be nondeterministic and not complete, even if $\mathcal{A}_1$ and $\mathcal{A}_2$ were.
- Let $\mathcal{A}_1, \mathcal{A}_2$ be deterministic and complete. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ with
  - $Q = Q_1 \times Q_2$, $Q_f = Q_{f1} \times Q_2 \cup Q_1 \times Q_{f2}$, and $\Delta = \Delta_1 \times \Delta_2$ where

$$\Delta_1 \times \Delta_2 := \{f((q_1, q_1'), \ldots, (q_n, q_n')) \to (q, q') \mid \\ f(q_1, \ldots, q_n) \to q \in \Delta_1 \wedge f(q_1', \ldots, q_n') \to q' \in \Delta_2\}$$

  - Then $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ and $\mathcal{A}$ is deterministic and complete.

Union

- Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Assume, wlog, $Q_1 \cap Q_2 = \emptyset$
  - Let $\mathcal{A} = (Q_1 \cup Q_2, \mathcal{F}, Q_{f1} \cup Q_{f2}, \Delta_1 \cup \Delta_2)$
  - Straightforward: $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$
- However: $\mathcal{A}$ may be nondeterministic and not complete, even if $\mathcal{A}_1$ and $\mathcal{A}_2$ were.
- Let $\mathcal{A}_1, \mathcal{A}_2$ be deterministic and complete. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ with
  - $Q = Q_1 \times Q_2$, $Q_f = Q_{f1} \times Q_2 \cup Q_1 \times Q_{f2}$, and $\Delta = \Delta_1 \times \Delta_2$ where

$$\begin{aligned}\Delta_1 \times \Delta_2 := \{f((q_1, q_1'), \dots, (q_n, q_n')) \to (q, q') \mid \\ f(q_1, \dots, q_n) \to q \in \Delta_1 \wedge f(q_1', \dots, q_n') \to q' \in \Delta_2\}\end{aligned}$$

  - Then $L(\mathcal{A}) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ and $\mathcal{A}$ is deterministic and complete.
  - Intuition: Recognize with both automata in parallel.

Complement

- Assume L is recognized by the complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$

Complement

- Assume L is recognized by the complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
- Define $\mathcal{A}^c = (Q, \mathcal{F}, Q \setminus Q_f, \Delta)$

Complement

- Assume L is recognized by the complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
- Define $\mathcal{A}^c = (Q, \mathcal{F}, Q \setminus Q_f, \Delta)$
- Obviously, $L(\mathcal{A}^c) = T(\mathcal{F}) \setminus L(\mathcal{A})$

Complement

- Assume L is recognized by the complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
- Define $\mathcal{A}^c = (Q, \mathcal{F}, Q \setminus Q_f, \Delta)$
- Obviously, $L(\mathcal{A}^c) = T(\mathcal{F}) \setminus L(\mathcal{A})$
- If a nondeterministic automaton is given, determinization may cause exponential blowup

Intersection

- The easy way: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

Intersection

- The easy way: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
  - Exponential blowup for NFTA.

Intersection

- The easy way: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
  - Exponential blowup for NFTA.
- Product construction: Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.

Intersection

- The easy way: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
  - Exponential blowup for NFTA.
- Product construction: Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Define $\mathcal{A} = (Q_1 \times Q_2, \mathcal{F}, Q_{f1} \times Q_{f2}, \Delta_1 \times \Delta_2)$

Intersection

- The easy way: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
  - Exponential blowup for NFTA.
- Product construction: Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Define $\mathcal{A} = (Q_1 \times Q_2, \mathcal{F}, Q_{f1} \times Q_{f2}, \Delta_1 \times \Delta_2)$
  - $L(\mathcal{A}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$

Intersection

- The easy way: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
  - Exponential blowup for NFTA.
- Product construction: Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Define $\mathcal{A} = (Q_1 \times Q_2, \mathcal{F}, Q_{f1} \times Q_{f2}, \Delta_1 \times \Delta_2)$
  - $L(\mathcal{A}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$
    - Intuition: Automata run in parallel. Accept if both accept.

Intersection

- The easy way: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
  - Exponential blowup for NFTA.
- Product construction: Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Define $\mathcal{A} = (Q_1 \times Q_2, \mathcal{F}, Q_{f1} \times Q_{f2}, \Delta_1 \times \Delta_2)$
  - $L(\mathcal{A}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$
    - Intuition: Automata run in parallel. Accept if both accept.
  - $\mathcal{A}$ is deterministic/complete if $\mathcal{A}_1$ and $\mathcal{A}_2$ are.

Intersection

- The easy way: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
  - Exponential blowup for NFTA.
- Product construction: Given automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$.
  - Define $\mathcal{A} = (Q_1 \times Q_2, \mathcal{F}, Q_{f1} \times Q_{f2}, \Delta_1 \times \Delta_2)$
  - $L(\mathcal{A}) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$
    - Intuition: Automata run in parallel. Accept if both accept.
  - $\mathcal{A}$ is deterministic/complete if $\mathcal{A}_1$ and $\mathcal{A}_2$ are.
- Product construction can also be combined with reduction algorithm, to avoid construction of inaccessible states.

Summary

- For DFTA: Polynomial time intersection, union, complement

Summary

- For DFTA: Polynomial time intersection, union, complement
- For NFTA: Polynomial time intersection, union. Exp-time complement.

More Algorithms on FTA

- Membership for NFTA. In time $O(|t| * |\mathcal{A}|)$ On-the-fly determinization.

More Algorithms on FTA

- Membership for NFTA. In time $O(|t| * |\mathcal{A}|)$ On-the-fly determinization.
- Emptiness check: Time $O(|\mathcal{A}|)$. Exercise!

Table of Contents

Tree Homomorphisms

- Map each symbol of tree to new subtree

Tree Homomorphisms

- Map each symbol of tree to new subtree
- Example: Convert ternary tree to binary tree
  - $f(x_1, x_2, x_3) \mapsto g(x_1, g(x_2, x_3))$

Tree Homomorphisms

- Map each symbol of tree to new subtree
- Example: Convert ternary tree to binary tree
  - $f(x_1, x_2, x_3) \mapsto g(x_1, g(x_2, x_3))$
- Example: Eliminate conjunction from Boolean formulas
  - $x_1 \wedge x_2 \mapsto \neg(\neg x_1 \vee \neg x_2)$

Formal definition

- Let $\mathcal{F}$ and $\mathcal{F}'$ be ranked alphabets, not necessarily disjoint

Formal definition

- Let $\mathcal{F}$ and $\mathcal{F}'$ be ranked alphabets, not necessarily disjoint
- Let, for any n, $\mathcal{X}_n := \{x_1, \ldots, x_n\}$ be variables, disjoint from $\mathcal{F}$ and $\mathcal{F}'$

Formal definition

- Let $\mathcal{F}$ and $\mathcal{F}'$ be ranked alphabets, not necessarily disjoint
- Let, for any n, $\mathcal{X}_n := \{x_1, \ldots, x_n\}$ be variables, disjoint from $\mathcal{F}$ and $\mathcal{F}'$
- Let $h_\mathcal{F}$ be a mapping that maps $f \in \mathcal{F}_n$ to $h_\mathcal{F}(f) \in T(\mathcal{F}', \mathcal{X}_n)$

Formal definition

- Let $\mathcal{F}$ and $\mathcal{F}'$ be ranked alphabets, not necessarily disjoint
- Let, for any n, $\mathcal{X}_n := \{x_1, \ldots, x_n\}$ be variables, disjoint from $\mathcal{F}$ and $\mathcal{F}'$
- Let $h_{\mathcal{F}}$ be a mapping that maps $f \in \mathcal{F}_n$ to $h_{\mathcal{F}}(f) \in T(\mathcal{F}', \mathcal{X}_n)$
- $h_{\mathcal{F}}$ determines a *tree homomorphism* $h : T(\mathcal{F}) \to T(\mathcal{F}')$:

$$h(f(t_1, \ldots, t_n)) := h_{\mathcal{F}}(f)(x_1 \mapsto h(t_1), \ldots, x_n \mapsto h(t_n))$$

Preservation of Regularity

- Tree homomorphisms do not preserve regularity in general

Preservation of Regularity

- Tree homomorphisms do not preserve regularity in general
  - Let $L = \{f(g^i(a)) \mid i \in \mathbb{N}\}$. Obviously regular.

Preservation of Regularity

- Tree homomorphisms do not preserve regularity in general
    - Let $L = \{f(g^i(a)) \mid i \in \mathbb{N}\}$. Obviously regular.
    - Let $h_{\mathcal{F}}\colon f(x) \mapsto f(x, x)$

Preservation of Regularity

- Tree homomorphisms do not preserve regularity in general
  - Let $L = \{f(g^i(a)) \mid i \in \mathbb{N}\}$. Obviously regular.
  - Let $h_{\mathcal{F}}$: $f(x) \mapsto f(x, x)$
  - $h(L) = \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$. Not regular.

Preservation of Regularity

- Tree homomorphisms do not preserve regularity in general
  - Let $L = \{f(g^i(a)) \mid i \in \mathbb{N}\}$. Obviously regular.
  - Let $h_{\mathcal{F}}$: $f(x) \mapsto f(x, x)$
  - $h(L) = \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$. Not regular.
- But:

Preservation of Regularity

- Tree homomorphisms do not preserve regularity in general
  - Let $L = \{f(g^i(a)) \mid i \in \mathbb{N}\}$. Obviously regular.
  - Let $h_{\mathcal{F}}$: $f(x) \mapsto f(x, x)$
  - $h(L) = \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$. Not regular.
- But:
  - A tree homomorphism determined by $h_{\mathcal{F}}$ is *linear*, iff for all $f \in \mathcal{F}$, the term $h_{\mathcal{F}}(f)$ is linear.

Preservation of Regularity

- Tree homomorphisms do not preserve regularity in general
  - Let $L = \{f(g^i(a)) \mid i \in \mathbb{N}\}$. Obviously regular.
  - Let $h_{\mathcal{F}}$: $f(x) \mapsto f(x, x)$
  - $h(L) = \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$. Not regular.
- But:
  - A tree homomorphism determined by $h_{\mathcal{F}}$ is *linear*, iff for all $f \in \mathcal{F}$, the term $h_{\mathcal{F}}(f)$ is linear.

Theorem

Let L be a regular language, and h a linear tree homomorphism. Then $h(L)$ is also regular.

Preservation of Regularity

- Tree homomorphisms do not preserve regularity in general
  - Let $L = \{f(g^i(a)) \mid i \in \mathbb{N}\}$. Obviously regular.
  - Let $h_{\mathcal{F}}$: $f(x) \mapsto f(x, x)$
  - $h(L) = \{f(g^i(a), g^i(a)) \mid i \in \mathbb{N}\}$. Not regular.
- But:
  - A tree homomorphism determined by $h_{\mathcal{F}}$ is *linear*, iff for all $f \in \mathcal{F}$, the term $h_{\mathcal{F}}(f)$ is linear.

Theorem

Let L be a regular language, and h a linear tree homomorphism. Then $h(L)$ is also regular.

- Proof idea: For each original rule $f(q_1, \ldots, q_n)$, insert rules that recognize $h_{\mathcal{F}}[q_1, \ldots, q_n]$

Positions

- Identify position in tree by sequence of natural numbers

Positions

- Identify position in tree by sequence of natural numbers
- Let t be a tree, and $p \in \mathbb{N}^*$. We define the subtree of t at position p by:

$$t(\varepsilon) := t \qquad\qquad (f(t_1, \ldots, t_n))(ip) := t_i(p)$$

Positions

- Identify position in tree by sequence of natural numbers
- Let t be a tree, and $p \in \mathbb{N}^*$. We define the subtree of t at position p by:

$$t(\varepsilon) := t \qquad\qquad (f(t_1, \dots, t_n))(ip) := t_i(p)$$

- $Pos(t)$ is the set of valid positions in t

Construction (Preservation of regularity)

- Assume L is accepted by reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$.

Construction (Preservation of regularity)

- Assume L is accepted by reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$.
- Construct NFTA $A' = (Q', \mathcal{F}', Q'_f, \Delta')$:

Construction (Preservation of regularity)

- Assume L is accepted by reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$.
- Construct NFTA $A' = (Q', \mathcal{F}', Q'_f, \Delta')$:
  - With $Q \subseteq Q'$ and $Q'_f = Q_f$

Construction (Preservation of regularity)

- Assume L is accepted by reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$.
- Construct NFTA $A' = (Q', \mathcal{F}', Q'_f, \Delta')$:
  - With $Q \subseteq Q'$ and $Q'_f = Q_f$
  - For each rule $r = f(q_1, \ldots, q_n) \to q$, $t_f = h_{\mathcal{F}}(t)$, and position $p \in Pos(t_f)$:

Construction (Preservation of regularity)

- Assume L is accepted by reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$.
- Construct NFTA $A' = (Q', \mathcal{F}', Q'_f, \Delta')$:
  - With $Q \subseteq Q'$ and $Q'_f = Q_f$
  - For each rule $r = f(q_1, \ldots, q_n) \to q$, $t_f = h_{\mathcal{F}}(t)$, and position $p \in Pos(t_f)$:
    - States $q^r_p \in Q'$

Construction (Preservation of regularity)

- Assume L is accepted by reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$.
- Construct NFTA $A' = (Q', \mathcal{F}', Q'_f, \Delta')$:
  - With $Q \subseteq Q'$ and $Q'_f = Q_f$
  - For each rule $r = f(q_1, \ldots, q_n) \to q$, $t_f = h_{\mathcal{F}}(t)$, and position $p \in Pos(t_f)$:
    - States $q^r_p \in Q'$
    - If $t_f(p) = g(\ldots) \in \mathcal{F}_k$: $g(q^r_{p1}, \ldots, q^r_{pk}) \to q^r \in \Delta'$

Construction (Preservation of regularity)

- Assume L is accepted by reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$.
- Construct NFTA $A' = (Q', \mathcal{F}', Q_f', \Delta')$:
  - With $Q \subseteq Q'$ and $Q_f' = Q_f$
  - For each rule $r = f(q_1, \ldots, q_n) \to q$, $t_f = h_{\mathcal{F}}(t)$, and position $p \in Pos(t_f)$:
    - States $q_p^r \in Q'$
    - If $t_f(p) = g(\ldots) \in \mathcal{F}_k$: $g(q_{p1}^r, \ldots, q_{pk}^r) \to q^r \in \Delta'$
    - If $t_f(p) = x_i$: $q_i \to q_p^r \in \Delta'$

Construction (Preservation of regularity)

- Assume L is accepted by reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$.
- Construct NFTA $A' = (Q', \mathcal{F}', Q'_f, \Delta')$:
  - With $Q \subseteq Q'$ and $Q'_f = Q_f$
  - For each rule $r = f(q_1, \ldots, q_n) \to q$, $t_f = h_{\mathcal{F}}(t)$, and position $p \in Pos(t_f)$:
    - States $q^r_p \in Q'$
    - If $t_f(p) = g(\ldots) \in \mathcal{F}_k$: $g(q^r_{p1}, \ldots, q^r_{pk}) \to q^r \in \Delta'$
    - If $t_f(p) = x_i$: $q_i \to q^r_p \in \Delta'$
    - $q^r_\varepsilon \to q \in \Delta'$

Proof sketch

- Prove $h(L) \subseteq L(\mathcal{A}')$. Straightforward.

Proof sketch

- Prove $h(L) \subseteq L(\mathcal{A}')$. Straightforward.
- Prove $L(\mathcal{A}') \subseteq h(L)$ (Sketch on board).

Proof sketch

- Prove $h(L) \subseteq L(\mathcal{A}')$. Straightforward.
- Prove $L(\mathcal{A}') \subseteq h(L)$ (Sketch on board).
  - Idea: Split derivation of $t \rightarrow_{\mathcal{A}'} q \in Q$ at rules of the form $q^r_\varepsilon \rightarrow q$.

Proof sketch

- Prove $h(L) \subseteq L(\mathcal{A}')$. Straightforward.
- Prove $L(\mathcal{A}') \subseteq h(L)$ (Sketch on board).
  - Idea: Split derivation of $t \to_{\mathcal{A}'} q \in Q$ at rules of the form $q_\varepsilon^r \to q$.
  - Assume $r = f(\ldots) \to q$. Without using states from Q, automaton accepts subtree of the form $h_{\mathcal{F}}(f)$.

Proof sketch

- Prove $h(L) \subseteq L(\mathcal{A}')$. Straightforward.
- Prove $L(\mathcal{A}') \subseteq h(L)$ (Sketch on board).
  - Idea: Split derivation of $t \rightarrow_{\mathcal{A}'} q \in Q$ at rules of the form $q_\varepsilon^r \rightarrow q$.
  - Assume $r = f(\ldots) \rightarrow q$. Without using states from Q, automaton accepts subtree of the form $h_{\mathcal{F}}(f)$.
  - Cases:
    - Constant (0-ary symbol)
    - Due to rule $q_i \rightarrow q_p^r \in \Delta'$, $q_i \in Q$ (use IH)

Proof sketch

- Prove $h(L) \subseteq L(\mathcal{A}')$. Straightforward.
- Prove $L(\mathcal{A}') \subseteq h(L)$ (Sketch on board).
  - Idea: Split derivation of $t \rightarrow_{\mathcal{A}'} q \in Q$ at rules of the form $q_\varepsilon^r \rightarrow q$.
  - Assume $r = f(\ldots) \rightarrow q$. Without using states from Q, automaton accepts subtree of the form $h_\mathcal{F}(f)$.
  - Cases:
    - Constant (0-ary symbol)
    - Due to rule $q_i \rightarrow q_p^r \in \Delta'$, $q_i \in Q$ (use IH)
  - Formally: Induction on size of derivation $t \rightarrow_{\mathcal{A}'} q$

Last lecture

- Closure properties: Union, intersection, complement
- Tree homomorphisms
  - Idea: Replace node by tree with „holes"
  - $\mathit{and}(x_1, x_2) \mapsto \mathit{not}(\mathit{or}(\mathit{not}(x_1), \mathit{not}(x_2)))$
- Regular languages closed under *linear* homomorphisms
  - Linear: No subtrees are duplicated

Inverse Homomorphism

- Motivation: Reconsider elimination of $\wedge$ in Boolean formulas

Inverse Homomorphism

- Motivation: Reconsider elimination of $\wedge$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\wedge$.

Inverse Homomorphism

- Motivation: Reconsider elimination of $\wedge$ in Boolean formulas
    - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\wedge$.
        - Not really useful

Inverse Homomorphism

- Motivation: Reconsider elimination of $\wedge$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\wedge$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\wedge$, construct automaton for formulas with $\wedge$.

Inverse Homomorphism

- Motivation: Reconsider elimination of $\wedge$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\wedge$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\wedge$, construct automaton for formulas with $\wedge$.
    - This would be nice

Inverse Homomorphism

- Motivation: Reconsider elimination of $\wedge$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\wedge$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\wedge$, construct automaton for formulas with $\wedge$.
    - This would be nice
    - From automaton for simple language, and mapping of complex to simple language, obtain automaton for complex language!

Inverse Homomorphism

- Motivation: Reconsider elimination of $\wedge$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\wedge$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\wedge$, construct automaton for formulas with $\wedge$.
    - This would be nice
    - From automaton for simple language, and mapping of complex to simple language, obtain automaton for complex language!
- Fortunately

Theorem

Let h be a tree homomorphism, and L a regular language. Then $h^{-1}(L) := \{t \mid h(t) \in L\}$ is regular.

Inverse Homomorphism

- Motivation: Reconsider elimination of $\wedge$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\wedge$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\wedge$, construct automaton for formulas with $\wedge$.
    - This would be nice
    - From automaton for simple language, and mapping of complex to simple language, obtain automaton for complex language!
- Fortunately

Theorem

Let h be a tree homomorphism, and L a regular language. Then $h^{-1}(L) := \{t \mid h(t) \in L\}$ is regular.

- Also holds for non-linear homomorphisms

Inverse Homomorphism

- Motivation: Reconsider elimination of $\wedge$ in Boolean formulas
  - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\wedge$.
    - Not really useful
  - Inverse homomorphism: Given automaton for formulas without $\wedge$, construct automaton for formulas with $\wedge$.
    - This would be nice
    - From automaton for simple language, and mapping of complex to simple language, obtain automaton for complex language!
- Fortunately

Theorem

Let h be a tree homomorphism, and L a regular language. Then $h^{-1}(L) := \{t \mid h(t) \in L\}$ is regular.

  - Also holds for non-linear homomorphisms
- Common technique to show regularity/decidability

Inverse Homomorphism

- Motivation: Reconsider elimination of $\wedge$ in Boolean formulas
    - Homomorphism: Given automaton that recognizes true formulas, construct automaton for true formulas without $\wedge$.
        - Not really useful
    - Inverse homomorphism: Given automaton for formulas without $\wedge$, construct automaton for formulas with $\wedge$.
        - This would be nice
        - From automaton for simple language, and mapping of complex to simple language, obtain automaton for complex language!
- Fortunately

Theorem

Let h be a tree homomorphism, and L a regular language. Then $h^{-1}(L) := \{t \mid h(t) \in L\}$ is regular.

- Also holds for non-linear homomorphisms
- Common technique to show regularity/decidability
    - Can be generalized to (macro) tree transducers

Generalized Acceptance Relation

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ and $t \in T(\mathcal{F} \dot\cup Q)$.

Generalized Acceptance Relation

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ and $t \in T(\mathcal{F} \dot\cup Q)$.
- We define $t \to_{\mathcal{A}} q$ as the least relation that satisfies

$$q \to_{\mathcal{A}} q$$
$$f(q_1, \ldots, q_n) \to q \in \Delta, \forall i \leq n.\ t_i \to_{\mathcal{A}} q_i \implies f(t_1, \ldots, t_n) \to_{\mathcal{A}} q$$

Generalized Acceptance Relation

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ and $t \in T(\mathcal{F} \dot{\cup} Q)$.
- We define $t \to_{\mathcal{A}} q$ as the least relation that satisfies

$$q \to_{\mathcal{A}} q$$
$$f(q_1, \ldots, q_n) \to q \in \Delta, \forall i \leq n.\ t_i \to_{\mathcal{A}} q_i \implies f(t_1, \ldots, t_n) \to_{\mathcal{A}} q$$

- This is obviously a generalization of the acceptance relation we defined earlier

Inverse Homomorphism, construction

- Let $h : T(\mathcal{F}) \to T(\mathcal{F}')$ be a tree homomorphism determined by $h_{\mathcal{F}}$

Inverse Homomorphism, construction

- Let $h : T(\mathcal{F}) \to T(\mathcal{F}')$ be a tree homomorphism determined by $h_{\mathcal{F}}$
- Let $\mathcal{A}' = (Q', \mathcal{F}', Q'_f, \Delta')$ be a DFTA with $L = L(\mathcal{A}')$

Inverse Homomorphism, construction

- Let $h : T(\mathcal{F}) \to T(\mathcal{F}')$ be a tree homomorphism determined by $h_{\mathcal{F}}$
- Let $\mathcal{A}' = (Q', \mathcal{F}', Q'_f, \Delta')$ be a DFTA with $L = L(\mathcal{A}')$
- We define DFTA $\mathcal{A} = (Q' \mathbin{\dot\cup} \{s\}, \mathcal{F}, Q'_f, \Delta)$, with the rules

$$f(q_1, \dots, q_n) \to q \in \Delta \text{ if } f \in \mathcal{F}_n,\ h_{\mathcal{F}}(f)[p_1, \dots, p_n] \to_{\mathcal{A}'} q$$
$$\quad \text{where } q_i = p_i \text{ if } x_i \text{ occurs in } h_{\mathcal{F}}(f), \text{ and } q_i = s \text{ otherwise}$$
$$a \to s \in \Delta, \quad f(s, \dots, s) \to s \in \Delta$$

Inverse Homomorphism, construction

- Let $h : T(\mathcal{F}) \to T(\mathcal{F}')$ be a tree homomorphism determined by $h_{\mathcal{F}}$
- Let $\mathcal{A}' = (Q', \mathcal{F}', Q'_f, \Delta')$ be a DFTA with $L = L(\mathcal{A}')$
- We define DFTA $\mathcal{A} = (Q' \,\dot{\cup}\, \{s\}, \mathcal{F}, Q'_f, \Delta)$, with the rules

$$f(q_1, \ldots, q_n) \to q \in \Delta \text{ if } f \in \mathcal{F}_n,\ h_{\mathcal{F}}(f)[p_1, \ldots, p_n] \to_{\mathcal{A}'} q$$
$$\text{where } q_i = p_i \text{ if } x_i \text{ occurs in } h_{\mathcal{F}}(f), \text{ and } q_i = s \text{ otherwise}$$
$$a \to s \in \Delta, \quad f(s, \ldots, s) \to s \in \Delta$$

  - Intuition: Accept node f, if its image is accepted by $\mathcal{A}'$

Inverse Homomorphism, construction

- Let $h : T(\mathcal{F}) \to T(\mathcal{F}')$ be a tree homomorphism determined by $h_\mathcal{F}$
- Let $\mathcal{A}' = (Q', \mathcal{F}', Q'_f, \Delta')$ be a DFTA with $L = L(\mathcal{A}')$
- We define DFTA $\mathcal{A} = (Q' \dot\cup \{s\}, \mathcal{F}, Q'_f, \Delta)$, with the rules

$$f(q_1, \dots, q_n) \to q \in \Delta \text{ if } f \in \mathcal{F}_n,\ h_\mathcal{F}(f)[p_1, \dots, p_n] \to_{\mathcal{A}'} q$$
$$\text{where } q_i = p_i \text{ if } x_i \text{ occurs in } h_\mathcal{F}(f), \text{ and } q_i = s \text{ otherwise}$$
$$a \to s \in \Delta, \quad f(s, \dots, s) \to s \in \Delta$$

- Intuition: Accept node f, if its image is accepted by $\mathcal{A}'$
  - If image does not depend on a subtree, accept any subtree (state s)

Inverse Homomorphism, proof

- Show $t \to_{\mathcal{A}} q$ iff $h(t) \to_{\mathcal{A}'} q$

Inverse Homomorphism, proof

- Show $t \rightarrow_{\mathcal{A}} q$ iff $h(t) \rightarrow_{\mathcal{A}'} q$
- On board

Table of Contents

Last Lecture

- Inverse homomorphisms preserve regularity
- Started Myhill-Nerode Theorem

Reminder: Equivalence relation

- A relation $\equiv \subseteq A \times A$ is called *equivalence relation*, iff it is reflexive, transitive and symmetric

Reminder: Equivalence relation

- A relation $\equiv \subseteq A \times A$ is called *equivalence relation*, iff it is reflexive, transitive and symmetric
- The set $[a]_{\equiv} := \{a' \mid a \equiv a'\}$ is called the *equivalence class* of a

Reminder: Equivalence relation

- A relation $\equiv \subseteq A \times A$ is called *equivalence relation*, iff it is reflexive, transitive and symmetric
- The set $[a]_{\equiv} := \{a' \mid a \equiv a'\}$ is called the *equivalence class* of a
- An equivalence relation is of *finite index*, if there are only finitely many equivalence classes

Congruence

- An equivalence relation $\equiv$ on $T(\mathcal{F})$ is a *congruence*, iff

$$\forall f \in \mathcal{F}_n.\ (\forall i \leq n.\ u_i \equiv v_i) \implies f(u_1, \ldots, u_n) \equiv f(v_1, \ldots, v_n)$$

Congruence

- An equivalence relation $\equiv$ on $T(\mathcal{F})$ is a *congruence*, iff

$$\forall f \in \mathcal{F}_n.\ (\forall i \leq n.\ u_i \equiv v_i) \implies f(u_1, \ldots, u_n) \equiv f(v_1, \ldots, v_n)$$

  - Intuition: Functions are equivalent if applied to equivalent arguments.

Congruence

- An equivalence relation $\equiv$ on $T(\mathcal{F})$ is a *congruence*, iff

$$\forall f \in \mathcal{F}_n.\ (\forall i \leq n.\ u_i \equiv v_i) \implies f(u_1, \ldots, u_n) \equiv f(v_1, \ldots, v_n)$$

  - Intuition: Functions are equivalent if applied to equivalent arguments.
  - Note: $\equiv$ is congruence, iff closed under (1-hole) contexts, i.e.

$$\forall C\ u\ v.\ u \equiv v \implies C[u] \equiv C[v]$$

Congruence

- An equivalence relation $\equiv$ on $T(\mathcal{F})$ is a *congruence*, iff

$$\forall f \in \mathcal{F}_n.\ (\forall i \leq n.\ u_i \equiv v_i) \implies f(u_1, \ldots, u_n) \equiv f(v_1, \ldots, v_n)$$

  - Intuition: Functions are equivalent if applied to equivalent arguments.
  - Note: $\equiv$ is congruence, iff closed under (1-hole) contexts, i.e.

$$\forall C\ u\ v.\ u \equiv v \implies C[u] \equiv C[v]$$

- For a language L, we define the congruence $\equiv_L$ by

$$u \equiv_L v \text{ iff } \forall C.\ C[u] \in L \text{ iff } C[v] \in L$$

Congruence

- An equivalence relation $\equiv$ on $T(\mathcal{F})$ is a *congruence*, iff

$$\forall f \in \mathcal{F}_n.\ (\forall i \leq n.\ u_i \equiv v_i) \implies f(u_1, \dots, u_n) \equiv f(v_1, \dots, v_n)$$

  - Intuition: Functions are equivalent if applied to equivalent arguments.
  - Note: $\equiv$ is congruence, iff closed under (1-hole) contexts, i.e.

$$\forall C\ u\ v.\ u \equiv v \implies C[u] \equiv C[v]$$

- For a language L, we define the congruence $\equiv_L$ by

$$u \equiv_L v \text{ iff } \forall C.\ C[u] \in L \text{ iff } C[v] \in L$$

  - Obviously an equivalence relation. Obviously a congruence.

Congruence

- An equivalence relation $\equiv$ on $T(\mathcal{F})$ is a *congruence*, iff

$$\forall f \in \mathcal{F}_n.\ (\forall i \leq n.\ u_i \equiv v_i) \implies f(u_1, \ldots, u_n) \equiv f(v_1, \ldots, v_n)$$

  - Intuition: Functions are equivalent if applied to equivalent arguments.
  - Note: $\equiv$ is congruence, iff closed under (1-hole) contexts, i.e.

$$\forall C\ u\ v.\ u \equiv v \implies C[u] \equiv C[v]$$

- For a language L, we define the congruence $\equiv_L$ by

$$u \equiv_L v \text{ iff } \forall C.\ C[u] \in L \text{ iff } C[v] \in L$$

  - Obviously an equivalence relation. Obviously a congruence.
  - Intuition: L does not distinguish between u and v

Myhill-Nerode Theorem

Theorem

The following statements are equivalent

1. *L is a regular tree language*

Myhill-Nerode Theorem

Theorem

The following statements are equivalent

1. *L is a regular tree language*
2. *L is the union of some equivalence classes of a finite-index congruence*

Myhill-Nerode Theorem

Theorem

The following statements are equivalent

1. *L is a regular tree language*
2. *L is the union of some equivalence classes of a finite-index congruence*
3. *$\equiv_L$ is of finite index*

Convention

- Complete DFTAs are written as $(Q, \mathcal{F}, Q_f, \delta)$
  - with $\delta : (\mathcal{F}_n \times Q^n \to Q)_n$
  - Corresponds to Δ via

$$f(q_1, \dots, q_n) \to q \text{ iff } \delta(f, q_1, \dots, q_n) = q$$

  - Naturally extended to trees

$$\delta(f(t_1, \dots, t_n) = \delta(f, \delta(t_1), \dots, \delta(t_n))$$

  - Compatible with $\to_{\mathcal{A}}$, i.e.

$$t \to_{\mathcal{A}} q \text{ iff } \delta(t) = q$$

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \rightarrow 2$

- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \rightarrow 2$

- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \rightarrow 2$

- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \rightarrow 2$

- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \to 2$

- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \to 2$
- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \to 3$
- Let R be the finite-index congruence. Assume uRv.

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \rightarrow 2$

- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \rightarrow 3$

- Let R be the finite-index congruence. Assume uRv.
- Then, $C[u]RC[v]$ for all contexts C

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \to 2$

- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \to 3$

- Let R be the finite-index congruence. Assume uRv.
- Then, $C[u]RC[v]$ for all contexts C
- As L is union of eq-classes of R, we have $C[u] \in L$ iff $C[v] \in L$

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \rightarrow 2$
- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \rightarrow 3$
- Let R be the finite-index congruence. Assume uRv.
- Then, $C[u]RC[v]$ for all contexts C
- As L is union of eq-classes of R, we have $C[u] \in L$ iff $C[v] \in L$
- Thus, $u \equiv_L v$

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \to 2$

- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \to 3$

- Let R be the finite-index congruence. Assume uRv.
- Then, $C[u]RC[v]$ for all contexts C
- As L is union of eq-classes of R, we have $C[u] \in L$ iff $C[v] \in L$
- Thus, $u \equiv_L v$
- I.e., $\equiv_L$ has not more eq-classes then the finite-index R

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \to 2$
- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \to 3$
- Let R be the finite-index congruence. Assume uRv.
- Then, $C[u]RC[v]$ for all contexts C
- As L is union of eq-classes of R, we have $C[u] \in L$ iff $C[v] \in L$
- Thus, $u \equiv_L v$
- I.e., $\equiv_L$ has not more eq-classes then the finite-index R

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \to 2$
- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \to 3$
- Let R be the finite-index congruence. Assume uRv.
- Then, $C[u]RC[v]$ for all contexts C
- As L is union of eq-classes of R, we have $C[u] \in L$ iff $C[v] \in L$
- Thus, $u \equiv_L v$
- I.e., $\equiv_L$ has not more eq-classes then the finite-index R

$3 \to 1$
- Let Q_{min} be the set of eq-classes of $\equiv_L$

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \to 2$

- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \to 3$

- Let R be the finite-index congruence. Assume uRv.
- Then, $C[u]RC[v]$ for all contexts C
- As L is union of eq-classes of R, we have $C[u] \in L$ iff $C[v] \in L$
- Thus, $u \equiv_L v$
- I.e., $\equiv_L$ has not more eq-classes then the finite-index R

$3 \to 1$

- Let Q_{min} be the set of eq-classes of $\equiv_L$
- Let $\Delta_{min} := \{f([u_1]_{\equiv_L}, \dots, [u_n]_{\equiv_L}) \to [f(u_1, \dots, u_n)]_{\equiv_L} \mid f \in \mathcal{F}_n, u_1, \dots, u_n \in T(\mathcal{F})\}$

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \to 2$

- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \to 3$

- Let R be the finite-index congruence. Assume uRv.
- Then, $C[u]RC[v]$ for all contexts C
- As L is union of eq-classes of R, we have $C[u] \in L$ iff $C[v] \in L$
- Thus, $u \equiv_L v$
- I.e., $\equiv_L$ has not more eq-classes then the finite-index R

$3 \to 1$

- Let Q_{min} be the set of eq-classes of $\equiv_L$
- Let $\Delta_{min} := \{f([u_1]_{\equiv_L}, \dots, [u_n]_{\equiv_L}) \to [f(u_1, \dots, u_n)]_{\equiv_L} \mid f \in \mathcal{F}_n, u_1, \dots, u_n \in T(\mathcal{F})\}$
- Note that Δ_{min} is deterministic, as $\equiv_L$ is a congruence

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \rightarrow 2$
- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \rightarrow 3$
- Let R be the finite-index congruence. Assume uRv.
- Then, $C[u]RC[v]$ for all contexts C
- As L is union of eq-classes of R, we have $C[u] \in L$ iff $C[v] \in L$
- Thus, $u \equiv_L v$
- I.e., $\equiv_L$ has not more eq-classes then the finite-index R

$3 \rightarrow 1$
- Let Q_{min} be the set of eq-classes of $\equiv_L$
- Let $\Delta_{min} := \{f([u_1]_{\equiv_L}, \dots, [u_n]_{\equiv_L}) \rightarrow [f(u_1, \dots, u_n)]_{\equiv_L} \mid f \in \mathcal{F}_n, u_1, \dots, u_n \in T(\mathcal{F})\}$
- Note that Δ_{min} is deterministic, as $\equiv_L$ is a congruence
- Let $Q_{min_f} := \{[u] \mid u \in L\}$

Proof of Myhill-Nerode Theorem

1. L is a regular tree language
2. L is the union of some equivalence classes of a finite-index congruence
3. $\equiv_L$ is of finite index

$1 \to 2$
- Take complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$ with $L = L(\mathcal{A})$.
- Let $u \equiv v$ iff $\delta(u) = \delta(v)$ (Obviously a congruence)
- $\equiv$ has finite index (at most $|Q|$ equivalence classes)
- We have $L = \bigcup\{[u] \mid \delta(u) \in Q_f\}$

$2 \to 3$
- Let R be the finite-index congruence. Assume uRv.
- Then, $C[u]RC[v]$ for all contexts C
- As L is union of eq-classes of R, we have $C[u] \in L$ iff $C[v] \in L$
- Thus, $u \equiv_L v$
- I.e., $\equiv_L$ has not more eq-classes then the finite-index R

$3 \to 1$
- Let Q_{min} be the set of eq-classes of $\equiv_L$
- Let $\Delta_{min} := \{f([u_1]_{\equiv_L}, \dots, [u_n]_{\equiv_L}) \to [f(u_1, \dots, u_n)]_{\equiv_L} \mid f \in \mathcal{F}_n, u_1, \dots, u_n \in T(\mathcal{F})\}$
- Note that Δ_{min} is deterministic, as $\equiv_L$ is a congruence
- Let $Q_{min_f} := \{[u] \mid u \in L\}$
- The DFTA $\mathcal{A}_{min} := (Q_{min}, \mathcal{F}, Q_{min_f}, \Delta_{min})$ recognizes the language L

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.
  - It is given by $\mathcal{A}_{min}$ from the proof of Myhill-Nerode

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.
  - It is given by $\mathcal{A}_{min}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.
  - It is given by $\mathcal{A}_{min}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):
  - Assume L is recognized by complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.
  - It is given by $\mathcal{A}_{min}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):
  - Assume L is recognized by complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
  - The relation $\equiv_{\mathcal{A}}$ is *refinement* of $\equiv_L$
    - $\equiv_{\mathcal{A}} \subseteq \equiv_L$

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.
  - It is given by $\mathcal{A}_{min}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):
  - Assume L is recognized by complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
  - The relation $\equiv_{\mathcal{A}}$ is *refinement* of $\equiv_L$
    - $\equiv_{\mathcal{A}} \subseteq \equiv_L$
  - Thus $|Q| \geq |Q_{min}|$ (proves existence of minimal DFTA)

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.
  - It is given by $\mathcal{A}_{min}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):
  - Assume L is recognized by complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
  - The relation $\equiv_{\mathcal{A}}$ is *refinement* of $\equiv_L$
    - $\equiv_{\mathcal{A}} \subseteq \equiv_L$
  - Thus $|Q| \geq |Q_{min}|$ (proves existence of minimal DFTA)
  - Now assume $|Q| = |Q_{min}|$

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.
  - It is given by $\mathcal{A}_{min}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):
  - Assume L is recognized by complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
  - The relation $\equiv_{\mathcal{A}}$ is *refinement* of $\equiv_L$
    - $\equiv_{\mathcal{A}} \subseteq \equiv_L$
  - Thus $|Q| \geq |Q_{min}|$ (proves existence of minimal DFTA)
  - Now assume $|Q| = |Q_{min}|$
    - All states in Q are accessible (otherwise, contradiction to minimality)

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.
  - It is given by $\mathcal{A}_{min}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):
  - Assume L is recognized by complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
  - The relation $\equiv_{\mathcal{A}}$ is *refinement* of $\equiv_L$
    - $\equiv_{\mathcal{A}} \subseteq \equiv_L$
  - Thus $|Q| \geq |Q_{min}|$ (proves existence of minimal DFTA)
  - Now assume $|Q| = |Q_{min}|$
    - All states in Q are accessible (otherwise, contradiction to minimality)
    - Let $q \in Q$ with $\delta(u) = q$.

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.
  - It is given by $\mathcal{A}_{min}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):
  - Assume L is recognized by complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
  - The relation $\equiv_{\mathcal{A}}$ is *refinement* of $\equiv_L$
    - $\equiv_{\mathcal{A}} \subseteq \equiv_L$
  - Thus $|Q| \geq |Q_{min}|$ (proves existence of minimal DFTA)
  - Now assume $|Q| = |Q_{min}|$
    - All states in Q are accessible (otherwise, contradiction to minimality)
    - Let $q \in Q$ with $\delta(u) = q$.
    - Identify q and $\delta_{min}(u)$

Unique minimal DFTA

- Corollary: The minimal complete DFTA accepting a regular language exists and is unique.
  - It is given by $\mathcal{A}_{min}$ from the proof of Myhill-Nerode
- Proof sketch (more details on board):
  - Assume L is recognized by complete DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
  - The relation $\equiv_{\mathcal{A}}$ is *refinement* of $\equiv_L$
    - $\equiv_{\mathcal{A}} \subseteq \equiv_L$
  - Thus $|Q| \geq |Q_{min}|$ (proves existence of minimal DFTA)
  - Now assume $|Q| = |Q_{min}|$
    - All states in Q are accessible (otherwise, contradiction to minimality)
    - Let $q \in Q$ with $\delta(u) = q$.
    - Identify q and $\delta_{min}(u)$
    - This mapping is consistent and bijection

Minimization algorithm

- Given complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$

Minimization algorithm

- Given complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
- Idea: Refine an equivalence relation until consistent with $\mathcal{A}$

Minimization algorithm

- Given complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
- Idea: Refine an equivalence relation until consistent with $\mathcal{A}$

1. Start with $P = \{Q_f, Q \setminus Q_f\}$

Minimization algorithm

- Given complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
- Idea: Refine an equivalence relation until consistent with $\mathcal{A}$

1. Start with $P = \{Q_f, Q \setminus Q_f\}$
2. Refine P. Let P' be the new value. Set $qP'q'$, if
   - qPq'
   - $q \equiv q'$ is consistent wrt. the rules, i.e.

$$\forall f \in \mathcal{F}_n, q_1, \ldots, q_{i-1}, q_{i+1}, \ldots q_n.$$
$$\delta(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) P \delta(f, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_n)$$

Minimization algorithm

- Given complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
- Idea: Refine an equivalence relation until consistent with $\mathcal{A}$

1. Start with $P = \{Q_f, Q \setminus Q_f\}$
2. Refine P. Let P' be the new value. Set $qP'q'$, if
   - qPq'
   - $q \equiv q'$ is consistent wrt. the rules, i.e.

$$\forall f \in \mathcal{F}_n, q_1, \dots, q_{i-1}, q_{i+1}, \dots q_n.$$
$$\delta(f, q_1, \dots, q_{i-1}, q, q_{i+1}, \dots, q_n) P \delta(f, q_1, \dots, q_{i-1}, q', q_{i+1}, \dots, q_n)$$

3. Repeat until no more refinement possible

Minimization algorithm

- Given complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
- Idea: Refine an equivalence relation until consistent with $\mathcal{A}$

1. Start with $P = \{Q_f, Q \setminus Q_f\}$
2. Refine P. Let P' be the new value. Set $qP'q'$, if
   - qPq'
   - $q \equiv q'$ is consistent wrt. the rules, i.e.

$$\begin{aligned}&\forall f \in \mathcal{F}_n, q_1, \ldots, q_{i-1}, q_{i+1}, \ldots q_n. \\ &\qquad \delta(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) P \delta(f, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_n)\end{aligned}$$

3. Repeat until no more refinement possible
4. Define $\mathcal{A}_{min} := (Q_{min}, \mathcal{F}, Q_{minf}, \delta)$, where

Minimization algorithm

- Given complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
- Idea: Refine an equivalence relation until consistent with $\mathcal{A}$

1. Start with $P = \{Q_f, Q \setminus Q_f\}$
2. Refine P. Let P' be the new value. Set $qP'q'$, if
   - qPq'
   - $q \equiv q'$ is consistent wrt. the rules, i.e.

$$\forall f \in \mathcal{F}_n, q_1, \ldots, q_{i-1}, q_{i+1}, \ldots q_n.$$
$$\delta(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) P \delta(f, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_n)$$

3. Repeat until no more refinement possible
4. Define $\mathcal{A}_{min} := (Q_{min}, \mathcal{F}, Q_{minf}, \delta)$, where
   - $Q_{min} :=$ Equivalence classes of P

Minimization algorithm

- Given complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
- Idea: Refine an equivalence relation until consistent with $\mathcal{A}$

1. Start with $P = \{Q_f, Q \setminus Q_f\}$
2. Refine P. Let P' be the new value. Set $qP'q'$, if
   - qPq'
   - $q \equiv q'$ is consistent wrt. the rules, i.e.

$$\begin{aligned}&\forall f \in \mathcal{F}_n, q_1, \dots, q_{i-1}, q_{i+1}, \dots q_n.\\ &\qquad \delta(f, q_1, \dots, q_{i-1}, q, q_{i+1}, \dots, q_n) P \delta(f, q_1, \dots, q_{i-1}, q', q_{i+1}, \dots, q_n)\end{aligned}$$

3. Repeat until no more refinement possible
4. Define $\mathcal{A}_{min} := (Q_{min}, \mathcal{F}, Q_{minf}, \delta)$, where
   - $Q_{min} :=$ Equivalence classes of P
   - $Q_{minf} := \{[q] \mid q \in Q_f\}$

Minimization algorithm

- Given complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
- Idea: Refine an equivalence relation until consistent with $\mathcal{A}$

1. Start with $P = \{Q_f, Q \setminus Q_f\}$
2. Refine P. Let P' be the new value. Set $qP'q'$, if
   - qPq'
   - $q \equiv q'$ is consistent wrt. the rules, i.e.

$$\begin{aligned}&\forall f \in \mathcal{F}_n, q_1, \dots, q_{i-1}, q_{i+1}, \dots q_n.\\ &\qquad \delta(f, q_1, \dots, q_{i-1}, q, q_{i+1}, \dots, q_n) P \delta(f, q_1, \dots, q_{i-1}, q', q_{i+1}, \dots, q_n)\end{aligned}$$

3. Repeat until no more refinement possible
4. Define $\mathcal{A}_{min} := (Q_{min}, \mathcal{F}, Q_{minf}, \delta)$, where
   - $Q_{min} :=$ Equivalence classes of P
   - $Q_{minf} := \{[q] \mid q \in Q_f\}$
   - $\delta_{min}(f, [q_1], \dots, [q_n]) = [\delta(f, q_1, \dots, q_n)]$

Minimization algorithm

- Given complete and reduced DFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \delta)$
- Idea: Refine an equivalence relation until consistent with $\mathcal{A}$

1. Start with $P = \{Q_f, Q \setminus Q_f\}$
2. Refine P. Let P' be the new value. Set $qP'q'$, if
   - qPq'
   - $q \equiv q'$ is consistent wrt. the rules, i.e.

$$\forall f \in \mathcal{F}_n, q_1, \ldots, q_{i-1}, q_{i+1}, \ldots q_n. \\ \delta(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) P \delta(f, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q_n)$$

3. Repeat until no more refinement possible
4. Define $\mathcal{A}_{min} := (Q_{min}, \mathcal{F}, Q_{minf}, \delta)$, where
   - $Q_{min} :=$ Equivalence classes of P
   - $Q_{minf} := \{[q] \mid q \in Q_f\}$
   - $\delta_{min}(f, [q_1], \ldots, [q_n]) = [\delta(f, q_1, \ldots, q_n)]$

- $L(\mathcal{A}_{min}) = L(\mathcal{A})$. Proof on board.

Last Lecture

- Myhill-Nerode Theorem
- Minimization of tree automata

Table of Contents

Top-Down Tree Automata

- Recall: Tree automata rewrite tree to single state

Top-Down Tree Automata

- Recall: Tree automata rewrite tree to single state
  - Starting at the leaves, i.e. bottom-up

Top-Down Tree Automata

- Recall: Tree automata rewrite tree to single state
  - Starting at the leaves, i.e. bottom-up
  - $f(q_1, \ldots, q_n) \rightarrow q$

Top-Down Tree Automata

- Recall: Tree automata rewrite tree to single state
  - Starting at the leaves, i.e. bottom-up
  - $f(q_1, \ldots, q_n) \to q$
  - Intuition: Assign state to a given tree, consume tree

Top-Down Tree Automata

- Recall: Tree automata rewrite tree to single state
  - Starting at the leaves, i.e. bottom-up
  - $f(q_1, \ldots, q_n) \rightarrow q$
  - Intuition: Assign state to a given tree, consume tree
- Now: Rewrite state to a tree

Top-Down Tree Automata

- Recall: Tree automata rewrite tree to single state
  - Starting at the leaves, i.e. bottom-up
  - $f(q_1, \ldots, q_n) \rightarrow q$
  - Intuition: Assign state to a given tree, consume tree
- Now: Rewrite state to a tree
  - Starting at a single root state

Top-Down Tree Automata

- Recall: Tree automata rewrite tree to single state
  - Starting at the leaves, i.e. bottom-up
  - $f(q_1, \ldots, q_n) \to q$
  - Intuition: Assign state to a given tree, consume tree
- Now: Rewrite state to a tree
  - Starting at a single root state
  - $q \to f(q_1, \ldots, q_n)$

Top-Down Tree Automata

- Recall: Tree automata rewrite tree to single state
  - Starting at the leaves, i.e. bottom-up
  - $f(q_1, \ldots, q_n) \to q$
  - Intuition: Assign state to a given tree, consume tree
- Now: Rewrite state to a tree
  - Starting at a single root state
  - $q \to f(q_1, \ldots, q_n)$
  - Intuition: Assign tree to given state, produce tree.

Top-Down Tree Automata

- A tuple $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is called *top-down* tree automaton, where

Top-Down Tree Automata

- A tuple $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is called *top-down* tree automaton, where
    - $\mathcal{F}$ is a ranked alphabet

Top-Down Tree Automata

- A tuple $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is called *top-down* tree automaton, where
  - $\mathcal{F}$ is a ranked alphabet
  - Q is a finite set of states, with $Q \cap \mathcal{F} = \emptyset$

Top-Down Tree Automata

- A tuple $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is called *top-down* tree automaton, where
  - $\mathcal{F}$ is a ranked alphabet
  - Q is a finite set of states, with $Q \cap \mathcal{F} = \emptyset$
  - $I \subseteq Q$ is a set of initial states

Top-Down Tree Automata

- A tuple $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is called *top-down* tree automaton, where
  - $\mathcal{F}$ is a ranked alphabet
  - Q is a finite set of states, with $Q \cap \mathcal{F} = \emptyset$
  - $I \subseteq Q$ is a set of initial states
  - Δ is a set of rules of the form

$$q \to f(q_1, \dots, q_n) \text{ for } f \in \mathcal{F}_n, q, q_1, \dots, q_n \in Q$$

Top-Down Tree Automata

- A tuple $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is called *top-down* tree automaton, where
  - $\mathcal{F}$ is a ranked alphabet
  - Q is a finite set of states, with $Q \cap \mathcal{F} = \emptyset$
  - $I \subseteq Q$ is a set of initial states
  - Δ is a set of rules of the form

$$q \to f(q_1, \ldots, q_n) \text{ for } f \in \mathcal{F}_n, q, q_1, \ldots, q_n \in Q$$

- We define the *production relation* $q \to_{\mathcal{A}} t$ as the least relation that satisfies

$$q \to f(q_1, \ldots, q_n) \in \Delta, q_1 \to_{\mathcal{A}} t_1, \ldots, q_n \to_{\mathcal{A}} t_n \implies q \to_{\mathcal{A}} f(t_1, \ldots, t_n)$$

Top-Down Tree Automata

- A tuple $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is called *top-down* tree automaton, where
  - $\mathcal{F}$ is a ranked alphabet
  - Q is a finite set of states, with $Q \cap \mathcal{F} = \emptyset$
  - $I \subseteq Q$ is a set of initial states
  - Δ is a set of rules of the form

$$q \to f(q_1, \ldots, q_n) \text{ for } f \in \mathcal{F}_n, q, q_1, \ldots, q_n \in Q$$

- We define the *production relation* $q \to_{\mathcal{A}} t$ as the least relation that satisfies

$$q \to f(q_1, \ldots, q_n) \in \Delta, q_1 \to_{\mathcal{A}} t_1, \ldots, q_n \to_{\mathcal{A}} t_n \implies q \to_{\mathcal{A}} f(t_1, \ldots, t_n)$$

- The language of $\mathcal{A}$ is $L(\mathcal{A}) := \{t \mid \exists q \in I.\ q \to_{\mathcal{A}} t\}$

Equal expressiveness

Theorem

A language is regular if and only if it is the language of a top-down tree automaton.

- Proof

Equal expressiveness

Theorem

A language is regular if and only if it is the language of a top-down tree automaton.

- Proof
  - Straightforward induction (Hint: Reverse arrows, exchange I and Q_f)

Equal expressiveness

Theorem

A language is regular if and only if it is the language of a top-down tree automaton.

- Proof
  - Straightforward induction (Hint: Reverse arrows, exchange I and Q_f)
  - Exercise

Deterministic Top-Down Tree Automata

- A top-down tree-automaton $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is *deterministic*, iff

Deterministic Top-Down Tree Automata

- A top-down tree-automaton $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is *deterministic*, iff
  - $|I| = 1$

Deterministic Top-Down Tree Automata

- A top-down tree-automaton $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is *deterministic*, iff
  - $|I| = 1$
  - $q \to f(q_1, \ldots, q_n) \in \Delta \wedge q \to f(q'_1, \ldots, q'_n) \in \Delta \implies q_1 = q'_1 \wedge \ldots \wedge q_n = q'_n$

Deterministic Top-Down Tree Automata

- A top-down tree-automaton $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is *deterministic*, iff
  - $|I| = 1$
  - $q \to f(q_1, \ldots, q_n) \in \Delta \wedge q \to f(q'_1, \ldots, q'_n) \in \Delta \implies q_1 = q'_1 \wedge \ldots \wedge q_n = q'_n$
- Unfortunately: There are regular languages not accepted by any deterministic top-down FTA

Deterministic Top-Down Tree Automata

- A top-down tree-automaton $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is *deterministic*, iff
  - $|I| = 1$
  - $q \to f(q_1, \ldots, q_n) \in \Delta \wedge q \to f(q'_1, \ldots, q'_n) \in \Delta \implies q_1 = q'_1 \wedge \ldots \wedge q_n = q'_n$
- Unfortunately: There are regular languages not accepted by any deterministic top-down FTA
  - $L = \{f(a, b), f(b, a)\}$. Obviously regular. Even finite.

Deterministic Top-Down Tree Automata

- A top-down tree-automaton $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$ is *deterministic*, iff
  - $|I| = 1$
  - $q \to f(q_1, \ldots, q_n) \in \Delta \wedge q \to f(q'_1, \ldots, q'_n) \in \Delta \implies q_1 = q'_1 \wedge \ldots \wedge q_n = q'_n$
- Unfortunately: There are regular languages not accepted by any deterministic top-down FTA
  - $L = \{f(a, b), f(b, a)\}$. Obviously regular. Even finite.
  - But: Any deterministic top-down FTA that accepts the words in L also accepts $f(a, a)$.

Table of Contents

Table of Contents

Regular Tree Grammars

- Extend grammars to trees

Regular Tree Grammars

- Extend grammars to trees
- Here: Only for the regular case

Regular Tree Grammars

- Extend grammars to trees
- Here: Only for the regular case
- A *regular tree grammar* (RTG) is a tuple $G = (S, N, \mathcal{F}, R)$, where

Regular Tree Grammars

- Extend grammars to trees
- Here: Only for the regular case
- A *regular tree grammar* (RTG) is a tuple $G = (S, N, \mathcal{F}, R)$, where
  - $S \in N$ is a start symbol

Regular Tree Grammars

- Extend grammars to trees
- Here: Only for the regular case
- A *regular tree grammar* (RTG) is a tuple $G = (S, N, \mathcal{F}, R)$, where
  - $S \in N$ is a start symbol
  - N is a finite set of nonterminals with arity zero, and $N \cap \mathcal{F} = \emptyset$

Regular Tree Grammars

- Extend grammars to trees
- Here: Only for the regular case
- A *regular tree grammar* (RTG) is a tuple $G = (S, N, \mathcal{F}, R)$, where
  - $S \in N$ is a start symbol
  - N is a finite set of nonterminals with arity zero, and $N \cap \mathcal{F} = \emptyset$
  - $\mathcal{F}$ is a ranked alphabet

Regular Tree Grammars

- Extend grammars to trees
- Here: Only for the regular case
- A *regular tree grammar* (RTG) is a tuple $G = (S, N, \mathcal{F}, R)$, where
  - $S \in N$ is a start symbol
  - N is a finite set of nonterminals with arity zero, and $N \cap \mathcal{F} = \emptyset$
  - $\mathcal{F}$ is a ranked alphabet
  - R is a set of production rules of the form $n \rightarrow \beta$, where $n \in N$ and $\beta \in T(\mathcal{F} \cup N)$

Regular Tree Grammars

- Extend grammars to trees
- Here: Only for the regular case
- A *regular tree grammar* (RTG) is a tuple $G = (S, N, \mathcal{F}, R)$, where
  - $S \in N$ is a start symbol
  - N is a finite set of nonterminals with arity zero, and $N \cap \mathcal{F} = \emptyset$
  - $\mathcal{F}$ is a ranked alphabet
  - R is a set of production rules of the form $n \to \beta$, where $n \in N$ and $\beta \in T(\mathcal{F} \cup N)$
- These are almost top-down tree automata

Regular Tree Grammars

- Extend grammars to trees
- Here: Only for the regular case
- A *regular tree grammar* (RTG) is a tuple $G = (S, N, \mathcal{F}, R)$, where
  - $S \in N$ is a start symbol
  - N is a finite set of nonterminals with arity zero, and $N \cap \mathcal{F} = \emptyset$
  - $\mathcal{F}$ is a ranked alphabet
  - R is a set of production rules of the form $n \to \beta$, where $n \in N$ and $\beta \in T(\mathcal{F} \cup N)$
- These are almost top-down tree automata
  - But rules are a bit more complicated

Derivation Relation

- Intuition: Rewrite S to a tree, using the rules

Derivation Relation

- Intuition: Rewrite S to a tree, using the rules
- For an RTG $G = (S, N, \mathcal{F}, R)$, we define a derivation step $\beta \Rightarrow_G \beta'$ for $\beta, \beta' \in T(\mathcal{F} \cup N)$ by

$$\beta \Rightarrow_G \beta' \iff \exists C\ u\ n.\ \beta = C[n] \land n \to u \in R \land \beta' = C[u]$$

Derivation Relation

- Intuition: Rewrite S to a tree, using the rules
- For an RTG $G = (S, N, \mathcal{F}, R)$, we define a derivation step $\beta \Rightarrow_G \beta'$ for $\beta, \beta' \in T(\mathcal{F} \cup N)$ by

$$\beta \Rightarrow_G \beta' \iff \exists C\ u\ n.\ \beta = C[n] \land n \to u \in R \land \beta' = C[u]$$

- We write $\beta \to_G t'$, iff $t' \in T(\mathcal{F})$ and $\beta \Rightarrow_G^* t'$

Derivation Relation

- Intuition: Rewrite S to a tree, using the rules
- For an RTG $G = (S, N, \mathcal{F}, R)$, we define a derivation step $\beta \Rightarrow_G \beta'$ for $\beta, \beta' \in T(\mathcal{F} \cup N)$ by

$$\beta \Rightarrow_G \beta' \iff \exists C\ u\ n.\ \beta = C[n] \land n \to u \in R \land \beta' = C[u]$$

- We write $\beta \to_G t'$, iff $t' \in T(\mathcal{F})$ and $\beta \Rightarrow_G^* t'$
- For $n \in N$, we define $L(G, n) := \{t \in T(\mathcal{F}) \mid n \to_G t\}$

Derivation Relation

- Intuition: Rewrite S to a tree, using the rules
- For an RTG $G = (S, N, \mathcal{F}, R)$, we define a derivation step $\beta \Rightarrow_G \beta'$ for $\beta, \beta' \in T(\mathcal{F} \cup N)$ by

$$\beta \Rightarrow_G \beta' \iff \exists C\ u\ n.\ \beta = C[n] \wedge n \rightarrow u \in R \wedge \beta' = C[u]$$

- We write $\beta \rightarrow_G t'$, iff $t' \in T(\mathcal{F})$ and $\beta \Rightarrow_G^* t'$
- For $n \in N$, we define $L(G, n) := \{t \in T(\mathcal{F}) \mid n \rightarrow_G t\}$
- We define $L(G) := L(G, S)$

Reduced tree grammars

- A non-terminal n is *reachable*, iff there is a derivation from S to a tree containing n:

  $$\exists C.\ S \Rightarrow_G^* C[n]$$

Reduced tree grammars

- A non-terminal n is *reachable*, iff there is a derivation from S to a tree containing n:

  $$\exists C.\ S \Rightarrow_G^* C[n]$$

- A non-terminal n is *productive*, iff a tree without nonterminals can be derived from it:

  $$L(G, n) \neq \emptyset$$

Reduced tree grammars

- A non-terminal n is *reachable*, iff there is a derivation from S to a tree containing n:

$$\exists C.\ S \Rightarrow_G^* C[n]$$

- A non-terminal n is *productive*, iff a tree without nonterminals can be derived from it:

$$L(G, n) \neq \emptyset$$

- An RTG is *reduced*, if every nonterminal is reachable and productive

Computation of Equivalent Reduced Grammar

- For every RTG G, reduced tree grammar G' with $L(G) = L(G')$ can be computed

Computation of Equivalent Reduced Grammar

- For every RTG G, reduced tree grammar G' with $L(G) = L(G')$ can be computed
  - Provided that $L(G) \neq \emptyset$, otherwise S must not be productive.

Computation of Equivalent Reduced Grammar

- For every RTG G, reduced tree grammar G' with $L(G) = L(G')$ can be computed
  - Provided that $L(G) \neq \emptyset$, otherwise S must not be productive.

1. Remove unproductive non-terminals

Computation of Equivalent Reduced Grammar

- For every RTG G, reduced tree grammar G' with $L(G) = L(G')$ can be computed
  - Provided that $L(G) \neq \emptyset$, otherwise S must not be productive.

1. Remove unproductive non-terminals
   - Productive nonterminals can be computed by saturation algorithm:

Computation of Equivalent Reduced Grammar

- For every RTG G, reduced tree grammar G' with $L(G) = L(G')$ can be computed
  - Provided that $L(G) \neq \emptyset$, otherwise S must not be productive.

1. Remove unproductive non-terminals
   - Productive nonterminals can be computed by saturation algorithm:
   - n is productive, if there is a rule $n \to \beta$ such that every nonterminal in β is productive

Computation of Equivalent Reduced Grammar

- For every RTG G, reduced tree grammar G' with $L(G) = L(G')$ can be computed
  - Provided that $L(G) \neq \emptyset$, otherwise S must not be productive.

1. Remove unproductive non-terminals
   - Productive nonterminals can be computed by saturation algorithm:
   - n is productive, if there is a rule $n \to \beta$ such that every nonterminal in β is productive
2. Remove unreachable nonterminals

Computation of Equivalent Reduced Grammar

- For every RTG G, reduced tree grammar G' with $L(G) = L(G')$ can be computed
  - Provided that $L(G) \neq \emptyset$, otherwise S must not be productive.

1. Remove unproductive non-terminals
   - Productive nonterminals can be computed by saturation algorithm:
   - n is productive, if there is a rule $n \to \beta$ such that every nonterminal in β is productive
2. Remove unreachable nonterminals
   - Again saturation: S is reachable, n is reachable if there is a rule $\hat{n} \to C[n]$ such that $\hat{n}$ is reachable

Correctness

- Obviously, removing unproductive or unreachable nonterminals does not change the language

Correctness

- Obviously, removing unproductive or unreachable nonterminals does not change the language
- Remains to show: Removing unreachable nonterminals cannot create new unproductive ones

Correctness

- Obviously, removing unproductive or unreachable nonterminals does not change the language
- Remains to show: Removing unreachable nonterminals cannot create new unproductive ones
  - On board

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \rightarrow f(n_1, \ldots, n_n)$ for $n, n_1, \ldots, n_n \in N$

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \to f(n_1, \ldots, n_n)$ for $n, n_1, \ldots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \to f(n_1, \ldots, n_n)$ for $n, n_1, \ldots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \to f(s_1, \ldots, s_n)$ by $n \to f(n_1, \ldots, n_n)$

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \to f(n_1, \dots, n_n)$ for $n, n_1, \dots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \to f(s_1, \dots, s_n)$ by $n \to f(n_1, \dots, n_n)$
    - where $n_i = s_i$ if $s_i \in N$

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \to f(n_1, \ldots, n_n)$ for $n, n_1, \ldots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \to f(s_1, \ldots, s_n)$ by $n \to f(n_1, \ldots, n_n)$
    - where $n_i = s_i$ if $s_i \in N$
    - $n_i \in N$ fresh otherwise. In this case, add rule $n_i \to s_i$

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \to f(n_1, \dots, n_n)$ for $n, n_1, \dots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \to f(s_1, \dots, s_n)$ by $n \to f(n_1, \dots, n_n)$
    - where $n_i = s_i$ if $s_i \in N$
    - $n_i \in N$ fresh otherwise. In this case, add rule $n_i \to s_i$
  - After iteration, all rules have form $n \to f(n_1, \dots, n_n)$ or $n_1 \to n_2$

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \to f(n_1, \ldots, n_n)$ for $n, n_1, \ldots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \to f(s_1, \ldots, s_n)$ by $n \to f(n_1, \ldots, n_n)$
    - where $n_i = s_i$ if $s_i \in N$
    - $n_i \in N$ fresh otherwise. In this case, add rule $n_i \to s_i$
  - After iteration, all rules have form $n \to f(n_1, \ldots, n_n)$ or $n_1 \to n_2$
  - Eliminate the latter rules by replacing $s_1 \to s_2$ by rules $s_1 \to t$ for all $t \notin N$ with $s_2 \to^* n \to t$

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \to f(n_1, \dots, n_n)$ for $n, n_1, \dots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \to f(s_1, \dots, s_n)$ by $n \to f(n_1, \dots, n_n)$
    - where $n_i = s_i$ if $s_i \in N$
    - $n_i \in N$ fresh otherwise. In this case, add rule $n_i \to s_i$
  - After iteration, all rules have form $n \to f(n_1, \dots, n_n)$ or $n_1 \to n_2$
  - Eliminate the latter rules by replacing $s_1 \to s_2$ by rules $s_1 \to t$ for all $t \notin N$ with $s_2 \to^* n \to t$
    - Cf.: Elimination of epsilon rules

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \to f(n_1, \ldots, n_n)$ for $n, n_1, \ldots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \to f(s_1, \ldots, s_n)$ by $n \to f(n_1, \ldots, n_n)$
    - where $n_i = s_i$ if $s_i \in N$
    - $n_i \in N$ fresh otherwise. In this case, add rule $n_i \to s_i$
  - After iteration, all rules have form $n \to f(n_1, \ldots, n_n)$ or $n_1 \to n_2$
  - Eliminate the latter rules by replacing $s_1 \to s_2$ by rules $s_1 \to t$ for all $t \notin N$ with $s_2 \to^* n \to t$
    - Cf.: Elimination of epsilon rules
- Correctness (Ideas)

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \to f(n_1, \dots, n_n)$ for $n, n_1, \dots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \to f(s_1, \dots, s_n)$ by $n \to f(n_1, \dots, n_n)$
    - where $n_i = s_i$ if $s_i \in N$
    - $n_i \in N$ fresh otherwise. In this case, add rule $n_i \to s_i$
  - After iteration, all rules have form $n \to f(n_1, \dots, n_n)$ or $n_1 \to n_2$
  - Eliminate the latter rules by replacing $s_1 \to s_2$ by rules $s_1 \to t$ for all $t \notin N$ with $s_2 \to^* n \to t$
    - Cf.: Elimination of epsilon rules
- Correctness (Ideas)
  - Each step of the iteration preserves language

Normalized Regular Tree Grammars

- RTG is normalized, iff all productions have the form $n \to f(n_1, \ldots, n_n)$ for $n, n_1, \ldots, n_n \in N$
- Every RTG can be transformed into an equivalent normal one
  - Iterate: Replace a rule $n \to f(s_1, \ldots, s_n)$ by $n \to f(n_1, \ldots, n_n)$
    - where $n_i = s_i$ if $s_i \in N$
    - $n_i \in N$ fresh otherwise. In this case, add rule $n_i \to s_i$
  - After iteration, all rules have form $n \to f(n_1, \ldots, n_n)$ or $n_1 \to n_2$
  - Eliminate the latter rules by replacing $s_1 \to s_2$ by rules $s_1 \to t$ for all $t \notin N$ with $s_2 \to^* n \to t$
    - Cf.: Elimination of epsilon rules
- Correctness (Ideas)
  - Each step of the iteration preserves language
  - Elimination preserves language

Normalized RTGs and top-down NTFAs

- Obviously, normalized RTGs are isomorphic to top-down NTFAs

Normalized RTGs and top-down NTFAs

- Obviously, normalized RTGs are isomorphic to top-down NTFAs
- Thus, exactly the regular languages can be expressed by RTGs

Theorem

A language is regular if and only if it can be described by a regular tree grammar.

Last Lecture

- Myhill Nerode Theorem
- Minimization Algorithm
- Top-Down Tree Automata
- Regular Tree Grammars
- Started: Tree Regular Expressions

Table of Contents

Recall: Word regular expressions

- $e ::= \varepsilon \mid \emptyset \mid a \text{ for } a \in \Sigma \mid e \cdot e \mid e + e \mid e^*$

Recall: Word regular expressions

- $e ::= \varepsilon \mid \emptyset \mid a$ for $a \in \Sigma \mid e \cdot e \mid e + e \mid e^*$
  - Empty word | empty language | single character | concatenation | choice | iteration

Recall: Word regular expressions

- $e ::= \varepsilon \mid \emptyset \mid a$ for $a \in \Sigma \mid e \cdot e \mid e + e \mid e^*$
  - Empty word | empty language | single character | concatenation | choice | iteration
- For example: $(r + w + o)^* \cdot (r + w) \cdot (r + w + o)^*$

Recall: Word regular expressions

- $e ::= \varepsilon \mid \emptyset \mid a$ for $a \in \Sigma \mid e \cdot e \mid e + e \mid e^*$
  - Empty word | empty language | single character | concatenation | choice | iteration
- For example: $(r + w + o)^* \cdot (r + w) \cdot (r + w + o)^*$
  - Words containing at least one r or at least one w

Recall: Word regular expressions

- $e ::= \varepsilon \mid \emptyset \mid a$ for $a \in \Sigma \mid e \cdot e \mid e + e \mid e^*$
    - Empty word | empty language | single character | concatenation | choice | iteration
- For example: $(r + w + o)^* \cdot (r + w) \cdot (r + w + o)^*$
    - Words containing at least one r or at least one w
- Recall: $e^* = \varepsilon + e \cdot e^*$

Tree regular expressions

- Consider the set $\{0, s(0), s(s(0)), \ldots\}$

Tree regular expressions

- Consider the set $\{0, s(0), s(s(0)), \ldots\}$
  - Want to represent this as „regular expression"

Tree regular expressions

- Consider the set $\{0, s(0), s(s(0)), \ldots\}$
  - Want to represent this as „regular expression"
- $s(\Box)^* \cdot 0$

Tree regular expressions

- Consider the set $\{0, s(0), s(s(0)), \ldots\}$
  - Want to represent this as „regular expression”
- $s(\Box)^* \cdot 0$
  - Idea: $\Box$ indicates position for concatenation

Tree regular expressions

- Consider the set $\{0, s(0), s(s(0)), \ldots\}$
  - Want to represent this as „regular expression"
- $s(\square)^* \cdot 0$
  - Idea: $\square$ indicates position for concatenation
  - $t_1 \cdot t_2$ inserts t_2 at square-position in t_1

Tree regular expressions

- Consider the set $\{0, s(0), s(s(0)), \ldots\}$
  - Want to represent this as „regular expression"
- $s(\Box)^* \cdot 0$
  - Idea: $\Box$ indicates position for concatenation
  - $t_1 \cdot t_2$ inserts t_2 at square-position in t_1
  - $f(\ldots)^* = \Box + f(\ldots) \cdot f(\ldots)^*$ iterates over position $\Box$

Tree regular expressions

- Consider the set $\{0, s(0), s(s(0)), \ldots\}$
  - Want to represent this as „regular expression"
- $s(\Box)^* \cdot 0$
  - Idea: $\Box$ indicates position for concatenation
  - $t_1 \cdot t_2$ inserts t_2 at square-position in t_1
  - $f(\ldots)^* = \Box + f(\ldots) \cdot f(\ldots)^*$ iterates over position $\Box$
- There may be more than one iteration, over different positions

Tree regular expressions

- Consider the set $\{0, s(0), s(s(0)), \ldots\}$
  - Want to represent this as „regular expression"
- $s(\Box)^* \cdot 0$
  - Idea: $\Box$ indicates position for concatenation
  - $t_1 \cdot t_2$ inserts t_2 at square-position in t_1
  - $f(\ldots)^* = \Box + f(\ldots) \cdot f(\ldots)^*$ iterates over position $\Box$
- There may be more than one iteration, over different positions
  - Number position markers: $\Box_1, \Box_2, \ldots$

Tree regular expressions

- Consider the set $\{0, s(0), s(s(0)), \dots\}$
  - Want to represent this as „regular expression"
- $s(\Box)^* \cdot 0$
  - Idea: $\Box$ indicates position for concatenation
  - $t_1 \cdot t_2$ inserts t_2 at square-position in t_1
  - $f(\dots)^* = \Box + f(\dots) \cdot f(\dots)^*$ iterates over position $\Box$
- There may be more than one iteration, over different positions
  - Number position markers: $\Box_1, \Box_2, \dots$
  - $cons(s(\Box_1)^{*_1} \cdot_1 0, \Box_2)^{*_2} \cdot_2 nil$

Tree regular expressions

- Consider the set $\{0, s(0), s(s(0)), \ldots\}$
  - Want to represent this as „regular expression"
- $s(\square)^* \cdot 0$
  - Idea: $\square$ indicates position for concatenation
  - $t_1 \cdot t_2$ inserts t_2 at square-position in t_1
  - $f(\ldots)^* = \square + f(\ldots) \cdot f(\ldots)^*$ iterates over position $\square$
- There may be more than one iteration, over different positions
  - Number position markers: $\square_1, \square_2, \ldots$
  - $cons(s(\square_1)^{*_1} \cdot_1 0, \square_2)^{*_2} \cdot_2 nil$
- Note: TATA notation: $s(\square_1)^{*,\square_1} \cdot_{\square_1} nil$

Substitution and Concatenation

- Let $\mathcal{K} := \Box_1/0, \Box_2/0, \ldots$. Assume $\mathcal{K} \cap \mathcal{F} = \emptyset$

Substitution and Concatenation

- Let $\mathcal{K} := \Box_1/0, \Box_2/0, \ldots$. Assume $\mathcal{K} \cap \mathcal{F} = \emptyset$
- For trees $t \in T(\mathcal{F} \cup \mathcal{K})$, we define (simultaneous) substitution $t\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$, for $a_i \in \mathcal{K}$ and $i \neq j \implies a_i \neq a_j$:

$$
\begin{aligned}
&a\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} = a \text{ for } a \in \mathcal{F} \cup \mathcal{K} \text{ and } \forall i.\ a \neq a_i \\
&a_i\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} = L_i \\
&f(s_1, \ldots, s_m)\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} \\
&\qquad\qquad = \{f(t_1, \ldots, t_m) \mid t_i \in s_i\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}\}
\end{aligned}
$$

Substitution and Concatenation

- Let $\mathcal{K} := \square_1/0, \square_2/0, \ldots$. Assume $\mathcal{K} \cap \mathcal{F} = \emptyset$
- For trees $t \in T(\mathcal{F} \cup \mathcal{K})$, we define (simultaneous) substitution $t\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$, for $a_i \in \mathcal{K}$ and $i \neq j \implies a_i \neq a_j$:

$$a\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} = a \text{ for } a \in \mathcal{F} \cup \mathcal{K} \text{ and } \forall i.\ a \neq a_i$$

$$a_i\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} = L_i$$

$$\begin{aligned} &f(s_1, \ldots, s_m)\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} \\ &\qquad = \{f(t_1, \ldots, t_m) \mid t_i \in s_i\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}\} \end{aligned}$$

- And generalize this to languages

$$L\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} := \bigcup_{t \in L} (t\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\})$$

Substitution and Concatenation

- Let $\mathcal{K} := \Box_1/0, \Box_2/0, \ldots$. Assume $\mathcal{K} \cap \mathcal{F} = \emptyset$
- For trees $t \in T(\mathcal{F} \cup \mathcal{K})$, we define (simultaneous) substitution $t\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$, for $a_i \in \mathcal{K}$ and $i \neq j \implies a_i \neq a_j$:

$$\begin{aligned}
&a\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} = a \text{ for } a \in \mathcal{F} \cup \mathcal{K} \text{ and } \forall i.\ a \neq a_i \\
&a_i\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} = L_i \\
&f(s_1, \ldots, s_m)\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} \\
&\qquad\qquad = \{f(t_1, \ldots, t_m) \mid t_i \in s_i\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}\}
\end{aligned}$$

- And generalize this to languages

$$L\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\} := \bigcup_{t \in L} (t\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\})$$

- And define concatenation

$$L_1 \cdot_i L_2 := L_1\{\Box_i \leftarrow L_2\}$$

Iteration

- Iteration $L^{n,i}$

$$L^{0,i} := \square_i \qquad\qquad L^{n+1,i} = L^{n,i} \cup L \cdot_i L^{n,i}$$

Iteration

- Iteration $L^{n,i}$

$$L^{0,i} := \square_i \qquad\qquad L^{n+1,i} = L^{n,i} \cup L \cdot_i L^{n,i}$$

  - Note: All numbers $\leq n$ of iterations included.

Iteration

- Iteration $L^{n,i}$

$$L^{0,i} := \Box_i \qquad\qquad L^{n+1,i} = L^{n,i} \cup L \cdot_i L^{n,i}$$

  - Note: All numbers $\leq n$ of iterations included.
  - If there are many concatenation points, number of iterations is independent for each concatenation point.

Iteration

- Iteration $L^{n,i}$

$$L^{0,i} := \square_i \qquad\qquad L^{n+1,i} = L^{n,i} \cup L \cdot_i L^{n,i}$$

  - Note: All numbers $\leq n$ of iterations included.
  - If there are many concatenation points, number of iterations is independent for each concatenation point.
  - For example: $f(f(\square, f(\square, \square)), \square) \in \{f(\square, \square)\}^3$

Iteration

- Iteration $L^{n,i}$

$$L^{0,i} := \Box_i \qquad\qquad L^{n+1,i} = L^{n,i} \cup L \cdot_i L^{n,i}$$

  - Note: All numbers $\leq n$ of iterations included.
  - If there are many concatenation points, number of iterations is independent for each concatenation point.
  - For example: $f(f(\Box, f(\Box, \Box)), \Box) \in \{f(\Box, \Box)\}^3$
- Closure L^{*_i}

$$L^{*_i} := \bigcup_{n \in \mathbb{N}} L^{n,i}$$

Preservation of Regularity (Concatenation)

Theorem

Substitution preserves regularity, i.e., let $L, L_1, \ldots, L_n$ be regular languages, then $L' := L\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$ is a regular language

- Proof sketch:

Preservation of Regularity (Concatenation)

Theorem

Substitution preserves regularity, i.e., let $L, L_1, \ldots, L_n$ be regular languages, then $L' := L\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$ is a regular language

- Proof sketch:
  - Let $L, L_1, \ldots, L_i$ be represented by RTGs over disjoint nonterminals
    - $G = (S, N, \mathcal{F}, R)$ with $L = L(G)$ and $G_i = (S_i, N_i, \mathcal{F}, R_i)$ with $L_i = L(G_i)$

Preservation of Regularity (Concatenation)

Theorem

Substitution preserves regularity, i.e., let $L, L_1, \ldots, L_n$ be regular languages, then $L' := L\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$ is a regular language

- Proof sketch:
  - Let $L, L_1, \ldots, L_i$ be represented by RTGs over disjoint nonterminals
    - $G = (S, N, \mathcal{F}, R)$ with $L = L(G)$ and $G_i = (S_i, N_i, \mathcal{F}, R_i)$ with $L_i = L(G_i)$
  - Then let $G' = (S, N \cup N_1 \cup \ldots \cup N_n, \mathcal{F}, R' \cup R_1 \cup \ldots \cup R_n)$ where R' contains the rules of R, but a_i replaced by S_i.

Preservation of Regularity (Concatenation)

Theorem

Substitution preserves regularity, i.e., let $L, L_1, \ldots, L_n$ be regular languages, then $L' := L\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$ is a regular language

- Proof sketch:
  - Let $L, L_1, \ldots, L_i$ be represented by RTGs over disjoint nonterminals
    - $G = (S, N, \mathcal{F}, R)$ with $L = L(G)$ and $G_i = (S_i, N_i, \mathcal{F}, R_i)$ with $L_i = L(G_i)$
  - Then let $G' = (S, N \cup N_1 \cup \ldots \cup N_n, \mathcal{F}, R' \cup R_1 \cup \ldots \cup R_n)$ where R' contains the rules of R, but a_i replaced by S_i.
  - $L' \subseteq L(G')$: Produce word from L first (the $\Box_i$ are replaced by S_i), then rewrite the S_i to words from L_i

Preservation of Regularity (Concatenation)

Theorem

Substitution preserves regularity, i.e., let $L, L_1, \ldots, L_n$ be regular languages, then $L' := L\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$ is a regular language

- Proof sketch:
  - Let $L, L_1, \ldots, L_i$ be represented by RTGs over disjoint nonterminals
    - $G = (S, N, \mathcal{F}, R)$ with $L = L(G)$ and $G_i = (S_i, N_i, \mathcal{F}, R_i)$ with $L_i = L(G_i)$
  - Then let $G' = (S, N \cup N_1 \cup \ldots \cup N_n, \mathcal{F}, R' \cup R_1 \cup \ldots \cup R_n)$ where R' contains the rules of R, but a_i replaced by S_i.
  - $L' \subseteq L(G')$: Produce word from L first (the $\square_i$ are replaced by S_i), then rewrite the S_i to words from L_i
  - $L(G') \subseteq L'$: Re-order derivation of G' to stop at the S_i

Preservation of Regularity (Concatenation)

Theorem

Substitution preserves regularity, i.e., let $L, L_1, \ldots, L_n$ be regular languages, then $L' := L\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$ is a regular language

- Proof sketch:
  - Let $L, L_1, \ldots, L_i$ be represented by RTGs over disjoint nonterminals
    - $G = (S, N, \mathcal{F}, R)$ with $L = L(G)$ and $G_i = (S_i, N_i, \mathcal{F}, R_i)$ with $L_i = L(G_i)$
  - Then let $G' = (S, N \cup N_1 \cup \ldots \cup N_n, \mathcal{F}, R' \cup R_1 \cup \ldots \cup R_n)$ where R' contains the rules of R, but a_i replaced by S_i.
  - $L' \subseteq L(G')$: Produce word from L first (the $\square_i$ are replaced by S_i), then rewrite the S_i to words from L_i
  - $L(G') \subseteq L'$: Re-order derivation of G' to stop at the S_i
    - Formally, show:
      $\forall A \in N.\ A \rightarrow_{G'} s' \implies \exists s.\ A \rightarrow_G s \wedge s' \in s\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$

Preservation of Regularity (Concatenation)

Theorem

Substitution preserves regularity, i.e., let $L, L_1, \ldots, L_n$ be regular languages, then $L' := L\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$ is a regular language

- Proof sketch:
  - Let $L, L_1, \ldots, L_i$ be represented by RTGs over disjoint nonterminals
    - $G = (S, N, \mathcal{F}, R)$ with $L = L(G)$ and $G_i = (S_i, N_i, \mathcal{F}, R_i)$ with $L_i = L(G_i)$
  - Then let $G' = (S, N \cup N_1 \cup \ldots \cup N_n, \mathcal{F}, R' \cup R_1 \cup \ldots \cup R_n)$ where R' contains the rules of R, but a_i replaced by S_i.
  - $L' \subseteq L(G')$: Produce word from L first (the $\Box_i$ are replaced by S_i), then rewrite the S_i to words from L_i
  - $L(G') \subseteq L'$: Re-order derivation of G' to stop at the S_i
    - Formally, show: $\forall A \in N.\ A \rightarrow_{G'} s' \implies \exists s.\ A \rightarrow_G s \wedge s' \in s\{a_1 \leftarrow L_1, \ldots, a_n \leftarrow L_n\}$
    - By induction on derivation length

Preservation of Regularity (Concatenation)

Theorem

Substitution preserves regularity, i.e., let $L, L_1, \dots, L_n$ be regular languages, then $L' := L\{a_1 \leftarrow L_1, \dots, a_n \leftarrow L_n\}$ is a regular language

- Proof sketch:
  - Let $L, L_1, \dots, L_i$ be represented by RTGs over disjoint nonterminals
    - $G = (S, N, \mathcal{F}, R)$ with $L = L(G)$ and $G_i = (S_i, N_i, \mathcal{F}, R_i)$ with $L_i = L(G_i)$
  - Then let $G' = (S, N \cup N_1 \cup \dots \cup N_n, \mathcal{F}, R' \cup R_1 \cup \dots \cup R_n)$ where R' contains the rules of R, but a_i replaced by S_i.
  - $L' \subseteq L(G')$: Produce word from L first (the $\Box_i$ are replaced by S_i), then rewrite the S_i to words from L_i
  - $L(G') \subseteq L'$: Re-order derivation of G' to stop at the S_i
    - Formally, show: $\forall A \in N.\ A \rightarrow_{G'} s' \implies \exists s.\ A \rightarrow_G s \wedge s' \in s\{a_1 \leftarrow L_1, \dots, a_n \leftarrow L_n\}$
    - By induction on derivation length
- Corollary: Concatenation preserves regularity, i.e., for regular languages L_1, L_2, the language $L_1 \cdot L_2$ is regular.

Preservation of Regularity (Closure)

Theorem

Closure preserves regularity, i.e., let L be a regular language. Then, L^* is a regular language.

- Proof sketch

Preservation of Regularity (Closure)

Theorem

Closure preserves regularity, i.e., let L be a regular language. Then, L^* is a regular language.

- Proof sketch
  - Let L be represented by RTG $G = (S, N, \mathcal{F}, R)$

Preservation of Regularity (Closure)

Theorem

Closure preserves regularity, i.e., let L be a regular language. Then, L^* is a regular language.

- Proof sketch
  - Let L be represented by RTG $G = (S, N, \mathcal{F}, R)$
  - Construct $G' = (S', N \,\dot\cup\, \{S'\}, \mathcal{F} \cup \mathcal{K}, R')$, such that

Preservation of Regularity (Closure)

Theorem

Closure preserves regularity, i.e., let L be a regular language. Then, L^* is a regular language.

- Proof sketch
  - Let L be represented by RTG $G = (S, N, \mathcal{F}, R)$
  - Construct $G' = (S', N \dot{\cup} \{S'\}, \mathcal{F} \cup \mathcal{K}, R')$, such that
    - R' contains the rules from R, with $\Box$ replaced by S'

Preservation of Regularity (Closure)

Theorem

Closure preserves regularity, i.e., let L be a regular language. Then, L^* is a regular language.

- Proof sketch
  - Let L be represented by RTG $G = (S, N, \mathcal{F}, R)$
  - Construct $G' = (S', N \,\dot{\cup}\, \{S'\}, \mathcal{F} \cup \mathcal{K}, R')$, such that
    - R' contains the rules from R, with $\Box$ replaced by S'
    - $S' \to \Box \in R'$ and $S' \to S \in R'$

Preservation of Regularity (Closure)

Theorem

Closure preserves regularity, i.e., let L be a regular language. Then, L^* is a regular language.

- Proof sketch
  - Let L be represented by RTG $G = (S, N, \mathcal{F}, R)$
  - Construct $G' = (S', N \dot{\cup} \{S'\}, \mathcal{F} \cup \mathcal{K}, R')$, such that
    - R' contains the rules from R, with $\square$ replaced by S'
    - $S' \to \square \in R'$ and $S' \to S \in R'$
  - $L^* \subseteq L(G')$: Obvious by construction

Preservation of Regularity (Closure)

Theorem

Closure preserves regularity, i.e., let L be a regular language. Then, L^* is a regular language.

- Proof sketch
  - Let L be represented by RTG $G = (S, N, \mathcal{F}, R)$
  - Construct $G' = (S', N \dot{\cup} \{S'\}, \mathcal{F} \cup \mathcal{K}, R')$, such that
    - R' contains the rules from R, with $\Box$ replaced by S'
    - $S' \to \Box \in R'$ and $S' \to S \in R'$
  - $L^* \subseteq L(G')$: Obvious by construction
  - $L(G') \subseteq L^*$: Re-ordering derivation. Formally: Induction on derivation length.

Tree Regular Expressions

- Syntax

$$e ::= \emptyset \mid f(\underbrace{e, \ldots, e}_{n \text{ times}}) \text{ for } f \in \mathcal{F}_n \mid e + e \mid e \cdot_i e \mid e^{*_i}$$

Tree Regular Expressions

- Syntax

$$e ::= \emptyset \mid f(\underbrace{e, \ldots, e}_{n \text{ times}}) \text{ for } f \in \mathcal{F}_n \mid e + e \mid e \cdot_i e \mid e^{*_i}$$

- Semantics

$$\begin{aligned}
[\![\emptyset]\!] &= \emptyset \\
[\![f(e_1, \ldots, e_n)]\!] &= \{f(t_1, \ldots, t_n) \mid t_i \in [\![e_i]\!]\} \\
[\![e_1 + e_2]\!] &= [\![e_1]\!] \cup [\![e_2]\!] \\
[\![e_1 \cdot_i e_2]\!] &= [\![e_1]\!] \cdot_i [\![e_2]\!] \\
[\![e_1^{*_i}]\!] &= [\![e_1]\!]^{*_i}
\end{aligned}$$

Kleene Theorem for Tree Languages

Theorem

A tree language L is regular if and only if there is a regular expression e with $L = [\![e]\!]$

- Proof ($\Longleftarrow$): Straightforward, by induction on e, using preservation of regularity by union, concatenation, and closure

Kleene Theorem for Tree Languages

Theorem

A tree language L is regular if and only if there is a regular expression e with $L = [\![e]\!]$

- Proof ($\Longleftarrow$): Straightforward, by induction on e, using preservation of regularity by union, concatenation, and closure
- Proof ($\Longrightarrow$): Construct reg-exp inductively over increasing number of states

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(\mathcal{F} \cup K)$ that can be rewritten to q_i using only **internal** states from $\{q_1, \ldots, q_k\}$

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(\mathcal{F} \cup K)$ that can be rewritten to q_i using only **internal** states from $\{q_1, \ldots, q_k\}$
  - Note: We do not require $q_i \in \{q_1, \ldots, q_k\}$, nor $K \subseteq \{q_1, \ldots, q_k\}$

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(\mathcal{F} \cup K)$ that can be rewritten to q_i using only **internal** states from $\{q_1, \ldots, q_k\}$
  - Note: We do not require $q_i \in \{q_1, \ldots, q_k\}$, nor $K \subseteq \{q_1, \ldots, q_k\}$
- $L(\mathcal{A}) = \bigcup_{i \mid q_i \in Q_F} T(i, n, \emptyset)$

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(\mathcal{F} \cup K)$ that can be rewritten to q_i using only **internal** states from $\{q_1, \ldots, q_k\}$
  - Note: We do not require $q_i \in \{q_1, \ldots, q_k\}$, nor $K \subseteq \{q_1, \ldots, q_k\}$
- $L(\mathcal{A}) = \bigcup_{i | q_i \in Q_F} T(i, n, \emptyset)$
- $T(i, 0, K)$ is finite

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(\mathcal{F} \cup K)$ that can be rewritten to q_i using only **internal** states from $\{q_1, \ldots, q_k\}$
  - Note: We do not require $q_i \in \{q_1, \ldots, q_k\}$, nor $K \subseteq \{q_1, \ldots, q_k\}$
- $L(\mathcal{A}) = \bigcup_{i | q_i \in Q_F} T(i, n, \emptyset)$
- $T(i, 0, K)$ is finite
  - Runs accepting $t \in T(i, 0, K)$ contain no internal states

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(\mathcal{F} \cup K)$ that can be rewritten to q_i using only **internal** states from $\{q_1, \ldots, q_k\}$
  - Note: We do not require $q_i \in \{q_1, \ldots, q_k\}$, nor $K \subseteq \{q_1, \ldots, q_k\}$
- $L(\mathcal{A}) = \bigcup_{i \mid q_i \in Q_F} T(i, n, \emptyset)$
- $T(i, 0, K)$ is finite
  - Runs accepting $t \in T(i, 0, K)$ contain no internal states
  - I.e., $t = a()$ or $t = f(a_1, \ldots a_m)$, for $a, a_1, \ldots a_m \in \mathcal{F} \cup K$

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \dots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(\mathcal{F} \cup K)$ that can be rewritten to q_i using only **internal** states from $\{q_1, \dots, q_k\}$
  - Note: We do not require $q_i \in \{q_1, \dots, q_k\}$, nor $K \subseteq \{q_1, \dots, q_k\}$
- $L(\mathcal{A}) = \bigcup_{i \mid q_i \in Q_F} T(i, n, \emptyset)$
- $T(i, 0, K)$ is finite
  - Runs accepting $t \in T(i, 0, K)$ contain no internal states
  - I.e., $t = a()$ or $t = f(a_1, \dots a_m)$, for $a, a_1, \dots a_m \in \mathcal{F} \cup K$
  - Thus, representable by regular expression

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(\mathcal{F} \cup K)$ that can be rewritten to q_i using only **internal** states from $\{q_1, \ldots, q_k\}$
  - Note: We do not require $q_i \in \{q_1, \ldots, q_k\}$, nor $K \subseteq \{q_1, \ldots, q_k\}$
- $L(\mathcal{A}) = \bigcup_{i \mid q_i \in Q_F} T(i, n, \emptyset)$
- $T(i, 0, K)$ is finite
  - Runs accepting $t \in T(i, 0, K)$ contain no internal states
  - I.e., $t = a()$ or $t = f(a_1, \ldots a_m)$, for $a, a_1, \ldots a_m \in \mathcal{F} \cup K$
  - Thus, representable by regular expression
- For $j > 0$:

$$T(i,j,K) = \underbrace{T(i, j-1, K \cup \{q_j\})}_{\text{Initial segment}} \cdot_{q_j} \underbrace{T(j, j-1, K \cup \{q_j\})^{*, q_j}}_{\text{Runs between } q_j\text{s}} \cdot_{q_j} \underbrace{T(j, j-1, K)}_{\text{Final segment}}$$

Kleene Theorem for Tree Languages (Proof)

- Let $\mathcal{A} = (Q, \mathcal{F}, Q_F, \Delta)$ be bottom-up automaton.
  - Let $Q = \{q_1, \ldots, q_n\}$
- Define $T(i, j, K)$ for $K \subseteq Q$ as those trees over $T(\mathcal{F} \cup K)$ that can be rewritten to q_i using only **internal** states from $\{q_1, \ldots, q_k\}$
  - Note: We do not require $q_i \in \{q_1, \ldots, q_k\}$, nor $K \subseteq \{q_1, \ldots, q_k\}$
- $L(\mathcal{A}) = \bigcup_{i \mid q_i \in Q_F} T(i, n, \emptyset)$
- $T(i, 0, K)$ is finite
  - Runs accepting $t \in T(i, 0, K)$ contain no internal states
  - I.e., $t = a()$ or $t = f(a_1, \ldots a_m)$, for $a, a_1, \ldots a_m \in \mathcal{F} \cup K$
  - Thus, representable by regular expression
- For $j > 0$:

$$T(i, j, K) = \underbrace{T(i, j-1, K \cup \{q_j\})}_{\text{Initial segment}} \cdot_{q_j} \underbrace{T(j, j-1, K \cup \{q_j\})^{*, q_j}}_{\text{Runs between } q_j\text{s}} \cdot_{q_j} \underbrace{T(j, j-1, K)}_{\text{Final segment}}$$

- Regular expression for $L(\mathcal{A})$ can be constructed

Last Lecture

- Tree regular expressions
- Kleene theorem
  - Tree regular expressions can express exactly the tree regular languages

Table of Contents

Table of Contents

Program Analysis

- Theorem of Rice: Properties of programs undecidable
- Need approximations
- Standard approximation: Ignore branching conditions
  - **if** (b) ... **else** ... Consider both branches, independent of *b*
  - Nondeterministic program

Attack Plan

- Properties: Reachability of configuration/regular set of configurations
- First, consider programs with recursion
  - Modeled by pushdown systems (PDS)
- Then, add process creation
  - Modeled by dynamic pushdown systems (DPN)
- Then synchronization through well-nested locks
  - DPN with locks

Recursion

- If program has no procedures
  - Runs can be described by word automaton
  - Example on board
- If program has procedures
  - Runs can be described by push-down system (PDS)

Example

```
    void p() {
1:    if (...) p() else return;
2:    x=y;
3:    return;
    }
```

$1 \overset{\tau}{\hookrightarrow} 12$

$1 \overset{\tau}{\hookrightarrow} \varepsilon$

$2 \overset{x=y}{\hookrightarrow} 3$

$3 \overset{\tau}{\hookrightarrow} \varepsilon$

Table of Contents

Push-Down Systems (PDS)

- In order to model (finitely many) return values, we add state
- A *push-down system* (PDS) M is a tuple $(P, \Gamma, \mathrm{Act}, p_0, \gamma_0, \Delta)$ where
  - P is a finite set of states
  - Γ is a finite stack alphabet
  - Act is a finite set of actions
  - $p_0\gamma_0 \in P\Gamma$ is the initial configuration
  - Δ is a finite set of rules, of the form

$$p\gamma \overset{a}{\hookrightarrow} p'w \text{ where } p, p' \in P,\ a \in \mathrm{Act},\ \gamma \in \Gamma, \text{ and } w \in \Gamma^*$$

PDS - Semantics

- Configurations have the form $pw \in P\Gamma^*$
- The step-relation $\rightarrow \subseteq P\Gamma^* \times \mathrm{Act} \times P\Gamma^*$ is defined by

$$p\gamma w \xrightarrow{a} p'w'w \text{ if } p\gamma \overset{a}{\hookrightarrow} p'w' \in \Delta$$

- $\rightarrow^* \subseteq P\Gamma^* \times \mathrm{Act}^* \times P\Gamma^*$ is its extension to sequences of steps
  - $pw \xrightarrow{l}{}^* p'w'$ iff $l = a_1 \ldots a_n$ and $pw \overset{a_1}{\hookrightarrow} \ldots \overset{a_n}{\hookrightarrow} p'w'$

Normalized PDS

- Simplifying assumptions
  - There are only three types of rules

$$p\gamma \overset{a}{\hookrightarrow} p'\gamma' \qquad \text{for } p, p' \in P \text{ and } \gamma, \gamma' \in \Gamma \qquad \text{(base)}$$

$$p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \qquad \text{for } p, p' \in P \text{ and } \gamma, \gamma_1, \gamma_2 \in \Gamma \qquad \text{(call)}$$

$$p\gamma \overset{a}{\hookrightarrow} p' \qquad \text{for } p, p' \in P \text{ and } \gamma \in \Gamma \qquad \text{(return)}$$

    - Does not reduce expressiveness. Emulate rule $p\gamma \overset{\gamma}{\hookrightarrow}_1 \dots \gamma_n$ by sequence of call rules.
  - The empty stack must not be reachable
    - Does not reduce expressiveness
    - Introduce fresh $\perp$ stack symbol, a rule $p_0\perp \overset{\tau}{\hookrightarrow} p_0\gamma_0\perp$, and set initial state to $p_0\perp$
    - τ models an action that has no effect (skip)
- From now on, we assume that PDS are normalized

Execution Trees

- Model executions of PDS as tree
  - Also incomplete executions, i.e., execution may stop everywhere
  - This describes all reachable configurations
- A node represents a step
- If a call returns, the call-node has two successors
  - Left successor describes execution of procedure
  - Right successor describes execution of remaining program
- Execution trees described by the following tree grammar

$$XR ::= \langle Base \rangle(XR) \mid \langle Call \rangle^R(XR, XR) \mid \langle Return \rangle$$

$$XN ::= \langle Base \rangle(XN) \mid \langle Call \rangle^N(XN) \mid \langle Call \rangle^R(XR, XN) \mid \langle P \times \Gamma \rangle$$

  - Where *Base*, *Call*, *Return* are rules of respective type
  - Intuition: XR – Returning execution trees, XN – non-returning execution trees

Example

$p1 \overset{\tau}{\hookrightarrow} p12$

$p2 \overset{x=y}{\hookrightarrow} p3$

$p3 \overset{\tau}{\hookrightarrow} p$

$p1 \overset{\tau}{\hookrightarrow} p$

- Example execution tree
  - $\langle p1 \overset{\tau}{\hookrightarrow} p12\rangle^N(\langle p1 \overset{\tau}{\hookrightarrow} p12\rangle^R(\langle p1 \overset{\tau}{\hookrightarrow} p\rangle, \langle p2 \overset{x=y}{\hookrightarrow} p3\rangle(\langle p3\rangle)))$

Execution Trees of PDS

- Execution trees of PDS $M = (P, \Gamma, \mathrm{Act}, p_0, \gamma_0, \Delta)$ described by tree automata $\mathcal{A}_M = (Q, \mathcal{F}, I, \Delta_{\mathcal{A}_M})$
- States: $Q = P\Gamma \cup P\Gamma|P$
  - $p\gamma$ – produce non-returning execution trees (from XN)
  - $p\gamma|p''$ – produce execution trees that return to state p'' (from XR)
  - Initial state: $I = \{p_0\gamma_0\}$
- Rules

$$
\begin{array}{ll}
p\gamma \to \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma' \rangle (p'\gamma') & \text{if } p\gamma \overset{a}{\hookrightarrow} p'\gamma' \in \Delta \\
p\gamma \to \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^N (p'\gamma_1) & \text{if } p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \in \Delta \\
p\gamma \to \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^R (p'\gamma_1|p'', p''\gamma_2) & \text{if } p'' \in P \text{ and } p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \in \Delta \\
p\gamma \to \langle p\gamma \rangle & \\
p\gamma|p'' \to \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma' \rangle (p'\gamma'|p'') & \text{if } p\gamma \overset{a}{\hookrightarrow} p'\gamma' \in \Delta \\
p\gamma|p'' \to \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^R (p'\gamma_1|p''', p'''\gamma_2|p'') & \text{if } p''' \in P \text{ and } p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \in \Delta \\
p\gamma|p'' \to \langle p\gamma \overset{\tau}{\hookrightarrow} p'' \rangle & \text{if } p\gamma \overset{\tau}{\hookrightarrow} p'' \in \Delta
\end{array}
$$

Execution Trees – Intuition of rules

- $p\gamma \to \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma' \rangle (p'\gamma')$ (Base)
  - Make a base step, then continue execution from $p'\gamma'$
- $p\gamma \to \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^N (p'\gamma_1)$ (Call, no-return)
  - Continue execution from $p'\gamma_1$.
  - As call does not return, γ_2 is never looked at again, and remaining execution does not depend on it
- $p\gamma \to \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^R (p'\gamma_1 | p'', p''\gamma_2)$ (Call, return)
  - Execute procedure, it returns with state p''. Then continue execution from $p''\gamma_2$.
- $p\gamma \to \langle p\gamma \rangle$ (Finish)
  - Non-deterministically decide that execution ends here
- $p\gamma | p'' \to \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma' \rangle (p'\gamma' | p'')$ (Base)
  - Base step, then continue execution
- $p\gamma | p'' \to \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^R (p'\gamma_1 | p''', p'''\gamma_2 | p'')$ (Call, return)
  - Return from called procedure in state p''', then continue execution
- $p\gamma | p'' \to \langle p\gamma \overset{\tau}{\hookrightarrow} p'' \rangle$ (Return)
  - Return rule returns to specified state p''

Reached Configuration

- Function $c : XN \to P\Gamma$ extracts reached configuration from execution tree

$$
\begin{aligned}
c(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma'\rangle(t)) &= c(t) \\
c(\langle p\gamma \overset{\tau}{\hookrightarrow} p'\gamma_1\gamma_2\rangle^R(t_1, t_2)) &= c(t_2) \\
c(\langle p\gamma \overset{\tau}{\hookrightarrow} p'\gamma_1\gamma_2\rangle^N(t)) &= c(t)\gamma_2 \\
c(\langle p\gamma\rangle) &= p\gamma
\end{aligned}
$$

- Side note: This is a tree to string transducer
  - Thus, set of execution trees that reach a regular set of configurations is regular

Last Lecture

- Pushdown systems
  - Configuration $pw \in P\Gamma^*$
  - Semantics by step relation
- Execution trees
  - Intuition: Node for steps. Returning call nodes are binary.
  - Set of execution trees of PDS is regular
  - Mapping of execution tree to reached configuration
- Correlation:
  - Reachable configurations wrt. step relation and execution trees match

Relating Execution Trees and PDS Semantics

Theorem

Let M be a PDS. Then $\exists l.\ p_0\gamma_0 \xrightarrow{l}{}^{*} p'w$ *iff* $\exists t.\ t \in L(\mathcal{A}_M) \wedge c(t) = p'w$

- Note, a more general theorem would also relate the sequence of actions *l* and the execution tree
  - Proof ideas are the same

Last Lecture

- Proof of relation between execution trees and PDS semantics

Proof Outline

- Prove, for returning executions: $\exists l.\ p\gamma \xrightarrow{l}{}^{*} p''$ iff $\exists t.\ p\gamma | p'' \rightarrow t$
  - As c ignores returning executions, this simple statement is enough
- Prove, for non-returning executions:

$$\exists l.\ p\gamma \xrightarrow{l}{}^{*} p'w \wedge w \neq \varepsilon \text{ iff } \exists t.\ p\gamma \rightarrow t \wedge c(t) = p'w$$

- Main lemmas that are required
  - An execution can be repeated when we append some symbols to the stack:

$$\text{lemma stack-append: } pw \xrightarrow{l}{}^{*} p'w' \implies pwv \xrightarrow{l}{}^{*} p'w'v$$

  - If we have an execution, the topmost stack-symbol is either popped at some point, or the execution does not depend on the stack below the topmost symbol. Lemma return-cases:

$$p\gamma w \xrightarrow{l}{}^{*} p'w' \implies$$

$$\exists p''\ l_1\ l_2.\ p\gamma \xrightarrow{l_1}{}^{*} p'' \wedge p''w \xrightarrow{l_2}{}^{*} p'w' \wedge l = l_1 l_2 \qquad \text{(ret)}$$

$$\vee\ \exists w''.\ w' = w''w \wedge w'' \neq \varepsilon \wedge p\gamma \xrightarrow{l}{}^{*} p'w'' \qquad \text{(no-ret)}$$

  - Corollary: On a returning execution, we can find the point where the topmost stack symbol is popped

$$\text{lemma find-return: } p\gamma w \xrightarrow{l}{}^{*} p' \implies \exists l_1\ l_2\ p''.\ p\gamma \xrightarrow{l_1}{}^{*} p'' \wedge p''w \xrightarrow{l_2}{}^{*} p'$$

Proofs:

- On board
  - lemma return-cases (find-return is corollary)
  - Proofs for returning and non-returning executions

Table of Contents

Thread Creation

- Concurrent programs may create threads
- These run in parallel

Example

```
void p () {
  if (...) {
    spawn p;
    p();
  }
}

main () {
  p();
}
```

Dynamic Pushdown Networks

- Pushdown systems

Dynamic Pushdown Networks

- Pushdown systems
- Spawn-rules may have side-effect of creating a new PDS

Dynamic Pushdown Networks

- Pushdown systems
- Spawn-rules may have side-effect of creating a new PDS
- A DPN $M = (P, \Gamma, \mathrm{Act}, p_0, \gamma_0, \Delta)$ consists of

Dynamic Pushdown Networks

- Pushdown systems
- Spawn-rules may have side-effect of creating a new PDS
- A DPN $M = (P, \Gamma, \mathrm{Act}, p_0, \gamma_0, \Delta)$ consists of
  - A finite set of states P

Dynamic Pushdown Networks

- Pushdown systems
- Spawn-rules may have side-effect of creating a new PDS
- A DPN $M = (P, \Gamma, \mathrm{Act}, p_0, \gamma_0, \Delta)$ consists of
  - A finite set of states P
  - A finite set of stack symbols Γ

Dynamic Pushdown Networks

- Pushdown systems
- Spawn-rules may have side-effect of creating a new PDS
- A DPN $M = (P, \Gamma, \mathrm{Act}, p_0, \gamma_0, \Delta)$ consists of
  - A finite set of states P
  - A finite set of stack symbols Γ
  - A finite set of actions Act

Dynamic Pushdown Networks

- Pushdown systems
- Spawn-rules may have side-effect of creating a new PDS
- A DPN $M = (P, \Gamma, \mathrm{Act}, p_0, \gamma_0, \Delta)$ consists of
  - A finite set of states P
  - A finite set of stack symbols Γ
  - A finite set of actions Act
  - An initial configuration $p_0\gamma_0 \in P\Gamma$

Dynamic Pushdown Networks

- Pushdown systems
- Spawn-rules may have side-effect of creating a new PDS
- A DPN $M = (P, \Gamma, \mathrm{Act}, p_0, \gamma_0, \Delta)$ consists of
  - A finite set of states P
  - A finite set of stack symbols Γ
  - A finite set of actions Act
  - An initial configuration $p_0\gamma_0 \in P\Gamma$
  - Rules Δ of the form

$$
\begin{aligned}
p\gamma &\overset{a}{\hookrightarrow} p'\gamma' && \text{for } p, p' \in P \text{ and } \gamma, \gamma' \in \Gamma && \text{(base)}\\
p\gamma &\overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 && \text{for } p, p' \in P \text{ and } \gamma, \gamma_1, \gamma_2 \in \Gamma && \text{(call)}\\
p\gamma &\overset{a}{\hookrightarrow} p' && \text{for } p, p' \in P \text{ and } \gamma \in \Gamma && \text{(return)}\\
p\gamma &\overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 && \text{for } p, p_1, p_2 \in P \text{ and } \gamma, \gamma_1, \gamma_2 \in \Gamma && \text{(spawn)}
\end{aligned}
$$

Dynamic Pushdown Networks

- Pushdown systems
- Spawn-rules may have side-effect of creating a new PDS
- A DPN $M = (P, \Gamma, \mathrm{Act}, p_0, \gamma_0, \Delta)$ consists of
  - A finite set of states P
  - A finite set of stack symbols Γ
  - A finite set of actions Act
  - An initial configuration $p_0\gamma_0 \in P\Gamma$
  - Rules Δ of the form

$$
\begin{aligned}
p\gamma &\overset{a}{\hookrightarrow} p'\gamma' && \text{for } p, p' \in P \text{ and } \gamma, \gamma' \in \Gamma && \text{(base)} \\
p\gamma &\overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 && \text{for } p, p' \in P \text{ and } \gamma, \gamma_1, \gamma_2 \in \Gamma && \text{(call)} \\
p\gamma &\overset{a}{\hookrightarrow} p' && \text{for } p, p' \in P \text{ and } \gamma \in \Gamma && \text{(return)} \\
p\gamma &\overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 && \text{for } p, p_1, p_2 \in P \text{ and } \gamma, \gamma_1, \gamma_2 \in \Gamma && \text{(spawn)}
\end{aligned}
$$

- Assumption: Empty stack not reachable in any spawned thread

Configurations

- Configurations are trees over the alphabet $\langle \mathit{pw} \rangle/1 \mid \mathit{Cons}/2 \mid \mathit{Nil}/0$

Configurations

- Configurations are trees over the alphabet $\langle pw \rangle/1 \mid Cons/2 \mid Nil/0$
  - For all $pw \in P\Gamma^*$

Configurations

- Configurations are trees over the alphabet $\langle pw \rangle/1 \mid Cons/2 \mid Nil/0$
  - For all $pw \in P\Gamma^*$
- They have the structure
  $conf ::= \langle pw \rangle(conflist) \quad conflist ::= Nil|Cons(conf, conflist)$

Configurations

- Configurations are trees over the alphabet $\langle pw \rangle/1 \mid \mathit{Cons}/2 \mid \mathit{Nil}/0$
  - For all $pw \in P\Gamma^*$
- They have the structure
  $\mathit{conf} ::= \langle pw \rangle(\mathit{conflist}) \quad \mathit{conflist} ::= \mathit{Nil} | \mathit{Cons}(\mathit{conf}, \mathit{conflist})$
- Intuitively, a node $\langle pw \rangle(l)$ represents a thread in state pw, that has already spawned the threads in l

Configurations

- Configurations are trees over the alphabet $\langle pw \rangle/1 \mid Cons/2 \mid Nil/0$
  - For all $pw \in P\Gamma^*$
- They have the structure
  $conf ::= \langle pw \rangle(conflist) \quad conflist ::= Nil | Cons(conf, conflist)$
- Intuitively, a node $\langle pw \rangle(l)$ represents a thread in state pw, that has already spawned the threads in l
- Convention: We identify c with the singleton list $Cons(c, Nil)$, and use $l_1 l_2$ for the concatenation of l_1 and l_2.

Configurations

- Configurations are trees over the alphabet $\langle pw \rangle/1 \mid Cons/2 \mid Nil/0$
  - For all $pw \in P\Gamma^*$
- They have the structure
  $conf ::= \langle pw \rangle(conflist) \quad conflist ::= Nil | Cons(conf, conflist)$
- Intuitively, a node $\langle pw \rangle(l)$ represents a thread in state pw, that has already spawned the threads in l
- Convention: We identify c with the singleton list $Cons(c, Nil)$, and use $l_1 l_2$ for the concatenation of l_1 and l_2.
  - We may use $[c_1, \ldots, c_n]$ for the list $Cons(c_1, Cons(\ldots, Cons(c_n, Nil)\ldots)$ for clarification of notation.

Last Lecture

- Finished proof: Relation of execution trees and PDS semantics
- DPN (PDS + Thread creation)
- DPN-Semantics:
  - Configuration are trees, each node holds PDS-configuration (state+stack)
  - Children are threads that have been spawned by parent
- Extract reached configuration from execution tree

Semantics

- A step modifies a thread's state according to a rule

$$
\begin{aligned}
& C[\langle p\gamma w\rangle(I)] \overset{a}{\to} C[\langle p' w' w\rangle(I)] \\
& \qquad\qquad \text{if } p\gamma \overset{a}{\hookrightarrow} p' w' \in \Delta && \text{(no-spawn)} \\
& C[\langle p\gamma w\rangle(I)] \overset{a}{\to} C[\langle p_1\gamma_1 w\rangle(I\langle p_2\gamma_2\rangle(Nil))] \\
& \qquad\qquad \text{if } p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta && \text{(spawn)}
\end{aligned}
$$

Semantics

- A step modifies a thread's state according to a rule

$$C[\langle p\gamma w\rangle(l)] \xrightarrow{a} C[\langle p' w' w\rangle(l)]$$
$$\text{if } p\gamma \overset{a}{\hookrightarrow} p' w' \in \Delta \qquad \text{(no-spawn)}$$
$$C[\langle p\gamma w\rangle(l)] \xrightarrow{a} C[\langle p_1\gamma_1 w\rangle(l\langle p_2\gamma_2\rangle(Nil))]$$
$$\text{if } p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta \qquad \text{(spawn)}$$

  - For any context C with exactly one occurrence of x_1, such that $C[\langle p\gamma w\rangle(l)] \in \mathit{conf}$ is a configuration
    - Having exactly one occurrence of x_1 ensures that exactly one thread makes a step

Semantics

- A step modifies a thread's state according to a rule

$$
\begin{aligned}
& C[\langle p\gamma w\rangle(l)] \stackrel{a}{\to} C[\langle p' w' w\rangle(l)] \\
& \qquad\qquad \text{if } p\gamma \stackrel{a}{\hookrightarrow} p' w' \in \Delta && \text{(no-spawn)} \\
& C[\langle p\gamma w\rangle(l)] \stackrel{a}{\to} C[\langle p_1\gamma_1 w\rangle(l\langle p_2\gamma_2\rangle(Nil))] \\
& \qquad\qquad \text{if } p\gamma \stackrel{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta && \text{(spawn)}
\end{aligned}
$$

  - For any context C with exactly one occurrence of x_1, such that $C[\langle p\gamma w\rangle(l)] \in \mathit{conf}$ is a configuration
    - Having exactly one occurrence of x_1 ensures that exactly one thread makes a step
- Intuition:

Semantics

- A step modifies a thread's state according to a rule

$$C[\langle p\gamma w\rangle(l)] \xrightarrow{a} C[\langle p' w' w\rangle(l)]$$
$$\text{if } p\gamma \overset{a}{\hookrightarrow} p' w' \in \Delta \qquad \text{(no-spawn)}$$
$$C[\langle p\gamma w\rangle(l)] \xrightarrow{a} C[\langle p_1\gamma_1 w\rangle(l\langle p_2\gamma_2\rangle(Nil))]$$
$$\text{if } p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta \qquad \text{(spawn)}$$

  - For any context C with exactly one occurrence of x_1, such that $C[\langle p\gamma w\rangle(l)] \in \mathit{conf}$ is a configuration
    - Having exactly one occurrence of x_1 ensures that exactly one thread makes a step
- Intuition:
  - (no-spawn) rule just changes single thread's configuration

Semantics

- A step modifies a thread's state according to a rule

$$C[\langle p\gamma w\rangle(l)] \stackrel{a}{\to} C[\langle p' w' w\rangle(l)]$$
$$\text{if } p\gamma \stackrel{a}{\hookrightarrow} p' w' \in \Delta \qquad \text{(no-spawn)}$$
$$C[\langle p\gamma w\rangle(l)] \stackrel{a}{\to} C[\langle p_1\gamma_1 w\rangle(l\langle p_2\gamma_2\rangle(Nil))]$$
$$\text{if } p\gamma \stackrel{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta \qquad \text{(spawn)}$$

  - For any context C with exactly one occurrence of x_1, such that $C[\langle p\gamma w\rangle(l)] \in \mathit{conf}$ is a configuration
    - Having exactly one occurrence of x_1 ensures that exactly one thread makes a step
- Intuition:
  - (no-spawn) rule just changes single thread's configuration
  - (spawn) rule changes thread's configuration, and adds new thread to spawned thread's list

Execution Trees

- Binary node $\langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \rangle(t_1, t_2)$ describes execution of spawn-step

Execution Trees

- Binary node $\langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2\rangle(t_1, t_2)$ describes execution of spawn-step
  - t_1 describes remaining execution of spawning thread

Execution Trees

- Binary node $\langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \rangle(t_1, t_2)$ describes execution of spawn-step
  - t_1 describes remaining execution of spawning thread
  - t_2 describes execution of spawned thread

Execution Trees

- Binary node $\langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2\rangle(t_1, t_2)$ describes execution of spawn-step
  - t_1 describes remaining execution of spawning thread
  - t_2 describes execution of spawned thread
- Execution trees

$$XR ::= \langle Base\rangle(XR) \mid \langle Call\rangle^R(XR, XR) \mid \langle Return\rangle \mid \langle Spawn\rangle(XR, XN)$$

$$XN ::= \langle Base\rangle(XN) \mid \langle Call\rangle^N(XN) \mid \langle Call\rangle^R(XR, XN) \mid \langle P \times \Gamma\rangle \mid \langle Spawn\rangle(XN, XN)$$

List Operations

- We lift list-operations to concatenate lists and trees

List Operations

- We lift list-operations to concatenate lists and trees
  - $l_1\langle pw\rangle(l_2) = \langle pw\rangle(l_1 l_2)$

Configuration of Execution Tree

- Function $c : XN \to conf$

Configuration of Execution Tree

- Function $c : XN \rightarrow conf$
  - $c(\langle Spawn \rangle(t_1, t_2)) = [c(t_2)]c(t_1)$

Configuration of Execution Tree

- Function $c : XN \to conf$
  - $c(\langle Spawn \rangle(t_1, t_2)) = [c(t_2)]c(t_1)$
    - Prepend configuration reached by spawned thread

Configuration of Execution Tree

- Function $c : XN \to conf$
  - $c(\langle Spawn\rangle(t_1, t_2)) = [c(t_2)]c(t_1)$
    - Prepend configuration reached by spawned thread
  - $c(\langle Call\rangle^R(t_1, t_2)) = s(t_1)c(t_2)$

Configuration of Execution Tree

- Function $c : XN \to conf$
  - $c(\langle Spawn \rangle(t_1, t_2)) = [c(t_2)]c(t_1)$
    - Prepend configuration reached by spawned thread
  - $c(\langle Call \rangle^R(t_1, t_2)) = s(t_1)c(t_2)$
    - Have to collect configurations reached by threads spawned during call

Configuration of Execution Tree

- Function $c : XN \rightarrow conf$
  - $c(\langle Spawn \rangle(t_1, t_2)) = [c(t_2)]c(t_1)$
    - Prepend configuration reached by spawned thread
  - $c(\langle Call \rangle^R(t_1, t_2)) = s(t_1)c(t_2)$
    - Have to collect configurations reached by threads spawned during call
  - The remaining equations are unchanged (Complete definition on next slide)

Reached configurations

Define $c : XN \to \mathit{conf}$ and $s : XR \to \mathit{conflist}$

$$
\begin{aligned}
c(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma'\rangle(t)) &= c(t) \\
c(\langle p\gamma \overset{\tau}{\hookrightarrow} p'\gamma_1\gamma_2\rangle^R(t_1,t_2)) &= s(t_1)c(t_2) \\
c(\langle p\gamma \overset{\tau}{\hookrightarrow} p'\gamma_1\gamma_2\rangle^N(t)) &= c(t)\gamma_2 \qquad \text{where } \langle pw\rangle\gamma(l) = \langle pw\gamma\rangle(l) \\
c(\langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2\rangle(t_1,t_2)) &= [c(t_2)]c(t_1) \\
c(\langle p\gamma\rangle) &= \langle p\gamma\rangle \\
s(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma'\rangle(t)) &= s(t) \\
s(\langle p\gamma \overset{\tau}{\hookrightarrow} p'\gamma_1\gamma_2\rangle^R(t_1,t_2)) &= s(t_1)s(t_2) \\
s(\langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2\rangle(t_1,t_2)) &= [c(t_2)]s(t_1) \\
s(\langle p\gamma \overset{a}{\hookrightarrow} p'\rangle) &= \mathit{Nil}
\end{aligned}
$$

Execution trees of DPN

- Execution trees are regular set

Execution trees of DPN

- Execution trees are regular set
- Same idea as for PDS. New rules for $\mathcal{A}_M$:

$$p\gamma \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2\rangle(p_1\gamma_1, p_2\gamma_2) \qquad \text{if } p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta$$

$$p\gamma|p'' \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2\rangle(p_1\gamma_1|p'', p_2\gamma_2) \qquad \text{if } p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta$$

Execution trees of DPN

- Execution trees are regular set
- Same idea as for PDS. New rules for $\mathcal{A}_M$:

$$p\gamma \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2\rangle(p_1\gamma_1, p_2\gamma_2) \qquad \text{if } p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta$$

$$p\gamma|p'' \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2\rangle(p_1\gamma_1|p'', p_2\gamma_2) \qquad \text{if } p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta$$

  - Complete rules on next slide

Rules for execution trees

$$
\begin{array}{rll}
p\gamma & \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma' \rangle (p'\gamma') & \text{if } p\gamma \overset{a}{\hookrightarrow} p'\gamma' \in \Delta \\
p\gamma & \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^N (p'\gamma_1) & \text{if } p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \in \Delta \\
p\gamma & \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^R (p'\gamma_1 | p'', p''\gamma_2) & \text{if } p'' \in P \text{ and } p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \in \Delta \\
p\gamma & \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \rangle (p_1\gamma_1, p_2\gamma_2) & \text{if } p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta \\
p\gamma & \rightarrow \langle p\gamma \rangle & \\
p\gamma|p'' & \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma' \rangle (p'\gamma' | p'') & \text{if } p\gamma \overset{a}{\hookrightarrow} p'\gamma' \in \Delta \\
p\gamma|p'' & \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^R (p'\gamma_1 | p''', p'''\gamma_2 | p'') & \text{if } p''' \in P \text{ and } p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \in \Delta \\
p\gamma|p'' & \rightarrow \langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \rangle (p_1\gamma_1 | p'', p_2\gamma_2) & \text{if } p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \in \Delta \\
p\gamma|p'' & \rightarrow \langle p\gamma \overset{\tau}{\hookrightarrow} p'' \rangle & \text{if } p\gamma \overset{\tau}{\hookrightarrow} p'' \in \Delta
\end{array}
$$

Relating Execution Trees and DPN Semantics

Theorem

Let M be a DPN. Then $\exists l.\ p_0\gamma_0 \xrightarrow{l}{}^{*} c'$ iff $\exists t.\ t \in L(\mathcal{A}_M) \wedge c(t) = c'$

- Note: Relating the action sequences is more difficult

Relating Execution Trees and DPN Semantics

Theorem

Let M be a DPN. Then $\exists l.\ p_0\gamma_0 \xrightarrow{l}{}^{*} c'$ *iff* $\exists t.\ t \in L(\mathcal{A}_M) \wedge c(t) = c'$

- Note: Relating the action sequences is more difficult
  - They are *interleavings* of the thread's action sequences

Relating Execution Trees and DPN Semantics

Theorem

Let M be a DPN. Then $\exists l.\ p_0\gamma_0 \xrightarrow{l}^* c'$ *iff* $\exists t.\ t \in L(\mathcal{A}_M) \wedge c(t) = c'$

- Note: Relating the action sequences is more difficult
  - They are *interleavings* of the thread's action sequences
  - One execution tree corresponds to many such interleavings

Interleaving

- We define $s_1 \otimes s_2$ to be the set of *interleavings* of lists s_1 and s_2

$$s_1 \otimes \varepsilon = \{s_1\} \qquad \varepsilon \otimes s_2 = \{s_2\}$$
$$a_1 s_1 \otimes a_2 s_2 = a_1(s_1 \otimes a_2 s_2) \cup a_2(a_1 s_1 \otimes s_2)$$

- Intuitively: All sequences of steps that may be observed if one thread executes s_1 and another independently executes s_2.

Proof Ideas

- Execution of different threads is almost independent

Proof Ideas

- Execution of different threads is almost independent
  - Only spawn should be executed before other steps of spawned thread

Proof Ideas

- Execution of different threads is almost independent
  - Only spawn should be executed before other steps of spawned thread
  - Re-order step: On spawn, all steps of spawned thread first, and then the rest

Proof Ideas

- Execution of different threads is almost independent
  - Only spawn should be executed before other steps of spawned thread
  - Re-order step: On spawn, all steps of spawned thread first, and then the rest
  - Lemma indep-steps:

$$\langle pw\rangle([c]) \xrightarrow{s}{}^{*} \langle p'w'\rangle(l') \iff$$
$$\exists c'\ l''\ s_1\ s_2.\ l' = c'l'' \wedge s \in s_1 \otimes s_2 \wedge \langle pw\rangle(\varepsilon) \xrightarrow{s_1}{}^{*} \langle p'w'\rangle(l'') \wedge c \xrightarrow{s_2}{}^{*} c'$$

Proof Ideas

- Execution of different threads is almost independent
  - Only spawn should be executed before other steps of spawned thread
  - Re-order step: On spawn, all steps of spawned thread first, and then the rest
  - Lemma indep-steps:

$$\langle pw\rangle([c]) \xrightarrow{s}{}^{*} \langle p'w'\rangle(l') \iff$$
$$\exists c'\ l''\ s_1\ s_2.\ l' = c'l'' \wedge s \in s_1 \otimes s_2 \wedge \langle pw\rangle(\varepsilon) \xrightarrow{s_1}{}^{*} \langle p'w'\rangle(l'') \wedge c \xrightarrow{s_2}{}^{*} c'$$

- Proof, by induction on number of steps:

$$\langle p\gamma\rangle(\varepsilon) \rightarrow^{*} \langle p'\rangle(c') \iff \exists t.p\gamma|p' \rightarrow t \wedge s(t) = c'$$
$$\langle p\gamma\rangle(\varepsilon) \rightarrow^{*} \langle p'w'\rangle(c') \wedge w' \neq \varepsilon \iff \exists t.p\gamma \rightarrow t \wedge c(t) = \langle p'w'\rangle(c')$$

Proof Ideas

- Execution of different threads is almost independent
  - Only spawn should be executed before other steps of spawned thread
  - Re-order step: On spawn, all steps of spawned thread first, and then the rest
  - Lemma indep-steps:

$$\langle pw\rangle([c]) \xrightarrow{s}{}^{*} \langle p'w'\rangle(l') \iff$$
$$\exists c'\ l''\ s_1\ s_2.\ l' = c'l'' \land s \in s_1 \otimes s_2 \land \langle pw\rangle(\varepsilon) \xrightarrow{s_1}{}^{*} \langle p'w'\rangle(l'') \land c \xrightarrow{s_2}{}^{*} c'$$

- Proof, by induction on number of steps:

$$\langle p\gamma\rangle(\varepsilon) \to^{*} \langle p'\rangle(c') \iff \exists t. p\gamma|p' \to t \land s(t) = c'$$
$$\langle p\gamma\rangle(\varepsilon) \to^{*} \langle p'w'\rangle(c') \land w' \neq \varepsilon \iff \exists t. p\gamma \to t \land c(t) = \langle p'w'\rangle(c')$$

  - Need to prove both propositions simultaneously

Proof Ideas

- Execution of different threads is almost independent
  - Only spawn should be executed before other steps of spawned thread
  - Re-order step: On spawn, all steps of spawned thread first, and then the rest
  - Lemma indep-steps:

$$\langle pw\rangle([c]) \xrightarrow{s}{}^{*} \langle p'w'\rangle(l') \iff$$
$$\exists c'\ l''\ s_1\ s_2.\ l' = c'l'' \wedge s \in s_1 \otimes s_2 \wedge \langle pw\rangle(\varepsilon) \xrightarrow{s_1}{}^{*} \langle p'w'\rangle(l'') \wedge c \xrightarrow{s_2}{}^{*} c'$$

- Proof, by induction on number of steps:

$$\langle p\gamma\rangle(\varepsilon) \rightarrow^{*} \langle p'\rangle(c') \iff \exists t.p\gamma|p' \rightarrow t \wedge s(t) = c'$$
$$\langle p\gamma\rangle(\varepsilon) \rightarrow^{*} \langle p'w'\rangle(c') \wedge w' \neq \varepsilon \iff \exists t.p\gamma \rightarrow t \wedge c(t) = \langle p'w'\rangle(c')$$

  - Need to prove both propositions simultaneously
  - But may separate $\Longrightarrow$ and $\Longleftarrow$ directions

Example Proof Step

- Example step for $\Rightarrow$-direction

$$\langle p\gamma\rangle(\varepsilon) \rightarrow^* \langle p'\rangle(l') \implies \exists t.p\gamma|p' \rightarrow t \wedge s(t) = l'$$
$$\langle p\gamma\rangle(\varepsilon) \rightarrow^* \langle p'w'\rangle(l') \wedge w' \neq \varepsilon \implies \exists t.p\gamma \rightarrow t \wedge c(t) = \langle p'w'\rangle(l')$$

Example Proof Step

- Example step for $\Rightarrow$-direction

$$\langle p\gamma\rangle(\varepsilon) \rightarrow^* \langle p'\rangle(l') \implies \exists t. p\gamma|p' \rightarrow t \wedge s(t) = l'$$
$$\langle p\gamma\rangle(\varepsilon) \rightarrow^* \langle p'w'\rangle(l') \wedge w' \neq \varepsilon \implies \exists t. p\gamma \rightarrow t \wedge c(t) = \langle p'w'\rangle(l')$$

- Case: Returning path makes a spawn-step

Example Proof Step

- Example step for $\Rightarrow$-direction

$$\langle p\gamma\rangle(\varepsilon) \rightarrow^* \langle p'\rangle(l') \implies \exists t. p\gamma|p' \rightarrow t \wedge s(t) = l'$$
$$\langle p\gamma\rangle(\varepsilon) \rightarrow^* \langle p'w'\rangle(l') \wedge w' \neq \varepsilon \implies \exists t. p\gamma \rightarrow t \wedge c(t) = \langle p'w'\rangle(l')$$

- Case: Returning path makes a spawn-step
  - We have $r := p\gamma \hookrightarrow \hat{p}\hat{\gamma} \triangleright p_1\gamma_1 \in \Delta$ and $\langle \hat{p}\hat{\gamma}\rangle(p_1\gamma_1) \rightarrow^* \langle p'\rangle(c')$

Example Proof Step

- Example step for $\Rightarrow$-direction

$$\langle p\gamma\rangle(\varepsilon) \to^* \langle p'\rangle(l') \implies \exists t. p\gamma|p' \to t \wedge s(t) = l'$$
$$\langle p\gamma\rangle(\varepsilon) \to^* \langle p'w'\rangle(l') \wedge w' \neq \varepsilon \implies \exists t. p\gamma \to t \wedge c(t) = \langle p'w'\rangle(l')$$

- Case: Returning path makes a spawn-step
  - We have $r := p\gamma \hookrightarrow \hat{p}\hat{\gamma} \rhd p_1\gamma_1 \in \Delta$ and $\langle \hat{p}\hat{\gamma}\rangle(p_1\gamma_1) \to^* \langle p'\rangle(c')$
  - Using indep-steps, to separate executions of spawned and spawning thread, we obtain c', l'' where

$$l' = c'l'' \wedge \langle \hat{p}\hat{\gamma}\rangle\varepsilon \to^* \langle p'\rangle(l'') \wedge \langle p_1\gamma_1\rangle(\varepsilon) \to^* c'$$

Example Proof Step

- Example step for $\Rightarrow$-direction

$$\langle p\gamma\rangle(\varepsilon) \to^* \langle p'\rangle(l') \implies \exists t. p\gamma|p' \to t \wedge s(t) = l'$$
$$\langle p\gamma\rangle(\varepsilon) \to^* \langle p'w'\rangle(l') \wedge w' \neq \varepsilon \implies \exists t. p\gamma \to t \wedge c(t) = \langle p'w'\rangle(l')$$

- Case: Returning path makes a spawn-step
  - We have $r := p\gamma \hookrightarrow \hat{p}\hat{\gamma} \triangleright p_1\gamma_1 \in \Delta$ and $\langle\hat{p}\hat{\gamma}\rangle(p_1\gamma_1) \to^* \langle p'\rangle(c')$
  - Using indep-steps, to separate executions of spawned and spawning thread, we obtain c', l'' where

$$l' = c'l'' \wedge \langle\hat{p}\hat{\gamma}\rangle\varepsilon \to^* \langle p'\rangle(l'') \wedge \langle p_1\gamma_1\rangle(\varepsilon) \to^* c'$$

  - With IH, we obtain t_1, t_2 with

$$\hat{p}\hat{\gamma}|p' \to t_1 \wedge s(t_1) = l'' \wedge p_1\gamma_1 \to t_2 \wedge c(t_2) = c'$$

Example Proof Step

- Example step for $\Rightarrow$-direction

$$\langle p\gamma\rangle(\varepsilon) \to^* \langle p'\rangle(l') \implies \exists t. p\gamma|p' \to t \wedge s(t) = l'$$
$$\langle p\gamma\rangle(\varepsilon) \to^* \langle p'w'\rangle(l') \wedge w' \neq \varepsilon \implies \exists t. p\gamma \to t \wedge c(t) = \langle p'w'\rangle(l')$$

- Case: Returning path makes a spawn-step
  - We have $r := p\gamma \hookrightarrow \hat{p}\hat{\gamma} \rhd p_1\gamma_1 \in \Delta$ and $\langle \hat{p}\hat{\gamma}\rangle(p_1\gamma_1) \to^* \langle p'\rangle(c')$
  - Using indep-steps, to separate executions of spawned and spawning thread, we obtain c', l'' where

$$l' = c'l'' \wedge \langle \hat{p}\hat{\gamma}\rangle\varepsilon \to^* \langle p'\rangle(l'') \wedge \langle p_1\gamma_1\rangle(\varepsilon) \to^* c'$$

  - With IH, we obtain t_1, t_2 with

$$\hat{p}\hat{\gamma}|p' \to t_1 \wedge s(t_1) = l'' \wedge p_1\gamma_1 \to t_2 \wedge c(t_2) = c'$$

  - By definition of the rules for $\mathcal{A}_M$, we get

$$p\gamma|p' \to \langle r\rangle(\hat{p}\hat{\gamma}|p', p_1\gamma_1) \to \langle r\rangle(t_1, t_2)$$

Example Proof Step

- Example step for $\Rightarrow$-direction

$$\langle p\gamma\rangle(\varepsilon) \to^* \langle p'\rangle(l') \implies \exists t. p\gamma|p' \to t \wedge s(t) = l'$$
$$\langle p\gamma\rangle(\varepsilon) \to^* \langle p'w'\rangle(l') \wedge w' \neq \varepsilon \implies \exists t. p\gamma \to t \wedge c(t) = \langle p'w'\rangle(l')$$

- Case: Returning path makes a spawn-step
  - We have $r := p\gamma \hookrightarrow \hat{p}\hat{\gamma} \triangleright p_1\gamma_1 \in \Delta$ and $\langle\hat{p}\hat{\gamma}\rangle(p_1\gamma_1) \to^* \langle p'\rangle(c')$
  - Using indep-steps, to separate executions of spawned and spawning thread, we obtain c', l'' where

$$l' = c'l'' \wedge \langle\hat{p}\hat{\gamma}\rangle\varepsilon \to^* \langle p'\rangle(l'') \wedge \langle p_1\gamma_1\rangle(\varepsilon) \to^* c'$$

  - With IH, we obtain t_1, t_2 with

$$\hat{p}\hat{\gamma}|p' \to t_1 \wedge s(t_1) = l'' \wedge p_1\gamma_1 \to t_2 \wedge c(t_2) = c'$$

  - By definition of the rules for $\mathcal{A}_M$, we get

$$p\gamma|p' \to \langle r\rangle(\hat{p}\hat{\gamma}|p', p_1\gamma_1) \to \langle r\rangle(t_1, t_2)$$

  - And, by definition of $s()$, we have

$$s(\langle r\rangle(t_1, t_2)) = [c(t_2)]s(t_1) = c'l'' = l' \quad \square$$

Lock-Insensitive Reachability

- Can perform a simultaneous reachability analysis

Lock-Insensitive Reachability

- Can perform a simultaneous reachability analysis
- By asking: „Is a configuration from a regular set of configurations reachable?”

Lock-Insensitive Reachability

- Can perform a simultaneous reachability analysis
- By asking: „Is a configuration from a regular set of configurations reachable?"
    - If the analysis returns no, we are sure that no such configuration is reachable

Lock-Insensitive Reachability

- Can perform a simultaneous reachability analysis
- By asking: „Is a configuration from a regular set of configurations reachable?"
  - If the analysis returns no, we are sure that no such configuration is reachable
  - If the analysis returns yes, such a configuration may be reachable

Lock-Insensitive Reachability

- Can perform a simultaneous reachability analysis
- By asking: „Is a configuration from a regular set of configurations reachable?"
  - If the analysis returns no, we are sure that no such configuration is reachable
  - If the analysis returns yes, such a configuration may be reachable
    - Or it may be a false positive due to over-approximation

Lock-Sensitive Analysis

- Consider locks.

Lock-Sensitive Analysis

- Consider locks.
- Locks can be acquired and released, each lock can be acquired by at most one thread at the same time.

Lock-Sensitive Analysis

- Consider locks.
- Locks can be acquired and released, each lock can be acquired by at most one thread at the same time.
- Used to protect access to shared resources

Lock-Sensitive Analysis

- Consider locks.
- Locks can be acquired and released, each lock can be acquired by at most one thread at the same time.
- Used to protect access to shared resources
- We assume there is a finite set $\mathbb{L}$ of locks, and the actions $[_l$ (acquire) and $]_l$ (release) for every $l \in \mathbb{L}$

Decidability

- Reachability with arbitrary locking is undecidable

Decidability

- Reachability with arbitrary locking is undecidable
  - Emptiness of intersection of CF-Languages

Decidability

- Reachability with arbitrary locking is undecidable
  - Emptiness of intersection of CF-Languages
- Consider nested locking, like synchronized-methods in Java

Decidability

- Reachability with arbitrary locking is undecidable
  - Emptiness of intersection of CF-Languages
- Consider nested locking, like synchronized-methods in Java
  - Bind locks to procedures: Acquisition on call, release on return

Undecidability

- Well-Known: Emptiness of intersection of CF-languages is undecidable

Undecidability

- Well-Known: Emptiness of intersection of CF-languages is undecidable
  - Already over alphabet $\{0, 1\}$

Undecidability

- Well-Known: Emptiness of intersection of CF-languages is undecidable
  - Already over alphabet $\{0, 1\}$
- CF-language can be simulated by PDS, where only base-transitions produce output

Undecidability

- Well-Known: Emptiness of intersection of CF-languages is undecidable
  - Already over alphabet $\{0, 1\}$
- CF-language can be simulated by PDS, where only base-transitions produce output
  - Idea: Run two PDS concurrently, and ensure that sequences of base transitions must run in lock-step

Undecidability

- Well-Known: Emptiness of intersection of CF-languages is undecidable
  - Already over alphabet $\{0, 1\}$
- CF-language can be simulated by PDS, where only base-transitions produce output
  - Idea: Run two PDS concurrently, and ensure that sequences of base transitions must run in lock-step
  - These encode output of 0 and 1. Lockstep ensures, that the other thread must output the same.

Undecidability

- Well-Known: Emptiness of intersection of CF-languages is undecidable
  - Already over alphabet $\{0, 1\}$
- CF-language can be simulated by PDS, where only base-transitions produce output
  - Idea: Run two PDS concurrently, and ensure that sequences of base transitions must run in lock-step
  - These encode output of 0 and 1. Lockstep ensures, that the other thread must output the same.
  - Check for simultaneous reachability of final states

Undecidability

- Synchronizing two threads with locks

Undecidability

- Synchronizing two threads with locks
  - Locks: $0, 0!, 0?$ and $1, 1!, 1?$

Undecidability

- Synchronizing two threads with locks
  - Locks: $0, 0!, 0?$ and $1, 1!, 1?$
  - Assumption: Thread one initially holds $0!, 1!$, thread two initially holds $0?, 1?$

Undecidability

- Synchronizing two threads with locks
  - Locks: 0, 0!, 0? and 1, 1!, 1?
  - Assumption: Thread one initially holds 0!, 1!, thread two initially holds 0?, 1?
- To produce a 0:

Undecidability

- Synchronizing two threads with locks
  - Locks: 0, 0!, 0? and 1, 1!, 1?
  - Assumption: Thread one initially holds 0!, 1!, thread two initially holds 0?, 1?
- To produce a 0:
  - Thread 1 executes: $[_{0?}]_{0!}[_{0}]_{0?}[_{0!}]_{0}$

Undecidability

- Synchronizing two threads with locks
  - Locks: $0, 0!, 0?$ and $1, 1!, 1?$
  - Assumption: Thread one initially holds $0!, 1!$, thread two initially holds $0?, 1?$
- To produce a 0:
  - Thread 1 executes: $[_{0?}]_{0!}[_{0}]_{0?}[_{0!}]_{0}$
  - Thread 2 executes: $[_{0}]_{0?}[_{0!}]_{0}[_{0?}]_{0!}$

Undecidability

- Synchronizing two threads with locks
  - Locks: $0, 0!, 0?$ and $1, 1!, 1?$
  - Assumption: Thread one initially holds $0!, 1!$, thread two initially holds $0?, 1?$
- To produce a 0:
  - Thread 1 executes: $[_{0?}]_{0!}[_{0}]_{0?}[_{0!}]_{0}$
  - Thread 2 executes: $[_{0}]_{0?}[_{0!}]_{0}[_{0?}]_{0!}$
- The only possible execution of these two sequences is

Thread 1:			$[_{0?}$	$]_{0!}$			$[_{0}$	$]_{0?}$			$[_{0!}$	$]_{0}$
Thread 2:	$[_{0}$	$]_{0?}$			$[_{0!}$	$]_{0}$			$[_{0?}$	$]_{0!}$		

Undecidability

- Synchronizing two threads with locks
  - Locks: $0, 0!, 0?$ and $1, 1!, 1?$
  - Assumption: Thread one initially holds $0!, 1!$, thread two initially holds $0?, 1?$
- To produce a 0:
  - Thread 1 executes: $[_{0?}]_{0!}[_{0}]_{0?}[_{0!}]_{0}$
  - Thread 2 executes: $[_{0}]_{0?}[_{0!}]_{0}[_{0?}]_{0!}$
- The only possible execution of these two sequences is

$$\begin{array}{l|cccccccccccc}
\text{Thread 1:} & & & [_{0?} &]_{0!} & & & [_{0} &]_{0?} & & & [_{0!} &]_{0} \\
\text{Thread 2:} & [_{0} &]_{0?} & & & [_{0!} &]_{0} & & & [_{0?} &]_{0!} & &
\end{array}$$

  - And when Thread 2 has finished, it cannot re-enter the synchronization sequence until Thread 1 has also finished, and released 0.

Undecidability

- Synchronizing two threads with locks
  - Locks: $0, 0!, 0?$ and $1, 1!, 1?$
  - Assumption: Thread one initially holds $0!, 1!$, thread two initially holds $0?, 1?$
- To produce a 0:
  - Thread 1 executes: $[_{0?}\,]_{0!}\,[_{0}\,]_{0?}\,[_{0!}\,]_{0}$
  - Thread 2 executes: $[_{0}\,]_{0?}\,[_{0!}\,]_{0}\,[_{0?}\,]_{0!}$
- The only possible execution of these two sequences is

Thread 1:			$[_{0?}$	$]_{0!}$			$[_{0}$	$]_{0?}$			$[_{0!}$	$]_{0}$
Thread 2:	$[_{0}$	$]_{0?}$			$[_{0!}$	$]_{0}$			$[_{0?}$	$]_{0!}$		

  - And when Thread 2 has finished, it cannot re-enter the synchronization sequence until Thread 1 has also finished, and released 0.
- The sequences for producing 1 are analogously

Undecidability

- Remaining problem: Ensure that the locks are initially allocated, before the threads start the production of output symbols

Undecidability

- Remaining problem: Ensure that the locks are initially allocated, before the threads start the production of output symbols
- Solution: Additional locks l_1 and l_2

Undecidability

- Remaining problem: Ensure that the locks are initially allocated, before the threads start the production of output symbols
- Solution: Additional locks l_1 and l_2
  - Thread 1: $[_{0!}\,[_{1!}\,[_{l_1}\,]_{l_1}\,[_{l_2}$<start of output>

Undecidability

- Remaining problem: Ensure that the locks are initially allocated, before the threads start the production of output symbols
- Solution: Additional locks l_1 and l_2
  - Thread 1: $[_{0!}[_{1!}[_{l_1}]_{l_1}[_{l_2}$<start of output>
  - Thread 2: $[_{0?}[_{1?}[_{l_2}]_{l_2}[_{l_1}$<start of output>

Undecidability

- Remaining problem: Ensure that the locks are initially allocated, before the threads start the production of output symbols
- Solution: Additional locks l_1 and l_2
  - Thread 1: $[_{0!}[_{1!}[_{l_1}]_{l_1}[_{l_2}$<start of output>
  - Thread 2: $[_{0?}[_{1?}[_{l_2}]_{l_2}[_{l_1}$<start of output>
  - If one thread starts before the other has finished initialization, the other will be stuck at $[_{l_i}]_{l_i}$ forever

Undecidability

- Remaining problem: Ensure that the locks are initially allocated, before the threads start the production of output symbols
- Solution: Additional locks l_1 and l_2
  - Thread 1: $[_{0!}[_{1!}[_{l_1}]_{l_1}[_{l_2}$<start of output>
  - Thread 2: $[_{0?}[_{1?}[_{l_2}]_{l_2}[_{l_1}$<start of output>
  - If one thread starts before the other has finished initialization, the other will be stuck at $[_{l_i}]_{l_i}$ forever
- Thus, final states of PDSs simultaneously reachable, iff encoded CF-languages have non-empty intersection

Complexity for nested locks

- NP-Hardness

Complexity for nested locks

- NP-Hardness
    - Reachability analysis for nested locks and procedures is NP-hard

Complexity for nested locks

- NP-Hardness
  - Reachability analysis for nested locks and procedures is NP-hard
  - Problem: Deadlocks may prevent reachability

Complexity for nested locks

- NP-Hardness
  - Reachability analysis for nested locks and procedures is NP-hard
  - Problem: Deadlocks may prevent reachability
- Reduction to 3-SAT:

Complexity for nested locks

- NP-Hardness
  - Reachability analysis for nested locks and procedures is NP-hard
  - Problem: Deadlocks may prevent reachability
- Reduction to 3-SAT:
  - One lock per literal: Allocated — literal is false, Free — literal is true

Complexity for nested locks

- NP-Hardness
  - Reachability analysis for nested locks and procedures is NP-hard
  - Problem: Deadlocks may prevent reachability
- Reduction to 3-SAT:
  - One lock per literal: Allocated — literal is false, Free — literal is true
  - Use nested procedures and non-determinism to allocate locks according to configuration

Complexity for nested locks

- NP-Hardness
  - Reachability analysis for nested locks and procedures is NP-hard
  - Problem: Deadlocks may prevent reachability
- Reduction to 3-SAT:
  - One lock per literal: Allocated — literal is false, Free — literal is true
  - Use nested procedures and non-determinism to allocate locks according to configuration
  - Check for clause $l_1 \vee l_2 \vee l_3$: Nondeterministically run one of $[_{l_i};]_{l_i}$

Complexity for nested locks

- NP-Hardness
  - Reachability analysis for nested locks and procedures is NP-hard
  - Problem: Deadlocks may prevent reachability
- Reduction to 3-SAT:
  - One lock per literal: Allocated — literal is false, Free — literal is true
  - Use nested procedures and non-determinism to allocate locks according to configuration
  - Check for clause $l_1 \vee l_2 \vee l_3$: Nondeterministically run one of $[_{l_i};]_{l_i}$
  - Enforce correct order of guessing assignment and checking: One additional lock

Reduction to 3-SAT

- Reminder (3-SAT)

Reduction to 3-SAT

- Reminder (3-SAT)
  - Variables $x_0, \ldots, x_n$, *literal*: x_i or $\bar{x}_i$

Reduction to 3-SAT

- Reminder (3-SAT)
  - Variables $x_0, \ldots, x_n$, *literal*: x_i or $\bar{x}_i$
  - Formula $\Phi = \bigwedge_{i=1\ldots m} \bigvee_{j=1\ldots 3} l_{ij}$, where the l_{ij} are literals

Reduction to 3-SAT

- Reminder (3-SAT)
  - Variables $x_0, \ldots, x_n$, *literal*: x_i or $\bar{x}_i$
  - Formula $\Phi = \bigwedge_{i=1\ldots m} \bigvee_{j=1\ldots 3} l_{ij}$, where the l_{ij} are literals
    - $\bigvee_{j=1\ldots 3} l_{ij}$ is called *clause*

Reduction to 3-SAT

- Reminder (3-SAT)
  - Variables $x_0, \ldots, x_n$, *literal*: x_i or $\bar{x}_i$
  - Formula $\Phi = \bigwedge_{i=1\ldots m} \bigvee_{j=1\ldots 3} l_{ij}$, where the l_{ij} are literals
    - $\bigvee_{j=1\ldots 3} l_{ij}$ is called *clause*
  - It is NP-complete to decide whether Φ is *satisfiable*.

Reduction to 3-SAT

- Reminder (3-SAT)
  - Variables $x_0, \ldots, x_n$, *literal*: x_i or $\bar{x}_i$
  - Formula $\Phi = \bigwedge_{i=1\ldots m} \bigvee_{j=1\ldots 3} l_{ij}$, where the l_{ij} are literals
    - $\bigvee_{j=1\ldots 3} l_{ij}$ is called *clause*
  - It is NP-complete to decide whether Φ is *satisfiable*.
    - i.e. whether there is a valuation of the variables such that Φ holds.

Reduction to 3-SAT

```
ass(i):
  if ... then {
    acquire x_i  ass(i+1)  release x_i
  } else {
    acquire x̄_i  ass(i+1)  release x̄_i
  }
  return

ass(n+1):
  acquire(s); release(s);
  label1: return

thread1: ass(1)
```

```
check(i):
  if (...) {
    acquire l_i1 ; release l_i1 ;
  } else if (...) {
    acquire l_i2 ; release l_i2 ;
  } else {
    acquire l_i3 ; release l_i3 ;
  }

thread2:
  acquire(s);
  check(1); ...; check(m);
  label2: skip
  release(s)
```

- `label1` and `label2` simultaneously reachable, iff formula is satisfiable.

Last Lecture

- Execution trees of DPN
- Locks: Negative results
  - Reachability in DPN (even 2-PDS) wrt. arbitrary locking is undecidable
    - Reduction to deciding intersection of CF languages
  - Reachability in DPN (even 2-PDS) wrt. nested locking is NP-hard
    - Reduction to 3-SAT

Table of Contents

2-PDS with locks

- Two PDS with locks. Both share same rules.

2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_1^0\gamma_1^0, p_2^0\gamma_2^0, \Delta)$

2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_1^0\gamma_1^0, p_2^0\gamma_2^0, \Delta)$
    - P, Γ, Δ: States, stack alphabet, rules

2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_1^0\gamma_1^0, p_2^0\gamma_2^0, \Delta)$
    - P, Γ, Δ: States, stack alphabet, rules
    - $\mathrm{Act} = \mathrm{Act}_{nl} \mathbin{\dot\cup} \{[_x \mid x \in \mathbb{L}\} \mathbin{\dot\cup} \{]_x \mid x \in \mathbb{L}\}$

2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_1^0\gamma_1^0, p_2^0\gamma_2^0, \Delta)$
    - P, Γ, Δ: States, stack alphabet, rules
    - $\mathrm{Act} = \mathrm{Act}_{nl} \mathbin{\dot\cup} \{[_x \mid x \in \mathbb{L}\} \mathbin{\dot\cup} \{]_x \mid x \in \mathbb{L}\}$
    - $\mathbb{L}$: Finite set of locks

2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_1^0\gamma_1^0, p_2^0\gamma_2^0, \Delta)$
    - P, Γ, Δ: States, stack alphabet, rules
    - $\mathrm{Act} = \mathrm{Act}_{nl} \,\dot\cup\, \{[_x \mid x \in \mathbb{L}\} \,\dot\cup\, \{]_x \mid x \in \mathbb{L}\}$
    - $\mathbb{L}$: Finite set of locks
    - $p_1^0\gamma_1^0$, $p_2^0\gamma_2^0$: Initial states of left and right PDS

2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_1^0\gamma_1^0, p_2^0\gamma_2^0, \Delta)$
    - P, Γ, Δ: States, stack alphabet, rules
    - $\mathrm{Act} = \mathrm{Act}_{nl} \mathbin{\dot\cup} \{[_x \mid x \in \mathbb{L}\} \mathbin{\dot\cup} \{]_x \mid x \in \mathbb{L}\}$
    - $\mathbb{L}$: Finite set of locks
    - $p_1^0\gamma_1^0$, $p_2^0\gamma_2^0$: Initial states of left and right PDS
- Assumption: Locks are well-nested and non-reentrant

2-PDS with locks

- Two PDS with locks. Both share same rules.
  - $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_1^0\gamma_1^0, p_2^0\gamma_2^0, \Delta)$
    - P, Γ, Δ: States, stack alphabet, rules
    - $\mathrm{Act} = \mathrm{Act}_{nl} \,\dot\cup\, \{[_x \mid x \in \mathbb{L}\} \,\dot\cup\, \{]_x \mid x \in \mathbb{L}\}$
    - $\mathbb{L}$: Finite set of locks
    - $p_1^0\gamma_1^0$, $p_2^0\gamma_2^0$: Initial states of left and right PDS
- Assumption: Locks are well-nested and non-reentrant
  - In particular, thread does not free „foreign" locks

Semantics

- Configurations: $(p_1 w_1, p_2 w_2, L) \in P\Gamma^* \times P\Gamma^* \times 2^{\mathbb{L}}$

Semantics

- Configurations: $(p_1 w_1, p_2 w_2, L) \in P\Gamma^* \times P\Gamma^* \times 2^{\mathbb{L}}$
  - $cond([_x, L) = x \notin L$, $eff([_x, L) = L \cup \{x\}$

Semantics

- Configurations: $(p_1 w_1, p_2 w_2, L) \in P\Gamma^* \times P\Gamma^* \times 2^{\mathbb{L}}$
  - $\mathit{cond}([_x, L) = x \notin L$, $\mathit{eff}([_x, L) = L \cup \{x\}$
  - $\mathit{cond}(]_x, L) = \mathit{true}$, $\mathit{eff}(]_x, L) = L \setminus \{x\}$

Semantics

- Configurations: $(p_1 w_1, p_2 w_2, L) \in P\Gamma^* \times P\Gamma^* \times 2^{\mathbb{L}}$
  - $cond([_x, L) = x \notin L$, $eff([_x, L) = L \cup \{x\}$
  - $cond(]_x, L) = true$, $eff(]_x, L) = L \setminus \{x\}$
  - $cond(a, L) = true$, $eff(a, L) = L$ for $a \in \mathrm{Act}_{nl}$

Semantics

- Configurations: $(p_1 w_1, p_2 w_2, L) \in P\Gamma^* \times P\Gamma^* \times 2^{\mathbb{L}}$
  - $cond([_x, L) = x \notin L$, $eff([_x, L) = L \cup \{x\}$
  - $cond(]_x, L) = true$, $eff(]_x, L) = L \setminus \{x\}$
  - $cond(a, L) = true$, $eff(a, L) = L$ for $a \in \mathrm{Act}_{nl}$
- Step

$$(p\gamma w_1, p_2 w_2, L) \xrightarrow{a}_{\mathrm{ls}} (p' w' w_1, p_2 w_2, eff(a, L)) \quad \text{if } p\gamma \overset{a}{\hookrightarrow} p' w' \in \Delta \text{ and } cond(a, L) \qquad \text{(left)}$$

$$(p_1 w_1, p\gamma w_2, L) \xrightarrow{a}_{\mathrm{ls}} (p_1 w_1, p' w' w_2, eff(a, L)) \quad \text{if } p\gamma \overset{a}{\hookrightarrow} p' w' \in \Delta \text{ and } cond(a, L) \qquad \text{(right)}$$

Lock sensitive scheduling

- Idea: Abstraction from PDS

Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved

Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved
- Configurations: $(l_1, l_2, L) \in \mathrm{Act}^* \times \mathrm{Act}^* \times 2^{\mathbb{L}}$

Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved
- Configurations: $(l_1, l_2, L) \in \mathrm{Act}^* \times \mathrm{Act}^* \times 2^{\mathbb{L}}$
- Step

$$
\begin{aligned}
(al_1, l_2, L) &\overset{a}{\hookrightarrow} (l_1, l_2, \mathit{eff}(a, L)) && \text{if } \mathit{cond}(a, L) && \text{(left)} \\
(l_1, al_2, L) &\overset{a}{\hookrightarrow} (l_1, l_2, \mathit{eff}(a, L)) && \text{if } \mathit{cond}(a, L) && \text{(right)}
\end{aligned}
$$

Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved
- Configurations: $(l_1, l_2, L) \in \mathrm{Act}^* \times \mathrm{Act}^* \times 2^{\mathbb{L}}$
- Step

$$
\begin{aligned}
&(al_1, l_2, L) \xhookrightarrow{a} (l_1, l_2, \mathit{eff}(a, L)) && \text{if } \mathit{cond}(a, L) && \text{(left)}\\
&(l_1, al_2, L) \xhookrightarrow{a} (l_1, l_2, \mathit{eff}(a, L)) && \text{if } \mathit{cond}(a, L) && \text{(right)}
\end{aligned}
$$

- Lemma

$$
\begin{aligned}
&(p_1 w_1, p_2 w_2, L) \xrightarrow{l}{}^{*} (p_1' w_1', p_2' w_2', L')\\
&\qquad \text{iff } \exists l_1, l_2.\ p_1 w_1 \xrightarrow{l_1}{}^{*} p_1' w_1' \wedge p_2 w_2 \xrightarrow{l_2}{}^{*} p_2' w_2' \wedge (l_1, l_2, L) \xrightarrow{l}{}^{*} (\varepsilon, \varepsilon, L')
\end{aligned}
$$

Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved
- Configurations: $(l_1, l_2, L) \in \mathrm{Act}^* \times \mathrm{Act}^* \times 2^{\mathbb{L}}$
- Step

$$(al_1, l_2, L) \xhookrightarrow{a} (l_1, l_2, \mathit{eff}(a, L)) \qquad \text{if } \mathit{cond}(a, L) \qquad \text{(left)}$$

$$(l_1, al_2, L) \xhookrightarrow{a} (l_1, l_2, \mathit{eff}(a, L)) \qquad \text{if } \mathit{cond}(a, L) \qquad \text{(right)}$$

- Lemma

$$(p_1 w_1, p_2 w_2, L) \xrightarrow{l}{}^* (p_1' w_1', p_2' w_2', L')$$
$$\text{iff } \exists l_1, l_2.\ p_1 w_1 \xrightarrow{l_1}{}^* p_1' w_1' \wedge p_2 w_2 \xrightarrow{l_2}{}^* p_2' w_2' \wedge (l_1, l_2, L) \xhookrightarrow{l}{}^* (\varepsilon, \varepsilon, L')$$

  - Intuition: Schedule lock-insensitive executions of the single PDSs

Lock sensitive scheduling

- Idea: Abstraction from PDS
  - Check whether two execution sequences can be interleaved
- Configurations: $(l_1, l_2, L) \in \mathrm{Act}^* \times \mathrm{Act}^* \times 2^{\mathbb{L}}$
- Step

$$(al_1, l_2, L) \overset{a}{\hookrightarrow} (l_1, l_2, \mathit{eff}(a, L)) \quad \text{if } \mathit{cond}(a, L) \quad \text{(left)}$$

$$(l_1, al_2, L) \overset{a}{\hookrightarrow} (l_1, l_2, \mathit{eff}(a, L)) \quad \text{if } \mathit{cond}(a, L) \quad \text{(right)}$$

- Lemma

$$(p_1 w_1, p_2 w_2, L) \xrightarrow{l}{}^* (p_1' w_1', p_2' w_2', L')$$
$$\text{iff } \exists l_1, l_2.\ p_1 w_1 \xrightarrow{l_1}{}^* p_1' w_1' \wedge p_2 w_2 \xrightarrow{l_2}{}^* p_2' w_2' \wedge (l_1, l_2, L) \overset{l}{\hookrightarrow}{}^* (\varepsilon, \varepsilon, L')$$

  - Intuition: Schedule lock-insensitive executions of the single PDSs
  - Proof: Straightforward simulation proof

Execution trees of 2-PDS

- Intuitively: Append execution trees of left and right PDS to binary root node $\circ$.

Execution trees of 2-PDS

- Intuitively: Append execution trees of left and right PDS to binary root node $\circ$.
  - $X2 ::= \circ(XN, XN)$

Execution trees of 2-PDS

- Intuitively: Append execution trees of left and right PDS to binary root node $\circ$.
  - $X2 ::= \circ(XN, XN)$
- Tree automata: Tree automata for PDS execution trees, but

Execution trees of 2-PDS

- Intuitively: Append execution trees of left and right PDS to binary root node $\circ$.
  - $X2 ::= \circ(XN, XN)$
- Tree automata: Tree automata for PDS execution trees, but
  - Initial state i, and additional rule $i \to \circ(p_1^0\gamma_1^0, p_2^0\gamma_2^0)$

Execution trees of 2-PDS

- Intuitively: Append execution trees of left and right PDS to binary root node $\circ$.
  - $X2 ::= \circ(XN, XN)$
- Tree automata: Tree automata for PDS execution trees, but
  - Initial state i, and additional rule $i \to \circ(p_1^0\gamma_1^0, p_2^0\gamma_2^0)$
- We have (with lemma from previous slide)

$$(p_1 w_1, p_2 w_2, L) \xrightarrow{l}{}^{*} (p_1' w_1', p_2' w_2', L')$$
$$\text{iff } \exists t_1, t_2.\ i \to \circ(t_1, t_2) \land c(t_1) = p_1' w_1' \land c(t_2) = p_2' w_2'$$
$$\land\ (a(t_1), a(t_2), L) \xrightarrow{l}{}^{*} (\varepsilon, \varepsilon, L')$$

Execution trees of 2-PDS

- Intuitively: Append execution trees of left and right PDS to binary root node $\circ$.
  - $X2 ::= \circ(XN, XN)$
- Tree automata: Tree automata for PDS execution trees, but
  - Initial state i, and additional rule $i \to \circ(p_1^0\gamma_1^0, p_2^0\gamma_2^0)$
- We have (with lemma from previous slide)

$$
\begin{aligned}
&(p_1 w_1, p_2 w_2, L) \xrightarrow{l}{}^* (p_1' w_1', p_2' w_2', L') \\
&\text{iff } \exists t_1, t_2.\ i \to \circ(t_1, t_2) \wedge c(t_1) = p_1' w_1' \wedge c(t_2) = p_2' w_2' \\
&\qquad\qquad\qquad\qquad \wedge (a(t_1), a(t_2), L) \xrightarrow{l}{}^* (\varepsilon, \varepsilon, L')
\end{aligned}
$$

  - Where $c : XN \to \mathit{conf}$ extracts reached configuration from execution tree and $a : XN \to \mathrm{Act}^*$ extracts labeling sequence from execution tree (cf. Homework 9.2)

Attack Plan

- Compute information $ah(l_1), ah(l_2)$ which

Attack Plan

- Compute information $ah(l_1), ah(l_2)$ which
  - Can be used to decide whether $(l_1, l_2, \emptyset) \rightarrow^* (\varepsilon, \varepsilon, \_)$

Attack Plan

- Compute information $ah(l_1), ah(l_2)$ which
  - Can be used to decide whether $(l_1, l_2, \emptyset) \rightarrow^* (\varepsilon, \varepsilon, \_)$
  - Sets of which can be computed by tree automaton over execution trees

Attack Plan

- Compute information $ah(l_1), ah(l_2)$ which
  - Can be used to decide whether $(l_1, l_2, \emptyset) \to^* (\varepsilon, \varepsilon, \_)$
  - Sets of which can be computed by tree automaton over execution trees
- Thus, we get a tree automaton for schedulable execution trees.

Attack Plan

- Compute information $ah(l_1), ah(l_2)$ which
  - Can be used to decide whether $(l_1, l_2, \emptyset) \rightarrow^* (\varepsilon, \varepsilon, \_)$
  - Sets of which can be computed by tree automaton over execution trees
- Thus, we get a tree automaton for schedulable execution trees.
- Checking the intersection of this, the tree automaton for execution trees, and the error property for emptiness gives us lock-sensitive model-checker

Acquisition Histories: Intuition

- Categorize an action $[_x$ in an execution sequence as

Acquisition Histories: Intuition

- Categorize an action $[_x$ in an execution sequence as

  Final acquisition If lock x is not released afterwards

Acquisition Histories: Intuition

- Categorize an action $[_x$ in an execution sequence as

  Final acquisition If lock x is not released afterwards

  Usage If lock l is released afterwards

Acquisition Histories: Intuition

- Categorize an action $[_x$ in an execution sequence as

  Final acquisition If lock x is not released afterwards

  Usage If lock l is released afterwards
- When can two sequences l_1 and l_2 be scheduled?

Acquisition Histories: Intuition

- Categorize an action $[_x$ in an execution sequence as

  Final acquisition If lock x is not released afterwards

  Usage If lock l is released afterwards
- When can two sequences l_1 and l_2 be scheduled?
  - No lock is finally acquired in both, l_1 and l_2

Acquisition Histories: Intuition

- Categorize an action $[_x$ in an execution sequence as

  Final acquisition If lock x is not released afterwards

  Usage If lock l is released afterwards
- When can two sequences l_1 and l_2 be scheduled?
  - No lock is finally acquired in both, l_1 and l_2
  - There must be no deadlock pair

Acquisition Histories: Intuition

- Categorize an action $[_x$ in an execution sequence as

  Final acquisition: If lock x is not released afterwards

  Usage: If lock l is released afterwards
- When can two sequences l_1 and l_2 be scheduled?
  - No lock is finally acquired in both, l_1 and l_2
  - There must be no deadlock pair
    - I.e., l_1 finally acquires x_1 and then uses x_2, and l_2 finally acquires x_2 and then uses x_1

Acquisition Histories: Intuition

- Categorize an action $[_x$ in an execution sequence as

  Final acquisition If lock x is not released afterwards

  Usage If lock l is released afterwards
- When can two sequences l_1 and l_2 be scheduled?
  - No lock is finally acquired in both, l_1 and l_2
  - There must be no deadlock pair
    - I.e., l_1 finally acquires x_1 and then uses x_2, and l_2 finally acquires x_2 and then uses x_1
- We will now prove: This characterization is sufficient and necessary

Acquisition Histories: Intuition

- Categorize an action $[_x$ in an execution sequence as

  Final acquisition If lock x is not released afterwards

  Usage If lock l is released afterwards
- When can two sequences l_1 and l_2 be scheduled?
  - No lock is finally acquired in both, l_1 and l_2
  - There must be no deadlock pair
    - I.e., l_1 finally acquires x_1 and then uses x_2, and l_2 finally acquires x_2 and then uses x_1
- We will now prove: This characterization is sufficient and necessary
  - And can be computed for the sets of all executions by tree automata

Acquisition Histories: Definition

- Given an execution sequence $l \in \mathrm{Act}^*$, we define $ah(l) := (A(l), G(l))$ where

Acquisition Histories: Definition

- Given an execution sequence $l \in \mathrm{Act}^*$, we define $ah(l) := (A(l), G(l))$ where
  - $A(l) \subseteq \mathbb{L}$ is the set of finally acquired locks:

$$
\begin{aligned}
A(\varepsilon) &= \emptyset \\
A(al) &= A(l) && \text{if } a \in \mathrm{Act}_{nl} \text{ or } a =]_x \text{ for } x \in \mathbb{L} \\
A([_x l) &= A(l) && \text{if }]_x \in l \\
A([_x l) &= A(l) \cup \{x\} && \text{if }]_x \notin l
\end{aligned}
$$

Acquisition Histories: Definition

- Given an execution sequence $l \in \mathrm{Act}^*$, we define $ah(l) := (A(l), G(l))$ where
  - $A(l) \subseteq \mathbb{L}$ is the set of finally acquired locks:

$$
\begin{aligned}
A(\varepsilon) &= \emptyset \\
A(al) &= A(l) && \text{if } a \in \mathrm{Act}_{nl} \text{ or } a =]_x \text{ for } x \in \mathbb{L} \\
A([_x l) &= A(l) && \text{if }]_x \in l \\
A([_x l) &= A(l) \cup \{x\} && \text{if }]_x \notin l
\end{aligned}
$$

  - $G(l) \subseteq \mathbb{L} \times \mathbb{L}$ is the lock graph:

$$
\begin{aligned}
G(\varepsilon) &= \emptyset \\
G(al) &= G(l) && \text{if } a \in \mathrm{Act}_{nl} \text{ or } a =]_x \text{ for } x \in \mathbb{L} \\
G([_x l) &= G(l) && \text{if }]_x \in l \\
G([_x l) &= G(l) \cup \{x\} \times \mathrm{acq}(l) && \text{if }]_x \notin l
\end{aligned}
$$

  where $\mathrm{acq}(l) := \{x \mid [_x \in l\}$

Acquisition Histories: Definition

- Given an execution sequence $l \in \mathrm{Act}^*$, we define $ah(l) := (A(l), G(l))$ where
  - $A(l) \subseteq \mathbb{L}$ is the set of finally acquired locks:

$$
\begin{aligned}
A(\varepsilon) &= \emptyset \\
A(al) &= A(l) && \text{if } a \in \mathrm{Act}_{nl} \text{ or } a =]_x \text{ for } x \in \mathbb{L} \\
A([_x l) &= A(l) && \text{if }]_x \in l \\
A([_x l) &= A(l) \cup \{x\} && \text{if }]_x \notin l
\end{aligned}
$$

  - $G(l) \subseteq \mathbb{L} \times \mathbb{L}$ is the lock graph:

$$
\begin{aligned}
G(\varepsilon) &= \emptyset \\
G(al) &= G(l) && \text{if } a \in \mathrm{Act}_{nl} \text{ or } a =]_x \text{ for } x \in \mathbb{L} \\
G([_x l) &= G(l) && \text{if }]_x \in l \\
G([_x l) &= G(l) \cup \{x\} \times \mathrm{acq}(l) && \text{if }]_x \notin l
\end{aligned}
$$

  where $\mathrm{acq}(l) := \{x \mid [_x \in l\}$
- Lemma

$$
(l_1, l_2, \emptyset) \rightarrow^* (\varepsilon, \varepsilon, \_) \text{ iff } A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2))
$$

Proof ideas

- $\Longrightarrow$

Proof ideas

- $\Longrightarrow$
  - Generalize to

$$\forall L.\ (l_1, l_2, L) \to^* (\varepsilon, \varepsilon, \_) \implies A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2))$$

Proof ideas

- $\Longrightarrow$
  - Generalize to

$$\forall L.\ (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, \_) \implies A(l_1) \cap A(l_2) = \emptyset \land \mathrm{acyclic}(G(l_1) \cup G(l_2))$$

  - Induction on $\rightarrow^*$

Proof ideas

- $\Longrightarrow$
  - Generalize to

    $$\forall L.\ (l_1, l_2, L) \to^* (\varepsilon, \varepsilon, \_) \implies A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2))$$

  - Induction on $\to^*$
    - Interesting case: First step is final acquisition: $[_x$

Proof ideas

- $\Longrightarrow$
  - Generalize to

$$\forall L.\ (l_1, l_2, L) \to^* (\varepsilon, \varepsilon, \_) \implies A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2))$$

  - Induction on $\to^*$
    - Interesting case: First step is final acquisition: $[_x$
    - $[_x$ will not occur in remaining execution

Proof ideas

- $\Longrightarrow$
  - Generalize to

$$\forall L.\ (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, \_) \implies A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2))$$

  - Induction on $\rightarrow^*$
    - Interesting case: First step is final acquisition: $[_x$
    - $[_x$ will not occur in remaining execution
    - Thus, it cannot close a cycle in the lock graphs

Proof ideas

- $\Longrightarrow$
  - Generalize to

$$\forall L.\ (l_1, l_2, L) \to^* (\varepsilon, \varepsilon, \_) \implies A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2))$$

  - Induction on $\to^*$
    - Interesting case: First step is final acquisition: $[_x$
    - $[_x$ will not occur in remaining execution
    - Thus, it cannot close a cycle in the lock graphs
- $\Longleftarrow$

Proof ideas

- $\implies$
  - Generalize to

$$\forall L.\ (l_1, l_2, L) \to^* (\varepsilon, \varepsilon, \_) \implies A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2))$$

  - Induction on $\to^*$
    - Interesting case: First step is final acquisition: $[_x$
    - $[_x$ will not occur in remaining execution
    - Thus, it cannot close a cycle in the lock graphs
- $\impliedby$
  - Generalize to

$$\begin{aligned} &A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2)) \\ &\quad \implies \forall L.\ L \cap (\mathrm{acq}(l_1) \cup \mathrm{acq}(l_2)) = \emptyset \implies (l_1, l_2, L) \to^* (\varepsilon, \varepsilon, \_) \end{aligned} \tag{1}$$

Proof ideas

- $\Longrightarrow$
  - Generalize to

$$\forall L.\ (l_1, l_2, L) \to^* (\varepsilon, \varepsilon, \_) \implies A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2))$$

  - Induction on $\to^*$
    - Interesting case: First step is final acquisition: $[_x$
    - $[_x$ will not occur in remaining execution
    - Thus, it cannot close a cycle in the lock graphs
- $\Longleftarrow$
  - Generalize to

$$\begin{aligned} & A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2)) \\ & \quad \implies \forall L.\ L \cap (\mathrm{acq}(l_1) \cup \mathrm{acq}(l_2)) = \emptyset \implies (l_1, l_2, L) \to^* (\varepsilon, \varepsilon, \_) \end{aligned} \tag{1}$$

  - Induction on $|l_1| + |l_2|$

Proof ideas

- $\implies$
  - Generalize to

$$\forall L.\ (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, \_) \implies A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2))$$

  - Induction on $\rightarrow^*$
    - Interesting case: First step is final acquisition: $[_x$
    - $[_x$ will not occur in remaining execution
    - Thus, it cannot close a cycle in the lock graphs
- $\impliedby$
  - Generalize to

$$\begin{aligned} &A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2)) \\ &\qquad \implies \forall L.\ L \cap (\mathrm{acq}(l_1) \cup \mathrm{acq}(l_2)) = \emptyset \implies (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, \_) \end{aligned} \tag{1}$$

  - Induction on $|l_1| + |l_2|$
    - Schedule usages of locks first

Proof ideas

- $\Longrightarrow$
  - Generalize to

$$\forall L.\ (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, \_) \implies A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2))$$

  - Induction on $\rightarrow^*$
    - Interesting case: First step is final acquisition: $[_x$
    - $[_x$ will not occur in remaining execution
    - Thus, it cannot close a cycle in the lock graphs
- $\Longleftarrow$
  - Generalize to

$$\begin{aligned} &A(l_1) \cap A(l_2) = \emptyset \wedge \mathrm{acyclic}(G(l_1) \cup G(l_2)) \\ &\quad \implies \forall L.\ L \cap (\mathrm{acq}(l_1) \cup \mathrm{acq}(l_2)) = \emptyset \implies (l_1, l_2, L) \rightarrow^* (\varepsilon, \varepsilon, \_) \end{aligned} \tag{1}$$

  - Induction on $|l_1| + |l_2|$
    - Schedule usages of locks first
    - If both, l_1 and l_2 start with final acquisitions:
      Choose acquisition that comes first in topological ordering of $G(l_1) \cup G(l_2)$

Computation of acquisition histories

- There are only finitely many acquisition histories

Computation of acquisition histories

- There are only finitely many acquisition histories
  - Exponentially many in number of locks

Computation of acquisition histories

- There are only finitely many acquisition histories
    - Exponentially many in number of locks
- Set of all schedulable 2-PDS execution trees is regular

Computation of acquisition histories

- There are only finitely many acquisition histories
  - Exponentially many in number of locks
- Set of all schedulable 2-PDS execution trees is regular
- In practice: Avoid computing unnecessary states of tree automata

Last Lecture

- 2-PDS with locks
- Acquisition histories
- Deciding lock-sensitive reachability

Table of Contents

DPNs with locks

- Same ideas as for 2-PDS

DPNs with locks

- Same ideas as for 2-PDS
- $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_0\gamma_0, \Delta)$

DPNs with locks

- Same ideas as for 2-PDS
- $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_0\gamma_0, \Delta)$
  - P, Γ, Δ: States, stack alphabet, rules (with spawns)

DPNs with locks

- Same ideas as for 2-PDS
- $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_0\gamma_0, \Delta)$
  - P, Γ, Δ: States, stack alphabet, rules (with spawns)
  - $\mathrm{Act} = \mathrm{Act}_{nl} \mathbin{\dot\cup} \{[_x \mid x \in \mathbb{L}\} \mathbin{\dot\cup} \{]_x \mid x \in \mathbb{L}\}$

DPNs with locks

- Same ideas as for 2-PDS
- $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_0\gamma_0, \Delta)$
  - P, Γ, Δ: States, stack alphabet, rules (with spawns)
  - $\mathrm{Act} = \mathrm{Act}_{nl} \mathbin{\dot\cup} \{[_x \mid x \in \mathbb{L}\} \mathbin{\dot\cup} \{]_x \mid x \in \mathbb{L}\}$
  - $\mathbb{L}$: Finite set of locks

DPNs with locks

- Same ideas as for 2-PDS
- $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_0\gamma_0, \Delta)$
  - P, Γ, Δ: States, stack alphabet, rules (with spawns)
  - $\mathrm{Act} = \mathrm{Act}_{nl} \mathbin{\dot\cup} \{[_x \mid x \in \mathbb{L}\} \mathbin{\dot\cup} \{]_x \mid x \in \mathbb{L}\}$
  - $\mathbb{L}$: Finite set of locks
  - $p_0\gamma_0$: Initial state

DPNs with locks

- Same ideas as for 2-PDS
- $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_0\gamma_0, \Delta)$
  - P, Γ, Δ: States, stack alphabet, rules (with spawns)
  - $\mathrm{Act} = \mathrm{Act}_{nl} \,\dot\cup\, \{[_x \mid x \in \mathbb{L}\} \,\dot\cup\, \{]_x \mid x \in \mathbb{L}\}$
  - $\mathbb{L}$: Finite set of locks
  - $p_0\gamma_0$: Initial state
- Assumption: Locks are well-nested and non-reentrant

DPNs with locks

- Same ideas as for 2-PDS
- $M = (P, \Gamma, \mathrm{Act}, \mathbb{L}, p_0\gamma_0, \Delta)$
    - P, Γ, Δ: States, stack alphabet, rules (with spawns)
    - $\mathrm{Act} = \mathrm{Act}_{nl} \,\dot\cup\, \{[_x \mid x \in \mathbb{L}\} \,\dot\cup\, \{]_x \mid x \in \mathbb{L}\}$
    - $\mathbb{L}$: Finite set of locks
    - $p_0\gamma_0$: Initial state
- Assumption: Locks are well-nested and non-reentrant
    - In particular, thread does not free „foreign" locks

Semantics

- As for 2-PDS: Add set of locks

Semantics

- As for 2-PDS: Add set of locks
  - Recall: $\text{conf} ::= \langle \mathit{pw} \rangle(\text{conflist}) \quad \text{conflist} ::= \mathit{Nil} | \mathit{Cons}(\text{conf}, \text{conflist})$

Semantics

- As for 2-PDS: Add set of locks
    - Recall: $\text{conf} ::= \langle \mathit{pw} \rangle(\text{conflist}) \quad \text{conflist} ::= \mathit{Nil} | \mathit{Cons}(\text{conf}, \text{conflist})$
    - $\text{conf}_{\text{ls}} := \text{conf} \times \mathbb{L}$

Semantics

- As for 2-PDS: Add set of locks
  - Recall: $\mathrm{conf} ::= \langle pw \rangle(\mathrm{conflist}) \quad \mathrm{conflist} ::= \mathit{Nil} | \mathit{Cons}(\mathrm{conf}, \mathrm{conflist})$
  - $\mathrm{conf}_{\mathrm{ls}} := \mathrm{conf} \times \mathbb{L}$
- Step relation:

$$(c, L) \xrightarrow{a} (c', \mathit{eff}(a, L)) \text{ iff } \mathit{cond}(a, L) \wedge c \xrightarrow{a} c'$$

Lock-Sensitive Scheduling

- Abstract from DPN-configurations

Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

$$BL ::= \mathit{Nil} \mid \mathit{Cons}(a, BL) \mid \mathit{Spawn}(a, BL, BL) \quad \text{for all } a \in \mathrm{Act}$$
$$ST ::= \langle BL \rangle (SL) \quad SL ::= \mathit{Nil} \mid \mathit{Cons}(ST, SL)$$

Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

$$\begin{aligned} \mathit{BL} &::= \mathit{Nil} \mid \mathit{Cons}(a, \mathit{BL}) \mid \mathit{Spawn}(a, \mathit{BL}, \mathit{BL}) \quad \text{for all } a \in \mathrm{Act} \\ \mathit{ST} &::= \langle \mathit{BL} \rangle (\mathit{SL}) \quad \mathit{SL} ::= \mathit{Nil} \mid \mathit{Cons}(\mathit{ST}, \mathit{SL}) \end{aligned}$$

  - Combination of configurations and sequences of actions to be executed

Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

$$\mathit{BL} ::= \mathit{Nil} \mid \mathit{Cons}(a, \mathit{BL}) \mid \mathit{Spawn}(a, \mathit{BL}, \mathit{BL}) \quad \text{for all } a \in \mathrm{Act}$$
$$\mathit{ST} ::= \langle \mathit{BL} \rangle (\mathit{SL}) \quad \mathit{SL} ::= \mathit{Nil} \mid \mathit{Cons}(\mathit{ST}, \mathit{SL})$$

  - Combination of configurations and sequences of actions to be executed
  - Each thread in configuration is labeled by actions it still has to execute

Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

$$\mathit{BL} ::= \mathit{Nil} \mid \mathit{Cons}(a, \mathit{BL}) \mid \mathit{Spawn}(a, \mathit{BL}, \mathit{BL}) \quad \text{for all } a \in \mathrm{Act}$$
$$\mathit{ST} ::= \langle \mathit{BL} \rangle (\mathit{SL}) \quad \mathit{SL} ::= \mathit{Nil} \mid \mathit{Cons}(\mathit{ST}, \mathit{SL})$$

  - Combination of configurations and sequences of actions to be executed
  - Each thread in configuration is labeled by actions it still has to execute
  - Spawn actions have two successors: Actions of spawning thread and actions of spawned thread

Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

$$BL ::= Nil \mid Cons(a, BL) \mid Spawn(a, BL, BL) \quad \text{for all } a \in \mathrm{Act}$$
$$ST ::= \langle BL \rangle (SL) \quad SL ::= Nil \mid Cons(ST, SL)$$

  - Combination of configurations and sequences of actions to be executed
  - Each thread in configuration is labeled by actions it still has to execute
  - Spawn actions have two successors: Actions of spawning thread and actions of spawned thread
- Scheduler semantics

$$(C[\langle Cons(a, l)\rangle(s)], L) \xrightarrow{a} (C[\langle l\rangle(s)], eff(a, L)) \text{ iff } cond(a, L) \qquad \text{(no-spawn)}$$

$$(C[\langle Spawn(a, l_1, l_2)\rangle(s)], L) \xrightarrow{a} (C[\langle l_1\rangle(s[\langle l_2\rangle(Nil)])], eff(a, L)) \text{ iff } cond(a, L) \qquad \text{(spawn)}$$

where C is a context with exactly one occurrence of x_1.

Lock-Sensitive Scheduling

- Abstract from DPN-configurations
- Scheduling tree:

$$BL ::= Nil \mid Cons(a, BL) \mid Spawn(a, BL, BL) \quad \text{for all } a \in \mathrm{Act}$$
$$ST ::= \langle BL \rangle (SL) \qquad SL ::= Nil \mid Cons(ST, SL)$$

  - Combination of configurations and sequences of actions to be executed
  - Each thread in configuration is labeled by actions it still has to execute
  - Spawn actions have two successors: Actions of spawning thread and actions of spawned thread
- Scheduler semantics

$$(C[\langle Cons(a, l) \rangle (s)], L) \xrightarrow{a} (C[\langle l \rangle (s)], eff(a, L)) \text{ iff } cond(a, L) \qquad \text{(no-spawn)}$$

$$(C[\langle Spawn(a, l_1, l_2) \rangle (s)], L) \xrightarrow{a} (C[\langle l_1 \rangle (s[\langle l_2 \rangle (Nil)])], eff(a, L)) \text{ iff } cond(a, L) \qquad \text{(spawn)}$$

where C is a context with exactly one occurrence of x_1.

- Terminated scheduling tree: All steps are executed, i.e., all nodes labeled with *Nil*

$$ST_{term} ::= \langle Nil \rangle (SL_{term}) \qquad SL_{term} ::= Nil \mid Cons(ST_{term}, SL_{term})$$

Operations on Branching Lists

- Generalized concatenation

$$
\begin{aligned}
(\mathit{Nil})l' &:= l' \\
\mathit{Cons}(a, l)l' &:= \mathit{Cons}(a, ll') \\
\mathit{Spawn}(a, l_1, l_2)l' &:= \mathit{Spawn}(a, l_1 l', l_2)
\end{aligned}
$$

Operations on Branching Lists

- Generalized concatenation

$$\begin{aligned}(Nil)l' &:= l'\\ Cons(a,l)l' &:= Cons(a,ll')\\ Spawn(a,l_1,l_2)l' &:= Spawn(a,l_1l',l_2)\end{aligned}$$

- This thread's steps: $this : BL \to \mathrm{Act}^*$

$$\begin{aligned}this(Nil) &:= Nil\\ this(Cons(a,l)) &:= Cons(a,this(l))\\ this(Spawn(a,l_1,l_2)) &= Cons(a,this(l_1))\end{aligned}$$

Operations on Branching Lists

- Generalized concatenation

$$
\begin{aligned}
(\mathit{Nil})l' &:= l' \\
\mathit{Cons}(a, l)l' &:= \mathit{Cons}(a, ll') \\
\mathit{Spawn}(a, l_1, l_2)l' &:= \mathit{Spawn}(a, l_1 l', l_2)
\end{aligned}
$$

- This thread's steps: $\mathit{this} : \mathit{BL} \to \mathrm{Act}^*$

$$
\begin{aligned}
\mathit{this}(\mathit{Nil}) &:= \mathit{Nil} \\
\mathit{this}(\mathit{Cons}(a, l)) &:= \mathit{Cons}(a, \mathit{this}(l)) \\
\mathit{this}(\mathit{Spawn}(a, l_1, l_2)) &= \mathit{Cons}(a, \mathit{this}(l_1))
\end{aligned}
$$

- Set of steps

$$
\begin{aligned}
x \in \mathit{Nil} &:= \mathit{false} \\
x \in \mathit{Cons}(a, l) &:= x = a \vee x \in l \\
x \in \mathit{Spawn}(a, l_1, l_2) &:= x = a \vee x \in l_1 \vee x \in l_2
\end{aligned}
$$

Relation of execution tree and scheduling tree

- Execution trees correspond to scheduling trees: $st : XN \rightarrow ST$ and $st' : XN \rightarrow BL$ where

$$
\begin{aligned}
st(t) &:= \langle st'(t) \rangle (Nil) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma' \rangle(t)) &:= Cons(a, st'(t)) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2 \rangle(t_1, t_2)) &:= Spawn(a, st'(t_1), st'(t_2)) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^N(t)) &:= Cons(a, st'(t)) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2 \rangle^R(t_1, t_2)) &:= [a] st'(t_1) st'(t_2) \\
st'(\langle p\gamma \rangle) &:= Nil \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p' \rangle) &:= Cons(a, Nil)
\end{aligned}
$$

Relation of execution tree and scheduling tree

- Execution trees correspond to scheduling trees: $st : XN \to ST$ and $st' : XN \to BL$ where

$$
\begin{aligned}
st(t) &:= \langle st'(t)\rangle(Nil) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma'\rangle(t)) &:= Cons(a, st'(t)) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2\rangle(t_1, t_2)) &:= Spawn(a, st'(t_1), st'(t_2)) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2\rangle^N(t)) &:= Cons(a, st'(t)) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2\rangle^R(t_1, t_2)) &:= [a]st'(t_1)st'(t_2) \\
st'(\langle p\gamma\rangle) &:= Nil \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\rangle) &:= Cons(a, Nil)
\end{aligned}
$$

- It can be proved

$$
\begin{aligned}
&(\langle p_0\gamma_0\rangle(\varepsilon), \emptyset) \xrightarrow{l}{}^{*} (c', L) \\
&\iff \exists t \in XN.\, \exists t' \in ST_{term}.\ t \in L(\mathcal{A}_M) \wedge c(t) = c' \wedge (st(t), \emptyset) \xrightarrow{l}{}^{*} (t', L)
\end{aligned}
$$

Relation of execution tree and scheduling tree

- Execution trees correspond to scheduling trees: $st : XN \to ST$ and $st' : XN \to BL$ where

$$
\begin{aligned}
st(t) &:= \langle st'(t)\rangle(Nil) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma'\rangle(t)) &:= Cons(a, st'(t)) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p_1\gamma_1 \rhd p_2\gamma_2\rangle(t_1, t_2)) &:= Spawn(a, st'(t_1), st'(t_2)) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2\rangle^N(t)) &:= Cons(a, st'(t)) \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\gamma_1\gamma_2\rangle^R(t_1, t_2)) &:= [a]st'(t_1)st'(t_2) \\
st'(\langle p\gamma\rangle) &:= Nil \\
st'(\langle p\gamma \overset{a}{\hookrightarrow} p'\rangle) &:= Cons(a, Nil)
\end{aligned}
$$

- It can be proved

$$
\begin{aligned}
&(\langle p_0\gamma_0\rangle(\varepsilon), \emptyset) \xrightarrow{l}^{*} (c', L) \\
&\iff \exists t \in XN.\, \exists t' \in ST_{term}.\, t \in L(\mathcal{A}_M) \wedge c(t) = c' \wedge (st(t), \emptyset) \xrightarrow{l}^{*} (t', L)
\end{aligned}
$$

  - Note: This proof requires a generalization from a single-thread start configuration to arbitrary start configurations.

Acquisition Histories for Scheduling Trees

- Assumption: Acquisition and release only on base rules

Acquisition Histories for Scheduling Trees

- Assumption: Acquisition and release only on base rules
- Compute set of final acquisitions

$$
\begin{aligned}
A(Nil) &= \emptyset \\
A(Spawn(a, l_1, l_2)) &= A(l_1) \cup A(l_2) \\
A(Cons(a, l)) &= A(l) && \text{if } a \in \mathrm{Act}_{nl} \text{ or } a =]_x \text{ for } x \in \mathbb{L} \\
A(Cons([_x, l)) &= A(l) && \text{if }]_x \in this(l) \\
A(Cons([_x, l)) &= A(l) \cup \{x\} && \text{if }]_x \notin this(l)
\end{aligned}
$$

Acquisition Histories for Scheduling Trees

- Assumption: Acquisition and release only on base rules
- Compute set of final acquisitions

$$
\begin{aligned}
A(Nil) &= \emptyset \\
A(Spawn(a, l_1, l_2)) &= A(l_1) \cup A(l_2) \\
A(Cons(a, l)) &= A(l) && \text{if } a \in \mathrm{Act}_{nl} \text{ or } a =]_x \text{ for } x \in \mathbb{L} \\
A(Cons([_x, l)) &= A(l) && \text{if }]_x \in this(l) \\
A(Cons([_x, l)) &= A(l) \cup \{x\} && \text{if }]_x \notin this(l)
\end{aligned}
$$

- Check consistency of final acquisitions

$$
fac(Nil) = true \quad fac(Cons(a, l)) = fac(l) \quad fac(Spawn(a, l_1, l_2)) = fac(l_1
$$

Acquisition Histories for Scheduling Trees

- Assumption: Acquisition and release only on base rules
- Compute set of final acquisitions

$$
\begin{aligned}
A(Nil) &= \emptyset \\
A(Spawn(a, l_1, l_2)) &= A(l_1) \cup A(l_2) \\
A(Cons(a, l)) &= A(l) && \text{if } a \in \mathrm{Act}_{nl} \text{ or } a =]_x \text{ for } x \in \mathbb{L} \\
A(Cons([_x, l)) &= A(l) && \text{if }]_x \in this(l) \\
A(Cons([_x, l)) &= A(l) \cup \{x\} && \text{if }]_x \notin this(l)
\end{aligned}
$$

- Check consistency of final acquisitions

$$
fac(Nil) = true \quad fac(Cons(a, l)) = fac(l) \quad fac(Spawn(a, l_1, l_2)) = fac(l_1
$$

- Compute acquisition graph

$$
\begin{aligned}
G(Nil) &= \emptyset \\
G(Spawn(a, l_1, l_2)) &= G(l_1) \cup G(l_2) \\
G(Cons(a, l)) &= G(l) && \text{if } a \in \mathrm{Act}_{nl} \text{ or } a =]_x \text{ for } x \in \mathbb{I} \\
G(Cons([_x, l)) &= G(l) && \text{if }]_x \in this(l) \\
G(Cons([_x, l)) &= G(l) \cup \{x\} \times \mathrm{acq}(l) && \text{if }]_x \notin this(l)
\end{aligned}
$$

where $\mathrm{acq}(l) := \{x \mid [_x \in l\}$

Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl \rangle(Nil) \in ST$ and labeling sequence $l \in \mathrm{Act}^*$, we have

$$\exists t'.(\langle bl \rangle(Nil), \emptyset) \xrightarrow{l}{}^{*} (t', L) \wedge t' \in ST_{term} \iff \mathrm{acyclic}(G(bl)) \wedge fac(bl)$$

Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl \rangle(Nil) \in ST$ and labeling sequence $l \in \mathrm{Act}^*$, we have

$$\exists t'.(\langle bl \rangle(Nil), \emptyset) \xrightarrow{l}{}^{*} (t', L) \land t' \in ST_{term} \iff \mathrm{acyclic}(G(bl)) \land fac(bl)$$

- Proof Ideas:

Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl\rangle(Nil) \in ST$ and labeling sequence $l \in \mathrm{Act}^*$, we have

$$\exists t'.(\langle bl\rangle(Nil), \emptyset) \xrightarrow{l}{}^{*} (t', L) \wedge t' \in ST_{term} \iff \mathrm{acyclic}(G(bl)) \wedge fac(bl)$$

- Proof Ideas:
  - $\Longrightarrow$

Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl \rangle(Nil) \in ST$ and labeling sequence $l \in \mathrm{Act}^*$, we have

$$\exists t'.(\langle bl \rangle(Nil), \emptyset) \xrightarrow{l}{}^* (t', L) \wedge t' \in ST_{term} \iff \mathrm{acyclic}(G(bl)) \wedge fac(bl)$$

- Proof Ideas:
  - $\Longrightarrow$
    - $G(t)$ expresses constraints due to locking, that any schedule has to follow

Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl \rangle (Nil) \in ST$ and labeling sequence $l \in \mathrm{Act}^*$, we have

$$\exists t'.(\langle bl \rangle (Nil), \emptyset) \xrightarrow{l}{}^{*} (t', L) \land t' \in ST_{term} \iff \mathrm{acyclic}(G(bl)) \land fac(bl)$$

- Proof Ideas:
  - $\Longrightarrow$
    - $G(t)$ expresses constraints due to locking, that any schedule has to follow
    - Formally: Generalize to arbitrary initial set of locks and arbitrary scheduling trees, induction on scheduling tree.

Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl \rangle(Nil) \in ST$ and labeling sequence $l \in \mathrm{Act}^*$, we have

$$\exists t'.(\langle bl \rangle(Nil), \emptyset) \xrightarrow{l}^{*} (t', L) \wedge t' \in ST_{term} \iff \mathrm{acyclic}(G(bl)) \wedge fac(bl)$$

- Proof Ideas:
  - $\Longrightarrow$
    - $G(t)$ expresses constraints due to locking, that any schedule has to follow
    - Formally: Generalize to arbitrary initial set of locks and arbitrary scheduling trees, induction on scheduling tree.
  - $\Longleftarrow$

Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl \rangle(Nil) \in ST$ and labeling sequence $l \in \mathrm{Act}^*$, we have

$$\exists t'.(\langle bl \rangle(Nil), \emptyset) \xrightarrow{l}^{*} (t', L) \wedge t' \in ST_{term} \iff \mathrm{acyclic}(G(bl)) \wedge fac(bl)$$

- Proof Ideas:
  - $\Longrightarrow$
    - $G(t)$ expresses constraints due to locking, that any schedule has to follow
    - Formally: Generalize to arbitrary initial set of locks and arbitrary scheduling trees, induction on scheduling tree.
  - $\Longleftarrow$
    - Scheduling strategy: Schedule usages first. Final acquisitions in topological ordering of acquisition graph

Acquisition Graphs characterize Schedulability

- For scheduling tree $\langle bl\rangle(Nil) \in ST$ and labeling sequence $l \in \mathrm{Act}^*$, we have

$$\exists t'.(\langle bl\rangle(Nil), \emptyset) \xrightarrow{l}{}^{*} (t', L) \wedge t' \in ST_{term} \iff \mathrm{acyclic}(G(bl)) \wedge fac(bl)$$

- Proof Ideas:
  - $\Longrightarrow$
    - $G(t)$ expresses constraints due to locking, that any schedule has to follow
    - Formally: Generalize to arbitrary initial set of locks and arbitrary scheduling trees, induction on scheduling tree.
  - $\Longleftarrow$
    - Scheduling strategy: Schedule usages first. Final acquisitions in topological ordering of acquisition graph
    - Formally: Generalize to initial set of locks disjoint from locks that occur in scheduling tree. Generalize to arbitrary scheduling tree. Induction on scheduling tree.

Set of schedulable execution trees is regular

- Schedulable scheduling trees are regular (compute acquisition graphs by tree automata)

Set of schedulable execution trees is regular

- Schedulable scheduling trees are regular (compute acquisition graphs by tree automata)
- st^{-1} preserves regularity: Just another tree transducer construction

Set of schedulable execution trees is regular

- Schedulable scheduling trees are regular (compute acquisition graphs by tree automata)
- st^{-1} preserves regularity: Just another tree transducer construction
- Thus, we can decide lock-sensitive reachability of a regular set of configurations of a DPN.

Remark on complexity

- The lock-sensitive reachability problem is in NP:

Remark on complexity

- The lock-sensitive reachability problem is in NP:
  - For a sequential run, only polynomially many acquisition graphs/final acquisition sets occur

Remark on complexity

- The lock-sensitive reachability problem is in NP:
  - For a sequential run, only polynomially many acquisition graphs/final acquisition sets occur
  - So, for 2-PDS, we can guess these in advance

Remark on complexity

- The lock-sensitive reachability problem is in NP:
  - For a sequential run, only polynomially many acquisition graphs/final acquisition sets occur
  - So, for 2-PDS, we can guess these in advance
- For DPN: There may be exponentially many acquisition graphs!

Remark on complexity

- The lock-sensitive reachability problem is in NP:
  - For a sequential run, only polynomially many acquisition graphs/final acquisition sets occur
  - So, for 2-PDS, we can guess these in advance
- For DPN: There may be exponentially many acquisition graphs!
  - However, not for schedulable runs

Remark on complexity

- The lock-sensitive reachability problem is in NP:
  - For a sequential run, only polynomially many acquisition graphs/final acquisition sets occur
  - So, for 2-PDS, we can guess these in advance
- For DPN: There may be exponentially many acquisition graphs!
  - However, not for schedulable runs
  - Problem remaining: There may be exponentially many sets of used locks

Remark on complexity

- The lock-sensitive reachability problem is in NP:
  - For a sequential run, only polynomially many acquisition graphs/final acquisition sets occur
  - So, for 2-PDS, we can guess these in advance
- For DPN: There may be exponentially many acquisition graphs!
  - However, not for schedulable runs
  - Problem remaining: There may be exponentially many sets of used locks
  - Solution: Only check that certain locks are not used

Remark on complexity

- The lock-sensitive reachability problem is in NP:
  - For a sequential run, only polynomially many acquisition graphs/final acquisition sets occur
  - So, for 2-PDS, we can guess these in advance
- For DPN: There may be exponentially many acquisition graphs!
  - However, not for schedulable runs
  - Problem remaining: There may be exponentially many sets of used locks
  - Solution: Only check that certain locks are not used
    - Set of used locks only required at final acquisition.

Remark on complexity

- The lock-sensitive reachability problem is in NP:
  - For a sequential run, only polynomially many acquisition graphs/final acquisition sets occur
  - So, for 2-PDS, we can guess these in advance
- For DPN: There may be exponentially many acquisition graphs!
  - However, not for schedulable runs
  - Problem remaining: There may be exponentially many sets of used locks
  - Solution: Only check that certain locks are not used
    - Set of used locks only required at final acquisition.
    - Just check that less locks are used afterwards

Remark on complexity

- The lock-sensitive reachability problem is in NP:
  - For a sequential run, only polynomially many acquisition graphs/final acquisition sets occur
  - So, for 2-PDS, we can guess these in advance
- For DPN: There may be exponentially many acquisition graphs!
  - However, not for schedulable runs
  - Problem remaining: There may be exponentially many sets of used locks
  - Solution: Only check that certain locks are not used
    - Set of used locks only required at final acquisition.
    - Just check that less locks are used afterwards
    - Accepts executions with the guess acquisition graph, or with smaller ones

Main Theorem

Lock-sensitive reachability of a regular set of configurations is NP-complete for DPNs

Complexity of related problems

	DPN	PPDS	2PDS	DFN	PFSM	*n*FSM
$\mathrm{EF}(p_1 \parallel p_2)$	$\mathrm{NP}^{*?}$	$\mathrm{NP}^{\dagger?}$	$\underline{\mathrm{NP}}^{\dagger?}$	$\underline{\mathrm{NP}}^{*!}$	P	P
$\mathrm{EF}(A)$	NP	NP	$\mathrm{NP}^{\dagger?}$	NP	$\underline{\mathrm{NP}}$	P
$\mathrm{EF}(p_1 \parallel p_2 \wedge \mathrm{EF}(p_3 \parallel p_4))$	NP	NP	$\underline{\mathrm{NP}}$	$\mathrm{NP}^{*!}$	P	P
$\mathrm{EF}(A_1 \wedge \mathrm{EF}(A_2))$	NP	NP	NP	NP	NP	P
$\mathrm{EF}^{\setminus \text{neg}}$ (fixed #ops)	NP	NP	NP	NP	NP	P
EF (fixed #ops)	$\underline{\geq}$ $\underline{\mathrm{PSPACE}}^{\ddagger}$			$\underline{\geq}$NP		P
$\mathrm{EF}^{\setminus \text{neg}}$	$\underline{\geq}$ $\underline{\mathrm{PSPACE}}^{\ddagger \text{reg?}}$				$\underline{\geq}$ $\underline{\mathrm{NP}}^{\ddagger}$	P
EF	$\underline{\geq}$ $\underline{\mathrm{PSPACE}}^{\ddagger}$					P

- $*$ Requires spawn inside lock
  - $*!$ Polynomial algorithm if no spawn inside lock
  - $*?$ Complexity unknown if no spawn inside lock
- $\dagger?$ Hardness proof requires deadlocks/escapable locks. Complexity without this unknown.
- $\ddagger$ Hardness result requires no locks
- reg? Hardness requires regular APs. Complexity for double-indexed APs unknown ($\geq$NP)

The End

Thank you for listening